Hardwin Jungclaussen

Kausale Informatik

Einführung in die Lehre vom aktiven sprachlichen Modellieren
durch Mensch und Computer

Dieses Buch widme ich dem Andenken an meine Freunde,
die Informatiker

Klaus Fritzsch
und
Eberhard Klett.

Geleitwort

Das vorliegende Buch “Kausale Informatik” stellt die Summa einer jahrzehntelangen Befassung des Autors mit der Informatik, insbesondere aus systemtheoretischer, aber auch aus physikalischer und philosophischer Sicht dar und “will den Weg zu einer kausalen Wissenschaft Informatik (in Analogie zur Physik) ebnen helfen”. Mit diesem Anspruch wendet es sich an einen breiten Leserkreis: Insider, Informatikinteressierte und Anwender, aber auch an Informatikstudenten. “Kausale Informatik” ist kein Textbuch oder Kompendium; der mathematische Apparat ist bewusst zurückgenommen eingesetzt, um den Ideengehalt, dargestellt auf der Basis meines Erachtens kreativer, aber gewöhnungsbedürftiger Begriffsbildungen, mehr hervortreten zu lassen.

Aus dieser Sicht werden die Hauptbereiche der Informatik (einschließlich Hardware) behandelt, ein gesonderter Abschnitt ist dabei der Künstlichen Intelligenz gewidmet. Den Abschluss bildet die Auseinandersetzung des Autors mit philosophischen und ethischen Problemen der Informatik und ihrer Anwendungsbereiche.

“Besinnungsbücher”, wie das vorliegende, sind für die in rasanter und inhomogener Entwicklung befindliche Wissenschaftsdisziplin Informatik sehr notwendig und von großem Nutzen. “Kausale Informatik” liefert neben einer Gesamtdarstellung der Informatik in diesem Sinne viele Denkanstöße, die zu Neuem führen können, und Ansatzpunkte für durchaus konträre Diskussionen, fordert auch den Widerspruch heraus. Das muss ein Buch dieser Art.

Erwin P. Stoschek

Vorwort

Was hast du aber, das du nicht empfangen hast.
1 Kor 4,7

Ich glaube, ich habe nie eine Gedankenbewegung erfunden, sondern sie wurde mir immer von jemand anderem gegeben. Ich habe sie nur sogleich leidenschaftlich zu meinem Klärungswerk aufgegriffen.
Ludwig Wittgenstein

Wittgensteins Satz trifft sicher auf jeden zu, der sich einem “Klärungswerk” verschreibt, sei es, weil er andere aufklären will, sei es, weil er, wie ich, zunächst einmal selber zur Klarheit kommen will. Darin, dass ich Wittgensteins Satz auf mich beziehe, liegt keine Anmaßung. Ich weiß, dass sich meine Überlegungen und Einsichten nicht an der Schärfe und Fundamentalität seiner Gedanken und Sätze messen können. Zudem muss ich in dem Zitat die Worte “sogleich leidenschaftlich” ersetzen durch “soweit ich es mit zähem Bemühen vermochte”.

Antrieb meiner Bemühungen waren die Frage “Was ist Informatik?” und der Wunsch, ein in sich geschlossenes Gedanken- und Begriffsgebäude der Informatik zu erarbeiten. Ich habe den verschlungenen Weg zum Ziel nachträglich begradigt, um ihn für andere gangbar zu machen. Dabei ist ein Buch entstanden, das sich nicht so sehr durch markante, klärende Sätze, schon gar nicht durch mathematische Formulierungen auszeichnet, sondern eher durch wortreiche Erklärungen.

Wenn ich das Ergebnis meiner Arbeit nun der Öffentlichkeit vorlege, wende ich mich in erster Linie an junge, vorwärtsstrebende Menschen, die sich anschicken, als Informatiker die technische Basis der sich herausbildenden Informationsgesellschaft mitzugestalten und die das erlernte und zu erlernende Wissen in einen größeren Zusammenhang stellen möchten. Je tiefer diese jungen Informatiker um die naturwissenschaftlichen und auch um die philosophischen Wurzeln ihres Faches wissen, umso sicherer werden sie die Entwicklung in eine Richtung lenken, die zu einer funktionierenden und menschenwürdigen Gesellschaft führt. Darüber hinaus hoffe ich, dass auch erfahrene Informatiker, die sich für das wissenschaftliche und philosophische Umfeld, insbesondere für die physikalischen Grundlagen ihres Arbeitsgebietes interessiern, diese oder jene Anregung erhalten. Schließlich wendet sich das Buch an Leser, die sich einen Überblick über das Gesamtgebiet oder tiefere Einblicke in verborgene Zusammenhänge verschaffen wollen und die verstehen möchten, was sich hinter der Informatik und insbesondere hinter der künstlichen Intelligenz verbirgt, ohne Informatik studieren zu müssen. An diese Leser habe ich vor allem beim Schreiben der Kapitel 1 bis 7 und 11 sowie des Teils 3 und des Kapitels 21.4 gedacht. Das Buch setzt keine mathematischen Kenntnisse, wohl aber Lust zum Denken voraus.

Ich habe die Hoffnung, dass das Buch zu einem breiteren Verständnis dessen, was Informatik ist, und gleichzeitig zur Klärung des Inhalts der Wissenschaft Informatik beiträgt, selbst wenn nicht alle Steine des Anstoßes - wie Unschärfe in den Formulierungen und Schlussfolgerungen oder mangelnde Klarheit der Darlegungen - aus dem Wege geräumt sind, den das Buch durch das Dickicht von tausend Fragen und tausend Ideen dorthin bahnt, wo sich ein freierer Blick über das weite Feld der Informatik bietet. Natürlich sind auch andere Wege zu anderen "Standpunkten" möglich, die andere "Ansichten" der Informatik bieten.

Ich bedanke mich bei all denen, die meine Arbeit an dem Buch gefördert haben, sei es durch menschliche oder fachliche Unterstützung, durch Motivation, durch Diskussion oder durch Kritik. Mein erster Dank gilt meiner Frau. Sie hat mir Ruhe zu ungestörtem Arbeiten in der heimischen Studierstube geschenkt. Mein zweiter Dank gilt Prof. Hans Heinold, Prof. Dietrich Schubert und Dr. med. Klaus Weinert für die mir zuteil gewordene Unterstützung, die ich brauchte, um trotz aller Forderungen und auch Überforderungen, mit denen ich in meiner beruflichen Umwelt vor 1989 konfrontiert war, unbeirrt an meinem Anliegen arbeiten zu können und die Grundlage zu diesem Buche zu legen.

Für das kritische Lesen des gesamten Manuskripts und für viele wichtige Hinweise sage ich Dr. Manfred Bonitz, Dr. Horst Piehler, Prof. Dietrich Schubert und Prof. Erwin Stoschek meinen tiefen Dank.

Für die Durchsicht von Teilen des Manuskripts und wertvolle Hinweise bedanke ich mich bei Prof. Gerhard Banse, Bernd Dupal, Prof. Peter Fleissner, Prof. Hartmut Fritzsche, Dr. Dietbert Gütter, Dr. Claude-Joachim Hamann, Prof. Rolf Hebenstreit, Prof. Gerhard Hertz†, Prof. Dieter Jungmann, Prof. Alexander Leitsch, Dr. Wolfgang Oertel, Prof. Jörg Pflüger, Dr. Michael Posegga, Dr. Heinz Rötger, Dr. Wolfgang Schwarz, Prof. Rainer G. Spallek und Prof. Helmut Thiele.

Aus der großen Zahl derer, denen ich für Diskussionen oder für Unterstützung in dieser oder jener Form dankbar bin, möchte ich einige nennen: Prof. Gerhard Diener, Prof. Volkmar Dienhold, Prof. Klaus Fritzsch†, Dr. Eberhard Klett†, Dr. Anatolij Korneitschuk, Prof. Klaus Meißner, Dr. Wolfgang Oertel, Prof Hans-Georg Schöpf, Dr. Wolfgang Schubert und Prof. Paul Ziesche.

Mein besonderer Dank gilt Reinhart Schmidt für die ständige, selbstlose Unterstützung bei der Arbeit mit der Technik und bei der Erstellung des Schriftsatzes und der Bilder, sowie Bernd Dupal und Dr. Wolfgang Oertel für die Erstellung der Programme.

Alle, die zu nennen ich versäumt habe, bitte ich um Entschuldigung; sie mögen meiner Dankbarkeit gewiss sein. Ich habe nicht alle Bemerkungen und Einwände berücksichtigt. Für die Richtigkeit der Aussagen des Buches trage ich die alleinige Verantwortung. Ich werde jedem dankbar sein, der mich auf Fehler hinweist.

Schließlich möchte ich mich beim Deutschen Universitäts-Verlag und bei Dr. Reinald Klockenbusch bedanken, der in einem Gespräch betreffs des Buches im

Jahre 1991 die Grundrichtung mit sinngemäß folgenden Worten vorgezeichnet hat: Es ist das Allgemeine mit dem Detail in ausgewogenem Verhältnis zu verbinden. Ich bedanke mich für die mir entgegengebrachte Nachsicht und Geduld.

Ich bin dankbar für die Freude, die mir die Arbeit an dem Buch bereitet hat.

Dresden, August 2000 Hardwin Jungclaussen

Inhaltsverzeichnis

Geleitwort VII

Vorwort IX

Inhaltsverzeichnis XIII

Symbole und Abkürzungen XIX

Einleitung 1
Anliegen und Besonderheiten des Buches . . . 1
Hinweise zum Lesen des Buches . . . 8

Teil 1 Grundbegriffe und Grundideen

Zusammenfassung der Kapitel 1 bis 3 11

1 Codierung, Evolution und Information 13

2 Gedanke und Sprache 19

3 Informatik - Lehre vom aktiven sprachlichen Modellieren 25
3.1 Modellklassen. Begriffsbestimmung der Informatik . . . 25
3.2 Fundamente der Informatik . . . 28

4 Analoges und digitales Modellieren 31
Zusammenfassung . . . 31
4.1 Messen und reelle Zahlen . . . 32
4.2 Analoges Rechnen . . . 35

5 Syntax und Semantik 43
Zusammenfassung . . . 43
5.1 Sprache und Evolution . . . 44
5.2 Hierarchische Komponierung . . . 47
5.3. Syntaxregeln . . . 48
5.4 Externe, interne und formale Semantik . . . 51
5.5 Begriffsbildung . . . 55
5.6* Umcodierungseffizienz und syntaktische Information . . . 61

6 Arbitrarität und Zirkularität 69
Zusammenfassung . . . 69
6.1 Arbitrarität des Artikulierens . . . 69
6.2 Referenzielle Zirkularität . . . 70
6.3 Operationale Zirkularität . . . 72

7 Evolution der Intelligenz 75
Zusammenfassung . . . 75

7.1 Deduktive, assoziative und intuitive Intelligenz. 76
7.2 Beschriftung der tabula rasa. Algorithmenbegriff 81

8 Formale Grundbegriffe und Methoden 87
Zusammenfassung 87
8.1 Uniforme Systembeschreibung (USB) 88
8.2 Kausaldiskrete Prozessbeschreibung 95
8.2.1 Reale und sprachliche Operatoren 95
8.2.2 Petrinetze 101
8.2.3 Abstrakter Automat 105
8.2.4 Elementare kausaldiskrete Operatoren 108
8.2.5 Logisch-kausale Äquivalenz 111
8.3 Operation, Funktion, Prädikat, Berechenbarkeit 114
8.4* Universelle algorithmische Systeme 125
8.4.1 Problemstellung 125
8.4.2 Turingmaschine 127
8.4.3 Unbeschränkte Registermaschine (URM) 129
8.4.4 Markovalgorithmus 130
8.4.5 Rekursive Funktionen und USB-Funktionen 131
8.4.6 Einschub: Rekursives Berechnen 142
8.4.7 Lambda-Kalkül 146
8.5* CHURCH'sche These und Kalkültransformation 151
8.6 Vier Grundideen des elektronischen Rechnens 158

Teil 2 Vom Bit zur Maschinensprache

9 Grundlagen der Komponierung informationeller Operatoren 167
Zusammenfassung 167
9.1 Statische und dynamische Codierung 168
9.2 Elementare informationelle Operatoren 170
9.2.1 Elementare boolesche Operatoren und die erste Grundidee des elektronischen Rechnens 170
9.2.2 Künstliche Neuronen 176
9.3* Ergänzung der formalen Methoden: Boolesche Algebra 178
9.4 Netzklassen 188
9.5 Ein-Bit-Speicher 197
9.6 Einschub: Magnetische und optische Speicherverfahren 199

10 Realisierungsprinzip zirkelfreier boolescher Netze 201
Zusammenfassung der Kapitel 10 bis 12 201
10.1 Die zweite Grundidee des elektronischen Rechnens 203
10.2 Zirkelfreie Schalternetze 205

11 Historischer Einschub: Entwicklung der begrifflichen und physikalischen Basis der Rechentechnik 207
11.1 Von der Syllogistik des Aristoteles zur Schaltalgebra 207
11.2 Rechnergenerationen 213

12 Technische zirkelfreie boolesche Netze 219
12.1 Codeumsetzer und elektronischer Festwertspeicher 219
12.2 Herstellung von Diodenmatrizen 225
12.3 Anwendungen von Diodenmatrizen 227
12.3.1 Elementare Hardware-Software-Schnittstelle 227
12.3.2 Multiplexer, Demultiplexer und Bus 228
12.3.3 Funktionsgeneratoren 234
12.3.4 Steuermatrix 234

13 Von der Kombinationsschaltung zum Von-Neumann-Rechner 237
Zusammenfassung 237
13.1 Die dritte Grundidee des elektronischen Rechnens 238
13.2 Elektronische Speicher 243
13.2.1 Register 243
13.2.2 Adressierbarer elektronischer Speicher 244
13.3* Einschub: Berechenbarkeits-Äquivalenzsatz 246
13.4 Taschenrechner 250
13.5 Prozessor 253
13.5.1 Idee des Prozessors und seiner Programmierung 253
13.5.2 Maschinensprache 257
13.5.3 Arbeitsweise der RALU 261
13.5.4 Sprungbefehl 266
13.5.5 Matrixsteuerwerk 269
13.5.6 Operatorenkomponierung mittels Firmware 270
13.5.7 Mikroprozessor 271
13.6 Von-Neumann-Rechner 273
13.7 Netzparadigma und imperatives Paradigma 274

Teil 3 Von der Maschinensprache zur künstlichen Intelligenz

14 Zwischenbilanz 283
Zusammenfassung 283
14.1 Probleme des Softwareweges zur KI 284
14.2 Drei Rollen des Computers als eines intelligenten Partners des Menschen 286

15 Lösen mathematischer Probleme 289
Zusammenfassung 289
15.1 Funktionales Programmieren 291

15.2 Imperatives Programmieren . . . 294
15.3 Näherungsverfahren . . . 299
15.4 Assembler, Binder, Lader . . . 300
15.5 Semantische Lücke . . . 303
15.6 Numerisches Rechnen . . . 305
15.7 Programmierung von Operatorenhierarchien . . . 309
15.8 Analytisches Rechnen . . . 312
15.9 Bemerkung zur Programmübersetzung . . . 320

16 Lösen nichtmathematischer Probleme 323
Zusammenfassung . . . 323
16.1 Schlussfolgern . . . 324
16.2 Wissensverarbeitung . . . 335
16.3 Erfinden . . . 345
16.4 Übersetzen und die vierte Grundidee des elektronischen Rechnens 353
16.5* Theorie formaler Sprachen . . . 364

17 Unterhaltung 373
Zusammenfassung . . . 373
17.1 Der Computer als Gesprächspartner . . . 374
17.2 Der Computer als Unterhalter . . . 381
17.3 Der Computer als Schachpartner . . . 385

18 Evolution der Programmiersprachen 399
Zusammenfassung . . . 399
18.1 Der Flaschenhals des linearsprachlichen Modellierens . . . 400
18.2 Semantische Verdichtung . . . 407
18.3 Technisches Semantikproblem und Programmierparadigmen . . . 416

Teil 4 Vertiefende Ergänzungen

19 Ergänzungen zum Problemlösungsweg der Rechentechnik 429
Zusammenfassung . . . 429
19.1 Vorbemerkung . . . 430
19.2 Hardwarearchitektur unterhalb der Prozessorebene . . . 431
19.2.1 Befehlssatz und Speicherhierarchie . . . 431
19.2.2 Pipelining und Vektorrechner. . . . 437
19.2.3 ALU-Array. Simulation von Nahwirkungen . . . 440
19.3 Hardwarearchitektur oberhalb der Prozessorebene . . . 444
19.4 Architektur kausaldiskreter Systeme . . . 452
19.5 Betriebssystem . . . 461
19.5.1 Anwendungsprogramme und Organisationsprogramme . . . 461
19.5.2 Prozessbegriff und geteilte Nutzung von Hardwareoperatoren . 463
19.5.3 Geteilte Nutzung von Speicherplätzen, Daten und Programmen 470

19.5.4 Verteilte Systeme 472
19.5.5 Systemrufe und geschützte Prozesskomponierung 480

20 Programmbeispiele 485
Zusammenfassung 485
20.1 Imperative Programme 486
20.2 CommonLisp-Programme 488
20.2.1 Funktionale Programme 488
20.2.2 Imperatives Programm 491
20.2.3 Objektorientiertes Programm 492
20.2.4 Logisches Programm 494
20.3 Objektorientiertes Programm in Borland-Pascal 499
20.4 Netzprogrammierung und Software-Lebenszyklus 515

21 Komplexität 519
Zusammenfassung der Kapitel 21 und 22 519
21.1 Zum Begriff der Komplexität 520
21.2* Berechnungskomplexität 524
21.3 Modellierung komplexer Systeme und Prozesse 528
21.3.1 Strukturelle und nichtlineare Komplexität 528
21.3.2* Fuzzy-Kalkül 534
21.4 Offene Fragen und die Komplexität des Denkens 545

22 Resümee und Perspektiven 557

Schlusswort 563
Zur gesellschaftlichen Bedeutung der Informatik 563

Glossar 577

Literatur 597

Namen- und Sachverzeichnis 607

Symbole und Abkürzungen

Symbole

- ^ (Dach) 1. (in Formeln): Potenzoperator; 2. (im Glossar vor einem Begriff): zu lesen als “siehe unter...”
- Symbole der booleschen Algebra siehe Bild 9.1.
- Abkürzungen und Symbole der USB-Methode siehe die Bilder 8.1 und 8.2.
- Abkürzungen für die Bausteine von Prozessoren und Computern siehe die Bilder 13.7. und 19.1.
- Maßeinheiten siehe Bild 19.1

Abkürzungen

AC	Akkumulator
ADR-Matrix	Adressiermatrix
ALU	arithmetisch-logische Einheit
aop	Arbeitsoperator
BNF	Backus-Naur-Form
BR	Befehlsregister
BZ	Befehlszähler
CD	Compact Disk
CD-RW	ReWritable (wiederholt beschreibbare) CD
Computer-IV	Informationsverarbeitung durch den Computer
CPU	Central Processing Unit
DIN	Deutsches Institut für Normung
DNF	disjunktive Normalform
DR	Datenregister
DRAM	Dynamischer RAM
DVD	Digital Versatile Disk
E/A	Eingabe/Ausgabe
EPROM	wiederholt programmierbarer ROM
FIFO	First In First Out
HK	Halbkommutator
HS	Hauptspeicher
Human-IV	Informationsverarbeitung durch den Menschen
i.d.R.	in der Regel
i.e.S.	im engen Sinne
INC	Inkrementierer
IV	Informationsverarbeitung
IV-System	Information verarbeitendes System, informationelles System
i.w.S.	im weiten Sinne
K	1.Kombinationsschaltung; 2. Kommutator

KDNF	kanonische disjunktive Normalform
KKNF	kanonische konjunktive Normalform
KR-Netz	Netz aus Kombinationsschaltungen und Registern
ld	Duallogarithmus (Logarithmus zur Basis 2)
LSI	Large Scale Integration
LIFO	Last In First Out
MByte	Megabyte, $2^{20} \approx 10^6$ Byte
OC	Operationscode
od	Operand
on	Operation
ON	Operatorennetz
op	Operator
PAL	Programmable Array Logic
PAP	Programmablaufplan
PLA	Programmable Logic Array
PROM	programmierbarer ROM
PS-Netz	Netz aus Prozessoren und Speichern
R	Register (in Kap.10 Widerstand)
RALU	ALU mit ihren Arbeitsregistern
RAM	Random Access Memory, Schreib-Lese-Speicher
ROM	Read Only Memory, Lesespeicher
S	Sammelweiche (in Kap.10 Schalter)
sop	Steueroperator
SPEI-Matrix	Speichermatrix
TNS	Transport Nach dem Speicher (vom AC)
TVS	Transport Vom Speicher (zum AC)
USB	Uniforme Systembeschreibung
VLSI	Very Large Scale Integration
Z	Zweigeweiche

Einleitung

Anliegen und Besonderheiten des Buches

Seid fruchtbar und mehret euch und füllet die Erde und machet sie euch untertan.

1.Mose 1, 28

Denn das wissenschaftliche Denken verlangt nun einmal nach Kausalität, insofern ist wissenschaftliches Denken gleichbedeutend mit kausalem Denken, und das letzte Ziel einer jeden Wissenschaft besteht in der vollständigen Durchführung der kausalen Betrachtungsweise.

MAX PLANCK[1]

Die Gesichter und Hände der Mitmenschen, ein Stück Himmel und vielleicht noch eine Zimmerpflanze oder ein Blumenstrauß - das ist so ziemlich das einzige, was ein Großstädter an Natürlichem in seiner Umwelt zu sehen bekommt. Aus der natürlichen Umwelt ist eine künstliche geworden. Das bedeutet nicht, dass die heutigen Menschen die Welt besser beherrschen und die Zukunft besser gestalten als die Menschen vor 1000 oder 10000 Jahren, denn sie können die Wirkungen ihrer eigenen Ideen und Errungenschaften nicht voraussehen. So wie man die Auswirkungen der Dampfmaschine auf die menschliche Gesellschaft nicht voraussehen konnte, so können wir Heutigen die Auswirkungen des Computers nicht voraussehen.

Die Hilflosigkeit vor der eigenen technischen Machtfülle kennzeichnet die gegenwärtige Situation der Menschheit. Sie ist die Fortsetzung der Hilflosigkeit des primitiven Menschen der Natur gegenüber, und in ihrer Hilflosigkeit brauchen die Menschen von heute genauso wie ihre Vorfahren einen Hoffnungsträger, einen Glauben, einen Gott, eine Religion. Das gilt sicher nicht für jeden einzelnen Zeitgenossen, wohl aber für die Menschheit insgesamt, die sich bedroht und ohnmächtig fühlt. Denn die Beherrschung der Welt liegt immer jenseits des Horizontes, dem die Menschheit seit Adam zustrebt gemäß dem eingangs zitierten biblischen Befehl. Es ist jedoch nicht der Befehl einer höheren Instanz, sondern ein innerer Befehl, es ist das Produkt der genetischen und der Motor der kulturellen Evolution.

Im Spannungsfeld zwischen Technik und Psyche läuft eine atemberaubende Entwicklung ab, in der die Informatik ständig an Bedeutung gewinnt. Die Welt wird

---

1 Zitat aus dem Vortrag “Kausalgesetz und Willensfreiheit”, abgedruckt in [Planck 90]

immer künstlicher, und das betrifft in zunehmendem Maße auch die sprachliche Kommunikation. Immer mehr Informationen, die auf den Menschen einstürmen, werden von Maschinen "hergestellt", und zwar werden sie nicht nur maschinell codiert, sondern auch artikuliert, d.h. gedruckt oder gesprochen, sodass der Eindruck entstehen kann, nicht nur die Zeichen, sondern auch ihre Bedeutung, die der Mensch in die Zeichen hineinlegt, seien vom Computer produziert.

Der Computer, das Produkt menschlicher Erfindungsgabe, verändert die Welt und verändert unser Weltbild, ohne dass wir wissen, wohin die Reise geht und ohne dass wir die Entwicklung nach eigenem Gutdünken, nach eigenem Wissen und Gewissen steuern könnten. Die Folge ist ein zunehmender Druck in Richtung einer organisatorischen und sittlichen Erneuerung der Gesellschaft und speziell in Richtung einer notwendigen Anpassung an das eigene Kind, den Computer, und umgekehrt der Computertechnik an den Menschen, womit den Informatikern eine besondere Verantwortung auferlegt ist.

Die Informatik befindet sich ihrer Jugend gemäß gegenwärtig noch in der expansiven Phase des Erkundens, Erfindens und Sammelns. Die Entwicklung schreitet divergent in zahllosen Richtungen voran, wobei es nicht ausbleiben kann, dass die rechte Hand oft nicht weiß, was die linke tut. Der Evolutionsdruck in Richtung nachträglicher Konvergenz der divergierenden Zweige kommt erst in Ansätzen zur Wirkung, und der Übergang von der differenzierenden zur generalisierenden Phase, den jede Wissenschaft im Großen ebenso wie jede Abstraktion im Kleinen durchläuft, ist Aufgabe der Zukunft. Es ist also noch ein weiter Weg zurückzulegen, ehe die Informatik die Geschlossenheit beispielsweise der Physik erreicht haben und zu einer fundamentalen Wissenschaft geworden sein wird. Im Gegensatz zu manchen meiner Fachkollegen glaube ich, dass am Ende des Weges eine "*exakte Natur-Wissenschaft* Informatik" stehen wird, wobei unter "Natur" alles zu verstehen ist, was uns umgibt und was wir wahrnehmen bzw. direkt oder indirekt beobachten können. Angesichts der immer durchgreifenderen Beeinflussung der Natur durch den Menschen erweitert sich der Gegenstand der "Naturwissenschaften" zunehmend vom Natürlichen zum Wahrnehmbaren oder Beobachtbaren, das Künstliche eingeschlossen. Man denke nur an die "Naturwissenschaft Chemie", die sich mit Stoffen beschäftigt, die in der unberührten Natur nicht vorkommen. Die *Naturwissenschaft* als Wissenschaft vom Naturgegebenben hat sich fast unbemerkt zur *Wissenschaft des Beobachtbaren* erweitert.

Den Weg der Informatik zu einer exakten "Naturwissenschaft" (in dem genannten erweiterten Sinne) ebnen zu helfen, ist ein Anliegen des Buches. Es bemüht sich um eine einheitliche Sicht auf die verschiedenen Teilgebiete der Informatik und will das "Verständnis im Überblick" mit dem "Verständnis des Details" verbinden. Die Einheitlichkeit der Sicht spiegelt sich in der Einheitlichkeit des begrifflichen Apparates wider, mit dessen Hilfe sich sowohl die Hardware (die gerätetechnische Basis) als auch die Software (der programmtechnische Überbau) informationeller Systeme, darüber hinaus aber auch die Strukturen von Fertigungssystemen und der Ablauf von

Fertigungsprozessen, beispielsweise der Automobilherstellung, in einheitlicher Weise beschreiben lassen.

Angesichts der thematischen Breite des Buches verbietet sich das Eingehen auf tiefere Details. Einige speziellere rechentechnische Probleme sind aus dem Gedankengang vom Bit zur künstlichen Intelligenz, der sich durch die Teile 2 und 3 zieht, herausgelöst und in Teil 4 zusammengefasst. Das weite Feld der Anwendung der Rechentechnik wird nur hier und da in Form von Beispielen berührt. Ausführliche Darstellungen der verschiedenen Gebiete der Informatik findet der Leser in der angegebenen Literatur. Die Literaturauswahl ist bewusst klein gehalten. Sie beschränkt sich im Wesentlichen auf deutschsprachige Monographien mit umfangreichen Literaturangaben sowie auf einige moderne Lehrbücher und Lexika.

Die Begriffe, in denen ich meine Gedanken zur Informatik in diesem Buch niedergelegt habe, und speziell die in Kap.8 eingeführte Methode der *uniformen Systembeschreibung* (USB) sind das Ergebnis einer kritischen Betrachtung der Phänomene der Informationsverarbeitung mit den Augen eines Naturwissenschaftlers, dessen Denken ursprünglich nicht durch die Informatik, sondern durch die Physik geprägt worden ist. Angestrebt wird ein System von Begriffen, das zunächst auf Sinnfälligkeit ausgerichtet ist, das aber eine Schärfung der Begriffe bis zu mathematischer Strenge erlaubt, die allerdings in dem Buch nicht erreicht wird. Das Ziel des Buches besteht also nicht darin, eine einheitliche *formale Theorie* der Informatik vorzulegen, sondern eher darin, die gedanklichen und begrifflichen Grundlagen für eine solche Theorie zu erarbeiten.

Der in Teil 1 eingeführte Begriffsapparat wird in den nachfolgenden Teilen auf die *Prozessorinformatik* angewendet, d.h. auf die Lehre von der traditionellen Informationsverarbeitung, die sich des klassischen Prozessors bedient. Die Anwendung auf die *technische Neuroinformatik*, d.h. auf die Lehre von der Informationsverarbeitung durch künstliche neuronale Netze, wird lediglich angedeutet. Das moderne Gebiet der *Quanteninformatik* wird nicht behandelt[2].

Die Bemühungen um einen einheitlichen Begriffsapparat führen hier und da zur Einführung unkonventioneller Begriffe, zur Verwendung unkonventioneller Bezeichnungen für gängige Begriffe oder zur Verwendung gängiger Wörter in unüblichem Sinne. Letzteres trifft z.B. für den Intelligenzbegriff zu (s.u.). Das wohl auffallendste Beispiel unkonventioneller Begriffe ist das Begriffspaar **Realem - Idem** (Betonung jeweils auf der letzten Silbe, wie bei Morphem). Grob gesagt bezeichnet ein Realem-Idem-Paar ein reales Objekt und seine Widerspiegelung im Bewusstsein.

Der Grund für die Einführung dieses Begriffspaares ist der Wunsch, den wohl schwierigsten Begriff, mit dem die Informatiker zu tun haben, den Begriff der

2 Eine kompakte Darstellung der physikalischen Grundlagen des Quantencomputers und des gegenwärtigen Entwicklungsstandes findet der Leser in [Briegel 99].

Bedeutung, wie er im täglichen Umgang verwendet wird, von vornherein aus allen technischen und logischen Gedankengängen auszuklammern und ihm eine Sonderstellung zuzuweisen. Das dient der Klarheit der Darstellung. Denn der Sinn des Wortes "Bedeutung" ist in seinem vollen umgangssprachlichen Gehalt kaum zu erfassen. Das Buch akzentuiert die Schwierigkeit mit der Feststellung, dass Bedeutung (im umgangssprachlichen Sinne des Wortes) an *Bewusstsein* gekoppelt ist. Anders wird man dem Wort "Bedeutung" nicht gerecht. Genau genommen ist "Bedeutung" nämlich etwas zutiefst Individuelles. Ein reales Objekt, sei es ein Gegenstand oder ein gesprochenes oder gedrucktes Wort, hat für jeden, der es wahrnimmt bzw. interpretiert, *seine eigene*, d.h. seine personenspezifische *Bedeutung*. Diese personenspezifische und niemals vollständig mitteilbare Bedeutung wird vom Begriff *Idem* erfasst.

Darüber, was Bedeutung und was Bewusstsein "wirklich" sind, fällt in dem Buch kein Wort. Es handelt sich um unmittelbare Erfahrungen, denen jeder Menschen "sprachlos" gegenübersteht, die keine sprachliche Erklärung zulassen. Auf sie trifft der wittgensteinsche Imperativ zu: *"Wovon man nicht sprechen kann, darüber muss man schweigen.*" An ihn halten wir uns. Dennoch muss der Begriff der *Bedeutung als persönlicher, in Worte nicht fassbarer Erfahrung* benannt werden, denn ohne ihn ist es nicht möglich über Sprache und Informatik nachzudenken und sich mitzuteilen.

Als Autor wusste ich mir keinen besseren Rat als die Erfindung eines neuen Wortes für diesen Begriff und eines zweiten Wortes für sein Pendant in der Realität, falls ein solches existiert. Nur so lassen sich Missverständnisse vermeiden, die allzu leicht dadurch entstehen, dass man meint, jeder verstünde unter einem bestimmten Wort genau dass Gleiche wie man selber. Die Gefahr, falsch verstanden zu werden ist besonders groß im Falle von Wörtern, die nicht einheitlich verwendet werden oder die bedeutungsmäßig überladen sind wie z.B die Wörter *Information* und *Semantik*.

Die Einführung neuer Wörter führt zuweilen zu einem unkonventionellen Sprachgebrauch, der für den Fachmann ohne Frage eine Unannehmlichkeit darstellt und wohlwollende Nachsicht und die Bereitschaft voraussetzt, ungewohnte Wörter in gewohnte zu übersetzen. Das Glossar am Ende des Buches soll das Übersetzen erleichtern.

Die Einführung des Begriffspaares *Realem - Idem* erfolgt gleich zu Beginn des Buches. Das Idem wird unter Verwendung des Wortes *Bewusstsein* eingeführt, also eines Begriffs, der außerhalb der konventionellen Informatik liegt und der für eine rein technische Darstellung nicht erforderlich ist. Denn es gibt keinen vernünftigen Grund, dem Computer Bewusstsein zuzubilligen. Dieser Sachverhalt hat zusammen mit der genannten Kopplung zwischen Bedeutung und Bewusstsein folgende Konsequenz: *Computer verstehen und verarbeiten* ***keine "Bedeutung"***. Diesen Schluss nenne ich das ***Prinzip der Bedeutungsfreiheit*** *der Informationsverabeitung durch den Computer*. Das Wort "Prinzip" soll die "prinzipielle" Wichtigkeit des Schlusses unterstreichen. Das Prinzip der Bedeutungsfreiheit hat in der klassischen Logik sein Pendant im sog. *Extensionalitätsprinzip*.

2

Die Herangehensweise des Buches an die Probleme der Informatik zeichnet sich durch eine weitere konzeptionelle Besonderheit aus. Sie besteht darin, dass die Stellung jeder Frage und die Suche nach Antworten stets vom materiellen *Träger* der informationellen Prozesse ausgeht. Das gilt insbesondere für die Frage*: Zu welchen Leistungen ist ein Information verarbeitendes System fähig?* Im Falle der Informationsverarbeitung durch den Computer, kurz der *Computer-IV*, ist der Träger das als Computer bezeichnete *technische Gerät*. Im Falle der Informationsverabeitung durch den Menschen, kurz der *Human-IV*, ist der Träger das *Gehirn*. Die genannte konzeptionelle Besonderheit nenne ich das **Trägerprinzip**. Nach diesem Prinzip sind die Probleme der Informatik vom Träger her und nicht von der Mathematik, von der Logik oder von irgend einem anderen abstrakten Denksystem her anzugehen. Da das Trägerprinzip grundlegend für die Herangehensweise des Buches ist, sollte der Leser schon jetzt klar erkennen, worin das Prinzip genau besteht und welche Konsequenzen es hat. In einem ähnlichen, aber auf die Computer-IV beschränkten Sinne wird zuweilen vom *Prozessorprinzip* gesprochen. 3

Es würde naheliegen, unter dem Träger technischer informationeller Prozesse den Computer als Einheit von Hardware (gerätetechnischer Basis) und Software (programmtechnischem Überbau) zu verstehen. Die obige Frage lautet dann: *Zu welchen Leistungen kann eine gegebene Software eine gegebene Hardware befähigen?* In konkreten Fällen kann diese Frage von großer Wichtigkeit sein. Es ist aber nicht diejenige Frage, der unser Interesse gilt. Unsere Frage lautet: *Zu welchen Leistungen kann eine gegebene Hardware "**im Prinzip**" befähigt werden?* Hier bedeutet "im Prinzip" soviel wie "durch beliebige Software" oder genauer "durch die Gesamtheit aller denkbaren Softwaresysteme", was im Grunde gleichbedeutend ist mit "ohne Berücksichtigung der Software".

Die Frage ist scheinbar sinnlos, denn ohne Software ist die Hardware zu nichts in der Lage, sie ist "tot". Dass die Frage trotzdem nicht sinnlos ist, zeigt folgender Vergleich. Wir ersetzen die Hardware durch ein Auto und die Software, die das Verhalten der Hardware *steuert*, durch einen Fahrer, der das Verhalten des Autos *steuert*. Obwohl ein Auto ohne Fahrer "tot" ist, hat die Frage Sinn, was ein Auto "im Prinzip", d.h. ohne Berücksichtigung der Fähigkeiten des Fahrers, zu leisten vermag. Man kann z.B. nach der Höchstgeschwindigkeit oder nach dem Kraftstoffverbrauch fragen. Genauso kann hinsichtlich der Rechnerhardware fragen, welche Funktionen sie "im Prinzip" berechnen kann.

Die Forderung des Trägerprinzips stellt keine Notwendigkeit dar. Das Nachdenken der theoretischen Informatiker geht in der Regel nicht vom Trägerprinzip, sondern vom "*Denkprinzip*" aus, d.h. davon, *wie und was* der Mensch denkt. Das führt zwangsläufig zu der Frage, ob ein Computer "dasselbe und ebenso *denkt*" wie der Mensch, und zu der speziellen Frage, ob er dasselbe (dieselben Funktionen) *berechnen* kann wie der Mensch. Das Trägerprinzip stellt diese Frage *nicht*, sondern fragt, welche Hardware wie und was "denken" bzw. berechnen kann. Wir werden die Frage hinsichtlich einer Hardware stellen, die wir in Kap.13 entwickeln (nacher-

finden) werden. Es handelt sich um den sog. *Prozessorrechner*. Um Aussagen über seine "prinzipielle" Leistungsfähigkeit machen zu können, benötigen wir eine Methode, nach der sich sämtliche denkbaren Steuervorschriften für die Hardware (sog. Maschinenprogramme) erzeugen ("komponieren") lassen. Eine solche Methode wird in Kap.8 entwickelt und USB-Methode genannt (USB von Uniforme System-Beschreibung). Auf diesem Wege werden wir zu der bekannten Aussage gelangen, dass der Computer "nur" sog. *rekursive Funktionen* berechnen kann.

In enger Beziehung zum Trägerprinzip steht das **Realisierbarkeitsprinzip** der Computer-IV. Es schränkt den Bereich, über den nachgedacht wird, auf das *effektiv Machbare*, das *praktisch Realisierbare* ein. Das betrifft nicht nur den Träger, z.B. den Computer, sondern auch das, was der Träger "macht" (denkt, berechnet) bzw. machen soll. Gemäß diesem Prinzip werden nur solche informationellen Systeme betrachtet, die mit *endlichem* Aufwand gebaut werden können, und nur solche Verarbeitungsprozesse, die in *endlicher* Zeit "terminieren", d.h. ihr Ende erreichen.

Die drei genannten Prinzipien,

- das Prinzip der Bedeutungsfreiheit,
- das Trägerprinzip,
- das Realisierbarkeitsprinzip,

bestimmen die Herangehensweise des Buches an die Probleme nicht nur der Computertechnik, sondern der Informatik überhaupt.

Die Prinzipien ergeben sich aus dem Begriff der Informatik, wie er in Kap.3.1 definiert wird und im Untertitel des Buches verwendet ist. In Kap.3.1 wird die **Informatik** als *Lehre vom aktiven sprachlichen Modellieren* definiert. Ein Modell heißt *sprachlich*, wenn es das Original in codierter Form wiedergibt ("beschreibt"). Es heißt aktiv, wenn der Träger (z.B. der Computer) als zum Modell gehörig betrachtet wird. Dies ist ein Beispiel für die Verwendung eines gängigen Wortes in einer unüblichen Bedeutung. Die so definierte *Wissenschaft Informatik* besitzt viele Bezüge zur *Kognitionswissenschaft*, der Lehre von den menschlichen Methoden der Organisation und Verarbeitung von Wissen. Die Verwandtschaft beider Wissenschaften tritt besonders deutlich in den Kapiteln 2, 5, 7, im gesamten Teil 3 sowie in Kap.21 zutage. Der Gegenstand der Kognitionswissenschaft überdeckt sich weitgehend mit dem der Human-IV (s.o).

Ein anderes Beispiel für die Verwendung eines gängigen Wortes in einer unüblichen Bedeutung ist das Wort "Intelligenz". In diesem Buch wird mit **Intelligenz** die *Fähigkeit zum sprachlichen Modellieren* bezeichnet, und das Computermodell dieser menschlichen Fähigkeit, m.a.W. die simulierte natürliche Intelligenz, wird **künstliche Intelligenz** genannt.

Als Originale, also als Objekte des sprachlichen Modellierens, werden i.d.R. nur Ausschnitte der Realität, des "Wirklichen" betrachtet. Das stellt insofern eine Beschränkung dar, als der Mensch auch das "Unwirkliche" denken kann. Er kann beispielsweise bewusst oder unbewusst etwas Sinnloses denken und auch artikulieren. Ferner kann er sein eigenes Denken modellieren. Sein Denken kann zu endlosen

Schlussfolgerungsketten oder zu Antinomien führen. All diese Möglichkeiten sind nicht der eigentliche Gegenstand des Buches und werden nur am Rande erwähnt.

Diese Beschränkung bedeutet einen Verzicht auf viele tiefgründige und faszinierende Fragen, auf die man stößt, wenn man über das *Denken* nachdenkt. Doch kann man auf die Behandlung dieser Fragen verzichten, wenn man verstehen will, wie ein Computer funktioniert und was er kann. Wer sich für die hier ausgegrenzten Probleme interessiert, findet eine ausführliche und gleichzeitig kurzweilige Darstellung mit vielen Beispielen in [Hofstadter 85].

Auf ein anderes Buch möchte ich besonders hinweisen, weil es gewissermaßen ein Pendant zu diesem Buch darstellt. Es handelt sich um das Buch "Nichtphysikalische Grundlagen der Informationstechnik" von Siegfried Wendt [Wendt 89]. Dort wird - ebenso wie in diesem Buch - versucht, ein einheitliches Gebäude der Informatik zu "konstruieren". Ausgangspunkt ist jedoch nicht das Trägerprinzip; das Fundament der Konstruktion ist nicht das Stofflich-physikalische, nicht das *Naturnotwendige*, sondern das *Logische*, das *Denknotwendige*. Im Titel des vorliegenden Buches kennzeichnet das Wort "kausal" die Herangehensweise im Sinne des Trägerprinzips. Das bedeutet, dass die Informatik nicht als Gebäude logischer, sondern kausaler Beziehungen entwickelt wird und dass die elementaren Bausteine des Gebäudes nicht Zeichen oder Code, sondern codierende Zustände des Trägers sind. "*Kausale* Informatik" ist als Pendant zu einer "nichtkausalen", d.h. zu einer rein logischen Informatik zu verstehen, keinesfalls als Gegenentwurf zu einer "akausalen" Informatik , was keinen Sinn ergäbe. In dem Wort "kausal" findet folgender Sachverhalt seinen kurzen und treffenden Ausdruck. Sprachliches Modellieren beruht letzten Endes auf Abbildungen kausaler Ereignisketten im Original in kausale Ereignisketten im Modellträger (Computer oder Gehirn) sowie in Abbildungen zwischen logischen Schlussketten über das Original und kausalen Ereignisketten im Modellträger.

Die kausale Herangehensweise liegt dem Gedankengang zugrunde, der das gesamte Buch durchzieht. Sein Ziel ist die kausale Beschreibung des sprachlichen Modellierens. Am Anfang und am Ende des Buches steht jeweils ein Zitat von MAX PLANCK; das eine ist diesem Abschnitt, das andere dem Kapitel 22 als Motto vorangestellt. Im Sinne dieser Zitate artikuliert der Titel des Buches das *notwendige*, aber *unerreichbare* Ziel der "Wissenschaft Informatik". Das Adjektiv "kausal" im Buchtitel ist im Sinne des Planck-Zitats als Forderung nach "*der vollständigen Durchführung der kausalen Betrachtungsweise*" vom physikalischen Modellträger bis zum sprachlichen Modell zu verstehen. Freilich können wir die Forderung nur hinsichtlich der maschinellen Informationsverarbeitung und der künstlichen Intelligenz, nicht aber hinsichtlich der natürlichen Intelligenz erfüllen.

Die Beschränkung auf das *Machbare* bedeutet nicht, dass ich ein reiner Pragmatiker bin. Vielmehr sind es gerade die tieferen menschlichen, gesellschaftlichen und philosophischen Probleme, in welche die Informatik involviert ist, die mich veranlasst haben, meinen Weg zur Informatik und zur künstlichen Intelligenz allgemein-

verständlich und dennoch detailliert darzustellen. Doch bilden diese Probleme nicht den Inhalt des Buches. Auf sie gehe ich sehr kurz im Schlusswort ein.

Ich habe das Buch im Vertrauen auf die Vernunft des Menschen geschrieben, die ihn befähigt, seiner Hilflosigkeit vor der eigenen technischen Machtfülle Herr zu werden.

Hinweise zum Lesen des Buches

Das Lesen, sowohl das Überfliegen als auch das Arbeiten mit dem Buch, wird durch Zusammenfassungen vor den Kapiteln, durch ein Glossar am Ende des Buches und durch unterschiedlichen Druck unterstützt. Neu eingeführte Begriffe und nichtnummerierte Abschnittsüberschriften sind fett gedruckt. Zitate, wichtige Textstellen und gedanklich zu betonende Wörter sind kursiv gedruckt. Kursiv gedruckte Textstellen von besonderer Bedeutung enthalten fettgedruckte Wörter. Programme sind in Schreibmaschinenschrift gedruckt. Sogenannte objektsprachliche Wörter (Wörter, *über* die etwas ausgesagt wird) werden durch Anführungsstriche oder durch Kursivdruck gekennzeichnet.

Das Buch enthält viele Querverweise. Das Verweisen erfolgt nicht über Seitenzahlen, sondern über spezielle *Verweiszahlen*. Textstellen, *auf* die verwiesen wird, sind kapitelweise durchnummeriert. An derjenigen Textstelle, *von* der aus verwiesen wird, ist die Verweiszahl in eckigen Klammern angegeben. Falls auf ein anderes Kapitel verwiesen wird, ist auch die Kapitelnummer angegeben, z.B. [8.12]. Die Textstelle, *auf* die verwiesen wird, findet der Leser über die betreffende Verweiszahl auf dem Rand neben der betreffenden Stelle.

Ich habe mich um eine Darstellung bemüht, die ein *durchgängiges* Lesen ohne Vor- und Zurückblättern ermöglicht, sodass die Verweise i.Allg. außer Acht gelassen werden können. Die Verweise sollen denjenigen helfen, die ein Kapitel ohne Kenntnis der vorangehenden Kapitel lesen möchten, aber auch denjenigen die das Buch nicht nur durchlesen, sondern als in sich geschlossenes Gedankengebäude erfassen wollen. Wenn es jedoch nur darum geht, sich an Definitionen zu erinnern, wird es oft einfacher sein, nicht die Verweise, sondern das Glossar zu benutzen.

Um sich ein Bild vom Inhalt des Buches zu machen, kann es ausreichen, die Zusammenfassungen vor den einzelnen Kapiteln zu lesen. Diejenigen Leser, welche sich für die mathematischen Grundlagen der Informatik weniger interessieren, können die Kapitel, die im Inhaltsverzeichnis mit einem Stern gekennzeichnet sind, ohne wesentlichen Verlust überspringen. Das Entsprechende gilt hinsichtlich der Kapitel 19 und 20 für Leser, die auf Ergänzungen zur Rechnerarchitektur, zu Betriebssystemen und zu Programmiersprachen verzichten wollen. Bei nichtdurchgängiger Lektüre des Buches kann das Glossar gute Dienste leisten.

Teil 1

Grundbegriffe und Grundideen

Zusammenfassung der Kapitel 1 bis 3

Ein relativ abgeschlossener Bewusstseinsinhalt, der im Denken als selbständige Einheit, als Objekt des Denkens auftritt, heißt *Idem*. Ein Ausschnitt der realen Welt, der sich im Bewusstsein eines Menschen widerspiegelt, dem also ein Idem entspricht, heißt *Realem*. Zeichenkörper, z.B. einzelne Zeichen, Zeichenketten oder Muster, die Ideme auslösen, also Bewusstseinsinhalte hervorrufen, heißen *Zeichenrealeme* im Gegensatz zu den *Urrealemen*, den Gegenständen selbst, die durch Zeichenrealeme beschrieben (sprachlich modelliert) werden. Das Zuordnen eines Zeichenkörpers zu einem Idem heißt *Codieren*. Das Zeichenrealem "codiert" das Idem. Darum wird es auch *Code* genannt. Die Zusammenfassung eines Realems mit dem zugeordneten Idem heißt *Information*, die Zusammenfassung eines Zeichenrealems mit dem zugeordneten Idem heißt *Zeicheninformation*. Ein Zeichenrealem ist ein Zeichenkörper, dem ein Artikulierer (Sender) eine Bedeutung (ein Idem) zugeordnet *hat* und dem ein Interpretierer (Empfänger) eine Bedeutung zuordnen *kann*.

Sprache ist Mittel der Codierung und dient der sprachlichen, d.h. codierten Modellierung der Welt. Natürliche Sprachen sind Mittel der Informationsübertragung zwischen Menschen zum Zwecke ihrer Kooperation. Informationsübertragung setzt voraus, dass der Empfänger versteht, was der Sender meint, m.a.W. dass das Idem des Interpretierers mit dem Idem des Artikulierers in ausreichendem Maße zusammenfällt. *Sprechen* bzw. *Denken* ist extern codiertes (mitteilendes) bzw. extern nicht codiertes (nicht mitteilendes, sondern nur "gedachtes") sprachliches Modellieren. *Externes Codieren* und *Artikulieren* sind Synonyme. Die Fähigkeit zum sprachlichen Modellieren heißt *Intelligenz*.

Information ist das Elixier der intellektuellen und kulturellen Evolution. Unter *intellektueller* Evolution wird die Entwicklung der individuellen Denkfähigkeit eines Menschen verstanden, unter *kultureller* Evolution die Entwicklung der sprachlichen Modellierung der Welt durch eine *Kulturgemeinschaft* (über Generationen kooperierende Gemeinschaft von Menschen) und die darauf aufbauende technische, wissenschaftliche, künstlerische und weltanschauliche Entwicklung. Von Generation zu Generation wird Information (Erfahrung) weitergegeben und akkumuliert, sodass eine Entwicklung resultiert. Individuelle wie kulturelle Evolution sind wegen der Notwendigkeit der Informationsweitergabe an Codierung gebunden.

Ein *Symbol* ist eine Zeicheninformation, deren Zeichenkörper relativ kompakt, evtl. ein elementares Zeichen ist und deren Bedeutung für alle Mitglieder einer Kulturgemeinschaft durch Vereinbarung oder Gewohnheit einheitlich festgelegt ist (Additionszeichen, Kruzifix).

Die Informationsverarbeitung durch den Menschen kann auf zwei Ebenen untersucht werden, die als *symbolische* bzw. *subsymbolische* Betrachtungsebene bezeichnet werden, dabei ist der Wortteil "symbol" als Zeicheninformation im allgemeinen Sinne zu verstehen, also nicht unbedingt im speziellen Sinne von "Symbol". Auf der *symbolischen Ebene* wird jeder Code, jede Zeichenkette als interpretierbar angenom-

men; jeder Zeichenkörper ist ein Zeichenrealem. Für die *subsymbolische Ebene* gilt das nicht. Zustände des materiellen *Trägers* der Informationsverarbeitung werden nicht als *codierende* Zustände aufgefasst. “Informationsverarbeitung” ist reine Zeichenverarbeitung und erfolgt ausschließlich nach den Gesetzen der Physiologie des Gehirns. Das Interpretieren (Zuordnen von Bedeutung) ist eine von der Zeichenverarbeitung losgelöste, nachträgliche, bewusste Tätigkeit des Menschen. Auf der subsymbolischen bzw. symbolischen Ebene werden die Prozesse der Informationsverarbeitung aus der Sicht der Neurophysiologie bzw. der Psychologie untersucht. Informationsverarbeitung durch den Menschen lässt sich auf symbolischer wie auf subsymbolischer Ebene technisch simulieren. Als Träger der Simulationsprozesse (der technischen Informationsverarbeitung) dient der Computer.

Informatik ist die Wissenschaft und die Technik der aktiven (den Träger einschließenden) sprachlichen Modellierung. Es wird zwischen *biologischer* und *technischer Informatik* unterschieden, je nachdem, ob der Träger der informationellen Prozesse ein biologisches oder ein technisches System ist. Die biologische Informatik umfasst zwei Bereiche, die Lehre von der sprachlichen Modellierung der Welt mit Hilfe der “Sprache des Nervensystems” und die Lehre von der sprachlichen Modellierung des eigenen Organismus mit Hilfe der “Sprache des genetischen Systems. Dieses Buch behandelt die Probleme der Informatik auf der symbolischen Ebene.

1 Codierung, Evolution und Information

Im Anfang war das Wort.
Joh.1,1

Das Motto dieses Kapitels ist sicher einer der tiefsinnigsten Sätze der Weltliteratur. Das Bibelzitat hat die Form eines Aussagesatzes, es ist eine Feststellung, freilich eine sehr unterschiedlich und sehr weit auslegbare Feststellung. Faust versucht, das griechische “logos” nicht mit “Wort”, sondern mit “Kraft” oder “Tat” zu übersetzen. Die folgenden Darlegungen werden uns zu einer anderen, in der Form sehr ähnlichen, inhaltlich aber viel vordergründigeren und eindeutigeren Feststellung führen, zu der Aussage: *Im Anfang war das Symbol*, oder etwas nüchterner: *Im Anfang war der Code.*

1

Um zu erklären, was mit dem Wort “Code” gemeint ist, führen wir zunächst den Begriff des Zeichenkörpers ein. *Ein* ***Zeichenkörper*** *ist entweder ein elementares Zeichen, z.B. ein Buchstabe, oder ein zusammengesetztes Zeichen (sog. Kompositzeichen), z.B. eine Kette oder ein Muster elementarer Zeichen*. Ein Zeichenkörper wird oft kurz *Zeichen* genannt. Nun vereinbaren wir: *Ein* ***Code*** *ist ein realer (akustischer oder visueller) oder ein gedachter Zeichenkörper, der einen bestimmten Bedeutungsinhalt verschlüsselt oder “codiert”*. Die Zuordnung heißt *Codierung*, doch wird dieses Wort in einer allgemeineren Bedeutung verwendet. *Als* ***Codierung*** *wird jede Zuordnung von Zeichenkörpern zu Bedeutungsinhalten oder von Zeichenkörpern zu Zeichenkörpern bezeichnet.* Im letzteren Falle sprechen wir auch von **Umcodierung**. Wenn einem Zeichenkörper ein Bedeutungsinhalt zugeordnet ist, muss dieser bei Umcodierung erhalten bleiben.

Wenn man in dem Bibelzitat “Wort” durch “Code” ersetzt und unter “Anfang” den Beginn der menschlichen Gesellschaft versteht, ergibt sich die fast selbstverständliche Feststellung, dass die Entwicklung der menschlichen Kultur mit der Verwendung von Zeichen als Bedeutungsträgern beginnt. Tatsächlich gilt eine viel allgemeinere Aussage, nämlich: *Evolution ist an Codierung gebunden*. Dabei ist das Wort Evolution in einem erweiterten, aber dennoch eingeschränkten Sinne zu verstehen; es umfasst die genetische, die intellektuelle und die kulturelle Evolution, nicht aber die kosmische Evolution insgesamt. Unter **intellektueller Evolution** soll die Entwicklung der individuellen Denkfähigkeit, speziell des begrifflichen Gebäudes verstanden werden, in welchem ein Mensch denkt, die Entwicklung seines Welt- und Selbstverständnisses. Unter **kultureller Evolution** soll die Entwicklung des Welt- und Selbstverständnisses einer Kulturgemeinschaft und der von ihr hervorgebrachten Artefakte verstanden werden (siehe in Bild 1.1 den Punkt 3 unter “Welt 1”). Genetische, intellektuelle und kulturelle Evolution können unter der Bezeichnung **codierende Evolution** zusammengefasst werden. Diese ist eine Komponente der

kosmischen Evolution. Im Weiteren ist unter Evolution stets codierende Evolution zu verstehen.

2 Evolution - man denke zunächst an die genetische - setzt voraus, dass die Subjekte, welche die Evolution tragen, sich in leicht veränderter, aber existenzfähiger Form reproduzieren können. Wenn die Reproduktion autonom erfolgt, also nicht unter der Leitung einer übergeordneten Instanz, muss eine Reproduktionsanleitung von Glied zu Glied der Evolutionskette weitergegeben werden. Jedes Glied muss seinen Nachfolger "bilden", ihm seine Form geben, d.h. ihn "informieren" im ursprünglichen Sinne des lateinischen Wortes `informare`. Es ist also gar nicht so fernliegend, dasjenige, was durch die Evolutionskette weitergereicht wird und den Nachfolger informiert, als *Information* zu bezeichnen[1]. Die weitergereichte Information muss mindestens drei Bedingungen erfüllen: 1. Sie muss die Reproduktion des Vorgängers ermöglichen; 2. sie muss verändert werden können, wobei die Veränderung so eng begrenzte Wirkungen haben muss, dass die Reproduktion (der Nachfolger) existenzfähig ist; 3. sie muss so kompakt sein, dass ihre Übergabe effektiv möglich ist.

Wenn der so charakterisierte Begriff "Information" genannt wird, dann hat das
3 zunächst wenig mit dem Informationsbegriff zu tun, wie er im Zusammenhang mit der zwischenmenschlichen Kommunikation verwendet wird. Dort lässt er sich folgendermaßen bestimmen: *Eine* ***Information*** *ist ein Zeichenkörper zusammen mit der Bedeutung, die ihm der Artikulierer (Sender) oder ein Interpretierer (Empfänger) zugeordnet hat.* In mathematischer Schreibweise ist eine Information als Paar zu notieren:

Information = (Zeichenkörper, Bedeutung).

Man beachte, dass diese Definition nicht die Übereinstimmung der artikulierten mit der interpretierten Bedeutung verlangt, sodass Fehlinterpretationen möglich sind. Ferner beachte man, dass Artikulieren und Interpretieren Bewusstsein voraussetzen. Danach ist Information an Bewusstsein gekoppelt.

Wir sind zu zwei scheinbar sehr unterschiedlichen Informationsbegriffen gelangt, einem seiner Natur nach "*evolutiven*" und einem "*kommunikativen*". Analysiert man den kommunikativen Informationsbegriff hinsichtlich der drei Bedingungen, die von evolutiver Information erfüllt werden müssen, stellt man Folgendes fest. Kompaktheit ist eine wesentliche Forderung der sprachlichen Kommunikation und wird durch effektive *Codierung* erreicht, d.h. Bedeutungsinhalte - eventuell sogar sehr komplexe - werden durch kompakte Zeichenkörper (Wörter, Namen, Sätze) codiert (benannt). Das *Interpretieren*, also das Zuordnen einer Bedeutung zu einem Zeichenkörper, entspricht dem Reproduzieren gemäß Vorschrift. Und schließlich kann durch Verändern des Zeichenkörpers (der Zeichenkette) oder durch Fehlinterpretation die Bedeutung verändert werden.

---

1 Der Etymologie des Informationsbegriffs ist R. Capurro in [Capurro 78] nachgegangen.

Die Verwendung des Wortes Information für das, was durch eine Evolutionskette läuft, scheint also gerechtfertigt zu sein. Dabei darf aber ein wesentlicher Umstand nicht übersehen werden. Die Definition des kommunikativen Informationsbegriffs basiert darauf, dass die verwendeten Zeichenkörper Träger von Bedeutungsinhalten sind, die für die Kommunikationspartner durch Konvention (Vereinbarung, Gewöhnung) einheitlich festgelegt sind. Verwendet man diesen Informationsbegriff im Zusammenhang mit der genetischen Evolution, ergibt sich eine Ungereimtheit. Die Bedeutung des genetischen Codes ist keine Sache der Konvention und ist nicht an Bewusstsein gekoppelt.

Offensichtlich gehört der Begriff der *genetischen Information* einer begrifflichen Ebene an, die unterhalb derjenigen liegt, auf der kommunikative Information ausgetauscht wird; dabei bedeutet “unterhalb” soviel wie “näher an der organischen Basis, am stofflichen Träger”. Es handelt sich um zwei Betrachtungsebenen, die als *symbolische* bzw. *subsymbolische* Ebene bezeichnet werden. Bevor wir sie charakterisieren, holen wir die Bestimmung des Symbolbegriffs nach, obwohl wir ihn wegen seiner Vieldeutigkeit nach Möglichkeit vermeiden werden.

Ein *Symbol* ist eine Information (präziser eine Zeicheninformation; siehe Kap.2 [2.3]) deren Zeichenkörper relativ kompakt, evtl. ein elementares Zeichen ist und deren Bedeutung für alle Mitglieder der Kulturgemeinschaft, in der das Symbol verwendet wird, durch Vereinbarung oder Gewohnheit einheitlich festgelegt ist, man denke beispielsweise an das mathematische Symbol der Addition oder an das religiöse Symbol des Kreuzes. Das Kreuz macht deutlich, wie weit der Bedeutungsumfang, wie “tief die Symbolik” eines Symbols sein kann.

In den Bezeichnungen der beiden Betrachtungsebenen als symbolische bzw. subsymbolische Ebene ist der Wortteil “symbol” als Zeicheninformation im allgemeinen Sinne zu verstehen, nicht unbedingt im speziellen Sinne von “Symbol”. Wenn von Informationsverarbeitung auf subsymbolischer Ebene gesprochen wird, verbindet sich damit die Vorstellung, dass von Prozessen die Rede ist, die stets als physikalisch-chemische Prozesse verstanden werden können, welcher Art und wie komplex die Informationsverarbeitung aus “symbolischer” und damit aus psychologischer Sicht auch sei. In diesem Buch wird unter symbolischer bzw. subsymbolischer Betrachtungsebene Folgendes verstanden.

4

*Auf der **symbolischen Ebene** werden Zeichenkörper stets in Verbindung mit ihrer Bedeutung betrachtet; es werden also Informationen und ihre Verarbeitung betrachtet.* Die semantische Belegung der Zeichenkörper wird grundsätzlich als bereits erfolgt angenommen. Der Computer “weiß” von ihr nichts. *Auf der **subsymbolischen Ebene** werden primär keine Informationen betrachtet, sondern nur stoffliche Träger informationeller Prozesse.* Die Zuordnung von Bedeutungen zu bestimmten *Zuständen* im Träger wird nicht vorausgesetzt. “Informationsverarbeitung” ist reine Zeichenverarbeitung (“Zeichen” im Sinne von “Zustand ohne Bedeutung”) und erfolgt ausschließlich nach den Gesetzen der Physiologie des Gehirns. Das Interpretieren (Zuordnen von Bedeutung) ist eine von der Zeichenverarbeitung losgelöste, sekun-

däre, evtl. bewusste Tätigkeit des Menschen. Es kann *nach*, aber auch *während* der Zeichenverarbeitung erfolgen, z.B. durch Gewöhnung oder Lernen[2]. Wenn eine Zuordnung getroffen wird, sodass die Zustände zu Codes werden, bedeutet das den "Aufstieg" auf die symbolische Ebene.

Im Zusammenhang mit dem genetischen Code wird zuweilen von Symbolen gesprochen. Das bedeutet ein stillschweigendes Überwechseln von der subsymbolischen Ebene der chemischen Verbindungen bzw. der Genotypen in die symbolische Ebene der Phänotypen, der in den "Symbolen codierten" Eigenschaften von Individuen. Die Zusammenfassung eines Genotyps mit seinem Phänotyp stellt eine genetische "Information" (im Sinne unserer Definition) dar, eine Erb-Information.

Informationsverarbeitung durch den Menschen ist im Grunde immer sprachliches Modellieren und kann auf beiden Betrachtungsebenen auf dem Computer simuliert werden. Die Entwicklung der Informatik und der *künstlichen Intelligenz* verlief über Jahrzehnte auf der symbolischen Ebene. Erst in jüngerer Zeit besinnt man sich auf die tieferliegende, subsymbolische Ebene. Das hat zur Aufspaltung der *technischen* Informatik, d.h. der Lehre von der Informationsverarbeitung durch technische Systeme, in zwei Teilgebiete geführt, die "*traditionelle Informatik*" und die "*technische Neuroinformatik*", deren Gegenstand künstliche neuronale Netze sind. Letztere finden in diesem Buch nur am Rande Erwähnung.

Die fundamentalste Frage, die man hinsichtlich der Information stellen kann, ist die nach ihrem Ursprung. Dahinter verbirgt sich die Frage nach dem Ursprung der Evolution. Einen exakten naturwissenschaftlichen Lösungsansatz dieser Frage in Bezug auf die *genetische* Evolution und damit auf den Ursprung des Lebens hat MANFRED EIGEN vorgeschlagen, wobei er explizit von *Entstehung von Information* spricht [Eigen 76].

Bezüglich der *intellektuellen* und *kulturellen* Evolution bewegt sich die Diskussion im Bereich der Philosophie. Man sucht noch nach den geeigneten Begriffen. Einen bedeutungsvollen Schritt auf diesem Wege hat KARL POPPER mit der Einführung seiner drei Welten getan [Popper 82] (siehe Bild 1.1[3]). Welt 1 umfasst die physischen Gegebenheiten, Gegenstände und Zustände (anorganische und organische, natürliche und künstliche), Welt 2 die Bewusstseinszustände (subjektives Wissen, Erfahrungen) und Welt 3 das objektivierte und materialisierte Wissen (kulturelles Erbe, theoretische Systeme). Der Bezug der drei Welten zu den Begriffen Evolution und Information ist offensichtlich. Welt 2 bzw. Welt 3 beinhaltet die in der intellektuellen bzw. kulturellen Evolution weitergereichte Information. Der Sprung von der symbolischen in die subsymbolische Ebene bei der Betrachtung der

---

2 In [Fleissner 98] wird anhand eines Computerspiels "Der blinde Springer" ein sehr durchsichtiger Mechanismus simuliert, nach welchem sich zwei Partner während ihrer Kooperation an bestimmte Zeichen (Symbole) gewöhnen, mit deren Hilfe sie sich nach einer Lernphase (Gewöhnungsphase) fehlerfrei verständigen können.

3 Entnommen aus [Eccles 88]

Welt 1 Physische Objekte und Zustände	**Welt 2** Bewusstseins-zustände	**Welt 3** Wissen im objektiven Sinn
1. Anorganische Materie und Energie des Kosmos 2. Biologie: Struktur und Wirkung aller lebenden Wesen, menschliches Gehirn 3. Artefakte: materielle Substrate menschlicher Kreativität: Werkzeuge Maschinen, Bücher, Kunstwerke, Musik	Subjektive Erkenntnisse, Erfahrungen von: Wahrnehmungen, Denken, Emotionen zielgerichteten Strebungen, Erinnerungen Träumen schöpferischer Phantasie	1. Aufzeichnungen intellektueller Arbeiten: philosophische, theologische, wissenschaftl., geschichtliche literarische, künstlerische, technologische 2. Theoretische Systeme Wissenschaftliche Probleme, kritische Argumente

Bild 1.1 Die drei Welten Karl Poppers.

Informationsverarbeitung durch das menschliche Gehirn bedeutet den Übergang von der Welt 2 in die Welt 1 als Träger der intellektuellen Information.

Abschließend soll ein kurzer Vergleich unseres derzeitigen Erkenntnisstandes hinsichtlich der Informationsverarbeitung durch das genetische System einerseits und durch das Zentralnervensystems (ZNS) andrerseits angestellt werden. Der Kürze halber werden wir von DNS-Code bzw. ZNS-Code sprechen, womit die Realisierung der Codierung durch die DNS (Desoxyribonucleinsäure) bzw. das ZNS gemeint ist. Hinsichtlich der Einsichten in die Realisierung des Codes ist die genetische Forschung erheblich weiter fortgeschritten als die Gehirnforschung. Die Struktur der DNS ist praktisch vollständig als Folge der Zeichen des genetischen Alphabets erkannt.

Dieser Erfolg wird in den Medien zuweilen durch spektakuläre Meldungen wie z.B. "Der genetische Code ist geknackt" ziemlich irreführend dargestellt. Üblicherweise wird unter dem "Knacken" eines Codes, z.B. eines Geheimcodes, dessen Entschlüsselung verstanden, was bedeutet, dass die verschlüsselte Nachricht *verstan-*

den worden ist. Davon kann jedoch hinsichtlich des genetischen Codes nicht die Rede sein. Die Aufgabe der Genforscher ähnelt derjenigen eines Archäologen, der einen Text entschlüsseln will, der auf den Scherben einer zerbrochenen Tontafel in einer ihm unbekannten Sprache und in unbekannten Zeichen aufgeschrieben ist. Die einzelnen Zeichen hat der Archäologe bereits entziffert und die Serben hat er wie die Teile eines Puzzles weitgehend zusammengesetzt; aber die ungleich schwierigere Arbeit steht noch bevor, nämlich das Entschlüsseln, d.h. das "Übersetzen" des Textes in eine moderne Sprache. Für das Entschlüsseln werden die Genforscher voraussichtlich noch Jahrzehnte benötigen. Entschlüsseln bedeutet in diesem Falle das Verstehen der Funktion des genetischen Codes, d.h. die Einsicht in seine Wirkung auf die Merkmale des durch die DNS "beschriebenen" Individuums.

Über die Realisierung des ZNS-Codes gibt es zwar viele Hypothesen, doch sind die neurophysiologischen Befunde noch sehr mager. Die experimentelle Technik reicht gegenwärtig nicht aus, um ohne körperliche Eingriffe ausreichend scharf in das Gehirn hineinsehen zu können. Wir wissen nicht, wie ein Bewusstseinsinhalt, wie Denken und wie Gefühle intern "codiert" sind. Wir wissen nicht einmal, ob es in Analogie zum "DNS-Alphabet" überhaupt so etwas wie ein "ZNS-Alphabet" gibt. Angesichts der hohen Parallelität der Tätigkeit des Gehirns wird der Begriff des Alphabets wahrscheinlich wenig sinnvoll sein. Ferner ist anzunehmen, dass das Gehirn im Gegensatz zum genetischen System mit *dynamischer* Codierung arbeitet, worauf in Kap.9.1 eingegangen wird.

2 Gedanke und Sprache

Über die selbstverständlichsten Phänomene wissen wir wenig oder nichts.

Moshé Feldenkrais

Dieser Satz von Feldenkrais trifft insbesondere für die Sprache zu. Was Sprache "wirklich" ist, wie sie "wirkt", wird uns noch lange beschäftigen. Nach den vorangehenden Überlegungen ist Sprache ein Ergebnis der Evolution. Dass die Evolution Sprache hervorbringt, wird verständlich, wenn man bedenkt, dass die Überlebenschancen einer Population, speziell einer menschlichen Gemeinschaft oder Gesellschaft, im Kampf mit der Natur und im Konkurrenzkampf mit anderen Populationen umso günstiger sind, je besser die Individuen der Population miteinander kooperieren. Kooperation aber setzt Kommunikation voraus. Im frühen Stadium der Entwicklung erfolgt sie mittels einfacher Signale, später mittels Sprache. *Natürliche Sprachen sind Mittel der Kommunikation zwischen Menschen zum Zwecke ihrer Kooperation.*

Folglich muss Sprache der Beschreibung der Umwelt dienen, in der die Kooperation stattfindet, die Beschreibung der Ziele und Handlungen der Kooperationspartner eingeschlossen. Mit Hilfe der Sprache muss sich die Welt (die Umwelt des Sprechenden) *modellieren* lassen. Insofern stellt Sprache ein Mittel der Modellierung dar, der *sprachlichen Modellierung*. *Ein **sprachliches Modell** ist eine durch Idealisierung vereinfachte Beschreibung eines Originals*.[1] Eine Aussage über die Welt, ein *Aussagesatz*, ist demzufolge die primäre sprachliche Einheit. Das zeigt sich z.B. darin, dass Kinder ihre Muttersprache in Form ganzer Sätze lernen. Tatsächlich enthält jede abgeschlossene Äußerung die Bedeutung eines ganzen Satzes. Das können auch einzelne Wörter sein, wie z.B. die Entgegnung "Doch!" oder der Ausruf "Au!".

Für das sprechende Subjekt ist Sprechen das Artikulieren (das sprachliche Wiedergeben) von gedachten Sachverhalten. Objektiv betrachtet, aus der Sicht des Biologen, ist dieser Wiedergabeprozess sehr kompliziert. Subjektiv, introspektiv ist er dagegen so einfach, so selbstverständlich, dass man meinen könnte, Denken und Sprechen sei ein und dasselbe. Dass dies ein Irrtum ist, erkennt ein Mensch, welcher mehrere Sprachen perfekt beherrscht, spätestens dann, wenn er sich fragt oder gefragt wird, in welcher Sprache er denkt. Dass hier ein Missverständnis vorliegen muss, wird anhand der Frage deutlich, in welcher Sprache ein Dolmetscher denkt, der simultan übersetzt. Offensichtlich ist das Sprechen ein dem Denken nachgeschalteter Prozess, für den verschiedene sprachzpezifische Reflexsysteme "eingeschaltet" werden können. Das Entsprechende gilt auch für den entgegengesetzten Prozess, das Hören und Verstehen sprachlicher Äußerungen anderer.

Die Selbstverständlichkeit des Sprechens ist einer der Gründe für unser mangelndes Verständnis dessen, was wir mit *Denken* und *Sprechen* bezeichnen, und für die Schwierigkeiten, die mit der Bestimmung der Begriffe Gedanke und Sprache verbunden sind, sowie für die Missverständnisse, die solche Begriffe wie Bedeutung oder *Semantik* eines sprachlichen Ausdrucks hervorrufen können. Um von vornherein Irrtümer und Missverständnisse nach Möglichkeit auszuschließen, wollen wir uns von dem üblichen Sprachgebrauch lösen. Zu diesem Zweck führen wir zwei neue Wörter für scheinbar geläufige Inhalte ein, die Wörter *Realem* und *Idem* (mit der Betonung jeweils auf der letzten Silbe).

Ein Bewusstseinsinhalt oder Bewusstseinsausschnitt, der relativ abgeschlossen ist und der im Denken als selbständige Einheit fungiert, heißt ***Idem****. Ein Ausschnitt der realen Welt, der sich im Bewusstsein eines Menschen widerspiegelt, dem also ein Idem entspricht, heißt das dem Idem zugeordnete* ***Realem****.*

Die Wörter *Idem* und *Realem* sind nach dem Vorbild solcher Wörter wie *Morphem* oder *Lexem* gebildet. Die Begriffsbestimmung setzt voraus, dass eine reale Außenwelt tatsächlich existiert und dass die Vorstellung eines realen Objektes ihre Entsprechung in der Außenwelt besitzt. Dass sich beides nicht beweisen lässt, ist Ursache für uralte philosophische Diskussionen.

Wir gehen davon aus, dass Gedanken und Sprachen, wie eingangs dargelegt, primär der Widerspiegelung einer unabhängig von uns existierenden Wirklichkeit dienen. In diesem Sinne verbindet unsere Definition ganz unbekümmert etwas Reelles, das Realem, mit etwas Ideellem, dem Idem, allerdings in unsymmetrischer Weise. Ein Realem setzt die Existenz eines zugeordneten Idems voraus. Das Umgekehrte gilt nicht. Die Phantasie und die Abstraktionsfähigkeit des Menschen ermöglichen die Bildung von Idemen, die keine Entsprechung in der realen Welt besitzen. Die Frage, ob es Dinge gibt, die sich nicht im Bewusstsein widerspiegeln können, für die es also keine Ideme geben kann, sind vom Menschen nicht zu beantworten, denn was nicht in seinem Bewusstsein existiert, das existiert für ihn überhaupt nicht. Dieser Schverhalt lässt sich auch so ausdrücken: *Die Welt, in der ein Mensch lebt, wird durch sein Bewusstsein* ***definiert****.*

Eine besondere Klasse von Realemen sind die Zeichenrealeme. Ein Zeichenrealem ist die materielle Realisierung eines Codes. Hier ist eine Präzisierung des Begriffs "Code" (und ebenso der Begriffe "Name" und "Satz"), wie er im vorangehenden Kapitel [1.1] verwendet wurde, notwendig. Genau genommen muss nämlich zwischen abstraktem (gedachtem) und konkretem (materialisiertem) Code bzw. Namen oder Satz unterschieden werden.

Ein Code braucht einen Träger, um übertragen und verarbeitet werden zu können, er muss materialisiert sein, z.B. als Schwärzungsmuster, als Lochungsmuster, als Magnetisierungsmuster oder auch als Schwingungsmuster der Luftdichte (Schallwellen) u.ä.m. All diese Muster sind Realeme und gleichzeitig Zeichen. Darum sprechen wir von **Zeichenrealemen**. Im Unterschied dazu nennen wir das ursprüngliche, also das bezeichnete Realem **Urrealem**. Analog ist zwischen dem ursprüngli-

chen Bewusstseinsinhalt, dem **Uridem** und seiner gedachten Bezeichnung, dem **Zeichenidem** zu unterscheiden. Den Uridemen bzw. Zeichenidemen entspricht in gewissem Sinne das erste bzw. zweite Signalsystem nach PAWLOV.

Mit den eingeführten vier Begriffen sind vier Objektklassen definiert, deren Beziehungen untereinander die Linguistik untersucht. In Bild 2.1 sind sie in ihrem Zusammenhang dargestellt. In Klammern sind einige linguistische Begriffe hinzugefügt, die unseren Begriffen etwa entsprechen, jedoch in der Literatur nicht einheitlich verwendet werden.

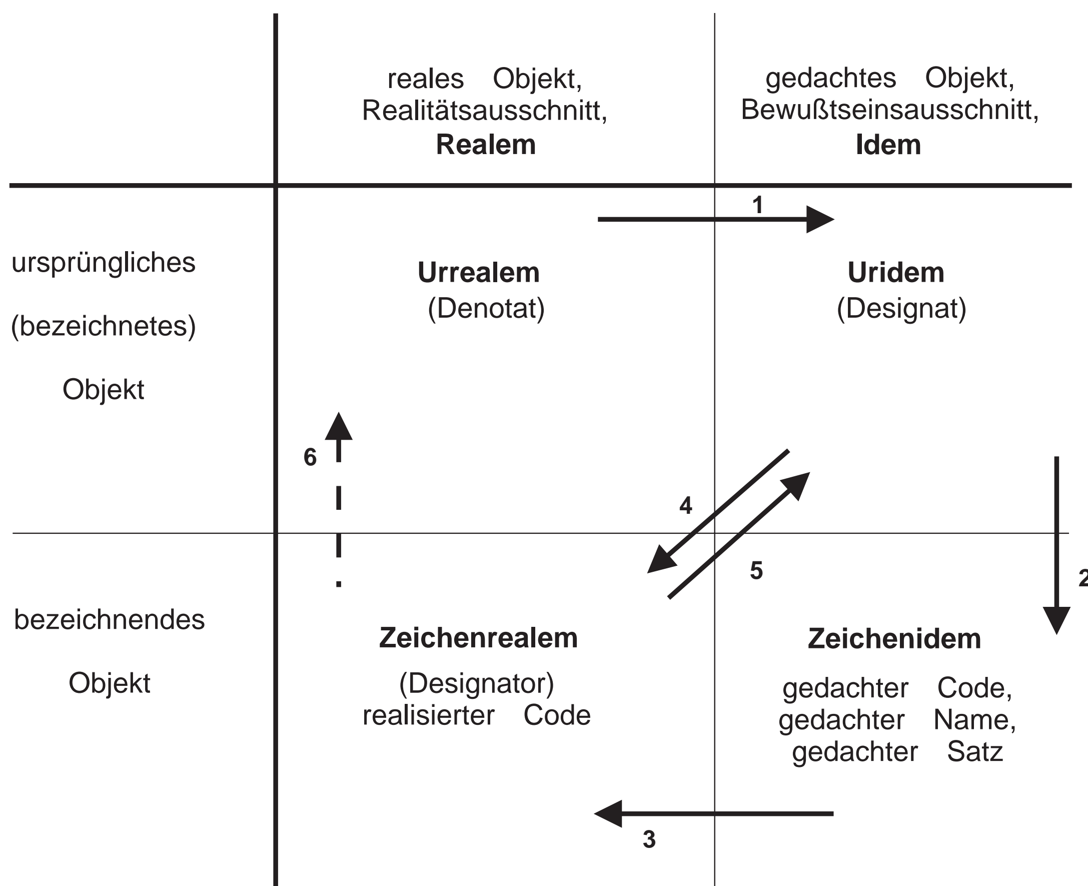


Bild 2.1 Abbildung des sprachlichen Modellierens.
1- Interpretieren; 2 - internes Codieren, Benennen; 3 - externes Codieren, Ausprägen, materielles Instanzieren; 4 - Artikulieren; 5 - Interpretieren; 6- Realisieren eines sprachlichen Modells.

Die Pfeile geben die Übergänge zwischen den Objektklassen an. Sie sind mit den Bezeichnungen für die jeweiligen Übergangsprozesse benannt. *Artikulieren* ist das *Codieren* gedachter Bedeutungsinhalte mit Hilfe einer vereinbarten Sprache. Es erfolgt in zwei Schritten, dem *internen* und dem *externen Codieren*. (Man beachte, dass unser *Code*begriff die Begriffe *Zeichenrealem* und *Zeichenidem* umfasst.) Mit *Interpretieren* bezeichnen wir das Erkennen oder Ausdeuten von Urrealemen (nichtsymbolischen Objekten der Außenwelt) oder von Zeichenrealemen (symbolischen Objekten der Außenwelt).

Das Wort *Instanzieren*[1] (hier als Synonym zu *Ausprägen* verwendet) hat sich erst in jüngerer Zeit in der Informatik eingebürgert. Es kommt nicht vom Wort ”Instanz" im Sinne von Dienststelle, sondern vom englischen Wort “instance” im Sinne von “einzelner Fall” oder “Beispiel” und bezeichnet ganz allgemein den Übergang von einem Sammelbegriff oder von einer Klasse zu einem Exemplar der Klasse (vgl. Bild 5.5). Speziell bezeichnet materielles Instanzieren das Produzieren eines materiellen Exemplars, z.B. das hier gedruckte “A” als ein Exemplar des ersten Buchstabens des Alphabets. Mit *materiellem Instanzieren* kann man auch das *Prägen* eines Datenträgers, z.B. das Ein*stanzen* von Löchern in einen Lochstreifen oder eine Lochkarte assoziieren , was heutzutage allerdings praktisch ausgestorben ist.

Die Wege über die Pfeilfolgen 1-2-3 und 1-4 entsprechen beide dem externsprachlichen Modellieren. Die Pfeile zeigen vom Original zum Abbild (bzw. zur Widerspiegelung). Nur der gestrichelte Pfeil 6 zeigt in entgegengesetzter Richtung, wenn er als Abbildung der *sprachlichen* Modellierung aufgefasst wird. Er kann aber auch als Abbildung der *nichtsprachlichen* (*analogen*) Modellierung aufgefasst werden, von der in Kap.4 die Rede sein wird. Dann zeigt er von einem externsprachlichen Realem zu einem realen Objekt (z.B. von einer Differenzialgleichung zu einem entsprechend programmierten Analogrechner; siehe Kap.4.2 [4.5]).

Die eingeführten Begriffe sollen anhand eines Beispiels illustriert werden. Urrealem (primäres Objekt) möge mein alter Bekannter namens *Hans* sein. Wenn ich auf der Straße eine Person sehe und als *Hans* erkenne, so heißt dies, dass *Hans* in mein Bewusstsein getreten ist, m.a.W. dass mein Gehirn das gesehene Objekt als *Hans* interpretiert hat, dass also das Idem *Hans* ausgelöst wird. Angenommen, wir haben uns lange nicht gesehen, sodass ich mich erst auf seinen Namen besinnen muss. Erst wenn dieser in mein Bewusstsein getreten ist, kann ich meinen Bekannten anreden, d.h. den gedachten Namen aussprechen (materiell instanzieren).

Die Selbstverständlichkeit des Sprechens, z.B. des Anredens mit Namen, hat ihre Ursache darin, dass einem das Interpretieren und gedankliche Benennen nicht zum Bewusstsein kommt, sondern reflektorisch überbrückt ist. Tatsächlich haben sich diese Stufen des bewussten Artikulierens erst im Laufe der genetischen Evolution herausgebildet. Eine motorische Reaktion z.B. des Kehlkopfes auf eine Empfindung, z.B. eine visuelle oder akustische, setzt noch kein bewusstes Denken in Vorstellungen, Begriffen und Symbolen voraus. Auch Tiere können auf Eindrücke mit Lauten reagieren. Die Lautbildung erfolgt also ursprünglich auf subsymbolischer Ebene [1.4].

Mit Hilfe der eingeführten Begriffe können wir die in Kap.1 [1.3] gegebene
3 Definition der *Information* präzisieren: *Die Zusammenfassung eines Idems mit dem zugeordneten Realem heißt* ***Information****. Im Falle eines Zeichenrealem heißt sie*

---

1 In der Literatur wird i.Allg. “Instantiieren” geschrieben. Im Sinne der neuen Rechtschreibung ist “Instanzieren” vorzuziehen.

artikulierte Information oder ***Zeicheninformation****. Andernfalls heißt sie nichtartikulierte oder* ***uneigentliche Information****.* Wenn nichts anderes gesagt wird, ist im Weiteren unter Information stets Zeicheninformation zu verstehen. Dem Begriff der Zeicheninformation entspricht etwa der linguistische Begriff des *Denotators* (siehe z.B. [Conrad 75]. Der in Kap.1 eingeführte Symbolbegriff kann nun auch als *vereinbarte Zeicheninformation* definiert werden.

Die Präzisierung des Informationsbegriffs hat mehrere Konsequenzen. Zunächst einmal bedeutet sie, dass nicht nur ein Idem, sondern auch eine Information an einen individuellen Träger, an den Sender oder Empfänger gebunden ist (insofern impliziert der Informationsbegriff den Sender bzw. Empfänger). Diese Vorstellung ist nicht üblich, bedeutet aber eine begriffliche Klärung. Danach schließt nämlich das Übertragen von Information nicht nur das Transportiern, sondern auch das Artikulieren und Interpretieren ein. Information wird also nur soweit übertragen, als Senderidem und Empfängeridem zusammenfallen. Andernfalls kommt es zu Missverständnissen. Oft wird das Wort *Informationsübertragung* irreführenderweise im Sinne von *Zeichenrealemübetragung* verwendet.

Eine andere Konsequenz ergibt sich aus der Möglichkeit der mehrfachen Instanzierung von Zeichenidemen sowie der Vervielfältigung von Zeichenrealemen und damit von Informationen. Es gibt also keinen Erhaltungssatz der Information, wie er z.B. für die Energie gilt. Hier wird deutlich, dass in diesem Buch der Informationsbegriff nicht in der üblichen Weise verwendet wird. Wenn beispielsweise jemand ein und dieselbe Auskunft zweimal erhält, handelt es sich nach obiger Definition um verschiedenen Informationen. Zwar sind die Ideme die gleichen, doch die Realeme sind unterschiedliche, z.B. zwei verschiedene bedruckte Blätter.

Aber nicht nur das Instanzieren, sondern auch das Benennen (Codieren) kann wiederholt werden. Es kann auch reflexiv (zirkulär) erfolgen; ein Name kann sich selbst benennen. Auf die Konsequenzen und begrifflichen Schwierigkeiten, die sich daraus ergeben, gehen wir in Kap.6 ein.

Abschließend sei auf ein fundamentales Problem des sprachlichen Modellierens hingewiesen. Wie ist es möglich, die unendliche Vielfalt der Welt mit den endlichen Mitteln der Sprache darzustellen? Man ist daran gewöhnt, dass man alles, was man sieht, beschreiben kann, und kommt gar nicht auf den Gedanken, sich darüber zu wundern, dass das überhaupt möglich ist, wie man sich eben auch nicht darüber wundert, dass man das, was man denkt, auch sagen kann, wovon eingangs die Rede war. In Kap.5 werden wir näher untersuchen, wie die Evolution das angedeutete Problem gelöst hat. Dabei handelt es sich nicht um die genetische, sondern um die intellektuelle und die kulturelle Evolution. *Sprache ist Mittel und Gegenstand sowohl der intellektuellen Evolution des Individuums als auch der kulturellen Evolution der Menschheit.*

Nachden wir den Begriff der Information definiert haben, sind wir für das nächste Kapitel gerüstet. Wir werden versuchen, den Begriff der Informatik zu definieren, indem wir den Gegenstand bestimmen, mit dem sich die Informatik als Wissenschaft

beschäftigt. Wir werden uns nicht damit begnügen, eine Reihe von Teilgebieten aufzuzählen, in deren Bezeichnung das Wort “Information” vorkommt, sondern wir werden nach der Rolle suchen, die der zu bestimmende Gegenstand im menschlichen Leben spielt.

3 Informatik - Lehre vom aktiven sprachlichen Modellieren

3.1 Modellklassen. Begriffsbestimmung der Informatik

In der Literatur findet man die unterschiedlichsten Definitionen der Informatik. Meistens handelt es dich um Definitionen durch Aufzählung von Teilgebieten, z.B.: *Die Informatik ist die Wissenschaft von der Verarbeitung, Speicherung und Übertragung von Informationen*, oder: *Die Informatik ist die Wissenschaft von der Erstellung und Nutzung der Hardware und Software informationeller Systeme*, oder mehr aus der Sicht der letztendlichen technischen Zielstellung: *Die Informatik ist die Wissenschaft von der Entwicklung und Nutzung künstlicher Intelligenz.* Zuweilen wird auch einfach gesagt: *Die Informatik ist die Wissenschaft von der Informationsverarbeitung*. Das ist sicher nicht falsch, aber wenig aussagekräftig. Nach diesem Vorbild könnte man beispielsweise die Chemie als Wissenschaft von der Chemikalienverarbeitung definieren. Im Informatik-Duden [Duden 89] ist die Informatik etwas präziser als "*Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Digitalrechnern (Computern)*" definiert, ähnlich in [Werner 95]: "*Die Informatik (computer science) ist eine Wissenschafts- und Technologiedisziplin, die sich mit Methoden und Verfahren der automatisierten Informationsverarbeitung befasst*".

Vielleicht sollte man sich mit einer Definition analog derjenigen begnügen, die DAVID HILBERT für die Mathematik vorgeschlagen hat, und sagen: *Informatik ist das, worüber sich kompetente Personen auf Informatikerkongressen unterhalten*. Hinter diesem Vorschlag steht die Einsicht, dass eine Wissenschaft ihr lebendiger Inhalt ist, der sich kaum durch eine tote Definition erfassen lässt. Das ist sicher richtig. Dennoch soll versucht werden, eine Begriffsbestimmung zu finden, die keine Aufzählung verschiedener Gegenstände der Informatik darstellt und die nicht auf Begriffen wie Information oder Intelligenz beruht.

Gesucht sind nicht die Attribute und Erscheinungsformen der Informationsverarbeitung, sondern ihr wesentlicher Kern, die Wurzel, aus der alles wächst, sich alles "von selber" ergibt, auch der Informationsbegriff. Diesen hatten wir aber bereits eingeführt [1.3] und zwar ausgehend vom *sprachlichen Modellieren*. Danach liegt es nahe, die Informatik als Lehre vom sprachlichen Modellieren zu definieren. Diese Bezeichnung kann aber auch die Physik für sich beanspruchen, im Grunde sogar jede Wissenschaft, die sprachliche - z.B. mathematische - Modelle der Welt entwickelt. Die Definition ist offensichtlich zu allgemein. Die notwendige Präzisierung ergibt sich aus Bild 3.1. Es zeigt eine Klassifikation von Modellen nach verschiedenen Merkmalen.

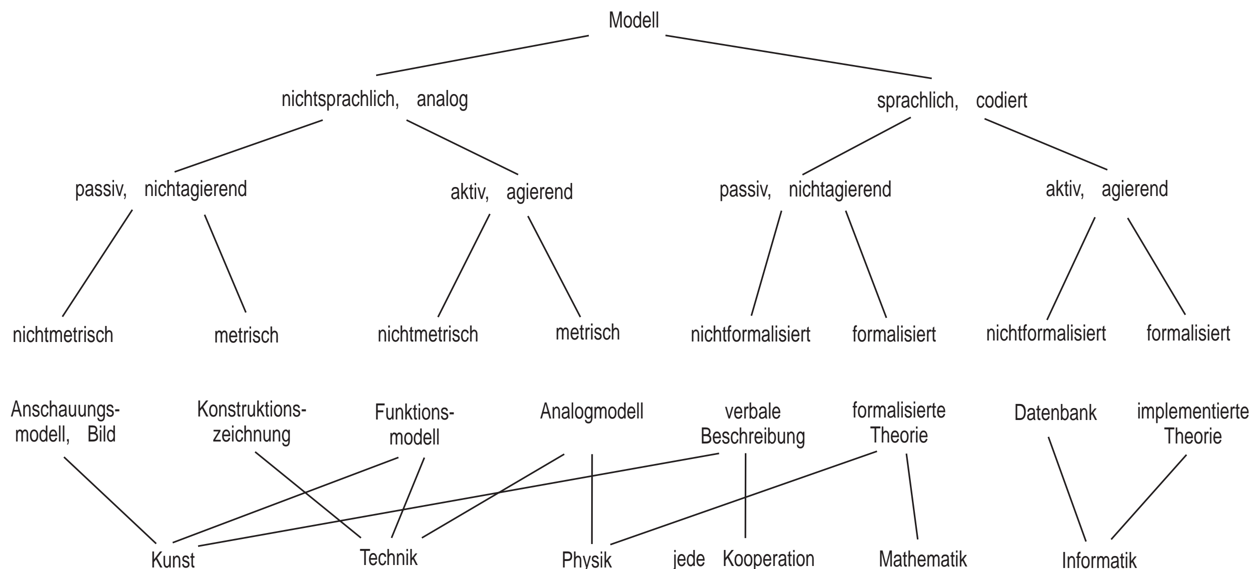


Bild 3.1 Modellklassen

Die dargestellte Systematik unterteilt die Menge möglicher Modelle nach drei Kriterien in je zwei Teilmengen und zwar:

1. Nichtsprachliche oder **analoge** und **sprachliche** oder **codierte Modelle**. *Analoge* Modelle verwenden keine Codierung. Die zu modellierenden Eigenschaften des Originals haben ihr anschauliches Analogon im Modell. Ein sprachliches Modell ist eine Menge von *Aussagen* über das Original. In erweitertem Sinne nennen wir jedes Zeichenrealem, dem ein Urrealem entspricht, sprachliches Modell des Urrealems (des Originals).

2. Aktive oder agierende und **passive** oder **nichtagierende Modelle.** *Aktive analoge* Modelle veranschaulichen Prozesse, indem sie selber agieren (z.B. eine Modelleisenbahn); sie sind dynamisch.

Aktive oder *agierende* sprachliche Modelle sind solche, welche die Modellaussagen selber artikulieren bzw. interpretieren. *Ein aktives sprachliches Modell schließt also das Trägersystem - das kann ein Gerät (Computer) oder ein Mensch sein - in sich ein*. Für passive sprachliche Modelle gilt das nicht. Beispielsweise ist ein Kursbuch ein passives, ein Auskunftsautomat ein aktives sprachliches Modell der Zugverbindungen.

3. Exakte und **nichtexakte Modelle.** *Exakte analoge* Modelle gestatten es, Eigenschaften des Originals durch Zählen oder Messen am Modell zu bestimmen. Aus diesem Grunde sind die Bezeichnungen **metrisches Modell** bzw., falls die Bedingung der Meßbarkeit nicht gegeben ist, **nichtmetrisches Modell** gerechtfertigt.

Vorgreifend auf Kap. 5.4 vereinbaren wir: **Exaktes sprachliches Modellieren** ist das Artikulieren *objektiver* Aussagen über ein Original, d.h. solcher Aussagen, die von allen Beteiligten einheitlich, also *subjektunabhängig* interpretiert werden, wobei

die Objektivierung durch Anbindung der *externen* Semantik an eine *formale* Semantik mittels Formalisierung erreicht wird. Aus diesem Grunde sind die Bezeichnungen **formalisiertes** bzw. - im Falle nichtexakter Artikulation - **nichtformalisiertes Modell** gerechtfertigt.

Bei der soeben getroffendnen Vereinbarung wurde davon ausgegangen, dass die Wortverbindungen "externe Semantik" und "formale Semantik" vom Leser intuitiv richtig verstanden werden. Die Definitionen dieser Begriffe erfolgt in Kap.5.4. Auch die Erklärung des in Bild 3.1 auftretenden Begriffs der "formalisierten Theorie" muss auf Kap. 5.4 [5.11] (dort "*formale* Theorie genannt") verschoben werden, weil vorher der *Kalkülbegriff* [5.9] eingeführt werden muss. Es wird sich herausstellen, dass für die Klassifikation sprachlicher Modelle das Wort "kalkülisiert" noch treffender ist als das Wort "formalisiert".

Die behandelten drei Unterscheidungen von je zwei Fällen liefern insgesamt 8 Modellklassen. In Bild 3.1 sind Beispiele für die verschiedenen Klassen angeführt, die gleichzeitig darauf hinweisen, dass die Grenzen zwischen den Klassen nicht scharf sind. Wie sind z.B. Kunstwerke einzuordnen, die Symbole enthalten, wie beispielsweise Chagalls Bilder den Eselskopf oder Wagners Opern die Klangmotive der Helden. Das sind interessante Fragen, die hier nicht zur Debatte stehen.

Schließlich ist in Bild 3.1 durch Verbindungsgerade (z.B. zwischen "Kunst" und "Bild") angegeben, welche Bereiche menschlicher Tätigkeit sich vorzugsweise welcher Modellklassen bedienen. Genau genommen wären weit mehr Verbindungsgeraden einzuzeichnen, beispielsweise von "verbale Beschreibung" zu sämtlichen Bereichen. Man könnte versuchen, daraus Begriffsbestimmungen z.B. der Physik, der Mathematik und auch der Informatik abzuleiten. Für letztere ergibt sich: *Die **Informatik** ist die Lehre vom aktiven sprachlichen Modellieren*[1], eine ungewöhnliche, aber, wie sich zeigen wird, tragfähige Begriffsbestimmung. Die Tragfähigkeit muss sich im lebendigen Sprachgebrauch erweisen und ihre Konsequenzen müssen diskutiert werden. Gegen sie lassen sich verschiedene Einwände erheben. Auf zwei Einwände soll eingegangen werden.

Erster Einwand. Nach der gegebenen Definition ist ein Mensch mit seinem inneren Modell der Welt ein aktives sprachliches Modell und als solches Gegenstand der Informatik. Die Informatik ist aber eine technische Wissenschaft und der Mensch nicht ihr Gegenstand. Außerdem ist es eine extrem reduktionistische Herangehensweise, im Menschen ein agierendes sprachliches Modell zu sehen. Tatsächlich ist die Auffassung, die Informatik sei eine rein technische Wissenschaft, eine Gewohnheit aus der Vergangenheit, und die Themen heutiger Informatikerkongresse zeigen, dass diese Gewohnheit überlebt ist. Mit dem Vorwurf des Reduktionismus werden wir uns im Nachwort auseinandersetzen. Er kann nur gegen kategorische Erklärungen

---

1 Diese Definition ist erstmalig in [Jungclaussen 88a] veröffentlicht worden.

erhoben werden. Solche ergeben sich jedoch nicht zwangsläufig aus obiger Definition.

Zweiter Einwand. Es ist üblich, zwei Klassen informationeller Systeme zu unterscheiden: *Digitalrechner* und *Analogrechner*. Letztere verwenden keine Codierung und keine Symbole. Sie verarbeiten keine Information im Sinne unserer Definition. Ein "programmierter", d.h. für die Modellierung eines Originalbereiches *konditionierter* (vorbereiteter, konkret: "verdrahteter") Analogrechner ist also kein sprachliches, sondern ein analoges, metrisches Modell. Er ist also weder Mittel noch Gegenstand der Informatik im oben definierten Sinne. Das ist ungewöhnlich, aber konsequent und vermeidet viele Missverständnisse und inhaltlose Diskussionen. Hier sind wir auf einen Sachverhalt gestoßen, der für ein tieferes Verständnis dessen, was Informatik ist, hervorragende Bedeutung hat. Darum soll auf ihn in Kap.4 näher eingegangen werden.

3.2 Fundamente der Informatik

Im vorangehenden Kapitel wurde gefordert, dass sich die Tragfähigkeit der dort gegebenen Definition der Informatik als "Lehre vom aktiven sprachlichen Modellieren" im lebendigen Sprachgebrauch zu erweisen habe. Es soll versucht werden, den Nachweis dadurch zu erbringen, dass aus der gegebenen *intensionalen*, also logisch-inhaltlichen Definition eine *extensionale* Definition abgeleitet wird, also eine Aufzählung der Teilbereiche der Informatik oder, im Sinne von Hilbert, eine Aufzählung dessen, worüber sich Informatiker auf ihren Kongressen unterhalten. Zu diesem Zwecke stellen wir folgende Fragen: Was muss man verstehen, um das aktive sprachliche Modellieren zu verstehen und was muss man tun, um es zu realisieren, kurz: *Was sind die Fundamente der Informatik*? [2]

Bild 3.2 gibt Antwort auf die gestellten Fragen. In ihr sind die wichtigsten Wissenschaftsdisziplinen, die für das Verständnis und die Verwirklichung des aktiven sprachlichen Modellierens die Voraussetzungen liefern, zusammengestellt. Der eigentliche Tabelleninhalt (die beiden rechten Spalten ohne die Kopfzeile) enthält 10 Felder. In jedem Feld ist eine Disziplin (ein Wissenschaftszweig) angegeben, welche die Grundlagen für die Untersuchung bzw. Realisierung des in der 2. Spalte der jeweiligen Zeile bezeichneten Gegenstandes liefert. Dabei ist zwischen natürlichem (biologischem, organismischem) sprachlichem Modellieren (linke Spalte) und künstlichem (technischem) sprachlichem Modellieren (rechte Spalte) unterschieden. Die Teilbereiche sind in 2 Gruppen zusammengefasst. Die Teilbereiche der ersten (oberen) Gruppe beschäftigen sich mit dem Trägersystem. Sie untersuchen die Funktionsprinzipien des *systeminternen* (primär *subsymbolischen*) Modellierens.

2 Mit geringen Änderungen [Jungclaussen 88] entnommen.

Die Teilbereiche der zweiten Gruppe untersuchen die Möglichkeiten und Mittel der *externen* Modellierung, genauer der externen, symbolischen Darstellung interner Modelle.

Ein Beispiel soll das Lesen der Tabelle demonstrieren. Die Wissenschaftsdisziplin, die sich mit den Artikulierungsmitteln beschäftigt, ist im Falle des künstlichen sprachlichen Modellierens die *Wissenschaft von den Programmiersprachen*, im Falle des natürlichen sprachlichen Modellierens die *Sprachwissenschaft*, speziell die *Semiotik*.

	Gegenstand	natürliche Modelle	künstliche Modelle
interne Modelle	Trägersystem	Neurophysiologie	Gerätetechnik der Computer-IV
	Entwicklung bzw. Herstellung des Trägersystems	Wissenschaft von der Evolution des Zentralnervensystems	Hardwaretechnologie
externe Modelle	Artikulierungsmittel	Sprachwissenschaft, Semiotik	Wissenschaft von den Programmiersprachen
	Artikulierung	Schreib- und Redekunst (Rhetorik)	Programmierungstechnik, Softwaretechnologie
	Metamodellierung	Wissenschaft von der natürlichen Intelligenz	Wissenschaft von der künstlichen Intelligenz

Bild 3.2 Fundamente des sprachlichen Modellierens

Die letzte Zeile der Tabelle enthält zwei Begriffe, die erklärt werden müssen. Unter ***Metamodellierung*** ist hier die *externe sprachliche Modellierung der internen sprachlichen Modellierung* zu verstehen. (Nach dem üblichen Sprachgebrauch wäre unter sprachlichem Metamodell (metasprachlichem Modell) das externsprachliche Modell eines *externsprachlichen* Modells zu verstehen). Das Wort “Intelligenz” werden wir hier und im Weiteren in einem Sinne verwenden, der sich ganz natürlich und zwanglos aus den bisherigen Gedankengängen ergibt: ***Intelligenz*** *ist die Fähigkeit ihres Trägers zur Erstellung und Nutzung interner, sprachlicher Modelle. Beschränkt sich die Fähigkeit auf die Nutzung, sprechen wir von reproduktiver, andernfalls von produktiver Intelligenz.* Demnach ist *künstliche Intelligenz* die Fähigkeit *technischer* informationeller Systeme zum sprachlichen Modellieren.

Diese Begriffsbestimmung mag insofern mangelhaft erscheinen, als sie Intelligenz auf das Modellieren beschränkt und das Erfinden als typische Fähigkeit intelligenter Wesen nicht berücksichtigt. Tatsächlich ist aber auch Erfinden sprachliches Modellieren, wenn auch einer noch nicht existierenden Realität, oder besser und allgemeiner: *Sprachliches Modellieren ist Mittel des Erfindens und Problemlösens*. In Kap.7 werden wir untersuchen, inwiefern umgekehrt Erfinden Mittel des sprachlichen Modellierens ist.

Gehirnprozesse, die dem gedanklichen Modellieren der Welt oder dem Erfinden gedachter Welten zugrunde liegen, bezeichnen wir als Denken. Dazu gehört auch das bildliche (zwei- oder dreidimensionale) Modellieren, das *Sich-Vorstellen* der Welt. ***Denken** ist sprachliches Modellieren ohne externes Codieren (Artikulieren).*

Sieht man sich die rechte Spalte von Bild 3.2 genauer an, erkennt man eine Liste von Teilgebieten, die eine extensionale Definition der Informatik darstellt, wie sie häufig zu finden ist und die einer Definition im HILBERTschen Sinne sehr nahe kommt. Diesen Umstand werten wir als Nachweis der Tragfähigkeit unserer intensionalen Definition der Informatik als *Lehre vom aktiven sprachlichen Modellieren*. Man beachte, dass in der rechten Spalte von Bild 3.2 die Fundamente nur eines Teils der Informatik aufgelistet sind und zwar desjenigen Teils, welcher das sprachliche Modellieren durch technische Geräte (Computer) betrifft. Diesen Teil nennen wir **technische Informatik**.[3] Demgegenüber sind in der vorletzten Spalte die Fundamente desjenigen Teils der Informatik aufgelistet, welcher das sprachliche Modellieren durch den Menschen betrifft. Man könnte ihn **Humaninformatik** nennen. Dieser Teil ist ein Unterbereich der **biologischen Informatik**. Sie beschäftigt sich mit der Informationsverarbeitung durch Lebewesen, den Menschen eingeschlossen[4]. Die biologische Informatik umfasst zwei Bereiche, die Lehre von der sprachlichen Modellierung der Welt mit Hilfe der “Sprache des Nervensystems” und die Lehre von der sprachlichen Modellierung des eigenen Organismus mit Hilfe der “Sprache des genetischen Systems”. Die biologische Informatik ist nicht der eigentliche Gegenstand des Buches. Dennoch muss auf die Informationsverarbeitung durch den Menschen eingegangen werden, um verstehen zu können, wie eine sprachliche Modellierung der Welt durch den Computer und wie die Kommunikation zwischen Mensch und Maschine möglich ist.

Wir verlassen jetzt den Bereich der Informatik, um die Grenze zwischen der digitalen Rechentechnik und der analogen Rechentechnik, die nicht zur Informatik (im Sinne dieses Buches) gehört, herauszuarbeiten. In Kap.5 werden wir unseren Gedankengang zum sprachlichen Modellieren fortsetzen.

---

3 Es sei darauf hingewiesen, dass die Wortverbindung “technischen Informatik” in der Literatur zuweilen in anderen, spezielleren Bedeutungen verwendet wird.

4 Mit *Bioinformatik* wird die Anwendung der technischen Informatik in der Biologie bezeichnet, in Analogie zur Wirtschaftsinformatik, der Anwendung der Technischen Informatik im Bereich der Wirtschaft.

4 Analoges und digitales Modellieren

Zusammenfassung

Die kulturelle Evolution hat dazu geführt, dass wir uns Raum, Zeit und Kausalität als Kontinuum vorstellen. Der Versuch, die räumlichen, zeitlichen und kausalen Gegebenheiten der Welt sprachlich zu modellieren, stößt auf eine grundsätzliche Schwierigkeit, auf den Widerspruch zwischen der kontinuierlichen Natur des Gedachten, des Vorgestellten, und der nichtkontinuierlichen Natur des sprachlichen Modellierens, und zwar sowohl des *extern nichtcodierten* Modellierens, des Denkens, als auch des *extern codierten* Modellierens, des Sprechens und Schreibens.

Um die kontinuierliche Welt quantitativ und kontinuierlich modellieren zu können, sind die reellen Zahlen und die Infinitesimalrechnung erfunden worden. Dennoch bleibt das quantitative Modellieren in Zahlen ein diskontinuierliches Modellieren. Die kontinuierliche Folge der reellen Zahlen oder der Punkte einer Linie sind sprachlich nicht realisierbar, sondern nur mit Hilfe abstrakter Begriffe beschreibbar, z.B. mit Hilfe des Begriffs der reellen Zahl, des Grenzwertes oder des Differenzialquotienten.

Das diskontinuierliche sprachliche Modellieren, das Denken und Sprechen, basiert auf der kontinuierlichen physischen Realität (im Sinne der klassischen Physik) als dem stofflichen Träger des Modellierens. Beim Übergang aus dem kontinuierlichen (*analogen*) in den diskontinuierlichen (*digitalen*) Bereich muss eine Analog-digital-Konversion der Merkmalswerte stattfinden, derer sich das Modellieren bedient. Sie kann von einem Gerät (Konverter) oder vom Menschen ausgeführt werden, indem er z.B. einen Wert misst oder berechnet. Ein agierendes Modell heißt analog oder nichtsprachlich, wenn die Konversion nicht Teil des Modells ist, sondern *nach* der Bestimmung der Merkmalswerte vorgenommen wird. Ein Modell heißt digital oder sprachlich, wenn die Konversion *vor* der Bestimmung der Merkmalswerte vorgenommen wird, also Teil des Modells ist.

Ein analoges Modell wird durch dieselben Gleichungen (z.B. Differenzialgleichungen) beschrieben wie das Original. Das Original selbst stellt eine analoge Lösung der Gleichungen dar; ein analoges Modell "*berechnet*" die Lösung, d.h. es generiert die *Lösungsfunktion*. Dafür hat die Analogrechentechnik elektronische Geräte entwickelt. Die Merkmalswerte (die Werte der Lösungsfunktion) sind Werte elektrischer Größen. Sie werden vom Modell in analoger Form geliefert und können analog, z.B. als Zeigerausschlag oder als Strecke auf dem Bildschirm ausgegeben werden. Die Ausgabe kann auch digital als Dezimalzahl erfolgen, wobei der Digitalwert ein Näherungswert ist (falls die betreffende Gleichung nicht ausnahmsweise eine rationalzahlige Lösung besitzt). Für die Digitalausgabe ist ein Konverter erforderlich, der die Grenze des analogen Bereichs überschreitet.

4.1 Messen und reelle Zahlen

In Kap.2 war an Sprachen die Forderung gestellt worden, dass sie die Möglichkeit bieten müssen, die Unendlichkeit der Welt sprachlich wiederzugeben. Dieser Satz demonstriert mit der Verwendung des Wortes *Unendlichkeit* die Methode, wie Sprachen die gestellte Forderung erfüllen können. Es bilden sich im Laufe der Zeit geeignete *Begriffe* heraus, also *benannte Ideme,* wie beispielsweise der Begriff der Unendlichkeit. Das ist ein Beispiel für ein Idem ohne entsprechendes anschauliches Urrealem; das Unendliche ist nicht vorstellbar. Dennoch ist der Begriff nützlich und sogar objektivierbar bis hin zum mathematischen Begriff des Unendlichen, genauer des unendlich Großen. Er ist ein erstaunliches Resultat der kulturellen Evolution und löst den fundamentalen *Widerspruch zwischen der Unendlichkeit des Gedachten und der Endlichkeit des Denkens und der Sprache.* Es ist offensichtlich, dass eine Sprache, genauer die Anzahl ihrer Zeichenrealeme und der ihnen entsprechenden Ideme, endlich sein muss, um ihre Artikulierbarkeit und Interpretierbarkeit zu gewährleisten.

Daneben hat sich der Begriff des “*unendlich Kleinen*” gebildet. Man gelangt zu ihm, wenn man ein Stück Realität, z.B. einen Stein, eine Länge oder ein Zeitintervall gedanklich “unendlich” verkleinert. Intuitiv ist man davon überzeugt, dass unendliches Verkleinern tatsächlich möglich ist, dass die Punkte des Raumes unendlich dicht nebeneinander liegen, dass der Raum ein *Kontinuum* bildet und dass dasselbe auch für die Zeit gilt. Diese Vorstellung ist uns infolge ständiger Erfahrung beim Beobachten und “Begreifen” der Welt in Fleisch und Blut übergegangen.

Der Wunsch (der evolutionäre Zwang), das Kontinuum mit Worten zu erfassen, führt zu einem zweiten fundamentalen Widerspruch des sprachlichen Modellierens, 1 dem *Widerspruch zwischen der kontinuierlichen Natur des Gedachten und der nichtkontinuierlichen Natur des Denkens.* Letztere ergibt sich wiederum aus der Endlichkeit der Menge der Zeichenrealeme und der ihnen entprechenden Ideme. Auch die Begriffe des unendlich Kleinen und des Kontinuums sind objektivierbar und mathematisierbar. Darauf soll näher eingegangen werden, weil hier der Schlüssel zur Unterscheidung zwischen *analogem* und *digitalem* Modellieren liegt.

Wir beginnen mit der gedanklichen und sprachlichen Modellierung des *linearen* (*eindimensionalen*) räumlichen Kontinuums, der *Linie.* Global erfolgt sie eben durch das Wort “Linie”. Eine Linie lässt sich als unendliche Menge aneinandergereihter Punkte gedanklich und sprachlich (begrifflich) modellieren. Die exakte (quantitative, metrische) Modellierung erfolgt durch Zuordnung von Zahlen zu den Punkten der Linie. Dazu werden zwei Punkte markiert und mit 0 und 1 benannt. Das Linienstück zwischen den beiden Punkten ist die Längeneinheit. Nun wird jedem Punkt der Linie diejenige Zahl zugeordnet, die seinen Abstand vom Nullpunkt gemessen in Längeneinheiten angibt, normalerweise in Form einer gebrochenen Dezimalzahl. Wenn die Linie eine Gerade ist, spricht man von einer *Zahlengeraden.*

Die Zuordnung einer Zahl zu einer Länge nach der eben beschriebenen Vorschrift wird *Digitalisierung* oder *Analog-digital-Konvertierung* genannt. Das Wort *digital* kommt vom englischen digit = einstellige Zahl, Ziffer. Das Wort *analog* in diesem Zusammenhang ist das Ergebnis eines Bedeutungswandels, auf den wir noch zu sprechen kommen. Die genannte Konvertierung ist der Übergang vom analogen zum digitalen, in diesem Fall vom geometrisch-kontinuierlichen zum sprachlich-diskreten Modellieren. Das ist leicht gesagt aber schwer getan. Es stellt sich nämlich heraus, dass den meisten Punkten der Zahlengeraden gar keine in Ziffern genau angebbare Zahl entspricht. Das manifestiert sich in der Ungenauigkeit, mit der sich die Länge eines Abschnitts der Zahlengeraden (einer Strecke) messen bzw. berechnen lässt.

2

Die Ungenauigkeit hat unterschiedliche Ursachen. Jedes Messen ist, genau genommen, ein *Klassifizieren.* Alle Längen, die innerhalb eines bestimmten Wertebereichs liegen oder innerhalb der *Messgenauigkeit* einander gleich sind, werden zu einer *Klasse* zusammengefasst, und die Klasse wird mit einer Zahl *benannt,* dem gemessenen Wert. Die Größe des Wertebereichs stellt die minimale Längeneinheit dar, also die Genauigkeit, mit der gemessen wird. Die Prozedur des Messens (Klassifizierens) ist ein Zählen; es wird gezählt, wie oft die Längeneinheit in den zu messenden Abschnitt hineinpasst. Das gilt für alle Messungen, die auf Längenmessungen zurückgeführt werden, z.B. für Druck- oder Temperaturmessungen, die auf die Messung der Länge einer Flüssigkeitssäule (z.B.Quecksilbersäule) zurückgeführt werden, aber auch für alle Messungen mit Hilfe von Zeigerintrumenten. In jedem Fall handelt es sich um ein “Zerstückeln”, ein *Diskretisieren des Kontinuums* und Zuordnung von Zahlen zu den Stücken (nicht zu Punkten), also um ein *Digitalisieren.*

Die Begriffe Messen und Digitalisieren sind demnach Synonyme. Diese Feststellung ist ungewohnt, jedoch selbstverständlich, wenn man sich vergegenwärtigt, dass Messen das *Kontinuumproblem* lösen, d.h. den oben genannten fundamentalen Widerspruch überbrücken muss.

Üblicherweise werden die Wörter “Messen” und “Digitalisieren” nicht als Synonyme aufgefasst, sondern es wird nur dann von Digitalisieren gesprochen, wenn dies durch ein Gerät, einen sog. *Umsetzer* oder *Konverter* erfolgt, nicht aber, wenn es durch den Menschen erfolgt, wie z.B. beim Ablesen eines analog anzeigenden Messinstruments. Tatsächlich spielt in diesem Fall der Mensch die Rolle eines Analog-digital-Konverters.

Technische Analog-digital-Konverter arbeiten nach dem Schwellwertprinzip. Sie enthalten ein oder mehrere *Schwellenoperatoren.* Bild 4.1 zeigt die Funktionsweise eines solchen Operators. Auf den Eingang wird eine re-

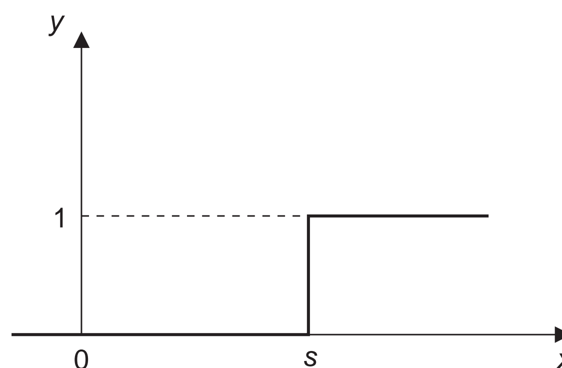


Bild 4.1 Funktionsweise des Schwellenoperators

elle Größe x gegeben. Das ist in der Regel eine Spannung, die von einem geeigneten Messfühler geliefert wird. Die Ausgabegröße y kann zwei Werte annehmen, die mit 0 und 1 bezeichnet sind. Der Wert zeigt an, ob x eine bestimmte *Schwelle s* überschreitet (dann ist $y = 1$) oder nicht ($y = 0$).

Mit Hilfe von zwei Schwellenoperatoren, deren Schwellen die Werte s und $s + \Delta s$ haben, lässt sich feststellen, ob der Wert von x in den Bereich (in den *Spalt*) zwischen s und $s + \Delta s$ fällt. Eine Messung von x mit der Genauigkeit Δs lässt sich nun dadurch bewerkstelligen, dass s, beginnend mit 0, schrittweise um Δs erhöht wird, und dabei gezählt wird, wieviele Schritte getan werden müssen, bis x die untere Schwelle überschreitet, die obere jedoch nicht, also in den momentanen Spalt fällt. Dies ist das Grundprinzip jedes Messens.

Die Umkehrung der Längenmessung, also das Abtragen einer zahlenmäßig gegebenen Länge auf einer Geraden als Strecke, ist eine Überführung einer zahlenmäßigen, also sprachlichen in eine kontinuierliche Darstellung, eine sog. *Digital-analog-Konvertierung*.

3 Die Unmöglichkeit des absolut genauen Messens, m.a.W. der prinzipielle Näherungscharakter des Digitalisierens, ist also nicht einfach die Folge technischer Unzulänglichkeiten, sondern letzten Endes eine prinzipielle Gegebenheit, die ihre Ursache in der diskreten Natur des Denkens und der Sprache hat. Wenn man dagegen im analogen Bereich bleibt, wenn man z.B. *analog rechnet* (siehe weiter unten), ohne zu digitalisieren, dann freilich ist die "Rechenungenauigkeit" allein die Folge technischer Unzulänglichkeiten.

Ähnliches wie für das Messen gilt auch für das Berechnen. *Die absolut genaue Berechnung einer durch Konstruktion festgelegten Länge ist - zumindest in der Mehrzahl der Fälle - nicht möglich.* Diese Unmöglichkeit überrascht wohl niemanden, obwohl sie nicht weniger merkwürdig ist als die prinzipielle Unmöglichkeit des absolut genauen Messens. Man hat sich an sie gewöhnt.

Konstruiert man z.B. über der Zahlengeraden ein gleichschenklig-rechtwinkliges Dreieck, dessen Katheten die Länge 1 besitzen und dessen Hypotenuse auf die Zahlengerade und dessen linke Ecke in den Nullpunkt fällt, so markiert die rechte Ecke einen Punkt, der nicht exakt digitalisierbar, also nicht exakt mit Hilfe von Ziffern angebbar ist. Indirekt ist er allerdings dennoch sprachlich modellierbar, indem man eine Vorschrift für seine Berechnung angibt oder ihm einfach irgendeinen Namen zuweist, z.B. $\sqrt{2}$. Versucht man aber, die Wurzel exakt zu ziehen, stellt man fest, dass dies nicht möglich ist, weil die Prozedur der *numerischen (zahlenmäßigen) Berechnung* niemals abbricht, sondern sich ins Unendliche fortsetzt. Das exakte Ergebnis ist eine Unendliche Folge von Ziffern. Derartige Zahlen heißen *Irrationalzahlen.*

Eine andere Irrationalzahl ergibt sich, wenn man einen Kreis mit dem Durchmesser 1 auf der Zahlengeraden abrollt, beginnend im Nullpunkt. Nach einer vollen Umdrehung des Kreises markiert dieser auf der Zahlengeraden einen Punkt, dem sich wiederum keine Zahl in Form einer endlichen Ziffernfolge zuordnen lässt. Um ihn

dennoch sprachlich exakt angeben zu können, hat man ihm den Namen π zugewiesen. Irrationalzahlen sind im Gegensatz zu den sog. *Rationalzahlen* nicht als Quotient zweier ganzer Zahlen darstellbar. Beide Zahlenklassen werden zur Klasse (oder Menge) der *reellen Zahlen* zusammengefasst.

Man kann nun die scheinbar dumme Frage stellen, wie viele reelle Zahlen es gibt. Natürlich gibt es unendlich viele, denn die Zahlengerade besitzt unendlich viele Punkte, sie ist ein Kontinuum. Nun gibt es aber bereits unendlich viele rationale Zahlen, und man könnte glauben, dass sie ausreichen, um alle Punkte der Zahlengeraden darzustellen (auf sie abzubilden). Das ist jedoch, wie wir gesehen haben, nicht der Fall. Zwischen den Punkten, denen rationale Zahlen entsprechen, liegen unendlich viele weitere Punkte. Tatsächlich ist die Anzahl der reellen Zahlen in "höherem Grade unendlich" als die der rationalen Zahlen. Die Mathematiker drücken sich exakter aus und sagen: *Die Menge der rationalen Zahlen ist **abzählbar unendlich**, die der reellen Zahlen ist **überabzählbar unendlich**.*

Diese Ausdrucksweise hat folgende Ursache. Man kann zeigen, dass sich die rationalen Zahlen in einer lückenlosen und unendlich fortsetzbaren Reihe ordnen und also auch abzählen lassen. Für die reellen Zahlen ist das nicht möglich. Nach welcher Regel man auch versucht, sie zu ordnen, es bleiben immer unendlich viele Lücken erhalten. Hier zeigt sich der Widerspruch zwischen der diskreten Natur des Denkens und der kontinuierlichen Natur des Gedachten in *zahlentheoretischer* Form.

4.2 Analoges Rechnen

Wir wollen nun auch die Zeit in unsere Überlegungen einbeziehen. Das zeitliche Kontinuum ist ebenso wie die Linie eindimensional und lässt sich analog als Zahlengerade darstellen. Zusätzlich ist dabei deren positive Richtung vorgegeben, nämlich als zeitlicher Fortschritt aus der Vergangenheit in die Zukunft.

Die Vorstellung der kontinuierlichen Zeit führt zur Vorstellung der Kontinuität von *Prozessen,* d.h. der Kontinuität der zeitlichen Änderung von Merkmalen realer Objekte, z.B. ihrer Lage (Kontinuität der Bewegung). Es erhebt sich nun die Frage, wie sich die Kontinuität von Prozessen und speziell die Kontinuität der Bewegung sprachlich modellieren lässt. Diese Frage beschäftigte die Menschen seit Jahrtausenden, nachweislich seit dem Streit zwischen zwei philosophischen Schulen des griechischen Altertums, den Herakliten und den Eleaten. Gelöst wurde das Problem erst in der Neuzeit, und zwar durch ISAAC NEWTON, als er versuchte, die Planetenbewegung zu *erklären.* Es ist aufschlussreich, den Weg dahin zu verfolgen, denn er ist charakteristisch dafür, wie die Physiker die Natur sprachlich, und zwar mathematisch modellieren.

Am Anfang jeder Modellbildung steht die Beobachtung und deren Protokoll. Im Falle der newtonschen Himmelsmechanik waren das die Messergebnisse von TYCHO

DE BRAHE. Er hat genau protokolliert, welcher Planet wann und wo zu sehen war. Das Beobachtungsprotokoll eines Planeten hat die Form einer zeitlichen Reihe

$$z_0, z_1, ..., z_{t-1}, z_t, ...,$$

wo z_t den Ort zum Zeitpunkt t_1, also den gemessenen Azimut und Erhebungswinkel, bezeichnet, z stellt also ein Wertepaar dar. Aus solchen Protokollen hat JOHANNES KEPLER seine berühmten Gesetze abgeleitet, aus denen sich die Bewegung jedes Planeten als stetige Funktion der Zeit $z(t)$ berechnen lässt. Die keplerschen Gesetze beschreiben die Planetenbewegung mit erstaunlicher Genauigkeit, aber sie erklären sie nicht. Es sind empirische Gesetze. Ihre Erklärung gelang Newton 70 Jahre später. Er konnte sie aus seinen Prinzipien der Mechanik und seinem Gravitationsprinzip ableiten. Dazu musste er die Differenzialrechnung erfinden.

Newton ging davon aus, dass der Zustand eines dynamischen Systems die kausale Folge des vorangehenden Zustandes ist. Das lässt sich formal durch

$$z_t = f(z_{t-1}) \text{ bzw. } z_t = f(z_{t-1}, x_{t-1})$$

ausdrücken. Die erste Gleichung gilt für abgeschlossene oder *autonome*, die zweite für nicht abgeschlossene oder *nichtautonome* Systeme, wobei x die Einwirkung der Umwelt bezeichnet. Wenn f eine bekannte Funktion ist, lässt sich z für alle möglichen (diskreten) Zeitpunkte berechnen. Beispielsweise lässt sich die Bewegung des Sekundenzeigers einer Uhr, der in jeder Sekunde einen Sprung macht, durch die Gleichung $z(t) = z(t-1) + 6$ beschreiben, wobei z die Zeigerstellung in Grad ist und der Winkel im Uhrzeigersinn gemessen wird.

4 Newton gab sich mit einer solchen diskreten, genauer *kausaldiskreten* Beschreibung nicht zufrieden, denn sie gibt den Ursache-Wirkungs-Zusammenhang nur in Sprüngen, also unvollständig wieder. Er war überzeugt, dass nicht nur Raum und Zeit, sondern auch die Kausalität, der Ursache-Wirkungs-Zusammenhang kontinuierlicher Natur ist, dass also der jeweilige Zustand eines abgeschlossenen Systems - letztendlich des Weltalls - die Folge des "unendlich kurz" vorangehenden Zustandes ist. Er suchte nach einer *kausalkontinuierlichen* Beschreibung. Das stieß auf eine grundsätzliche Schwierigkeit. Wenn man das Zeitintervall Δt immer kleiner und schließlich zu *0* werden lässt, dann wird, wie die Erfahrung zeigt, auch die Zustandsänderung Δz zu *0* (die Natur macht keine Sprünge, "natura non facit saltus"), und die obige kausaldiskrete Beschreibung verliert ihren Sinn. Newton fand folgenden Ausweg.

Bekanntlich hat der Quotient 0/0 i.Allg. keinen bestimmten Wert, d.h. er kann jeden beliebigen Wert annehmen. Das gilt auch für $\Delta z/\Delta t$, wenn beide Intervalle gleich 0 sind. Wenn aber z eine *glatte* Funktion der Zeit ist, also eine Funktion ohne Sprünge und Ecken, dann strebt der *Differenzenquotient* $\Delta z/\Delta t$ mit verschwindendem Δt einem bestimmten *Grenzwert* zu, dem sog. *Differenzialquotienten*, der mit $\mathrm{d}z/\mathrm{d}t$ bezeichnet wird. Beispielsweise sei $z = 3t$. Dann gilt stets $\Delta z/\Delta t = 3$, unabhängig von

der Größe des Zeitintervalls Δt, und auch im Grenzfall $\Delta t \to 0$ gilt $dz/dt = 3$. Geometrisch bedeutet dies, dass die Funktion $z(t)$ eine Gerade mit der *Steigung* 3 beschreibt. Physikalisch wird dadurch eine Bewegung mit der konstanten *Geschwindigkeit* 3 beschrieben, z.B. mit der Geschwindigkeit 3 m/s, falls z in Metern und t in Sekunden gemessen wurde. Wenn mit Größen gerechnet wird, deren Wert in dem beschriebenen Sinne “gegen den Grenzwert Null streben”, spricht man von *Infinitesimalrechnung*.

Der Differenzialquotient dz/dt wird auch als *Ableitung der Funktion* $z(t)$ *nach der Zeit* bezeichnet. Seine Berechnung wird *Differenzieren* oder *Ableiten* genannt. Die Ableitung einer Funktion ist wieder eine Funktion von der gleichen Variablen.

Die Erfindung des Differenzierens ermöglicht eine kausalkontinuierliche Prozessbeschreibung, indem nicht der Modellparameter z als kausale Folge des vorangehenden Zustandes aufgefasst wird, sondern seine zeitliche Ableitung dz/dt als kausale Folge von z, formal notiert:

$$dz/dt = f(z) \quad \text{bzw.} \quad dz/dt = f(z,x) \tag{4.1}$$

Derartige Gleichungen heißen *Differenzialgleichungen*. Die erste Gleichung gilt wieder für autonome, die zweite für nichtautonome Systeme. Ebenso wie z ist auch x eine Funktion der Zeit. Wenn die Funktion f bekannt ist, lässt sich der zeitliche Verlauf des Zustandes $z(t)$ aus der Differanzialgleichung berechnen, oft allerdings nur näherungsweise. Für die kausalkontinuierliche Beschreibung der Planetenbewegung leitete Newton Differenzialgleichungen ab, deren Lösung auf die keplerschen Gesetze führt. Die Gleichungen sind allerdings komplizierter als (4.1) und enthalten nicht nur die erste Ableitung (s.u.). Damit begann eine neue Epoche der Physik. Die Differenzialgleichung wurde zum wichtigsten Arbeitsmittel der theoretischen Physiker.

Es sei noch einmal unterstrichen, dass die Prozessbeschreibung mittels Differenzialgleichungen eine Modellierung des Kontinuums mit sprachlichen, also nichtkontinuierlichen (diskreten) Mitteln darstellt. Erst bedeutend später erfand man eine systematische und allgemein verwendbare Methode, den kausalkontinuierlich gedachten Ablauf von Prozessen auch kausalkontinuierlich, also *nicht*sprachlich zu *modellieren*. Der Methode liegt folgende Idee zugrunde.

Wenn zwei verschiedene Prozesse durch formal identische Differenzialgleichungen beschrieben werden, so verlaufen sie offensichtlich “analog” (die Anführungsstriche zeigen an, dass das Wort nicht in der bisherigen, sondern in seiner umgangssprachlichen Bedeutung verwendet wurde) d.h. die Prozessparameter werden durch (im Wesentlichen) gleiche Zeitfunktionen beschrieben. Dabei können die analogen Prozesse ganz unterschiedlicher Natur sein. So lassen sich beispielsweise Luft-, Licht- und Wasserwellen bei geeigneter Idealisierung [5.12] durch ein und dieselbe Differenzialgleichung, die sog. *Wellengleichung*, beschreiben. Es ist also möglich, einen dieser Prozesse durch einen anderen zu modellieren. Auf dieser Tatsache fußt

die Idee des *Analogrechners*. Der erfindungsreiche Weg zur Verwirklichung dieser Idee soll in großen Zügen skizziert werden.

Wir gehen von dem obigen Beispiel der *geometrisch-analogen Konstruktion* der Größe $\sqrt{2}$ aus. Derartige analoge Konstruktionsmethoden lassen sich auch für andere numerische Rechenoperationen angeben. So kann die Summe $a+b$ durch Aneinanderlegen zweier Strecken der Länge a und b bestimmt werden. Das Produkt $a \times b$ lässt sich analog als Fläche eines Rechtecks mit den Seitenlängen a und b konstruieren. Um das Resultat zu digitalisieren, kann man den Flächeninhalt z.B. mit Hilfe eines Planimeters bestimmen. Man kann auch das Rechteck ausschneiden und sein Gewicht und das Gewicht der Flächeneinheit bestimmen. Ferner besteht die Möglichkeit, das Multiplizieren auf das Aneinanderlegen von Strecken zurückzuführen, indem man als Maßzahlen für die Längen nicht die Zahlenwerte der Faktoren, sondern deren Logarithmen verwendet. Nach diesem Prinzip arbeitet der Rechenschieber.

Wenn man nun zu jeder arithmetischen Operation eine geometrisch-konstruktive Variante realisiert, sozusagen ein "*analoges Analogon*", dann lässt sich jede numerische Rechnung, also jedes Rechnen mit Zahlen auch konstruktiv, also kontinuierlich (genauer: räumlich kontinuierlich) ausführen. Man spricht dann von "*analogem Rechnen*". Die Wortverbindung "analoges Analogon" ist kein Pleonasmus (wie z.B. "runder Kreis"), sie widerspiegelt vielmehr den Bedeutungswandel des Wortes "analog", von dem bereits die Rede war. Ursprünglich bezeichnete es die Entsprechung, die Ähnlichkeitsbeziehung, präziser die *Homomorphie* zwischen diskontinuierlich-zahlenmäßigem und kontinuierlich-konstruktivem "Rechnen", allgemeiner kann man auch sagen: zwischen exaktem (formalisiertem) sprachlichem und exaktem (metrischem) nichtsprachlichem Modellieren. Später bürgerte es sich ein, das kontinuierliche Rechnen selber als - im übertragenen Sinne - *analog* zu bezeichnen und auch die Operanden und Resultate des analogen Rechnens *analoge Größen* zu nennen. In der obigen Wortverbindung "analoges Analogon" ist das erste Wort im übertragenen, das zweite im ursprünglichen Sinne zu verstehen.

Vom logischen Standpunkt aus ist ein solcher Sprachgebrauch irritierend. Aber Sprachen entwickeln sich eben nicht nach logischen Gesichtspunkten, selbst die Sprache der Techniker und sogar der Mathematiker nicht unbedingt. Noch irritierender wird die Terminologie, wenn das zeitliche Kontinuum von dieser Sprachregelung ausgenommen wird, wie es häufig der Fall ist, wenn also hinsichtlich der Zeit von kontinuierlichen, sonst aber von analogen Größen gesprochen wird.

Der nächste Schritt in Richtung Analogrechner besteht im Übergang vom geometrischen zum physikalischen, speziell zum elektronischen analogen Rechnen. Um das physikalische Analogon einer arithmetischen Operation zu realisieren, muss man einen physikalischen, z.B. elektrischen Effekt finden, der einer Gesetzmäßigkeit gehorcht, die sich mathematisch durch die betreffende Operation ausdrückt. Wenn beispielsweise zwei Spannungsquellen in Reihe geschaltet werden, ergibt sich die Summe der Einzelspannungen. Entsprechend kann man Differenzen bilden. Tatsäch-

lich lassen sich sämtliche arithmetische Operationen analog-elektronisch realisieren und die Resultate "*analog berechnen*" (generieren), allerdings nur mit einer bestimmten Genauigkeit, die von den technischen Parametern der Schaltung abhängt und in der Regel mit Mühe 0,001% erreicht.

Der letzte Schritt zum Analogrechner besteht in der elektronischen Realisierung des *Integrierens*. Mit Integrieren wird die Umkehrung des Differenzierens bezeichnet, also das Bestimmen einer Funktion aus ihrer Ableitung. Das Ergebnis heißt *Integral*. Geometrisch kann das Integrieren als Flächenbestimmung aufgefasst werden. Sein geometrisch-konstruktives Analogon ist also das Planimetrieren. Das elektronische Analogon lässt sich mit Hilfe eines Kondensators realisieren. Die Geschwindigkeit, mit der sich die Spannung an einem Kondensator ändert, also die Ableitung der Spannung nach der Zeit, ist nämlich proportional dem elektrischen Strom, der durch den Kondensator fließt, m.a.W. $\mathrm{d}u/\mathrm{d}t$ ist proportional dem Ladungsfluss i, der den Kondensator auf- bzw. entlädt. Folglich ist die Spannung proportional dem Integral des Stromes, formal notiert:

$$u(t) = \mathrm{const} \int i(t)\, \mathrm{d}t$$

Dabei ist angenommen, dass im Zeitpunkt t=0 auch u=0 gilt. Mit const ist eine Proportionalitätskonstante bezeichnet; ihr Kehrwert heißt *Kapazität* des Kondensators. Es sind elektronische Integratoren hoher Genauigkeit entwickelt worden, deren Kernstück ein Kondensator ist.

Ein ***Analogrechner*** *ist in seiner ursprünglichen Form ein Baukasten elektronischer Standardschaltungen zur analogen Berechnung derjenigen Operationen, die in Gleichungen (z.B. algebraischen Gleichungen oder Differenzialgleichungen) auftreten.* Für seine Verwendung zur *analogen Prozessmodellierung* ergibt sich aus den vorangehenden Überlegungen folgende Vorgehensweise. Zunächst versucht man, die zeitliche Änderung des zu modellierenden Prozessparameters mittels einer Differenzialgleichung zu beschreiben. Wenn das gelungen ist, verbindet man die Elemente des elektronischen Baukastens (des Analogrechners) so miteinander, wie die Differenzialgleichung es vorschreibt. Dann stellt der so "programmierte" (d.h. verschaltete) Analogrechner ein analoges Prozessmodell dar, d.h. der zeitliche Verlauf seiner Ausgangsspannung ist identisch mit dem des modellierten Prozessparameters.

Die *Programmierung* eines Analogrechners - diese Redeweise ist irritierend, hat sich aber eingebürgert - entspricht dem Pfeil 6 in Bild 2.1, also dem Übergang von einem externen sprachlichen zu einem externen nichtsprachlichen Objekt, speziell in unserem Falle von einem formalisierten sprachlichen zu einem metrischen nichtsprachlichen Modell. Die Programmierung soll an zwei Beispielen demonstriert werden, zunächst anhand der rechten Differenzialgleichungen (4.1)

5

$$\mathrm{d}z/\mathrm{d}t = f(z(t), x(t)),$$

wobei die Abhängigkeit der Variablen z und x von der Zeit explizit angegeben ist. Integration beider Seiten ergibt (von einer additiven Konstanten abgesehen)

$$z(t) = \int f(z(t), x(t))\, \mathrm{d}t.$$

Wir wollen annehmen, dass die Funktion $f(t) = z(t) + x(t)$ ist. Bild 4.2a zeigt die Analogrechnerschaltung (einen programmierten Analogrechner), der die Differenzialgleichung *löst,* d.h. der am Ausgang $z(t)$ liefert. Das Programmieren besteht darin, dass zwei analoge Operatoren, ein Summator und ein Integrator zu einem Ring (einer Rückkopplungsschleife) miteinander verbunden und Ein- und Ausgabeleitungen angekoppelt werden, so wie es die Differenzialgleichung vorschreibt. Dabei entspricht dem Gleichheitszeichen das Schließen der Rückkopplungsschleife vom Integrator zum Summator. Die Schließung hat zur Folge, dass der Ausgabewert des Summators, also $z+x$, stets gleich ist dem Eingabewert des Integrators, also $\mathrm{d}z/\mathrm{d}t$, wie die Differenzialgleichung es verlangt. Auf Probleme der Dimensionierung soll nicht eingegangen werden.

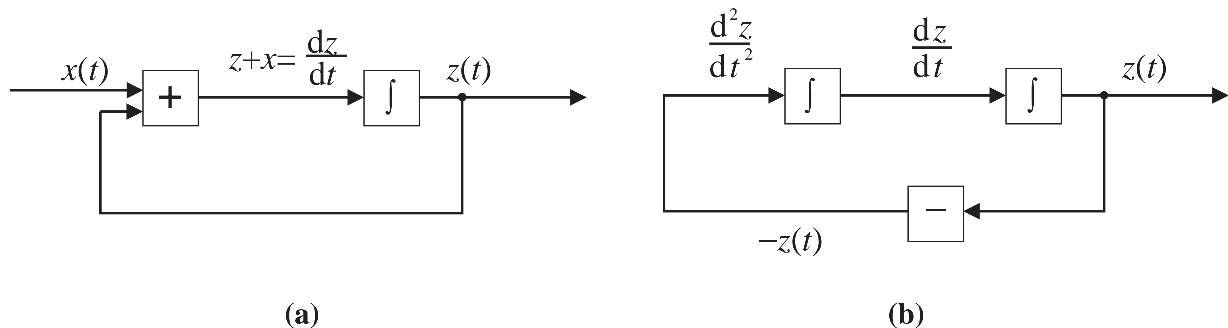


Bild 4.2 Analogrechnerschaltung zur Lösung zweier einfacher Differenzialgleichungen

Bild 4.2b zeigt ein zweites Beispiel. Diese Schaltung löst die denkbar einfachste *Differenzialgleichung zweiter Ordnung*. So wird eine Differenzialgleichung genannt, welche die *zweite Ableitung*, das ist die Ableitung der Ableitung, enthält (aber keine höhere Ableitung). Anstelle von $\mathrm{d}(\mathrm{d}z/\mathrm{d}t)/\mathrm{d}t$ (Ableitung der Ableitung) schreibt man $\mathrm{d}^2z/\mathrm{d}t^2$. Zur Lösung der Gleichung sind zwei Integratoren erforderlich. Die dargestellte Schaltung enthält einen weiteren Operator, der den negativen Wert des Eingabewertes liefert (Multiplikation mit −1). Die Schaltung löst die Gleichung

$$\mathrm{d}^2z/\mathrm{d}t^2 = -z. \tag{4.2}$$

Das ist die Wellengleichung in ihrer einfachsten Form. Die Lösung lautet (von Konstanten abgesehen) $z = \sin(t)$, denn die zweite Ableitung der Sinusfunktion ist die Sinusfunktion mit negativem Vorzeichen. Das ist eine periodische Schwingung mit der Frequenz 1 pro Zeiteinheit (von der Dimensionierung sehen wir wieder ab).

Die Schaltung in Bild 4.2b besitzt keinen Eingang, sie stellt ein autonomes System dar, sie schwingt “von selbst”, sie *generiert* Sinusschwingungen. Sie kann also als

sogenannter *Sinusgenerator* eingesetzt werden. Die Leitungsführung auch dieser Schaltung läuft in sich selbst zurück, sie ist zu einem “Kreis” oder “Zirkel” kurzgeschlossen, der Input ist gleich dem Output. Darin spiegelt sich die für die newtonschen Bewegungsgleichungen charakteristische Relativität von Ursache und Wirkung wider. In ihnen kann man die Kraft als Ursache und die Beschleunigung als Wirkung auffassen, mit demselben Recht aber auch die Beschleunigung als Ursache und die Kraft (Trägheitskraft) als Wirkung. In diesem Sinne sprechen wir von *Ursache-Wirkung-Zirkularität*. In Kap.6.3 [6.4] kommen wir darauf zurück.

6

Bei diesen wenigen Bemerkungen zur Analogrechentechnik wollen wir es bewenden lassen. Viele wichtige Details sind nicht zur Sprache gekommen, u.a. die Eingabe von Anfangswerten, die Vorzeichenumkehr, die Dimensionierung und Normierung.[1]

Die Analogrechentechnik hat bei der Entwicklung der modernen *Regelungstechnik* eine bedeutende Rolle gespielt. Ein starker Impuls ging von Norbert Wiener aus, von seiner Idee einheitlicher Prinzipien der Regelung und Informationsübertragung in Lebewesen und in Maschinen [Wiener 63]. Er stellte das Regelungsproblem, genauer das Problem der Selbststeuerung und Selbststabilisierung von Prozessen durch Rückkopplung, in einen neuen, fast universell anmutenden Zusammenhang. Die Popularität seiner Bücher und seiner Wortschöpfung “*Kybernetik*” löste einen regelrechten Boom aus.

Inzwischen sind die wienerschen Ideen von allerhand nachträglichem Beiwerk wieder befreit worden, um das andere Autoren sie “bereichert” hatten. Das Wertvolle ist geblieben, aber leider auch die Gewohnheit, das, was in einen Analogrechner eingegeben wird und was er ausgibt, als *Information* und den Analogrechner selbst als informationelles System zu bezeichnen. Das ist eine schlechte, weil irreführende Gewohnheit. Denn mit dem gleichen Recht könnte man beispielsweise auch Ein- und Ausgangsspannungen eines Transformators als Information bezeichnen, ja, sogar den Durchmesser eines Werkstücks, das auf einer Drehbank bearbeitet wird. Schließlich könnte jede kausale Folge als Information über die Ursache aufgefasst werden. Derartige Vorschläge gibt es. Aber sie führen zu inhaltlosen Diskussionen über den Begriff der Information, der bei einer derartigen Verallgemeinerung seinen eigentlichen Inhalt verliert, sodass er überflüssig wird. Seine eigenständige Bedeutung erhält der Informationsbegriff erst in Verbindung mit Zeichen, mit Codieren.

Wenn wir die Analogrechentechnik relativ ausführlich behandelt haben, obwohl sie nach unseren Begriffsbestimmungen nicht zur Informatik gehört, so nicht nur, um die Einordnung des Analogrechners in die Modellklassifikation gemäß Bild 3.1 zu begründen und um das Verständnis des Unterschiedes zwischen analogem und digitalem Modellieren zu vertiefen, sondern auch, weil in vielen Bereichen der Praxis der Analogrechner mit zum Instrumentarium des Informatikers gehört. Damit hat es folgende Bewandtnis.

---

1 Eine ausführliche Darstellung findet der Leser z. B. in [Schwarz 71]

Wenn einem Analogrechner ein Digital-analog -Konverter vorgeschaltet und ein Analog-digital-Konverter nachgeschaltet wird, ergibt sich ein Gerät, das für den, der es als *schwarzen Kasten* betrachtet und sich für sein Innenleben nicht interessiert, als informationelles System darstellt. Ein Analogrechner lässt sich in jedes echte, d.h. durchgängig digital funktionierende informationelle System einbauen. Eine solche Vorgehensweise kann von Vorteil sein, wenn eine Berechnung mit vorgegebener Rechengenauigkeit auf digitalem Wege aufwendiger ist, als auf analogem. Das ist häufig bei der Lösung von Differenzialgleichungen der Fall.

7

Weit öfter ist in der Praxis das umgekehrte Vorgehen anzutreffen. Wenn einem Digitalrechner ein Analog-digital-Konverter vorgeschaltet und ein Digital-analog-Konverter nachgeschaltet wird, ergibt sich ein Gerät, das als schwarzer Kasten betrachtet einen Analogrechner darstellt. Man kann es z.B. in einen Regelkreis einbauen, wo es die Rolle des früher verwendeten echten Analogrechners übernimmt. Auf diese Weise lässt sich eine bedeutend flexiblere und auch präzisere Regelung erreichen. Das hat zu einer weiteren sprunghaften Entwicklung der Regelungstechnik geführt.

Abschließend sei eine Bemerkung gemacht, die für jeden, der mit Computern zu tun hat, eine Selbstverständlichkeit ist. Das Modellieren mit dem Digitalrechner muss aus der Sicht des Nutzers kein zahlenmäßiges Modellieren sein, kein numerisches Rechnen, was das Wort “Digitalrechner” eigentlich erwarten ließe. Es braucht überhaupt kein mathematisches Modellieren zu sein, sondern es kann ein Modellieren in irgendeiner Sprache, vielleicht in einer geeigneten Fachsprache sein, möglicherweise sogar in der natürlichen Umgangssprache. Ganz allgemein kann es ein Modellieren mit Hilfe irgendwelcher Symbole sein. Aus dieser Sicht wäre die Bezeichnung *Symbolrechner* treffender. Wenn dennoch von “Digitalrechner” gesprochen wird, so verbindet sich damit ein verdeckter Bedeutungswandel des Wortes “digital” von “zahlenmäßig” zu “zeichenmäßig” und schließlich zu “sprachlich”. Freilich ist dieser Bedeutungswandel nicht so tiefgehend wie der des Wortes “analog” von “entsprechend” über “kontinuierlich” zu “nichtsprachlich”.

Um schließlich noch einmal auf die Überschrift dieses Kapitels “Analoges und digitales Modellieren” zurückzukommen, so hätte sie, bei Verwendung der in Kap.3.1 benutzten Bezeichnungen für die verschiedenen Modellklassen auch “Nichtsprachliches und sprachliches Modellieren” lauten können.

5 Syntax und Semantik

Sprache ist das Vermögen, von endlichen Begriffen einen unendlichen Gebrauch zu machen
Wilhelm von Humboldt

Zusammenfassung

Sprachliches Modellieren beginnt mit der Unterscheidung von Gegenständen auf Grund unterschiedlicher Merkmale und mit der Herausbildung entsprechender *Denkobjekte* (Ideme). Ein *sprachliches Modell* ist eine Menge von Aussagen über das Original. *Aussagen* ordnen Objekten Merkmale zu. Die Ausdrucksstärke und Flexibilität der menschlichen Sprache, die erforderlich ist, um die Welt modellieren zu können, wird durch Begriffsbildung und syntaktische Regeln erreicht. Begriffe sind benannte Denkobjekte. Neue Begriffe können aus alten durch komponierende, idealisierende, klassifizierende oder generalisierende Abstraktion gebildet werden (siehe Bild 5.4).

Syntaktische Regeln, kurz *Syntaxregeln*, dienen der Reduzierung des Codierungsaufwandes. Um nicht für jedes Idem (jeden Bewusstseinsinhalt, jedes Denkobjekt) ein besonderes Zeichen ("Hieroglyph") für seine Artikulierung benutzen zu müssen, werden aus einer relativ kleinen Anzahl elementarer Zeichenrealeme (Buchstaben, Morpheme, Wörter) nach bestimmten Regeln Kompositzeichen gebildet, sodass ein Empfänger (Interpretierer) jedem nach den Syntaxregeln komponierten Zeichenrealem (Wort, Satz) mehr oder weniger genau dasjenige Idem zuordnen kann, welches der Sender (Artikulierer) ihm zugeordnet hat.

Wenn davon ausgegangen wird, dass alle Beteiligten (z.B. alle Mitlieder einer Sprachgemeinschaft oder einer Diskussionsrunde) ein Zeichenrealem ausreichend einheitlich interpretieren, sodass eine Verständigung möglich ist, wird das zugeordnete Idem die *Semantik* des Zeichenrealems genannt. Das Wort *Bedeutung* wird umgangssprachlich sowohl im Sinne von Idem als auch im Sinne von Semantik verwendet. Wir folgen dem umgangssprachlichen Gebrauch, wenn aus dem Kontext hervorgeht, in welchem Sinne das Wort benutzt wird. Exakte Bedeutungsgleichheit kann durch Formalisierung erreicht werden. Dazu muss der Diskursbereich, d.i. der Originalbereich, der sprachlich modelliert wird, über den man sich unterhält, *kalkülisiert* werden, d.h. es muss ein "passender" Kalkül ge- oder erfunden werden, sodass sich der Originalbereich als Interpretation des Kalküls auffassen lässt. Eine solche Interpretation heißt *extern* (bezüglich des Kalküls).

Ein Kalkül ist durch eine formale (z.B. mathematische) Sprache und durch Regeln festgelegt, nach denen sich Ausdrücke der Sprache (d.h. Ausdrücke, die nach den Syntyxregeln der Sprache gebildet sind) in andere Ausdrücke überführen lassen. Die Bedeutung, die ein Zeichenrealem im Rahmen eines Kalküls besitzt (d.h. ohne externe Interpretation), heißt *formale Semantik*. Die Schwierigkeit des Kalkülisierens

liegt - abgesehen vom Finden oder Erfinden eines geeigneten Kalküls - in der Anbindung der *externen Semantik* (der Bedeutung von Zeichenrealemen hinsichtlich des Originalbereiches) an die formale Semantik.

Die Schwierigkeit des Modellierens durch den Computer (des Simulierens) liegt in der Anbindung der externen Semantik (*Nutzersemantik*, also der Semantik, “in der” ein Computernutzer den Originalbereich gedanklich modelliert) an die *interne Semantik* bezüglich des modellierenden Computers (*Computersemantik*). Die *trägerinterne* oder kurz *interne Semantik* eines Zeichenrealems ist die physische Wirkung, die das Zeichenrealem im Interpretierer (im Träger des Interpretationsprozesses) auslöst. Die trägerinterne Semantik bezüglich eines Menschen ist die Gesamtheit der neurophysiologischen Prozesse, die das betreffende Zeichenrealem im Gehirn auslöst. Die Computersemantik ist die Gesamtheit der elektronischen Prozesse, die das Zeichenrealem im Computer auslöst.

Die Informations- und Codierungstheorie vergleicht den Aufwand von Codierungen mittels Zeichenketten (also Zeichenkettenlängen) in verschiedenen Codierungssystemen (in verschiedenen Sprachen) ohne Berücksichtigung der Semantik. Es wird nicht die Codierung von Idemen durch Zeichenrealeme, sondern die *Um*codierung einer gegebenen Codierung in eine andere, speziell in die binäre Codierung mittels Bitketten untersucht. Mit Hilfe der Theorie lässt sich der Codierungsaufwand durch Umcodierung minimieren.

Das Verhältnis der Länge einer durch Umcodierung erhaltenen Bitkette zur Länge der ursprünglichen Zeichenkette (zur Länge der “*Nachricht*”) heißt **Informationsgehalt** der Nachricht pro Zeichen. Bei unbeschränkter Verlängerung der Nachricht geht der analytische Ausdruck für den Informationsgehalt in die Formel für die Entropie eines thermodynamischen Systems über (von einem konstanten Faktor abgesehen). Dabei entsprechen die Wahrscheinlichkeiten in der Entropieformel den Auftrittswahrscheinlichkeiten der Alphabetzeichen in der (unendlich langen) Nachricht, d.h. in der Sprache, in der die Nachricht artikuliert ist. Der physikalische Entropiebegriff wird bei der Betrachtung der sprachlichen Modellierung auf subsymbolischer Ebene bedeutungsvoll.

5.1 Sprache und Evolution

Nach dem Einschub über die Analogrechentechnik setzen wir den unterbrochenen Gedankengang zur *sprachlichen* Modellierung fort. Kapitel 2 begann mit der Feststellung, dass Sprache der Kooperation dient und die Überlebenschancen einer Population erhöht. Diese steigen, je besser die Kommunikation innerhalb der Population funktioniert, d.h. je *schneller, umfassender und unmissverständlicher* sich die Individuen der Population untereinander verständigen können. Es ist also zu erwarten, dass die Herausbildung natürlicher Sprachen einem Selektionsdruck in Richtung dieser drei Eigenschaften unterliegt. Außerdem ist eine ständige Spracherweiterung

zu erwarten. Denn mit zunehmendem Wissen über die Welt erweitert sich die Sprache, mit deren Hilfe die Welt modelliert wird.

Die Entwicklung und Erweiterung natürlicher Sprachen ist ein Spezialfall der allgemeineren Gesetzmäßigkeit, dass sich mit forschreitender Evolution auch der Code (die Sprache) weiterentwickelt, mit dessen Hilfe der Nachfolger über das bisherige Evolutionsergebnis informiert wird. Das gilt für alle Arten der Evolution. Der genetische Code wird von den niederen Tieren bis zu den Primaten umfangreicher (wenn auch in erstaunlich geringem Maße); der Sprachumfang eines Menschen nimmt mit seiner geistigen Entwicklung zu; und die Sprachen der Völker wachsen mit ihrer Kultur, mit dem Wachsen der Welt 3, um mit Popper zu sprechen.

Nach dem Gesagten muss eine natürliche Sprache widersprüchlichen Forderungen genügen. Einerseits muss sie beliebig erweiterbar sein, und andrerseits muss ihr Umfang endlich bleiben, und die Artikulationen müssen räumlich und zeitlich endlich und sogar möglichst kompakt und schnell interpretierbar sein. Aber trotz der Endlichkeit der Sprache muss die Welt mit ihrer unendlichen Erscheinungsvielfalt ausdrückbar sein.

Die "Aufgabe", Sprachen zu entwickeln, die den genannten Forderungen gerecht werden, hat die Evolution elegant "gelöst". Das Ergebnis lässt sich mit drei Schlagwörtern charakterisieren:

- Begriffsbildung,
- Grammatik,
- Idemobjektivierung.

Zur Begriffsbildung. Die Erweiterung einer Sprache beinhaltet im wesentlichen die Erweiterung ihrer Lexik durch Bildung neuer *Begriffe*, d.h. neuer *benannter Ideme*. Auf diese Weise wird der modellierbare Originalbereich erweitert und gleichzeitig die Ausdrucksstärke der Sprache erhöht, wodurch Artikulierungen kürzer und die sprachlichen Modelle kompakter werden. Viele Denkinhalte (Ideme), sogar sehr komplexe, lassen sich durch Wörter (durch kurze Zeichenrealeme) benennen (codieren). Wörter wie "Ich", "Tier" und "All" mögen das illustrieren. Begriffsbildung macht das sprachliche Modellieren der "unendlichen Welt" möglich und dient der Erhöhung der Kompaktheit sprachlicher Ausdrücke und damit der Beschleunigung der Kommunikation, denn je kürzer eine Mitteilung ist, umso weniger Zeit erfordert in der Regel ihre Artikulierung und Interpretierung.

Begriffsbildung dient außerdem, wie das Codieren überhaupt, der Anpassung an die physiologischen Gegebenheiten, insbesondere der Anpassung des Umfanges einer Mitteilung an die begrenzte Kapazität des Kurzzeitgedächtnisses, das beim Artikulieren und Interpretieren benötigt wird.

Zur Grammatik. Unter der *Grammatik* einer Sprache, wie sie in der Schule gelehrt wird, versteht man die *Regeln, nach denen Wörter gebildet und geformt, z.B. dekliniert oder konjugiert werden (**Morphologie**), sowie die Regeln, nach denen Sätze gebildet werden (**Syntax**)*. Diese Regeln gewährleisten die Flexibilität, die notwendig ist, um mit einem begrenzten Wortschatz praktisch alle denkbaren Ideme

artikulieren zu können. Außerdem unterstützen sie das Interpretieren insofern, als die Bedeutung einer Aussage aus ihrem Aufbau *hergeleitet* werden kann. Das kommt dem Nutzer einer Sprache jedoch nur dann zum Bewusstsein, wenn er die Sprache nicht perfekt beherrscht. In der technischen Informationsverarbeitung (Computer-IV) spielt das Ableiten der Bedeutung von Befehlen und Programmen aus ihrer syntaktischen Struktur eine besondere Rolle.

Zur Idemobjektivierung. Die dritte der eingangs genannten Forderungen betrifft die Unmissverständlichkeit sprachlicher Ausdrücke. In Kap.2 hatten wir festgestellt, dass Missverständnisse dann auftreten können, wenn sich die Ideme, die der Sender und der Empfänger eines Zeichenrealems diesem zuordnen, voneinander unterscheiden. Völlig identisch sind sie sicher nie, doch muss eine für die Kooperation notwendige Übereinstimmung gewährleistet sein. Diese wird durch *Idemobjektivierung* erreicht. *Unter* ***Idemobjektivierung*** *verstehen wir die Vereinheitlichung der Bedeutungen, die verschiedene Menschen in ein und dieselbe sprachliche Äußerung hineinlegen, m.a.W. die Annäherung aller subjektiven Ideme an ein "objektives" Idem.* Idemobjektivierung findet ständig im Prozess der Kommunikation statt. Sowohl die Denkinhalte (Ideme) der Kooperationspartner als auch die Artikulationen der Denkinhalte passen sich aneinander an. Idemobjektivierung ist Voraussetzung und Bestandteil der kulturellen Evolution. Das gilt zunächst für die gesprochene und in der Folge auch für die geschriebene Sprache.

Zwischen nicht miteinander kommunizierenden Kulturkreisen kann keine sprachliche Anpassung stattfinden. Dennoch ähneln sich die Prinzipien der Bildung von Begriffen, Wörtern und Sätzen auch in nicht verwandten Sprachen stärker, als man angesichts der objektiv gegebenen Möglichkeiten annehmen könnte. Theoretisch wäre es beispielsweise denkbar, jedem Idem sein eigenes elementares Realem (Alphabetzeichen) zuzuordnen. Die chinesische Begriffsschrift geht ziemlich weit in dieser Richtung. Die konsequente Durchführung dieser Methode würde aber am begrenzten menschlichen Gedächtnis scheitern. Die relative Ähnlichkeit der Sprachen der Welt wird verständlich, wenn man bedenkt, dass sich alle Sprachen unter den gleichen biologischen Nebenbedingungen entwickelt haben, insofern als alle Menschen über praktisch die gleichen Hör- und Sprechwerkzeuge und über die gleichen bzw. sehr ähnlichen neuronalen sprachverarbeitenden Strukturen verfügen. Offenbar hat der Selektionsdruck die Sprachen, begonnen mit den elementaren Lautzeichen, den *Phonemen*, optimal an die Anatomie der Menschen und an die jeweiligen stabilen Umweltbedingungen angepasst.

Unsere Überlegungen führen zu dem Schluss, dass Sprachen Resultate sowohl der genetischen als auch der kulturellen Evolution sind. Wenn erworbene Eigenschaften vererbt werden können, besteht sogar die Möglichkeit einer Rückwirkung der intellektuellen und kulturellen auf die genetische Evolution. Das Ergebnis könnte beispielsweise die Entwicklung eines angeborenen Interpretationsapparates sein. Dieser scheint tatsächlich zu existieren, denn anders lässt sich die Schnelligkeit kaum

erklären, mit der ein Kind seine Muttersprache erlernt. Noam Chomsky hat diese Frage ausführlich diskutiert [Chomsky 70].

5.2 Hierarchische Komponierung

Bei der “Lösung des Sprachproblems” hat die Natur sich eines ihrer “Lieblings-*tricks*” bedient, der Methode des *hierarchischen Komponierens*. (Der Leser gestatte die teleonomische Deutung der Evolution als zielstrebigen Prozess.) ***Hierarchisches Komponieren*** *besteht ganz allgemein darin, dass aus Bausteinen neue Objekte aufgebaut - wir sagen komponiert - werden, sog.* ***Komposite****. Diese können ihrerseits als Bausteine verwendet werden, sodass eine Hierarchie von Kompositen entsteht.* Nach diesem Prinzip ist die tote wie die lebende Materie aufgebaut, nach ihm organisieren sich menschliche Gesellschaften, und nach ihm haben sich auch die natürlichen Sprachen entwickelt. Bild 5.1 illustriert die hierarchische Komponierung am Beispiel eines Satzes der deutschen Schriftsprache. Dabei gilt das Alphabetprinzip: *Die Kompositzeichen einer Sprache werden aus Elementarzeichen, den* ***Alphabetzeichen****, hierarchisch komponiert. Die Anzahl der Alphabetzeichen und der Kompositzeichen ist endlich.* Das Komponieren erfolgt nach Regeln der jeweiligen Sprache.

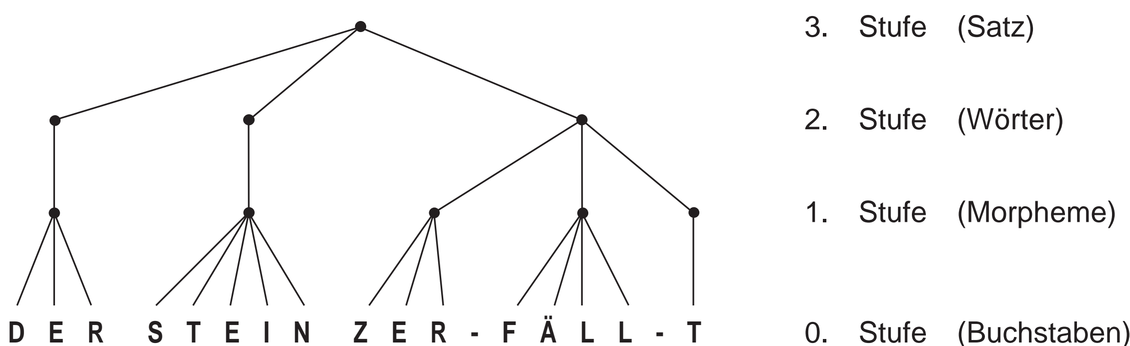


Bild 5.1 Hierarchische Zeichenkomponierung eines Satzes

Es liegt nahe, im hierarchischen Komponierungsprinzip, nach dem die Welt aufgebaut ist, ein ökonomisches Vorgehen der Natur zu sehen, in Analogie zum Vorgehen eines Ingenieurs, der seine Konstruktion aus Normteilen zusammensetzt, um die Herstellung und Wartung des Produktes kostengünstiger zu gestalten. In Wirklichkeit ist die scheinbar beabsichtigte Ökonomie ein Resultat der Evolution. Diese kann bei ihren Versuchen immer nur in sehr kleinen (z.B. kombinatorischen oder komponierenden) Schritten über das bereits Erprobte hinausgehen; auch hier gilt “natura non facit saltus”. Für die genetische Evolution ist das sinnvoll, denn große Mutationen führen in der Regel zur Lebensunfähigkeit. Aus diesem Grunde konnte die Natur z.B. das Rad nicht “erfinden”. Aus dem bereits Vorhandenen ließ es sich nicht komponieren.

Auch die Schritte der intellektuellen und in der Folge auch der kulturellen Evolution sind begrenzt. Phantasie und Denkkraft lassen nur kleine Abweichungen vom Gewohnten und damit nur kleine *Fort*schritte zu. Das gilt für alle Bereiche menschlicher intellektueller Tätigkeit, für die Technik ebenso wie für Kunst und Wissenschaft. Durch die Möglichkeit des gedanklichen Probierens wird die Evolution jedoch enorm beschleunigt und auch die Gedanken*sprünge* können größer werden. Der Mensch war in der Lage, das Rad zu erfinden, und die Entwicklung der künstlichen Sprachen macht unvergleichlich schnellere Fortschritte als die der natürlichen Sprachen. Nichtsdestoweniger ist es erstaunlich, wie langsam und mühsam sich die Programmiersprachen den Bedürfnissen der Nutzer anpassen. Das wird in Teil 3 deutlich werden.

Die hierarchische Struktur sprachlicher Ausdrücke ist der Syntax zuzurechnen, die anschließend behandelt wird. Wir haben die Besprechung des hierarchischen Komponierens vorgezogen, weil es ein Grundprinzip der Informatik ist, nach dem sowohl die Hardware als auch die Software informationeller Systeme aufgebaut ist.

5.3. Syntaxregeln

Das Wort *Syntax* wird in zwei Bedeutungen verwendet:

1. *Die **Syntax** eines Kompositzeichens ist dessen formale Komponierungsstruktur, d.h. sein Aufbau aus elementareren Zeichen (Bausteinzeichen).*
2. *Die **Syntax** einer Sprache ist die Gesamtheit aller Regeln (sog. Syntaxregeln), nach denen die Kompositzeichen der Sprache (z.B. Wörter, Sätze, Kommandos, Programme) formal, ohne Berücksichtigung der Bedeutung komponiert werden dürfen.*

Die aktuelle Bedeutung des Wortes “Syntax” geht aus dem jeweiligen Kontext hervor.

Die wichtigste Syntaxregel der deutschen Sprache legt fest, wie ein Aussagesatz normalerweise aufgebaut ist. Sie lautet:

<Aussagesatz> → <Subjekt> <Prädikat> [<Objekt>] (5.1)

3 z.B. “Hunde jagen Katzen.” Die Form, in der die Regel angegeben ist, lehnt sich an die sog. *Backus-Naur-Form* an. Der Pfeil (früher wurde an seiner Stelle ::= geschrieben) kann gelesen werden als “hat folgende Struktur”. Die Regel sagt aus, dass ein Aussagesatz mit dem Subjekt beginnt, ihm folgt das Prädikat und diesem das Objekt. Subjekt, Prädikat und Objekt werden unter dem Begriff *Satzglied* zusammengefasst. Die Bezeichnungen der Satzglieder sind in spitze Klammern eingeschlossen. Die eckigen Klammern zeigen an, dass das Objekt fehlen kann.

Für Programmiersprachen werden ähnliche Regeln vereinbart, nach denen u.a. Kommandos und Befehle komponiert werden, z.B. das Kommando “17+239=”. Es handelt sich um *Vereinbarungen*. Für das Additionskommando könnte ebenso gut

vereinbart werden, dass das Operationszeichen nicht zwischen den Operanden, sondern ihnen voran- oder nachzustellen ist. Die Rechnerhardware muss entsprechend aufgebaut bzw. die Software entsprechend programmiert sein. Befehle von *Maschinenprogrammen* für *Drei-Adress-Maschinen* (das sind Rechner, deren *Maschinenbefehle* drei Adressen enthalten) sind i.Allg. nach der Regel

<Befehl> → <Operationscode> <Operandenadresse1> <Operandenadresse2>

<Resultatadresse> (5.2)

aufgebaut (Code und Adressen in einer Zeile). Sogenannte *Assemblersprachen* gestatten die Verwendung von Namen anstelle von Adressen. Gemäß der angegebenen Regel würde z.B. der Befehl "ADD x y z" der Variablen z den Wert der Summe x+y zuweisen. Computer können i.Allg. nur syntaktisch richtige, der Mensch kann auch "verdrehte" Sätze verstehen.

Die Syntaxregel für Aussagesätze stellt eine Aussage über Aussagen dar. In diesem Sinne sprechen wir von *Metaaussagen*. Allgemein heißt eine Sprache, die der Beschreibung einer Sprache dient, *Metasprache*, die beschriebene Sprache heißt *Objektsprache*. Begriffe einer Metasprache wie z.B. "Aussagesatz", "Satzglied" oder "Subjekt" sind *metasprachliche* oder metalinguistische Begriffe. Sie spielen in einer Syntaxregel, wo sie durch spitze Klammern gekennzeichnet werden, die Rolle von Platzhaltern für alle möglichen Wörter oder Wortgruppen der Objektsprache. Darum heißen sie auch **metasprachliche Variable**.

Satzglieder können oft weiter dekomponiert werden. Das Subjekt kann z.B. aus einem Artikel, einer Substantivform und Attributen bestehen gemäß der Syntaxregel

<Subjekt> → [<Artikel>] {<Attribut>} <Substantivform>. (5.3)

Die geschweiften Klammern bedeuten, dass die Anzahl der Attribute beliebig ist. Für Substantivformen gilt die Syntaxregel

<Substantivform> → [<Präfix>] <Stamm> [<Suffix>] [<Endung>]. (5.4)

Es ergibt sich eine hierarchische syntaktische Struktur.

Syntaxregeln können eine Verkürzung (Verdichtung) sprachlicher Ausdrücke bewirken. Das lässt sich recht eindrucksvoll am Beispiel der Zahlendarstellung veranschaulichen. So ist die arabische Darstellung (Dezimalsystem) i.Allg. kürzer als die römische, und diese ist kürzer als die Darstellung mittels Strichen oder Kerben (Unärsystem). Für die Darstellung einer zweistelligen Dezimalzahl braucht man im römischen System bis zu 8 Zeichen (LXXXVIII) und im unären System bis zu 99 Zeichen. Das hat zwei Gründe. Zum einen steht eine unterschiedliche Anzahl elementarer Zeichen (Ziffern) zur Verfügung, nämlich 10 bzw. 7 bzw. ein einziges. Zum anderen spielt der Platz einer Ziffer in einer Zahl eine unterschiedliche Rolle. Im Unärsystem spielt sie gar keine und im römischen System eine sehr eingeschränkte Rolle. Hier ist die Reihenfolge im wesentlichen vorgegeben. Beispielsweise darf

auf die Ziffer V nur die Ziffer I oder das Ende der Zahl folgen. Dagegen gibt der Platz (die “Stelle”) einer Ziffer in einer Dezimalzahl die Zehnerpotenz der “Stelle” an. Tauscht man zwei nicht identische Ziffern einer Dezimalzahl aus, ergibt sich eine andere Zahl. Dies ist ein Beispiel dafür, wie aus der Syntax eines Kompositzeichens auf dessen Bedeutung geschlossen werden kann.

Ähnlich ist es bei der Satzsyntax. Wenn Subjekt und Prädikat ihre Plätze wechseln, wird aus einem Aussagesatz ein Fragesatz. Um das richtige Interpretieren zu sichern oder zu erleichtern, wird am Satzende die Stimme gehoben bzw. ein Fragezeichen gesetzt. Dadurch wird auch ein “verdrehter” Fragesatz wie z.B. “Hunde jagen Katzen?” richtig interpretierbar.

Syntaxregeln der natürlichen Sprachen sind keine strengen Regeln; man darf eventuell gegen sie verstoßen. Das gilt auch für die Syntax des Aussagesatzes. Beispielsweise lässt die deutsche Sprache auch die umgekehrte Interpretation des Satzes “Hunde jagen Katzen” zu im Sinne “Hunde werden von Katzen gejagt”. Denn Subjekt und Objekt dürfen ihre Plätze wechseln, und aus den Endungen der Substantiva “Hunde” und “Katzen” geht nicht hervor, welches im Nominativ und welches im Akkusativ steht. Durch die Anwendung des Passivs wird die Bedeutung eindeutig.

Nichteindeutigkeiten werden in der Regel durch den Kontext, durch die Stimmführung oder durch Satzzeichen aufgehoben. Doch erschweren sie demjenigen das Verständnis sehr, der versucht, einen fremdsprachigen Text mit Hilfe von Lexikon und Grammatik zu übersetzen. Er muss mit einer *Syntaxanalyse* beginnen und jeden Satz zerlegen, d.h. er muss mit Hilfe der Syntaxregeln schrittweise seinen *Syntaxbaum* konstruieren. Bild 5.2 zeigt den Syntaxbaum des Satzes von Bild 5.1. Dabei sind die Knoten des dortigen Graphen (die Punkte) mit den entsprechenden metasprachlichen Variablen benannt.

4

Rechner können den Kontext i.d.R. nicht berücksichtigen. Darum müssen Syntaxregeln streng befolgt werden. Die Zahlendarstellung hatte bereits gezeigt, dass die Bedeutung eines Kompositzeichens mit dessen syntaktischem Aufbau gekoppelt sein kann. Das gilt auch für Sätze. Aufbau und Bedeutung eines Satzes stehen miteinander in Beziehung und zwar über die abstrakte (metasprachliche) Bedeutung der Satzglieder (der metasprachlichen Variablen). Die abstrakte Bedeutung des Prädikats besteht beispielsweise darin, dass es dem Subjekt ein Merkmal (Eigenschafts- oder Tätigkeitsmerkmal) zuweist.

Die Beziehung zwischen Aufbau und Bedeutung eines Satzes ist Voraussetzung dafür, dass man ihn verstehen kann. Will man einen Satz aus einer Sprache, die man nicht beherrscht, übersetzen, muss man ihn zunächst syntaktisch analysieren. Darauf wird in Kap.16.4 eingegangen.

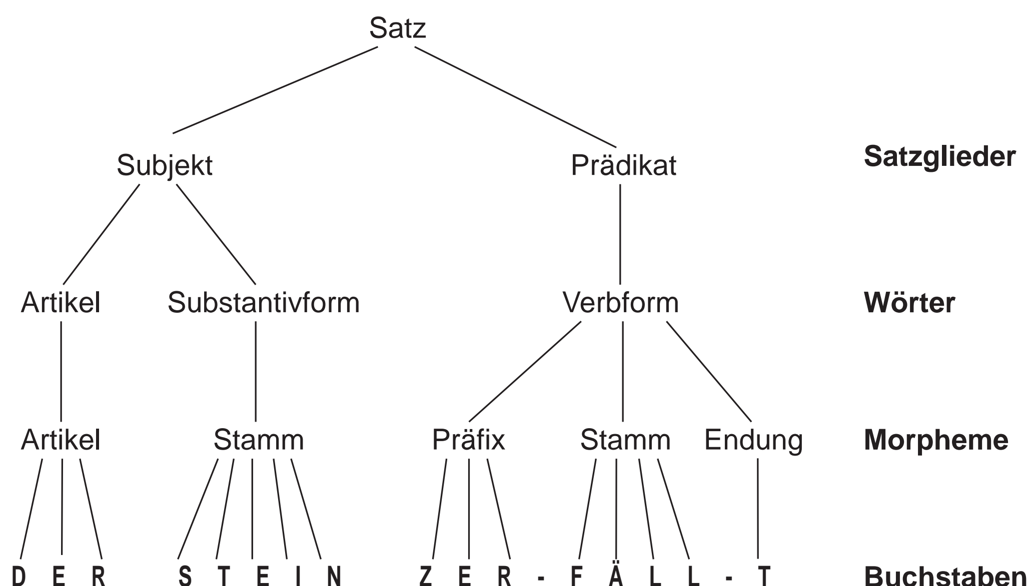


Bild 5.2 Syntaxbaum eines Aussagesatzes

5.4 Externe, interne und formale Semantik

Wir wenden uns nun der dritten der oben genannten Wirkungen des Selektionsdruckes zu, der *Idemobjektivierung*. In letzter Konsequenz hat sie zur Entwicklung der Mathematik geführt.

Vielen Lesern wäre es vielleicht natürlicher, nicht von Idemobjektivierung, sondern von der Herausbildung einer subjektunabhängigen *Semantik*, also von “Semantikobjektivierung” oder von “semantischer Objektivierung” zu sprechen. Wir haben den Semantikbegriff bisher bewusst vermieden, weil er vieldeutig ist und leicht zu Missverständnissen führt. Mit *Semantik* wird zum einen die Bedeutung sprachlicher Einheiten, zum anderen die Lehre von der Bedeutung sprachlicher Einheiten bezeichnet. Wir werden das Wort in der Regel im erstgenannten Sinne verwenden, jedoch mit folgender Präzisierung: *Die* ***Semantik*** *eines Kompositzeichens einer Sprache ist dessen Bedeutung unter der stillschweigenden Annahme, sie sei für alle Nutzer der Sprache die gleiche bzw. Unterschiede seien vernachlässigbar*. Die Ausmerzung vorhandener und die Vermeidung möglicher Unterschiede ist das Ziel der Idemobjektivierung, die wir im Weiteren auch **semantische Objektivierung** nennen werden. Sie ist die Voraussetzung der popperschen Welt 3..

5

6

In der technischen Informationsverarbeitung ist die Unmissverständlichkeit besonders wichtig, denn als Interpretierer sprachlicher Ausdrücke fungieren nicht nur Menschen, sondern auch Computer, und denen muss absolut eindeutig gesagt werden, was sie zu tun haben. Die Folge ist, dass in der Rechentechnik der Semantikbegriff i.Allg. nicht aus der Sicht des Nutzers, sondern aus der Sicht des Computers definiert wird. Wenn ein Informatiker, der sich mit der Entwicklung von Program-

miersprachen oder Compilern beschäftigt, allgemeinverständlich erklären soll, was 7 unter Semantik zu verstehen ist, wird er vielleicht sagen: *“Die **Semantik** von Befehlen oder Programmen ist deren **Wirkung** im Computer, es ist der durch sie ausgelöste Prozess bzw. dessen Resultat, es ist ihre **Interpretation** durch den Computer.”*

Diese Begriffsbestimmung bindet die Semantik an den Computer, an das System, in dem der Interpretierungsprozess abläuft, das ihn “trägt”. Darum sprechen wir von trägerinterner oder kurz von **interner Semantik**. Wenn man dagegen von der Semantik eines umgangssprachlichen Ausdrucks spricht, meint man nicht deren trägerinterne Semantik, also nicht irgendwelche neuronalen Prozesse im Gehirn. Nach unserer Charakterisierung natürlicher Sprachen als Mittel der Modellierung der Umwelt bedeutet ein Aussagesatz einen Sachverhalt, der in der Außenwelt gegeben ist. Darum sprechen wir von **externer Semantik**.

Es erhebt sich nun folgendes Problem. Wie kann der Mensch, der externsemantisch denkt und spricht, einem Computer, der internsemantisch “versteht” und 8 “denkt”, Aufträge erteilen? Er muss nämlich beim Programmieren die *semantische Lücke* zwischen externer und interner Semantik gedanklich *überspringen*. Dies ist das Kernproblem des Programmierens, man kann sogar sagen das *Kernproblem der technischen Informatik*; wir nennen es das **technische Semantikproblem**. Wir werden uns ausführlich mit ihm beschäftigen müssen (siehe Kap.15.5). Hier soll nur auf den besonderen Fall eingegangen werden, dass die Lücke praktisch nicht existiert.

Wenn man beispielsweise in seinen Taschenrechner das *Kommando* “17×239=” eintastet, braucht man keinerlei “Semantiksprünge” zu machen, denn der Rechner “versteht” die Sprache der Arithmetik. Er kann arithmetische Ausdrücke interpretieren, weil die arithmetischen Operationen im Rechner hardwaremäßig “verdrahtet” sind. Es existieren die entsprechenden Operatoren, und eine sinnreiche Steuerung des Datenflusses sorgt dafür, dass die Operanden dem geforderten Operator zugeführt werden und das Resultat im Anzeigefeld erscheint.

Ergänzt man die arithmetischen durch einige *logische Operatoren*, so bedarf es weiter keiner Hardwareoperatoren für die Verarbeitung von Zeichenketten, um aus der “Rechentechnik” eine “Denktechnik” zu machen, um künstliche Intelligenz zu fabrizieren. Das erscheint auf den ersten Blick unglaublich, und man fragt sich, wie das möglich sein soll. Diese Frage wird im Weiteren Schritt für Schritt beantwortet.

Die scheinbare Nichtexistenz der semantischen Lücke bei der Nutzung eines Taschenrechners hat ihre Ursache in der impliziten Verwendung einer dritten Art von Semantik, die uns ganz besonders interessieren wird, denn es handelt sich um das Ergebnis uralter zielstrebiger Bemühungen der Menschen um einen exakten Sprachgebrauch zum Zwecke der Vermeidung von Missverständnissen, also um Idemobjektivierung. Um zu erreichen, dass alle Kommunikationspartner Mitteilungen identisch interpretieren, werden *formale Sprachen*, *Kalküle* und *formale Theorien* konstruiert.

9 *Eine **formale Sprache** ist eine Menge elementarer, semantikfreier Zeichen (Alphabet) zusammen mit einer Menge von Syntaxregeln*. Sämtliche Vereinbarungen

und Regeln sind exakt zu befolgen. *Ein **Kalkül** ist eine formale Sprache zusammen mit einer Menge von Transformations- oder Rechenregeln zum Umformen (Transformieren) von Ausdrücken der formalen Sprache*. Die Regeln legen fest, welche Ausdrücke in welche überführt (durch welche *substituiert*) werden dürfen. Beispielsweise sind 3+5=8 oder 3×5=15 Rechenregeln der Arithmetik, des *arithmetischen Kalküls*. Üblicherweise wird in Transformationsregeln der Pfeil anstelle des Gleichheitszeichens verwendet und z.B. "3+5→8" notiert. Ein anderes Beispiel ist die trigonometrische Transformationsregel sinx/cosx → tanx.

Die Formulierung derartiger Regeln beinhaltet eine Zuordnung bestimmter Bedeutungen zu den Zeichen der Sprache, beispielsweise die Festlegung, dass "+" das Operationszeichen der Addition ist. Dadurch werden Zeichen zu *Symbolen* und die zunächst völlig semantikfreie formale Sprache wird zu einer *Kalkülsprache*. Die Bedeutungszuweisung heißt kalkülspezifische oder **formale Interpretation**. *Die zugeordnete Semantik heißt kalkülinterne oder **formale Semantik**. Eine formal (kalkülspezifisch) interpretierte formale Sprache heißt **Kalkülsprache***. 10

Um einen Kalkül für die sprachliche Modellierung eines Ausschnitts der Welt, eines "externen" (außerhalb des Kalküls existierenden) Originalbereichs verwenden zu können, müssen die Objekte des Originalbereichs sowie Aussagen über diesen Bereich Entsprechungen im Kalkül erhalten, m.a.W. der Kalkül muss durch den Originalbereich *interpretiert* werden. Diese Interpretation nennen wir **externe Interpretation**. *Ein Kalkül, der durch einen Ausschnitt der Realität interpretiert ist, heißt **formale Theorie** des Realitätsausschnitts*. Diese Bezeichnung trifft z.B. auf physikalische Theorien zu. Durch externe Interpretation wird die *reine* zur *angewandten Mathematik*. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter Interpretation stets *externe* Interpretation zu verstehen. In der Mathematik wird unter Interpretation in der Regel die formale Interpretation verstanden. 11

Externe Interpretation kann auf unterschiedlichen Abstraktionsniveaus erfolgen. Beispielsweise kann ein Zeichen einen Gegenstand, eine Energie, eine Wahrscheinlichkeit, den Wahrscheinlichkeitswert einer Aussage oder noch abstraktere Inhalte bezeichnen. Aber selbst wenn ein konkreter Gegenstand der realen Welt bezeichnet wird, handelt es sich im Modell bereits um ein abstraktes Objekt. Newton musste, um seine Theorie der Planetenbewegung aufzubauen, den Begriff des Planeten - also auch den der Erde - auf den Begriff des Massenpunktes reduzieren und die Erde (ebenso wie die Planeten und die Sonne) durch einen einzigen metrischen Merkmalswert, ihre Masse beschreiben, abgesehen von den Zustandsmerkmalen Ort und Impuls. Dem stelle man den *Assoziationskomplex* gegenüber, den man mit dem Wort "Erde" in dem Satz "Die Erde war wüst und leer" verbindet. Man hat es hier mit einer immensen bedeutungsmäßigen Verkürzung, mit einer *Idemschärfung* zu tun, ohne welche die notwendige semantische Objektivierung nicht möglich gewesen wäre. Sie führt letzten Endes auf Zahlen, also auf Ideme, die für alle Menschen identisch sind und die wir darum *universelle Ideme* nennen. Sie sind die Voraussetzung für die mathematische Modellierung der Welt.

12 Die verwendete Methode, also das Verkürzen der Bedeutung um nichtrelevante (für das Modellierungsziel unwichtige) Inhalte, wird als *idealisierendes Abstrahieren* oder kurz *Idealisieren* bezeichnet. Idealisieren ist von fundamentaler Bedeutung, denn jeder Mensch, nicht nur der Mathematiker, idealisiert ständig beim Artikulieren von Idemen. ***Idealisierende Abstraktion*** *ist die Voraussetzung sprachlicher Modellierung und sprachlicher Kommunikation.*

Es ist eine ergänzende Bemerkung zum Kalkülbegriff erforderlich, denn die oben gegebene Definition ist insofern nicht ganz befriedigend, als sie nichts über den Umfang der Menge der Rechenregeln aussagt. Welche Regeln müssen beispielsweise aufgelistet werden, um die Arithmetik oder die Geometrie als Kalkül zu definieren. Das Problem wäre gelöst, wenn eine endliche Menge von Regeln angegeben werden könnte, aus denen sich alle anderen Regeln ableiten lassen. Diese Vorstellung führte zur sog. *Axiomatisierung* von Kalkülen und zu ihrer Definition mittels *Axiomensystemen. Ein* ***Axiomensystem*** *ist eine Menge voneinander unabhängiger und untereinander widerspruchsfreier Rechenregeln (allgemein von Sätzen), aus denen sich sämtliche Rechenregeln (sämtlich Sätze) eines Kalküls ableiten lassen.*

So lassen sich z.B. alle Aussagen der euklidischen Geometrie aus den von EUKLID formulierten Axiomen ableiten, und alle Aussagen über das Verhalten eines Systems von Massenpunkten, allgemein alle Aussagen der newtonschen Mechanik, lassen sich aus den newtonschen Axiomen ableiten. *Ein Kalkül mit Axiomensystem heißt* ***axiomatisierter Kalkül****.*

Wir fassen die getroffenen Vereinbarungen noch einmal mit anderen Worten zusammen (vgl. Bild 5.3): *Wir sprechen von* ***externer*** *bzw.* ***interner*** *bzw.* ***formaler Semantik***[1] *eines sprachlichen Ausdrucks, wenn dieser bedeutungsmäßig an die Außenwelt bzw. an die Struktur und Funktion des Trägers bzw. an ein Kalkül*

Bezeichnung der Semantik eines sprachlichen Ausdrucks	Bedeutungsmäßige Anbindung des sprachlichen Ausdrucks
extern	an die Umwelt des modellierenden Menschen
formal	an ein Kalkül
intern (trägerintern)	an die Struktur und Funktion des Modellträgers

Bild 5.3 Drei Semantiken

1 Die Bezeichnung *formale Semantik* wird in der Literatur zuweilen in einem anderen, spezielleren Sinne verwendet.

angebunden ist. Abschließend sei eine im Grunde selbstverständliche Bemerkung angefügt. Grundlage des exakten, d.h. formalisierten oder genauer kalkülisierten Modellierens ist die Mathematik, denn das Erfinden von und Hantieren mit Kalkülen ist Gegenstand der Mathematik. 13

5.5 Begriffsbildung

Begriffsbildung[2] und Grammatik tragen dazu bei, die Ausdrucksstärke einer Sprache zu erhöhen und so die Kommunikation zwischen den Kooperationspartnern schneller und flexibler zu gestalten. Anstelle von Ausdrucksstärke werden wir auch von *semantischer Dichte* sprechen. Dieser Begriff ist, wenn er sich auf natürliche Sprachen bezieht, ebenso wie der Idembegriff, introspektiv und insofern unscharf definiert. *Die* ***semantische Dichte*** *eines natürlichsprachigen Textes ist durch den Umfang des im Mittel pro Wort assoziierbaren Gedächtnisinhaltes (Menge der assoziierbaren Ideme) gegeben.* Die semantische Dichte hängt also wesentlich vom Empfänger, von dessen Erfahrung und Wissen, von seiner Aufnahmefähigkeit und Phantasie und, wenn man die emotionale Komponente von Idemen einbezieht, auch von seinem Empfindungs- und Einfühlungsvermögen ab, was z.B. beim Interpretieren von Kunstwerken ganz entscheidend sein kann. Die semantische Dichte von Programmiersprachen bzw. Programmen lässt sich schärfer definieren, nämlich als *mittlere Anzahl elementarer Operationen (ALU-Operationen) pro Zeichen.* 14

Die *Grammatik* kann eine semantische Verdichtung durch Optimierung der Syntaxregeln mit dem Ziel einer Minimierung der Artikulierungen (der mittleren Länge der Zeichenrealeme) erreichen. Die *Begriffsbildung* kann eine semantische Verdichtung dadurch bewirken, dass Begriffe mit hoher semantischer Belegung gebildet werden.

Ein ***Begriff*** *ist ein objektiviertes, benanntes Idem.* Seine semantische Belegung ist umso höher, je mehr Gedächtnisinhalte der Interpretierer dem codierenden Zeichenrealem zuordnet, je mehr er mit ihm “assoziiert”. Wörter mit hoher semantischer Belegung sind z.B. “Tierreich”, “Erde”, “Gott”. In Kap.5.4 [12] hatten wir die Umkehrung, die Verkürzung der semantischen Belegung, die Idemschärfung durch *idealisierende Abstraktion* zum Zwecke der *semantischen Objektivierung* besprochen. Jetzt wenden wir uns den Methoden der semantischen Verdichtung durch solche Begriffsbildungen zu, die weder unscharf sind, noch der Idemschärfung dienen, sondern der sprachlichen Modellierbarkeit der unendlichen Welt mittels Abstraktion. ***Abstraktion*** *ist die einzige Möglichkeit, die unendliche Welt sprachlich zu modellieren.*

2 Zur Begriffsbildung siehe z.B. [Klix 80].

Bei den folgenden Methoden der Begriffsbildung handelt es sich um gedankliche Zusammenfassungen mehrerer, eventuell auch sehr vieler Objekte nach bestimmten Gesichtpunkten, wobei die Zusammenfassungen mit Namen *benannt* werden. Zwei Beispiele sollen das verdeutlichen. Durch die Sätze

Hose, Rock und Weste bilden einen Anzug

Hose, Rock und Weste sind Kleidungsstücke

werden drei Objekte nach unterschiedlichen Gesichtspunkten zusammengefasst und die Zusammenfassungen werden mit "Anzug" bzw. "Kleidungsstück" benannt. Der erste Satz bildet den Begriff "Anzug" dadurch, dass die Bestandteile eines Anzuges aufgezählt werden. Der Begriff "Anzug" wird durch Komponierung gebildet. Darum sprechen wir von **komponierender Abstraktion**. Im zweiten Satz werden die Begriffe Hose, Rock und Weste zum Begriff Kleidungsstück verallgemeinert oder, wie auch gesagt wird, unter dem Begriff Kleidungsstück *subsumiert*. In diesem Fall sprechen wir von *generalisierender Abstraktion* oder kurz von **Generalisierung**.

Es ist zu beachten, dass im zweiten Satz mit Hose, Rock und Weste keine konkreten Objekte, sondern Mengen (Klassen) von Objekten bezeichnet sind. Aber auch im ersten Satz sind keine kokreten Hosen, Röcke und Westen gemeint. Genauer müsste es heißen: Je ein Kleidungsstück von Typ Hose, Typ Rock und Typ Weste oder je ein Kleidungsstück der Klassen Hose, Rock und Weste bilden einen Anzug. Das Bilden von Begriffen ist eine gedankliche Tätigkeit und die Begriffe, mit denen gearbeitet wird, sind Ideme mit objektivierten Merkmalen, d.h. mit Merkmalen, die den Begriffen von allen Beteiligten einheitlich (mit der gleichen Bedeutung) zugeordnet werden, wie z.B. das Merkmal "ärmellos" dem Begriff "Weste".

Um eine kompakte Beschreibung der verschiedenen Methoden der Begriffsbildung zu ermöglichen, führen wir die Begriffe Klasse und Typ ein. Eine **Klasse** ist die begriffliche Zusammenfassung von Objekten (den Elementen der Klasse), die in einem oder mehreren **Merkmalen** übereinstimmen, also dieselben *Merkmalsausprägungen* oder **Merkmalswerte** besitzen; mit anderen Worten: *eine **Klasse** ist eine Menge, die durch ein oder mehrere **Merkmalswerte** festgelegt ist, die alle Elemente der Menge besitzen.* Beispielsweise sind "rot" und "grün" Ausprägungen (Werte) des Merkmals "Farbe". Umgangssprachlich wird i.Allg. nicht zwischen Merkmal und Merkmalswert unterschieden. Wenn Missverständnisse ausgeschlossen sind, wird auch in diesem Buch gelegentlich nicht zwischen Merkmal und Merkmalswert unterschieden. Von einem Element einer Klasse sagt man, dass es der betreffenden Klasse *angehört* oder dass es vom **Typ** der betreffenden Klasse ist. Beispielsweise gehört die Zahl π, aufgerundet gleich 3.1416, der Klasse der reellen Zahlen an, sie ist vom Typ der reellen Zahlen.

Mit Hilfe des Klassenbegriffs sind wir in der Lage, die Begriffsbildungsmethoden, die in der ersten Spalte von Bild 5.4 aufgeführt sind, kurz und klar zu beschreiben.

15 ***Klassifizierendes Abstrahieren*** *(kurz: Klassifizieren) ist das Zusammenfassen mehrerer Objekte, die in einem oder in mehreren Merkmalen übereinstimmen, zu*

einem neuen abstrakten Objekt, Klasse genannt. Das Klassifizieren von Daten wird in der Informatik auch **Typisieren** genannt. Vom Klassifizieren (wörtlich übersetzt: Klassen machen) ist das *Klassieren* zu unterscheiden. *Unter* ***Klassieren*** *verstehen wir das Einordnen eines Objektes in eine bereits existierende Klasse.*. Umgangssprachlich wird anstelle des Wortes *Klassieren* i.d.R. das Wort *Identifizieren* benutzt, beispielsweise, wenn man einen Baum als Buche "identifiziert" (klassiert). Klassifizieren und Klassieren beinhalten *Idealisieren* insofern, als beim Definieren einer Klasse bzw. beim Einordnen eines Objekts in eine Klasse alle nichtrelevanten Merkmale vernachlässigt werden, von ihnen wird "abstrahiert". In der Umgangssprache wird im allgemeinen nicht zwischen Klassifizieren und Klassieren unterschieden, sondern in beiden Fällen von Klassifizieren gesprochen. Wir schließen uns diesem Sprachgebrauch an, solange Missverständnisse ausgeschlossen sind.

Komponierendes Abstrahieren *ist das Bilden einer Kompositklasse aus Komponentenklassen. Aus den Elementen der Komponentenklassen werden Elemente der Kompositklasse komponiert*. Im obigen ersten Beispielsatz sind Hose, Rock und Weste die Namen der Komponentenklassen und Anzug ist der Name der Kompositklasse. Komponierende Begriffsbildung liegt vor, wenn im Rahmen einer Komponierungshierarchie ein neues Komposit gebildet wird, das dann die Rolle eines selbständigen Denkobjekts spielt. Das komponierende Abstrahieren wird beim Komponieren von Operatorenhierarchien eine wichtige Rolle spielen. Auf höheren Komponierungebenen wird Komponieren zuweilen als *Aggregieren* bezeichnet. 16

Generalisierendes Abstrahieren *ist das Zusammenfassen mehrerer Klassen,* ***Unterklassen*** *genannt, zu einer einzigen Klasse,* ***Oberklasse*** *genannt. Generalisieren erfolgt durch Außerachtlassung (Streichung) eines oder mehrerer Merkmale.* In dem obigen zweiten Beispielsatz sind Hose Rock und Weste die Namen der Unterklassen und Kleidungsstück ist der Name der Oberklasse. Durch Generalisierung werden z.B. die Klassen der rationalen und irrationale Zahlen zur Klasse der reellen Zahlen zusammengefasst durch Streichung des Merkmals "Darstellbarkeit als Bruch ganzer Zahlen", das für Rationalzahlen den Wert "darstellbar", für Irrationalzahlen den Wert "nichtdarstellbar" hat.

Man beachte, dass jede der drei genannten Arten des Abstrahierens *Merkmalsbildung* voraussetzt und *idealisierendes* Abstrahieren einschließt. Ohne Merkmale und Merkmalswerte und ohne Idealisieren, ohne Verzicht auf Beschreibungsdetails ist keine Klassenbildung möglich. Mit dem Erfassen von Merkmalen beginnt das sprachliche Modellieren der Welt durch ein Kind, sicherlich bereits vor seiner Geburt. Bei der Bildung neuer Denkobjekte und Begriffe finden Merkmalsbildung und Klassenbildung häufig gleichzeitig statt.

In Bild 5.4 sind die vier Arten des Abstrahierens zusammengestellt. Sie können als vier *begriffsbildende Operationen* aufgefasst werden. In der letzen Zeile ist das *Benennen* hinzugefügt, jedoch separat, denn das Benennen ist kein Abstrahieren, wohl aber eine Komponente der Begriffsbildung, denn Begriffe sind *benannte* objektivierte Ideme.

Begriffsbildende Operation		Aufwärtsrelation
aufwärts: Abstrahieren	abwärts: Konkretisieren	Übergang von — nach beim Abstrahieren
Idealisieren	Realisieren	der Realität näher — ferner
Klassifizieren, Typisieren	Instanzieren, Ausprägen	Exemplar — Klasse, Typ
Komponieren, Aggregieren	Dekomponieren	Teil — Ganzes
Generalisieren	Präzisieren	Unterklasse — Oberklasse
Benennen, Referenzieren	Dereferenzieren	Benanntes — Name

Bild 5.4 Begriffsbildende Operationen

In der letzten Spalte ist die jeweilige Art des Abstrahierens durch einen begrifflichen Übergang “von — nach” charakterisiert, beispielsweise das komponierende Abstrahieren durch den Eintrag “Teil — Ganzes”.

In der zweiten Spalte sind die Operationen des *Konkretisierens* genannt, die dem Abstrahieren entgegengerichtet sind und das Abstraktionsniveau erniedrigen. Die Bezeichnungen der Abwärtsoperationen bedürfen kaum einer Erläuterung. Das *Ausprägen* oder *Instanzieren* entspricht dem Übergang von einer Klasse zu einem Element der Klasse. Der letzte Schritt des Instanzierens führt zu einem *konkreten* (materiellen) Objekt, zu einem *Exemplar* der betreffenden Klasse. In diesem Fall hatten wir in Verbindung mit Bild 2.1 von *materiellem Instanzieren* gesprochen. Dort war auch auf die Herkunft des Wortes “Instanzieren” eingegangen worden.

Wegen der großen Bedeutung des Generalisierens und Präzisierens im Bereich der Wissenschaft und speziell der Informatik soll auf diese Art der Begriffsbildung etwas ausführlicher eingegangen werden. Durch schrittweises Generalisieren und/oder Präzisieren lässt sich eine Hierarchie abstrakter Begriffe aufbauen. Ein Schulbeispiel ist die Systematik des Tierreiches, d.h. seine Unterteilung durch Präzisierung in Unterreiche, Stämme, Arten usw. In einem Abwärtsschritt in der Hierarchie wird die Beschreibung der zu klassifizierenden (systematisierenden) Objekte (Tiere) durch Hinzunahme eines oder mehrerer Merkmale präzisiert. Bei einem Aufwärtsschritt wird sie durch Streichung eines oder mehrerer Merkmale verallgemeinert (generalisiert).

Das Aufbauen einer Begriffshierarchie und das Arbeiten mit ihr soll am Syntaxbaum von Bild 5.2 veranschaulicht werden. Der Sprachunkundige muss beim Übersetzen des Satzes diesen zunächst syntaktisch analysieren, d.h. er muss die einzelnen Teile (die Bausteinrealeme) des Satzes als bestimmte Wort- bzw. Satzglieder erkennen (z.B. die Wortgruppe "Der Stein" als Satzsubjekt), er muss jedem Knoten des Graphen den richtigen grammatikalischen (metasprachlichen) Begriff zuordnen. Der so entstehende Syntaxbaum enthält zwei orthogonal zueinander liegende Hierarchien, eine senkrechte, komponierende Hierarchie mit vier Ebenen (die Buchstabenebene nicht mitgezählt) und eine waagerechte, generalisierende Hierarchie mit zwei Ebenen. Beim Generalisieren werden die Begriffe einer Ebene zu einem Oberbegriff (fett gedruckt in der rechten Spalte) zusammengefasst, d.h. aus Klassen werden Oberklassen gebildet. Beim Komponieren werden Kompositklassen der nächsthöheren Ebene gebildet. Für den jeweils obersten Begriff der Komponierungs- bzw. Generalisierungshierarchie gelten die folgende Aussagen:

> Subjekt, Prädikat und Objekt bilden einen Satz.
>
> Subjekt, Prädikat und Objekt sind Satzglieder.

Die beiden Sätze demonstrieren - ebenso wie die beiden Beispielsätze über Kleidungsstücke - das Komponieren (erster Satz) und das Generalisieren (zweiter Satz).

Der recht komplizierte Prozess der Syntaxanalyse, d.h. das Einordnens der einzelnen Wörter in metasprachliche Klassen, läuft im Gehirn jedes Menschen ab, der Sprache interpretiert. Doch falls der Interpretierer die Sprache perfekt beherrscht, braucht sein Oberbewusstsein dabei nicht bemüht zu werden, die Analyse ist reflektorisch überbrückt.

Der schrittweise Aufbau des Syntaxbaumes zeigt noch einmal sehr deutlich, wie Syntax und Semantik miteinander verflochten sind. Denn dadurch, dass man eine aus dem Satz herausgeschnittene Buchstabenkette als ein bestimmtes Wort- oder Satzglied erkennt, weist man ihr bereits "ein Stück Semantik" zu. Wenn z.B. die Kette "Der Stein" als Subjekt erkannt ist, dann folgt daraus, dass der Satz eine Aussage über den Stein macht.

17

Grundlage der Begriffsbildung ist, wie wir gesehen haben, die Beschreibung (sprachliche Modellierung) von Objekten der Realität oder des Denkens (von Realemen oder Idemen) durch *Merkmalswerte*. Das gedankliche Verbinden (Assoziieren) bestimmter Gegenstände mit bestimmten Eigenschaften (und umgekehrt) ist ein wesentlicher Bestandteil der Anpassung des Menschen an seine Umwelt. Das Ergreifen und Begreifen der Welt durch den Säugling beginnt mit der Unterscheidung von Gegenständen aufgrund unterschiedlicher Eigenschaften wie Temperatur, Festigkeit, Farbe und Geruch, über die ihn seine Sinnesorgane informieren. Im Weiteren werden wir sehen, wie das *sprachliche Modellieren mittels Merkmalen und Merkmalswerten* zum Grundprinzip der Wissensrepräsentation in der technischen Informationsverarbeitung, speziell in der Datenbanktechnik [16.5] geworden ist. Es sei noch einmal

wiederholt, dass Begriffe durch Abstraktion entstehen und insofern Ideme sind, die [18] sich durch Merkmale auszeichnen. Wenn zum Ausdruck gebracht werden soll, dass ein Idem ein durch Merkmale charakterisiertes gedachtes Objekt ist, nennen wir es **Denkobjekt**. Das *Denkobjekt* der Human-IV (Informationsverarbeitung durch den Menschen) entspricht dem *Datenobjekt* in der Computer-IV (Informationsverarbeitung durch den Computer). Verallgemeinernd kann man sagen: ***Sprachliches Modellieren*** *ist das Hantieren mit Denkobjekten bzw. mit Datenobjekten und damit ein Operieren mit* ***Merkmalswerten****.*

In diesem Zusammenhang spielt der Begriff der *Assoziation* eine wichtige Rolle. Wir haben ihn wiederholt verwendet, ohne ihn zu definieren. In Kap.7 werden wir uns überlegen, wie der Assoziationsbegriff in der Human-IV im Zusammenhang mit der *menschlichen* Intelligenz verwendet wird. Hier geben wir eine Begriffsbestim- [19] mung, die sowohl in der Human-IV als auch in der Computer-IV anwendbar ist. *Unter* ***Assoziieren*** *verstehen wir das Aufrufen von Objekten (Denkobjekten oder Datenobjekten) über ihre Merkmalswerte oder das Aufrufen von Merkmalswerten über Objekte, welche sich durch diese Merkmalswerte auszeichnen, oder das Aufrufen mittels Kombination dieser beiden Aufrufmethoden.* Beim Aufrufen aus dem menschlichen Gedächtnis spricht man von "ins Bewusstsein rufen", beim Aufrufen aus einem Computerspeicher von "zugreifen auf".

Drei Beispiele sollen die verschiedenen Methoden des Assoziierens illustrieren. Wenn ich beim Anblick eines Bekannten an dessen Geburtsdatum erinnert werde, assoziiere ich mit einem Objekt einen Merkmalswert. Wenn mich die Beschreibung einer Person in einem Roman an einen Bekannten erinnert, assoziiere ich mit Merkmalswerten ein Objekt. Wenn mich eine Wolke an einen Hund erinnert, assoziiere ich mit einem Objekt ein anderes Objekt über gemeinsame Merkmale.

In Teil 3 werden wir sehen, welch bedeutende Rolle die begriffsbildenden Operationen in der Softwaretechnik spielen, konkret die komponierende Abstraktion als sog. *prozedurale Abstraktion* beim Aufbau sprachlicher Operatorenhierarchien (Kap.18.2 [18.3]) und die klassifizierende und generalisierende Abstraktion in der Datenbanktechnik (Kap.16.2 [16.7] und im objektorientierten Programmierparadigma (Kap.18.2 [18.4]).

An dieser Stelle unterbrechen wir unsere Überlegungen zu dem großen Thema "Gedanke und Sprache", um eine Diskussion zum Begriff der "*syntaktischen Information*" einzuschieben. Er spielt in der Informatik-Literatur eine so große Rolle, dass wir ihn dem Leser nicht vorenthalten wollen, obwohl wir ihn nicht benötigen, um den Computer und die künstliche Intelligenz nachzuerfinden. Das Kapitel 5.6 kann ohne Beeinträchtigung der weiteren Lektüre übersprungen werden.

5.6* Umcodierungseffizienz und syntaktische Information

Die stürmische Entwicklung der Kommunikationstechnik und die explosionsartige Zunahme der Menge an Informationen, die über die Medien verbreitet werden, und andrerseits die zunehmende Formalisierung der Kommunikationssprachen (man denke an die vielen Fach- und Programmiersprachen), führt einem die gegenwärtigen Wirkungen des Selektionsdruckes, von dem zu Beginn des Kapitels 5.1 die Rede war, drastisch vor Augen. Freilich spielt dabei die bewusste, zielstrebige Tätigkeit der Menschen eine erheblich größere Rolle als bei der Herausbildung der natürlichen Sprachen, mit anderen Worten, aus dem Selektionsdruck werden technische Zielstellungen.

So wurde aus dem Druck in Richtung Beschleunigung der Kommunikation die Zielstellung, die Zeit zu verkürzen, die für die Übertragung einer Nachricht über einen *Nachrichtenkanal* notwendig ist. Dabei spielen zwei Faktoren eine Rolle,

- die semantische Dichte der Nachricht und
- die Anzahl der Zeichen, die der Kanal pro Zeiteinheit übertragen kann.

Der zweite Faktor hat eine technische Entwicklung ausgelöst, die durch zwei Entscheidungen geprägt ist, die Entscheidung für binäre Codierung, also für ein Alphabet, das nur zwei Zeichen enthält, und die Entscheidung für elektromagnetische Wellen als Träger. Die Beschränkung auf zwei Zeichen hat in erster Linie ökonomische Gründe, denn je mehr Zeichen unterschieden werden müssen, umso höher wird der Aufwand, der getrieben werden muss, um die Störanfälligkeit des Nachrichtenkanals zu senken und die fehlerfreie Zeichenübertragung zu sichern. *Ein Zeichen des Binäralphabets heißt* ***Bit***, so wie ein Zeichen des deutschen Alphabets *Buchstabe* heißt. Ein Nachrichtenkanal, der nur zwei verschiedene Zeichen, also Bits übertragen kann, heißt *Binärkanal*. 20

Die Nutzung elektromagnetischer Wellen ermöglicht die Annäherung an die maximal mögliche Geschwindigkeit, mit der sich Energie und folglich auch Nachrichten übertragen lassen, an die Lichtgeschwindigkeit. Dabei ist man bemüht, mit möglichst hoher *Trägerfrequenz* (Frequenz der tragenden Welle) zu arbeiten, denn je höher die Frequenz (genauer gesagt die *Bandbreite*) ist, umso mehr Zeichen können der Welle pro Zeiteinheit durch Modulation aufgeprägt und durch sie transportiert werden. Infolgedessen setzen sich immer mehr die sehr hochfrequenten Lichtwellen als Träger durch. Man ist heute in der Lage, Licht mit Hilfe von *Lichtleitern* über große Entfernungen und "um viele Ecken" dem Empfänger zuzuleiten. *Die Anzahl der Binärzeichen oder Bits, die ein Binärkanal pro Sekunde fehlerfrei übertragen kann, heißt (nicht sehr treffend)* ***Kanalkapazität***.

Nach diesen kurzen Bemerkungen über die Entwicklungen zur Erhöhung der Kapazität von Nachrichtenkanälen wenden wir uns nun dem ersten oben genannten Faktor zu, der die Kommunikationsgeschwindigkeit mitbestimmt, der *semantischen Dichte*. Im vorangehenden Kapitel ist die semantische Verdichtung durch Begriffsbildung und in Kap.5.3 durch Syntaxregeln behandelt worden. In beiden Fällen wird

die semantische Verdichtung durch effektiveres *primäres* Codieren erreicht, d.h. dadurch dass mehr Ideme durch weniger Zeichenrealeme artikuliert werden. Jetzt interessieren wir uns dafür, wie Nachrichten, also bereits extern codierte Ideme, durch *Umcodierung* (durch *sekundäres* Codieren) verkürzt werden können, ohne dabei Semantik zu verlieren. Es handelt sich um eine rein *syntaktische Verdichtung*, d.h. um eine Erhöhung der Codierungseffizienz durch Umcodieren. Den Codierungsaufwand des Sekundärcodes zu dem des Primärcodes nennen wir **Umcodierungseffizienz**. Im Weiteren ist unter Codieren stets das sekundäre Codieren (Umcodieren) eines Zeichenrealems zu verstehen.

Je höher die Codierungseffizienz ist, umso kürzer ist das einem bestimmten Idem zugeordnete Zeichenrealem. Erhöhung der Codierungseffizienz bewirkt primär eine syntaktische und sekundär eine *semantische* Verdichtung. Ein Beispiel dafür ist der Übergang von der römischen zur arabischen Zahlendarstellung. Beim Umcodieren erniedrigt sich die mittlere Anzahl der Ziffern pro Zahl.

Soll eine arabische Zahl über einen Binärkanal übertragen werden, muss sie binär umcodiert werden, z.B. durch Übergang zu *Dualzahlen* oder dadurch, dass jede Ziffer durch ein zu vereinbarendes Binärwort ersetzt wird. Wenn eine Nachricht über einen Binärkanal übertragen werden soll, muss die gesamte Nachricht binär umcodiert werden, im einfachsten Fall dadurch, dass jedem Zeichen ein *Binärwort* zugeordnet wird, wie beispielsweise durch das *Morsealphabet*.

Beim Umcodieren lässt sich die Codierungseffizienz dadurch erhöhen, dass für häufiger auftretende Zeichen (Buchstaben), z.B. für das “e” und “n” im Deutschen, kürzere Codewörter reserviert werden. Umgekehrt kann man durch gezielte Verlängerung der Codewörter, durch Hinzunahme zusätzlicher Bits (durch Erhöhung der *Redundanz*, s.u.) Übertragungsfehler erkennen und die Zuverlässigkeit erhöhen, am einfachsten dadurch, dass jedes Zeichen wiederholt wird. Wo liegt der optimale Kompromiss zwischen Kürze und Sicherheit?

21 Die Technik verlangte exakte Antworten auf die gestellten Fragen. Auf der Grundlage der Wahrscheinlichkeitsrechnung hat Claude Shannon eine Theorie entwickelt, welche die gesuchten Antworten liefert. Shannon selbst hat seine Theorie *“Kommunikationstheorie”* genannt. Leider wurde sie später in *“Informationstheorie”* umbenannt[3]. Diese Bezeichnung führte - ähnlich wie die Bezeichnung “Kybernetik” - zu manchen Missverständnissen.

Voraussetzung für den Aufbau einer Theorie war eine rigorose Idealisierung und zwar die vollständige Abstraktion vom semantischen Aspekt des Informationsbegriffs, sodass nur der syntaktische Aspekt übrig blieb. Wenn im Rahmen der “Informationstheorie” von “Information” und “Informationsinhalt” die Rede ist, handelt es sich stets nur um den syntaktischen Aspekt der Codierung, um “syntakti-

3 Gestraffte Darstellungen der Informationstheorie findet der Leser z.B. in [Kreß 77] oder [Werner 95].

sche Information" oder "syntaktischen Informationsinhalt". Der Informationsbegriff der Informationstheorie entspricht also nicht dem Informationsbegriff dieses Buches.

Auf ein Resultat der Informationstheorie, das hinsichtlich der Codierungseffizienz von besonderer Bedeutung ist, soll näher eingegangen werden, auf den shannonschen Codierungssatz. Der Satz gibt die untere Grenze für die Länge einer Nachricht bei optimaler binärer Codierung an. Für den bereits erwähnten Fall, dass den Alphabetzeichen einer Nachrichtenquelle (den Zeichen vor der Umcodierung) binäre Codewörter zugeordnet werden, und für sehr lange Mitteilungen lautet der Satz: *Die* ***minimale mittlere Codewortlänge*** l_{min} *pro Alphabetzeichen der Quelle, die sich durch optimale binäre Codierung erreichen lässt, ist gleich der* ***Entropie*** *H der Quelle*, formal notiert:

$$l_{\text{min}} = H = \Sigma p_i \text{ld}(1/p_i) \tag{5.5}$$

dabei bezeichnet p_i die Auftrittswahrscheinlichkeit des i-ten Quellalphabetzeichens. Die Summe ist über alle i zu nehmen. Mit ld ist der *Duallogarithmus*, also der Logarithmus zur Basis 2 bezeichnet. Man beachte, dass (5.5) eine Mittelwertbildung der ld$(1/p_i)$-Werte darstellt. Wir werden Formel (5.5) nicht beweisen, jedoch versuchen, sie plausibel zu machen. Der Anschaulichkeit halber gehen wir von folgender Aufgabenstellung aus, die zwar etwas konstruiert, dafür aber leicht zu verstehen und zu durchschauen ist.

Aus einem Quellalphabet mit B Buchstaben werden sämtliche möglichen Ketten, sog. Quellwörter, der Länge L_{Q} gebildet. B sei eine ganzzahlige Potenz von 2, z.B. B=32. Die Quellwörter sollen nun in möglichst kurze binäre Zielwörter einheitlicher Länge umcodiert werden. Die notwendige Zielwortlänge L_{Z} lässt sich aus der Bedingung berechnen, dass die Anzahl der Zielwörter $2^{L_{\text{Z}}}$ gleich der Anzahl der Quellwörter $B^{L_{\text{Q}}}$ sein muss. Durch Logarithmieren erhält man für die Zielwortlänge $L_{\text{Z}} = L_{\text{Q}}\text{ld}B$. Für die mittlere Codewortlänge pro Quellalphabetzeichen $l = L_{\text{Z}}/L_{\text{Q}}$ ergibt sich damit

$$l = \text{ld}B. \tag{5.6}$$

Bevor wir andeuten, wie man von (5.6) zu (5.5) gelangt, wollen wir der Größe ld$B$, die auch *Entscheidungsgehalt* genannt wird, eine anschaulichere Bedeutung geben. Es gibt ein bekanntes Ratespiel zu Zweien, bei dem der Eine, der *Wissende*, sich ein Objekt ausdenkt, das der Andere, der *Fragende*, durch Fragen erraten muss, auf die der Wissende nur mit "ja" oder "nein" antworten darf. Wenn das zu erratende Objekt ein Element aus einer beiden Spielern bekannten Menge ist, kann der Ratende folgende optimale Ratestrategie anwenden. Er teilt die Menge in zwei gleiche Teilmengen und fragt, ob das Objekt Element der einen (genau zu beschreibenden) Teilmenge ist. Aus der Antwort (Ja oder Nein) erfährt er, in welcher Teilmenge sich

das Objekt befindet. Diese teilt er nun ihrerseits in zwei gleiche Teilmengen und fährt so fort, bis er das zu erratende Objekt identifiziert hat.

Wenn die Gesamtmenge $m = 2^n$ Elemente besitzt, muss er mindestens $\mathrm{ld}m = n$ mal fragen, und der Befragte muss ebenso viele *Entscheidungen* treffen. In diesem Sinne sagt man, dass der Entscheidungsgehalt der Identifikation eines Elementes aus einer Menge von m Elementen den Wert $\mathrm{ld}m$ bit besitzt. Die Folge der Antworten kann als Binärwort aufgefasst werden, welches das Objekt identifiziert ("codiert"). Es besitzt die Länge n Bit (man beachte die unterschiedlichen Maßeinheiten bit bzw. Bit, die gleich erklärt werden). Um nach dieser Strategie einen bestimmten Buchstaben des Quellalphabets zu erraten, muss man, falls B eine Potenz von 2 ist, $\mathrm{ld}B$ mal raten, d.h. von dieser Länge ist das "codierende Binärwort" der Antworten. Dieser Wert stimmt mit dem von l in (5.6) überein. Wenn B keine Potenz von 2 ist, wenn also $\mathrm{ld}B$ keine ganze Zahl ist, ergibt sich als Frageanzahl die nächsthöhere ganze Zahl. Dennoch wird als *Entscheidungsgehalt* der Wert von $\mathrm{ld}B$ bezeichnet mit der "Maßeinheit" bit. Demgegenüber ist ein Wert mit der "Maßeinheit" Bit stets ganzzahlig (ein Bit ist ein Binärzeichen).

Betrachtet man nun noch einmal die Formel (5.6), erkennt man, dass die Entropie H als mittlerer *individueller* Entscheidungsgehalt aufgefasst werden kann, wobei der individuelle Entscheidungsgehalt eines Zeichens in Übereinstimmung mit (5.6) den Wert $\mathrm{ld}(1/p_i)$ besitzt.

Offenbar ist die Entropie der mittlere Entscheidungsgehalt, d.h. die mittlere minimale binäre Codewortlänge pro Zeichen einer (sehr langen) Nachricht, in der die Alphabetzeichen mit unterschiedlicher Häufigkeit auftreten. Das genau beinhaltet der shannonsche Codierungssatz (5.5), dessen Gültigkeit durch unsere Überlegungen zwar plausibel gemacht, aber nicht exakt bewiesen worden ist. Doch lässt er sich beweisen, sodass es gerechtfertigt ist, die Entropie als verallgemeinerten Entscheidungsgehalt zu bezeichnen. Häufig wird die Entropie *syntaktischer Informationsgehalt* oder kurz **syntaktische Information** genannt.

Um den Codierungssatz (5.5) exakt zu beweisen, muss in Analogie zur Herleitung der Formel (5.6) die Anzahl aller Zeichenketten der Länge L berechnet werden, jetzt aber unter der Nebenbedingung, dass die einzelnen Buchstaben mit den relativen Häufigkeiten p_i auftreten. Der erhaltene Wert (bei Gleichverteilung ist er gleich B^L) ist zu logarithmieren und durch L zu teilen. Lässt man schließlich L gegen Unendlich gehen (dabei werden die Häufigkeiten zu Wahrscheinlichkeiten), erhält man den angegebenen Wert für die Entropie. Je stärker sich die p_i-Werte voneinander unterscheiden, umso kleiner ist die Entropie und umso größer ist der Entscheidungsgehalt und die mittlere Codewortlänge und umso mehr lässt sich die Codierungseffizienz durch optimale Umcodierung steigern. Bei *Gleichverteilung* (alle p_i sind einander gleich) nimmt *die Entropie den Maximalwert* $H_{\max} = \mathrm{ld}B$ an.

Die Differenz $H_{\max} - H$ gibt an, wieviele Binärzeichen pro Quellzeichen im Mittel eingespart werden können, wenn die Nachrichten einer Quelle mit der Entropie H optimal umcodiert werden. Die Differenz geteilt durch $H_{\max}$ wird *Redundanz* genannt

(vom englischen Wort für “Überfluss”,"Überzähligkeit"). Sie ist ein Maß für die erreichbare prozentuale Umodierungseffizienz. Die Redundanz der deutschen Sprache beträgt etwa 73%. Zwar vermindert Redundanz die semantische Dichte, doch hat sie, wie bereits erwähnt, auch Vorteile. Je größer sie ist, umso eher lässt sich erraten oder rekonstruieren, was mit einer fehlerhaften (z.B. bruchstückhaften) Nachricht gemeint ist und umso mehr Fehler lassen sich korrigieren.

Man beachte, dass Formel (5.5) *kein* Rezept dafür liefert, *wie* optimale Codierung zu erreichen ist. Es sind viele praktische Codierungsmethoden entwickelt worden, auf die wir jedoch nicht eingehen. Einige von ihnen findet man in [Kreß 77].

Die Bezeichnung *Entropie* ist aus der Physik und zwar aus der Thermodynamik übernommen, denn die thermodynamische Entropie berechnet sich (bis auf einen konstanten Faktor) nach der Formel (5.5). Aus diesem Grunde werden wir im Weiteren zwischen *thermodynamischer* Entropie und *informatischer* Entropie oder *Informationsentropie* unterscheiden.

Die Bedeutung der thermodynamischen Entropie liegt darin, dass ihr Wert für abgeschlossene Systeme (Systeme ohne Energieaustausch mit der Umgebung) ständig zunimmt, bis sie ihren Maximalwert erreicht hat.

Die Tatsache, dass sich die thermodynamische und die informatische Entropie nach der gleichen Formel berechnen, hat einen formalen oder besser syntaktischen, jedoch keinen inhaltlichen, keinen kausalen Grund. In beiden Fällen hat man es mit ein und demselben *kombinatorischen* Problem zu tun. Das soll an einem einfachen Beispiel für die Berechnung der thermodynamischen Entropie veranschaulicht werden. Die folgenden Überlegungen gehen auf LUDWIG BOLTZMANN zurück, dem es gelang, die Aussagen der Thermodynamik aus den Vorstellungen der kinetischen Gastheorie abzuleiten, d.h. sie mikroskopisch zu interpretieren.

Man stelle sich ein geschlossenes 1-Liter-Gefäß vor, in dem sich N gleiche Gasmoleküle befinden. Gedanklich unterteilen wir das Gesamtvolumen in B=10^6 Zellen von je 1mm^3 Inhalt und machen eine mikroskopische Momentaufnahme, auf der zu erkennen ist, in welcher Zelle sich jedes einzelne Molekül befindet; wir nehmen also an, dass sich die Moleküle voneinander unterscheiden lassen. Die Aufnahme zeigt einen sogenannten *Mikrozustand* des Gasvolumens. Wir zählen nun die Moleküle in jeder Zelle, in der i-ten Zelle seien es n_i. Die Gesamtheit der n_i beschreibt die *Dichteverteilung* des Gases. 22

Eine bestimmte Dichteverteilung kann offensichtlich durch unterschiedliche Mikrozustände realisiert werden. Vertauschen nämlich zwei Moleküle ihre Plätze, so ändert sich die Dichteverteilung nicht, während der Mikrozustand in einen anderen übergeht, vorausgesetzt, die beiden Moleküle befinden sich in unterschiedlichen Zellen. Befinden sie sich dagegen in ein und derselben Zelle, so ändert sich bei ihrer Permutierung weder die Dichteverteilung noch der Mikrozustand. Das ist zu beachten, wenn man die Anzahl aller möglichen Vertauschungen (Permutationen) zählt, also aller Mikrozustände, die eine gegebene Dichteverteilung realisieren. Nach den Regeln der Kombinatorik ist die Anzahl gleich $N!/(n_1!n_2!...n_B!)$. Dabei ist $N!$ die

Anzahl aller möglichen Permutationen[4], $n_i!$ die Anzahl der (nicht zu berücksichtigenden) Permutationen in der i-ten Zelle und B ist die Anzahl der Zellen. Im Nenner steht das Produkt aller $n_i!$.

Auf das gleiche kombinatorische Problem stößt man, wenn man fragt, wie viele unterschiedliche Buchstabenketten (Wörter) sich aus N Buchstabenexemplaren bilden lassen, also wie viele verschiedene Wörter der Länge N. Man versetze sich in die Lage eines Schriftsetzers, dem von jedem Alphabetbuchstaben, im Weiteren *Buchstabentyp* genannt, mehrere Exemplare zur Verfügung stehen, und zwar vom i-ten Buchstabentyp n_i Exemplare. Er verkettet nun alle Buchstabenexemplare zu einem Wort der Länge N. Sodann bildet er neue Wörter durch Permutation, d.h. durch Vertauschen einzelner Buchstabenexemplare. Zählt man nun alle unterschiedlichen Wörter, die sich auf diese Weise durch fortgesetztes Permutieren bilden lassen, hat man - ebenso wie beim Vertauschen der Gasmoleküle - zu beachten, dass sich nur dann ein neues Wort ergibt, wenn *unterschiedliche* Buchstaben miteinander vertauscht werden. Werden gleiche Buchstaben vertauscht, bleibt das Wort erhalten. Demzufolge berechnet sich die Anzahl der unterschiedlichen Wörter bei vorgegebenen Buchstabenhäufigkeiten nach derselben Formel, nach der sich auch die Anzahl der Mikrozustände bei gegebener Dichteverteilung berechnet. (Hier liegt der Grund für die syntaktische Übereinstimmung der Formeln für die thermodynamische und die informatische Entropie.) Die Anzahl ist gleich $N!/(n_1!n_2!...n_B!)$. Für sehr große N (sehr viele Buchstabenexemplare bzw. Moleküle) und sehr große n_i (sehr viele Exemplare pro Buchstabentyp bzw. Moleküle pro Zelle) lassen sich die Fakultäten nach der Stirlingschen Formel in Exponentialausdrücke überführen. Durch Logarithmieren und Übergang von Häufigkeiten (Besetzungszahlen) zu Wahrscheinlichkeiten (Dividieren durch N) ergibt sich die in (5.5) angegebene Summe. In der Physik ist es üblich, nicht den dualen, sondern den natürlichen Logarithmus zu verwenden. Außerdem wird die Summe mit einem dimensionierten konstanten Faktor multipliziert, sodass die Entropie diejenige Dimension erhält (Energie durch Temperatur), mit der sie in der Thermodynamik ursprünglich eingeführt worden ist. Damit ist gezeigt, dass die formelmäßige Übereinstimmung von informatischer und thermodynamischer Entropie eine formale und keine inhaltliche (physikalische) Ursache hat.

Wir sind so ausführlich auf den Begriff der Entropie eingegangen, um der Gefahr vorzubeugen, den Begriff der informatischen Entropie oder des syntaktischen Informationsgehalts mit dem Begriff der thermodynamischen Entropie zu identifizieren und z.B. zu schlussfolgern, dass bei der Übertragung einer Nachricht mit einem bestimmten syntaktischen Informationsgehalt (einer bestimmten Menge informatischer Entropie) die entsprechende Menge thermodynamischer Entropie (unter Berücksichtigung einer dimensionierten Verhältniszahl) vom Sender an den Empfänger

4 Zur Bedeutung des Ausrufungszeichens als Operationssymbol siehe (8.12).

übertragen wird. Das wäre ein Trugschluss. Eine "*wirkliche*", d.h. physikalisch *wirksame* Beziehung zwischen physikalischer und informatischer Entropie kann nur über die Trägersysteme bestehen, die sprachlich miteinander kommunizieren. Denn nur diese, nicht aber die von ihnen produzierten Sprachen als solche, gehorchen den Gesetzen der Thermodynamik.

Ohne Frage hat der Umstand, dass informationelle Systeme den Gesetzen der Thermodynamik gehorchen, Konsequenzen hinsichtlich ihrer Entstehung und Verhaltensweise, beispielsweise hinsichtlich der phylogenetischen und ontogenetischen Entwicklung des menschlichen Gehirns und seines Funktionierens. Wenn man die Probleme des sprachlichen Modellierens konsequent von der Physik her, das bedeutet *sub*symbolisch, angeht und wenn es gelingt, auf diesem Weg einen Informationsbegriff zu definieren und eine Wissenschaft vom aktiven sprachlichen Modellieren zu entwickeln, könnte eine *Naturwissenschaft* "Informatik" entstehen. In ihr wird die *thermodynamische* Entropie zu den Grundbegriffen gehören. Erste Schritte auf diesem Wege sind bereits getan[5].

5 Siehe z.B. [Ebeling 82], [Nicolis 87], [Ebeling 91], [Ebeling 98].

6 Arbitrarität und Zirkularität

Zusammenfassung

Das Artikulieren (Zuordnen von Zeichenrealemen zu Idemen) ist seiner Natur nach *arbiträr*, d.h. es ist beliebig in dem Sinne, dass es durch keine Naturgesetze erzwungen wird. Die Arbitrarität hat zwei Wurzeln, die individuelle Freiheit einer Person, sich auszudrücken, und die kollektive Freiheit einer Sprachgemeinschaft, Zeichenrealeme mit bestimmten Bedeutungen zu vereinbaren.

Unter dem Begriff der Zirkularität werden beliebige Rückwirkungen realer Objekte auf sich selbst (*operationale Zirkularität*) und beliebige Rückbezüglichkeiten sprachlicher Ausdrücke auf sich selbst (*referenzielle Zirkularität*) zusammengefasst. Zirkularitäten können sinnvoll oder sinnlos sein, sie können widerspruchsfrei oder widersprüchlich sein. Infolge referenzieller Zirkularität kann eine Aussage *unentscheidbar* werden, dass heißt, es kann nicht bewiesen werden, ob die Aussage richtig oder falsch ist.

Die formale Untersuchung der Entscheidbarkeit von Aussagen hat Kurt Gödel 1931 zu seinem *Unvollständigkeitssatz* geführt: In einem ausreichend komplexen Kalkül lassen sich Sätze formulieren, die zwar richtig, in dem betreffenden Kalkül aber nicht entscheidbar sind.

6.1 Arbitrarität des Artikulierens

Die Gedanken sind frei.
Wer kann sie erraten?
Volkslied

Die Gedanken sind frei, nicht nur, weil man denken kann, was man will, sondern auch, weil man sagen kann, was man will, sodass ein anderer aus dem, was man sagt, nicht unbedingt schließen kann auf das, was man denkt. Mit dieser Feststellung setzen wir die Überlegungen des Kapitels 2 zum Thema “Gedanke und Sprache” fort. Denken und Sprechen sind dem freien Willen untergeordnet, zumindest im vollbewussten Zustand. Daraus ergeben sich erfreuliche, aber auch recht problematische Freiheiten. Ob und wie ein Mensch das, was er denkt, in Worten wiedergibt, ist ihm durch kein ihm bekanntes Naturgesetz vorgeschrieben. Er kann einen Gedanken (ein Idem) in verschiedenen Sprachen und in ein und derselben Sprache auf unterschiedliche Weise artikulieren. Er muss sich allerdings, um verstanden zu werden, an die Sprachgewohnheiten und Konventionen seiner Umgebung halten. Für diese Unbestimmtheit verwenden wir das Wort *Arbitrarität*.

Unter ***Arbitrarität*** *des Artikulierens verstehen wir das Fehlen eines zwangsläufigen, durch Naturgesetze eindeutig diktierten Zusammenhanges zwischen einem Idem*

und dem ihm zugeordneten Zeichenrealem. Die Arbitrarität hat zwei Wurzeln, die Willensfreiheit des einzelnen und die Willkür sprachlicher Gewohnheiten und Konventionen innerhalb sozialer Gruppen.

In Kap.2 hatten wir festgestellt, dass Sprechen ursprünglich dem Austausch interner Modelle der Welt zum Zwecke der Kooperation dient. Aussagesätze sind also Modellaussagen über die Welt. Es müssen wahre Aussagen sein, sonst würden sie die Kooperation stören. Unter diesem Aspekt erscheint die Freiheit des Artikulierens in ziemlich fragwürdigem Licht. Man kann sich etwas ausdenken, was keine Entsprechung in der Realität hat, und kann das auch sagen. Man kann aber auch etwas anderes sagen, als man denkt. Mit anderen Worten, man kann phantasieren und kann schwindeln. Zwischen beidem besteht ein gleitender Übergang. Je nachdem, in welchem Maße man seine Zuhörer über den Wahrheitsgehalt einer Aussage im Unklaren lässt, ist es mehr Phantasie oder mehr Schwindel, was man ihm "aufbindet". In beiden Fällen macht man sich die Freiheit des Artikulierens zunutze. Anders ist es, wenn man irrt, also sich nicht bewusst etwas Falsches ausdenkt. Doch angesichts der Möglichkeit FREUDscher Verdrängung kann auch der Übergang vom Irrtum zur Lüge ein gleitender sein.

Schließlich ist es jedem freigestellt, ganz bewusst blanken Unsinn zu reden. Das kann beim Interpretierer Verständnislosigkeit, Unwillen oder Amüsement hervorrufen, wie z.B. folgende Frage: "Was ist der Unterschied zwischen einem Raben?" und die Antwort: "Er hat zwei gleichlange Beine, besonders das rechte." Die Arbitrarität des Artikulierens lässt so etwas zu. Und es hat durchaus seine soziale Bedeutung, ebenso wie das Phantasieren und das Schwindeln.

Nach dieser Abschweifung präzisieren wir die zentrale Aussage. Artikulieren ist seiner Natur nach arbiträr. Die Arbitrarität hat zwei Ursachen, die individuelle Freiheit, etwas zu sagen wie und wann man will, und die kollektive Freiheit zu vereinbaren, welche Symbole syntaktisch und semantisch wie zu verwenden sind, m.a.W. was wie codiert wird. Im Falle natürlicher Sprachen ist diese Festlegung im wesentlichen ein Ergebnis der kulturellen Evolution, im Falle künstlicher Sprachen beruht sie auf Vereinbarung. Das gilt insbesondere für Programmiersprachen. Das muss nicht so bleiben. Es ist immerhin denkbar, dass die Computertechnik in eine Phase eintritt, in der sie selber aktiv an der kulturellen Evolution teilnimmt, und dass Computersprachen unter Mitwirkung des Computers selber evolutionieren und nicht allein nach Maßgabe des Menschen.

6.2 Referenzielle Zirkularität

Eine andere problematische Freiheit ist die Möglichkeit des rückbezüglichen Denkens, Schließens und Artikulierens. Sie tritt in unterschiedlichen Formen auf und kann zu merkwürdigen inneren Widersprüchlichkeiten und zu paradoxen Aussagen führen, durch die sich seit eh und je scharfsinnige Philosophen herausgefordert

fühlen. Deren Markenzeichen ist daher die Schlange, die sich selbst von ihrem eigenen Schwanz her verschlingt. Dabei wird ein in sich "verschlungener" Denkprozess durch einen in sich "verschlungenen" realen Prozess versinnbildlicht. Das Gemeinsame ist der zirkuläre Charakter. Im Weiteren wird das Wort Zirkularität (und entsprechend die Wörter Zirkel und zirkulär) in folgendem Sinne verwendet.

*Alle Arten von Rückbezüglichkeit und Rückwirkung auf sich selbst (Reflexivität, Rekursivität, Rückkopplung) werden unter dem Begriff "***Zirkularität***" zusammengefasst.* Auf einige Arten der Zirkularität soll näher eingegangen werden. Wir beginnen mit der Selbstbenennung. Zunächst definieren wir das Wort *Referenzieren.*

Unter **Referenzieren** *versteht man das Verweisen auf ein Objekt der Realität oder des Denkens (auf ein Realem oder Idem), durch Nennung eines Stellvertreters, eines Namens oder Pronomens, oder durch Angabe seines Ortes (z.B. postalische Adresse, bibliographische Angaben, Speicheradresse).* In der Computer-IV ist Referenzieren das Verweisen auf ein Zeichenrealem mittels eines anderen Zeichenrealems.

Wir wollen die Konsequenzen der Freiheit des Referenzierens genauer untersuchen. Dieser Freiheit sind an sich keinerlei natürliche oder logische Grenzen gesetzt. Sie erlaubt auch falsches, sinnloses und sogar widersinniges Referenzieren. Sie erlaubt selbstverständlich auch das Selbstreferenzieren. Wir sprechen dann von **referenzieller Zirkularität**. Diese liegt beispielsweise vor, wenn ein Artikulierer von sich selbst spricht, wenn ein Satz etwas über sich selbst aussagt oder wenn ein Wort sich selbst "meint", wie der zweite der folgenden Sätze demonstriert.

Hans ist ein Junge von 4 Jahren.

Hans ist eine Zeichenkette von 4 Buchstaben.

Im ersten Satz verweist das Zeichenrealem "Hans" auf eine Person, im zweiten auf sich selbst. Betrachten wir nun die folgenden Sätze.

Dieser Satz ist ein Aussagesatz.

Dieser Satz ist ein Fragesatz.

Das Demonstrativpronomen "dieser" zeigt die Rückbezüglichkeit an. Beide Sätze sind erlaubt. Aber der zweite Satz ist falsch, während der erste wahr ist. Diese Eindeutigkeit ist nicht immer gegeben, wie der zweite der folgenden Sätze zeigt.

Dieser Satz ist wahr.

Dieser Satz ist falsch. 1

Wenn der erste Satz wahr ist, so stimmt das mit dem, was er über sich selbst aussagt, überein. Der Satz ist wahr, allerdings inhaltslos. Wenn der zweite Satz wahr ist, dann ist er nach dem, was er über sich selber aussagt, falsch, d.h. der Satz ist falsch. Wenn der Satz falsch ist, dann ist das, was er über sich selbst aussagt, richtig; der Satz ist also richtig. Es liegt ein Widerspruch, eine *Antinomie* vor und es lässt sich nicht entscheiden, ob der Satz wahr oder falsch ist, man sagt: der Satz ist nicht

entscheidbar. Wie man sieht, *kann referenzielle Zirkularität zu Antinomien und zu nichtentscheidbaren Aussagen führen.*

Ein referenzieller Zirkel kann sich auch über mehrere Objekte erstrecken, wie die beiden folgenden Sätze zeigen, die wir mit A und B benennen.

2 Satz A: “Satz B ist falsch.”

Satz B: “Satz A ist falsch.”

Wie man sich leicht überzeugt, kommt es nur dann zu *keinem* Widerspruch, wenn einer der beiden Sätze wahr, der andere falsch ist. Einer ähnlichen Situation werden wir in Kap.9.4 [9.20] im Zusammenhang mit dem Flipflop begegnen.

Schon im Altertum haben sich die Menschen mit Antinomien herumgeschlagen. Die berühmteste ist die *Lügnerantinomie* in der Form “Alle Kreter lügen.” Dieser Satz ist an sich nicht widersprüchlich; er wird es aber im Munde eines Kreters. Doch auch dann stellt er genau genommen keine Antinomie dar, denn er kann entschieden werden. Er ist nämlich falsch, sowohl, wenn der Kreter lügt, als auch, wenn er nicht lügt (vgl. [Vollmer 88] Bd.1).Dagegen ist der Satz “Jetzt lüge ich” eine Antinomie und nicht entscheidbar.

Zuweilen werden Sätze, die genau genommen Antinomien sind, nicht als solche empfunden, sondern “automatisch” widerspruchsfrei interpretiert. Ein Beispiel ist Platons berühmter Ausspruch “Ich weiß, dass ich nichts weiß”. Der Widerspruch kann durch eine kleine Erweiterung aufgehoben werden: “Ich weiß, dass ich sonst nichts weiß”, oder ausführlicher: “Ich weiß, dass ich nichts weiß ausser dem einen, dass ich nichts weiß”.

Der Verdacht liegt nahe, dass all diese Komplikationen die Folge der Ungenauigkeit des Sprechens oder auch des Denkens sind, dass sich Antinomien und die Unentscheidbarkeit von Sätzen durch semantische Objektivierung vermeiden lassen und dass nach Übergang zu formaler Semantik Antinomien ausgeschlossen werden können. Dieser Verdacht hat sich als falsch erwiesen. Die Unentscheidbarkeit von Sätzen ist unvermeidlich. Das ergibt sich aus dem *ersten Unvollständigkeitssatz*[1], den Kurt Gödel bewiesen hat [Gödel 31]. Nach diesem Satz kann es in einem widerspruchsfreien formalen System wahre Sätze geben, die in dem System nicht entscheidbar sind, die sich also mit den Mitteln des formalen Systems weder beweisen noch widerlegen lassen. Wir kommen darauf in Kap.8.5 zurück.

6.3 Operationale Zirkularität

In der “Schlangenmetapher” wird die Vorliebe spitzfindiger Denker für rück*bezügliche* Zirkularität des Denkens durch rück*wirkende* Zirkularität einer Tätigkeit

1 Siehe z.B. [Schreiber 84], [Schöning 95]

versinnbildlicht. Dieser zweiten Art von Zirkularität wenden wir uns nun zu. Wir nennen sie **operationale Zirkularität**. Dabei ist zwischen *Prozesszirkularität* und *Operand-Operator-Zirkularität* zu unterscheiden. In der Schlangenmatapher handelt es sich um eine Rückwirkung, bei der das Subjekt der Tätigkeit (des Verschlingens) gleichzeitig deren Objekt ist. In mehr technischer Redeweise lässt sich die Situation folgendermaßen charakterisieren. *Der Operator, der die Operation ausführt, ist mit dem Operanden, an dem sie ausgeführt wird, identisch.* In diesem Sinne sprechen wir von **Operand-Operator-Zirkularität**.

3

Die Widersinnigkeit der Schlangenmetapher könnte vermuten lassen, dass Operand-Operator-Zirkularität immer etwas Widersinniges ist. Ganz einfache Beispiele zeigen aber, dass dies nicht der Fall ist. Der genannte Typ von Zirkularität liegt z.B. vor, wenn ein "Auto-mobil" fährt, oder wenn ich mich wasche, wobei das Reflexivpronomen "mich" die Zirkularität anzeigt. Der Zirkel kann sich auch über mehrere Operatoren schließen, beispielsweise wenn eine Hand die andere wäscht. Wenn dagegen eine Hand die andere zeichnet, wie in Bild 6.1 dargestellt, wird der Zirkel widersinnig, d.h. er ist zwar darstellbar, aber in Wirklichkeit unmöglich. Darin zeigt sich die Arbitrarität des "Artikulierens mittels *Zeichnen*", ähnlich der des Artikulierens mittels *Zeichen*. Bild 6.1 stammt von Maurits Cornelis Escher[2].

In einem Operand-Operator-Zirkel relativieren sich die Begriffe Operand und Operator. Diese Relativierung ist nicht an die Existenz eines Zirkels gebunden, wie am Beispiel einer Werkzeugmaschine deutlich wird. Aus der Sicht der Maschine ist das produzierte Werkzeug Operand, aus der Sicht der Operation, die mit dem Werkzeug ausgeführt wird, ist es Operator. Ein Zirkel läge vor, wenn die Maschine sich selbst produzieren würde. Es gibt zwar Selbstvernichtung, jedoch keine Selbstproduktion, sondern nur *Selbstreproduktion,* bei der ein Operator einen sich selbst gleichen aber nicht identischen Operator "der *nächsten Generation*" produziert. Hier kann man von Operand-Operator-Zirkularität im Sinn von Wiederholung des gleichen Prozesses sprechen, wobei der Operand eines Zirkeldurchlaufs zum Operator

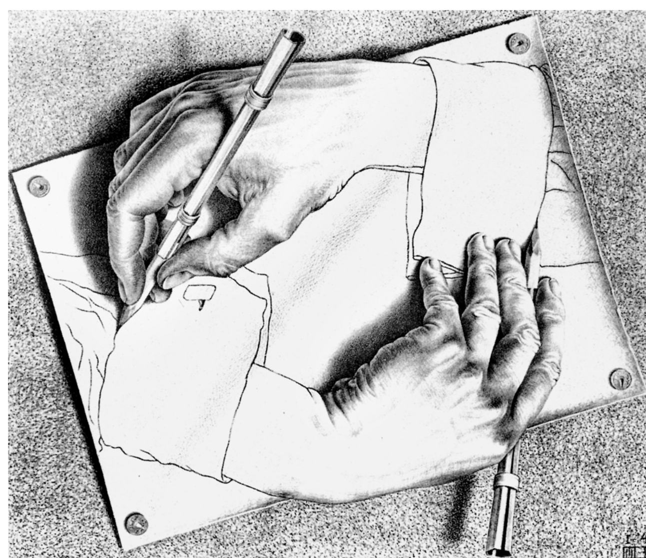


Bild 6.1 Widersinniger Operand-Operator-Zirkel

2 Aus [Ernst 85]

des nächsten Durchlaufs (der nächsten Generation) wird. Selbstreproduktion ist mit Informationsübergabe von Generation zu Generation verbunden, wie wir in Kap.1 [1.2] in Verbindung mit der Evolution festgestellt hatten. Doch gilt das auch für Selbstreproduktion künstlicher Systeme, wie JOHN VON NEUMANN gezeigt hat [Neumann 66].

Eine Zirkularität liegt auch dann vor, wenn ein Operator sich selber aufruft. Dann spricht man von *Zyklus*. Wenn dabei ein und derselbe Operand bearbeitet wird, liegt Mehrfachverarbeitung vor, die im Falle eines informationellen Operators *Iteration* genannt wird. Dabei handelt es sich nicht um Operand-Operator-Zirkularitäten. Die Rückwirkung eines Operators auf sich selbst und speziell den Selbstaufruf während einer Operationsausführung (eines Prozesses), nennen wir **Prozesszirkularität**. Der Begriff der Prozesszirkularität kann in unterschiedlichen spezielleren Bedeutungen verwendet werden. Ein Regelungstechniker wird zuerst vielleicht an einen Regelkreis denken, ein Techniker allgemein an eine Rückkopplungsschleife, über welche Signale oder Informationen vom Ausgang eines Gerätes auf seinen Eingang zurückgegeben werden. Rückkopplungsschleifen sind in informationellen Systemen auf Schritt und Tritt anzutreffen, sowohl in der Hardware als auch in der Software, wo sie allerdings nicht *"verlötet"*, sondern *beschrieben (sprachlich modelliert)* werden.

Wir kehren noch einmal zur Operand-Operator-Zirkularität und speziell zur Relativität der Begriffe Operator und Operand zurück. Diese Relativität ist ein charakteristisches Merkmal der Computer-IV. In Anlehnung an den mathematischen Sprachgebrauch kann ein Programm als Operator aufgefasst werden, der die Eingabedaten (Operanden) verarbeitet, auf die es "angewandt" wird. Andrerseits kann ein Programm von einem anderen Programm bearbeitet, z.B. editiert, korrigiert oder übersetzt werden, also die Rolle eines Operanden spielen. Ein Programm, das übersetzt und anschließend ausgeführt wird, ist zuerst Operand und dann Operator. Wenn ein Übersetzerprogramm sich selbst übersetzt, liegt Operand-Operator-Zirkularität vor. Das ist durchaus möglich und weit weniger überraschend als die sich selbst bearbeitende Maschine. Denn Zeichenrealeme, also auch Programme lassen sich kopieren, und bei der Selbstübersetzung übersetzt ein Exemplar eines Programms ein anderes Exemplar desselben Programms.

4 Wenn man in einem Operand-Operator-Zirkel den Operator als Ursache und den Operanden bzw. dessen Veränderung als Wirkung ansieht, gelangt man zu dem *Ursache-Wirkung-Zirkel*, von dem in Kap.4.2 [4.6] im Zusammenhang mit den newtonschen Bewegungsgleichungen die Rede war.

5 Schließlich sei angemerkt, dass die Informatik selber eine zirkuläre Wissenschaft ist. Ihr *Gegenstand* ist das sprachliche Modellieren und wie jede Wissenschaft bedient sie sich des sprachlichen Modellierens als *Mittel*. Das sprachliche Modellieren bildet einen Operand-Operator-Zirkel.

7 Evolution der Intelligenz

Zusammenfassung

Die natürliche Intelligenz hat sich von der Fähigkeit des Erkennens bis hin zur Fähigkeit des Rechnens entwickelt. Die künstliche Intelligenz hat sich in der entgegengesetzten Richtung entwickelt. Diese scheinbar widersprüchliche Beobachtung nennen wir das *Gegenläufigkeitsphänomen* der Evolution der Intelligenz. Das Wort *Erkennen* kann *Wiedererkennen* oder *Erkenntnisgewinnung* bedeuten.

Intelligenz ist die Fähigkeit zum sprachlichen Modellieren. Sprachliches Modellieren der Welt besteht im Finden richtiger Aussagen über die Welt. Aussagen ordnen Objekten (genauer: Denkobjekten) Merkmale zu. Richtige Aussagen können durch Deduktion, Assoziation oder Intuition gefunden werden. *Deduzieren* (Synonym zu *Ableiten*) kann in Rechnen oder Schlussfolgern bestehen. *Rechnen* ist das Ableiten von Aussagen mittels Rechenregeln im Rahmen eines Kalküls. *Schlussfolgern* durch den Menschen ist das Ableiten von Aussagen mittels nicht formalisierter Schlussregeln. Deduktion ist eine Intelligenzleistung des Bewusstseins, Assoziation und Intuition sind Intelligenzleistungen, an denen das Unterbewusstsein wesentlich beteiligt ist. *Assoziation* beinhaltet das *Wiederauffinden* bekannter Aussagen, d.h. bekannter Zuordnungen zwischen Objekten und Merkmalen, ohne bewusstes Suchen. *Intuition* beinhaltet das *Erfinden* neuer Aussagen, also neuer Zuordnungen zwischen Objekten und Merkmalen. Ihr liegen unbewusste Erfahrungen des Individuums zugrunde. Intuition, Kreativität und Phantasie sind im Kern miteinander wesensgleich. Intuition ist eine produktive, Assoziation eine reproduktive Intelligenzleistung.

Der Mensch verfügt über *Metaintelligenz*, d.h. über die Fähigkeit, das eigene sprachliche Modellieren zu modellieren und zielgerichtet weiterzuentwickeln. Im Vergleich zum Menschen besitzt der heutige Computer kaum Metaintelligenz.

Ein fundamentaler Begriff der modernen Rechentechnik ist der uralte Begriff des Algorithmus. Ein *Algorithmus* ist eine Handlungsvorschrift (Prozessbeschreibung), die angibt, in welcher Reihenfolge mehrere, als bekannt vorausgesetzte Einzelhandlungen oder Operationsschritte auszuführen sind, um ein bestimmtes Ziel zu erreichen. Die Handlungsfolge muss eindeutig und endlich sein (aus endlich vielen Schritten bestehen). Ein Operationsschritt heißt *Aktion*, wenn die Operanden der Operation explizit angegebenen sind. Eine Aktionsvorschrift heißt *Anweisung* oder *Befehl*. Ein Algorithmus, der aus Anweisungen (Befehlen) besteht, heißt *imperativer Algorithmus*.

Die Universalität des Computers beruht auf der Tatsache, dass sich jede arithmetische, analytische und logische Operation aus Inkrementier- und Vergleichsoperationen komponieren lässt und diese sich ihrerseits aus denjenigen arithmetischen und

logischen Operationen komponieren lässt, welche die ALU (Arithmetisch-Logische Unit) ausführen kann.

7.1 Deduktive, assoziative und intuitive Intelligenz.

Intelligenz hatten wir als Fähigkeit zum sprachlichen (codierenden) Modellieren definiert [3.1]. Wir werden uns nun für die Evolution dieser Fähigkeit interessieren. Geht man der Frage nach, in welcher Richtung Intelligenz evolutioniert, so kommt man zu entgegengesetzten Antworten, je nachdem, ob man die natürliche oder die künstliche Intelligenz betrachtet. Die Entwicklungsrichtungen sind einander diametral entgegengerichtet. Etwas verkürzt kann man sagen: *Die natürliche Intelligenz entwickelt sich* ***vom Erkennen zum Rechnen****, die künstliche Intelligenz* ***vom Rechnen zum Erkennen***. Diese Beobachtung nennen wir das ***Gegenläufigkeitsphänomen*** *der Evolution der Intelligenz*. Die Evolution der natürlichen Intelligenz ist phylogenetisch eine Komponente der genetischen und ontogenetisch eine Komponente der individuellen Evolution. Die Evolution der künstlichen Intelligenz ist eine Komponente der kulturellen Evolution.

Der gegenwärtige Stand der Technik lässt folgende grobe Charakterisierung der Fähigkeiten des Computers im Vergleich zu denen des Menschen zu. Der Computer kann besser *rechnen*, etwa ebenso gut *schlussfolgern*, aber schlechter *assoziieren* und *erkennen* als der Mensch. Die hervorgehobenen Wörter haben folgende Bedeutungen.

Das Wort *Erkennen* hat zwei Bedeutungen. Zum einen kann es das Erkennen eines Objekts ("Das ist Fritz") oder das Erkennen einer Klassenzugehörigkeit ("Das ist eine Linde") bedeuten. Dann sprechen wir von **Wiedererkennen** oder **Identifizieren**. Zum anderen kann "Erkennen" das Gewinnen neuer Erkenntnisse bedeuten, das "Begreifen" eines unbekannten Objekts, Phänomens oder Sachverhalts. Dann sprechen wir von Erkenntnisgewinnung. **Erkenntnisgewinnung** ist das Finden neuer Aussagen über die Welt. Die Fähigkeit des Menschen, Erkenntnisse zu gewinnen, zu sammeln und weiterzugeben ist zusammen mit der Fähigkeit zu abstrahieren und Begriffe zu bilden eine Grundvoraussetzung der kulturellen Evolution[1]. Damit eine Entwicklung stattfinden kann, müssen Erkenntnisse *sinnvolle* Aussagen sein, d.h. sie müssen in das bereits vorhandene Wissen (Modell der Welt) hineinpassen. Neue sinnvolle Aussagen (Erkenntnisse) verändern das vorhandene Modell so, dass ein umfassenderes, ein genaueres oder ein in sich konsistenteres Modell entsteht.

Rechnen *ist das Ableiten von Aussagen mittels Rechenregeln im Rahmen eines Kalküls*. Es setzt eine *formale Semantik* voraus. ***Schlussfolgern*** durch den Menschen ist das Ableiten von Schlüssen aus gegebenen Fakten, wobei aus introspektiver Sicht

1 Zu diesem Thema siehe z.B. [Klix 80].

des denkenden Menschen das Ableiten i.d.R. nicht explizit formalisiert ist. Verkürzt sagen wir: ***Schlussfolgern** ist das Ableiten von Aussagen mittels nichtformalisierter* ***Schlussregeln***. Unter Schlussregeln sind Regeln zu verstehen, nach denen Aussagen durch logisches Denken unter Verwendung einer beliebigen Sprache ohne Inanspruchnahme eines Kalküls in andere Aussagen umgeformt werden können. Beispielsweise kann man aus den Aussagen "A ist Kind von B" und "B ist Schwiegermutter von C" *schlussfolgern*, dass A und C miteinander verheiratet oder verschwägert sind (vgl. die Denksportaufgabe 1 in Kap.16.1 [16.2]). Rechnen und Schlussfolgern fassen wir unter der Bezeichnung **Deduzieren** zusammen. In Kap.16 werden wir sehen, wie sich das Schlussfolgern mathematisieren lässt, sodass die Grenze zwischen Schlussfolgern und Rechnen verschwimmt.

Deduzieren ist ein weitgehend bewusster Prozess, wir sagen: Deduktion ist eine Leistung *bewusster* Intelligenz. Der deduzierende Mensch weiß, wie er zu der deduzierten Aussage gekommen ist. Das gilt sowohl für das Rechnen als auch für das Schlussfolgern. Auch das Identifizieren kann ein bewusster Prozess sein, z.B. wenn man eine Pflanze durch Suchen im Gedächtnis oder in einem Pflanzenerkennungsbuch erkennt. Das Erkennen kann aber auch *spontan*, ohne Suchen und ohne 2 Nachdenken erfolgen, also weitgehend im Unterbewusstsein stattfinden, z.B. wenn man einen Bekannten erkennt. In diesem Fall sprechen wir von *Assoziation*.

Für die Erkenntnisgewinnung gilt Ähnliches. Erkenntnisgewinnung durch Deduzieren ist eine Leistung *bewusster* Intelligenz. Eine neue Erkenntnis kann einem aber auch "einfallen" (zufliegen, überkommen), ohne dass man weiß, wie der Einfall genau zustande gekommen ist. Der entscheidende Punkt eines solchen Erkenntnisprozesses liegt im Unbewussten. Es handelt sich um eine Leistung *unbewusster* Intelligenz. In diesem Fall sprechen wir von *Intuition*.

Da uns Assoziation und Intuition später noch beschäftigen werden, präzisieren wir ihre Definitionen (eingedenk der Tatsache, dass Aussagen Zuordnungen zwischen Objekten und Merkmalen sind): ***Assoziation** ist die Reproduktion (das **Auffinden** im Gedächtnis) bekannter Zuordnungen zwischen Objekten und Merkmalen ohne bewusstes Suchen. **Intuition** ist die Produktion (das **Erfinden**) neuer Zuordnungen zwischen Objekten und Merkmalen ohne bewusstes Deduzieren.* 3 Worauf Assoziation, Intuition und Erfinden beruht, welche Gehirnprozesse ihnen zugrunde liegen, ist weitgehend unbekannt. Die Introspektion sagt darüber wenig aus.

Wir fassen zusammen und ergänzen. Wir unterscheiden drei Wege, die zu Aussagen über die Welt führen, drei Methoden des sprachlichen Modellierens: Deduktion, Assoziation und Intuition. Die Fähigkeit, den einen oder anderen Weg des sprachlichen Modellierens zu gehen, nennen wir **deduktive** bzw. **assoziative** bzw. **intuitive Intelligenz**. Beim Wiedererkennen kommt in erster Linie assoziative Intelligenz zum Tragen, bei Erkenntnisgewinnung deduktive und intuitive Intelligenz.

Gegen die eingeführten Begriffsbestimmungen können verschiedene Einwände erhoben werden. Zunächst kann das Wort *Erfinden* irritieren. Denn üblicherweise

verbindet man damit das Erfinden eines Mechanismus, einer Maschine bzw. des Wirkprinzips einer Maschine. Danach passt das Wort Erfinden eher auf das *Konstruieren* als auf das sprachliche *Modellieren*. Doch ist zu bedenken, dass jedem Konstruieren sprachliches Modellieren vorausgeht. Das rechtfertigt unseren Gebrauch des Wortes **Erfinden** im Sinne von intuitivem Finden neuer Zusammenhänge zwischen Objekten und Merkmalen. Ein ähnlicher Einwand kann gegen die Definition des Begriffs *Intuition* erhoben werden. Es wird weit öfter von intuitivem Handeln als von intuitivem Denken gesprochen. Wieder ist zu beachten, dass jedem Handeln ein Modellieren vorangeht. Ferner kann gefragt werden, worin nach der gegebenen Begriffsbestimmung die Unterschiede zwischen Intuition, Kreativität und Phantasie liegen. Die Antwort lautet: Sie liegen im Kontext, in welchem das eine oder andere Wort vorzugsweise benutzt wird. Diese und ähnliche Einwände zeigen, dass es vielleicht besser gewesen wäre, neue Wörter zu erfinden. Wir haben vorgezogen, die Wörter “Intelligenz”, “Assoziation” und “Intuition” beizubehalten, sie aber in einer nicht ganz üblichen Bedeutung zu verwenden.

Zur Vertiefung des Verständnisses der eingeführten Begriffe seien einige Probleme angeführt, deren Lösung die eine oder andere Intelligenzart erfordert. Viele Rätsel verlangen intuitive Intelligenz, z.B. das Rätsel der Sphinx aus dem Ödipusmythos: “Früh geht’s auf vier Beinen, mittags auf zwei, abends auf drei” (Lösung: der Mensch). Das Multiplizieren zweier mehrstelliger Zahlen verlangt deduktive Intelligenz. Die Erfindung der Differenzialrechnung verlangte gleichzeitig intuitive *und* deduktive Intelligenz. Das soll näher erläutert werden.

Newton hat neue Sprachelemente (Differenzial, Differenzialquotienten verschiedener Ordnung) eingeführt (*erfunden*), d.h. deren Syntax und Semantik definiert. Sein Modellieren schloß also Begriffs- und Sprachbildung ein. Aus der Semantik und den bereits bekannten Rechenregeln konnte er neue Rechenregeln *ableiten*. Das Resultat war ein Kalkül, die Differenzialrechnung, wozu Leibniz den “invertierten” Kalkül, die Integralrechnung, erfunden hat. Differenzial- und Integralrechnung werden unter der Bezeichnung *Infinitesimalrechnung* zusammengefasst. Das große Gebiet der Mathematik, das auf Differenzieren und Integrieren beruht - dazu gehören u.a. die Differenzialgleichungen - wird *Analysis* genannt.[2] Die Aufstellung einer Differenzialgleichung zur Beschreibung eines Prozesses (z.B. der Bewegung der Erde, eines Pendels oder eines Schiffes) verlangt intuitive Intelligenz; ihre Lösung verlangt, falls der Lösungsweg bekannt ist, deduktive Intelligenz, andernfalls deduktive und intuitive Intelligenz.

Es ist nicht zu leugnen, dass die getroffene Unterscheidung zwischen drei Arten natürlicher Intelligenz (Deduktion, Assoziation, Intuition) etwas unmotiviert, zumin-

2 Es sei schon jetzt darauf hingewiesen, dass wir in Kap. 15.8 mit “*analytischem Rechen*” jedes Rechnen mit Variablen bezeichnen werden, also weit mehr, als nur das Rechnen im “Kalkül der Analysis”.

dest unüblich erscheinen kann und dass sie sich auch nicht in allen Punkten mit dem gängigen Lehrbuchwissen im Einklang befindet. So ist es beispielsweise nicht üblich, zwischen produktiver und reproduktiver Intelligenz zu unterscheiden. Andrerseits wird gewöhnlich der Deduktion die *Induktion* gegenübergestellt, die hier gar nicht erwähnt worden ist (der Grund wird in Kap.21.4 [21.8] genannt). Doch ist die getroffene Klassifikation im Hinblick auf die künstliche Intelligenz zweckmäßig, denn auf ihrer Grundlage lassen sich die oft gestellten Fragen nach Inhalt, Wesen und Grenzen der künstlichen Intelligenz präzisieren und zumindest partiell sinnvoll beantworten. So werden wir uns in den Kapiteln 15 und 16 genauer überlegen, was Programme leisten müssen, um den Computer zum Rechnen und Schlussfolgern, d.h. zum Deduzieren zu befähigen.

Das eingangs beschriebene Gegenläufigkeitsphänomen der natürlichen und künstlichen Intelligenz lässt sich nun so formulieren: Die künstliche Intelligenz ist von der berechnenden zur schlussfolgernden Stufe "*aufgestiegen*", und manches spricht dafür, dass sie vielleicht auch die intuitive Stufe erklimmen wird. Die natürliche Intelligenz dagegen beginnt ihren "*Aufstieg*" auf der intuitiven Stufe, um über die Stufe des Schlussfolgerns schließlich die des Rechnens zu erreichen.

Da die Evolution der natürlichen Intelligenz nicht mit der Intuition begonnen haben kann, ist die Frage berechtigt, worauf diese ihrerseits fußt, woraus sie sich entwickelt hat. Die naheliegende Anwort lautet: aus den Instinkten. Instinkte sind - ebenso wie die Intuition - unbewusste Auslöser bestimmter Verhaltens- oder Handlungsabläufe (im Gegensatz zu Reflexen, die scharf begrenzte Reaktionen darstellen wie Augenzwinkern oder innere Sekretion), wobei Instinkte entwicklungsgeschichtlich sicherlich älter sind als die Intuition.

Wenn man Instinkte und Intuition nicht als "Eingebungen" vonseiten höherer Instanzen erklären will, müssen sie entweder auf genetischer oder auf individueller *Erfahrung* beruhen, genauer gesagt, die Mechanismen, auf denen Instinkte und Intuition beruhen, müssen das Ergebnis der genetischen Evolution bzw. der individuellen, intellektuellen Evolution sein. Wir werden im Weiteren von folgender (den biologischen Erkenntnissen nicht widersprechenden) Vorstellung ausgehen. ***Instinkten*** *liegen genetisch codierte Erfahrungen der Gattung zugrunde,* **Intuitionen** *liegen neuronal codierte erworbene, unbewusste Erfahrungen des Individuums zugrunde.*

5

Erworbene, unbewusste Erfahrungen können entweder unbewusst erworben sein oder sie können bewusst erworben (z.B. gelernt), sodann aber *interiorisiert* (verinnerlicht, aus dem Bewusstsein verdrängt) worden sein. In jedem Fall ist die Folge, dass Intuition ein nicht erklärbares Phänomen zu sein scheint. Die Unzugänglichkeit für die Introspektion und die daraus folgende Unerklärbarkeit ist ein charakteristisches Merkmal der Intuition, zumindest in der umgangssprachlichen Bedeutung des Wortes. Insofern scheint obige Definition hypothetischen Charakter zu haben, sodass folgende Formulierung vorzuzuziehen ist: *Interiorisierte Erfahrung führt zum* ***Erscheinungsbild*** *der Intuition.* Die Hervorhebung des Wortes "*Erscheinungsbild*" soll folgenden Umstand unterstreichen. Welche Gehirnvorgänge der Intuition zugrunde

liegen, wissen wir nicht oder nur andeutungsweise. Das Gehirn wird als *schwarzer Kasten* aufgefasst, d.h. als nichtdekomponierbarer Operator, der nur durch seine Input-Output-Relation beschrieben werden kann, weil nur sie in *Erscheinung* tritt, d.h. beobachtet werden kann.

6 Wir sind hier auf einen Sachverhalt von grundsätzlicher Bedeutung gestoßen. Er trifft nämlich nicht nur auf die Intuition, sondern auf jedes menschliche und insbesondere auf intelligentes Verhalten zu. *Die künstliche Intelligenz simuliert* ***Erscheinungsbilder***, zumindest die traditionelle KI, von der in diesem Buch die Rede ist. Unter diesem Blickwinkel gewinnt das Wort "Simulieren" seine umgangssprachliche Doppelbedeutung von "*Nach*machen *und Vor*machen", Vormachen im Sinne von Weismachen, von "so tun, als ob". Die traditionelle, symbolische Informatik, modelliert nicht die Natur, insbesondere nicht die natürliche Intelligenz, sondern Erscheinungsbilder der Natur, insbesondere die Erscheinungsbilder der natürlichen Intelligenz. Sie macht also genau dasselbe wie ein Mensch, der über Natur*erscheinungen* nachdenkt und sprachlich modelliert. Interessanterweise trägt das Wort *Erscheinung*, ebenso wie das Wort *Simulieren*, häufig eine Bedeutungskomponente von "*Anschein*" oder "*so als ob*".

Diese Überlegungen liefern zwar keine Argumente zugunsten der Simulierbarkeit menschlicher Intelligenz, doch schwächen sie die Gegenargumente. Insbesondere entkräften sie diejenigen Argumente, die von der Unerkennbarkeit dessen ausgehen, was *hinter* den Erscheinungen steht, *hinter* dem Phänomen des Lebens und *hinter* dem Phänomen der menschlichen Intelligenz. Denn der "eigentliche" Hintergrund, der Urgrund der Erscheinungen, "die Wahrheit" stehen gar nicht zur Debatte. Simuliert werden Erscheinungen. Was "wirklich ist", entzieht sich nicht nur dem Zugriff des Computers, sondern auch dem des Menschen.

Aus diesen Überlegungen ziehen wir die Berechtigung, bei der ursprünglichen Formulierung zu bleiben: *Der Intuition liegen neuronal codierte, unbewusste, vom Individuum erworbene Erfahrungen zugrunde*. Noch prägnanter formulieren wir:
7 ***Intuition*** *beruht auf nichtbewusster Verarbeitung von nichtbewusstem Wissen.* Diese Schlussfolgerung ist zwingend, wenn man von *Eingebungen* vonseiten höherer Instanzen absieht.

Wir werden die Intuition als Komponente der natürlichen Intelligenz zunächst nicht in unsere Betrachtungen einbeziehen. Lediglich zwei sehr spezielle Erscheinungsformen der Intuition werden näher untersucht, *reduzible* Intuition (Kap.16.3 [16.11]) und *scheinbare* Intuition (Kap.17.3 [17.6]). Erst in Kap.21.4 [21.5] werden wir einige allgemeinere Überlegungen hinsichtlich der Implementierbarkeit (Simulierbarkeit) der Intuition anstellen.

Die eingeführte Terminologie und die illustrierenden Beispiele provozieren folgende Frage. Die *Entdeckung* von Naturgesetzen, beispielsweise der newtonschen Prinzipien der Mechanik, ist nach obiger Systematik als intuitive Leistung, als *Erfindung* zu klassifizieren. Die Terminologie scheint die Begriffe *Entdecken* und *Erfinden* zu verwechseln oder zu vermischen. Wie sind die beiden Begriffe in die

Systematik des sprachlichen Modellierens einzuordnen und wodurch unterscheiden sie sich? Die Frage trifft in gewissem Sinne den Kern der "Evolutionären Erkenntnistheorie", und die Antwort liegt nicht auf der Hand. Die übliche Unterscheidung scheint klar zu sein: Entdecken bezieht sich auf existierende, Erfinden auf zuvor nicht existierende Objekte. Aus der Sicht des sprachlichen Modellierens wird die Grenze zwischen beiden Begriffen unklar. Man kann z.B. fragen, ob mathematisches Modellieren der Welt Entdecken oder Erfinden ist, ob physikalische Gesetze entdeckt oder erfunden werden.

Die Beobachtung eines unbekannten Naturphänomens, seine Bewusstwerdung als Idem, ist eine *Entdeckung*. Seine sprachliche Modellierung (z.B. durch eine Differenzialgleichung) ist insoweit *Erfindung*, als für die Artikulierung neue Begriffe und neue Namen erfunden werden müssen, wie z.B. der Begriff des Differenzialquotienten oder die Notation $\mathrm{d}x/\mathrm{d}t$. Auch Begriffe wie Masse oder Temperatur wurden *erfunden*. Allgemein sind Begriffe, also benannte Ideme, die keine unmittelbaren Entsprechungen in der Wirklichkeit haben (deren Idemen keine Urrealeme entsprechen), Erfindungen der Menschen. Bei der Modellierung mit Hilfe der Mathematik oder des Computers kommt als intuitive Komponente die *Interpretation* von Rechengrößen als Modellgrößen hinzu, also die Anbindung der externen an die formale (kalkülinterne) bzw. an die rechnerinterne Semantik. Auch sie ist nicht ableitbar, sondern muss erfunden werden. Insofern werden physikalische Gesetze *erfunden*.

Damit ist gezeigt, inwieweit Erfinden Bestandteil des mathematischen Modellierens und Bestandteil des Modellierens mittels Computer ist, nämlich insoweit, wie das Bilden von Begriffen und von Sprachelementen (ihrer Syntax und Semantik) und das Anbinden von externer an interne Semantik Bestandteil des Modellierens ist. 8

7.2 Beschriftung der tabula rasa. Algorithmenbegriff

Wenn man Gedächtnisleistungen, wie es üblich ist, als intelligente Leistungen auffasst, stellt sich die Frage, warum sie in den bisherigen Überlegungen kaum eine Rolle gespielt haben und wie sie in die Intelligenzsystematik einzuordnen sind. Die Antwort ergibt sich aus der Feststellung, dass wir bei der Begriffsbestimmung der Intuition und Deduktion, um die es vor allem ging, stillschweigend nur die *produktive* (modell*bildende*) Komponente als die für die Bildung *neuer* Aussagen wesentliche Komponente im Auge hatten, obwohl in jedem Falle eine *reproduktive* (modell*nutzende*) Komponente in Form von Gedächtnisleistungen beteiligt ist. 9

***Gedächtnisleistungen** sind Leistungen der reproduktiven, assoziativen Intelligenz.* Ohne reproduktive Intelligenz, ohne Nutzung vorhandener Gedächtnisinhalte (vorhandenen Wissens) ist keine sinnvolle Erweiterung des Modells der Welt möglich. Darin liegt der evolutionäre Charakter der menschlichen Erkenntnis.

Hier tritt ein wesentlicher Unterschied zwischen natürlicher und künstlicher Intelligenz zutage. Der Computer ist offenbar nicht fähig, selber seine Fähigkeit zum

sprachlichen Modellieren (seine eigene Intelligenz) weiterzuentwickeln. Diese "übergeordnete" Intelligenz oder **Metaintelligenz** scheint dem Computer abzugehen, denn seine Intelligenz wird durch den Menschen gesteigert, indem dieser zusätzliche Daten und Programme einspeichert. Könnte es sein, dass der Schein trügt? In Kap.21.4 [21.7] kommen wir auf diese Frage zurück.

Wenn der Computer keine Metaintelligenz besitzt, kann seine Intelligenz nicht evolutionieren. Der Nutzer muss ihn "belehren". Sonst bleibt er so "dumm" wie er "geboren" (vom Produzenten mit Software ausgestattet) wurde. Welcher Computernutzer hat sich nicht schon über die Begriffsstutzigkeit seines Denkassistenten geärgert. Jedes Kind versteht, was man meint, der Computer nicht. Der Mensch bringt die Voraussetzungen für seine Dialogfähigkeit zum Teil in Form angeborener Gehirnstrukturen mit auf die Welt, zum anderen Teil erwirbt und verbessert er sie im Laufe seines Lebens als Folge zusätzlicher Strukturierung seines Gehirns unter dem Einfluss der Welt. Er besitzt Metaintelligenz.

In der Umgangssprache und auch in der Literatur ist mit *Intelligenz* sehr oft *Metaintelligenz* (in der Terminologie dieses Buches) gemeint. Als intelligent wird nicht derjenige bezeichnet, der sprechen (sprachlich modellieren) kann, sondern derjenige, der eine Sprache schnell lernt. Ein Schüler, dem Mathematik leichter fällt (der die Sprache der Mathematik schneller erfasst und beherrscht) als seine Mitschüler, wird als *intelligenter* eingestuft, womit gemeint ist, dass er seine Fähigkeit zum sprachlichen Modellieren schneller weiterentwickeln kann. Wir sind bei der Definition der Intelligenz stillschweigend davon ausgegangen, dass die Metaintelligenz Teil der Intelligenz ist. Dieser Standpunkt ist vernünftig, denn er führt zu einer ausreichend umfassenden Bestimmung des Intelligenzbegriffs, an der im Weiteren festgehalten wird.

Der Computer muss *vom Menschen* ausreichend intelligent gemacht werden, um dialogfähig zu sein, und zwar durch Anfüllung seines Speichers mit geeigneter Software, mit geeigneten Bitketten. Vorher ist er ein unbeschriebenes Blatt, eine tabula rasa. Mit der "angeborenen Ignoranz" des Rechners muss jeder Nutzer leben. Es wäre aber voreilig, aus dieser Erfahrung den Schluss zu ziehen, dass der Rechner prinzipiell "dumm" ist und niemals zu einem ebenbürtigen Dialogpartner des Menschen werden kann, oder, noch rigoroser, dass es überhaupt keine künstliche Intelligenz gibt und nicht geben kann. Derartige Meinungsäußerungen gründen sich auf dem genannten Umstand, dass der Nutzer selber dafür sorgen muss, dass sein Computer die notwendige Intelligenz besitzt. Er erreicht dies durch Entwicklung oder Kauf geeigneter Software, die er seinem Computer als Wissen "eintrichtert". Dabei handelt es sich primär um Produkte der *natürlichen* Intelligenz.

Diese Argumentation gegen die Bezeichnung "*künstliche Intelligenz*", gegen die sogenannte KI, ist richtig, wenn man Intelligenz von ihrer Entstehung her definiert. Wir haben sie jedoch als Eigenschaft ihres Trägers definiert. Wenn dieser künstlich, ein "Artefakt" ist, sprechen wir von künstlicher Intelligenz, ungeachtet ihres Ursprungs. Wo ihre Grenzen liegen, wissen wir damit freilich noch nicht. Die Antwort

auf die Frage, ob prinzipielle Grenzen der KI existieren, ist aus unserem derzeitigen Wissen und mit unseren derzeitigen Begriffen nicht ableitbar, sondern in gewissem Grade immer “Erfindung”. Jede diesbezügliche definitive Aussage ist als Hypothese, möglicherweise als wertvolle Arbeitshypothese anzusehen.

Natürlich erscheint es fraglich, ob das Ergebnis der Jahrmillionen langen genetischen Evolution sich so “auf die Schnelle” technisch kopieren, vielleicht sogar übertreffen lässt. Aber wer kennt die Grenzen der Erfindungskraft der menschlichen Intelligenz? Wie man sieht, führt die Frage sehr schnell auf ein zirkuläres Problem: Kann der Mensch die Grenzen seiner Erkenntnisfähigkeit erkennen? Wir wollen dieser introspektiv offensichtlich nicht beantwortbaren Frage nicht weiter nachgehen, sondern noch einmal auf das Gegenläufigkeitsphänomen zurückkommen und es begründen.

Das “Eintrichtern” von Wissen und damit von Intelligenz in den Computer, das Beschreiben der tabula rasa, beginnt natürlicherweise mit dem Rechnen. Denn der Mensch kann dem Rechner nur das beibringen, was er selber genügend genau verstanden (durchschaut) und sprachlich modelliert hat. Mit “beibringen” ist hier nicht das Anlernen durch Gewöhnen (“Einpauken”) gemeint, sondern das Eingeben von Daten und Vorschriften oder *Algorithmen*, nach denen der Computer zu verfahren hat, die ihn “schlau” machen.

Soeben ist einer der zentralen Begriffe der Informatik gefallen. Wir werden uns später eingehend mit Algorithmen beschäftigen, wollen den Begriff aber schon jetzt einführen. Darum unterbrechen wir die Überlegungen zur Beschriftung der tabula rasa für einen Augenblick.

Ein ***Algorithmus***[3] *ist eine Handlungsvorschrift (Prozessbeschreibung), die angibt, in welcher Reihenfolge mehrere, als bekannt vorausgesetzte Einzelhandlungen oder Operationsschritte auszuführen sind, um ein bestimmtes Ziel zu erreichen. Die Handlungsfolge muss eindeutig sein und ihre Ausführung muss terminieren, d.h. nach endlich vielen Schritten enden.* Manche Autoren verzichten auf die Forderung der Endlichkeit.

Ziel der Handlung kann beispielsweise das Produkt zweier Zahlen sein. Das Ziel kann auch ein Kuchen sein; dann spricht man i. Allg. allerdings nicht von Algorithmus, sondern von Rezept. Prozessbausteine (Teilhandlungen) können beim Multiplizieren z.B. das Addieren, beim Backen das Rühren sein.

Die Entwicklung von Algorithmen und die Herausbildung des Algorithmenbegriffs haben ihre Ursache offensichtlich darin, dass der Mensch “nur *einen* Kopf” hat. Er kann sich immer nur auf die Ausführung einer einzigen Aktion voll konzentrieren. Er kann nicht gleichzeitig zwei Zahlen addieren und zwei andere voneinander subtrahieren. Eine Rechenvorschrift, die als Algorithmus artikuliert ist,

---

3 Die Bezeichnung Algorithmus soll sich von dem Namen des persischen Mathematikers und Astronomen MOHAMMED IBN MUSA AL-CHWARISMI (um 820) herleiten.

entspricht dieser Eigenschaft des Menschen. Das erklärt die Rolle, die Algorithmen seit Jahrhunderten in der Rechenkunst spielen.

Heutzutage wird der Algorithmenbegriff häufig mit der Rechentechnik in Verbindung gebracht. Beispielsweise findet man im Informatik-Duden [Duden 89] die Definition: “*Unter einem Algorithmus versteht man eine Verarbeitungsvorschrift, die so präzise formuliert ist, dass sie von einem mechanisch oder elektronisch arbeitenden Gerät durchgeführt werden kann.*” Wir bleiben bei der zuvor gegebenen “klassischen” Definition, ergänzen sie aber durch Einführung eines spezielleren Algorithmenbegriffs.

Der engere Begriff ergibt sich aus dem allgemeineren dadurch, dass in der obigen Definition das Wort *Operationsschritt* durch *Aktion* ersetzt wird. *Als* ***Aktion*** *bezeichnen wir eine Operation an einem oder an mehreren bestimmten,* ***explizit ausgewiesenen*** *Operanden*. Eine Aktionsvorschrift wäre beispielsweise “Addiere die Werte von *a* und *b*” oder “Verrühre 1 Esslöffel Quark mit 1 Esslöffel Mehl”. Diese Sätze sind vollständige *Imperativsätze*. Je ein Imperativsatz eines Algorithmus schreibt je
10 eine Aktion vor. Darum nennen wir einen *Algorithmus, der eine terminierende Folge von Aktionen vorschreibt, einen* ***imperativen Algorithmus***. In der Literatur ist dieser Begriff nicht üblich. Wir führen ihn aus zwei Gründen ein, zum einen der Eindeutigkeit halber, denn der Algorithmenbegriff wird in unterschiedlichen Bedeutungen verwendet, oft im Sinne des imperativen Algorithmus, oft aber auch in einem allgemeineren Sinne, wie obiges Zitat aus dem Informatik-Duden zeigt. Zum anderen soll die Einführung des Begriffs des *imperativen Programms* in Kap.13.5.1 [13.9] vorbereitet werden. Die Begriffe “*Aktion*” und “*imperativer Algorithmus*” mögen im Augenblick gekünstelt erscheinen. Ein ganz einfaches Beispiels soll verdeutlichen,
11 worauf es ankommt. Es sei der Wert von *r* nach der Vorschrift $r = (a\text{-}b)+c$ für bestimmte Werte von *a, b* und *c* zu berechnen. Dazu müssen zwei Operationen ausgeführt werden, zuerst eine Subtraktion und anschließend eine Addition. Der erste Summand der Addition ist in der Vorschrift nicht explizit angegeben; die Vorschrift stellt also *nicht* die Beschreibung einer *Aktionsfolge* dar. Um sie in eine solche zu überführen hätte man beispielsweise zu schreiben: $a\text{-}b=d$; $d+c=r$. Die Notwendigkeit dieses Vorgehens wird sich in Verbindung mit dem Programmieren von Computern herausstellen (vgl. Kap.13.5.2 [13.11]).

Nach diesen Bemerkungen zum Begriff des Algorithmus wenden wir uns wieder der Beschriftung der tabula rasa des Computers mit Rechenvorschriften zu. Vorbild ist das “Beschriften der tabula rasa” des Menschen, das “Eintrichtern” von Wissen, speziell von Rechenvorschriften, in sein Gedächtnis. Besonders hilfreich ist es zu
12 beobachten, wie Kinder Rechnen lernen. Unabhängig davon, ob ein Kind im Klassenzimmer unter der Anleitung des Lehrers oder zu Hause am Computer lernt, beginnt alles mit dem *Zählen*, d.h. mit dem schrittweisen Erhöhen um Eins oder - in der Sprechweise der Mathematiker - mit *iterativem Inkrementieren.* Nach diesem Prinzip erfolgt beispielsweise das Addieren. Der ABC-Schütze addiert, indem er “mit

den Fingern zählt". Nach der gleichen Methode subtrahiert er auch, indem er zählt, wie oft er den Subtrahend inkrementieren muss, um den Minuend zu erreichen.

Ganz analog, nur "eine Stufe höher", wird multipliziert und dividiert, nämlich durch wiederholtes Addieren. Multiplizieren und Dividieren lassen sich also auf Addieren und weiter auf Inkrementieren zurückführen. Dabei muss ständig geprüft werden, ob die laufende Iteration fortzusetzen oder abzubrechen ist. Um das beispielsweise beim Subtrahieren zu entscheiden, muss der inkrementierte Subtrahend mit dem Minuend verglichen werden. Tatsächlich lässt sich *jede* arithmetische Operation auf Inkrementieren und Vergleichen zurückführen, oder anders ausgedrückt, *jede* ***arithmetische Operation*** *lässt sich aus Inkrementier- und Vergleichsoperationen* ***komponieren***. (Der exakte Nachweis der Richtigkeit dieser Aussage wird in den weiteren Kapiteln erbracht.)

Damit ein Computer rechnen kann, würde es im Prinzip also ausreichen, wenn er über Hardwareoperatoren (Schaltungen) für das Inkrementieren und Vergleichen verfügt und wenn ihm Vorschriften (Programme) für das Komponieren arithmetischer Operationen aus Inkrementier- und Vergleichsoperationen eingegeben sind und er diese auch ausführen (interpretieren) kann.

Computer besitzen jedoch eine "intelligentere" Hardware, als die soeben geforderte. Die Schaltung, die letzten Endes alle Operationen ausführt, heißt **arithmetisch-logische Einheit**, abgekürzt **ALU** ("U" von unit). Sie kann in der Regel die Addition, die Subtraktion, den Vergleich und einige weitere elementare Operationen ausführen.

Das Erstaunliche der Entwicklung der Rechentechnik besteht darin, dass es im Laufe der Zeit gelungen ist, die tabula rasa in einer Weise zu beschriften (Algorithmen in das Gedächtnis des Computers einzuspeichern), die ihm Fähigkeiten verleihen, die weit über das Rechnen hinausgehen, sodass man ihm Intelligenz im üblichen Sinne des Wortes zugestehen muss. Dass diese Entwicklung mit dem Addieren begonnen hat, ist nach den vorangehenden Überlegungen durchaus verständlich. Ebenso verständlich ist, dass die Entwicklung der natürlichen Intelligenz entgegengesetzt verlaufen ist und mit der Assoziation und Intuition begonnen hat. Denn die Überlebenschancen eines Individuums im Kampf ums Dasein sind wesentlich dadurch mitbestimmt, wie schnell es ihm gelingt, Gefahren zu erkennen oder nur zu wittern und richtig auf sie zu reagieren. Wie sich daraus aber die Fähigkeit zum Rechnen entwickelt hat, ist weitgehend unklar. Vielleicht könnte man es verstehen, wenn man genau wüsste, wie das Assoziieren neurophysiologisch ("hardwaremäßig") funktioniert.

Man kommt zu einer merkwürdigen Feststellung: Der Mensch kann sein eigenes Denken zwar simulieren, zumindest teilweise, er kann es aber bis heute im Grunde nicht verstehen. Verallgemeinert und etwas verkürzt kann man sagen: *Der Mensch kann mehr als er versteht*. Das war schon mit dem Fliegen so und mit der Erfindung der Dampfmaschine und mit vielen anderen Erfindungen, die zur Zeit, als sie gemacht

wurden, nicht nur nicht hergeleitet (aus bekanntem Wissen deduziert), sondern auch nicht erklärt, nicht verstanden werden konnten.

Damit ergibt sich ein zweites zirkuläres Problem: Inwieweit kann der Mensch verstehen, was er selber “kann”, was er produzieren kann, und inwieweit kann er die Grenze zwischen dem, was er kann und was er nicht kann, erkennen? Offensichtlich handelt es sich auch hier um eine ausweglose Zirkularität, und offensichtlich ist der Satz “*Der Mensch kann die Grenzen des eigenen Könnens (Vermögens) nicht erkennen*” wahr. Der Satz bezieht sich nicht auf die Grenzen des Denkens und Handelns in konkreten Situationen, z.B. darauf, wie viele Zahlen ich mir merken oder wie hoch ich springen kann. Ich weiß beispielsweise, dass ich nicht 10 Meter hoch springen kann.

8 Formale Grundbegriffe und Methoden

Zusammenfassung

Ein Prozess, der sich als zeitlich diskrete Folge kausal verknüpfter Ereignisse beschreiben lässt, heißt *kausaldiskret*. Der Träger eines kausaldiskreten Prozesses heißt *kausaldiskretes System*. Es wird die *USB-Methode* (USB - Uniforme Systembeschreibung) für die Beschreibung kausaldiskreter Prozesse und Systeme eingeführt. Die Methode ist für die Komponierung hierarchisch strukturierter Hard- und Softwaresysteme in Form von Operatorenhierarchien entwickelt worden.

Ein Operator ordnet Eingabeoperanden bzw. den Werten einer Eingabevariablen x Ausgabeoperanden bzw. Werte der Ausgabevariablen y zu. Wenn die Zuordnung durch einen Menschen oder ein Gerät vorgenommen wird, heißt der Operator *real*; wenn sie durch eine Vorschrift festgelegt wird, heißt er *sprachlich*. Ein sprachlicher Operator bedarf eines realen Operators, eines *Interpretierers*, der die Vorschrift ausführt. Wenn alle Zuordnungen, die ein Operator trifft, eindeutig sind, d.h. wenn einem x-Wert (bzw. Wertetupel) genau ein y-Wert (bzw. Wertetupel) zugeordnet wird, heißt die Gesamtheit der Zuordnungen *Funktion* oder *Abbildung*. Zuweilen wird der Abbildungsbegriff in einem weiteren Sinne verwendet ("*Abbildung i.w.S.*"), der die Eindeutigkeit der Zuordnung nicht einschließt. Im Weiteren ist unter Abbildung im mathematischen Sinne stets eine Funktion, also eine "Abbildung i.e.S." (im engen Sinne) zu verstehen. Eine Funktion heißt *berechenbar*, wenn sie durch eine Vorschrift (einen sprachlichen Operator) festgelegt ist und ein Interpretierer der Vorschrift existiert oder angebbar ist.

Die Komponierung eines sog. *Kompositoperators* aus *Bausteinoperatoren* erfolgt durch deren Verbindung zu einem *Operatorennetz* (ON) mit einem Eingang und einem Ausgang. Über die Verbindungen werden Operanden übergeben. Der Operandenfluss durch ein ON wird durch sog. *Flussknoten* bestimmt, die z.T. steuerbar sind. Kompositoperatoren können als Bausteinoperatoren eines Kompositoperators höherer *Komponierungsstufe* dienen. Auf diese Weise kann eine *Operatorenhierarchie* mit *Schichtenarchitektur* aufgebaut werden. Die Operatoren einer Schicht (mit Ausnahme der untersten) werden aus Operatoren der nächst tieferen Schicht komponiert und können ihrerseits als Bausteinoperatoren zur Komponierung eines Operators der nächsthöheren Schicht dienen.

Operatoren (reale oder sprachliche), die aus einem vorgegebenen Satz elementarer Operatoren nach der USB-Methode komponierbar sind, heißen *USB-Operatoren*. Funktionen, die durch USB-Operatoren berechenbar sind, heißen *USB-Funktionen*. Jede rekursive Funktion ist als USB-Funktion unter Verwendung des Inkrementierers als einzigem elementarem Operator darstellbar.

Für die Entwicklung der elektronischen Rechentechnik waren die folgenden vier Ideen richtungweisend:

- die Idee, die boolesche Algebra als sog. *Basiskalkül* zu implementieren und alle anderen zu implementierenden Kalküle auf das Basiskalkül zurückzuführen;
- die Idee, die elementaren booleschen Operatoren als Schalternetze zu realisieren;
- die Idee, einen steuerbaren Bitkettentransformator zu realisieren, der in der Lage ist sprachliche Operatoren als Operationsvorschriften zu verstehen und auszuführen;
- die Idee, das Übersetzen sprachlicher Operatoren in die Maschinensprache durch den Computer ausführen zu lassen.

8.1 Uniforme Systembeschreibung (USB)

Als Ziel der Informatik wird zuweilen der Computer mit menschlicher Intelligenz proklamiert, die Konstruktion eines informationellen Systems, das deduktive und intuitive Intelligenz besitzt, das wie der Mensch rechnen, schlussfolgern und erfinden kann. Gemäß der historischen Entwicklung der Rechentechnik beschränken wir uns zunächst auf deduktive Intelligenz, konkret auf die Fähigkeit zum *Rechnen*. Hinsichtlich des Rechnens soll das System universell sein. Was das genau bedeutet, wird uns noch beschäftigen. Im Augenblick wollen wir damit die Vorstellung verbinden, dass das System imstande ist, alles zu berechnen, was der Mensch berechnen kann. In diesem Fall wollen wir von *universellem Rechner* sprechen, wohl wissend, dass dies eine sehr unscharfe Begriffsbestimmung ist, die früher oder später durch eine schärfere zu ersetzen ist.

Da unser Geist zu schwach ist, ein so mächtiges System in seinen Einzelheiten mit einem einzigen Gedanken zu erfassen und "in einem Stück" zu entwerfen, müssen wir versuchen, es schrittweise, am besten - nach dem Vorbild der Natur - hierarchisch (vgl.Kap.5.2 [5.2]) aufzubauen. Das bedeutet, dass das Produkt eine *Schichtenstruktur* besitzt. Jede Schicht enthält Bausteine, die bestimmte Operationen ausführen können. Diese Bausteine nennen wir **Operatoren**, abgekürzt op. Aufgabe eines Operators ist es, Eingabeoperanden in Ausgabeoperanden zu überführen. Die genaue Definition des Operatorbegriffs erfolgt zu Beginn des Kapitels 8.2.1. Die Bausteine der untersten Schicht heißen **elementare Operatoren**. Aus ihnen werden neue Operatoren komponiert, indem sie zu Netzen, sog.**Operatorennetzen**, abgekürzt ON, verbunden werden. Die ON werden mit je einem Eingang und einem Ausgang versehen, sodass sie die Rolle von Operatoren spielen und ihrerseits zu Netzen verbunden werden können.

Durch schrittweises Komponieren kann eine **Operatorenhierarchie** mit Schichtenstrukur, kurz eine **Schichtenarchitektur** aufgebaut werden. Die Operatoren einer Schicht (mit Ausnahme der untersten) werden aus Operatoren der nächsttieferen Schicht komponiert und können ihrerseits als Bausteinoperatoren zur Komponierung eines Operators der nächsthöheren Schicht dienen. Um zwischen den Bausteinen und

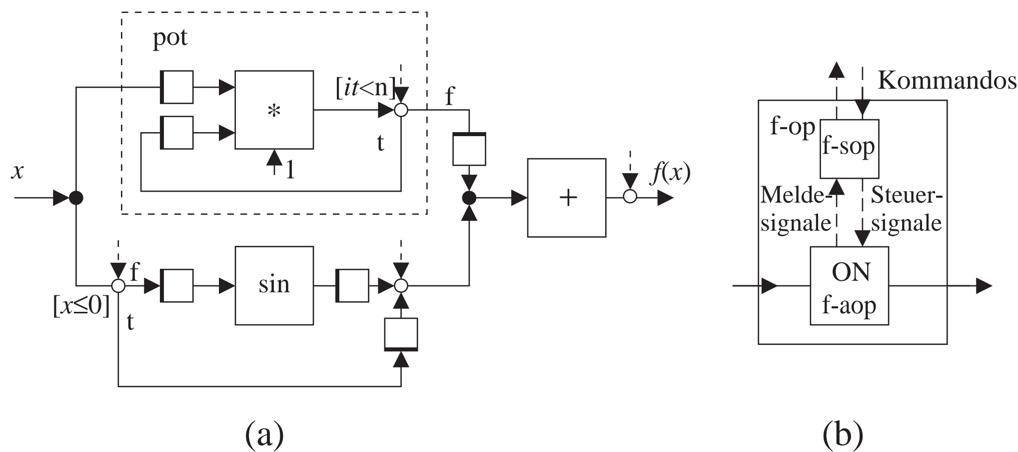


Bild 8.1 Kompositoperator zur Berechnung der Funktion (8.1). **(a)** - Darstellung als Operatorennetz (ON); große Quadrate - Operatoren; kleine Quadrate - Operandenplätze; kleine Kreise - Weichen. **(b)** - Darstellung als gesteuerter Operator; ON - das Operatorennetz von Bild (a); f-sop - Steueroperator des Operators f-op; f-aop - Arbeitsoperator des Operators f-op.

dem Ergebnis eines Komponierungsschrittes zu unterscheiden, sprechen wir von **Bausteinoperatoren** und **Kompositoperatoren** bezüglich des betreffenden Komponierungsschrittes. Je höher die Komponierungsstufe eines Kompositoperators ist, umso “höher” liegt die Schicht, der er angehört, und umso komplexer ist die Operation (die **Kompositoperation**), die er ausführt. Operationen zunehmender Komplexität sind beispielsweise das Inkrementieren, das Addieren und das Multiplizieren.

Das Komponieren eines Operatorennetzes soll anhand der beiden Funktionen

$$f_1(x) = x^n + x \quad \text{und} \quad f_2(x) = x^n + \sin x$$

demonstriert werden. Der Exponent n stellt keine Variable, sondern eine vorgebbare Konstante dar. Der erfahrene Leser wird wahrscheinlich sofort erkennen, dass sich mit Hilfe des in Bild 8.1a dargestellten Netzes die beiden Funktionen f_1 und f_2 berechnen lassen. Wir wollen die Funktionsweise des Netzes im Einzelnen verfolgen. Das wird etwas mühsam. Zur Motivation sei gesagt, dass wir nach der hier beschriebenen Vorgehensweise in Teil 2 einen universellen Rechner als Operatorenhierarchie entwerfen werden.

Das Netz von Bild 8.1a enthält je einen Bausteinoperator für das Addieren, für das Multiplizieren[1] und für die Berechnung der Sinusfunktion. Die Operatoren sind durch große Quadrate dargestellt. Die Operanden werden längs der Pfeile weitergegeben. In jedem möglichen Übergabeweg liegt ein Operandenplatz (Speicher) zur Ablage von Operanden, dargestellt als kleines Quadrat mit einer fetten Seite, durch

1 In der Rechentechnik wird für das Multiplizieren das Symbol ∗ verwendet.

die ein Operand in den Operandenplatz eintreten kann. Ein Operandenplatz kann als spezieller Operator aufgefasst werden, der einen Operanden nicht verändert, ihn aber ausreichend lange aufbewahren kann.

Der **Operandenfluss** durch das Netz kann durch **Steuersignale** gesteuert werden, die von einem **Steueroperator** (mit sop bezeichnet) generiert werden. Der Steueroperator ist Bestandteil des Kompositoperators, also einer seiner Bausteinoperatoren. Um die Operatoren des Netzes vom Steueroperator zu unterscheiden, nennen wir sie **Arbeitsoperatoren** (aop). Ein Kompositoperator besteht also in der Regel aus einem sop und einem oder mehreren aop, die zu einem Operatorennetz verbunden sind. In Bild 8.1a ist der sop nicht eingezeichnet, wohl aber in Bild 8.1b.

Die Darstellung in Bild 8.1b ist das Ergebnis einer *Abstraktion*. Es wird von den Details des Netzes abstrahiert. Das Netz wird zu einem *nicht* dekomponierten Operator, zu einem “schwarzen Kasten”, dessen Inneres nicht sichtbar ist. Wir nennen diese Abstraktion **Operatorabstraktion**. Der (abstrahierte) Operator ist in Bild 8.1b mit ON bezeichnet. Er ist **steuerbar**; sein Verhalten ist unterschiedlich, je nachdem wie der Steueroperator den Operandenfluss durch das Netz der Arbeitsoperatoren steuert. Der abstrahierte Operator wird - ebenso wie seine (unsichtbaren) Komponenten - *Arbeitsoperator* des Steueroperators genannt. Das ON in Bild 8.1b ist also identisch mit dem Arbeitsoperator f-aop des Steueroperators f-sop. Durch Operatorabstraktion auf der nächsthöheren Eben gelangt man zu dem Operator f-op, indem man von der Dekomponierung in f-sop und f-aop absrahiert.

Wir wollen uns die Funktionsweise des Netzes klarmachen und uns dabei vorstellen, dass die Operatoren als elektronische Schaltungen realisiert sind. Die Pfeile stellen dann leitende Verbindungen dar. Über den *externen* Eingabepfeil, das ist der Eingabepfeil des Netzes, werde ein Eingabeoperand x eingegeben. An der durch einen dicken Punkt in der Eingabeleitung markierten Stelle, **Kopiergabel** genannt, erfolgt eine Duplizierung des Operanden x; das eine Operandenexemplar wird an den Multiplizierer weitergegeben, das andere an den Sinusoperator. Im Falle elektrischer Leitungen, welche die Operanden in Form elektrischer Spannungen weiterleiten, findet das Duplizieren durch die Existenz einer Gabelung des Leiters statt; ein spezieller Kopieroperator erübrigt sich.

Der Ausgabeoperand (das Resultat) des Multiplizierers wird mit dem des Sinusoperators in dem rechten dicken Punkt zu einem *Operandenpaar* vereinigt, zum Summandenpaar des nachfolgenden Addierers. Dieser Punkt wird **Vereinung** genannt. Es ist zu beachten, dass die Zusammenführung der Übergabepfeile *nicht* in dem “Zusammenlöten” der beiden Verbindungen besteht. Es handelt sich um eine begriffliche Vereinigung, um die *gedankliche* Zusammenfassung zweier Operanden zu *einem* **Kompositoperanden**. In Wirklichkeit laufen die beiden leitenden Verbindungen auch nach der Vereinung getrennt weiter. Allgemein kann ein Kompositoperand ein Tupel aus mehreren **Bausteinoperanden** sein.

Für das Addieren sind beide Summanden gleichzeitig erforderlich, sodass eventuell der eine Summand auf den anderen warten muss. Dafür steht ihm der Speicher

in der betreffenden Eingabeleitung des Addierers zur Verfügung. Das gilt allgemein. Die Operanden, die in einer Vereinung zu einem Paar oder einem Tupel vereint werden, müssen die Vereinung nicht gleichzeitig erreichen, sie müssen die Vereinung aber gleichzeitig verlassen. Die Vereinung fungiert also als **Synchronisierer**.

Das Pendant der Vereinung ist die **Spaltegabel**. Sie spaltet ein eintreffendes Operandentupel in Teiltupel auf, die dann getrennt voneinander weitergeleitet werden. Spalte- und Kopiergabel fassen wir unter dem Oberbegriff **Gabel** zusammen. Werden die Ausgänge einer Gabel mit den Eingängen einer Vereinung verbunden - evtl. über einen oder mehrere Operatoren -, ergibt sich eine **starre Masche**. Das Operatorennetz von Bild 8.1a besteht aus einer starren Masche und einem nachgeschalteten Addierer.

Der Sinusoperator ist durch eine steuerbare **Vorkopplung** überbrückt, d.h. er kann durch entsprechende Stellung der davor liegenden **Weiche** umgangen werden. Die Weiche ist als kleiner Kreis dargestellt. Das sich so ergebende Teilnetz stellt eine **steuerbare Masche** dar. Sie enthält zwei **Weichen**, eine **Zweigeweiche** (der linke kleine Kreis) und eine **Sammelweiche** (der rechte kleine Kreis).

Eine Weiche hat eine ähnliche Funktion wie beispielsweise eine Straßenbahnweiche. Durch das Stellen der Weichen wird bewirkt, dass entweder x oder sin$x$ dem Addierer zugeführt wird, m.a.W. durch die Weichenstellung wird die *Alternative* “entweder x oder sin$x$” entschieden. Darum wird eine steuerbare Masche auch **Alternativmasche** genannt. Wenn gesichert ist, dass zwischen den beiden Weichen einer Alternativmasche sich niemals mehr als ein einziger Operand befindet, genügt es, nur eine von den beiden Weichen zu stellen. Wenn die Zweigeweiche gestellt wird, kann die Sammelweiche ständig für beide Wege geöffnet sein, d.h. sie kann durch eine Vereinung ersetzt werden; wenn die Sammelweiche gestellt wird, kann die Zweigeweiche durch eine Kopiergabel ersetzt werden.

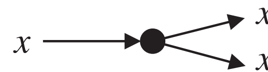
1

Vereinung, Gabeln und Weichen werden unter der Bezeichnung **Flussknoten** zusammengefasst. Weichen sind **steuerbare**, Vereinung und Gabeln sind **starre** Flussknoten. In Bild 8.2 sind die verschiedenen Flussknotentypen und die entsprechenden Symbole zusammengestellt. Das Symbol der Vereinung wird evtl. auch für eine Sammelweiche verwendet, wenn diese in einer Alternativmasche liegt. Der Vollständigkeit halber ist der Liste von Bild 8.2 das **Tor** als zusätzliches steuerbares Element hinzugefügt worden, obwohl es kein *Knoten* im eigentlichen Sinne ist. Mittels eines Tores kann ein Übergabeweg gesperrt oder geöffnet werden.

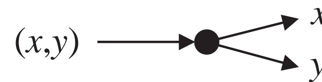
Es könnte sinnvoll erscheinen, die Spaltegabel als steuerbaren Flussknoten zu definieren, dessen Tupelaufspaltung durch eine Steuerinformation festgelegt wird. Doch ist das nicht notwendig. Es lässt sich nämlich zeigen, dass eine steuerbare Spaltegabel mit Hilfe der Flussknoten von Bild 8.2 realisiert werden kann und zwar in Form des in Bild 8.8b dargestellten steuerbaren Selektors. Gegebenenfalls werden die Ausgabeleitungen einer Spaltegabel mit den entsprechenden Teiltupeln beschriftet.

starre Flußknoten:

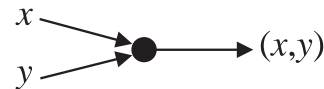
– Kopiergabel



– Spaltegabel

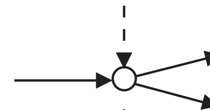


– Vereinung

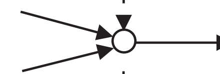


steuerbare Flußknoten:

– Zweigeweiche



– Sammelweiche



– Tor

Bild 8.2: Flussknotentypen und ihre Symbole. Gestrichelte Pfeile stellen Steuersignaleingänge dar.

Der Multiplizierer in Bild 8.1a ist durch eine steuerbare **Rückkopplung** überbrückt. Im Gegensatz zu Maschen (Vorkopplungen) werden Rückkopplungen auch **Schleifen** genannt. Die Schleife in Bild 8.1a enthält eine Vereinung und eine Zweigeweiche. Die Vereinung ist allerdings ausnahmsweise nicht als Punkt dargestellt, m.a.W. die beiden Eingabeoperanden (die beiden Faktoren des Produktes) werden *nicht* gedanklich zu *einem* Kompositoperanden zusammengefasst. Diese Abweichung von obiger Regel ist eine Konzession an die bessere Lesbarkeit des Bildes. Die Darstellung der Vereinung als Punkt ist kein Zwang. Ebenso ist die Verwendung der Gabel kein Zwang, sondern ein Operator darf auch zwei oder mehrere Ausgabeleitungen besitzen. Die Einführung der starren Flussknoten dient der Einheitlichkeit der Darstellung, wodurch spätere Gedankengänge übersichtlicher werden.

Über die Rückkopplung kann das Resultat auf den Eingang zurückgeführt und die Multiplikation mehrmals ausgeführt werden. Durch entsprechende Weichenstellung wird die Anzahl der Multiplikationen, m.a.W. die Potenz bestimmt, in die x erhoben wird. Bei der ersten Multiplikation soll x ausgegeben werden, d.h. als zweiter Faktor muss als Anfangswert oder **Initialwert** zuvor eine 1 eingegeben worden sein. Das ist in der Abbildung durch den kleinen zusätzlichen Eingabepfeil am Multiplizierer angedeutet.

Welche der beiden Funktionen, f_1 oder f_2, der f-Operator berechnet und für welchen Exponenten n er sie berechnet, hängt davon ab, wie die Weichen gestellt

werden. Das erfolgt durch **Steuersignale**, die vom Steueroperator generiert und den Weichen über die gestrichelten Eingänge zugeführt werden. Dem Steueroperator muss mitgeteilt werden, für welche Funktion und für welchen Exponenten er die Steuersignale generieren soll. Damit er sie zeitgerecht generiert, kann es notwendig sein, dass ihm gemeldet wird, wann im ON eine Bausteinoperation beendet ist (durch den gestrichelten Pfeil vom ON zum f-sop in Bild 8.1b angedeutet). Der f-Operator umfasst das Operatorennetz (ON, identisch mit f-aop) und dessen Steueroperator (f-sop). Er ist ein *steuerbarer* Operator.

Ein genauerer Blick auf die Schaltung in Bild 8.1a unter Einbeziehung der Speicher zeigt, dass die Weichen dadurch gestellt werden können, dass die Eingänge der Operandenplätze (die dicken Seiten der kleinen Quadrate) als Tore ausgebildet werden, die durch Steuersignale geöffnet bzw. geschlossen werden. Eine Weiche wird dann durch entsprechende Steuerung der Tore gestellt, die in den beiden Zweigen der Weiche liegen. Dadurch wird aus der *Weichensteuerng* eine **Torsteuerung**. Diese ist der Weichensteuerung überlegen, denn sie kann nicht nur Weichen steuern, sondern in *jedem* Punkt des Netzes den Operandenfluss freigeben oder sperren. Bei der Realisierung von Operatorennetzen durch elektronische Schaltkreise (Teil 2) kommt die Torsteuerung zur Anwendung. Für die Steuerung *einer* Weiche sind zwei *Torsteuersignale* erforderlich.

Die beiden Funktionen f_1 und f_2 können zu einer Funktion f zusammengefasst werden, wenn in eindeutiger Weise festgelegt wird, für welche x-Werte f_1 und für welche f_2 berechnet werden soll, beispielsweise folgendermaßen:

$$f(x) = f_1(x) = x^n + x \qquad \text{für } x \leq 0 \tag{8.1a}$$

$$f(x) = f_2(x) = x^n + \sin x \qquad \text{für } x > 0. \tag{8.1b}$$

Der gesamte Kompositoperator von Bild 8.1 einschließlich seines Steueroperators kann seinerseits als Bausteinoperator eines Kompositoperators der nächsthöheren Schicht dienen. Auf diese Weise lässt sich Schritt für Schritt eine Hierarchie von Operatoren zunehmender Komplexität aufbauen, eine *Operatorenhierarchie*. Ein Operator einer bestimmten Schicht wird in der darunterliegenden Schicht in ein Operatorennetz mit seinem Steueroperator **dekomponiert**.

Der Operator von Bild 8.1 kann bereits als Hierarchie mit zwei Komponierungsebenen aufgefasst werden, denn der gestrichelt umrahmte Potenzieroperator ist in einen (allerdings nur in einen einzigen) Arbeitsoperator, den Multiplizierer und eine Rückkopplungsschleife dekomponiert. In Bild 8.1b ist der Steueroperator, der die Zweigeweiche in der Ausgabeleitung des Multiplizierers steuert, in den Steueroperator des gesamten Netzes (f-sop) integriert. Später werden wir auch den Sinusoperator dekomponieren.

2

Wir hatten uns vorgestellt, dass unser Kompositoperator als Schaltung existiert. Das Komponieren erfolgte *hardwaremäßig*. Demgegenüber war in Kap.7.2 dargelegt worden, wie der ABC-Schütze das *Komponieren nach Vorschrift* lernt. So führt er z.B. das Addieren oder Subtrahieren auf das Inkrementieren zurück, indem er “mit

den Fingern rechnet", d.h. zählt. Der universelle Rechner, den wir bauen wollen, muss offenbar auch zu dieser Art des Komponierens in der Lage sein, er muss diese Fähigkeit menschlicher Intelligenz simulieren können, denn es ist wegen des erforderlichen Aufwandes praktisch unmöglich, für jede denkbare Funktion eine Schaltung zu bauen, selbst wenn man den sparsamen Weg des Komponierens einer Operatorenhierarchie geht (siehe dazu die Abschätzungen (9.3) und (9.4) in Kap.9.2).

Das bedeutet, dass der angestrebte Rechner über mindestens einen Operator verfügen muss, der *Vorschriften ausführen* (interpretieren) kann. Die Konstruktion eines derartigen **interpretierenden Operators** oder **Interpretators** wird Inhalt von Kap.13 sein. Für ihn hat sich die Bezeichnung **Prozessor** eingebürgert. Wenn er zur Verfügung steht, kann das *hardwaremäßige* Komponieren *softwaremäßig* fortgesetzt werden, sodass eine Hardware-Software-Hierarchie entsteht.

Die graphische Darstellung der Struktur von Kompositoperatoren gemäß Bild 8.1 nennen wir **Operandenflussgraph**. Die Bausteinoperatoren können sowohl reale, z.B. elektronische, als auch sprachliche, z.B. mathematische Operatoren sein. Dabei hatten wir bisher Operatoren im Auge, die Informationen verarbeiten, genauer Zeichenketten, durch welche Informationen verschlüsselt (codiert) sind. Im folgenden Kapitel werden wir uns überzeugen, dass die Darstellungsmethode auf eine weit umfangreichere Klasse von Operatoren anwendbar ist, sodass es berechtigt erscheint, von **uniformer**, d.h. "*formal einheitlicher*" Beschreibung zu sprechen.

Wir vereinbaren: *Die Beschreibung komplexer Operatoren als Operatorennetze unter Verwendung der Flussknotentypen von Bild 8.2 bezeichnen wir als* ***uniforme Systembeschreibung****, abgekürzt* ***USB****. Ein* ***Operandenflussgraph*** *ist die graphische Darstellung der nach der USB-Methode beschriebenen Struktur eines Operatorennetzes bzw. eines Kompositoperators.*

Man beachte, dass aus einem Operandenflussgraphen nicht unbedingt die Operation abzulesen ist, die der Kompositoperator ausführt, sodass er nicht als Operationsvorschrift dienen kann. Es fehlen die Bedingungen für die Steuerung der Weichen. Um einen Operadenflussgraphen zu einer Operationsvorschrift zu vervollständigen, muss jeder steuerbare Flussknoten mit einer Steuerbedingung beschriftet werden. Dadurch wird aus dem Operandefluss*graph* ein **Operandenfluss*plan***.

Das Operatorennetz (der Operandenflussgraph) von Bild 8.1 ist ein Operandenflussplan, denn er enthält (in eckigen Klammern) die Bedingungen für die Stellung der beiden Zweigeweichen. Wenn die Bedingung $x \leq 0$ erfüllt ist, wird die Weiche vor dem Sinusoperator nach unten gestellt und der Sinusoperator umgangen. Das t an diesem Ast kommt von dem englischen true und bedeutet, dass die Bedingung erfüllt ist. Bei Nichterfüllung wird der ankommende Operand über den Zweig, der mit f (von false) beschriftet ist, dem Sinusoperator übergeben. Entsprechend ist die Bedingung $it<n$ an der Weiche nach dem Multiplizierer zu lesen, wobei *it* die *Iterationszahl* bezeichnet, d.h. die Anzahl der bereits erfolgten Multiplikationen.

Der Begriff des Operandenflussplans kann nun folgendermaßen definiert werden: *Ein* ***Operandenflussplan*** *ist ein mit den Steuerprädikaten beschrifteter Operanden-*

flussgraph, der eine vollständige Operationsvorschrift darstellt. Damit haben wir eine Abstraktionsebene bezogen, auf der ein Operandenflussplan als *zweidimensionale* Notation einer Operationsvorschrift aufgefasst werden kann. Auf dem gleichen Abstraktionsniveau können die Formeln (8.1) als *eindimensionale* (lineare) *Beschreibung* des Operandenflussplans des Kompositoperators f-op von Bild 8.1, aufgefasst werden, wobei allerdings der Potenzoperator nicht dekomponiert ist. Die Formeln (8.1) enthalten die Bedingungen für die Steuerung der Zweigeweiche vor dem Sinus-Operator in Form zweier Ungleichungen.

8.2 Kausaldiskrete Prozessbeschreibung

8.2.1 Reale und sprachliche Operatoren

Mit dem Operatorbegriff sind wir recht großzügig umgegangen, indem wir ihn für Objekte ganz unterschiedlicher Natur verwendet haben, für reale (materielle, physische) und für sprachliche Objekte. Für eine formal einheitliche, “*uniforme*” Beschreibung der Hard- und Softwarekomponierung ist ein entsprechend allgemeiner Operatorbegriff erforderlich, den wir nun herausarbeiten wollen.

Der Begriff des Operators ist sowohl in der Mathematik als auch in der Technik üblich, jedoch mit unterschiedlichen Bedeutungen. Der wesentliche Unterschied besteht darin, dass der technische Operator, z.B. ein **Fertigungsoperator**, der Werkstücke bearbeitet bzw. anfertigt, ein stofflicher Operator ist, der selbst *agiert* (wie die *agierenden* Modelle in Bild 3.1), während ein mathematischer Operator[2] nicht selbst agieren kann, sondern ein Zeichen oder eine Zeichenkette darstellt, die der Mensch als Vorschrift interpretieren und ausführen kann. Der *agierende* Operator ist in dem Falle der Mensch.

Bei der Komponierung informationeller Systeme müssen wir mit einem Operatorbegriff arbeiten, der Eigenschaften des mathematischen wie des fertigungstechnischen Operatorbegriffs in sich vereinigt. Vom mathematischen Operator übernimmt er die Natur der Operanden im Sinne der Mathematik. Jeder Operator der Hierarchie, auch das System als Ganzes, verarbeitet Zeichenketten. Sämtliche Ein- und Ausgabeoperanden sind Zeichenketten. Einen solchen Operator nennen wir **Zeichenkettenoperator**.

Vom Fertigungsoperator soll unser Operator die Fähigkeit übernehmen, selbst zu agieren, seine Operation selbst auszuführen. Beide Eigenschaften sind Voraussetzungen dafür, dass unser System zur *aktiven sprachlichen Modellierung* befähigt werden kann.

---

2 Hier und im Weiteren verwenden wir den Begriff des mathematischen Operators im weiten Sinne, d.h. nicht eingeschränkt auf die Überführung von Funktionen in Funktionen.

Etwas verwirrend ist der Umstand, dass der Begriff des Zeichenkettenoperators in der Rechentechnik nicht nur im *agierenden*, sondern auch im *nichtagierenden* Sinne verwendet wird. Je nach Kontext kann ein Operator ein Mensch, ein Gerät oder eine Vorschrift sein. Zur Unterscheidung führen wir die Begriffe des *realen* und des *sprachlichen* Operators ein und definieren:

Ein ***Operator*** *ist ein Mensch, eine Einrichtung (ein Gerät) oder eine Vorschrift, der/die einem Eingabeoperanden* x *einen Ausgabeoperanden* y *zuordnet. Wird die Zuordnung durch einen Menschen oder ein Gerät vorgenommen, heißt er* ***realer Operator****; wird sie durch eine Vorschrift festgelegt, heißt er* ***sprachlicher Operator***.[3]

Die Operatordefinition sagt nichts darüber aus, ob einem bestimmten x stets dasselbe y zugeordnet wird, m.a.W. ob die Zuordnung *eindeutig* ist. Wenn sie eindeutig ist, d.h. wenn sie jedem x genau ein y zuordnet, wird die Gesamtheit aller Zuordnungen **Abbildung** oder **Funktion** genannt[4]. Zwei Notationsweisen sind üblich:

$$y = f(x), \tag{8.2a}$$

$$f : X \to Y. \tag{8.2b}$$

X bezeichnet die Menge aller x und Y die Menge aller y. Wir sagen, dass ein Operator, der die eindeutige Zuordnung (8.2) trifft, die Funktion f *realisiert*, und nennen ihn f-Operator. In der Mathematik ist es üblich, Bezeichner von Variablen kursiv zu schreiben, z.B. x und ein bestimmtes x als "Wert von x" zu bezeichnen.

Wenn wir ein informationelles System als Operatorenhierarchie entwerfen wollen, müssen wir offenbar verlangen, dass jeder Operator der Hierarchie eine eindeutige Abbildung realisiert, denn das System soll auf ein und denselben Auftrag immer gleich reagieren, es soll zu einer bestimmten Aufgabe stets dasselbe Ergebnis liefern und ein und dieselbe Frage stets gleich beantworten, allgemein, die Zuordnung der Ausgaben zu den Eingaben soll eindeutig sein. Nun lassen sich Gründe dafür angeben, dass die Zuordnungen eventuell nicht eindeutig sind. Drei Gründe seinen genannt:

1. Der Operator ist **steuerbar**. Das heißt, dass der Operator in der Lage ist, verschiedene Zuordnungen zu treffen, also verschiedene Operationen auszuführen. Vor einer Operationsausführung muss er durch ein oder mehrere Steuersignale auf eine bestimmte Operation *konditioniert* (eingestellt) werden. Der f-Operator in Bild 8.1 ist ein Beispiel dafür; er kann auf die Berechnung von f_1 oder f_2 konditioniert

---

3 Wir ziehen die Adjektive "real" und "sprachlich" den in der Literatur häufig verwendeten Adjektiven "physisch" und "logisch" vor.

4 In der Literatur wird zuweilen ein Unterschied zwischen Abbildung und Funktion gemacht. Entweder wird der Abbildungsbegriff allgemeiner oder der Funktionsbegriff wird spezieller definiert, z.B. als Abbildung zwischen Mengen von Zahlen oder als Abbildung, die durch eine Berechnungsvorschrift oder durch ein Prädikat (s.u.) festgelegt ist.

werden. Diese Art einer nichteindeutigen Abbildung wird zu einer eindeutigen Abbildung, wenn einem Eingabeoperanden *x* zusammen mit einem Steuersignal *u* (einem Kommando in Bild 8.1b) stets ein und dasselbe Resultat *y* zugeordnet wird. Dann realisiert der Operator eine Funktion

$$\text{f: } X \times U \rightarrow Y \tag{8.3}$$

Auf der linken Seite des Pfeils ist die Menge aller Paare (x,u) angegeben, notiert als sogenanntes *kartesisches Produkt* der Mengen *X* und *U*, wobei *U* die Menge aller Steuersignale (Kommanndos) *u* bezeichnet.

2. Der Operator besitzt ein **Gedächtnis**. Das bedeutet, dass außer dem momentanen auch frühere Eingabeoperanden darauf einwirken können, welches Resultat ausgegeben wird. Es stellt sich die Frage, unter welchen Bedingungen ein Operator mit Gedächtnis wie ein Operator ohne Gedächtnis behandelt werden kann, m.a.W. ob sich die Verhaltensweise eines Operators trotz seines Gedächtnisses als Funktion darstellen lässt. Es liegt der Gedanke nahe zu versuchen, die Notation (8.3) anzuwenden, worin *U* die Vergangenheit eindeutig charakterisieren müsste. Wenn wir eindeutig arbeitende informationelle Systeme als Operatorenhierarchien realisieren wollen, muss eine Notation der Form möglich sein, denn ein Kompositoperator enthält Speicherplätze für die Operanden, er ist also ein Operator mit Gedächtnis. Auf einer höheren Abstraktionsebene wird das Problem mit Hilfe des Automatenbegriffs behandelt (siehe Kap.8.2.3).

3

3. Der Operator ist ein **nichtdeterministischer** Operator. Ein Operator heißt **deterministisch**, wenn er unter gleichen Bedingungen (gleicher Eingabeoperand, gleiche Steuerung, gleiche Vorgeschichte) ein und demselben *x* stets das gleiche *y* zuordnet. Anderfalls heißt er **nichtdeterministisch**. Es lässt sich nicht vorhersagen, welches *y* ein nichtdeterministischer Operator einem gegebenen *x* zuordnet. Falls angegeben werden kann, mit welchen Wahrscheinlichkeiten die verschiedenen möglichen Ausgaben erfolgen, heißt der Operator **stochastisch**. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter einem Operator stets ein *deterministischer* Operator zu verstehen.

Hinsichtlich der ersten der genannten Interpretationen von (8.3) ist eine Ergänzung erforderlich. Bei Anwendung von (8.3) auf den Operator f-op von Bild 8.1b enthält *U* zwei Elemente, u_1 und u_2. Diese Steuersignale können dem f-sop eingegeben werden, der gemäß dem empfangenen Steuersignal die Berechnung von f_1 bzw. f_2 steuert, indem er die entsprechenden Steuersignale für die Weichensteuerung generiert. Offensichtlich ist (8.3) auch hinsichtlich dieser, vom f-sop generierten Signale anwendbar, also derjenigen Steuersignale, die nicht dem f-sop, sondern dem f-aop (dem ON) eingegeben werden. Die Elemente von *U* sind dann keine Steuersignale, sondern Tupel von Steuersignalen bzw. zeitliche Folgen solcher Tupel. Im Falle eines Operatorennetzes mit vielen steuerbaren Flussknoten können die einzelnen Steuersignaltupel und die Tupelfolgen sehr lang werden. Wenn der Steueroperator ein *interpretierender* Operator ist, der die Steuersignale für eine Operationsaus-

führung nach Maßgabe einer Operationsvorschrift (eines *sprachlichen* Operators, eines Programms) generiert, dann kann *U* eine Menge von Programmen bezeichnen.

Wenn ein sprachlicher Operator als Operationsvorschrift interpretiert werden soll, muss er den *Prozess* der Operationsausführung vollständig beschreiben. Das wirft eine unerwartete Frage auf. Prozesse verlaufen in der Zeit. Die Zeit muss also in der Beschreibung enthalten sein. Wo aber verbirgt sich die Zeit in einem Operandenflussplan, in einem sprachlichen Operator, in einem Algorithmus, in einem Computerprogramm, einem mathematischen Ausdruck oder gar in einem Operationssymbol?

Zur Beantwortung dieser Frage betrachten wir einen nichtdekomponierten Operator im Sinne der Operatorabstraktion als "schwarzen Kasten", d.h. wir abstrahieren von allem, was *in* ihm vorgeht. Das einzige, was wir "sehen", ist das Erscheinen von Operanden an seinem Eingang und Ausgang. Die *Ereignisse* des Auftretens von Operanden treten in bestimmten Zeitpunkten ein, durch welche die Zeit zerteilt (*segmentiert*) wird. Die Zeit kann als Folge dieser Ereigniszeitpunkte, beschrieben werden. Dies ist der Kern einer viel verwendeten Methode, den zeitlichen Ablauf von Prozessen zu beschreiben. Man spricht zuweilen von *ereignisorientierter Diskretisierung* der Zeit und von **ereignisorientierter Prozessbeschreibung**.

4 Wesentlich für diese Beschreibungsmethode ist ihre *kausale* Grundlage. Der Ausgabeoperand eines realen Operators ist die *kausale Folge* des Eingabeoperanden. Der Eingabeoperand ist die *Ursache*, der Ausgabeoperand die *Wirkung*. Dabei "*überspringt*" der kausale Zusammenhang die Prozesse im schwarzen Kasten. Die Beschreibung ist **kausaldiskret**.

Der *kausale* Zusammenhang zwischen Ein- und Ausgabeoperanden besteht auch im Falle sprachlicher Operatoren, wenn diese als Operationsvorschriften interpretiert werden. Denn sprachliche Operatoren werden zusammen mit ihren Interpretierern zu realen Operatoren und durch die Interpretation werden sprachliche Operanden zu realen Operanden. Eben dies ist gemeint, wenn von *Realisierung* sprachlicher Operatoren, Operationen und Operanden gesprochen wird.

Die Aussage des letzten Absatzes ist von tiefer, man kann sogar sagen von

5 philosophischer Bedeutung. Sie bedeutet nämlich, dass ***logische*** *Zusammenhänge im Grunde* ***kausale*** *Zusammenhänge sind*. Dieser Sachverhalt ist die Folge des Umstandes, dass ein sprachlich (z.B. mathematisch) artikulierter "logischer" Zusammenhang erst dann Sinn erhält, wenn er von einem realen Interpretator interpretiert wird. Die Schlussfolgerung mag überraschen und vielleicht sogar Protest hervorrufen. Wir werden uns mit ihr in Kap.8.2.5 auseinandersetzen.

Der Begriff der *kausaldiskreten* Prozessbeschreibung wurde bereits in Kap.4.2 [4.4] eingeführt, wo er der *kausalkontinuierlichen* Beschreibung gegenübergestellt wurde. Vergegenwärtigt man sich noch einmal die dortigen Überlegungen, erkennt man, dass Prozesse der Informationsverarbeitung nicht kausalkontinuierlich beschrieben werden müssen, sondern dass die kausaldiskrete Beschreibung ausreicht. Einem Ereignis, das einen Zeitpunkt der kausaldiskreten Beschreibung festlegt,

entspricht der Übergang des Trägermediums in einen neuen codierenden Zustand. Das wird uns im Weiteren noch beschäftigen. Im Augenblick konstatieren wir, dass Prozesse nur dann als Träger informationeller Prozesse geeignet sind, wenn sie kausaldiskret beschreibbar sind, und treffen folgende Vereinbarung: *Die Beschreibung des Ablaufs von Prozessen heißt* ***kausaldiskret****, wenn die Zeit durch Ereignisse des Auftretens von Operanden segmentiert wird und wenn jedes Ereignis die* ***kausale*** *Folge eines oder mehrerer vorangehender Ereignisse ist*. Danach ist die algorithmische Beschreibung und ebenso die Beschreibung durch einen Operandenflussplan eine kausaldiskrete Beschreibung der Ausführung einer Kompositoperation.

Die Bezeichnung "kausaldiskret" übertragen wir auf die beschriebenen Prozesse selber und werden (verkürzt) von *kausaldiskreten Operationen* und *Prozessen* sprechen, obwohl diese nicht selber, sondern ihre Beschreibungen kausaldiskret sind. Das kann zu keinen Missverständnissen führen, zumal es in Wirklichkeit gar keine kausaldiskreten Prozesse gibt, wenigstens nicht im Rahmen der klassischen Physik. Der Ursache-Wirkung-Zusammenhang ist im Grunde immer ein kontinuierlicher und kann seit NEWTON als solcher mit Hilfe der Infinitesimalrechnung beschrieben werden. Weiterhin übertragen wir die Bezeichnung auch auf Operatoren und nennen einen Operator kausaldiskret, wenn in ihm ein kausaldiskret *beschriebener* Prozess abläuft.

Die USB-Methode ist, soweit wir sie bisher kennengelernt haben, eine kausaldiskrete Beschreibung von Prozessen der Zeichenkettenverarbeitung. Bild 8.3 zeigt als weiteres Beispiel für die Anwendung der Methode das Operatorennetz eines dekomponierten, kausaldiskreten Operators, dessen Bausteinoperatoren im Gegesatz zu dem Netz von Bild 8.1 keine Zeichenkettenoperatoren, sondern *Fertigungsoperatoren* (*Werkstückoperatoren*) sind. Das Netz besitzt fast die gleiche Struktur, wie das von Bild 8.1a.

Der in Bild 8.3 dargestellte Kompositoperator dient der Herstellung von zwei verschiedenen Werkstücken. Zunächst zersägt (oder zerschlägt) der Trennautomat

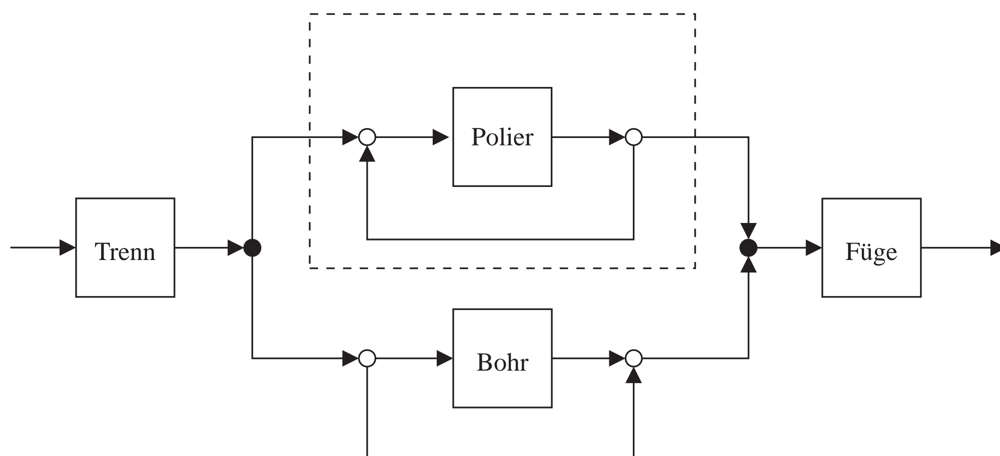


Bild 8.3: Operatorennetz eines Fertigungsoperators

(Trenn) ein Eisenblech in zwei Teile, die später der Fügeautomat (Füge) zu einem Eisenwinkel zusammenfügt, z.B. schweißt. Trenn liefert ein Operandenpaar, das in der nachfolgenden Spaltegabel aufgeteilt wird. Gabeln ohne Trennoperator, wie etwa die Gabelung des Leiters in Bild 8.1, kann es in Fertigungssystemen nicht geben.

Auf dem Wege zum Fügeoperator wird das eine Blech poliert (evtl. mehrmals) und das andere je nach Weichenstellung (je nach gewünschtem Produkttyp) entweder mit Bohrungen versehen oder unbearbeitet weitergeleitet. Vor dem Fügen muss eventuell das eine Blech auf das andere warten. Die Vereinung fungiert also ebenso wie in Bild 8.1a gleichzeitig als Synchronisierer.

Die Prozesse des Polierens und des Bohrens sind voneinander unabhängig, keiner ist die (kausale) Voraussetzung des anderen. Dasselbe gilt für das Potenzieren und das Berechnen der Sinusfunktion in Bild 8.1. Solche Prozesse heißen **nebenläufig**. Nebenläufige Prozesse können parallel (gleichzeitig) ausgeführt werden, wodurch die Arbeitsgeschwindigkeit eines Operators eventuell gesteigert werden kann.

Wenn beispielsweise das Bohren mehr Zeit in Anspruch nimmt als das Polieren, kann die Produktivität des Kompositoperators dadurch gesteigert werden, dass während des Bohrens gleichzeitig Eisenwinkel ohne Bohrungen hergestellt werden. Man hat es dann mit **parallelen** Prozessen zu tun. Dabei kann es vor der Sammel-
6 weiche nach dem Bohrer zu einem **Konflikt** kommen, wenn gleichzeitig zwei Operanden eintreffen, die von der Weiche nur der Reihe nach durchgelassen werden können. Die Sammelweiche fungiert dann als **Sequenzierer**. Vor einer sequenzierenden Sammelweiche muss eventuell ein Operand solange warten, bis für ihn die Weiche geöffnet wird. Das ist der Fall, sobald der nachfolgende Operator die zur Zeit laufende Operation beendet hat. Für das Warten muss eine Ablage (Operandenplatz, Speicher) vorgesehen sein, ebenso wie vor der Vereinung. In Bild 8.3 sind die Ablagen nicht eingezeichnet.

Wenn in Analogie hierzu die Berechnung der Sinusfunktion in Bild 8.1a länger dauert als das Potenzieren, kann eventuell die Rechengeschwindigkeit dadurch erhöht werden, dass der Operandenfluss durch die beiden Wege der Masche parallel hindurchgeleitet wird und während *einer* Operationsausführung des Sinusoperators *mehrere* Werte der Funktion f_1 (siehe (8.1)) berechnet werden. Die Sammelweiche nach dem Sinusoperator muss dann als Sequenzierer fungieren, und in ihren beiden Eingabeleitungen müssen Speicher vorgesehen sein.

Die beiden Operatorennetze in den Bildern 8.1 und 8.3 beschreiben Systeme ganz unterschiedlicher Natur, ein informationelles System und ein Fertigungssystem. Dennoch stimmen die Beschreibungen in zwei Punkten überein, sie sind beide kausaldiskret, und sie verwenden beide die gleichen Komponierungsmittel, nämlich die Flussknoten von Bild 8.2. Diese Übereinstimmung rechtfertigt die Bezeichnung
7 "*uniforme* Systembeschreibung". *Die **USB-Methode** ist auf jeden kausaldiskret beschreibbaren Prozess anwendbar*.

Die Methode kommt den Denkgewohnheiten von Informatikern einerseits und von Systemingenieuren andrerseits entgegen. Unter *Systemingenieuren* sollen hier

Ingenieure verstanden werden, die reale Systeme, die sich kausaldiskret beschreiben lassen, entwerfen, bauen oder warten. Für die Verständigung zwischen Systemingenieuren und Informatikern liefert die USB-Methode eine geeignete sprachliche Grundlage. Eine schnelle und fehlerfreie Verständigung kann für eine Produktion, in der Systemingenieure auf die Zusammenarbeit mit Informatikern angewiesen sind, von großer Bedeutung sein.

Neben der praktischen Stärke der USB-Methode kann ihre theoretische Schwäche nicht übersehen werden. Ihr Abstraktionsgrad reicht nicht aus, um kausaldiskrete Prozesse theoretisch zu untersuchen; doch reicht er aus, um einen Berechenbarkeitsbegriff zu definieren und ihn mit entsprechenden anderen Begriffen zu vergleichen. Darauf werden wir im Weiteren ausführlich eingehen. In Vorbereitung darauf wird in Kap.8.3 der Begriff der Berechenbarkeit eingeführt. Zuvor sollen noch zwei wichtige Methoden zur Sprache kommen, die ebenfalls der kausaldiskreten Prozessbeschreibung dienen, jedoch auf einer höheren Abstraktionsebene, als es bisher geschehen ist.

8.2.2 Petrinetze

Zunächst vergegenwärtigen wir uns, welche Rolle die Zeit in einer Prozessbeschreibung durch einen Algorithmus bzw. durch ein Operatorennetz spielt. Ein Algorithmus im ursprünglichen Sinne des Wortes sequenziert die Bausteinoperationen einer Kompositoperation vollständig. Bei der Operationsausführung gibt es keine gleichzeitige Operationsausführung, keine *parallelen Prozesse*. Die Bausteinprozesse sind *zeitlich vollständig geordnet*, sie bilden eine **vollständige zeitliche Ordnung**. In einem Operatorennetz dagegen können nebenläufige Operationen parallel (gleichzeitig) ausgeführt werden. Man sagt dann, dass die Bausteinprozesse eine **zeitliche Halbordnung** bilden. In jedem Fall beschränkt sich die Rolle der Zeit auf die einer "Ordnungsstifterin". Eine andere Rolle spielt die Zeit nicht.

Bei mehrfacher, insbesondere bei geschachtelter Parallelität (Parallelität in geschachtelten Maschen) kann der zeitliche Ablauf (die zeitliche Ordnung) sehr unübersichtlich werden. Es kann zu kritischen Situationen kommen. Vor einer Vereinung (Synchronisierung) oder vor einer Sammelweiche (Sequenzierung) können sich die Operanden stauen und eine *Warteschlange* bilden (wie im Straßenverkehr). Es kann zu *Konflikten* und sogar zu *Blockierungen* kommen und zwar dann, wenn eine Bedingung für die Fortsetzung des laufenden Prozesses nicht erfüllt werden kann, z.B. wenn von zwei Operatoren jeder auf den Ausgabeoperanden des anderen wartet.

Konflikte können den Ablauf paralleler Prozesse empfindlich stören. Das gilt auch für informationelle Prozesse. Um derartige Störungen vorauszusehen und nach Möglichkeit auszuschließen, sind spezielle Analysemethoden entwickelt worden. Eine von ihnen ist die *Petrinetz-Methode*. Sie ist der USB-Methode verwandt, jedoch abstrakter und für theoretische Untersuchungen geeigneter. Sie bietet sich an, wenn man wissen möchte, ob der Prozessablauf in einem Operatorennetz infolge paralleler Operationsausführung behindert werden oder sogar zum Erliegen kommen kann. Die

Petrinetz-Methode beschreibt ausschließlich die zeitliche Ordnung der Bausteinprozesse und abstrahiert von allem anderen. Die Methode soll in aller Kürze beschrieben werden.

Es sind zwei Arten von Bedingungen zu unterscheiden, die erfüllt sein müssen, damit eine Operation ausgeführt werden kann, Operandenbedingungen und Operatorbedingungen. Die Operandenbedingung einer Operation ist erfüllt, wenn die erforderlichen Eingabeoperanden vorhanden sind. Die Operatorbedingung ist erfüllt, wenn der Operator, der Träger der Operation, frei ist. Von dem Unterschied zwischen beiden Arten von Bedingungen wird abstrahiert.

Außerdem wird davon abstrahiert, wodurch und auf welchem Wege Bedingungen erfüllt werden. Es wird lediglich das *Ereignis* betrachtet, *dass* eine Bedingung erfüllt ist, die vorher nicht erfüllt war, m.a.W. dass ein neuer **Bedingungszustand** eingetreten ist. Zur Fixierung dieser Idee wird der Begriff der **Transition** (in gewissem Sinne eine Idealisierung des Operatorbegriffs) eingeführt und gesagt: Wenn eine Transition *schaltet*, geht das Netz in einen neuen Bedingungszustand über. Ein Prozess wird als Folge von *Schaltereignissen* beschrieben. Die Beschreibung ist kausaldiskret.

Bild 8.4 demonstriert an zwei Beispielen die übliche graphische Veranschaulichung dieser Vorstellungs- und Redeweise. Transitionen werden durch Quadrate (zuweilen auch durch kurze fette Querstriche) und Bedingungen durch Kreise dargestellt, die **Plätze** genannt werden. Wenn ein Platz mit einer **Marke** (einem dicken Punkt) belegt ist, bedeutet dies, dass die betreffende Bedingung erfüllt ist. Zwischen zwei Plätzen liegt jeweils eine Transition und zwischen zwei Transitionen ein Platz. Wenn alle Eingabeplätze einer Transition belegt sind, kann sie "*schalten*" (oft wird auch "*feuern*" gesagt). Dabei wird den Eingabeplätzen je eine Marke entnommen und jeder Ausgabeplatz wird mit je einer Marke belegt. Wenn ein Petrinetz mit einer ausreichenden Anzahl von Marken gestartet wird, kommt es zu einem Markenfluss, d.h. zu einem *Fluss erfüllter Bedingungen.*

Eine Transition kann als *Markenoperator* und ein Petrinetz als *Markenoperatorennetz* aufgefasst werden, dessen Weichen in die Plätze und dessen Vereinungen und Gabeln in die Transitionen integriert sind. Die Steuerung des Markenflusses wird durch die Darstellung *nicht* erfasst.

Wie unschwer zu erkennen ist, befindet sich das Petrinetz von Bild 8.4a in einem blockierten Zustand; er wird in der Betriebssystemtechnik auch **Deadlock** genannt ("tödlicher" Konflikt). Jede Transition wartet darauf, dass die andere den mittleren Platz mit einer Marke belegt. Der dargestellte Bedingungszustand lässt sich durch die unterschiedlichsten realen Situationen interpretieren.

8 Zur Anregung der Phantasie des Lesers seien einige Situationen genannt, die sich durch Bild 8.4a beschreiben lassen.

- Zwei Autos begegnen sich auf einer Brücke, die kein Ausweichen erlaubt. Einer versperrten Wegstrecke entspricht im Petrinetz ein Platz ohne Marke. In den

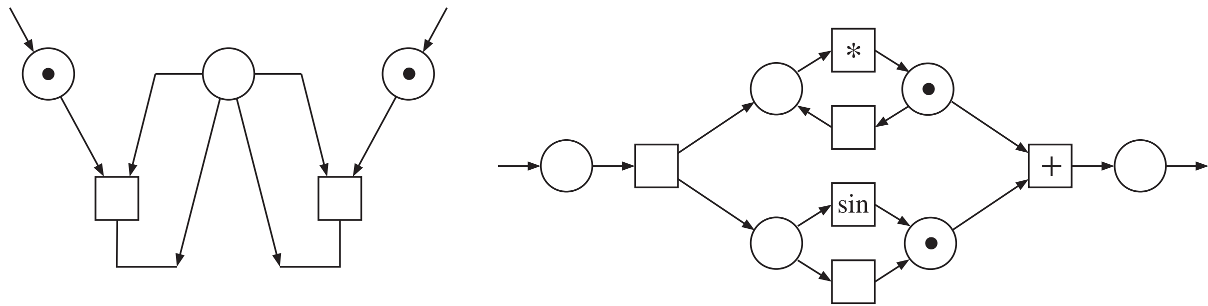


Bild 8.4 Petrinetze. **(a)** - Blockierungssituation dargestellt mittels Petrinetz; **(b)** - Petrinetz zu den Operatorennetzen der Bilder 8.1 und 8.3.

weiteren Beispielen werden die Wegstrecken auf der Brücke durch andere Objekte ersetzt (Werkzeuge, Gabeln, Geld, Betriebsmittel).

- Auf Montage ziehen zwei Monteure je ein Paar von Kontermuttern an. Dafür benötigt jeder von ihnen die beiden einzigen vorhandenen Schraubenschlüssel. Jeder hält bereits einen in der Hand.
- Wie im vorangehenden Beispiel, doch sind es zwei Fischesser (Fisch wird mit zwei Gabeln gegessen).
- Eine vorhandene Geldmenge ist für den Bau von zwei Fabriken ausgegeben worden. Um die Produktion aufnehmen zu können, benötigen beide einen zusätzlichen Betrag, der durch die Produktion erwirtschaftet werden könnte.
- In einem Computer laufen zwei Rechenprozesse ab. Beiden ist Speicherplatz zugeteilt worden. Beide benötigen zur Beendigung der Rechnung zusätzlichen Speicherplatz, der nicht vorhanden ist.

Die Vielfältigkeit der Beispiele macht die Bedeutung von Petrinetzen als Analysemittel unterschiedlichster Prozesse, insbesondere auch informationeller Prozesse verständlich. Um das noch deutlicher zu machen, soll Bild 8.4a unter Verwendung des Wortes *Betriebsmittel* kommentiert werden. *Unter **Betriebsmittel** oder **Ressource** wird in der Informatik jede Komponente der Hardware oder Software eines informationellen Systems verstanden, die der Ausführung von Operationen durch das System dient*. Zu den Betriebsmitteln gehören Programme, Daten (Dateien), Prozessoren, Speicher, Signale sowie Ein- und Ausgabegeräte.

Den Transitionen von Bild 8.4a mögen zwei Prozesse (Ausführungen von zwei Programmen) entsprechen, die gleichzeitig in einem Computer ausgeführt werden. Die Plätze entsprechen benötigten und die Marken verfügbaren Betriebsmitteln. Jeder der beiden Prozesse benötigt zwei Ressourcen, jedem Prozess ist bereits eine Ressource fest zugewiesen und beide warten auf eine gemeinsame Ressource, die

aber nicht verfügbar ist, solange nicht einer der beiden Prozesse stattgefunden (eine der beiden Transitionen geschaltet) hat. Das Betriebsmittel, auf das gewartet wird, kann z.B. das Resultat einer Rechnung sein oder - wie in obigem Beispiel - ein Speicherbereich, der bei Beendigung des einen oder anderen Prozesses frei wird.

9 Blockierungen müssen - wie alle Konflikte - durch Absprache (*dezentrale Steuerung*) oder durch eine höhere Instanz (*zentrale Steuerung*) gelöst werden. Das kann bei komplizierten und stark vernetzten Prozessen schwierig sein. Vernünftigerweise entwirft man ein System von vornherein so, dass derartige Schwierigkeiten nicht auftreten können. Dies ist eines der Ziele, denen die Theorie der Petrinetze und ihre Implementierung dient. Implementierung der Theorie bedeutet: Installation von Programmen auf einem Computer, die es erlauben, den Markenfluss in umfangreichen Petrinetzen nach den Vorgaben der Theorie durchzuspielen[5]. Auf diese Weise lässt sich voraussagen (berechnen oder simulieren), ob in einem vorgegebenen Netz gestartete Prozesse zu Blockierungen führen können.

Bild 8.4b zeigt das Petrinetz zum Operatorennetz (Operandenflussgraphen) von Bild 8.1. Es ist gleichzeitig das Petrinetz zum Operatorennetz von Bild 8.3. Die Operationssymbole *, sin und + sind in die entsprechenden Transitionen eingetragen. Wir gehen davon aus, dass für jede Operation ein spezieller Operator zur Verfügung steht. Das bedeutet, dass die Bedingungen ausschließlich in der Verfügbarkeit der jeweiligen Operanden bestehen. Hinter den Marken verbergen sich also Operanden und mit den Marken bewegen sich Operanden durch das Netz (obwohl Marken begrifflich etwas ganz anderes sind als Operanden). Dies ist der Grund dafür, dass Petrinetz und Operandenflussgraph einander recht ähnlich, wenn auch nicht miteinander identisch sind. Die Gabel in Bild 8.1a (ihr entspricht in Bild 8.3 der Trennoperator) ist im Petrinetz zu einer Transition geworden. Außerdem enthält das Petrinetz zwei zusätzliche Transitionen, denen keine Operatoren, sondern Operanden*wege* entsprechen und zwar die Rückkopplung vom Ausgang zum Eingang des Multiplizieres bzw. Polierers und die Vorkopplung, die den Sinusoperator bzw. Bohrer überbrückt. Die zusätzlichen Transitionen müssen eingefügt werden, um die Vorschrift zu erfüllen, dass zwischen zwei Plätzen eine Transition liegen muss. Die Markenbelegung entspricht einer Situation, in der ein Operand gerade den Multiplizierer durchlaufen und ein anderer den Sinusoperator durch- bzw. umlaufen hat. Es kann entweder die dem Addierer entsprechende Transition oder die unter dem Multiplizierer liegende Transition schalten.

Infolge der Ähnlichkeit von Petrinetz und Operandenflussgraph ist die Analyse des Petrinetzes in diesem Falle von begrenztem Wert. Zwar treten die Besonderheiten des Prozessablaufs im Petrinetz augenfälliger zutage, wie beispielsweise die Abwesenheit von Deadlocks, doch sind sie durchaus auch dem Operandenflussgraph zu entnehmen. Nichtsdestoweniger wird in Kap.20.3 bei der Überführung des Operan-

5 Siehe z.B. [Reisig 90]

denflussgraphen von Bild 8.1 in ein Programm ein Petrinetz wertvolle Dienste leisten. Dabei handelt es sich allerdings um ein *steuerbares* Petrinetz (Bild 20.10), das gegenüber dem Petrinetz von Bild 8.4b um *Steuertransitionen* erweitert ist.

Mit Hilfe von Petrinetzen lassen sich Operatorennetze nicht nur hinsichtlich der Gefahr von Blockierungen, sondern auch hinsichtlich anderer Eigenschaften analysieren. Beispielsweise kann untersucht werden, ob die Anzahl der Marken in einem markierten Petrinetz (einem Netz dessen Plätze teilweise mit Marken belegt sind) zunimmt und sich die Marken auf diesem oder jenem Platz ansammeln (Warteschlangenbildung). Es kann auch untersucht werden, ob die Anzahl der Marken mit der Zeit abnimmt, ob der eine oder andere Platz eventuell niemals belegt werden kann oder ob der Markenfluss irgendwann völlig versiegt ("stirbt"). Es kann auch untersucht werden, ob der Markenfluss in einem markierten Netz niemals abbricht. Ist dies der Fall, sagt man, dass das markierte Petrinetz *lebendig* ist.

Im Petrinetz haben wir ein Werkzeug zur kausaldiskreten Beschreibung nebenläufiger Prozesse kennen gelernt, also solcher Prozesse, die parallel in einem Kompositoperator ablaufen können. Dabei haben wir von den Operationen selber abstrahiert und nur die möglichen Reihenfolgen ihrer Ausführung betrachtet. Im folgenden Kapitel werden wir einen in gewissem Sinne "entgegegesetzten" Standpunkt beziehen. Wir werden von der Reihenfolge der Bausteinoperationen abstrahieren, uns aber für die Operationen selber interessieren. Operanden und Operandenplätze, die in Petrinetzen nicht in Erscheinung traten, werden demzufolge eine wesentliche Rolle spielen.

8.2.3 Abstrakter Automat

In Kap. 8.2.1 [3] wurde festgestellt, dass sich die Verhaltensweise von Kompositoperatoren als Funktion darstellen lassen muss, wenn als Operatorenhierarchie konzipierte informationelle Systeme eindeutig arbeiten sollen. Das ist insofern problematisch, als Kompositoperatoren Speicher enthalten, also ein Gedächtnis besitzen, sodass die Abbildung $X \rightarrow Y$ i.Allg. nicht eindeutig ist. Diesem Problem wenden wir uns nun zu.

Speicherplätze sind überall da erforderlich, wo Operanden eventuell warten müssen. Das ist, wie wir gesehen hatten, vor Vereinungen und Sammelweichen der Fall. Ferner kann sich vor jedem Operator eine Warteschlange bilden, wenn seine Operationsdauer größer ist, als die der vorangehenden Operatoren. Das bedeutet, dass Kompositoperatoren, insbesondere steuerbare, in aller Regel Operatoren mit Gedächtnis sind. Damit ergibt sich folgendes Problem. Unser erklärtes Ziel ist es, durch hierarchische Komponierung Systeme (Kompositoperatoren) zu schaffen, die eindeutige Abbildungen *Eingabemenge* $\rightarrow$ *Ausgabemenge* realisieren. Die Eindeutigkeit der Abbildung darf nicht durch die internen Gedächtnisse der Operatoren der Hierarchie zunichte gemacht werden.

Es gibt eine rigorose Methode, das Problem zu lösen. Sie besteht darin, vor jedem Start einer Kompositoperation sämtliche Speicherplätze mit einem ein für allemal

vorgeschriebenen Initialwert zu belegen, z.B. mit 0, d.h. die Speicher zu löschen, und außerdem dafür zu sorgen, dass jeder Speicherplatz während einer Kompositoperation nur ein einziges Mal belegt wird. Unter dieser Bedingung realisiert der Kompositoperator trotz seines Gedächtnisses eine eindeutige Abbildung und kann wie ein Operator ohne Gedächtnis behandelt werden. Diese Methode kommt in der Rechentechnik auf Schritt und Tritt zur Anwendung (siehe Kap.19.5), obwohl sie der Idee der Operatorennetze und damit der USB-Methode widerspricht.

Es gibt eine andere Möglichkeit, sich von dem Gedächtnis zu "befreien", das in Form von Speichern über das Operatorennetz eines Kompositoperators verteilt ist. Die Befreiung ist zwar nicht vollständig, doch ausreichend, um theoretische Untersuchungen in weit umfangreicherem Maße anstellen zu können, als die USB-Methode sie erlaubt. Zwei Ideen liegen der nun zu behandelnden Methode zur Beschreibung kausaldiskreter Prozesse zugrunde.

Die **erste Idee** besteht darin, die einzelnen Speicherplätze, die das Operatorennetz eines Kompositoperators enthält, gedanklich zu einem einzigen Speicher zusammenzufassen und aus dem Netz herauszuziehen, ohne die einzelnen Verbindungen aufzutrennen. Dann ergibt sich Bild 8.5. Das ursprüngliche Netz möge n Bausteinoperatoren mit je einem Ausgabespeicherplatz enthalten. Es reicht aus, nur diese herauszuziehen, denn selbst wenn weitere Speicherplätze in dem Netz existieren, enthalten diese keine zusätzlichen Operanden.

Auch die Ein- und Ausgabeverbindungen der einzelnen Speicherplätze werden gedanklich zusammengefasst, sodass die Eingabeleitung und die Ausgabeleitung des herausgezogenen Speichers aus je n Einzelleitungen besteht. In dem Speicher ist in jedem Moment ein Tupel von n Operanden gespeichert. Dieses n-Tupel heißt **innerer Zustand** des Kompsitoperators und wird mit z bezeichnet. Der innere Zustand wird gedanklich über eine Rückkopplung auf den Eingang zurückgeführt, wie in Bild 8.5 gezeigt ist.

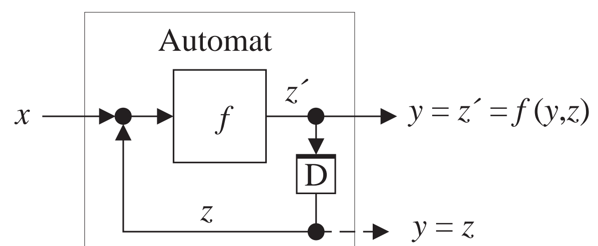


Bild 8.5 Operatorennetz des abstrakten Automaten. f - Folgefunktion; D - Taktverzögerer (Delay).

Auf den Eingang des mit f bezeichneten Operators wird ein Paar gegeben, das Paar (x,z), ähnlich wie im Falle eines steuerbaren Operators. Doch tritt an die Stelle des Steuersignals u der innere Zustand z. Wie aber lässt sich erreichen, dass der f-Operator eine eindeutige Abbildung $X \times Z \to Y$ realisiert? Man bedenke, dass die einzelnen Elemente von z zeitlich völlig ungeordnet im Speicher eintreffen, sodass der Begriff des Tupels genau genommen keinen Sinn hat.

Die **zweite Idee** bringt die Lösung des Dilemmas: Sämtliche Bausteinoperatoren arbeiten *synchron*, d.h. sie werden gleichzeitig gestartet. Der nächste Start erfolgt, nachdem alle Bausteinoperationen ausgeführt und die Resultate sich als Komponente von z im gemeinsamen Speicher befinden. Eine solche Arbeitsweise heißt **getaktet**. Der Speicher spielt dabei die Rolle eines **Taktverzögerers**. Die gesamte Anordnung heißt **abstrakter Automat**, was darauf hindeutet, dass es sich um eine gedankliche Konstruktion handelt. Im Gegensatz zum Petrinetz arbeitet ein abstrakter Automat von außen betrachtet rein sequenziell; es gibt keine parallelen Prozesse. Der Automat stellt einen *schwarzen Kasten* dar, dessen Innenleben - bis auf den inneren Zustand - unbekannt ist.

Die Abbildung, die ein abstrakter Automat realisiert bzw. die Funktion, die er berechnet, kann auf zweierlei Weise notiert werden:

$$X \times Z \rightarrow Z \quad \text{oder} \quad z' = f(x,z), \tag{8.4}$$

wobei z den alten und z' den neuen inneren Zustand bezeichnet. Beide Zustände sind Elemente von Z. Die Funktion f heißt **Folgefunktion**.

Hinsichtlich des Outputs y (im Zusammenhang mit Automaten wird i.Allg. von **In**- und **Output** gesprochen) sind verschiedene Vereinbarungen üblich. Für theoretische Untersuchungen wird der Output häufig mit dem alten oder mit dem neuen inneren Zustand identifiziert. Zuweilen wird für die Berechnung von y eine spezielle Funktion g eingeführt, **Ergebnisfunktion** genannt. In (8.5) sind vier verschiedene Möglichkeiten angegeben, von denen zwei in Bild 8.5 dargestellt sind.

$$\begin{aligned} y &= z \\ y &= z' \\ y &= g(z) \\ y &= g(x,z) \end{aligned} \tag{8.5}$$

Im Allgemeinen wird angenommen, dass die Mengen X, Y und Z endlich sind. Dann wird der abstrakte Automat auch **endlicher Automat** genannt. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter einem Automaten stets ein endlicher Automat zu verstehen. Die Verhaltensweise eines endlichen Automaten wird durch seine **Automatentafel** angegeben. Als Automatentafel wird die Wertetafel der Folge- und Ergebnisfunktion des Automaten bezeichnet. Jede Zeile der Tafel enthält ein (x,z)-Paar und das zugeordnete (z',y)-Paar.

Es mag befremdlich wirken, die gedankliche Konstruktion von Bild 8.5 als *Automaten* zu bezeichnen, also als etwas, das "sich selbst bewegt". Man könnte hiermit das ständige Zirkulieren der Operanden in der Rückkopplungsschleife assoziieren, insbesondere dann, wenn es sich um einen **autonomen Automaten** handelt, das heißt um einen Automaten ohne externen Eingang.

Andrerseits erinnert der *Takt*betrieb an *Takt*straßen und damit an Produktions*automaten* oder an die *Takt*frequenz eines Prozessors, was die Vermutung nahe legt, dass Prozessoren als Automaten beschreibbar sind. Wir werden in Kap.13.6 [13.13]

sehen, dass die Vermutung zutrifft. Diese Andeutungen lassen erwarten, dass die beiden Ideen, die dem Automatenbegriff zugrunde liegen, also die Zusammenfassung von Speicherplätzen und der Taktbetrieb, beim Entwurf von Rechnern Pate gestanden haben. Tatsächlich wird der Computerbau ganz entscheidend von diesen beiden Ideen mitbestimmt.

Die besondere Bedeutung des Automatenbegriffs liegt jedoch auf theoretischem Gebiet. Er ist der Grundbegriff einer eigenständigen Theorie, der **Automatentheorie**[6]. In Kap.8.4 wird die Anwendung der Automatentheorie in der Algorithmentheorie und in Kap.16.4 ihre Anwendung in der Theorie formaler Sprachen angedeutet.

Aber auch ohne den Einblick in konkrete Anwendungen erhält man eine Vorstellung von der Breite der Anwendbarkeit des Automatenbegriffs, wenn man sich in ihren Kern vertieft, nämlich in den Umstand, dass die Zukunft irgendeines sehr abstrakt zu verstehenden Objekts mit seiner Vergangenheit verknüpft wird und zwar über seinen ebenso abstrakt zu verstehenden inneren Zustand. Im inneren Zustand, in welchem sich das Objekt im gegenwärtigen Augenblick befindet, ist die gesamte Vergangenheit enthalten, soweit sie sich auf die Zukunft auswirkt, soweit sie die *Ursache* der Zukunft ist.

Diese automatentheoretische Sicht auf Ursache-Wirkungszusammenhänge stellt gewissermaßen eine auf die Spitze getriebene *kausaldiskrete* Prozessbeschreibung dar. Aus dieser Sicht kann das Verhalten jedes Objekts oder besser jedes Trägers von Prozessen betrachtet werden, ja, die ganze Welt lässt sich so betrachten. Ob einem das etwas hilft, ist eine andere Frage.

Abschließend soll diejenige Besonderheit der Prozessbeschreibung mittels Automaten noch einmal hervorgehoben werden, durch die ihre Anwendbarkeit eingeschränkt wird. Die spezielle Art, einen kausaldiskreten Prozess als *Automatenprozess* zu beschreiben, d.h. als Prozess, der in einem Automaten abläuft, zeichnet sich dadurch aus, dass jede räumliche Vorstellung bezüglich des beschriebenen Prozesses (des modellierten Originals) eliminiert ist. Für Prozessbeschreibungen nach der USB- und Petrinetz-Methode trifft das nicht zu. Denn ihnen liegt die Vorstellung eines Flusses durch ein räumliches (zwei- oder dreidimensionales) *Netz* zugrunde. 11 Abstrakter kann man den Unterschied folgendermaßen kennzeichnen. *Die netzorientierte Prozessbeschreibung ist eine* ***raum-zeitliche****, die automatenorientierte eine* ***rein zeitliche*** *Beschreibung*.

8.2.4 Elementare kausaldiskrete Operatoren

In den bisherigen Überlegungen zur kausaldiskreten Prozessbeschreibung ist der Umstand nicht zur Sprache gekommen, dass eine kausaldiskrete Operatorenhierarchie eine *unterste Schicht* haben muss. Dieser Umstand ist so selbstverständlich, dass

6 Siehe z.B. [Stetter 88],.[Sander 92], [Schöning 95]

es überflüssig scheint, ihm besondere Aufmerksamkeit zu widmen. Dass dem nicht so ist, erkennt man, wenn man der Natur der *elementaren* Operatoren und dem Wort "nichtdekomponierbar" auf den Grund geht. Dann stellen sich nämlich sofort zwei Fragen: Wodurch zeichnen sich die elementaren ("nichtdekomponierbaren") Operatoren aus und welche elementaren Ereignisse segmentieren die Zeit?

Da die Operatorenhierarchie, die wir entwerfen wollen, die menschliche Fähigkeit zum Rechnen simulieren soll, liegt es nahe zu fragen, wieweit der Mensch beim Rechnen dekomponiert. Wir wissen, dass er beim Rechnen mit *Zahlen* bis zum *Zählen* dekomponieren kann. Soweit wird sich jedoch kaum jemand "herablassen". Wenn man einen Wert der Funktion (8.1) berechnet und dazu die Bausteinoperationen von Bild 8.1 der Reihe nach per Hand ausführt, wird man z.B. beim Addieren das Dekomponieren höchstens bis zur Addition zweier einstelliger Zahlen fortsetzen. Analog wird man beim Multiplizieren bis zur Multiplikation zweier einstelliger Zahlen dekomponieren, denn das kleine Einmaleins hat man "im Kopf". Beim Rechnen im Kopf oder mit Papier und Stift dekomponiert man bis zu den *interiorisierten* (verinnerlichten, "eingefleischten", gewissermaßen zu "Reflexen" gewordenen) Operationen.

Beim Schreiben eines Computerprogramms muss bis zu denjenigen Operationen dekomponiert werden, die der Computer zur Verfügung stellt. In Kap.15.2 werden wir uns überlegen, wie ein Programm zur Berechnung der Funktion (8.1) aussehen könnte. Die elementaren Operationen, mit denen wir als Programmierer dort arbeiten werden, sind durch den Konstrukteur des Computers festgelegt.

Das führt auf die nächste Frage: Wieweit lassen sich die Hardwareoperatoren von Computern dekomponieren? Diese Frage wird uns in Kap.9.2 beschäftigen. Im Augenblick geht es uns darum, das entscheidende Charakteristikum der elementaren Operatoren herauszufinden. Um dieses zu erkennen, fragen wir zunächst, wieweit sich die Operatoren (Operationen) von Fertigungsprozessen dekomponieren lassen, z.B. die Bausteinoperatoren in Bild 8.3.

Beispielsweise kann das Bohren schrittweise erfolgen. Auch das Fügen kann aus mehreren Schritten bestehen. Wieweit ein Operator (eine Operation) bei Beibehaltung der kausaldiskreten Beschreibung dekomponiert werden kann, hängt von der Konstruktion des Operators, z.B. der betreffenden Werkzeugmaschine, ab. Die Grenze der Dekomponierbarkeit ist durch den Konstrukteur der Maschine festgelegt. Für das Polieren durch den gestrichelt umrahmten Operator in Bild 8.3 ist die Grenze z.B. dadurch festgelegt, dass es in mehreren Schritten ausgeführt und insofern kausaldiskret beschrieben werden kann, dass eine kausaldiskrete Beschreibung jedes einzelnen Schrittes jedoch ihren Sinn verliert.

Will man eine Operation noch genauer beschreiben, will man beispielsweise den Prozess des Spanabhebens beim Bohren oder den Prozess des Abtrennens kleinster Metallpartikel beim Polieren kausal beschreiben, ist man gezwungen, von der kausal*diskreten* zur kausal*kontinuierlichen* Beschreibung überzugehen. Die *Ereignisse* der kausaldiskreten Beschreibung sind Beginn und Ende eines Bearbeitungs- [12]

schrittes. In dem zu Grunde liegenden kausalkontinuierlichen Prozess existieren solche Ereignisse nicht. Damit ist die erste Frage für Werkstückoperatoren beantwortet: Es gibt aus der Sicht der kausaldiskreten Beschreibung elementare Operationen, die durch die Konstruktion der Fertigungsoperatoren festgelegt sind. Es liegt aber jedem kausaldiskreten Prozess stets ein physikalischer, kausalkontinuierlicher Prozess zugrunde.

Das Gleiche gilt für die elektrischen Prozesse in einem Computer. Sie verlaufen aus der Sicht des Physikers kontinuierlich und werden durch Differenzialgleichungen beschrieben, zumindest solange die elektrischen Parameter keine Sprünge machen. Diese können z.B. durch das Öffnen oder Schließen von Schaltern hervorgerufen werden. Tatsächlich werden wir in Kap.10 den Schalter als elementaren Hardwareoperator einführen. Das Umschalten ist aber schon eine spezielle Art von Ereignissen. Das elementare Ereignis im allgemeinsten Sinne ist der Sprung als solcher, die *Unstetigkeitsstelle* im zeitlichen Verlauf einer physikalischen Größe, z.B. einer Spannung. Unstetigkeitsstellen zerstückeln die stetige Beschreibung physikalischer Prozesse und segmentieren damit die Zeit. *Unstetigkeitsstellen erzwingen - aus der Sicht des Physikers - eine* ***stückweise*** *kontinuierliche Prozessbeschreibung; sie ermöglichen - aus der Sicht des Informatikers - eine* ***kausaldiskrete*** *Prozessbeschreibung.*

Es fragt sich, wodurch die Sprünge, die wir für eine kausaldiskrete Beschreibung brauchen, produziert werden, wenn wir das nicht selber tun, z.B. durch Ausschalten der Netzspannung. Wie kann ein kontinuierlicher Prozess *selber* einen Sprung erzeugen? Vor genau diesem Problem steht der Naturwissenschaftler, wenn er Sprünge, welche die Natur macht (Brüche, Kippvorgänge, Katastrophen), "physikalisch", kausalkontinuierlich beschreiben will.

Auch wir sind in Kap.4.1 auf dieses Problem gestoßen, als von Messen und Digitalisieren die Rede war. Wir haben es dort sozusagen ad hoc mit Hilfe einer Einrichtung gelöst, die wir *Schwellenoperator* genannt haben. Ein *Schwellenoperator* ist *definiert* als Operator, der Sprünge auf folgende Weise produziert. Sobald der Eingabewert x (vgl. Bild 4.1) einen bestimmten Wert, den sogenannten *Schwellenwert* überschreitet, führt der Ausgabewert y einen Sprung aus. Nach den zugrunde liegenden physikalischen Prozessen hatten wir nicht gefragt.

In Kap.4.1 wurden Schwellenoperatoren für den Bau von Analog-digital-Konvertern benötigt, mit deren Hilfe nicht nur das Messen analoger Werte, sondern auch das Ankoppeln eines Digitalrechners an einen Analogrechner (siehe Kap.4.2[4.7]) möglich wird. Jetzt benötigen wir sie zum "*Ankoppeln*" der untersten (kausaldiskret arbeitenden) Schicht einer Operatorenhierarchie an die "darunterliegende" kausalkontinuierliche Schicht, d.h. an die physikalische Realität oder richtiger an die Welt der klassischen Physik.

Die Rückerinnerung an die Funktion des Schwellenoperators beim Ankoppeln diskreter an analoge Prozesse legt die Schlussfolgerung nahe, dass Schwellenoperatoren die elementaren Operatoren informationeller Systeme sind. Dieser Schluss

bedarf jedoch einer Korrektur. Die richtige Antwort auf die eingangs gestellte Frage lautet: *Die **Basisoperatoren** informationeller Systeme sind **Schwellenoperatoren***. 13
Als *Basisoperatoren* sind diejenigen Operatoren bezeichnet, die das Ankoppeln des kausaldiskreten Bereiches an den kausalkontinuierlichen bewerkstelligen. Sie gehören zu beiden Bereichen und bilden gewissermaßen das Niemandsland im Grenzbereich. Man könnte sie auch *Grenzoperatoren* nennen. Sie sind erforderlich, um die Grenze aus dem kontinuierlichen, nichtsprachlichen, physikalischen Bereich in den diskreten, sprachlichen, informationellen Bereich zu überschreiten. Es gilt also die fundamentale Aussage: *Ohne Schwellenoperatoren ist **keine Information** und **keine Informationsverarbeitung** möglich*, zumindest nicht auf dem Boden der klassischen Physik[7].

In Kap.9.2.2 werden wir sehen, dass Schwellenoperatoren nicht nur die *Basis*operatoren der Informationsverarbeitung sind, sondern auch als *elementare* Operatoren für die Komponierung von Kompositoperatoren verwendet werden können. Biologische informationelle Systeme demonstrieren diese Möglichkeit. Denn die Bausteine der grauen Materie des Gehirns, die *Neuronen*, sind offensichtlich Schwellenoperatoren. Dieser experimentelle Befund bestätigt die obige fundamentale Aussage.

8.2.5 Logisch-kausale Äquivalenz

In Kap.8.2.1 [5] waren wir zu der Schlussfolgerung gelangt, dass logischen Zusammenhängen kausale Zusammenhänge zugrunde liegen. Insofern besteht eine Äquivalenz zwischen Logik und Kausalität. Da dies eine *prinzipielle* Feststellung ist, sprechen wir vom **Prinzip der logisch-kausalen Äquivalenz**. Es ist die Folge des Umstandes, dass ein sprachlich (z.B. mathematisch) artikulierter "logischer" Zusammenhang erst dann Sinn erhält, wenn er von einem realen Interpretator interpretiert wird. Das Prinzip verlagert das Problem der Beziehung zwischen Logik und Kausalität in den Interpretator, in den Träger des betreffenden informationellen Prozesses. In Bild 8.6 sind die Beziehungen zwischen logischen und kausalen Zusammenhängen in einer Übersicht wiedergegeben.

Die Zusammenhänge werden auf den drei semantischen Ebenen beschrieben, auf denen sich sprachliches Modellieren vollzieht. Jeder Ebene entspricht ihre charakteristische Art von Semantik, externe bzw. formale bzw. interne Semantik. Auf der externen bzw. internen Schicht bezieht sich Semantik auf die trägerexterne bzw. trägerinterne Realität. Auf der formalen Ebene bezieht sich Semantik auf ein Kalkül (vgl. Bild 5.3). Wir betrachten zunächst die Beziehungen zwischen der externen und der formalen Schicht.

---

7 Für die "Quanteninformatik" gilt die Aussage nicht. Die Diskretisierung erfolgt dort durch die quantenmechanische Natur der Operatoren, der Bausteine des Quantencomputers (siehe z.B. [Briegel 99]). Er wird in diesem Buch nicht betrachtet. Überhaupt kommen quantenmechanische Phänomene nicht zur Sprache.

Semantische Ebene	Bedingung, Ursache	Grundlage des Zusammenhanges	Folge, Wirkung
extern	Zustand der Welt	Kausalität (im Original) →	Folgezustand
formal	Satz des Kalküls	Logik (Deduktion durch den Modellträger) →	Folgerung
intern	Zustand des Modellträgers	Kausalität (im Modellträger) →	Folgezustand

Bild 8.6 Kausale und logische Zusammenhänge auf den drei semantischen Ebenen

Bei der Modellierung eines Ausschnitts der Realität mittels einer formalen Theorie, also eines interpretierten Kalküls, stellt der Realitätsausschnitt eine Interpretation des Kalküls dar. Die Zuordnungen von Bezeichnern des Kalküls zu Objekten und Merkmalen der Realität trifft der modellierende Mensch. Dadurch werden kausale Zusammenhänge der Realität zu logischen Zusammenhängen der Theorie. In der Realität (auf der externen Ebene) wird durch einen realen Prozess ein realer Zustand z in einen *Folge*zustand z' als *kausale Folge* überführt. Auf der mittleren, formalen Ebene wird durch Deduktion ein Satz s des Kalküls in einen Satz s' als *logische Folge* (Schluss*folge*rung) überführt. Wenn z die Interpretation von s ist, muss z' die Interpretation von s' sein, falls die Theorie richtig ist.

Diesen Sachverhalt hat HEINRICH HERTZ im Vorwort zu den “Prinzipien der Mechanik [Hertz 1894] schöner ausgedrückt: “*Wir machen uns innere Scheinbilder oder Symbole der äußeren Welt, und zwar machen wir sie von solcher Art, dass die denknotwendigen Folgen der Bilder stets wieder die Bilder seien von den naturnotwendigen Folgen der abgebildeten Gegenstände*.”

Wenn ein Computer die Rolle des Interpretators spielt, sind die Interpretationsprozesse bekannt, sodass die Zurückführung logischer Zusammenhänge, die das System liefert (z.B. durch mathematische Berechnung) auf die zugrunde liegenden kausalen (physikalischen) Zusammenhänge im Computer offen vor Augen liegt. Wenn aber der Mensch die Rolle des Interpretators spielt, ist die Zurückführung nicht möglich, weil die Gehirnprozesse nicht ausreichend bekannt sind.

Das Prinzip der logisch-kausalen Äquivalenz bildet die Grundlage der Rechentechnik und der Herangehensweise dieses Buches an die Probleme der Informatik. Gäbe es keine logisch-kausale Äquivalenz, ließen sich keine logischen Zusammenhänge hardwaremäßig realisieren und hätte das Trägerprinzip [Einleitung.3] keinen Sinn.

Beim Implementieren eines extern interpretierten Kalküls (beim Schreiben entsprechender Computerprogramme) wird dieser ein zweites Mal interpretiert, jedoch nicht extern, nicht durch den zu modellierenden Ausschnitt der Realität, sondern *intern*, indem den Bezeichnern des Kalküls codierende Zustände des Computers zugeordnet werden. Die Zuordnung trifft der Programmierer. Dadurch werden logische Zusammenhänge des Kalküls zu kausalen Zusammenhängen im Computer. Durch die zweifache Interpretation des Kalküls werden verschiedenen Zuständen der externen Realität verschiedene Zustände der trägerinternen Realität zugeordnet und ***externe*** *kausale Zusammenhänge werden zu* ***internen*** *kausalen Zusammenhängen.*

Es stellt sich die Frage, ob die dargelegten Beziehungen zwischen formaler und interner Schicht vielleicht auch dann ihren Sinn behalten, wenn an die Stelle des Computers der Mensch tritt und der Kalkül intern durch Gehirnzustände "interpretiert" wird. Das würde bedeuten, dass Idemen codierende Zustände im Gehirn, also **interne Realeme** entsprechen. Ob das zutrifft, ist gegenwärtig nicht experimentell gesichert. Doch nicht wenige Forscher meinen, dass die Annahme codierender Gehirnzustände zutrifft. Zumindest stellt sie eine vernünftige Hypothese dar. Wenn sie sich als wahr herausstellt, ist in Bild 2.1 zwischen der Realem- und der Idemspalte eine Spalte "Internrealem" einzufügen. Aus einer Zuordnung Realem $\rightarrow$ Idem wird eine Zuordnung Externrealem $\rightarrow$ Internrealem $\rightarrow$ Idem.

Durch diese Erweiterung der "Abbildungen des sprachlichen Modellierens" (des Bildes 2.1) ergeben sich viele neue "objektive" Fragen, d.h. Fragen, die den Bereich der Ideme nicht berühren und eine formale Behandlung erlauben. Lässt man die Welt der Ideme außer Acht, öffnet sich ein Weg zu einer exakten Wissenschaft vom aktiven sprachlichen Modellieren, zu einer exakten, *subsymbolischen* Informatik und zu einer exakten Erkenntnistheorie. Physiologie und Psychologie würden zu einer einheitlichen wissenschaftlichen Disziplin verschmelzen. Trägerexterne und trägerinterne Phänomene würden eine Einheit bilden und gemeinsam (auf der mittleren Ebene) durch eine formale Theorie beschrieben werden. Komponenten *der Welt und der Vorstellung* würden ein gemeinsames System miteinander wechselwirkender Komponenten bilden. All diese Konjunktivsätze artikulieren Möglichkeiten, der aus der Sicht eines ganzheitlich (monistisch) denkenden Menschen eine Selbstverständlichkeiten sind. Am Horizont erscheint die Vision einer naturwissenschaftlichen Untermauerung vieler Bereiche der Geisteswissenschaften.

8.3 Operation, Funktion, Prädikat, Berechenbarkeit

Nach unserem Ausflug an die Grenze zwischen Natur- und Geisteswissenschaften kehren wir nochmals zu den Begriffen Operator, Operation und Funktion zurück. Sie wurden in einigen Fällen wie Synonyme verwendet, was möglicherweise Unverständnis hervorgerufen hat, denn es handelt sich um unterschiedliche Begriffe. Die Unterschiede werden sinnfällig, wenn man, ausgehend vom realen Operator, die Begriffe der Operation und der Abbildung bzw. der Funktion durch zwei Abstraktionsschritte einführt[8].

Gegeben sei ein realer (stofflich realisierter) Operator, z.B. eine Addiermaschine oder die elektronische Realisierung des in Bild 8.1 beschriebenen Operators. Für eine Folge von Eingabewerten liefert der Operator eine Zuordnungstafel, d.h. eine Liste von (x,y)-Paaren, er *berechnet* diese sogenannte **Wertetafel**. Jede Zeile der Tafel ordnet einem x bzw. einem x-Wert genau ein y bzw. einen y-Wert zu. Eine Wertetafel stellt also eine geordnete Menge eindeutiger Zuordnungen dar.

Erste Abstraktion. Wir abstrahieren von der konkreten Realisierung (z.B. von der elektronischen Schaltung) des Operators, nehmen aber an, dass ein realer Operator mit eben der Zuordnungstafel existiert oder gebaut werden kann. Auf diesem Abstraktionsniveau sagen wir, dass die Zuordnungstafel eine **Operation** definiert und nennen die Tafel **Operationstafel** eines *gedachten* Operators.

Man beachte, dass auch der Begriff des *sprachlichen Operators* von dem ausführenden (interpretierenden) realen Operator abstrahiert. In diesem Sinne sind die Bezeichnungen “sprachlicher Operator” und “Operation” Synonyme, doch werden sie nicht wie Synonyme verwendet. Wir werden die beiden Wörter vorwiegend in folgenden Bedeutungen benutzen: *Eine **Operation** ist eine durch ein Operationssymbol oder einen Operationscode benannte Zuordnungs**tafel**, ein sprachlicher Operator ist eine Zuordnungs**vorschrift***. Der Unterschied zwischen beiden Begriffen verschwindet, wenn man ein Operationssymbol als Vorschrift auffasst. Umgangssprachlich wird unter Operation oft der *Vorgang* des Zuordnens verstanden. Wir werden, wenn der Vorgang gemeint ist, der Eindeutigkeit halber von **Operationsausführung** oder von **Prozess** sprechen.

14

Zweite Abstraktion. Wir abstrahieren nun nicht nur vom realen Operator, sondern von seiner *Existenz* und lassen die Möglichkeit zu, dass *kein* Operator angebbar ist, der die Zuordnung trifft. Dann sprechen wir von **Abbildung** und **Abbildungstafel** oder von **Funktion** und **Funktionstafel**. Jede Zeile einer Funktionstafel ordnet einem *Argumentwert* genau einen *Funktionswert* zu. Als “Werte” sind auch “Wertetupel” zugelassen. (Der Leser hat vielleicht bemerkt, dass die Definition

8 Die Definitionen sind an die USB-Methode angepasst und nicht unbedingt identisch mit denjenigen, die in Mathematikbüchern angegeben werden. Aber auch in der Mathematik werden die Begriffe nicht immer ganz einheitlich verwendet.

des Abbildungs- und Funktionsbegriffs das Trägerprinzip verletzt. Darauf kommen wir sogleich zurück.)

Mit der Abstraktion von der stofflichen Realität des Operators wird gleichzeitig von der stofflichen Realität der Operanden abstrahiert. Die Operanden können Objekte des Denkens, also Ideme sein, und der Mensch kann sich in seinem Geiste beliebige Abbildungen zwischen gedachten Objektmengen und er kann sich beliebige Funktionen ausdenken. Der Leser wird ahnen, welche Schwierigkeiten durch diese Abstraktion heraufbeschworen werden. Es lassen sich nämlich Funktionen definieren, die weder vom Computer noch vom Menschen berechnet werden können, denn offensichtlich werden mit der Abstraktion vom Träger *nichtberechenbare* Funktionen zugelassen. Wenn eine Funktion durch Worte festgelegt wird, hinter denen kein Träger, kein Gerät, kein Rechner stehen muss, der die Worte in konkrete Wertepaare (Argumentwert, Funktionswert) umsetzen kann, besteht immer die Gefahr, dass die Worte etwas Unausführbares, etwas nicht Berechenbares, vielleicht sogar etwas Nichtexistierendes oder Unsinniges definieren. Wie wir aus Kap.6 wissen, erlaubt Sprache, Wahres wie Falsches, Sinnvolles wie Sinnloses zu artikulieren.

Dieser Gefahr wirkt das *Trägerprinzip* entgegen. Würden wir uns konsequent daran halten, wäre für den Funktionsbegriff kein Platz in unserem Begriffsgebäude und in unseren Gedankengängen und die Begriffe der Berechenbarkeit und Nichtberechenbarkeit von Funktionen verlören ihren Sinn. Dann würde aber mit dem Funktionsbegriff eine entscheidende Schnittstelle zur Mathematik verloren gehen und die Kommunikation zwischen Informatikern und Mathematikern wäre in Frage gestellt. Das Trägerprinzip darf also nicht zum Dogma erhoben werden, sondern ist eher als unverbindliche Charakterisierung unserer Herangehensweise an die Probleme der Informatik anzusehen.

An dieser Stelle sei auf einen weiteren Begriff hingewiesen, der eine Verletzung des Trägerprinzips bedeutet, auf den wir aber ebenfalls nicht verzichten dürfen, weil die Mathematiker, mit denen wir kommunikabel sein müssen, auf ihn nicht verzichten dürfen, auf den Begriff des Unendlichen. Nach dem Trägerprinzip und dem aus ihm resultierenden Realisierbarkeitsprinzip dürften wir eigentlich nur endliche Wertetafeln zulassen, also nur Funktionen mit endlichen Argument- und Funktionswertemengen. Dann würden wir aber in Konflikt mit dem mathematischen Funktionsbegriff kommen, z.B. mit dem Begriff der Additionsfunktion, denn die Additionsfunktion ist für unendlich viele Summanden definiert, die Wertetafel ist unendlich lang in dem Sinne, dass sie beliebig lange fortsetzbar ist.

Über die Vorstellung der Fortsetzbarkeit führen wir folgenden in der Mathematik üblichen Unendlichkeitsbegriff ein. *Eine Menge, die dadurch entsteht, dass eine endliche Menge unendlich oft um endlich viele Elemente erweitert wird, heißt* ***abzählbar unendliche Menge***. Die Bezeichnung kann in demselben Sinne auf Folgen angewendet werden, z.B. auf Ziffernfolgen, Ereignisfolgen (d.h. auf kausaldiskrete Prozesse) oder auf Wertetafeln. Es ist offensichtlich, dass die Menge (Folge) der natürlichen Zahlen eine abzählbar unendliche Menge (Folge) in dem soeben defi-

nierten Sinne ist, denn man kann "beliebig lange" zählen, vorausgesetzt, man lebt beliebig lange. Das Unendliche ist immer nur *gedanklich*, d.h. "abstrakt realisierbar". Falls eine Menge abzählbar unendlich sein kann, aber nicht sein muss, sprechen wir von **unbeschränkter Menge**[9].

Wir verletzen also das Realisierbarkeitsprinzip und lassen abzählbar unendliche Funktionstafeln zu. Nicht nur die Anzahl, sondern auch die Länge der Zeilen darf abzählbar unendlich sein. Damit haben wir uns dem mathematischen Funktionsbegriff angepasst und der Begriff der Berechenbarkeit bekommt konkreten Sinn und muss definiert werden. Um den Träger eines Berechnungsprozesses nicht aus dem Auge zu verlieren unterscheiden wir zwischen der *Berechnung* und der *Realisierung* einer Funktion und vereinbaren:

Eine Funktion heißt ***realisierbar*** *bzw.* ***realisiert****, wenn ein realer Operator gebaut werden kann bzw. existiert, der jedem Argumentwert den Funktionswert zuordnen kann. Eine Funktion heißt* ***berechenbar****, wenn ein sprachlicher Operator (eine Operationsvorschrift, ein Algorithmus, eine Formel) und ein realer Operator existiert oder angebbar ist, der durch Interpretierung des sprachlichen Operators (durch Ausführung der Vorschrift) zu jedem Argumentwert den Funktionswert bestimmen kann.*

Danach ist *Berechenbarkeit* ein Spezialfall von *Realisierbarkeit*. Der Begriff der Realisierbarkeit setzt genau das voraus, wovon der Funktionsbegriff abstrahiert: den realen Operator bzw. Interpretator. Wie bereits festgestellt, ist es aus diesem Grunde nicht verwunderlich, dass es nichtberechenbare Funktionen gibt, genauer gesagt Funktionen, die sich zwar definieren, aber nicht berechnen lassen.

Man kann die Frage aufwerfen, unter welchen Bedingungen eine Funktion definiert, aber nicht realisiert bzw. berechnet werden kann. Im Rahmen der Algorithmentheorie wird diese Frage gestellt und versucht, sie zu beantworten. Auch wir werden uns in Kap.8.4 mit dem Problem der Berechenbarkeit beschäftigen, aber lediglich mit dem Ziel, einige Gedankengänge, Bergriffsbildungen und Schlussfolgerungen der Theorie der Algorithmen und der Berechenbarkeit zu verstehen. Das wird uns helfen, die USB-Methode mit dem wichtigen Begriff der *rekursiven Funktion* in Beziehung zu setzen. Außerdem wird es uns helfen, die Problemstellungen solcher Teilgebiete der Informatik wie Programmierungstechnik und Künstliche Intelligenz, die stark von der Algorithmentheorie mitgeprägt worden sind, deutlicher zu erkennen und auch die Wege, auf denen bestimmte Problemlösungen gefunden worden sind.

Es sei noch einmal wiederholt: *Der Funktionsbegriff ohne Bindung an einen realen Operator ist ebenso wie der Berechenbarkeitsbegriff für den Bau und die Nutzung von Computern nicht erforderlich.* Im Sinne des Realisierbarkeitsprinzips

---

9 In der Mathematik wird der Begriff der unbeschränkten Menge auch in einer anderen, spezielleren Bedeutung verwendet.

werden wir, wenn von einer Funktion die Rede und nichts Gegenteiliges gesagt ist, stillschweigend voraussetzen, dass sie realisierbar ist, d.h. *wir* ***identifizieren*** *den Begriff der Funktion mit dem der Operation und damit auch mit dem des sprachlichen Operators*, denn wir hatten oben [4] festgestellt, dass die Bezeichnungen "Operation" und "sprachlicher Operator" de facto Synonyme sind. Unter Voraussetzung der Realisierbarkeit sind wir also berechtigt, *die Bezeichnungen* ***sprachlicher Operator***, ***Operation*** *und* ***Funktion*** *als* ***Synonyme*** *zu verwenden*. Im Sinne der Trägerprinzips fragen wir nicht, ob eine Funktion berechnet werden kann, sondern wir fragen, welche Input-Output-Zuordnungen ein bestimmtes (existierendes, oder eindeutig beschriebenes) informationelles System treffen und welche Wertetafeln (sprich Funktionen) das System realisieren bzw. berechnen kann. Das hat zur Folge, dass viele interessante und schwierige Fragen der Algorithmentheorie in diesem Buch nicht zur Sprache kommen. 16

Der Begriff der nichtberechenbaren Funktion wirft die Frage auf, wie sich eine Funktion definieren lässt, wenn sie nicht berechnet werden kann. Es wird sich zeigen, dass dies möglich ist, indem man sie durch ein *Prädikat* definiert (der Prädikatbegriff wird weiter unten eingeführt). Ganz allgemein gibt es folgende Möglichkeiten für die Festlegung von Funktionen:

1. Festlegung durch eine Wertetafel,
2. Festlegung durch einen realen Operator,
3. Festlegung durch einen sprachlichen Operator,
4. Festlegung durch ein Prädikat.

Es ist Aufgabe der *technischen* Informatik, Techniken zu entwickeln, die den Entwurf, die Realisierung und die Nutzung der vier genannten Möglichkeiten der Funktionsfestlegung unterstützen. Es bietet sich an, den zu behandelnden Stoff in diesem Sinne zu gliedern. Das spiegelt sich in der weiteren Kapitelfolge wider. Die folgenden Bemerkungen sollen den Leser darauf vorbereiten.

Zur Festlegung durch Wertetafeln. Jedes Schulkind kennt die Tafeln des Kleinen Einmaleins. Mathematische Tafelwerke sind Sammlungen von Funktionstafeln wie z.B. Logarithmentafeln, Tafeln der trigonometrischen Funktionen u.ä.m. Durch die Stellenzahl der Wertangaben ist die Genauigkeit festgelegt. Im allgemeinen Falle einer Abbildung kann die Wertetafel beliebige Wörter enthalten, z.B. die Namen und Adressen der Arbeitnehmer einer Firma. Das Faktenwissen von *Datenbanken* ist in der Regel in Form derartiger Tafeln strukturiert[16.5]. Auch ein Telefonbuch, das jedem Abonnenten eine Telefonnummer zuordnet, stellt eine Funktionstafel dar. Wenn ein Abonnent zwei Nummern hat, besteht der betreffende Funktionswert aus zwei Nummern.

Zur Festlegung durch reale Operatoren. In Kap.9.3 werden wir uns überlegen, wie man zu einer bestimmten Wertetafel einen realen Operator konstruieren kann, der die Zuordnungen realisiert. Es wird sich zeigen, dass dies immer möglich ist, vorausgesetzt die Wertetafel liegt vollständig vor, d.h. sie enthält sämtliche x,y-Paare.

Das mag zunächst überraschen. Denn wie lässt sich z.B. eine Funktion realisieren, deren Wertetafel zufällig, z.B. durch Würfeln oder mit Hilfe eines Zufallszahlengenerators erstellt worden ist. Der Prozess lässt sich nicht wiederholen. Doch nachdem das Würfeln beendet ist, liegt eine vollständige Wertetafel der erwürfelten Funktion vor.

Zur Festlegung durch sprachliche Operatoren. Diesem Problem ist das Kapitel 8.4 gewidmet. Wir werden nach Methoden für das Formulieren von Operationsvorschriften suchen und fragen, ob es eine *universelle* Methode (ein Rezept, einen Algorithmus) gibt, nach der sich Vorschriften zur Berechnung *aller* realisierbaren Funktionen formulieren lassen, d.h. aller Funktionen, die sich durch *reale* Operatoren berechnen lassen.

17 **Zur Festlegung durch Prädikate.** Auf diese Art der Festlegung soll schon an dieser Stelle etwas ausführlicher eingegangen werden. Um zu verstehen, worin sie besteht, muss zuerst der Prädikatbegriff erklärt werden. Wir werden uns nicht damit begnügen, ihn zu definieren, sondern wir werden ihn ausführlich erläutern. Viele der folgenden Überlegungen berühren tiefreichende theoretische Fragen und müssen nicht unbedingt angestellt werden, wenn es "nur" darum geht, den Computer nachzuerfinden. Trotzdem wird der Prädikatbegriff auch für uns eine wichtige Rolle spielen.

Wir beginnen mit einer vorläufigen Bestimmung des Prädikatbegriffs mit Hilfe
18 umgangssprachlicher Begriffe: *Ein Prädikat ist ein sprachlicher Ausdruck mit der Struktur eines Aussagesatzes*, also eines Satzes, der seinem Subjekt ein *Eigenschaftsmerkmal* zuweist; das Merkmal kann auch eine Tätigkeit sein. Im Unterschied zur Satzlehre stellt ein Prädikat nach obiger Definition nicht einen Satzteil dar, sondern hat die Bedeutung eines vollständigen Satzes.

Das Eigenschaftsmerkmal kann auch eine Beziehung zu anderen "Subjekten" darstellen, die dann aber nicht zum Satzsubjekt gehören müssen, sondern auch Objekt oder Teil des Satzprädikats sein können. Darin liegt eine gewisse Inkonsequenz der Bezeichnungsweise, die wir dadurch ausmerzen, dass wir das Wort *Subjekt* durch das Wort *Objekt* (nicht im grammatikalischen, sondern im umgangssprachlichen Sinne) ersetzen und definieren:

19 *Einen sprachlichen Ausdruck, der ein Merkmal eines Objekts oder eine Relation zwischen Objekten artikuliert, nennen wir* ***Prädikat***. (Die Ersetzung des Wortes *Beziehung* durch *Relation* stellet eine Annäherung an den Sprachgebrauch der Mathematik dar.) Die Definition verlangt nicht, dass es sich um ganz bestimmte (konkrete) Objekte handelt; sie können unbestimmt bleiben. Unbestimmte Objekte werden **Individuenvariable** oder einfach *Variable* genannt; eindeutig bestimmte ("konkrete") Objekte werden *Konstante* genannt. Zur Illustration folgen sieben Beispielprädikate P_1 bis P_7. Um zum Ausdruck zu bringen, dass eine Zeichenkette als Prädikat zu interpretieren ist, setzen wir sie in eckige Klammern.

P_1 [Max ist 5 Jahre alt]
P_2 [3 = 5]
P_3 [$z = 5$]
P_4 [*Nr* ist die Telefonnummer des Abonnenten *Ab*]
P_5 [*k* ist das jüngste Kind von *v* und *m*]
P_6 [$y < x$]
P_7 [$y = x$]

Die Beispiele verdeutlichen die Tragweite des Prädikatbegriffs. Der Leser wird sie ohne Kommentar richtig verstehen, trotz der unterschiedlichen Gegenstandsbereiche und Notationsweisen. Unbestimmte Objekte sind durch kursive Buchstaben bezeichnet, um sie als Variable auszuweisen.

Als Symbol für ein Prädikat wird oft der Buchstabe *P* verwendet. $P(x_1,x_2,...x_n)$ bezeichnet ein Prädikat mit *n* Variablen. Es wird n-stelliges Prädikat genannt (in Analogie zur n-stelligen Funktion). Beispielsweise ist P_3 einstellig, P_4 ist 2-stellig und P_5 ist 3-stellig. Prädikate, die gar keine Variablen enthalten (0-stellige Prädikate), wie z.B. P_1 und P_2, heißen **Aussagen** (im Sinne der Prädikatenlogik). Man beachte, dass P_2, P_3, P_6 und P_7 rein formale Semantik besitzen, während sich P_1, P_4 und P_5 externsemantisch interpretieren lassen.

Mancher Leser wird vielleicht über das zweite Prädikat verwundert sein, weil es eine falsche Aussage ist. Formal ist P_2 jedoch richtig, denn es entspricht der Definition des Prädikats. Diese verlangt nämlich *nicht*, dass die artikulierte Eigenschaft bzw. Relation zutreffen muss, die Relation wird nicht konstatiert, sondern lediglich artikuliert. Wenn sie zutrifft, sagt man, dass das Prädikat **erfüllt** ist, andernfalls, dass es **nicht erfüllt** ist. Oft wird auch gesagt, dass ein Prädikat *wahr* bzw. *falsch* ist. Auch wir werden diese Redeweise verwenden, obwohl sie nicht ganz korrekt ist. Um korrekt zu sein, müsste man sagen, dass ein Prädikat für bestimmte Werte der Variablen zu einer *wahren* bzw. *falschen Aussage* wird. Eine analoge Unkorrektheit lässt man sich zu Schulden kommen, wenn man sagt: “Die Funktion $f(x)$ nimmt den Wert 1 an”, ohne anzugeben für welche *x*-Werte. Ein Prädikat heißt **entscheidbar**, wenn für beliebige Variablenwerte entschieden werden kann, ob es erfüllt ist oder nicht.

Es ist wichtig, die Bedeutung der Gleichheitszeichen in einem Prädikat richtig zu verstehen. Wie bereits festgestellt wurde, konstatiert P_2 nicht die Gleichheit von 3 und 5. Das Entsprechende gilt für P_1 und P_3. P_3 ist *nicht* so zu verstehen, dass der Variablen *z* der Wert 5 *zugewiesen* wird, dass gewissermaßen *z* zur Konstanten 5 “gemacht” wird, genauso wie P_1 nicht aus Max einen Fünfjährigen “macht”. Das Gleichheitszeichen in P_2 und P_3 und überhaupt in jedem Prädikat artikuliert also keine *Wertzuweisung* sondern eine Beziehung, eine *Relation*, die Relation der Gleichheit. Darum sprechen wir von **relationaler Gleichung**. Wenn dagegen ein Ausdruck mit einem Gleichheitszeichen eine Wertzuweisung artikuliert, sprechen wir von **Ergibtgleichung**. Der Eindeutigkeit halber ist es zweckmäßig, das Gleichheitszeichen in einer Ergibtgleichung durch das **Ergibtzeichen** “:=” zu ersetzen.

In der mathematischen Literatur wird der Gleichungsbegriff in der Regel ausschließlich in relationalem Sinne definiert. Dann stellt die Bezeichnung “Ergibtgleichung” einen Widerspruch und die Bezeichnung “relationale Gleichung” einen Pleonasmus dar. Abweichend vom mathematischen Sprachgebrauch vereinbaren wir (in Konzession an die weit verbreitete doppeldeutige Verwendung des Wortes “Gleichung”): *Eine* ***relationale Gleichung*** *ist ein mathematischer Ausdruck, der das* ***Relationszeichen*** *“=” enthält. Aus einer relationalen Gleichung wird eine* ***Ergibtgleichung****, wenn sie nach einer Variablen aufgelöst und das Gleichheitszeichen durch das Ergibtzeichen ersetzt wird, wodurch zum Ausdruck gebracht wird, dass der Ausdruck als Vorschrift zur Berechnung des Wertes der eliminierten Variablen zu interpretieren ist.* Eine computerverständliche Ergibtgleichung heißt **Ergibtanweisung**. Es sei noch einmal wiederholt, dass das Gleichheitszeichen in einem Prädikat keine Gleichheit konstatiert oder fordert, sondern eine (unverbindliche) Aussage ist, die wahr oder falsch sein kann.

Wie man sieht, verbergen sich hinter dem Gleichheitszeichen logische Schwierigkeiten, die man vielleicht nicht erwartet hatte. Etwas einfacher liegen die Dinge im Falle von Ungleichungen. Denn es versteht sich von selbst, dass ein Symbol, das keine Gleichheit artikuliert (z.B. das Symbole “>”), kein Ergibtzeichen, sondern nur ein Relationszeichen sein kann und dass eine Ungleichung niemals eine Wertzuweisung ist, sondern stets eine Beziehung artikuliert.

Wir machen nun einen großen Sprung zurück zu Kapitel 7.2 und erinnern uns an den ABC-Schützen [7.12], der die Differenz 5-2 dadurch bestimmt, dass er 2 sooft inkrementiert (um 1 erhöht), bis er 5 erreicht. Er zählt die Inkrementierungsschritte und entscheidet nach jeder Inkrementierung das Prädikat [*z*=5], worin *z* das momentane inkrementierte Ergebnis ist. Er beendet das Inkrementieren, sobald das Prädikat erfüllt ist. Die Anzahl der Inkrementierungen ist die gesuchte Differenz.

Ganz ähnlich verfährt der Steueroperator in Bild 8.1b. Während der Berechnung der Potenz x^n durch iteratives Multiplizieren zählt er die Iterationsschritte und entscheidet in jedem Schritt das Prädikat [*it*<n]. Darin ist *it* das momentane Zählergebnis, die sog. Iterationszahl, und n ist die vorgegebene Potenz. Sobald das Prädikat falsch wird, generiert der Steueroperator ein Steuersignal, das die Zweigeweiche in der Ausgabeleitung des Multiplizierers umstellt, sodass sie nicht mehr auf den Eingang des Multiplizierers, sondern auf den des Addierers zeigt.

Der ABC-Schütze wie der Potenzierer berechnen beide den Wert einer 2-stelligen Funktion, einmal der Differenzfunktion Minuend minus Subtrahend, das andere mal der Potenzfunktion Basis hoch Exponent. In beiden Fällen ist das Entscheiden eines Prädikats in den Berechnungsprozess involviert und damit auch in die Berechnungsvorschrift (den sprachlichen Operator), der die Funktion festlegt. Wir haben es also mit zwei Beispielen dafür zu tun, dass eine Funktion unter Einbeziehung eines Prädikats festgelegt wird, wobei das Prädikat die Rolle eines Endekriteriums spielt. Ein solches Prädikat nennen wir **Ende-** oder **Abbruchprädikat**.

Die Frage liegt nahe, ob die Rolle von Prädikaten darauf beschränkt ist, ein mehr oder weniger wichtiger *Bestandteil* einer Berechnungsvorschrift zu sein, wie in obigen Beispielen, oder ob ein Prädikat selber die Berechnungsvorschrift darstellen kann, m.a.W. ob ein Prädikat allein eine Funktion festlegen kann. Betrachtet man unter diesem Gesichtspunkt das Prädikat P7, erkennt man, dass diejenigen (x,y)-Wertepaare, für die das Prädikat erfüllt ist, die Wertetafel der Funktion $y = f(x) = x$ bilden. Das Prädikat P7 legt also diese Funktion fest, wenn es als erfüllt vorausgesetzt wird. Um einem Argumentwert seinen Funktionswert zuzuordnen, muss die Ergibtanweisung $y{:=}x$ ausgeführt werden. (Am Rande sei angemerkt, dass der Prädikatbegriff in der Mathematik oft als “*Wahrheitsfunktion*” *definiert* wird.)

Bringt man in der Gleichung $y{=}x$ beide Variable auf die linke Seite, erhält man $x\text{-}y = 0$. Auf der linken Seite dieser Gleichung steht nun eine Funktion $f(x,y)$, die für alle Argumentwertepaare, für die P7 erfüllt ist, den Wert 0 besitzt. Folglich kann man das Prädikat P7 dadurch *entscheiden*, dass man die Nullstellen der Funktion $f(x,y)$ sucht. Das *Entscheiden* eines Prädikats wird in das *Berechnen* einer Funktion überführt.

Das Vorgehen scheint in diesem einfachen Fall nicht viel Sinn zu haben, doch lässt es sich verallgemeinern. Jedes Prädikat $P(x_1, x_2,...)$, das ein Relationszeichen enthält, lässt sich in die Form $f(x_1, x_2,...)$ # 0 überführen, worin # jedes Relationszeichen sein kann. Die Funktion auf der linken Seite heißt **charakteristische Funktion**. Ihr Wert zeigt an, ob das Prädikat erfüllt ist oder nicht. Folglich gilt: *Ein Prädikat ist entscheidbar, wenn seine charakteristische Funktion berechenbar ist.*

Anhand des Prädikats P7 wurde gezeigt, wie durch ein zweistelliges Prädikat eine Funktion festgelegt werden kann. Es soll nun gezeigt werden, dass auch durch ein einstelliges Prädikat eine Funktion festgelegt werden kann. Gegeben sei ein Prädikat $P(x)$. Dann kann man eine Funktion $f(x)$ dadurch festlegen, dass man ihr für alle x, für die P nicht erfüllt ist, den Wert 0 zuweist und für alle x, für die P erfüllt ist, den Wert 1. Auf diese Weise legt $P(x)$ eine Funktion $f(x)$ fest, die zwei Funktionswerte annehmen kann, also eine *binäre* Funktion ist.

Anhand eines Beispiels soll demonstriert werden, dass man auf diese Weise nichtberechenbare Funktionen definieren kann. Wir legen eine binäre Funktion $f(x)$ fest, die den Wert 1 annimmt, wenn das Prädikat

[in der Irrationalzahl x tritt die Ziffernfolge 0 bis 9 auf]

erfüllt ist. Beispielweise gilt $f(\sqrt{2}) = 1$, wenn bei der stellenweisen Berechnung der Wurzel aus 2 nach einem iterativen Algorithmus *irgendwann einmal* die Ziffern von 0 bis 9 in geordneter Reihenfolge auftreten. Diese Frage kann niemals mit “*nein*” beantwortet werden, denn “man kann nie wissen, was noch kommt”. Man weiß nicht, ob die Ausführung des Algorithmus tatsächlich ein Ende findet, ob der Algorithmus *terminiert*. Diese Unsicherheit besteht immer dann, wenn gefragt wird, ob ein Ereignis, das nicht mit Sicherheit eintritt, tatsächlich eintritt. Die Frage kann nie verneint werden. Erst nachdem das Ereignis eingetreten ist, kann sie bejaht werden.

21 Prädikate, die erst dann erfüllt sind ("wahr" sind, "Wahres sagen"), sobald ein *nichtvorhersehbares* Ereignis eingetreten ist, vorher aber nicht erfüllt sind, nennen wir **Wahrsageprädikate** (eine unübliche, aber sinnfällige Bezeichnung).

Die vorangehenden Überlegungen führen auf das allgemeine Problem der *Terminierung* von Programmen, also auf die Frage, ob die Abarbeitung eines Programms früher oder später ihr Ende findet. Wir werden uns dem Problem nähern, indem wir die Frage untersuchen, welche Rolle Prädikate in der USB-Methode spielen und welche Bedingungen sie erfüllen müssen. Zur Beantwortung der Fragen führen wir den Begriff des Steuerprädikats ein: *Ein Prädikat, dessen Wahrheitswert eine Weiche oder ein Tor steuert, heißt* ***Steuerprädikat***. Das Steuerprädikat der Weiche vor dem Sinusoperator in dem Operatorennetz von Bild 8.1a lautet beispielsweise [$x \leq 0$].

Damit ist der erste Teil obiger Frage beantwortet: Prädikate spielen im Rahmen der USB-Methode die Rolle von Steuerprädikaten. Der zweite Teil der Frage lautet damit: Welche Bedingungen müssen Steuerprädikate erfüllen? Zunächst ist festzustellen, dass Steuerprädikate vom Steueroperator entschieden werden, der die Steuersignale generiert. Wenn ein Kompositoperator eine bestimmte Operation ausführen soll, müssen alle Prädikate, die während der Operationsausführung zu entscheiden sind, entscheidbar sein. Um zu sichern, dass die Operationsausführung terminiert (ihr Ende erreicht), muss zusätzlich gefordert werden, dass Abbruchprädikate irgendwann einmal denjenigen Wahrheitswert annehmen, bei dem abgebrochen werden soll; m.a.W. ein Abbruchkriterium muss früher oder später erfüllt sein.

Wir fassen die Ergebnisse zusammen: *Im Rahmen der* ***USB-Methode*** *spielen Prädikate die Rolle von* ***Steuerprädikaten****, wozu auch die* ***Abbruchprädikate*** *gehören. Steuerprädikate müssen entscheidbar sein, und nach Beginn einer Operationsausführung muss jedes Abbruchprädikat in endlicher Zeit das Abbruchkriterium erfüllen*.

Als Steueroperator kann der Mensch fungieren. Andernfalls müssen technische Steueroperatoren komponiert werden. Unter ihren Bausteinoperatoren müssen solche sein, die Prädikate entscheiden. Wir nennen sie **Prädikatoperatoren**. Im Rahmen der USB-Methode werden Prädikatoperatoren und Steueroperatoren aus den gleichen elementaren Operatoren komponiert wie die Arbeitsoperatoren.

Aus der Rolle, die Prädikate in der USB-Methode spielen, ergibt sich folgender Tatbestand. *Der Grund dafür, dass ein Operatorennetz für einen Eingabewert des zugelassenen Typs* (beispielsweise für eine reelle Zahl oder ein Tupel reeller Zahlen) *keinen Ausgabewert liefert, kann nur in einer Rückkopplungsschleife liegen, deren Abbruchkriterium fehlt oder nie erfüllt wird*, vorausgesetzt, dass alle Steuerprädikate entscheidbar sind und dass keine Fehler der Struktur und Funktion des Netzes vorliegen, Deadlocks eingeschlossen. Beispielsweise hält die Berechnung des Quotienten 1:3 als Dezimalzahl niemals an, wenn die Vorschrift keine Angabe darüber enthält, auf wie viele Stellen genau der Quotient berechnet werden soll.

Das Gleiche gilt für die Berechnung jeder Irrationalzahl. Das Problem kann nicht dadurch aus der Welt geschafft werden, dass Irrationalzahlen deswegen verboten

werden, weil sie dem Realisierbarkeitsprinzip widersprechen. Ihre Benutzung in sprachlichen Modellen ist notwendig, wenn sie in der Wirklichkeit existieren, m.a.W. wenn sie durch die Wirklichkeit *definiert* werden. Das gilt z.B. für das Verhältnis der Länge des Umfanges zur Länge des Durchmessers eines Kreises (π) oder für das Verhältnis der Länge einer Diagonalen zur Länge einer Seite eines Quadrates ($\sqrt{2}$). Die Zahlenwerte dieser real existierenden Verhältnisse sind nicht exakt, d.h. nicht absolut genau berechenbar. Sie sind aber auch nicht absolut genau *messbar* (vgl. Kap.4.1 [4.3]). Damit muss man sich abfinden. *Nicht jede durch ein Prädikat definierbare Zahl ist exakt berechenbar*. Freilich kann die Ungenauigkeit beliebig herabgesetzt werden, wenn der Berechnungsprozess gedanklich nicht beschränkt wird, sondern als abzählbar unendliche Folge von Ereignissen angenommen wird.

Eine Berechnungsvorschrift, deren Abarbeitung nicht endet, die nicht terminiert, stellt keinen Algorithmus dar, denn ein Algorithmus terminiert definitionsgemäß. Erst durch Hinzufügen eines Abbruchkriteriums wird eine nichtterminierende Vorschrift zu einem Algorithmus. Im Sinne des Realisierbarkeitsprinzips verlangen wir, dass eine Berechnungsvorschrift stets ein Abbruch- oder Endekriterium explizit oder implizit enthält und sei es in Form einer Begrenzung der Rechenzeit.

In der Praxis ist der häufigste Grund dafür, dass ein Programm nicht terminiert, ein versehentlich falsch formuliertes Abbruchkriterium. Um das Programm “zum Laufen zu bringen”, muss der Fehler gefunden und beseitigt werden. Das kann recht zeitaufwendig sein. Es wäre wünschenswert *vor* dem Start eines Programms zu wissen, ob es terminiert oder nicht. Leider gibt es keinen Algorithmus (kein Programm), nach dem man für jedes Programm feststellen kann, ob es für bestimmte Eingabedaten terminiert. Dieser Tatbestand wird als **Halteproblem** bezeichnet. 22 Damit kommen wir zu einem zweiten Beispiel einer binären Funktion, die durch ein Prädikat festgelegt ist. Es handelt sich um die sog. *Haltefunktion*.

Nach bewährtem Vorbild kann man mit Hilfe des Prädikats

[das Programm *p* terminiert für den Eingabeoperanden *x*]

eine binäre Funktion $h(p,x)$ definieren, wobei x ein Tupel sein kann. Die Funktion h nehme den Wert 1 an, wenn p für x terminiert, andernfalls den Wert 0. Die so festgelegte Funktion heißt **Haltefunktion**. Sie ist nicht berechenbar, d.h. nicht für jedes (p,x) ist der Wert von h berechenbar. Den exakten Beweis findet der Leser in fast jedem Lehrbuch über theoretische Informatik[10]. Wir führen zwei Argumente für die Unberechenbarkeit der Haltefunktion an.

Das Programm, das die Haltefunktion für ein Paar (p, x) berechnen soll - wir nennen es *Analyseprogramm* - muss feststellen, ob während der Ausführung von p jedes der in ihm enthaltenen Abbruchprädikate irgendwann (in endlicher Zeit) das Abbruchkriterium erfüllt. Das ist in vielen Fällen möglich, selbst wenn das analy-

10 Genannte seien [Stetter 88], [Sander 92], [Schöning 95].

sierte Programm *nicht* terminiert. Wenn beispielsweise ein Programm durch mehrmaliges Multiplizieren die fünfte Potenz von 3 berechnen soll, könnte das Abbruchprädikat lauten: Breche ab, sobald das Prädikat [z=5] erfüllt ist. Wenn das Potenzierprogramm nach jeder Multiplikation z inkrementiert, beginnend mit z=0, terminiert das Programm, und zwar nach 5 Multiplikationen (die erste Multiplikation ist $1*3$). Wenn das Programm das Inkrementieren - aus welchen Gründen auch immer - mit 6 beginnt oder wenn es mit 0 beginnt, aber nicht inkrementiert, sondern dekrementiert, terminiert das Programm nicht. Ob das Programm terminiert oder nicht, lässt sich feststellen, ohne es ausführen zu lassen.

In derartigen Fällen ist es also durchaus möglich, ein Analyseprogramm zu schreiben, das erkennt, ob das Potenzierprogramm terminiert oder nicht, das also den Wert der Haltefunktion berechnet. Es ist aber *nicht* möglich, ein *universelles* Analyseprogramm zu schreiben, und zwar aus folgendem Grunde.

Ein universelles Analyseprogramm muss für jedes zu analysierende Programm terminieren. Um sich zu überzeugen, dass ein Analyseprogramm diese Forderung erfüllt, benötigt man ein "übergeordnetes" Analyseprogramm, das seinerseits terminieren muss und so fort. Es könnte eingewendet werden, dass ein universelles Analyseprogramm, wenn es denn existiert, sich auch selber analysieren kann, sodass die Notwendigkeit einer unendlichen Kette von Analyseprogrammen entfällt und durch einen Zyklus ersetzt wird. Es liegt eine Operand-Operator-Zirkularität vor, und zwar eine Zirkularität ohne Ausweg, ohne Ende (vgl.Kap.6.3; man erinnere sich an die Schlange, die sich vom Schwanz her auffrisst). Das bedeutet, dass das Analyseprogramm nicht terminiert, also nicht universell ist und dass folglich die Haltefunktion nicht berechenbar ist. Diese Schlussfolgerung stellt zwar keinen mathematischen Beweis, aber doch ein plausibles Argument für die Nichtberechenbarkeit der Haltefunktion dar.

Man kann ein anderes Argument für die Nichtberechenbarkeit der Haltefunktion anführen. Es ist durchaus möglich, dass ein Programm als Abbruchprädikat ein Wahrsageprädikat enthält. Es macht beispielsweise keine Schwierigkeiten, ein Programm p zur Beantwortung der Frage zu schreiben, ob der als Dezimalzahl dargestellte Wert der Wurzel aus 2 die Zahlenfolge von 0 bis 9 enthält. Kein Analyseprogramm kann vorhersagen, ob das Programm terminiert. Ob das der Fall ist, weiß man erst, nachdem es terminiert hat, nachdem es die gesuchte Ziffernfolge gefunden hat.

In der Mathematik spielt eine andere Art von Nichtberechenbarkeit eine wichtige Rolle, die ihren Grund nicht darin hat, dass eine Operationsvorschrift nicht terminiert, sondern darin, dass keine Vorschrift angebbar ist. Das ist z.B. der Fall, wenn eine Funktion durch eine Relationsgleichung festgelegt wird, die nicht lösbar ist, also nicht nach der Funktionsvariablen aufgelöst werden kann. Dann ist keine Ergibtgleichung angebbar, nach der sich die Funktionswerte aus den Argumentwerten errechnen ließen. Ein Beispiel sind algebraische Gleichungen höherer als vierter Ordnung. Für sie existiert keine analytische Lösung in Form einer Ergibtgleichung (keine geschlos-

sene Lösung in Radikalen). Die Funktionswerte lassen sich allerdings numerisch durch eine Näherungsrechnung bestimmen.

Wir werfen nun noch einmal einen Blick zurück auf die vier Methoden für das Festlegen von Funktionen und fragen, nach welchen Methoden nichtberechenbare Funktionen definiert werden können. Die Antwort lautet: *Die Funktionsfestlegung durch Prädikate oder mit Hilfe von Prädikaten ist die einzige Methode, nach der nichtberechenbare Funktionen definierbar sind*. Die übrigen drei oben aufgezählten Methoden setzen die Berechenbarkeit bzw. Realisierbarkeit der Funktion voraus.

Wir beenden unsere Überlegungen mit folgender Bemerkung. Der Begriff der Berechenbarkeit ist als eine Eigenschaft mit zwei Werten definiert worden: berechenbar oder nichtberechenbar. Das Merkmal "*schwer* berechenbar" oder "*mit hohem Aufwand* berechenbar" wurde nicht eingeführt. Aber gerade der erforderliche Aufwand ist in der Praxis oft von Interesse. Der Aufwand kann eine Funktion *praktisch unberechenbar* machen. Das Aufwandsproblem ist nicht Gegenstand der *klassischen* Algorithmentheorie. Doch ist aus ihr ein spezielles Gebiet der Mathematik hervorgegangen, die **Komplexitätstheorie**. Sie ist dem Aufwandsproblem gewidmet und kann als moderner Zweig der Algorithmentheorie aufgefasst werden. Ihre Behandlung hätte sich im Rahmen des Kapitels 8 angeboten, doch verschieben wir sie auf das Kapitel 21, wo wir uns mit dem Begriff der Komplexität aus sehr allgemeiner Sicht auseinandersetzen werden.

8.4* Universelle algorithmische Systeme

8.4.1 Problemstellung

Wir haben gesehen, dass mittels Prädikaten nichtberechenbare Funktionen festgelegt werden können, für deren Berechnung also weder eine vom Menschen interpretierbare, terminierende Vorschrift noch ein terminierendes Computerprogramm existiert. Man könnte nun vermuten, dass es eine allgemeine Methode geben müsste, nach der sich für jede realisierbare Funktion ein Berechnungsalgorithmus formulieren lässt. Eine solche universelle Methode nennen wir **universelles algorithmisches System**.

Da Algorithmen sprachliche Gebilde, in unserem Begriffssystem sprachliche Operatoren sind, bedarf es einer Sprache, um sie zu artikulieren, und da die Sprache eindeutig interpretierbar sein soll, muss es eine *formale* Sprache sein. Eine formale Sprache für die Artikulierung von Algorithmen heißt **algorithmische Sprache**. Um in ihr Algorithmen artikulieren zu können, muss sie interpretiert (im Sinne der Mathematik), also eine *Kalkülsprache* sein (vgl.Kap.5.4 [5.2]). Sie heißt *universell*, wenn in ihr sämtliche berechenbaren Funktionen artikuliert werden können. Demnach ist ein universelles algorithmisches System (eine universelle Methode zur Artikulierung von Algorithmen) nichts anderes als eine *universelle* algorithmische

Sprache. Wenn es eine solche Sprache gibt, muss sich jeder, in irgendeiner anderen Sprache artikulierte Algorithmus in diese universelle Sprache übersetzen lassen.

Viele Methoden zur Artikulierung von Algorithmen sind vorgeschlagen worden. Es kann nicht das Ziel der folgenden Ausführungen sein, einen vollständigen Überblick über die diesbezüglichen Ergebnisse der Algorithmentheorie zu geben. Unser Ziel ist es vielmehr, die Schlussfolgerungen, die sich aus der USB-Methode hinsichtlich der Berechenbarkeit von Funktionen ziehen lassen, mit den Schlussfolgerungen der Algorithmentheorie in Beziehung zu setzen. Darüberhinaus soll der Leser mit einigen Ideen und Begriffen bekannt gemacht werden, die sich stimulierend auf die Entwicklung der Rechentechnik ausgewirkt haben und die bei der Entwicklung von Computerhardware und von Programmiersprachen Pate gestanden haben.

Daneben soll die Diskussion ein Schlaglicht auf eine Art geistiger Produktivität werfen, die scheinbar unsinnig ist. Sie soll demonstrieren, zu welchen Meisterleistungen der suchende Intellekt durch unerreichbare Ziele angespornt werden kann. Ein ähnliches Beispiel haben wir schon kennen gelernt, den gödelschen Unvollständigkeitssatz als Ergebnis der Suche nach einem allgemeinen Entscheidungskriterium hinsichtlich der Wahrheit bzw. Falschheit von Aussagen formalisierter Theorien, das es nicht gibt.

Die Suche nach einem algorithmischen System, in welchem jede "irgendwie" berechenbare Funktion artikuliert werden kann, verfolgt ein unerreichbares Ziel. Erst mit der Zeit wurde klar, dass ein solches System nicht angebbar ist, jedenfalls nicht auf der Basis unseres gegenwärtigen Erkenntnisstandes. Denn zu den realen Operatoren, die Funktionen berechnen können, gehören nicht nur die Computer, sondern auch die Menschen. Was Menschen können oder nicht können, ist unbekannt, genauer gesagt, die Frage lässt sich - ebenso wie die nach der Willensfreiheit - introspektiv nicht beantworen. Um sie beantworten zu können, müssten die Berechnungsmechanismen des Gehirns vollständig bekannt sein, was nicht der Fall ist. 23

Es kann also nicht bewiesen und infolgedessen auch nicht unbedingt erwartet werden, dass sich für jede vom Menschen realisierbare Funktion ein Berechnungsalgorithmus angeben lässt. Dies ist der Grund dafür, dass die Quintessenz der (klassischen) Algorithmentheorie kein Theorem, sondern eine Hypothese ist, die *These von Church* oder **churchsche These**. Sie wird in Kap.8.5 behandelt. Doch nehmen wir ihren Inhalt schon jetzt voraus. Sie wurde in vielen Varianten formuliert. Wir wählen folgende: *Jede* ***effektiv berechenbare*** *Funktion ist eine* ***rekursive*** *Funktion*.

Der Begriff der *effektiv berechenbaren* Funktionen ist ein *intuitiver* Begriff. Intuitiv wird eine Funktion als berechenbar bezeichnet, wenn *irgendwer irgendwie* ihre Funktionswerte bestimmt hat oder bestimmen kann. Ist das der Fall, spricht man von *effektiver Berechenbarkeit*. In unserer Sprechweise ist eine effektiv berechenbare Funktion eine *realisierbare* Funktion.

Rekursive Funktionen, sind solche, die in einem bestimmten algorithmischen System, das in Kap.8.4.6 eingeführt wird, definierbar sind. In den folgenden Kapiteln

werden einige algorithmische Systeme in unterschiedlicher Ausführlichkeit dargelegt und zwar:

1. Turingmaschine oder Turingautomat
2. Unbeschränkte Registermaschine (URM)
3. Markovalgorithmus
4. Lambda-Kalkül von CHURCH
5. Rekursive Definition von Funktionen
6. Methode der uniformen Systembeschreibung (USB-Methode).

Ausführlichere Darstellungen der Methoden 1 bis 5 findet man in der Literatur[11]. Die Methoden 1, 4 und 5 sind fast gleichzeitig um das Jahr 1936 vorgeschlagen worden. Die Funktionen, die in einem bestimmten algorithmischen System beschrieben und berechnet werden können, fassen wir zu einer Klasse zusammen, sodass 6 Klassen unterschieden werden können, die wir folgendermaßen bezeichnen:

- Klasse der Turing-Funktionen
- Klasse der URM-Funktionen
- Klasse der Markov-Funktionen
- Klasse der Church-Funktionen
- Klasse der rekursiven Funktionen
- Klasse der USB-Funktionen.

In der Algorithmentheorie sind Bezeichnungen üblich, die das Wort *berechenbar* enthalten. So wird z.B. nicht von Turing-Funktionen, sondern von Turing-berechenbaren Funktionen gesprochen.

In den folgenden Ausführungen darf der Leser ***keine Einführung in die Algorithmentheorie*** *oder in die* ***Theorie der Berechenbarkeit*** *sehen, sondern lediglich eine Erläuterung derjenigen Begriffe und Schlussfolgerungen der Algorithmentheorie, die erforderlich sind, um die gestellten Ziele zu erreichen. Hauptziel ist der Nachweis, dass USB-Funktionen rekursive Funktionen und rekursive Funktionen USB-Funktionen sind.*

8.4.2 Turingmaschine

Wenn die Frage, welche Funktionen berechenbar sind, auch nicht allgemein entschieden werden kann, so sollte es doch möglich sein anzugeben, welche Funktionen ein Computer bekannter Konstruktion berechnen kann. Das wirft die Frage auf, ob es ein allgemeines *Konstruktionsprinzip* gibt, nach dem alle Computer aufgebaut sind, und ob es elementare Operationen gibt, aus denen sämtliche Computeroperationen komponiert werden. Dieser Frage werden wir in Kap.9 nachgehen.

---

11 Siehe z.B. [Hermes 71], [Schnorr 74], [Cutland 80], [Stetter 88], [Penrose 91], [Sander 94], [Schöning 95], [Werner 95]. Die USB-Methode ist in [Jungclaussen 80-90] entwickelt worden, zunächst als Sprache zur Beschreibung kausaldiskreter Systeme, nicht als algorithmisches System. Zusammenfassende Darstellungen der USB-Methode sind in [Jungclaussen 85] und [Jungclaussen 90] veröffentlicht.

Von einer ähnlichen Frage ging ALLEN TURING aus. Er suchte nach dem denkbar *einfachsten* realen, universellen Zeichenkettenoperator (informationellen Operator) und erfand eine Maschine, die seitdem **Turingmaschine** oder **Turingautomat** genannt wird.

In gewisser Hinsicht ähnelt die Turingmaschine einem Magnetbandgerät oder Kassettenrecorder. Sie verfügt über ein unbeschränktes Speicherband, das als Träger einer Kette beliebig vieler Speicherzellen dient. Jede Zelle kann ein Zeichen des *Automatenalphabets* aufnehmen. Ein Tastkopf dient dem Lesen und Beschreiben des Bandes. Er kann durch die Steuereinheit bewegt werden. Das Band dient gleichzeitig als Speicher und als Ein- und Ausgabegerät.

Die Turingmaschine arbeitet taktweise. In jedem Takt liest die Steuereinheit das unter dem Kopf befindliche Zeichen, den *Input*, überführt es in ein neues Zeichen, den *Output*, und überschreibt auf dem Band das alte Zeichen durch das neue. Die Steuereinheit besitzt eine endliche Anzahl von Zuständen, darunter einen Endzustand. In jedem Schritt geht sie aus dem jeweils "alten" in den "neuen" Zustand über und kann gleichzeitig den Kopf bewegen.

Welche konkrete Aktion in einem Schritt ausgeführt wird, hängt vom momentanen Zustand und vom Input ab. Die Analogie zum abstrakten Automaten ist offensichtlich und die Bezeichnung Turing*automat* gerechtfertigt. Ebenso wie der abstrakte Automat verfügt der Turingautomat über eine *Automatentafel*. Diese ordnet einem (nicht unbedingt jedem) Paar (Input, alter Zustand) ein Tripel (Output, neuer Zustand, Verschiebung) zu. Die Verschiebung bezieht sich auf den Tastkopf, der um eine Zelle nach rechts oder nach links verschoben werden kann; die Verschiebung kann auch unterbleiben.

Um die Turingmaschine eine Zeichenkette verarbeiten (transformieren) zu lassen, wird das Band mit der Zeichenkette, dem Eingabewort x, beschriftet, der Kopf auf irgendeine Zelle des Bandes eingestellt und die Turingmaschine in dem so definierten **Anfangszustand** gestartet. Nach dem Start sucht die Turingmaschine das erste Zeichen des Eingabewortes und transformiert es dann schrittweise. Sobald die Steuereinheit in den Haltezustand übergeht, hält sie an. Die Zeichenkette, die nun auf dem Band steht, ist das Ausgabewort y. Wenn man der Reihe nach die Wörter einer Wortmenge X eingibt und wenn die Turingmaschine zu jedem Eingabewort x ein Ausgabewort y ausgibt, sagt man, dass sie die Funktion $f\colon X \to Y$ berechnet. Für die Berechnung einer anderen Funktion ist ein Turingautomat mit einer anderen Automatentafel, eventuell auch mit einem anderen Alphabet erforderlich.

Wenn das aktuelle Paar (Input, alter innerer Zustand) nicht in der Automatentafel enthalten ist, hält der Turingautomat an, obwohl er sich nicht im Endzustand befindet. Es ist auch möglich, dass er niemals anhält, weil er den Endzustand niemals erreicht. In beiden Fällen sagt man, dass die Funktion $f(x)$ für die betreffenden x-Werte nicht definiert ist und spricht von **partieller** Funktion. Ist sie für sämtliche x-Werte definiert, heißt sie **total**. Diese Unterscheidung ist nicht an die Turingmaschine gebunden. Allgemein hat man vereinbart: *Eine Funktion heißt* ***total*** *bzw.* ***partiell***

hinsichtlich einer vorgegebenen Argumentwertemenge, wenn die Funktionswerte für sämtliche bzw. nicht für sämtliche Argumentwerte definiert sind.

Was Turingmaschinen zu leisten vermögen, ist nicht ohne weiteres zu erkennen. Angesichts ihrer äußerst elementaren Funktionsweise ist der empirische Fakt überraschend, dass keine effektiv berechenbare Funktion gefunden worden ist, für die sich nicht eine Turingmaschine angeben lässt, die sie berechnet. Turing selber hat seinen Automaten zum **universellen** Automaten erweitert, der die Funktionsweise *jedes* **speziellen** Automaten simulieren kann, also jedes Automaten mit einer bestimmten Automatentafel. Dazu wird ihm per Bandinschrift die Tafel desjenigen Automaten mitgeteilt, den er simulieren soll.

Turing war überzeugt, das eingangs gestellte Ziel erreicht zu haben, und meinte, dass sich zu jeder effektiv berechenbaren Funktion eine Berechnungsvorschrift in Form einer Automatentafel angeben lässt, oder, wie man auch sagt, dass jede effektiv berechenbare Funktion *turingberechenbar* ist, oder in unserer Sprechweise, dass jede effektiv berechenbare Funktion eine *Turingfunktion* ist. Doch lässt sich das, wie wir wissen, nicht beweisen. Die Begriffe der Turingmaschine und der Turingberechenbarkeit sind zu Grundbegriffen der Algorithmentheorie und der theoretischen Informatik geworden.

Turing befolgte das Trägerprinzip; seine Maschine ist realisierbar. Doch braucht sie nicht gebaut zu werden, um mit ihr zu arbeiten, beispielsweise um zu untersuchen, welche Funktionen die *gedachte* Maschine berechnen kann.

8.4.3 Unbeschränkte Registermaschine (URM)

Bedeutend jüngeren Datums ist die **unbeschränkte Registermaschine**, kurz **Registermaschine**, abgekürzt **URM**. Sie wurde von Nigel Cutland eingeführt und beschrieben [Cutland 80]. Vorgänger der URM war ein Vorschlag von J.C.Shepherdson und H.E.Sturgis aus dem Jahre 1963. Auch die Registermaschine ist eine *fiktive* Maschine, jedoch nicht so elementarer Konstruktion wie die Turingmaschine, und die Operationen, die sie ausführt, sind bedeutend komplexer.

Die URM besteht aus einer unbeschränkten Menge von **Registern** (Speicherplätzen für Zeichenketten, z.B. für Zahlen). In einem Register kann eine ganze positive Zahl unbeschränkter Länge abgspeichert werden. Jedes Register hat einen Namen (eine Adresse). Der Inhalt eines Registers kann mittels bestimmter Befehle verändert werden. Ein Programm ist eine durchnummerierte Folge von Befehlen, die von einem fiktiven Interpretierer (einem Menschen oder einem Gerät) ausgeführt werden.

Es gibt 4 Befehle:

1. Register löschen (0 einspeichern);
2. Inhalt eines Registers inkrementieren (1 addieren);
3. Inhalte zweier Register austauschen;
4. bedingter Sprung, d.h. Vor- oder Zurückspringen im Programm, wenn zwei bestimmte Register die gleiche Zahl enthalten. Der anzuspringende Befehl und die zu vergleichenden Register werden im Befehl angegeben.

Die Ähnlichkeiten und Unterschiede zwischen Registermaschine und universeller Turingmaschine sind offensichtlich. Beide sind Automaten mit unbeschränktem Speicher. Ein Programm der Registermaschine entspricht der Automatentafel der Turingmaschine, erinnert aber schon sehr an Programme, die in der sog. Assemblersprache gängiger Computer (siehe Kap.15.4) geschrieben sind. Ein Programm legt - ebenso wie eine Automatentafel - eine Funktion fest. Es lässt sich zeigen, dass die Registermaschine dasselbe leistet (dieselben Funktionen berechnen kann), wie die universelle Turingmaschine. Eine Funktion, die durch ein URM-Programm festgelegt und berechnet werden kann, heißt **URM-berechenbar**; wir nennen sie **URM-Funktion**.

8.4.4 Markovalgorithmus

Eine andere, von der turingschen scheinbar sehr unterschiedliche Idee liegt dem *Normalalgorithmus* von A.A.Markov[12], kurz dem **Markovalgorithmus** aus dem Jahre 1954 zugrunde. Auch Markov suchte nach einer universellen Vorschrift für das Transformieren von Zeichenketten, wobei er - im Gegensatz zu Turing - nicht das schrittweise Ersetzen einzelner Zeichen, sondern längerer Teile von Zeichenketten (von Wörtern) im Auge hatte. Er führte keine fiktive Maschine ein, sondern ging offenbar davon aus, dass seine Normalalgorithmen von Menschen interpretiert werden.

Aufgabe des Interpretierers ist es, Zeichenketten nach folgender Vorschrift zu transformieren. Gegeben ist eine Liste (geordnete Menge) von Ersetzungsregeln, sogenannten **Substitutionsregeln**, z.B. $2+3 \rightarrow 5$ oder $ab \rightarrow cd$, was bedeutet, dass 2+3 durch 5 bzw. *ab* durch *cd* ersetzt (substituiert) werden kann. Die Transformation erfolgt schrittweise. In jedem Schritt geht der Interpretierer die Regelliste durch. Er beginnt mit der ersten Regel und sucht die zu transformierende Zeichenkette von links beginnend nach der linken Seite der Regel durch. Wenn die Suche erfolglos ist, geht er zur nächsten Regel über. Sobald eine Regel anwendbar ist, wenn also ihre linke Seite in der zu transformierenden Zeichenkette enthalten ist, wird die betreffende Substitution vorgenommen. Wenn die Zeichenkette bis zum Ende durchsucht ist, wiederholt sich der Prozess mit der nächsten Regel. Wenn keine Regel mehr anwendbar ist oder wenn auf der rechten Seite der gefundenen Regel das *Stopzeichen* steht, wird der Transformationsprozess beendet. Das Stopzeichen gehört zum verwendeten Alphabet.

Das Transformieren einer Zeichenkette erinnert an die Arbeit mit einer Formelsammlung, in der man nach einer passenden Formel sucht. Dabei findet man z.B. $\sin x/\cos x = \tan x$, was bedeutet, dass in einem mathematischen Ausdruck $\sin x/\cos x$ durch $\tan x$ ersetzt werden darf. Natürlich wird man mit dem Suchen nicht auf Seite 1 beginnen, sondern intelligenter vorgehen.

---

12 Sohn des Wahrscheinlichkeitstheoretikers A.A.Markov.

Ein Operator, der Zeichenketten gemäß Formeln umwandelt, wird **Formelmanipulator** genannt. Ein Markovalgorithmus ist also, zusammen mit seinem Interpretierer, ein Formelmanipulator. Als Manipulator kann ein Computer dienen, der über entsprechende Programme verfügt. Wenn der Programmierer sich nichts Intelligenteres hat einfallen lassen, beginnt der Computer jede Suche mit der ersten Formel, sozusagen auf Seite 1. In Kap.15.8 wird näher auf die Formelmanipulation eingegangen.

Transformiert ein Markovalgorithmus Wörter einer Wortmenge X eindeutig in Wörter einer Menge Y, so sagt man, dass er die Funktion f: $X \rightarrow Y$ berechnet. Eine solche Funktion wird **markovberechenbar** genannt; wir nennen sie **Markovfunktion**. Wieder stellt sich die Frage, ob die Methode universell ist, d.h. ob jede effektiv berechenbare Funktion auch markovberechenbar ist. Markov selber war, ebenso wie Turing, der Meinung, dass seine Methode universell sei. Es stellte sich heraus, dass er damit ebenso Recht hatte wie Turing und dass die Klasse der Markovfunktionen mit der Klasse der Turingfunktionen identisch ist. Wir kommen darauf in Kap.15.8 zurück.[13]

Zwischen Markovalgorithmus und Turingmaschine sind gewisse Ähnlichkeiten zu erkennen. Ein *spezieller* Markovalgorithmus, also ein Markovalgorithmus mit einer bestimmten Regelliste, entspricht einer *speziellen* Turingmaschine; der Regelliste entspricht die Automatentafel und dem Interpretierer des Markovalgorithmus die Steuereinheit der Turingmaschine. Die Arbeitsweise des Interpretierers von Markovalgorithmen ist jedoch nicht soweit "durchautomatisiert" wie die der Steuereinheit der Turingmaschine.

8.4.5 Rekursive Funktionen und USB-Funktionen

Historisch gesehen hätte die nun zu besprechende *rekursive* Methode, Funktionen festzulegen, am Anfang stehen müssen. Die nach dieser Methode beschreibbaren und berechenbaren Funktionen heißen **rekursive Funktionen**. Wir werden die Methode gemeinsam mit der jüngsten der in Kap.8.4.1 aufgezählten sechs Methoden der Algorithmenartikulierung, der USB-Methode, darlegen und die rekursive Definition von Funktionen vollständig in die Sprache der uniformen Systembeschreibung übersetzen, was einer Veranschaulichung der rekursiven Definition gleichkommt. Denn die USB-Methode arbeitet mit "anschaulichen Bildern", ihre Sprache ist eine *zweidimensionale* Sprache, sie verwendet graphische Darstellungen. Auf diese Weise wird zugleich nachgewiesen, dass jede rekursiv definierbare Funktion eine USB-Funktion ist.

Der Idee der rekursiven Metode sind wir bereits in Kap.7.2 [7.12] begegnet. Dort war das Multiplizieren auf das iterative Addieren und dieses auf das iterative Inkrementieren (Erhöhen um 1) zurückgeführt worden. Man erinnere sich an den

---

13 Ausführlicher ist der Markovalgorithmus in [Malcew 74] beschrieben.

ABC-Schützen. Die Idee der Methode besteht nun darin, die Definition und die Berechnung *jeder beliebigen* Funktion auf das Inkrementieren zurückzuführen.

Es bedurfte der Bemühungen vieler Mathematiker - genannt seien J.Herbrand, K.Gödel und S.C.Kleene -, um diese Idee in eine mathematische Form zu gießen, und so die Begriffe der Funktion und des Algorithmus exakt zu definieren. Das Ergebnis war der Begriff der *rekursiven Funktion*. In ihm sind zwei Bedeutungskomponenten des Wortes "Rekursion" miteinander verknüpft, das Zurück*führen* und das Zurück*greifen*. In jedem Schritt einer rekursiven Berechnung wird auf das Resultat derjenigen Operation (z.B. des Addierens) *zurückgegriffen*, auf welche die auszuführende Operation (z.B. das Multiplizieren) *zurückgeführt* wird.

Um zu einer konstruktiven Methode zur Festlegung von Funktionen (Operationen, Operatoren) zu gelangen, m.a.W. um eine Komponierungsmethode beliebiger Kompositoperatoren zu entwickeln, ist es notwendig, eine Menge elementarer, d.h. nicht weiter dekomponierbarer Operatoren sowie eine Menge von Komponierungsregeln festzulegen. Beide Mengen sollten möglichst klein sein. Dafür gibt es viele Möglichkeiten. Eine Möglichkeit ist die USB-Methode. Eine andere Möglichkeit, die sehr häufig der Definition rekursiver Funktionen zugrunde gelegt wird (z.B. in [Hermes 71]) geht von *einer einzigen elementaren Operation* aus, dem **Inkrementieren** (Erhöhung um 1). Als Anfangswert des Inkrementierens wird die Null festgelegt. Es gibt vier Komponierungsregeln, nach denen aus dem Inkrementieren Kompositoperationen komponiert werden können:

1. Funktionale Substitution
2. Selektion
3. Rekursive Iteration
4. Minimalisierung.

Wir nennen sie *rekursive* **Komponierungsmittel** ; Funktionen, die sich mit ihrer Hilfe beschreiben lassen, nennen wir **rekursive Funktionen**. Die Operanden (Argument- und Funktionswerte) rekursiver Funktionen sind ganze nichtnegative Zahlen. Die Operandenmenge kann durch Umcodierung erweitert werden, wie in Kap.8.5 [37] gezeigt wird.

Den Begriff der *rekursiven Beschreibbarkeit* werden wir im Weiteren in folgendem verallgemeinertem Sinne verwenden: *Ein Komponierungsmittel heißt* ***rekursiv beschreibbar****, wenn es mit Hilfe der genannten vier rekursiven Komponierungsmittel beschreibbar ist.* In der Algorithmentheorie werden Komponierungsregeln eventuell als elementare Operatoren (Funktionen) aufgefasst. Die Wirkungsweise der Komponierungsregeln sollen der Reihe nach besprochen werden.

Funktionale Substitution

Substitutionen sind wir bereits begegnet, und zwar im Zusammenhang mit dem Markovalgorithmus. Dort wurden Zeichenketten durch andere Zeichenketten substituiert, ungeachtet der Bedeutung, die diese Zeichenketten in ihrem Kontext vielleicht

besitzen. Jetzt führen wir eine spezielle Art der Zeichenkettensubstitution ein. Sie besteht darin, dass eine Argumentvariable einer Funktion durch eine andere Funktion ersetzt wird, die den Wert der Argumentvariablen berechnet. In diesem Fall sprechen wir von **funktionaler Substitution**. Wenn beispielsweise in der Funktion $y = f(x)$ die Variable x durch $g(x)$ ersetzt wird, liegt eine funktionale Substitution vor; sie liefert die **geschachtelte** Funktion

$$y = f(g(x)) \tag{8.6}$$

Wenn nichts anderes gesagt wird, ist im Weiteren unter einer Substitution eine funktionale Substitution zu verstehen.

Eine Substitution, bei der eine einzige Argumentvariable substituiert wird, lässt sich nach der USB-Methode als *Sequenz* (Kette) zweier Operatoren darstellen. Gemeinsam bilden sie einen Kompositoperator, der eine *Kompositfunktion,* und zwar eine geschachtelte Funktion berechnet. Die Funktion (8.6) ist dafür ein Beispiel. Sie wird von einem f-Operator berechnet, dem ein g-Operator vorgeschaltet ist.

Auf diese Weise können auch mehrfach geschachtelte Funktionen dargestellt werden. Wenn beispielsweise der Operator h_1 in Bild 8.7a die Funktion $y = 2x$ berechnet, h_2 die Funktion $y = \sin x$ und h_3 die Funktion $y = x^2$, dann berechnet die Operatorenkette die doppelt geschachtelte Funktion $y = (\sin(2x))^2$. Ohne Angabe der konkreten Funktionen lässt sich die doppelte Schachtelung folgendermaßen notieren:

$$y = f(x) = h_3(h_2(h_1(x))) \tag{8.7}$$

Bild 8.7a zeigt das entsprechende Operatorennetz.

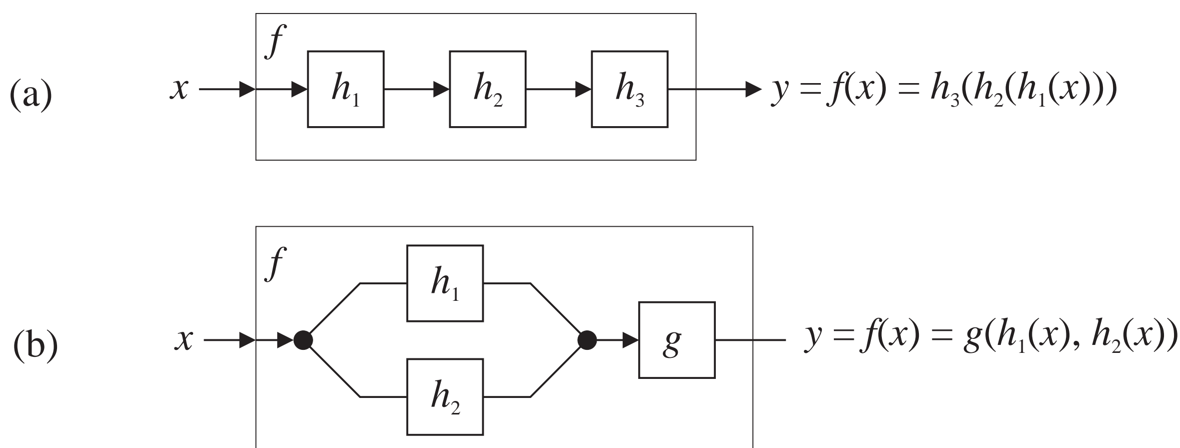


Bild 8.7 Funktionale Substitution; **(a)** - einstellig; **(b)** - zweistellig

In den bisherigen Beispielen für Substitutionen wird das Argument einer einstelligen Funktion substituiert. Solche Substitutionen nennen wir **einstellig**. Die funktionale Substitution lässt sich auch auf mehrstellige Funktionen anwenden. Dann sprechen wir von **mehrstelliger Substitution**. Bild 8.7b zeigt das Operatorennetz

einer zweistelligen Funktion. Die Argumente werden durch die Funktionen $h_1(x)$ bzw. $h_2(x)$ substituiert, die beide von derselben Variablen x abhängen. Die Gabel ist also eine Kopiergabel und die Kompositfunktion

$$f(x) = g((h_1(x),h_2(x)) \tag{8.8a}$$

ist einstellig. Das Operatorennetz von Bild 8.1 ist dafür ein konkretes Beispiel. Dabei ist h_1 der Potenzierer, h_2 ist je nach Stellung der Zweigeweiche vor dem Sinusoperator entweder der Sinusoperator oder der Identitätsoperator (der Übergabeweg, der den Sinusoperator überbrückt) und g ist der Additionsoperator. Wenn die substituierenden Funktionen h_1 und h_2 von verschiedenen Variablen abhängen, ist die Gabel eine Spaltegabel und die Kompositfunktion ist zweistellig:

$$f(x_1,x_2) = g((h_1(x_1),h_2(x_2)). \tag{8.8b}$$

Bild 8.7 und die Formeln (8.8a) und (8.8b) demonstrieren, wie sich starre Flussknoten mit Hilfe der rekursiven Komponierungsmittel beschreiben lassen. Den Gabeln entsprechen Funktionspaare und den Vereinungen zweistellige Funktionen.

An dieser Stelle sei auf folgenden Sachverhalt aufmerksam gemacht. Weder in den Operatorennetzen von Bild 8.7 noch in den entsprechenden geschachtelten Funktionen (8.7), (8.8a) und (8.8b) treten die Operanden, die übergeben bzw. substituiert werden, explizit auf, mit Ausnahme des externen Eingabeoperanden x bzw. (x_1,x_2). Es ist leicht einzusehen, dass dies nicht erforderlich ist. Denn die Operanden bzw. Argumente ergeben sich aus den Notationsregeln (Syntaxregeln) der jeweiligen Sprache. Die zweidimensionale Sprache von Bild 8.7 ist die Sprache der uniformen Systembeschreibung, die **USB-Sprache**. Die eindimensionale Sprache, in der die Ausdrücke (8.7) und (8.8) notiert sind, heißt **funktionale Sprache**, weil die Notation nur Funktionsbezeichner und Klammern verwendet, wenn man von den Bezeichnern der externen Eingabeoperanden absieht.

In der USB-Sprache zeigt eine Verbindung *von* einem Operator (Vorgängeroperator) *zu* einem anderen Operator (Nachfolgeoperator) an, dass der Eingabeoperand des Nachfolgeoperators mit dem Ausgabeoperanden des Vorgängeroperators identisch ist. Darum braucht er nicht angegeben zu werden. Die Rolle der Verbindungen der USB-Notation übernimmt in der funktionalen Notation die Schachtelung. Sie zeigt an, dass der Wert, den eine innere Funktion berechnet, der äußeren Funktion als Argumentwert "übergeben" wird.

24 Die Eigenschaft, dass innere Operanden nicht auftreten, wird uns im Weiteren wiederholt beschäftigen. Sie weist USB- und funktionale Sprachen als *nichtimperative* Sprachen aus, d.h. als Sprachen, in denen keine *imperativen Algorithmen* artikuliert werden, also Algorithmen, die in jedem Berechnungsschritt alle beteiligten Operanden explizit angeben (vgl. Kap.7.2 [7.10]). Ein imperativer Algorithmus zur Berechnung der Funktion (8.8a) könnte folgende Form haben:

$a := h_1(x)$
$b := h_2(x)$
$c := g(a,b)$
$f := c$

Man sagt, dass diese Vorschrift *imperativ*, die Vorschrift (8.8a) dagegen *funktional* notiert ist und spricht von **imperativer** bzw. **funktionaler Notation**.

In der Mathematik wird bekanntlich vorwiegend die funktionale Notation benutzt. Sie ist sehr kompakt, hat aber den Nachteil, dass sie zu Missverständnissen führen kann, die daher rühren, dass ein Funktionsbezeichner zum einen eine *Funktion* bezeichnet, und dass er zum anderen zusammen mit einem Argumentwert bzw. Argumentwertetupel den Wert der Funktion bezeichnet. Wenn ein Funktionsargument durch eine Funktion f substituiert wird, stellt f bei der Auswertung nicht die Funktion, sondern einen bestimmten Funktionswert dar. Dies ist die Wurzel des bekannten Streits, ob mit $f(x)$ ein *bestimmter* Wert oder *alle* Werte von f gemeint sind. Diese Zweideutigkeit hat CHURCH durch die Einführung des Lambda-Operators behoben (siehe Kap.8.4.7). 25

Selektion

Die Argumente von Funktionen können Tupel, z.B. Vektoren (Zahlentupel) sein. Um mit Vektoren zu rechnen, um z.B. das Skalarprodukt zweier Vektoren zu bilden, muss auf die einzelnen Komponenten der Vektoren (Tupel) zugeriffen werden. Dieses Zugreifen wird Komponentenselektion oder kurz **Selektion** genannt. Der ausführende Operator heißt **Selektor**. Er heißt ***n*-stellig**, wenn er aus einem Tupel n Elemente auswählt.

Bild 8.8 zeigt die USB-Darstellung von Selektoren und demonstriert damit, dass aus der Sicht der USB-Methode auch die Selektion keine Bausteinoperation, sondern ein Mittel des Komponierens ist. Sie kann durch eine Spaltegabel beschrieben werden, über deren einen Ausgang die zu selektierende Komponente weitergeleitet wird, während der andere Ausgang stillgelegt ist. Dabei handelt es sich um einen **starren** (nicht steuerbaren) Selektor (Bild 8.8a). Die Algorithmentheorie arbeitet i.Allg. nur mit *einstelligen* starren Selektoren.

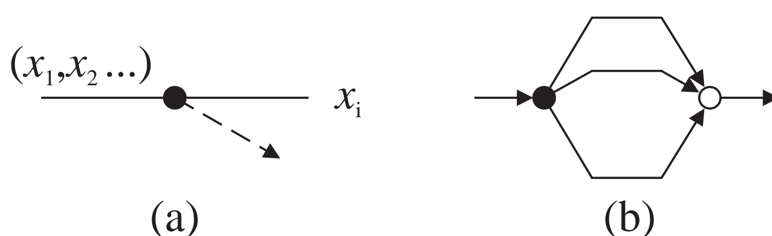


Bild 8.8 Selektoren. **(a)** - starrer Seletor; **(b)** - steuerbarer Selektor.

Bild 8.8b zeigt einen **steuerbaren Selektor**. Er besteht aus einer Spaltegabel und einer Sammelweiche mit soviel Eingängen, wie das Eingabetupel Elemente (bzw. Teiltupel) besitzt. Durch Steuerung der Weiche kann das gewünschte Element des Tupels ausgewählt werden. Steuerbare Selektoren kann man als "Entscheider" zwischen mehreren "Angeboten" auffassen. Der einfachste steuerbare Selektor, der aus einem Paar ein Element auswählt, entscheidet eine Alternative. Bild 8.8b zeigt, dass sich Sammelweichen mit Hilfe der Selektion rekursiv beschreiben lassen. Gleichzeitig zeigt es, dass sich Alternativmaschen rekursiv beschreiben lassen, vorausgesetzt, in der Masche befinden sich niemals mehrere Operanden gleichzeitig (vgl. Kap.8.1 [1])

Selektoren stellen **Kompositflussknoten** dar, also Flussknoten, die aus den "elementaren" Flussknoten von Bild 8.2 komponiert werden. Man kann sich die verschiedensten Kompositflussknoten ausdenken. Beispielsweise lassen sich mit Hilfe von Selektoren steuerbare Spaltegabeln realisieren. In Kap.12.3.2 werden umfangreiche Kompositflussknoten und ihre elektronische Realisierung besprochen.

Nachdem wir die *Gabel mit einem einzigen Ausgang* als Symbol für den *starren Selektor* eingeführt haben, bietet es sich an, die *Zweigeweiche mit einem einzigen Ausgang* (dargestellt als kleiner Kreis innerhalb einer Verbindungslinie) als Symbol für das *Tor* einzuführen. Ein **Tor** kann als Zweigeweiche mit einem toten Ausgabezweig angesehen werden.

Rekursive Iteration

Eine Kette identischer Operatoren führt ein und dieselbe Operation wiederholt (iteriert) aus. Der so entstehende Kompositoperator heißt **Iterator**. Eine Iteration kann demnach als *sequenzielle Substitution* aufgefasst werden, bei der jedes Mal die gleiche Substitution ausgeführt wird. *Iteration ist also eine spezielle Art der sequenziellen Substitution.* Die hardwaremäßige Realisierung kann bei hoher Iterationszahl (Wiederholungszahl) recht aufwendig werden. Der Aufwand lässt sich dadurch verringern, dass der Operator nur einmal realisiert, aber mit einer Rückkopplungsschleife versehen wird. Der Potenzierer (pot) in Bild 8.1 ist dafür ein Beispiel. Die Kette ist gewissermaßen zu einer Iterationsschleife *eingerollt.* Umgekehrt kann eine Iterationsschleife in eine Kette "*ausgerollt*" werden; bei ihrer Auswertung *muss* sie ausgerollt werden.

Die Möglichkeit, Iterationen durch Rückkopplung zu realisieren, besteht nicht nur für reale, sondern auch für sprachliche Operatoren und Algorithmen. Dann kann die Rückkopplung durch Zurückspringen im Algorithmus verwirklicht werden, oder dadurch, dass der Algorithmus sich selber enthält (sich selbst aufruft). Diese Art der Iteration nennen wir **rekursive Iteration**. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter Iteration stets *rekursive* Iteration und unter einem Iterator ein Operator zu verstehen, der eine *rekursive* Iteration ausführt.

Bild 8.9 zeigt den Iterator in seiner allgemeinsten Form als Operatorennetz. Der Leser wird erkennen, dass die Zweigeweiche Z2 und die Sammelweiche S2 die Weichen einer Alternativmasche und Z1 und S1 die Weichen einer Rückkopplungsschleife sind. Der Iterator führt die Operation *h* mehrmals aus. Die Rückkopplungsschleife, in der die eigentliche Iteration abläuft, ist fett gezeichnet. Mit *n* ist die Anzahl der ausgeführten Iterationsschritte bezeichnet, wobei die erste Operationsausführung mitgezählt wird, obwohl sie eigentlich noch keine Wiederholung darstellt. Die Funktion, die ein Iterator bei *n* Iterationsschritten liefert, bezeichnen wir mit f_n.

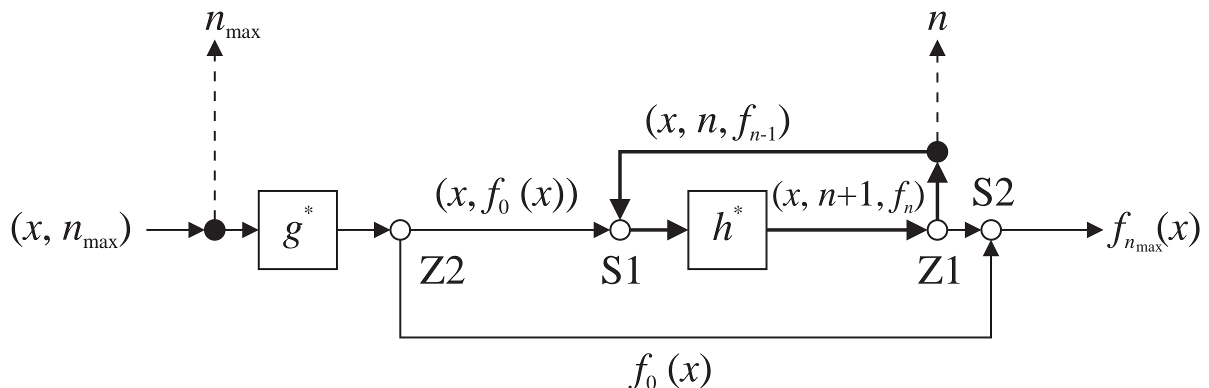


Bild 8.9 Rekursiver Iterator

Der Potenzierer von Bild 8.1 (pot-Operator, gestrichelt umrahmt) ist ein Beispiel für einen rekursiven Iterator, denn das Potenzieren wird durch wiederholtes Multiplizieren verwirklicht. Der *h*-Operator ist in diesem Fall ein Multiplizierer. Das Operatorennetz des Iterators in Bild 8.9 unterscheidet sich von dem des pot-Operators in Bild 8.1 in dreierlei Hinsicht. Erstens ist der *h*-Operator zu einem h^*-Operator erweitert worden (die Erweiterung wird sogleich erläutert), zweitens ist ein zusätzlicher *g*-Operator eingeführt worden und drittens ist die *Vereinung* vor dem Multiplizierer durch eine *Sammelweiche* (S1) ersetzt worden (in Bild 8.1 waren die beiden Faktoren ausnahmsweise nicht gedanklich zu einem Paar vereint worden). Mit den Änderungen werden zwei Ziele verfolgt. Zum einen soll der Iterator universell sein, zum anderen soll das Operatorennetz des Iterators die Notationsweise widerspiegeln, die in der Algorithmentheorie üblich ist, insbesondere die Notationsweise der Formeln (8.9). Dies ist auch der Grund dafür, dass die Iterationszahl, die in den übrigen Kapiteln mit *it* bezeichnet wird, in diesem Kapitel mit *n* bezeichnet wird.

Der h^*-Operator ist dem bisherigen *h*-Operator gegenüber in zweifacher Hinsicht erweitert. Erstens gibt der Operator *f* nicht nur den laufenden Funktionswert f_n, sondern auch den Argumentwert *x* aus. Wir sagen, dass der Operator den Eingabeoperand durchzieht oder *durchschleppt*. Das ist notwendig (und hardwaremäßig leicht realisierbar), wenn die Funktion *h* nicht nur vom laufenden Iterationsergebnis f_n, sondern auch von *x* selber explizit abhängt, wie z.B. bei der iterativen Berechung der

Potenzfunktion, wobei in jedem Schritt das Produkt x^n*x berechnet wird. Das Iterationsergebnis kann auch von der Iterationszahl abhängen, wie beispielsweise bei der iterativen Berechnung der Fakultät $n!$ oder bei der Berechnung der Sinusfunktion durch Reihenentwicklung (siehe Formel (15.5) in Kap.15.2). Dazu rüsten wir den Operator h^* mit einem Zähler aus, der die Iterationsschritte zählt und das aktuelle Zählergebnis ausgibt. Darin besteht die zweite Erweiterung des Operators *h*. Der vollständige h^*-Operator gibt das Tripel $(x,n+1,f_n)$ aus.

Der *g*-Operators stellt den Funktionswert f_0 bereit, mit dem die Iteration beginnt, es gilt $g(x) = f_0$. Im Falle des Potenzierers ist der Anfangswert $g(x) = f_0 = 1$, hängt also nicht von *x* ab. In Bild 8.1 ist seine Eingabe durch den senkrechten kurzen Eingabepfeil des Multiplizierers angedeutet. Der *h*-Operator berechnet im ersten Iterationsschritt das Produkt $1*x$. Beim Multiplizieren durch Iteration (durch wiederholtes Addieren) ist $f_0 = 0$. Der *h*-Operator benötigt beim Multiplizieren und beim Potenzieren durch Iteration außer dem f_0-Wert auch den *x*-Wert. Das ist auch bei vielen anderen iterativen Berechnungen der Fall. Der *g*-Operator muss dann das Paar (x,f_0) dem *h*-Operator übergeben, d.h. er muss den *x*-Wert durchschleppen.

Das Ausgabetripel des h^*-Operators kann über die Sammelweiche S1 auf den Eingang des h^*-Operators zurückgegeben werden, der es dann im nächsten Iterationsschritt weiterverarbeitet. In der Rückkopplungsschleife muss also - ebenso wie im Falle des Automaten in Bild 8.5 - ein “Schrittverzögerer” liegen; er ist nicht eingezeichnet. Im ersten Iterationsschritt wird über S1 dem h^*-Operator das Ausgabepaar des *g*-Operators zugeleitet. Über Z1, S2 und einen Selektor (nicht eingezeichnet), der f_n aus dem Ausgabetripel von h^* ausselektiert, kann das Resultat der Kompositoperation (der Iteration) ausgegeben werden. Wenn kein Iterationsschritt stattfinden soll, kann der Anfangswert f_0 vom *g*-Operator über Z2 und S2 ausgegeben werden.

Die Weichen müssen vom ausführenden realen Operator (Interpretierer) gestellt werden. Wird die Iteration vom Menschen ausgeführt, tritt die Weichenstellung als Entscheidung darüber ins Bewusstsein, was mit dem gerade berechneten Wert weiter zu geschehen hat. Ein technischer Iterator muss einen *Steueroperator* enthalten, der die erforderlichen Steuersignale generiert (vgl. Bild 8.1b). Der Steueroperator des Operatorennetzes von Bild 8.9 muss feststellen, ob die maximale Iterationszahl n_{max}, nach der die Iteration abgebrochen werden soll, erreicht ist. Dazu vergleicht er in jedem Iterationsschritt die laufende Iterationszahl *n* mit n_{max} und entscheidet das Abbruchprädikat, das z.B. $[n=n_{max}]$ oder $[n<n_{max}]$ lauten kann. Im zweiten Fall[14] muss der Steueroperator ein Steuersignal ausgeben, sobald das Steuerprädikat nicht mehr erfüllt ist. Das Steuersignal stellt die Zweigeweiche Z1, die bis dahin auf *Rückkopplung* gestellt war, auf *Ausgabe*.

---

14 Dem entspricht das Steuerprädikat $[it<n]$ in den Bildern 8.1 und 20.10 und das Steuerprädikat $[z<n]$ in Bild 15.4.

Aus Bild 8.9 ist abzulesen, dass der Iterator die Funktion $f(x,n)$ rekursiv gemäß den Formeln

$$f(x,n) = h(x,n,f(x,n\text{-}1)) \quad \text{für } n > 0 \tag{8.9a}$$

$$f(x,0) = f_0 = g(x) \quad \text{für } n = 0. \tag{8.9b}$$

berechnet. Dies ist eine in der Algorithmentheorie häufig benutzte Notation der Iteration.

Beim Multiplizieren bzw. Potenzieren durch Iteration geht (8.9) in (8.10) bzw. (8.11) über. Die Ausdrücke stellen die Rekursionsformeln zur Berechnung des Produktes $n*x$ bzw. der Potenz x^n dar.

$$f(x,n) = x + f(x,n\text{-}1)); \;\; \mathrm{f}_0 = 0 \tag{8.10}$$

$$f(x,n) = x * f(x,n\text{-}1)); \;\; \mathrm{f}_0 = 1 \tag{8.11}$$

Eine Funktion, die sich mit Hilfe des Inkrementierens, der Substitution, der Selektion und der Iteration definieren lässt, heißt ***primitiv-rekursive Funktion****. Primitiv-rekursive Funktionen sind* ***USB-Funktionen****.* Die Richtigkeit des letzten Satzes ergibt sich aus der Tatsache, dass sich Substitution, Selektion und Iteration mittels der USB-Methode beschreiben lassen.

Minimalisierung

Zunächst glaubte man, mit den primitiv-rekursiven Funktionen alle berechenbaren Funktionen erfasst zu haben. Diese Annahme stellte sich jedoch als Irrtum heraus, als es gelang, Funktionen zu konstruieren, die berechenbar, aber nicht primitiv-rekursiv sind. Das trifft z.B. für die Ackermann-Funktion[15] zu. Daraufhin suchte man nach einer Erweiterung der konstruktiven Funktionsdefinition durch Hinzunahme einer zusätzlichen Komponierungsregel und fand diese in der sog. *Minimalisierung*. Auch sie wird in der Algorithmentheorie i.Allg. nicht als Komponierungsregel, sondern als Operator (Funktion) aufgefasst.

Das Wort *Minimalisierung* bedeutet (im Zusammenhang mit der Definition rekursiver Funktionen) "*Bestimmung der minimalen Iterationszahl*". Minimalisierung ist anzuwenden, wenn n_{max} nicht vorgegeben ist, sondern sich erst während der Iteration ergibt. Der Sachverhalt lässt sich an der Vorgehensweise des ABC-Schützen aus Kap.7.2 [7.6] veranschaulichen, der eine Differenz, z.B. 8 minus 3 dadurch bildet, dass er den Subtrahend 3 so oft um 1 erhöht, bis er den Minuend 8 erreicht, wobei er die Anzahl der Inkrementierungen an den Fingern mitzählt. Der ABC-Schütze führt also eine Iteration *ohne* vorgegebene Iterationszahl durch. Er bricht die Iteration ab, sobald der Minuend m gleich dem erhöhten Subtrahenden wird, also sobald das

15 Siehe z.B. [Duden 89], [Werner 95].

Prädikat [*m = erhöhter Subtrahend*] erfüllt ist, m.a.W. sobald die charakteristische Funktion, *d.h. die Differenz* (*m* minus *erhöhter Subtrahend*), zu Null wird.

Analog kann das Dividieren ganzer Zahlen auf iteratives Addieren zurückgeführt werden. Wenn aber der Dividend kein Vielfaches des Divisors ist, kann das Resultat der wiederholten Addition niemals gleich dem Dividenden werden und der Iterationsprozess findet kein Ende. Damit er abbricht, muss das Prädikat lauten "Das Iterationsergebnis ist gleich oder größer als der Dividend". Der Iterationsprozess wird bei dem frühesten Iterationsergebnis (bei der *minimalen* Iterationszahl) abgebrochen, für welches das Prädikat erfüllt ist. Das Ergebnis ist ein aufgerundeter Quotient. Derjenige Operator, der die minimale Iterationszahl bestimmt, wird **μ-Operator** genannt.

Ein typisches Beispiel für die Anwendung des μ-Operators ist eine iterative Näherungsrechnung, die abgebrochen werden soll, sobald die Korrektur (absolut genommen) im laufenden Schritt eine vorgegebene Schranke nicht überschreitet. Man denke z.B. an die Berechnung der Sinusfunktion nach einer Näherungsformel (z.B. nach der Formel (15.5)). Auch in diesem Fall ist n_{max} nicht vorgegeben, sondern muss mittels μ-Operator aus dem Abbruchprädikat $[|f_n - f_{n-1}| \leq \varepsilon]$ bestimmt werden. Mit ε ist die vorgegebene Schranke bezeichnet.

Damit der Steueroperator die Abbruchsituation erkennen kann, muss er außer der Schranke die Größen f_n und f_{n-1} kennen. Sie müssen ihm über den rechten nach oben gerichteten gestrichelten Pfeil in Bild 8.9 übergeben werden. Der Steueroperator führt dann die Funktion eines μ-Operators aus, denn er bestimmt die minimale Iterationszahl, bei der die geforderte Bedingung erfüllt ist.

Die allgemeine Definition des μ-Operators lautet: *Ein Operator heißt* ***μ-Operator****, wenn er einem Prädikat P(x,n) den minimalen Wert von n zuordnet, für den P erfüllt ist, wobei n die Iterationzahl eines Iterationsprozesses darstellt und x ein Tupel beliebiger Werte sein kann, die während der Iteration bereits berechnet worden sind.* Die Operation des μ-Operators wird **Minimalisieren** genannt. Sie kann bei der Festlegung von Funktionen mittels Iteration zur Anwendung kommen.

Wir vereinbaren: *Eine Funktion, die sich mit Hilfe des Inkrementierens, der Substitution, der Iteration, der Selektion und der Minimalisierung definieren lässt* (wobei nicht alle Komponierungsmittel zum Einsatz kommen müssen), *heißt* ***μ-rekursive Funktion*** *oder kurz* ***rekursive Funktion***. Falls die Definition die Minimalisierung enthält, wird vorausgesetzt, dass das Prädikat *P(x,n)* entscheidbar ist. Damit ergibt sich der folgende wichtige

26

Satz: Rekursive Funktionen können stets nach der USB-Methode komponiert werden, m.a.W. rekursive Funktionen sind USB-Funktionen.

Um zu beweisen, dass auch das Umgekehrte gilt, dass USB-Funktionen rekursive Funktionen sind, muss unter anderem gezeigt werden, dass sich die Komponierungsmittel der USB-Methode, also die Flussknoten, rekursiv beschreiben lassen. Für

starre Flussknoten und Sammelweichen ist das bereits geschehen. Dass auch Zweigeweichen rekursiv beschreibbar sind, zeigt folgende Überlegung.

Da ein Operator definitionsgemäß nur einen einzigen Ausgang besitzt, müssen die Äste einer Zweigeweiche in einem Operandenflussgraph (in dem Operatorennetz eines Kompositoperators) früher oder später zusammenlaufen. Dabei kann sich eine Alternativmasche oder eine Rückkopplungschleife ergeben. Beide lassen sich rekursiv beschreiben, die Alternativmasche mittels des steuerbaren Selektors (siehe Bild 8.8), die Rückkopplungsschleife mittels des rekursiven Iterators (siehe Bild 8.9).

Damit ist gezeigt, dass sämtliche Flussknoten rekursiv beschreibbar sind. Es ist aber noch nicht gezeigt, dass USB-Funktionen immer rekursive Funktionen sind, denn in einem Operandenflussgraphen können sich sowohl Maschen als auch Schleifen überlappen, was rekursiv nicht beschreibbar ist. Auch Überlappungen von Schleifen mit Maschen sind möglich. Bild 8.10a zeigt als Beispiel zwei sich überlappende Alternativmaschen. Je nach Weichenstellung berechnet der Kompositoperator eine der rechts angegebenen Funktionen. Ein Operatorennetz, das keine Überlappungen enthält, heißt **wohlstrukturiert**. Die USB-Methode lässt *nichtwohlstrukturierte* Operatorennetze zu. Wir wollen uns überlegen, ob bzw. wie sich ein nichtwohlstrukturiertes Netz in ein wohlstrukturiertes überführen lässt. Das Nächstliegende ist, das Problem durch Duplizierung von Operatoren zu lösen, wofür Bild 8.10b ein Beispiel gibt. In das Operatorennetz von Bild 8.10a ist ein zweiter *h*-Operator so eingefügt, dass zwei sich nicht überlappende Maschen entstehen, der Kompositoperator aber trotzdem dieselben Funktionen berechnet, wie der ursprüngliche.

27

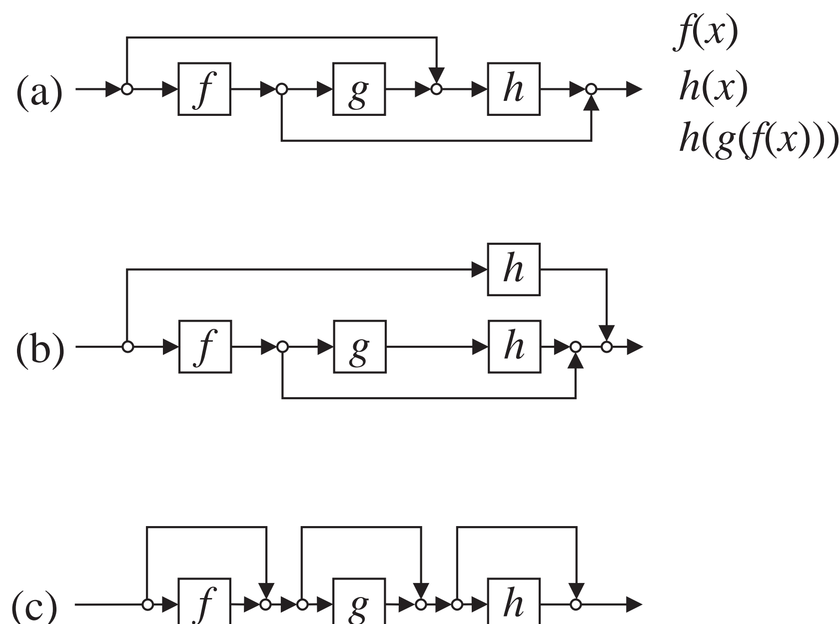


Bild 8.10 Wohlstrukturierung; **(a)** - Überlappung zweier Altenativmaschen; **(b)** - Wohlstrukturierung durch Operatorduplizierung; **(c)** - Wohlstrukturierung durch alternative Überbrückung

Bild 8.10c zeigt eine andere Methode der Wohlstrukturierung, die alternative Überbrückung. Sie besteht darin, dass die relevanten Operatoren durch einen Alternativzweig überbrückt werden. In dem Beispiel müssen alle drei Operatoren überbrückt werden. Dabei ergibt sich eine bedeutend höhere Variabilität des Kompositoperators. Da jeder der drei Bausteinoperatoren in den Kompositoperator einbezogen werden kann oder nicht, lassen sich 8 unterschiedliche Funktionen berechnen, darunter auch die identische Funktion $y = x$, was dem *Durchschleppen* des Eingabeoperanden durch den gesamten Kompositoperator entspricht.

28 Die beschriebenen Methoden der Wohlstrukturierung sind unabhängig davon anwendbar, ob sich Maschen oder Schleifen überlappen. Daraus folgt, dass sich ***nichtwohlstrukturierte*** *Operatorennetze stets in* ***wohlstrukturierte*** *überführen lassen*. Es lässt sich also jeder Operandenlaufplan in eine Struktur überführen, die rekursiv beschreibbar ist. Wenn auch die Bausteinoperatoren (Bausteinoperationen) eines Operandenflussplanes rekursiv beschreibbar sind, dann beschreibt auch der Operandenflussplan eine rekursive Funktion. Daraus folgt der

29 **Satz:** Eine USB-Funktion ist eine rekursive Funktion, falls sie aus rekursiven Bausteinoperationen komponiert ist.

Bisher wurde eine einzige rekursive elementare Bausteinoperation verwendet, die Inkrementierung. In Kap.9.2.1 werden wir als elementare Operationen sog. boolesche Operationen wählen und damit das Fundament für die Komponierung des universellen Rechners legen. Die Wahl boolescher Operationen ist bedingt durch die Beschränkung auf binäre Codierung.

Abschließend sei bemerkt, dass die Begriffe der *partiellen* und der *totalen* Funktion, die im Zusammenhang mit der Turingmaschine eingeführt wurden, übernommen werden. Eine **partiell-rekursive Funktion** ist eine rekursiv beschriebene Funktion, deren Funktionswerte nicht für alle Werte einer vorgegebenen Argumentwertemenge existieren.

8.4.6 Einschub: Rekursives Berechnen

Wir unterbrechen die Behandlung der algorithmischen Systeme, um einige Erläuterungen zum Begriff der Rekursion und einige Überlegungen zum rekursiven Berechnen einzuschieben.

30 Das Wort “rekursiv” (zurücklaufend) bringt den zirkulären Charakter der Iteration zum Ausdruck, der sich in der Rückkopplungsschleife in Bild 8.9 und darin widerspiegelt, dass die Funktion f in (8.9a) von sich selber abhängt. Das erste f in der Gleichung bezeichnet einen (sprachlichen) Operator, das zweite einen Operanden. Es liegt ein Operand-Operator-Zirkel vor (siehe Kap.6.3).

Wir wollen uns überlegen, wie man bei der Berechnung eines Funktionswertes gemäß (8.9) vorzugehen hat. Will man z.B. die fünfte Potenz von 1,2 berechnen, wird man durch wiederholtes Multiplizieren mit 1,2 zuerst $1{,}2^2$, dann $1{,}2^3$ u.s.w. berechnen. Bei diesem Vorgehen wird der Exponent n schrittweise inkrementiert, insofern ist die Berechnung *aufwärts* gerichtet. Demgegenüber ist die allgemeine Rekursions-

formel (8.9) und ebenso die spezielle Rekursionsformel (8.11) für das Potenzieren *abwärts* gerichtet, denn zur Berechnung von f_n wird auf f_{n-1} *zurück*gegriffen. Wollte man auf diese Weise $1{,}2^5$ berechnen, müsste man mit $n = 5$ beginnen, was sinnlos zu sein scheint.

Unter bestimmten Umständen ist das Abwärtsverfahren jedoch möglich, es kann die effektivere oder sogar einzige Möglichkeit sein. Hinsichtlich der Programmierungstechnik hat das Abwärtsverfahren den Vorteil, dass es häufig eine kompaktere Ausdrucksweise ermöglicht. Die Informatiker sprechen dann von **rekursiver Berechnung** und von **rekursivem Programmieren**. Das rekursive Berechnen soll am Beispiel der Fakultät demonstriert werden[16].

Das Produkt 1∗2∗3∗... ∗ n wird in der Mathematik abgekürzt $n!$ notiert und als "n Fakultät" gelesen. Das Ausrufungszeichen wird also als Operationssymbol verwendet. Die Fakultät-Funktion $f(n) = n!$ ist nur für ganze nichtnegative Zahlen definiert. Die folgenden Formeln sind zwei unterschiedliche Definitionen der Fakultät-Funktion:

$$f(n) = 1 * 2 * 3 * ... * n = n! \text{ oder} \tag{8.12}$$

$$f(n) = n * (n - 1)! = n!; \quad 0! = 1. \tag{8.13}$$

In beiden Definitionen ist das linke Gleichheitszeichen als Ergibtzeichen, das rechte als Benennungszeichen (Definitionszeichen) aufzufassen. Wie unschwer zu erkennen ist, lässt sich die Definition (8.12) ohne weiteres in einen imperativen Algorithmus überführen, der folgende Berechnungsschritte vorschreibt:

a := 1∗2
b := a∗3
c := b∗4
und so fort.

Es handelt sich um iteratives, nicht um rekursives Multiplizieren. Nach jedem Schritt muss geprüft werden, ob n erreicht ist.

Für die Definition (8.13) ist eine solche Überführung nicht möglich. Um sie als Operationsvorschrift benutzen zu können, muss man ihre Semantik, also das konkrete Vorgehen, das sie vorschreibt, verstehen. Die Semantik aber geht aus der Syntax der Formel nicht unmittelbar hervor. Sie ist durch "Vormachen" am anschaulichsten zu erklären. Dazu berechnen wir 5! rekursiv.

Nach (8.13) ist das Produkt 5 ∗ 4! zu berechnen. Soweit ist die Semantik klar. Doch wir können das Produkt nicht bilden, weil wir den Wert von 4! nicht kennen. Nun kommt die *interne Semantik* einer rekursiven Berechnung zum Tragen und es

31

16 In der Literatur wird die rekursive Beschreibung und Ausführung einer Operation häufig an einer bekannten Denksportaufgabe erläutert, dem "Turm von Hanoi", siehe z.B. [Duden 89].

beginnt der typische Prozess, der bei der Interpretation (Ausführung) der Definition (8.13) im Täger (Gehirn oder Computer) abläuft.

Wir notieren uns die 5 und berechnen 4! gemäß (8.13), wobei genauso verfahren wird, d.h. die 4 wird notiert und 3! berechnet. Der Prozess wird fortgesetzt, bis die Dekrementierung den Wert 0 liefert, für den der Wert der Fakultät als bekannt vorausgesetzt wird; es gilt 0!=1. Nun kann n! berechnet werden, indem die abgespeicherten Werte gemäß (8.11) iterativ miteinander multipliziert werden. Das Notieren der Zahlen, die zu multiplizieren sind, mag übertrieben erscheinen. Ein Computer dagegen muss sich *alles* notieren (abspeichern), da er immer nur diejenigen Werte “im Auge hat”, mit denen er gerade eine Operation (z.B. eine Multiplikation) ausführt. Wenn jedoch die einzelnen Schritte kompliziertere Operationen beinhalten, kann auch bei rekursiven Rechnungen per Hand das Notieren zweckmäßig oder sogar notwendig sein.

In Bild 8.11 ist der wesentliche Teil eines Operatorennetzes dargestellt, das die beschriebene rekursive Berechnung der Fakultät durchführt. Der Operator op_1 führt die wiederholte Dekrementierung und op_2 die wiederholte Multiplikation aus. Das Netz zeichnet sich durch einen besonderen Speicher aus, einen sog. **Keller-** oder **Stapelspeicher**, auch **Stack** genannt (vom englischen stack = Stapel). In ihn können “von oben her” Daten eingespeichert (gekellert, gestapelt) und nach oben ausgelesen (**entkellert**) werden. Beim Kellern (Speichern) wird der bereits vorhandene Inhalt des Kellerspeichers um einen Speicherplatz (um ein Rechteck in Bild 8.11) nach unten verschoben. Beim Entkellern (Lesen) wird der Inhalt des obersten Platzes ausgegeben und der restliche Speicherinhalt um einen Platz nach oben verschoben.

In Bild 8.11 ist der Zustand des Kellerspeichers nach der zweiten Unterbrechung (Dekrementierung) bzw. vor der vorletzten Multiplikation dargestellt. Der Kellerspeicher kehrt die Reihenfolge, in der op_1 seine Resultate ausgibt, um, sodass op_2 die Zahlen in aufsteigender Reihenfolge multipliziert.

Um den Zusammenhang zwischen *rekursivem Berechnen* und *rekursiver Iteration* zu verdeutlichen, schreiben wir (8.13) um, indem wir die Fakultät-Funktion unspezifisch mit f und die Multiplikationsfunktion mit h bezeichnen. Dann geht (8.13) über in

$$f(n) = h(n, f(n\text{-}1)). \qquad (8.14)$$

Ein Vergleich mit (8.9) zeigt, dass (8.9) durch Streichung von x in (8.14) übergeht. Die Fakultät ist eine Iteration, in

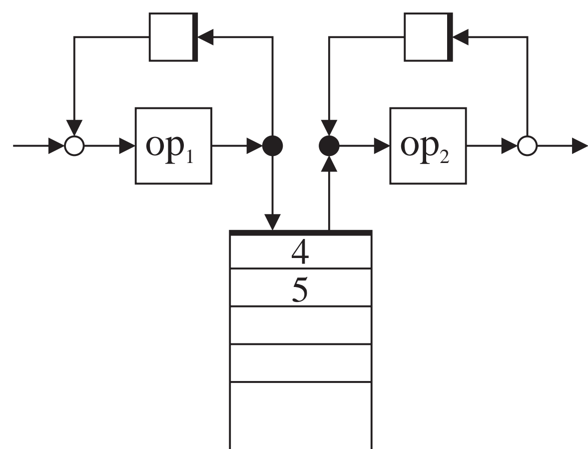


Bild 8.11 Operatorennetz zum rekursiven Berechnen mittels Kellerspeicher

die außer der Iterationszahl keine Variable eingeht.

Wie man sieht, ist rekursives Berechnen und rekursives Iterieren ein und dasselbe. Man beachte, dass sämtliche bisherigen Formeln für rekursives Berechnen oder Iterieren, also die Formeln (8.9), (8.10), (8.11), (8.13) und (8.14), funktional notiert sind. Das ist kein Zufall, sondern gilt offensichtlich allgemein. *Eine Vorschrift für eine rekursive Berechnung kann in Form einer Funktion, jedoch nicht in Form eines Befehls eines imperativen Programms notiert werden.* Um sie imperativ zu notieren, muss sie in eine Befehlsfolge zerlegt und dadurch de facto in eine gewöhnliche (nichtrekursive) Iteration überführt werden. Die Formeln (8.13) und (8.12) und der nachfolgende imperative Algorithmus zur Berechnung der Fakultät gaben dafür ein Beispiel. 32

Abschließend soll die Beziehung zwischen rekursivem Berechnen und einem in der theoretischen Informatik häufig verwendeten Begriff angedeutet werden, dem Begriff des *Kellerautomaten*. Dazu fassen wir die Operatoren op_1 und op_2 von Bild 8.11 gedanklich zu einem Operator zusammen (Operatorenabstraktion), dessen Ausgang auf den Eingang zurückgekoppelt ist. In der Rückkopplungsschleife liegt der Kellerspeicher, der auch die Funktion der Verzögerungsglieder von Bild 8.11 übernimmt. Von den konkreten Operationen des rekursiven Ab- bzw- Aufstiegs ist abstrahiert worden. Das resultierende Operatorennetz ähnelt demjenigen des abstrakten Automaten von Bild 8.5. Wir ersetzen nun die Sammelweiche im Eingang durch eine Vereinung, sodass der entstandene Kompositoperator - ebenso wie der abstrakte Automat - in jedem Takt einen neuen Eingabewert empfangen und verarbeiten kann. Schließlich verallgemeinern wir den Kompositoperator von Bild 8.11 dahingehend, dass f jede Funktion sein kann und der *Folgefunktion* des abstrakten Automaten entspricht. Der resultierende Kompositoperator heißt **Kellerautomat** (siehe Bild 8.12).

Der Kellerautomat zeichnet sich dadurch aus, dass er in jedem Takt seinen Zustand kellern kann, ohne die vorangehenden Zustände zu löschen, und dass sein neuer Zustand nicht nur vom letzten Zustand abhängen kann, sondern dass "in den Keller hinabgestiegen und Vergangenes heraufgeholt" werden kann. Die Folgefunktion f kann also von mehreren, im Prinzip von unbeschränkt vielen früheren Zuständen abhängen. In Kap.16.5 werden wir auf die Bedeutung des Kellerspeichers für die Theorie formaler Sprachen zu sprechen kommen.

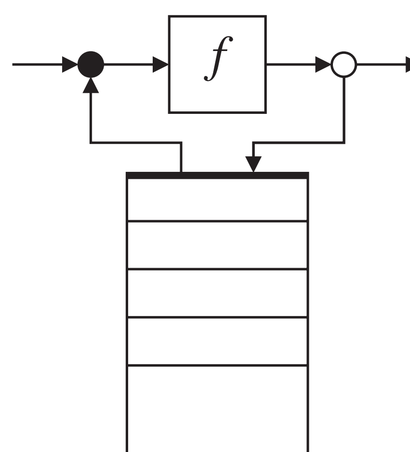


Bild 8.12 Kellerautomat; f Folgefunktion (vgl. Bild 8.5).

8.4.7 Lambda-Kalkül

Wir setzen nun die Besprechung universeller algorithmischer Systeme fort. In der Mitte der 30er Jahre entwickelte A.Church seinen λ-Kalkül (Lambda-Kalkül). Auch er hatte sich die Aufgabe gestellt, den intuitiven Begriff der berechenbaren Funktion zu objektivieren. Dazu hat er zunächst die Zweideutigkeit der funktionalen Notation ausgemerzt (man erinnere sich an die Bemerkung am Ende der Behandlung der funktionalen Iteration [25]) und kam auf diesem Wege zu einer universellen Methode, neue Funktionen aus gegebenen Funktionen auf formalem Wege abzuleiten, sie zu "berechnen", wobei hier unter Berechnen *nicht* das Berechnen der Werte bereits definierter Funktionen zu verstehen ist, also *nicht* das *Ausführen* von Operationsvorschriften, sondern das Definieren selbst, das Artikulieren von Operationsvorschriften. In den bisher behandelten algorithmischen Systemen waren zwar Komponierungsregeln definiert, die Operationsvorschriften selbst waren jedoch verbal beschrieben. Church suchte nach einer *Methode zum* ***Berechnen****, zum* ***formalen Ableiten*** *funktionaler Operationsvorschriften im Rahmen eines Kalküls*.

Dieses Ziel lässt sich offenbar nur auf einer hohen Abstraktionsebene erreichen, auf der die Unterscheidung zwischen den Begriffen *Operator* und *Operand* nur noch relative Bedeutung hat (begriffliche Konvergenz). Nach der Intention von Church ist eine Operationsvorschrift, d.h. ein sprachlicher *Operator* gleichzeitig *Operand* desjenigen Operators, der ihn "*berechnet*", d.h. durch Rechnung artikuliert. Zu diesem Zweck entwickelte Church seinen Lambda-Kalkül.

33 *Der* ***Lambda-Kalkül*** *ist ein Kalkül zum Definieren von Funktionen durch* ***Berechnung*** *ihrer Zuordnungsvorschriften. Funktionen, die sich im Rahmen des Lambda-Kalküls definieren lassen, nennen wir* ***lambda-definierbare Funktionen*** *oder* ***Church-Funktionen***. Die zentrale Lösungsidee ist diese: *Notiere Funktionen (Operationen, Operatoren) funktional und komponiere Kompositfunktionen aus Bausteinfunktionen* ***allein durch Substitution***.

Funktionsdefinitionen nach dem Lambda-Kalkül beruhen also ebenso wie Funktionsdefinitionen nach der rekursiven Methode oder nach der USB-Methode auf der Rekursionsidee. Doch werden sie durch die Beschränkungen allein auf die Substitution (bei funktionaler Notation) *berechenbar* (*ableitbar* durch analytisches Rechnen). Nachträglich scheint diese Lösungsidee gar nicht so fern zu liegen. Denn auf ihr beruht die Ausdrucksstärke der funktionalen Notationsweise, wie sie in der Mathematik üblich ist. Außerdem stellt die Substitution das entscheidende Komponierungsmittel der rekursiven Definitionsmethode dar, denn auch die rekursive Iteration beruht auf Substitution (vgl. (8.9) und Bild 8.9). Und auch der Markovalgorithmus beruht auf Substitution.

Um das Vorgehen des Lambda-Kalküls verständlich zu machen, gehen wir wieder von unserer Funktion (8.1) aus. Zunächst vereinbaren wir eine leicht abgewandelte Form der funktionalen Notationsweise. Dazu schreiben wir die Funktion f_1 aus (8.1) in zwei verschiedenen funktionalen Notationen an:

$$f_1 = +(\mathrm{pot}(x,n),x) \tag{8.15}$$

34

$$f_1 = (+\ (\mathrm{pot}\ x\ n)\ x) \tag{8.16}$$

Die beiden Notationen unterscheiden sich durch eine geringfügige syntaktische Änderung voneinander. In (8.16) ist die öffnende Klammer dem Operationszeichen (dem Funktionsbezeichner) nicht wie üblich *nach*gestellt, sondern *voran*gestellt. Anstelle von $f(x)$ wird also $(f\ x)$ notiert, wobei f der Funktionsbezeichner ist. In diesem Sinne unterscheiden wir zwischen **Präfixnotation** (8.15) und **Listennotation** (8.16). Beide Notationsarten kommen in der Literatur und in Programmiersprachen zur Anwendung. In diesem Kapitel werden wir die Listennotation bevorzugen.

Damit die Ausdrücke (8.15) und (8.16) *ausgewertet* werden können (als Operationsvorschriften dienen können) muss das Addieren und Potenzieren ebenfalls im Rahmen des Lambda-Kalküls durch Substitution definiert werden. Das ist möglich, indem sie mittels rekursiver Iteration festgelegt werden. Das Resultat ist zwangsläufig funktional notiert.

Nehmen wir an, die Definition der Funktionen $(+\ x\ y)$ und $(\mathrm{pot}\ x\ n)$ sei erfolgt. Die Bezeichnung der Argumente ist beliebig (ihre Codierung ist arbiträr). Dann stehen wir vor folgendem Problem. Wie kann man angeben, dass das erste Argument in der Additionsfunktion $(+\ x\ y)$ durch $(\mathrm{pot}\ x\ n)$ und das zweite durch x zu substituieren ist? Wegen der Kommutativität der Addition ist diese eindeutige Zuordnung zwar nicht notwendig, doch soll die Methode auch auf nichtkommutative Operationen anwendbar sein. (Man beachte, dass nicht das Ergebnis der Substitution, sondern eine Vorschrift für die Substitution gesucht ist.) Die Zuordnung zwischen den Variablen und den substituierenden Funktionen ließe sich durch die Ergibtgleichungen $x := (\mathrm{pot}\ x\ n)$ und $y := x$ festlegen. Das aber ist keine funktionale, sondern eine imperative Notation, also nicht zugelassen.

Church hat eine Methode entwickelt, nach der Substitutionen *funktional* und trotzdem exakt als Klammerausdrücke notiert werden. Sie soll in Anlehnung an die Bezeichnungsweise von Church erläutert werden. Wir notieren einen zunächst unverständlichen Ausdruck und erklären anschließend, was er bedeutet.

$$(\lambda x.\ +\ x\ 3)2 =: (+\ 2\ 3) =: 5. \tag{8.17}$$

35

Die Zeichenfolge ($\lambda x.$ kennzeichnet x als das *zu substituierende* Zeichen in dem nachfolgenden Ausdruck, der zwischen dem Punkt und der schließenden Klammer steht und als "x+3" zu lesen ist. Man sagt auch, dass x durch $\lambda x.$ in dem Ausdruck x+3 *gebunden* wird, und nennt x **gebundene Variable**. Nichtgebundene Variablen heißen **freie Variablen**. Die Ziffer 2 nach der schließenden Klammer gibt an, dass die gebundene Variable durch 2 zu substituieren ist. Bis zu dieser 2 einschließlich handelt es sich um einen Ausdruck des Lambda-Kalküls.

Die umgekehrten Ergibtzeichen gehören nicht zum Alphabet des Kalküls, sondern veranschaulichen die beiden Schritte *Substitution* und *Wertberechnung*, die bei der Interpretation (Ausführung) des gesamten vorangehenden Ausdrucks durchzuführen

sind. Auf das erste Ergibtzeichen folgt das Ergebnis der Substitution von x durch 3 und auf das zweite das Ergebnis der Wertberechnung. Beide Operationen gemeinsam, die Substitution und die Wertberechnung, werden **Auswertung** oder **Evaluierung** genannt, die erste der beiden Operationen, die das λ zum Verschwinden bringt, wird auch als **Lambda-Eliminierung** bezeichnet.

Wir geben ein zweites Beispiel für einen Audruck des Lambda-Kalküls.

$$(\lambda y.(\lambda x. + x\ y)2)3 \tag{8.18}$$

Der Leser wird den Ausdruck sicher richtig interpretieren. Die Eliminierung des zweiten und anschließend des ersten λ ergibt (+ 2 3).

Der entscheidende Punkt ist nun, dass nach einer schließenden Klammer ein *beliebiger Ausdruck* stehen darf. Als Beispiel diene die Lösung des Problems, an dem wir oben gescheitert waren, als wir in der Additionsfunktion (+ x y) die Variable x durch (pot x n) substituieren wollten und dafür eine funktional notierte Vorschrift suchten. Jetzt kennen wir die Lösung:

$$(\lambda x. + x\ y)(\text{pot}\ x\ n).$$

Damit lautet die Definition (Bildungsvorschrift) der Funktion f_1

$$\lambda y.(\lambda x. + x\ y)(\text{pot}\ x\ n))x).$$

Mit der oben benutzten Wortverbindung "beliebiger Ausdruck" ist jeder im Sinne des Lambda-Kalküls *auswertbare* Ausdruck oder auch das Ergebnis jeder Auswertung gemeint. Jeder derartige Ausdruck wird **Term** genannt. Auch Variablen- und Konstantenbezeichner sind Terme. Damit nimmt die Substitutionsvorschrift folgende allgemeine Form an:

$$(\lambda x.\ U)V \tag{8.19}$$

Darin bezeichnen U und V Terme. (8.19) schreibt vor, dass x *überall, wo es in U auftritt*, durch V zu substituieren ist.

Der Ausdruck (8.19) kann als 3-stellige Funktion aufgefasst werden mit den Argumenten x, U und V. Doch ist (8.19) nicht in Listenform notiert. Eine Listennotation wäre beispielsweise

$$(\#\ x\ U\ V), \tag{8.20}$$

worin das Doppelkreuz als Bezeichner für die **Substitutionsfunktion** verwendet ist. In der Programmierungstechnik kommen noch andere Notationsweisen zur Anwendung. In der theoretischen Literatur hat sich die ursprüngliche Lambda-Notation erhalten, wobei i.Allg. nicht von Lambda-Funktion, sondern von **Lambda-Operator**

gesprochen wird. Das entspricht dem mathematischen Sprachgebrauch, wonach ein Operator einer Funktion eine Funktion zuordnet.

Damit der Lambda-Kalkül Aussicht hat, universell zu sein, muss er über ein Pendant zur Zweigeweiche verfügen, eine "*Alternativ-Funktion*" oder "*bedingte Funktion*". Wir führen sie als 3-stellige Funktion mit dem Bezeichner "if" ein: 36

$$(\text{if } P\ U\ V) \tag{8.21}$$

Darin bezeichnet P ein Prädikat und U und V Terme. Wir nennen die Funktion (8.21) **if-Funktion**. Der Ausdruck (8.21) ist folgendermaßen zu lesen: "Falls das Prädikat P erfüllt ist, substituiere die if-Funktion (den gesamten Klammerausdruck) durch U, andernfalls durch V". Es kommen auch andere Funktionsbezeichner zur Anwendung, in der funktionalen Programmiersprache Lisp beispielsweise "cond" (von condition). Die in Kap.20.2 benutzte Programmiersprache CommonLisp verwendet die Syntax von (8.21). Damit kann die in (8.1) definierte Funktion $f(x,n)$ als Lambda-Ausdruck notiert werden:

$$(\lambda y.(\lambda x. + x\ y)(\text{pot } x\ n))(\text{if}(\leq x\ 0)\ x\ sinx).$$

Die Frage liegt nahe, ob in (8.20) der Term x eine Variable sein muss oder ob auch er - in Analogie zu (8.21) - ein beliebiger Term sein darf. Wäre das erlaubt, würde der Ausdruck (# U V W) bedeuten, dass in dem "Kompositterm" V der "Bausteinterm" U durch den Term W zu substituieren ist. Das sieht der Lambda-Kalkül nicht vor, denn seine Grundidee und die Rolle des Lambda-Operators ist das Binden einer *Variablen* in einem Ausdruck. Demgegenüber wäre der Ausdruck (# U V W) im Rahmen eines Markovalgorithmus durchaus sinnvoll und würde die Substitution von V durch W in U beschreiben, die erfolgen kann, falls in der Regelliste des Algorithmus die Regel $V \rightarrow W$ vorhanden ist.

Trotz dieses Unterschiedes ist eine deutliche Analogie zwischen der "*Termmanipulation*" des Lambda-Kalüls und der "*Wortmanipulation*" des Markovalgorithmus erkennbar. Das Wort "*Manipulation*" bringt zum Ausdruck, dass Zeichenketten mittels *Substitution* in andere überführt werden. Diese Analogie animiert zu einem umfassenderen Vergleich des Lambda-Kalküls mit anderen Methoden der Funktionsdefinition. Ein auffallender *Unterschied* zum Markovalgorithmus liegt darin, dass dieser die anzuwendenden Substitutionsregeln *vorgibt*. Die Regeln sind *Bestandteil* des Algorithmus, ähnlich wie im Falle der universellen Turingmaschine, wo die anzuwendende Automatentafel Bestandteil der Anfangsbeschriftung des Bandes ist.

Demgegenüber werden die Substitutionsregeln im Lambda-Kalkül nach den Regeln des Kalküls selbst *produziert*; es werden sowohl die *zu ersetzenden*, als auch die *ersetzenden* Zeichenketten (Terme) produziert, und zwar auch wieder durch selbstproduzierte Regeln. Es fragt sich, wo dieser Selbstproduktionsprozess seinen Anfang nimmt. Die Antwort lautet: In der sehr abstrakten rekursiven Definition des Termbegriffs. Um dem Leser eine Vorstellung vom Abstraktionsniveau zu geben,

auf dem Church gedacht hat, soll die Definition des Terms ohne Kommentar angegeben werden (in Anlehnung an [Hermes 71], S.209):

Jede Variable x ist ein Term; x kommt in x frei vor; keine andere Variable kommt in x frei vor. Sind T_1 und T_2 Terme, so ist (T_1T_2) ein Term; in diesem Term kommt eine Variable x frei vor, wenn x in T_1 oder in T_2 frei vorkommt. Ist T ein Term und x eine beliebige Variable, so ist $\lambda x.$T ein Term, in dem eine Variable y frei vorkommt, wenn y in T frei vorkommt und von x verschieden ist.

Auf den ersten Blick überrascht es, dass diese Definition zusammen mit den Regeln der Term-Manipulation gemäß (8.19) und (8.21) und einigen fast selbstverständlichen weiteren Regeln (wie Reflexivität, Symmetrie und Transitivität der Gleichheitsbeziehung $T_1=T_2$) ausreicht, um sämtliche effektiv berechenbaren Funktionen zu definieren. Es ist z.B. nicht ohne weiteres zu verstehen, wie durch den Lambda-Kalkül arithmetische Funktionen erfasst werden.

Überhaupt scheint unklar zu sein, wie man von der abstrakten, auf Termmanipulation beruhenden Funktionsdefinition zu einer Definition in Form einer Abbildung $X \rightarrow Y$ gelangen kann, insbesondere einer Abbildung in der Menge der ganzen Zahlen. Hierin liegt ein auffallender Unterschied zur rekursiven Methode und zur Registermaschine (URM-Methode). In allen vorangehenden Beschreibungsmethoden war die Definition einer Funktion als Abbildung kein Problem, denn es wurden stets Werte ("ausge*wert*ete Terme" in der Sprache des Lambda-Kalküls) in Werte transformiert. Im Lambda-Kalkül ist das nicht der Fall. Ein Term ist abstrakter, intensionaler Natur.

Angesichts des hohen Abstraktionsniveaus des Lambda-Kalküls hat es den Anschein, als ließe der Kalkül eine extensionale Definition einer Funktion als eindeutige Abbildung $X \rightarrow Y$ gar nicht zu. Es gibt keine aufzählbaren Mengen von Operanden, mit denen gerechnet werden könnte, kein Alphabet, keine Zahlen. All das muss nachträglich mit Hilfe der Regeln des Kalküls deduziert werden. Wie aber lassen sich die ganzen Zahlen, von denen die Registermaschine und die rekursive Definitionsmethode ausgehen, im Lambda-Kalkül einführen, d.h. als *Funktionen ableiten*?

Church erkannte, dass die ganzen positiven Zahlen in seinem Kalkül bereits implizit existieren und zwar als Iterationszahlen. Er definierte **Iterationsfunktionen** $\underline{0}$, $\underline{1}$, $\underline{2}$, $\underline{3}$ u.s.f. Um ihre Bedeutung (interne Semantik) zu verstehen, erinnere man sich an die Definition der Iteration in Kap.8.4.5 als Wiederholung ein und derselben Substitution, darstellbar durch eine Kette identischer Operatoren und notierbar als geschachtelte Funktion $f(f(f....(x)...))$ (vgl. (8.7) und Bild 8.7a mit $f_1=f_2=f_3=f$) oder in Listennotation $(f\ (f...(f\ x)...))$. Die Iterierte irgendeiner Funktion definieren wir nun als Iterationsfunktion und bezeichnen sie mit der Iterationszahl, die wir unterstreichen, um die Zahl als Funktionsnamen kenntlich zu machen. Danach ist beispielsweise die dritte Iterierte einer Funktion mit dem Namen f als $\underline{3}(f)$ bzw. $(\underline{3}\ f)$ zu notieren und die n-te Iterierte als $\underline{\mathrm{n}}(f)$ bzw. $(\underline{\mathrm{n}}\ f)$.

Es lassen sich nun neue Funktionen, auch 2-stellige Funktionen definieren, die bei Anwendung auf Iterationsfunktionen wieder Iterationsfunktionen liefern, und

zwar so, dass sich die Bezeichner (Nominalzahlen) der resultierenden Iterationsfunktionen als Resultat arithmetischer Operationen (z.B. der Addition) mit den Bezeichnern (Nominalzahlen) derjenigen Iterationsfunktionen interpretieren lassen, auf welche die neu definierte Operation angewandt wurde. Das berechtigt dazu, die Nominalzahlen als natürliche Zahlen zu interpretieren. Auf diese Weise konnte Church die arithmetischen Funktionen im Rahmen des Lambda-Kalküls definieren. Es konnte bewiesen werden, dass die lambda-definierbaren Funktionen genau die rekursiv definierbaren Funktionen sind.

Church hat sich den hohen Abstraktionsgrad durch die Aufgabe, die er sich gestellt hat, selbst auferlegt. Die Folge ist die Relativierung der Unterscheidung zwischen Operatoren (Operationen, Funktionen), Operanden und Werten (Funktionswerten). Ein Term kann sowohl ein Operator als auch ein Operand als auch ein Wert sein. Die Generalisierung des Operator-, Operanden- und Wertbegriffs zu einem einzigen abstrakten Oberbegriff ist ein wesentlicher Grund dafür, dass die Idee des Lambda-Kalküls sich als fruchtbar für die softwaremäßige Realisierung künstlicher Intelligenz erwiesen hat. Denn auch für das menschliche Denken ist charakteristisch, dass ein Idem, mit dem es hantiert, gleichzeitig als Operator (Subjekt) und als Operand (Objekt) auftritt. Das gilt speziell für das *Ich*-Idem, denn der Mensch sieht sich selbst ständig sowohl als *aktiven* als auch als *passiven* Teil der Welt.

Mit dieser etwas spekulativen Bemerkung beenden wir die Besprechung des Lambda-Kalküls. Ausführlichere Darstellungen findet der Leser in der einschlägigen Literatur[17].

8.5* CHURCH‘sche These und Kalkültransformation

Jede der sechs behandelten Methoden der Algorithmenbeschreibung definiert eine Klasse von Funktionen. Da den Methoden sehr unterschiedliche Ideen zugrunde liegen, sollte man erwarten, dass sie unterschiedliche Funktionsklassen definieren. Überraschenderweise ist das jedoch nicht der Fall. Die Algorithmentheoretiker konnten beweisen, dass nicht nur die genannten, sondern überhaupt sämtliche vorgeschlagenen algorithmischen Systeme ein und dieselbe Klasse von Funktionen festlegen. Die Funktionen werden, unabhängig von der Definitionsmethode, **rekursive Funktionen** genannt.

Auf Grund dieser Tatsache formulierte A.CHURCH die nach ihm benannte **churchsche These:** *Die Klasse der effektiv berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.* Jedes universelle algorithmische System, das entwickelt wurde oder das möglicherweise entwickelt werden wird, legt ein und dieselbe Klasse von Funktionen fest, die der rekursiven Funktionen. Nach den

17 Genannt seien [Hermes 71], [Louden 94], [Penrose 95].

Überlegungen in Kap.8.4.1 kann diese Aussage nur eine Hypothese sein. Doch hat sie sich ausnahmslos bestätigt, sodass sie als "praktisch nachgewiesen" gilt und sogar als richtige Voraussetzung in Beweisen herangezogen wird.

Die Tatsache, dass die verschiedenen Methoden die gleiche Klasse von Funktionen definieren, ist schon insofern überraschend, als sie von scheinbar ganz unterschiedlichen Objekten (Wertemengen) ausgehen. Die Operanden der Turingmaschine und des Markovalgorithmus sind Wörter über einem beliebig vorgebbaren Alphabet, die Operanden der Registermaschine und der rekursiven Funktionen sind die ganzen, nichtnegativen Zahlen, und die Operanden des Lambda-Kalküls sind abstrakte Objekte, Terme genannt.

Man könnte sich damit zufrieden geben, dass der Beweis der Identität der durch die verschiedenen Algorithmischen Systeme festgelegten Funktionsklassen von Autoritäten erbracht worden ist. So einfach wollen wir es uns jedoch nicht machen. Wir könnten den Beweis dadurch erbringen, dass wir nachweisen, dass nicht nur die Klasse der rekursiven Funktionen, sondern auch die anderen Funktionsklassen mit der Klasse der USB-Funktionen identisch sind. Im Weiteren werden wir die Identitäten zwar nicht exakt mathematisch beweisen, wir werden sie aber logisch plausibel machen.

Wir beginnen mit einer näheren Betrachtung des erwähnten Umstandes, dass in den verschiedenen algorithmischen Systemen Objekte ganz unterschiedlicher Natur die Rolle von Operanden spielen. Es kann z.B. rätselhaft erscheinen, dass die rekursiv berechenbaren Funktionen und die URM-berechenbaren Funktionen für natürliche Zahlen definiert sind, während nach der churchschen These auch Funktionen reeller Zahlen rekursive Funktionen sein müssen, wenn sie nur irgendwie berechenbar sind, z.B. durch einen Computer. Der aber hantiert ausschließlich mit Bitketten. Wie ist es dann zu verstehen, dass er rekursive Funktionen von reellen Veränderlichen berechnen kann?

37 Die Lösung des Rätsels ist nicht schwer zu erraten. Sie liegt in der Arbitrarität des Codierens (s. Kap.6.1) Die übliche Codierung reeller Zahlen ist ihre Darstellung als gebrochene Dezimalzahl, also als Kette arabischer Ziffern, die ein Komma enthält. Man kann auf das Komma als zusätzliches Zeichen verzichten, wenn ein für allemal ein fester Platz vereinbart wird, an dem das gedachte Komma zu stehen hat (sog. *Festkommadarstellung*). Wenn beispielsweise festgelegt wird, dass das Komma vor der dritten Ziffer von rechts steht, ist die reelle Zahl 3,51 durch die "ganze Zahl" 3510 darzustellen. Die Anführungsstriche sollen andeuten, dass die Interpretation der *Ziffernkette* 3510 als *ganze Zahl* im vorliegenden Kontext eine falsche semantische Belegung darstellt.

Der Platz des Kommas kann auch als Zehnerpotenz angegeben werden, z.B. $351*10^{-2}$. Wenn die Zehnerpotenz als Ziffernkette codiert wird, was natürlich möglich ist, ergibt sich wiederum eine "ganze Zahl" als Codewort für eine relle Zahl (sog. *Gleitkommadarstellung*). Auf die Codierung des Vorzeichens, die wir bisher unterschlagen haben, kommen wir sogleich zurück.

Eine weitere mögliche Codierungsmethode ist das *Durchnummerieren*, also das Zuweisen natürlicher (ganzer positiver) Zahlen an die reellen Zahlen durch Abzählen, m.a.W. die *Benennung* reeller Zahlen mit *Nominalzahlen*. (Nominalzahlen sind Ziffern oder Ziffernketten, die als Namen fungieren.) Das ist stets möglich, wenn davon ausgegangen wird, dass die durchzunummerierenden Mengen endliche oder abzählbar unendliche Mengen sind. Das Durchnummerieren legt folgende Methode für die Darstellung des Vorzeichens nahe. Angenommen, es sollen N positive und ebenso viele negative reelle Zahlen dargestellt werden. Es ergibt sich eine eindeutige Codierung, wenn die negativen reellen Zahlen mit den ganzen Zahlen von 1 bis N und die positiven mit den Zahlen von N+1 bis 2N benannt werden.

Um zu der üblichen rechnerinternen Zahlendarstellung zu gelangen, müssen die Ziffernketten in Bitketten umcodiert werden. Das kann z.B. dadurch erfolgen, das die Ziffernketten als Dezimalzahlen interpretiert und in Dualzahlen überführt werden, z.B. 0→0, 1→1, 2→10, 10→1010 oder durch irgendeine andere ein-eindeutige (in beiden Richtungen eindeutige) Abbildung. Die Umcodierung muss nicht zahlenweise (wortweise), sie kann auch ziffernweise (zeichenweise) erfolgen, indem jeder Ziffer eine vierstellige Bitkette zugewiesen wird, z.B. 0→0000, 1→0001 und folglich 10→00010000. Diese Methode liefert zwar längere Codeworte, hat aber ihre technischen Vorteile.

In der Rechentechnik ist eine Codewortlänge von 8 Bit üblich. Sie reicht aus, um alle **alphanumerischen** Zeichen (das sind die Ziffern und die Buchstaben) und alle **Sonderzeichen** (das sind zusätzliche Zeichen wie z.B. das Leerzeichen und Klammern) zu codieren. Eine solche Kette heißt **Byte**.

Die Codierung durch Nominalzahlen (mit anderen Worten die Abbildung in die Menge der natürlichen Zahlen) ist eine universelle Methode. Sie wird **Gödelisierung** und die codierenden Zahlen werden **Gödelnummern** genannt. Sie lässt sich auf jede abzählbare Menge anwenden, z.B. auf die Buchstaben eines beliebigen Alphabets oder auf die Wörter und Sätze jeder natürlichen oder künstlichen Sprache. Das bedeutet z.B., dass sich jedes Buch, ein Telefonbuch, eine Formelsammlung oder ein Roman, mittels natürlicher Zahlen codieren lässt. Es bedeutet aber auch, dass sich jeder Operand und die Wertetafel jeder Funktion mittels natürlicher Zahlen darstellen lässt.

Der letzte Satz wirft eine interessante Frage auf. Bedeutet die Umcodierbarkeit der Wertetafeln von Funktionen (Operatoren), dass die Funktionen (Operatoren) selber sich beliebig *umcodieren* lassen, z.B. mittels Nummerierung? Die Frage ist hinsichtlich der Operations*namen* offensichtlich zu bejahen, denn Namen sind arbiträr (Schall und Rauch). Aber was bedeutet es für die Operations*vorschriften*? Ein Beispiel soll die Sachlage illustrieren.

Viele Leser werden in der Schule gelernt haben, dass durch Logarithmieren eine Multiplikation in eine Addition, eine Division in eine Subtraktion und das Potenzieren in eine Multiplikation überführt wird. Es sei daran erinnert, dass der Dezimallog-

arithmus einer Zahl x, notiert als lg$x$, diejenige Zahl y ist, für die $10^y=x$ gilt. Daraus folgt beispielsweise, dass x_1*x_2 durch Logarithmieren in $\lg x_1+\lg x_2$ überführt wird:

$$\lg(x_1*x_2) = \lg x_1 + \lg x_2. \tag{8.22}$$

Das Resultat der Multiplikation ergibt sich durch Antilogarithmieren (die dem Logarithmieren inverse Operation) des Resultats der Addition.

Der Übergang zu Logarithmen führt zu einer ganz ähnlichen "Umcodierung" der Operanden, wie der Übergang von Festkommazahlen zu Gleitkommazahlen mittels Zehnerpotenzen, denn diese sind - neben den Mantissen - Bestandteil der Logarithmen. (Das Wort "Umcodierung" ist durchaus passend, denn beide "Codes" bezeichnen ein und dieselbe reelle Zahl.) Für die Umcodierung existiert eine *Vorschrift*, sodass der neue Code *berechnet* werden kann. Die Wahl der Umcodierungsvorschrift der Operanden (z.B. deren Logarithmieren) ist an sich beliebig. Doch ist mit der Wahl der Umcodierungsvorschrift für die Operanden gleichzeitig die neue Operationsvorschrift "ausgewählt", die sich hinter dem "umcodierten" Operationssymbol verbirgt. Operanden und Operatoren werden also gemeinsam umcodiert. Darum notieren wir die Umcodierung als mathematische Abbildung gemäß (8.23a). Das Ergebnis der Umcodierung ist durch einen Apostroph gekennzeichnet. Im Falle des Logarithmierens ist z.B. $100'$ identisch mit 2 (denn es gilt $10^2=100$) und $*'$ ist identisch mit +.

$$(Od,Op) \rightarrow (Od',Op') \tag{8.23a}$$

$$(X,Y,F) \rightarrow (X', Y', F') \tag{8.23b}$$

$$(Od,Alg) \rightarrow (Od',Alg') \tag{8.23c}$$

Auf Grund der vorausgesetzten Identität der Begriffe Operation und Funktion[18] kann (8.23a) in (8.23b) umgeschrieben werden. Darin bezeichnet F eine Menge von Funktionen und X bzw. Y die Menge der Argument- bzw. Funktionswerte. Die Unterscheidung zwischen Argument- und Funktionswerten in (8.23b) ist im Grunde überflüssig und eine Konzession an die gewohnte Notationsweise.

In (8.23c) sind die Funktionen durch die Algorithmen ersetzt, nach denen die Funktionen berechnet werden. Die Abbildung (8.23c) kann entweder als Übergang von einem algorithmischen System in ein anderes oder als **Übersetzung** aus einer algorithmischen Sprache in eine andere aufgefasst werden. Mit *Alg* ist die Menge aller Agorithmen bezeichnet, die in dem betreffenden algorithmischen System, d.h. in der algorithmischen Sprache des Systems artikuliert werden können.

Inhaltlich sind die Zusammenfassungen (*Od,Op*), (*F,X,Y*) und (*Od,Alg*) einander äquivalent, und die drei Abbildungen in (8.23) bezeichnen im Grunde ein und dieselbe Zuordnungsoperation. Wir nennen sie **Kalkültransformation**. Die Be-

18 Es sei daran erinnert, dass nur *eindeutige* Operationen und *realisierbare* Funktionen betrachtet werden, sodass die Worte *Operation* und *Funktion* als Synonyme verwendet werden dürfen.

zeichnung ist gerechtfertig, denn jeder der 6 Klammerausdrücke in (8.23) stellt eine Menge von Transformationsregeln (z.B. Op) und von Ausdrücken der jeweiligen formalen Sprache (z.B. Od) dar. Die Pfeile stellen also Transformationen zwischen Kalkülen dar. Wir ziehen das Wort *Transformation* den Wörtern *Umcodierung* und *Übersetzung* vor, weil es in diesem Zusammenhang dem üblichen Sprachgebrauch besser entspricht.

Aus der inhaltlichen Äquivalenz der drei Abbildungen in (8.23), bzw. der in ihnen zusammengefassten Einzelzuordungen, folgt die im ersten Augenblick sicher überraschende Feststellung, dass die Wörter bzw. Wortverbindungen *Kalkül*, *algorithmisches System* und *algorithmische Sprache* Synonyme sind. Tatsächlich haben sie auf einer Abstraktionsebene, an die man nicht unbedingt gewöhnt ist, ein und dieselbe Bedeutung. Auf dieser Abstraktionsebene ist auch das Wort Algebra ein Synonym mit den drei genannten Bezeichnungen. Ein Blick in die Literatur zeigt, dass die Zusammenfassung (*Od,Op*) als **Algebra**, zuweilen auch als **algebraische Struktur** bezeichnet wird.

Da die Bezeichnungen "Kalkül" und "Algorithmisches System" auf einem ausreichend hohen Abstraktionsniveau Synonyme sind, können statt der bisher benutzten Namen für die 6 behandelten algorithmischen Systeme mit gleichem Recht auch die folgenden Namen verwendet werden: Turing-*Kakül*, URM-*Kalkül*, Markov-*Kalkül*, rekursiver *Kalkül*, USB-*Kalkül* und Lambda-*Algorithmus*.

Historisch betrachtet haben sich die Begriffe Kalkül, Algebra, algorithmische Sprache und algorithmisches System bei der Untersuchung recht unterschiedlicher Fragestellungen herausgebildet und hatten infolgedessen zunächst unterschiedliche Bedeutungen. Doch diese näherten sich im Laufe eines langen Abstraktionsprozesses einander an, bis sie schließlich in einen einzigen Begriff zusammenflossen. Dieser Prozess der begrifflichen Annäherung soll **begriffliche Konvergenz** genannt werden. 38

Der Prozess der begrifflichen Konvergenz ist charakteristisch für das menschliche Denken überhaupt. Darum sprechen wir vom **Konvergenzprinzip des Denkens**. Es besteht darin, dass unterschiedliche Denkobjekte auf einer ausreichend hohen Abstraktionsebene zusammenfließen und miteinander identisch werden können. Wir werden dem Konvergenzprinzip im Weiteren wiederholt begegnen. Es wirkt im Sinne semantischer Verdichtung.

Die lange Entstehungsgeschichte der inhaltlich nahestehenden Begriffe Algebra, Kalkül, algorithmische Sprache und algorithmisches System hat zur Folge, dass jeder der Begriffe seinen speziellen Kontext hat, in dem er vorwiegend verwendet wird, und seine speziellen semantischen Nuancen. Das hat seinerseits zur Folge, dass nicht selten über ein und dasselbe Problem in unterschiedlichen Terminologien, in unterschiedlichen "Sprachen" gesprochen wird. Um die Beziehungen zwischen den verwendeten Terminologien deutlicher zu erkennen, vergegenwärtigen wir uns noch einmal einige Definitionen. Dabei werden sich interessante und für unser Vorhaben, einen Computer zu bauen, wichtige Schlussfolgerungen ergeben.

In Kap.5.4. war der Kalkülbegriff folgendermaßen verbal bestimmt worden: *Ein* ***Kalkül*** *ist ein System von Regeln für das Deduzieren* (Schlussfolgern, Berechnen); *es besteht aus Syntaxregeln und Deduktionsregeln*. Formal wird ein Kalkül in zwei Schritten definiert. Im ersten Schritt wird eine *formale Sprache* mittels Syntaxregeln festgelegt. Die Syntaxregeln geben an, welche Zeichen - elementare Zeichen (Alphabetzeichen) und Kompositzeichen (Zeichenketten, Wörter, Sätze) - erlaubt sind (zur Sprache gehören). In einem zweiten Schritt werden den zunächst völlig sinnfreien Zeichen formale Bedeutungen (Semantik) zugeordnet. Dieser Schritt heißt **formale Interpretation**[19]. Durch Interpretation werden die Zeichen zu **Bezeichnern** von Operanden, Operatoren oder Komponierungsmitteln. Sätze der formalen Sprache werden zu Deduktionsregeln (Operationsvorschriften).

Durch die formale Interpretation wird die formale Sprache zur **Kalkülsprache**, d.h. zu derjenigen Sprache, in der jeder komponierbare Operator beschrieben werden kann, m.a.W., in der die Vorschrift zur Ausführung jeder im Rahmen des Kalküls komponierbaren Operation artikuliert werden kann. Im gängigen Sprachgebrauch der Algorithmentheorie heißt eine Kalkülsprache *algorithmische* Sprache, wenn sie die Reihenfolge der Ausführungen der Bausteinoperationen eindeutig festlegt. Diese Voraussetzung wird mehr und mehr aufgegeben. Darum ist es gerechtfertigt, die Bezeichnungen Kalkülsprache und algorithmische Sprache als Synonyme zu verwenden und zu sagen: *Eine* ***formale*** *Sprache wird durch formale Interpretation zu einer* ***algorithmischen*** *Sprache*. Die Semantik einer formal interpretierten Kalkülsprache (die “formalen Bedeutungen” der mit ihrer Hilfe artikulierbaren Wörter) war in Kap.5.4 als *formale Semantik* bezeichnet worden.

In Kap.8.4.1 hatten wir ein algorithmisches System (eine algorithmische Sprache) *universell* genannt, wenn in ihm (in ihr) für jede effektiv berechenbare Funktion ein Berechnungalgorithmus artikuliert werden kann. Gemäß der churchschen These darf in dieser Definition “jede effektiv berechenbare Funktion” durch “jede rekursive Funktion” ersetzt werden. In dem gleichen Sinne sprechen wir auch von **universellem Kalkül**. Die in Kap.8.4 betrachteten algorithmischen Systeme sind Beispiele für universelle Kalküle.

Eine Kalkültransformation, also eine Abbildung (8.23a), muss folgende Bedingung erfüllen: *Das Resultat der Ausführung einer transformierten Operation (bzw. das Resultat der Berechnung einer transformierten Funktion) muss die Transformierte des Resultats der Ausführung der ursprünglichen Operation (bzw. der Berechnung der ursprünglichen Funktion) sein*. In der üblichen funktionalen Präfixnotation nimmt dieser Satz folgende Form an:

$$(f(x))' = f'(x'). \tag{8.24a}$$

19 In Kap.5.4 war ein zweiter Interpretationsschritt hinzugefügt worden. In ihm werden die Objekte eines Kalküls als Objekte der Realität interpretiert, wodurch die *formale* Semantik mit einer *externen* Semantik verbunden wird. Die externe Interpretation wird hier außer Acht gelassen.

Sämtliche Zuordnungen, die in der Abbildung (8.23b) zusammengefasst sind, müssen (8.24a) erfüllen. Darin kann x ein Tupel sein. Wenn es z.B. ein Paar ist, notieren wir

$$(f(x_1,x_2))' = f'(x_1', x_2') \tag{8.24b}$$

Diese Bedingung ist im Falle des Logarithmierens erfüllt. Verwendet man statt des Funktionssymbols das Operatorsymbol und wendet die Infixnotation an, geht (8.24b) in

$$(x_1 \; op \; x_2)' = x_1' \; op' \; x_2' \tag{8.24c}$$

über. Durch Vergleich erkennt man, dass die Bedingung (8.24c) die Verallgemeinerung von (8.22) ist.

Da in jedem universellen algorithmischen System Funktionen ein und derselben Klasse berechnet werden, nämlich rekursive Funktionen und *nur* diese, kann (8.24a) beim Übergang von einem algorithmischen System in ein anderes für beliebige Funktionen durch geeignete Umcodierung erfüllt werden. Zwischen den beiden Systemen existiert also eine Abbildung (8.23b).

Diese Schlussfolgerung kann wegen der inhaltlichen Äquivalenz der Abbildungen in (8.23) auch folgendermaßen formuliert werden: *Universelle Kalküle lassen sich ineinander transformieren*. Da auch in nichtuniversellen Kalkülen gemäß der These von Church nur rekursive Funktionen berechnet werden können, wenn auch nicht sämtliche, ergibt sich der **Kalkültransformationssatz**: *Jeder Kalkül lässt sich in einen universellen Kalkül transformieren*. Dieser Satz wird beim Entwurf eines universellen Rechners eine wichtige Rolle spielen. 39

Das Kapitel soll mit einer Bemerkung zur Entscheidbarkeit von Kalkülen beendet werden. Gödel hat die Frage untersucht, ob sich ein allgemeines Verfahren angeben lässt, nach dem für jeden Satz eines Kalküls, z.B. der Arithmetik oder des Prädikatenkalküls, entschieden werden kann, ob der betreffende Satz wahr ist oder nicht (Problem der **formalen Entscheidbarkeit** mathematischer Systeme). Dabei hat er den Objekten, mit denen ein Kalkül hantiert (z.B. den arithmetischen Operationen), Nominalzahlen zugeordnet, er hat sie also durch natürliche Zahlen *codiert*. Dafür ist z.B. die oben erwähnte Methode der Durchnummerierung geeignet, vorausgesetzt die Eindeutigkeit ist gewährleitstet. Gödel hat sich für die Bestimmung der Codezahlen eine spezielle Vorschrift ausgedacht, die allen Forderungen der Eindeutigkeit genügt. Mit den Codezahlen kann man *rechnen* und so einen nichtarithmetischen Kalkül *arithmetisieren*. Auf diese Weise hat Gödel die Unentscheidbarkeit verschiedener Kalküle bewiesen, nachdem er die Unentscheidbarkeit der Arithmetik bewiesen hatte. Der *Unvollständigkeitssatz*, auf den in Kap.6.2 Bezug genommen wurde, ist wohl das berühmteste Resultat der gödelschen Arbeiten[20]. Wir sind wiederholt 40

20 Siehe z.B. [Gödel 31],[Hermes 71],[Hofstadter 85],[Schöning 95].

auf das Problem der Unentscheidbarkeit gestoßen, allerdings in anderen Zusammenhängen, in Kap.6.2 im Zusammenhang mit widersprüchlichen Zirkularitäten [6.1] und in Kap.8.3 im Zusammenhang mit Wahrsageprädikaten [21] und mit dem Halteproblem [22].

8.6 Vier Grundideen des elektronischen Rechnens

Wir wollen nun versuchen, aus den teilweise recht abstrakten Gedankengängen und Begriffsbildungen des Teils 1, insbesondere des Kapitels 8, konkrete Hinweise für die Teile 2 und 3 herauszulesen und *Richtlinien* für den Entwurf des "universellen Computers" und für die Realisierung künstlicher Intelligenz abzuleiten. An den
41 Anfang stellen wir eine sehr weitgesteckte, aber unscharfe Zielbestimmung. *Es soll ein Gerät entworfen werden, das in der Lage ist, jedes* ***folgerichtige Denken des gesunden Menschenverstandes*** *zu simulieren, sodass es als "Denkassistent" dienen kann*. Dies ist das "**Fernziel der KI-Forschung**".

Folgerichtiges Denken ist immer ein Deduzieren, also ein formales oder nichtformales Ableiten, es ist immer ein *regelbasiertes* Denken und verläuft im Rahmen und nach den Denkregeln irgendeines geeigneten *speziellen Denkkalküls*. Wenn der Computer jedes folgerichtige Denken simulieren soll, müssen sämtliche speziellen Denkkalküle realisiert werden. Angesichts der unterschiedlichen Ausprägungen menschlichen Denkens, m.a.W. angesichts der großen Zahl möglicher spezieller Denkkalküle ist es unmöglich, sie alle hardwaremäßig zu realisieren. Den Ausweg weist der *Kalkül-Transformationssatz*, nach dem sich jeder spezielle Kalkül in einen universellen Kalkül transformieren lässt. Daraus ergegben sich zwei Richtlinien für den Entwurf und den Bau von Computern.

Erste Richtlinie. Um einen Computer zu bauen, der beliebige *Rechnungen* ausführen kann, implementiere man einen *universellen* Kalkül und für jede spezielle formale Kalkülsprache (für jede spezielle "mathematische" Sprache) ein Übersetzerprogramm, das die spezielle Sprache in die Sprache des universellen Kalküls übersetzt.

Zweite Richtlinie. Um menschliches *Denken* zu simulieren, *kalkülisiere* man es und verfahre anschließend gemäß der ersten Richtlinie. Das *Kalkülisieren* besteht in erster Linie im Herausfinden und Formulieren derjenigen Regeln, nach welchen das Schlussfolgern stattfindet. Ob das immer möglich ist, bleibt im Augenblick dahingestellt. Auf eine schwerwiegende Konsequenz soll schon jetzt hingewiesen werden. Die zweite Richtlinie bindet die KI an das Kalkülisieren und damit an die Mathematik, denn das Erfinden von und das Hantieren mit Kalkülen ist Gegenstand der Mathematik, wie wir bereits in Kap.5.4 [5.13] festgestellt hatten. Da die KI selber keine mathematischen Methoden erfindet, kann sie sich nur in den Fußstapfen der Mathematik entwickeln. Nichtsdestoweniger leistet die KI (der Computer) einen

selbständigen Beitrag zur mathematischen Modellierung der Welt - ein überraschendes und faszinierendes Ergebnis der kulturellen Evolution (siehe Kap. 21.3 [21.3]).

Wenn es die beiden Richtlinien schon zu Beginn der Rechentechnik gegeben hätte, wäre die Entwicklung sicherlich schneller verlaufen. Heute lassen sie sich aus einer Analyse des "state of the art" ablesen. Für uns werden sie als Wegweiser dienen, wenn wir uns daranmachen, in Teil 2 den Computer und in Teil 3 die Methoden der KI nachzuerfinden. Verfolgt man aus heutiger Sicht den verschlungenen Weg, den die Entwicklung der elektronischen Rechentechnik gegangen ist, heben sich aus den unzähligen Ideen, die den Weg gebahnt haben, vier besonders fruchtbare und folgenreiche Ideen heraus. Wir nennen sie die **vier Grundideen des elektronischen Rechnens**.

- **Die erste Grundidee** ist die Wahl des booleschen Kalküls (der boolesche Algebra) als hardwaremäßig zu realisierenden Kalkül (Kap.9.2). 42
- **Die zweite Grundidee** ist die Realisierung der elementaren booleschen Operatoren mitHilfe elektrischer Schalter (Kap.10.1).
- **Die dritte Grundidee** ist die Programmierbarkeit von Bitkettenoperatoren (Kap.13.1).
- **Die vierte Grundidee** ist die Automatisierung des Übersetzens algorithmischer Sprachen in die Maschinensprache eines Computers (Kap.16.4).

Der Begriff der Maschinensprache wird in Kap.13.5.2 eingeführt. Wir geben hier schon eine vorläufige Definition. *Ein Programm, das ein Computer unmittelbar, d.h. ohne vorherige Übersetzung interpretieren (verstehen und ausführen) kann, heißt **Maschinenprogramm** des Computers. Eine Sprache speziell für die Artikulierung von Maschinenprogrammen heißt **Maschinensprache** des betreffenden Computers. Die Gesamtheit der syntaktischen Regeln einer Maschinensprache und der semantischen Regeln, nach denen ein Computer seine Maschinensprache interpretiert, bilden ein Kalkül. Wir nennen ihn **Maschinenkalkül**.*

Ein Maschinenprogramm wird dadurch interpretiert (ausgeführt), dass den Operanden *innere Zustände* des ausführenden Computers als *codierende Zustände* und den Operatoren *Zustandsänderungen* zugeordnet werden. Innere Zustände sind stabile elektromagnetische Zustände elektronischer Bauelemente. Die Interpretation eines Programms, d.h. die ihm zugeordnete Semantik, *ist die Wirkung* des Programms im Computer. Allgemein hatten wir die Wirkung, die eine Zeichenkette in einem informationellen System auslöst, *interne Semantik* genannt (Kap.5.4 [5.7]). Ist der Träger ein Computer, sprechen wir von *maschineninterner Semantik* oder kurz **Maschinensemantik** oder **Computersemantik**.

Wenn ein Computer über ein Programm verfügt, das die Übersetzung einer Sprache in die eigene Maschinensprache ausführt, sagt man, dass die Sprache auf dem Computer **implementiert** ist und dass der Computer die Sprache "versteht". Eine Sprache wird durch ihre Implementierung zu einer **Programmiersprache** des Computers. In ihr können Aufträge an den Computer formuliert werden.

Wenn beispielsweise die Sprache der Arithmetik implementiert ist, kann der Computer arithmetische Ausdrücke auswerten. Wenn die Formelsprache für chemische Verbindungen implementiert ist und auch die Regeln, nach denen Ausdrücke der Sprache transformiert werden (d.h. chemische Prozesse stattfinden), kann der Computer aus den Eigenschaften von Atomen mögliche Strukturformeln von Molekülen und aus diesen mögliche oder hypothetische chemische Reaktionen herleiten, wofür Kapitel 16.3 ein Beispiel bringt. Dort wird die chemische Formelsprache als Kalkülsprache, d.h. als algorithmische Sprache, verwendet. Das kann Verwunderung hervorrufen, wenn unter chemischer Formelsprache die "Umgangsfachsprache" der Chemiker verstanden wird. Diese muss erst zu einer algorithmischen Sprache "hochstilisiert", d.h. sie muss kalkülisiert werden, bevor sie implementiert werden kann.

Dem Problem der Mathematisierung natürlicher Sprachen haben sich Mathematiker vieler Jahrhunderte gewidmet (siehe Kap.11.1). Vor ihm stehen heute die Informatiker, insbesondere die "KI-Forscher". Eine sehr allgemeine Lösung stellt aus mathematischer Sicht der *Prädikatenkalkül* und aus programmierungstechnischer Sicht die *logische Programmierung* dar. Das Kernproblem liegt in der Anbindung der externen Semantik natürlichsprachlicher Ausdrücke an die interne Semantik maschinensprachlicher Ausdrücke. Dies ist eines der zentralen Probleme der künstlichen Intelligenz. Wir hatten es in Kap.5.4 das *technische Semantikproblem* genannt (vgl.Kap.15.5).

Wenn ein Mensch dem Computer einen Auftrag oder eine Frage stellt, die er selber mit derjenigen Semantik belegt, in der er denkt (wir nennen sie **Nutzersemantik**; sie kann extern oder formal sein), dann muss die in die Maschinensprache übersetzte Frage im Computer genau diejenige Informationsverarbeitung *bewirken*, die der Nutzer bewirken wollte, und der Nutzer muss die Ausgabe (die Antwort) des
43 Computers interpretieren, d.h. ihr Nutzersemantik zuordnen können. Dabei muss sich die "*zu erwartende*" Antwort ergeben, also diejenige, die der Nutzer erhalten würde, wenn er sie selber ableiteten würden ohne dabei Fehler zumachen. In diesem Sinne sprechen wir von **Erhaltung der Nutzersemantik**.

Es mag zweifelhaft erscheinen, dass es überhaupt möglich ist, die Nutzersemantik zu erhalten. Denn für den Computer existiert sie nicht. Die Zeichenketten, die der Computer verarbeitet, besitzen für ihn weder formale noch externe Semantik, die sich auf die Außenwelt, die Welt des Nutzers bezieht. Der Computer "denkt" nicht in der Semantik, in welcher der Nutzer denkt. Insofern tragen die Zeichenketten, die der Computer empfängt, verarbeitet und ausgibt, nicht diejenige Bedeutung, die sie für den Nutzer tragen. Dies ist der Inhalt des *Bedeutungsprinzips*, das im Vorwort formuliert wurde. Nichtsdestoweniger muss die Bedeutung (die Nutzersemantik) erhalten bleiben.

Wir wollen das Problem der Semantikerhaltung etwas genauer analysieren. Dazu nehmen wir an, dass der Nutzer ein Physiker ist, der die Welt mit seinen Methoden modelliert und dabei auch den Computer nutzt. Der Modellierungsprozess, also das Ableiten neuer Aussagen über die Natur, erfolgt grob gesagt in 6 Schritten:

1. Messen,
2. Interpretation eines Kalküls durch die Messergebnisse,
3. Rechnen,
4. Programmieren,
5. Computerlauf,
6. Interpretation der Ausgaben des Computers.

Im zweiten Schritt wird externe Semantik an die formale Semantik eines Kalküls, sehr oft an die formale Semantik von Differenzialgleichungen angebunden. Diese muss bis zur Computerausgabe erhalten bleiben. Das bedeutet, dass in den Schritten 3 bis 5 keine formalen Fehler auftreten dürfen; der Physiker darf sich nicht verrechnen; mittels Kalkültransformationen muss die formale Semantik der Formeln (der Sprache des Physikers) in die formale Semantik der verwendeten Programmiersprache und weiter in die formale Semantik der Maschinensprache und in die interne Semantik des Computers fehlerfrei überführt werden; und schließlich darf der Computer sich nicht verrechnen. Im letzten Schritt wird an die Computerausgaben externe Semantik angebunden.

Für die Semantikerhaltung in den Schritten 1, 2, 3 und 6 ist der Physiker (allgemein der Computernutzer) verantwortlich. Für die Erhaltung der formalen Semantik in den Schritten 4 und 5 sind die Computerbauer, die Sprachentwickler und die Programmierer verantwortlich, die das Nutzerprogramm und das Übersetzerprogramm schreiben. In Kapitel 15 wird auf die Semantikerhaltung beim Übergang von mathematischen Formeln zur Maschinensprache und in Kapitel 13 beim Übergang von der Maschinensprache zu den Prozessen im Computer, d.h. zur Computersemantik eingengangen. Dabei müssen sprachlichen Ausdrücken elektronische Bauelemente bzw. deren Zustände zugeordnet werden, m.a.W. Softwareelemente müssen als Hardwareelemente *interpretiert* werden.

Viel Sorge und Mühe kann die Erhaltung der formalen Semantik beim Programmieren bereiten. Zur Sicherung der Semantikerhaltung kann ein altes, für andere Ziele entwickeltes Mittel herangezogen werden, die *Axiomatisierung*. Sie dient der Festlegung eines Kalküls durch ein sogenanntes Axiomensystem. Dieser Begriff war in Kap.5.4 eingeführt worden. Die Definition soll noch einmal wiederholt werden, wobei das Wort “*Regel*” durch “*Aussage*” ersetzt wird. *Das **Axiomensystem** eines Kalküls ist eine Menge von Aussagen, die voneinander unabhängig sind (d.h. keine Aussage des Axiomensystems ist aus den anderen ableitbar), die sich gegenseitig nicht widersprechen und aus denen sich sämtliche wahren Aussagen des Kalküls ableiten lassen.*

Das Aufstellen eines Axiomensystems für die Aussagen eines Wissensbereiches oder einer Sprache wird **Axiomatisierung** genannt. In Kap.5.4 war die Axiomatisierung der Geometrie durch EUKLID und der Mechanik durch NEWTON erwähnt worden. Wir werden auf die axiomatische Methode nicht näher eingehen und beschränken uns auf folgende Bemerkungen.

Eine Sprache heißt axiomatisiert, wenn ein Axiomensystem existiert, aus dem sämtliche Ausdrücke der Sprache und ihre formale Semantik abgeleitet werden können. Das bedeutet, dass nicht nur die Syntax, sondern auch die Semantik der Sprache formal definiert ist. Durch Axiomatisierung wird das Problem der Semantikerhaltung beim Übersetzen von Sprachen bzw. beim Transformieren von Kalkülen nicht aus der Welt geschafft, sondern auf das Transformieren (Übersetzen) zwischen Axiomensystemen reduziert, was erheblich übersichtlicher und infolgedessen weniger fehleranfällig ist.

In Teil 3 werden wir uns überlegen, ob bzw. wie das Fernziel der KI-Forschung erreicht werden kann. Dabei werden wir ständig auf die Begriffe, Methoden und Schlussfolgerungen dieses Kapitels zurückgreifen. Wir werden uns Methoden ausdenken, wie bestimmte Arten des Schlussfolgerns simuliert werden können, und uns so dem Fernziel nähern. Doch bleibt am Ende unseres Weges die Frage, ob das Fernziel erreichbar ist, offen. Die Etappen des Weges wurden durch obige Zuordnung der Grundideen des elektronischen Rechnens zu den einzelnen Kapiteln bereits angedeutet. Die Beziehungen zwischen dem Kapitel 8 und den folgenden Kapiteln sollen noch etwas detaillierter aufgezeigt werden.

Die Formulierung der vier Grundideen und die Angabe der Kapitel, in denen sie realisiert werden, hat beim Leser vielleicht das Verständnis dafür geweckt oder vertieft, warum in Kap.8.1 ein neues algorithmisches System, der USB-Kalkül eingeführt worden ist, obwohl es in theoretischer Hinsicht nichts Neues bringt. Die USB-Methode wird uns helfen, den universellen Rechner Schritt für Schritt durch hierarchisches Komponieren zu konzipieren. Dafür ist die Methode der uniformen Systembeschreibung von ihrer Idee her prädestiniert, denn wir haben sie ausdrücklich mit dem Ziel entwickelt, beliebige hierarchisch strukturierte Systeme adäquat beschreiben zu können.

Die USB-Methode entspricht der besonderen Herangehensweise dieses Buches an die Probleme der Berechnung und der Berechenbarkeit von Funktionen. Im Unterschied zur algorithmentheoretischen Fragestellung suchen wir - es sei noch einmal betont - *nicht* nach einem *algorithmischen*, sondern nach einem *technischen* System, nach einem *Gerät*. Für seine Beschreibung ist die USB-Methode eingeführt worden. Auch Turing hat nach einem Gerät gesucht, aber mit der Nebenbedingung maximaler Einfachheit und ohne das Ziel, das Gerät zu bauen. Wir werden in Kap.9.1 die Erfüllung einer anderen Bedingung fordern: die *binär-statische Codierbarkeit*. Unter dieser Bedingung wird das Problem der Berechenbarkeit in einem neuen Lichte erscheinen.

Bei der Konzeption des gesuchten universellen technischen informationellen Systems werden wir systematisch nach der USB-Methode vorgehen. Es ist also nicht verwunderlich, wenn uns die Begriffe der USB-Methode im Weiteren ständig begleiten werden. Aber auch die anderen algorithmischen Systeme werden wir nicht aus dem Auge verlieren. Die Sprachelemente verschiedener Programmiersprachen

werden uns an das eine oder andere algorithmische System, an den einen oder anderen Kalkül erinnern.

Als Abschluss unserer Überlegungen über das Codieren und das Transformieren (Umcodieren) von Kalkülen sei eine kleine philosophische Spekulation erlaubt. Letzten Endes ist es die Arbitrarität des Codierens, der wir die Universalität des Computers verdanken. Die Übertragung dieses Umstandes auf das Codieren durch den Menschen, das dem Denken und Sprechen zugrunde liegt, provoziert die Behauptung: *Die Arbitrarität des Codierens macht das Modellieren der Welt und speziell das mathematische (semantisch objektivierte) Modellieren möglich*. Aus der Sicht der Kapitel 1 und 5.1 müsste man eher - etwas paradox - sagen: *Die Arbitrarität des Codierens hat sich entwickelt, um die Welt modellierbar und sogar mathematisch modellierbar zu machen* (im Sinne einer teleonomischen Interpretation der Evolution).

Bevor wir das “theoretische” Kapitel 8 abschließen, sei noch einmal wiederholt, dass sein Inhalt *keine* Einführung in die mathematische Theorie der Algorithmen und der Berechenbarkeit darstellt. Die angestellten Überlegungen erheben nicht den Anspruch mathematischer Exaktheit. Sie sind eher Plausibilitätsbetrachtungen. Um die Methode der uniformen Systembeschreibung streng mathematisch auszuformulieren, muss weitere Arbeit geleistet werden, die über die Zielstellung dieses Buches, den Computer nachzuerfinden, hinausgeht. Der Verwirklichung dieser Zielstellung wenden wir uns nun zu. Der Weg, den wir gehen werden, ist abgesteckt.

Teil 2

Vom Bit zur Maschinensprache

9 Grundlagen der Komponierung informationeller Operatoren

Zusammenfassung

Die traditionelle Rechentechnik arbeitet fast ausschließlich mit *binär-statischer Codierung*, d.h. mit Codierung mittels zweier Zeichen, deren codierende Zustände (die materiellen Zustände, welche die Zeichen darstellen), statisch stabile Zustände des Trägermediums sind. Die Beschränkung auf zwei elementare Zeichen bedeutet wegen der Arbitarität der Codierung keine Einschränkung der Allgemeinheit. Ein Zustand heißt statisch stabil, wenn er sich nicht mit der Zeit verändert, sondern nur in einen anderen Zustand überspringen kann. Er heißt dynamisch stabil, wenn er ein stabil periodischer oder stabil repetierender Zustand ist, d.h. ein Zustand, in dem sich eine endliche Folge von Zuständen ständig wiederholt.

Die *elementaren* (d.h. nicht weiter dekomponierbaren) Operatoren eines informationellen Operators mit binär-statischer Codierung sind zwangsläufig die *elementaren booleschen* Operatoren. Unterhalb der elementaren informationellen Operatoren liegt der Bereich der kausalkontinuierlichen Prozesse. Den Übergang aus dem kontinuierlichen Bereich in den Bereich der informationellen, d.h. kausaldiskreten Prozesse leisten *Schwellenoperatoren*. Ohne sie ist auf dem Boden der klassischen Physik keine Informationsverarbeitung möglich. Doch sind sie nicht Bestandteil von traditionellen Computern, wohl aber von *Analog-digital-Konvertern*, die Computern in irgendeiner Form vorgeschaltet sein müssen. Bei Eingaben über die Tastatur durch den Nutzer fungiert dieser als Konverter. In Neurocomputern, die auf der Basis neuronaler Netze arbeiten, kommen Schwellenoperatoren in Form künstlicher Neuronen auch als *interne* Bausteinoperatoren zur Anwendung.

Um sämtliche booleschen Operationen komponieren und sämtliche booleschen Funktionen berechnen zu können, genügen einige wenige der insgesamt 16 elementaren booleschen Operatoren. Sehr häufig kommen der NOT-Operator (die Negation), der AND-Operator (die Konjunktion) und der OR-Operator (die Disjunktion) zum Einsatz. Jede Funktion mit endlicher Wertetafel kann im Prinzip (ohne Berücksichtigung des Aufwandsproblems) als *Kombinationsschaltung*, d.h. als zirkelfreies Netz elementarer boolescher Operatoren realisiert werden. Insbesondere kann sie auch als *disjunktive* Kombinationsschaltung realisiert werden. Jede als Kombinationsschaltung realisierbare Funktion ist eine rekursive Funktion.

Kombinationsschaltungen lassen sich zu zirkulären Netzen verbinden, wenn in jeden Zirkel (in jede Rückkopplungsschleife) ein Speicher mit vorgeschaltetem Tor eingebaut wird. Die Speicher erübrigen sich, wenn das Netz als Ganzes *Eigenzustände* besitzt, wenn es also als Ganzes einen Speicher darstellt. Das einfachste zirkuläre boolesche Netz, das sich als Speicher verwenden lässt, ist der *Flipflop* mit Eingangstoren. Er kann ein Bit speichern.

Informationelle Operatoren mit binär-statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und Speichern, wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegen muss. Das Entsprechende gilt für Neurocomputer mit statischer Codierung, wobei an die Stelle der Kombinationsschaltungen zirkelfreie Netze aus Schwellenoperatoren treten.

Für eine universelle Komponierungsmethode informationeller Operatoren gilt die *Vollständigkeitsforderung*: Sämtliche im Prinzip möglichen informationellen Operatoren mit statischer Codierung müssen komponierbar sein.

9.1 Statische und dynamische Codierung

Mit diesem Kapitel beginnen wir die Behandlung der zentralen Frage des Buches: Wie lässt sich Informationsverarbeitung technisch verwirklichen oder - im Sinne unserer Definition der Informatik - wie lässt sich ein technisches System aufbauen, das zur aktiven, sprachlichen Modellierung fähig ist?

Wir wollen das Gebiet der Computertechnik nicht nur als wissbegierige Interessenten betreten, die wissen wollen, "was es alles gibt" und "was man alles machen kann", sondern vor allem als Forscher und Erfinder, die sich überlegen, wie etwas funktioniert oder wie sich eine gewünschte Funktion realisieren lässt, z.B. das Addieren, das Gedächtnis oder das Schlussfolgern. In diesem Geiste werden wir die Gefilde der technischen Informationsverarbeitung erkunden und versuchen, ihre Ideen und Konzepte nachzuerfinden. Dabei werden wir vom Trägerprinzip ausgehen und konsequent die Methoden des hierarchischen Komponierens und der uniformen Systembeschreibung anwenden. Zunächst werden wir weniger Wert auf Systematik, dafür umso mehr Wert auf die gedankliche und logische Folgerichtigkeit der wesentlichen Ideen legen. In den Kapiteln 11 und 18 werden die Resultate unserer Gedankengänge in ihren systematischen und historischen Zusammenhang gestellt.

Im Sinne des Trägerprinzips geht die USB-Methode von der Tatsache aus, dass Informationsverarbeitung in einem stofflichen Träger abläuft. Demzufolge geht sie nicht, wie die Mathematik und die theoretische Informatik, von den Begriffen *Operation* und *Funktion* aus, die vom stofflichen Träger abstrahieren, sondern vom Begriff des *realen Operators*. Ebenso geht sie nicht von einem abstrakten Operandenbegriff, nicht von objektivierten *Idemen* aus, sondern von *Realemen*, von Zuständen des stofflichen Mediums, das die Operanden (die Zeichenketten) "*trägt*", in das die Zeichen in Form codierender Zustände *eingeprägt* sind.

Auf diese Weise werden wir ein Begriffsgebäude der Informatik "konstruieren", das auf einem *stofflich-physikalischen* Fundament steht, auf dem Fundament des *Naturnotwendigen*. Man kann ein solches Gebäude auch auf einem rein *logischen* Fundament aufbauen, auf dem Fundament des *Denknotwendigen*. Das hat den Vorteil, dass man weniger in philosophische, insbesondere erkenntnistheoretische

Fragen verwickelt wird. Der logische Weg wird beispielsweise in einem Buch mit dem Titel “Nichtphysikalische Grundlagen der Informationstechnik” [Wendt 89] beschritten. Die “kausale Informatik” des vorliegenden Buches stellt in gewissem Sinne ein Pendant oder eine Ergänzung des Buches von S.Wendt dar, indem sie die *nichtphysikalischen* Grundlagen *physikalisch* (kausal) untermauert.

Wenn wir mit dem Konstruieren, d.h. mit dem Komponieren von Operatorenhierarchien im Sinne des Trägerprinzips beginnen, müssen wir zuerst klären, welche Art von Zuständen sich als codierende Zustände eignen und aus welchen elementaren Operatoren Kompositoperatoren zu komponieren sind. Wir beginnen mit der ersten Frage und präzisieren sie:

Wie lassen sich Zeichen (elementare Zeichen und Kompositzeichen) *materialisieren, d.h. materiell codieren* (materiell instanzieren; vgl. Pfeil 3 in Bild 2.1), *sodass sie nicht verloren gehen, sondern solange erhalten bleiben (“gespeichert” sind), wie sie gebraucht werden*?

Die Antwort lautet: Die Codierung muss durch stabile Zustände eines geeigneten Trägermediums erfolgen, genauer gesagt, durch Zustandsparameter, die ausreichend stabil sind. Wir nennen sie **codierende Zustandsparameter** und die Zustände selber **codierende Zustände**.

Es sind zwei Arten von Stabilität zu unterscheiden, statische und dynamische. *Ein (makroskopischer) Zustandsparameter* (z.B. Lage, Temperatur, Magnetisierung, Ladung) *heißt **statisch stabil**, wenn er sich nicht mit der Zeit verändert. Er heißt **dynamisch stabil**, wenn er sich periodisch oder repetierend verändert*, m.a.W. wenn eine bestimmte zeitliche Folge von Parameterwerten sich ständig wiederholt. Dynamisch stabil (bezogen auf einen angemessenen Zeitmaßstab) ist z.B. die Planetenbewegung, die Rotation eines Kreisels und die Schwingung eines Pendels oder eines elektrischen Schwingkreises. Ein dynamisch stabiler Zustand, d.h. ein Zustand mit mindestens einem dynamisch stabilen Parameter, ist zirkulärer Natur. Mikroskopisch (statistisch oder quantenmechanisch[1]) betrachtet gibt es keine statischen, sondern nur dynamische Materiezustände. *Codierung, die statisch stabile bzw. dynamisch stabile codierende Zustände verwendet, heißt **satische** bzw. **dynamische Codierung**.*

In Kap.2.1 war von drei Evolutionen die Rede. Die dortigen Überlegungen lassen sich durch folgende Feststellung ergänzen: *Genetische und kulturelle Evolution beruhen auf **statischer**, die intellektuelle Evolution des Individuums offenbar auf **statischer und dynamischer** Codierung.* Das soll kurz erläutert werden.

Die *kulturelle Information*, also die Information, die im Verlauf der kulturellen Evolution gespeichert und weitergegeben wird, ist, soweit sie nicht mündlich überliefert wird, statisch codiert, zunächst in Bauwerken, Plastiken und Bildern, später

1 Alle physikalischen Überlegungen des Buches gehen von der klassischen Physik aus, d.h. es werden nur solche Prozesse betrachtet, in die relativ große Energien involviert sind, sodass das plancksche Wirkungsquantum vernachlässigt werden kann.

auch schriftlich, seit 500 Jahren vorzugsweise durch Bedrucken von Papier, neuerdings auch durch das “Beschreiben” von Datenträgern, z.B. von Kassettenbändern, Disketten oder CDs. Die codierenden Zustände sind Schwärzungs-, Magnetisierungs- oder andere Zustände des Datenträgers. Vorwiegend sind es Oberflächenzustände.

Die *genetische Information* ist bekanntlich in großen Molekülen, den DNS, codiert. Die codierenden Zustände sind statisch stabile Strukturen, die aus vier verschiedenen organischen Basen als Bausteinen komponiert sind. Jeder Baustein ist eine Ringverbindung aus Kohlenstoff-, Wasserstoff-, Sauerstoff- und Stickstoffatomen. Die Basen kann man als elementare Zeichen der genetischen “Schrift” auffassen und sagen, dass das genetische Alphabet 4 Buchstaben enthält.

Bei der Beschriftung der gängigen technischen Datenträger werden nur zwei Zeichen verwendet, es wird binär codiert. Das gilt generell für die interne Informationsdarstellung in Computern, m.a.W. für computerinterne Zeichenrealeme. Dieses Vorgehen war in Kap.5.6 [5.20] begründet worden. Das menschliche Gehirn scheint eine kombinierte statisch-dynamische Codierung zu verwenden. Die Ergebnisse der Gehirnforschung legen die Annahme nahe, dass strukturelle Zustände des Zentralnervensystems (z.B. Verknüpfungsmuster zwischen den Neuronen) der statischen Codierung dienen, während Anregungszustände des Nervensystems der dynamischen Codierung dienen. Wenn der experimentelle Befund, dass ein Neuron nur einen einzigen stabilen Zustand besitzt, seinen Ruhezustand, dann ist statische Codierung durch Anregungszustände neuronaler Netze unmöglich.

Für unsere weiteren Überlegungen treffen wir folgende Festlegung: *Die Hardware, die in den folgenden Kapiteln schrittweise aufgebaut wird, soll mit **binärer, statischer Codierung** arbeiten.* Die Beschränkung auf zwei elementare Zeichen bedeutet wegen der Arbitarität der Codierung keine Einschränkung der Allgemeinheit. Ob die Beschränkung auf statische Codierung eine Einschränkung bedeutet, werden wir später diskutieren.

9.2 Elementare informationelle Operatoren

9.2.1 Elementare boolesche Operatoren und die erste Grundidee des elektronischen Rechnens

Gemäß der vereinbarten Beschränkung auf binär-statische Codierung präzisiert sich unsere Aufgabe: *Es ist ein universelles informationelles System mit binär-statischer Codierung als Operatorenhierarchie zu entwerfen*. Das bedeutet, dass die Operanden *Bitketten* sind, auch *Binärwörter* genannt, und dass das System und seine Bausteinoperatoren Bitketten transformieren; darum nennen wir sie **Bitkettenoperatoren** oder **Binärwortoperatoren**.

Bei der Verwirklichung dieses Ziels werden wir uns folgender **Vollständigkeitsforderung** unterwerfen. *Keine Entwurfsentscheidung darf den Bereich der realisier-*

baren informationellen Operatoren mit binär-statischer Codierung einschränken, m.a.W. ***sämtliche im Prinzip möglichen informationellen Operatoren mit statischer Codierung müssen auf dem Wege, den wir gehen werden, komponierbar sein****.* Wenn wir uns beim Systementwurf an diese Bedingung halten, werden wir am Ende sagen können: Das entworfene System ist "universell" in folgendem Sinne. Das System kann jede mittels statischer Codierung realisierbare Funktion berechnen und es gibt keinen informationellen Operator mit statischer Codierung, der mehr oder etwas anderes zu leisten vermag, als das entworfene System. Auch die Turingmaschine - sie "arbeitet" mit statischer Codierung - kann nicht mehr leisten. *Eine Funktion, die von einem Operator berechnet werden kann, der mit statischer Codierung arbeitet, heißt* ***statisch berechenbar****.*

Der erste Schritt zur Lösung der Aufgabe ist die Festlegung der elementaren Operatoren. Der Allgemeingültigkeit halber, d.h. zum Zwecke der Vermeidung irgendwelcher Einschränkungen des Komponierens von Operatorenhierarchien, müssen die denkbar "elementarsten" Operatoren zugrunde gelegt werden, in die sich ein informationelles System dekomponieren lässt. Das bedeutet, dass ein elementarer Operator bei weiterer Dekomponierung den Charakter eines Operators, der Zeichen verarbeitet, verliert und dass die kausaldiskrete durch eine kausalkontinuierliche Beschreibung ersetzt werden muss.

Bevor wir die Suche nach den elementaren Operatoren informationeller Systeme beginnen, wollen wir, ausgehend vom Trägerprinzip, die gesuchten Operatoren genauer charakterisieren. Gesucht werden *reale Operatoren, die nicht weiter dekomponierbar sind und die ausschließlich den* ***Zuordnungsprozess*** *tragen*, d.h. den Prozess der Verarbeitung statisch codierter Operanden, die jedoch nicht die Operanden selber tragen (statisch codieren). Die gesuchten Operatoren besitzen keine stabilen Zustände, also kein Gedächtnis. Besäßen sie ein Gedächtnis, könnten sie in Zuordner und Speicher dekomponiert werden.

Zuordnungsprozesse in elementaren Operatoren nennen wir **Übergangsprozesse**. 2
Übergangsprozesse beginnen mit der Eingabe von Eingabeoperanden und enden mit der Ausgabe der zugeordneten Ausgabeoperanden. Aus der Sicht der kausaldiskreten Beschreibung verbindet ein Übergangsprozess zwei zeitlich benachbarte Ereignisse und überbrückt den nicht beschriebenen kausalkontinuierlichen Übergangsprozess. Der Übergangsprozess enthält keinen Zeitpunkt, der ein Ereignis der kausaldiskreten Beschreibung darstellt. Enthielte er einen solchen Zeitpunkt, könnte der Operator dekomponiert werden.

Die Schlussfolgerung unserer Überlegung lautet: *Die* ***elementaren Operatoren*** *informationeller Systeme mit statischer Codierung besitzen* ***kein Gedächtnis***; sie können "sich nichts merken". Der Ausgabeoperand (das Resultat) einer Operation liegt höchstens solange am Ausgang, wie der Eingabeoperand am Eingang liegt. Danach "wird er vergessen". Das kann durch Vorschalten eines Speichers verhindert werden. 3

Nach dieser Vorüberlegung beginnen wir die Suche nach konkreten Operatoren, die als elementare Operatoren verwendbar sind. Im Weiteren wird das einzelne Bit als spezielle Bitkette und ein Operator, der einem Bit ein Bit zuordnet (*Einbitoperator*) als spezieller Bitkettenoperator angesehen. Der Einbitoperator ist der denkbar einfachste Bitkettenoperator. Doch ist er als alleiniger elementarer Operator für das Komponieren von Kompositoperatoren nicht ausreichend. Man kann mit seiner Hilfe zwar Operatoren komponieren, die einzelne Bits in Bitketten transformieren, aber keine Operatoren, die Bitketten in Bits oder in Bitketten transformieren. Dafür sind Bausteinoperatoren mit mindestens zwei Eingängen (bzw. Bausteinoperatoren mit vorgeschalteter Vereinung) erforderlich.. Die einfachsten Operatoren, die diese Bedingung erfüllen, überführen Bitpaare in Bits. Derartige Operatoren zusammen mit den Einbitoperatoren heißen **elementare boolesche Operatoren**. Die Funktionen, die sie realisieren, heißen **elementare boolesche Funktionen**. Mit dieser Bezeichnung wird der englische Mathematiker George Boole (1815-1869) geehrt, der bei der Analyse logischer Gesetzmäßigkeiten durch Abstraktion zur Definition eines Kalküls gelangte, der nach ihm **boolesche Algebra** genannt wird (siehe Kapitel 11.1).

4 *Die elementaren Operatoren informationeller Systeme mit statischer Codierung sind also die **elementaren booleschen Operatoren***. Netze aus elementaren booleschen Operatoren nennen wir **boolesche Netze**. Wir werden uns zunächst nur für starre, zirkelfreie Netze interessieren. Solche Netze werden wir im Weiteren **Kombinationsschaltungen** nennen. Sie enthalten keine Weichen und keine Rückkopplungsschleifen, sondern nur starre Flussknoten (Gabeln und Vereinungen) und evtl. starre Maschen.

Kombinationsschaltungen, die ein einziges Bit als Ausgabeoperand liefern, heißen **boolesche Operatoren**. Die Funktionen, die sie realisieren, heißen **boolesche Funktionen**. Kombinationsschaltungen sind Bitkettenoperatoren (Binärwortoperatoren, informationelle Operatoren) der ersten Komponierungsebene, sofern sie unmittelbar (ohne Zwischenschritt) aus den elementaren booleschen Operatoren komponiert sind. Sie können aber auch als Operatoren der zweiten Komponierungsschicht aufgefasst werden, denn sie lassen sich in boolesche Operatoren dekomponieren, sodass jede Stelle der Ausgabebitkette durch je einen booleschen Operator berechnet wird. Die Komponierung und Untersuchung von Kombinationsschaltungen wird in Kap.9.3 mit mathematischer Exaktheit ausgeführt. Dort wird auch verständlich, wie es zu der vielleicht etwas merkwürdig anmutenden Bezeichnung gekommen ist.

Die Idee, als elementare Operatoren der technischen Informationsverarbeitung elementare boolesche Operatoren zu verwenden, m.a.W. einen universellen Rechner aus elementaren booleschen Operatoren zu komponieren, ist inhaltlich identisch mit der **ersten Grundidee des elektronischen Rechnens**, die darin besteht, die boolesche Algebra hardwaremäßig zu realisieren. Denn mit dem Aufbau einer Operatorenhierarchie aus elementaren booleschen Operatoren wird de facto die boolesche

Algebra realisiert. Wir haben diese “Idee” soeben aus dem Prinzip der binär-statischen Codierung “abgeleitet”.

Die erste Grundidee des elektronischen Rechnens scheint im Widerspruch zu der Aussage aus Kap.8.2.4 [8.13] zu stehen, dass ohne Schwellenoperatoren keine Informationsverarbeitung möglich ist, weil nur sie den Übergang aus dem nichtsprachlichen, kontinuierlichen Bereich in den sprachlichen, diskreten Bereich bewerkstelligen können. Demzufolge müssten Schwellenoperatoren die elementaren Operatoren informationeller Operatoren sein. Wir wollen dem Widerspruch nachgehen, obwohl wir damit den Weg zum *traditionellen* Computer, den wir entwerfen wollen und den wir später **Prozessorcomputer** nennen werden, verlassen und den Weg zum sogenannten **Neurocomputer** betreten. Wir werden also kurzzeitig die *traditionelle Informatik* verlassen und einen Abstecher in die *Neuroinformatik* machen.

Wir wollen versuchen, Kompositoperatoren aus Schwellenoperatoren zu komponieren. Dazu müssen zunächst **mehrstellige Schwellenoperatoren** (Schwellenoperatoren mit mehreren Eingängen) eingeführt werden, um Netze aufbauen zu können. Ein Schwellenoperator mit n Eingängen berechnet eine Funktion 5

$$y = f(x_1,x_2,...x_n), \tag{9.1}$$

worin die x_i reelle Größen sind, während y nur zwei Werte annehmen kann, die in Bild 4.1 mit 0 und 1 bezeichnet sind. Ob y den Wert 0 oder 1 annimmt, hängt davon ab, ob eine Funktion $g(x)$, die das Schwellenverhalten explizit beschreibt und **Schwellenfunktion** genannt wird, einen **Schwellenwert** s überschreitet oder nicht. Im einfachsten Fall kann die Schwellenfunktion eine Stufe sein, wie sie in Bild 4.1 dargestellt ist. Für eine n-stellige stufenförmige Schwellenfunktion gilt:

$$\begin{aligned} y &= 0, \text{ wenn } g(x_1,x_2,...x_n) \le s \\ y &= 1, \text{ wenn } g(x_1,x_2,...x_n) > s. \end{aligned} \tag{9.2}$$

Als Funktion g wird häufig die Summe der x_i verwendet. Unter Benutzung von ein- und mehrstelligen Schwellenoperatoren lassen sich Kompositoperatoren aufbauen. Doch scheint es wenig sinnvoll zu sein, Schwellenoperatoren hintereinander zu schalten, denn der Nachfolgeoperator empfängt von seinem Vorgänger bereits einen diskreten Wert, eine 0 oder eine 1. Wenn er mehrere Vorgänger hat, empfängt er - über eine Vereinung - ein Paar oder ein Tupel von Nullen und Einsen. Ein innerer Operator überführt also Bitketten in Bits, er ist ein *boolescher Operator*.

Ein Netz, dessen innere Operatoren boolesche Operatoren und dessen Eingangsoperatoren Schwellenoperatoren sind, arbeitet intern als Bitkettenoperator, dessen Eingabeoperanden reelle Wertetupel sind, sodass er mit der kontinuierlichen Umwelt kommunizieren kann. Die Schwellenoperatoren übernehmen die Funktion eines Analog-digital-Konverters.

Damit ist der Widerspruch ausgeräumt. Man könnte fragen, wo sich der Konverter verbirgt, wenn ein Mensch Daten in einen PC, also in einen reinen Binärwortoperator

eingibt? Die Antwort lautet: Der Mensch selber übernimmt die Funktion des Konverters, beispielsweise, wenn er die Tastatur des PC betätigt. Denn dabei wird die stetige (analoge) Bewegung seines Armes und seiner Finger in den binären Interncode des getasteten Zeichens überführt. (Genau genommen ist die Armbewegung eine kurze analoge Zwischenetappe zwischen der Codierung in Zentralnervensystem und der Codierung im Computer.) Früher erfolgte das Konvertieren durch Stanzen von Löchern in Karten oder Papierstreifen, ein Relikt aus der Zeit der Hollerithmaschinen, das inzwischen fast ausgestorben ist.

Hier unterbrechen wir den Abstecher in die Neuroinformatik, um ihn in den Kapiteln 9.2.2 und 9.4 fortzusetzen.

Man könnte auf die Idee kommen, einen "universellen" Computer dadurch zu bauen, dass man für jede Funktion, die er berechnen soll, eine Kombinationsschaltung entwirft und herstellt. In Kap.9.3 [9.3] wird sich zeigen, dass dies im Prinzip möglich ist. Doch kann man unschwer erkennen, dass sich die Idee nicht verwirklichen lässt. Nehmen wir an, wir wollten einen Computer bauen, der 3-stellige ganze positive Dezimalzahlen addiert. Die Eingabeoperandenpaare bestehen dann aus 6 Ziffern, die Ausgabeoperanden aus maximal 4 Ziffern. Bei ziffernweiser binärer Codierung durch 4-stellige Bitketten pro Ziffer (3-stellige Bitketten reichen nur für 8 der 10 arabischen Ziffern aus) beträgt die Länge des Eingabewortes 24 Bit, die des Ausgabewortes 16 Bit. Jedes der 16 Ausgabebits ist (im Prinzip) eine Funktion der 24 Eingabebits. Der Computer muss also 16 verschiedene 24-stellige boolesche Funktionen berechnen. Um uns zu überzeugen, dass unser Ziel auf diesem Wege nicht zu erreichen ist, schätzen wir ab, wie viele 10-stellige boolesche Funktionen es gibt (wir begnügen uns mit 10 Stellen, um die Abschätzung zu vereinfachen).

Zunächst fragen wir nach der Anzahl der Argumentwerte, d.h. nach der Anzahl der möglichen Eingabebitketten des zu realisierenden booleschen Operators. Da jede Stelle zwei Werte annehmen kann, beträgt die gesuchte Anzahl 2^{10}, wofür sich in der Informatik die Schreibweise 2^10 eingebürgert hat in Anpassung an die PC-Tastatur. Eine 10-stellige boolesche Funktion ordnet jedem dieser Argumentwerte entweder eine 0 oder eine 1 zu. Für die Menge aller Argumentwerte gibt es also 2^(2^10) verschiedene Abbildungen auf die Menge {0,1}, also gibt es ebenso viele Funktionen. Allgemein gilt:

$$\text{Anzahl } n\text{-stelliger boolescher Funktionen} = 2\text{\^{}}(2\text{\^{}}n). \tag{9.3}$$

Hinsichtlich Dualzahlen ist unser Gefühl für Größenordnungen i.Allg. wenig entwickelt. Darum wollen wir die Anzahl der 10-stelligen booleschen Funktionen näherungsweise als Zehnerpotenz angeben. Dafür nutzen wir den Umstand, dass 2^{10} gleich 1024, also etwa gleich 10^3 ist. Wir schreiben $2^{10} \approx 10^3$.

Wenn man beide Seiten dieser Näherungsgleichung in die (n/10)-te Potenz erhebt, d.h. die Exponenten auf beiden Seiten mit (n/10) multipliziert, ergibt sich (9.4a); wenn man beide Seiten mit n/3 multipliziert, ergibt sich (9.4b).

$$2^n \approx 10^{(3/10)n} \tag{9.4a}$$

$$10^n \approx 2^{(10/3)n}. \tag{9.4b}$$

Diese beiden Näherungsformeln können beim Übergang zwischen Dual- und Dezimalzahlen gute Dienste leisten. Die Anzahl der 10-stelligen booleschen Funktionen ist nach (9.3) gleich 2^(2^10) also etwa gleich 2^{1000}, was nach (9.4a) etwa gleich 10^{300} ist. Der Bau von 10^{300} realen Operatoren übersteigt die Möglichkeiten des Weltalls. Der Ausweg könnte in der Dekomponierung der booleschen Operatoren in elementare boolesche Operatoren gesucht werden, von denen es gemäß (9.3) nur 16 gibt. Das könnte zwar die Produktion verbilligen, das Aufwandsproblem würde jedoch nicht gelöst, weil die elementaren Operatoren in entsprechender Stückzahl hergestellt werden müssten. Der Ausweg aus dem Dilemma liegt im Bau *programmierbarer* Rechner, die durch Ausführung von Programmen Funktionen *berechnen*. Darauf wird später eingegangen.

Es könnte der Eindruck entstehen, dass unsere Überlegungen zur Komponierbarkeit *nichtprogrammierbarer* Computer aus Kombinationsschaltungen keinerlei praktische Bedeutung haben angesichts des hohen technischen Aufwandes. Dem ist jedoch nicht so. Denn in vielen Konsumgütern (Autos, Fotoapparaten, Waschmaschinen usw.) sind kleine "Computer" eingebaut, die Funktionen "berechnen", deren Wertetafeln ausreichend klein und zudem bekannt sind, z.B. in Form von Vorgaben, bei welcher Beleuchtungsstärke (d.h. bei welcher Ausgangsspannung des entsprechenden Sensors) welche Belichtungszeit einzustellen ist, oder bei welcher Temperatur die Heizung einer Waschmaschine abzuschalten ist. Diese Vorgaben werden als boolesche Funktionen codiert und als mikroelektronische Kombinationsschaltung "*implementiert*" (realisiert), häufig auf einem einzigen winzigen Siliziumplättchen, *Chip* genannt (siehe dazu Kap.12.3.4). Wenn die zu realisierende Funktion sehr komplex ist, kann es allerdings ökonomischer sein, einen programmierbaren Rechner einzusetzen.

7

Neben der Steuerung einfacher technischer Prozesse gibt es noch ein anderes weites Feld, wo Kombinationsschaltungen zum Einsatz kommen können, es ist der Bereich der "Information", hier im umgangssprachlichen Sinne von *Auskunft* verstanden. Auskunftstabellen wie Adressbücher, Telefonbücher oder Wörterbücher können als Wertetafeln von Funktionen (Abbildungen) aufgefasst, binär codiert und als Kombinationsschaltung realisiert werden. Die Eindeutigkeit geht nicht verloren, wenn die Auskunft mehrere Varianten enthält, wenn z.B. ein Wort in mehrere Wörter einer anderen Sprache übersetzt wird, denn die vollständige Auskunft stellt nach Umcodierung *eine* Bitkette dar. Selbstverständlich lassen sich auf diese Weise auch mathematische Tabellen (Logarithmentafeln, trigonometrische Tafeln u.ä.m.), also die *Wertetafeln* mathematischer Funktionen "in Hardware gießen", d.h. als Schaltung realisieren. Wenn die Funktion nur selten benutzt wird oder ihre Wertetafel sehr umfangreich ist, kann es ökonomischer sein, einen programmierbaren Rechner einzusetzen.

Die weite Verbreitung "*nichtprogrammierbarer* Computer" ist nicht der eigentliche Grund dafür, dass wir uns so ausführlich mit booleschen Funktionen beschäf-

tigen. Dieser liegt vielmehr darin, dass auch die Hardware der konventionellen *programmierbaren* Computer aus booleschen Operatoren aufgebaut ist. Damit sind wir gut motiviert, um zwei zentrale Probleme anzugehen, die Darstellung von Binärwortfunktionen durch elementare boolesche Funktionen (Kap.9.3), und die technische Realisierung dieser Funktionen (Kap.10.1). Zuvor wollen wir der im ersten Augenblick für wenig sinnvoll angesehenen Idee nachgehen, aus Schwellenoperatoren Netze aufzubauen.

9.2.2 Künstliche Neuronen

Im vorangehenden Kapitel wurde festgestellt, dass es sinnlos zu sein scheint, aus Schwellenoperatoren Netze aufzubauen, da die inneren Operatoren (die Operatoren ohne externe Eingänge) durch boolesche Operatoren ersetzt werden können, ohne dass sich das Verhalten des Netzes ändert. Das bedeutet nicht, dass sie ersetzt werden *müssen*. Auch Schwellenoperatoren können die Rolle innerer Operatoren übernehmen und zwar in Form sogenannter *künstlicher Neuronen* (s.u.). Hier liegt sogar die eigentliche Bedeutung von Schwellenoperatoren.

Netze aus künstlichen Neuronen, sogenannte *neuronale Netze*, sind dem biologischen Vorbild der neuronalen Strukturen im Gehirn nachgebildet. Sie zeigen booleschen Netzen gegenüber neue Eigenschaften, die sich daraus ergeben, dass die Eingabeoperanden *aller* Bausteinoperatoren, nicht nur derjenigen mit externen Eingängen, reelle Werte annehmen dürfen. Das bedeutet physikalisch, dass die codierenden Zustandsparameter sich auf dem Wege von einem Operator zum nächsten kontinuierlich ändern dürfen, denn der empfangende Operator akzeptiert jeden reellen Wert bzw. jedes reelle Wertetupel und ordnet ihm einen binären Wert zu. Ein derartiges Netz mit mehreren (vereinten) Ausgängen liefert eine Bitkette.

8 Freilich können auch auf den (vereinten) *Eingang* eines Netzes aus Schwellenoperatoren Bitketten gegeben werden. Dann haben wir es mit einem Bitkettenoperator (Binärwortoperator) zu tun. Da aber die internen Operanden auf dem Weg durch das Netz ihren Wert kontinuierlich ändern, arbeitet das Netz im Gegensatz zu einem booleschen Netz nicht rein digital, sondern sowohl analog als auch digital.

Welche Funktion ein neuronales Netz berechnet, hängt - ebenso wie im Falle boolescher Netze - von der Verbindungstopologie, der Struktur des Operandenflussgraphen ab, außerdem aber von der Veränderung der Operanden auf den internen Übergabewegen. Diese spezifische, aus booleschen Netzen unbekannte Abhängigkeit führt zu einer Eigenschaft, deren Bedeutung nicht sofort zu erkennen und kaum zu überschätzen ist. Sie lässt sich anhand eines Gedankenexperiments unschwer verstehen.

Gegeben sei ein zirkelfreies Netz elektronisch arbeitender Schwellenoperatoren, das eine bestimmte Binärwortfunktion berechnet. Als codierende Zustandsparameter sollen die Ein- und Ausgangsspannungen der Schwellenoperatoren dienen, die durch elektrische Leitungen miteinander verbunden sind. In die Leitungen fügen wir variable Widerstände ein, die es gestatten, die Spannungsabfälle in den Leitungen

und damit die Operandenwerte kontinuierlich zu variieren. Es stellt sich die Frage, wie die Verhaltensweise des Netzes auf Veränderungen der Widerstände reagiert. Folgende globale Antwort ist leicht zu geben. Solange die Veränderungen nicht dazu [9] führen, dass irgendeine Schwellenfunktion g (siehe (9.2)) den Schwellenwert s durchschreitet (sei es von oben her oder von unten her), ändert sich nichts. Sobald jedoch eine Schwelle infolge Widerstandsänderung über- oder unterschritten wird, ändert sich die Funktionsweise *sprunghaft*, d.h. die Binärwortfunktion, die das Netz [10] realisiert, geht in eine andere über. Wir haben es mit einem *digitalen* Operator zu tun, dessen Eingabe-Ausgabeverhalten mittels *analoger* Parameter gesteuert werden kann, entweder durch Veränderung der Widerstände oder durch Veränderung der Schwellen.

Wenn das schon merkwürdig erscheint, so ist der Effekt, der damit zu erreichen ist, frappierend. Er besteht in der Fähigkeit zum Lernen. Dem Netz kann eine gewünschte Verhaltensweise (eine bestimmte Funktion) "beigebracht" werden und zwar dadurch, dass die Widerstände solange in kleinen Schritten verändert werden, bis das Netz die gewünschte Funktion realisiert. Das Verändern kann in mehr oder weniger zielstrebigem *Probieren* bestehen.

Diese kurzen Andeutungen sind vielleicht wenig überzeugend, und dem Nichteingeweihten mag es unerfindlich erscheinen, dass es möglich und überhaupt sinnvoll ist, die Verhaltensweise eines Operators mit *digitalem* Ein-Ausgabeverhalten durch *analoge* Steuerung zu verändern. Tatsächlich wäre wohl kaum irgendjemand durch reines Nachdenken auf diese Idee gekommen, wenn es die Natur nicht vorgemacht hätte. Die Idee zum Bau von lernfähigen Netzen aus Schwellenoperatoren ist nicht am grünen Tisch entstanden, sondern der Struktur des Gehirns abgeschaut. Der Schwellenoperator zusammen mit den variablen Widerständen in seinen Eingängen stellt ein stark vereinfachtes Modell des biologischen *Neurons* mit seinen *Synapsen* dar, des Bausteins der "grauen Materie" des Gehirns.

In diesem sehr groben Modell entsprechen die variablen Widerstände den *Synapsen* eines Neurons, über die es mit anderen Neuronen verkoppelt ist. Die Funktion der Widerstände kann ebenso wie die der Synapsen als *Wichtung* der Eingabeleitungen bzw. der vorgeschalteten Neuronen aufgefasst werden. Je niedriger der Widerstand (je höher der Leitwert) ist, umso größer ist die Wirkung (das Gewicht) des betreffenden Vorgängerneurons, umso stärker ist die Kopplung. Darum ist es sinnfälliger, die Wirkung eines Neurons auf ein anderes Neuron nicht durch einen *Widerstand* zu charakterisieren, sondern durch einen *Leitwert* oder abstrakter durch ein *Gewicht*.

Für Schwellenoperatoren mit Eingabegewichten hat sich die Bezeichnung **künstliches Neuron** eingebürgert. Durch Vernetzung künstlicher Neuronen entsteht ein künstliches neuronales Netz. In der technischen Literatur wird das Adjektiv *künstlich* meistens unterdrückt und von Neuronen und **neuronalen Netzen** gesprochen. Doch können dadurch falsche Vorstellungen entstehen. Von den unendlich komplizierteren natürlichen Neuronen ist lediglich ihr Schwellenverhalten übernommen und von den

Verbindungen zwischen Neuronen über Synapsen lediglich die Variabilität ihrer Leitfähigkeit. Wir haben es mit einer "gewaltigen" (gewalttätigen) Idealisierung zu tun. Sie ist aus wissenschaftlicher Sicht legal, ähnlich wie Newtons Idealisierung der Erde zu einem Massepunkt. Aus suggestiver Sicht wäre es jedoch günstiger, das Wort *künstlich* hinzuzusetzen oder die Idealisierung auf andere Weise zu unterstreichen, indem man künstliche neuronale Netze beispielsweise als *neuromorphe* Netze bezeichnet. Dennoch schließen wir uns dem eingebürgerten Sprachgebrauch an und sprechen von neuronalen Netzen, auch wenn *künstliche* neuronale Netze gemeint sind.

Neuronale Netze eröffnen einen dritten Weg zur Realisierung bestimmter Funktionen (neben dem Bau von Kombinationsschaltungen und dem Programmieren programmierbarer Rechner): das **Lernen**. Wenn zur Realisierung einer Funktion die reine Hardwarelösung und auch die Formulierung eines Lösungsalgorithmus schwierig oder unmöglich ist, kann eventuell das *Erlernen* möglich sein. Das ist z.B. der Fall, wenn der Mensch dem Computer die Ausführung einer Operation übertragen will, die er selber zwar beherrscht, von der er aber nicht weiß, *wie* er dabei vorgeht, sodass er keine Operationsvorschrift angeben kann. Diese Sachlage ist charakteristisch für intuitive und assoziative Leistungen des Gehirns, z.B. für das *Erkennen*, etwa für das Erkennen eines Bekannten oder handschriftlicher Buchstaben. Von derartigen unverstandenen Operationen war in Kap.7.1 die Rede.

11

Damit wollen wir unseren Abstecher in die Neuroinformatik vorläufig abschließen, um ihn bei passender Gelegenheit fortzusetzen.

9.3* Ergänzung der formalen Methoden: Boolesche Algebra

Wir nehmen nun den Weg zum universellen Rechner wieder auf, den wir durch den Abstecher zu den neuronalen Netzen unterbrochen hatten. In Kap.8.5 hatten wie einen *universellen* Rechner dadurch gekennzeichnet, dass sich durch ihn jeder Kalkül realisieren lässt, d.h. dass sich die Operationen und Operanden jedes Kalküls in solche Operationen und Operanden abbilden lassen, die mit Hilfe des Computers komponierbar sind. Vorwegnehmend war erwähnt worden, dass die Realisierung eines Kalküls durch einen Computer in zwei Schritten erfolgt und zwar

1. in der Abbildung des zu realisierenden Kalküls in die boolesche Algebra und
2. in der Realisierung der booleschen Algebra als reale Operatorenhierarchie.

Wir waren zu dem Schluss gelangt, dass der erste Schritt infolge der Arbitrarität des Codierens und des Kalkültransformationssatzes [8.39] stets möglich ist. Er hat natürlich nur dann Sinn, wenn der zweite Schritt bereits getan ist, wenn also die

boolesche Algebra realisiert ist, d.h. in Form eines Gerätes, Computer genannt, real existiert.

Der zweite Schritt bildet demnach das *technische* Fundament der Rechentechnik. Er basiert seinerseits auf der booleschen Algebra als einem der wichtigsten *theoretischen* Fundamente der Rechentechnik. Ihr wenden wir uns nun zu, werden allerdings keine vollständige Einführung in dieses Gebiet der Mathematik geben. Vielmehr verfolgen wir zwei spezielle Ziele. Es soll erstens nachgewiesen werden, dass sich für jede boolesche Funktion ein zirkelfreies Netz aus elementaren booleschen Operatoren angegeben lässt, das die Funktion realisiert, und zweitens, dass boolesche Funktionen rekursive Funktionen sind. Es wird eine Methode entwickelt, die es gestattet, die Wertetafel jeder beliebigen booleschen Funktion in einen booleschen Ausdruck aus elementaren booleschen Funktionen zu überführen.

Wir beginnen ganz formal ("semantikfrei") damit, die elementaren, also die ein- und zweistelligen booleschen Funktionen (Operationen) einzuführen, indem wir allen möglichen Wertetafeln *Namen* und *Symbole* zuordnen. Bild 9.1 gibt eine Übersicht über die Wertetafeln (Spalte 2) und die entsprechenden gängigen Namen (Spalte 4) und Symbole (Spalte 5). Dass in einigen Fällen mehrere Symbole verwendet werden, erschwert das Lesen einschlägiger Texte; es ist die Folge der historischen Entwicklung und der kontextlichen Bedingungen in unterschiedlichen Anwendungsgebieten. Dass die Wörter *Funktion* und *Operation* als Synonyme verwendet werden, ist legal, da nur von realisierbaren Funktionen und eindeutigen Operationen die Rede ist [8.16]. Die beiden Argumente sind nicht wie bisher mit x_1 und x_2, sondern der besseren Lesbarkeit halber mit a und b bezeichnet

In der Kopfzeile der Spalte 2 sind die vier möglichen Argumentwertepaare angegeben, die beiden Werte eines Paares jeweils untereinander. Sie stellt (um 90 Grad gedreht) die Argumentwertespalte der Funktionstafeln der 16 elementaren booleschen Funktionen dar. Darunter, in den 16 Zeilen der zweiten Spalte, folgen alle denkbaren vierstelligen Bitfolgen. Die Bitfolge einer Zeile stellt (um 90 Grad gedreht) die Funktionswertespalte der Funktionstafel einer der 16 elementaren booleschen Funktionen dar. Die Kopfzeile der Spalte 2 bildet also zusammen je einer der darunter folgenden Bitketten die Funktionstafel einer elementaren booleschen Funktion. In Spalte 1 ist der allgemeine Funktionsbezeichner f mit derjenigen Zahl indiziert, die sich ergibt, wenn die betreffende Bitfolge in Spalte 2 als Dualzahl interpretiert und in eine Dezimalzahl überführt wird.

Die Funktionen/Operationen sind durch Spalte 2 *formal definiert*. Mehr bedarf es nicht, um mit dem "Rechnen in boolescher Algebra" zu beginnen. Der *Kalkül* der booleschen Algebra ist durch die Spalten 2 und 5 im wesentlichen festgelegt, abgesehen von einigen notwendigen zusätzlichen Festlegungen wie Klammerungs- oder Vorrangregeln und die Gesamtheit der erlaubten Zeichen (Alphabet). In Spalte 6 sind boolesche Ausdrücke für die jeweiligen Funktionen angegeben. Wenn zwei verschiedene Ausdrücke dieselbe Funktion beschreiben, müssen sie einander gleich sein. In diesem Fall steht in Spalte 6 eine Gleichung.

1	2	3	4	5	6
a b	0101 0011	f=1; wenn:	Name der Funktion/ Operation	Symbol, Operationscode	mögliche Boolesche Ausdrücke $f_K=\bar{f}_{15-K}$
f_0	0000	nie	Konstante 0		$a \wedge \bar{a} = \overline{\bar{a} \vee a}$, dgl. für b
f_1	0001	sowohl a als auch b	Konjunktion	^, &, AND	$a \wedge b = \overline{\bar{a} \vee \bar{b}}$
f_2	0010	nicht a, aber b			$\bar{a} \wedge b = \overline{a \vee \bar{b}}$
f_3	0011	b			$b = \bar{\bar{b}}$
f_4	0100	a, aber nicht b			$a \wedge \bar{b} = \overline{\bar{a} \vee b}$
f_5	0101	a			$a = \bar{\bar{a}}$
f_6	0110	entweder a oder b	Antivalenz, exklusives ODER	$\oplus$, $\not\Leftrightarrow$, XOR, EXOR	$(a \wedge \bar{b}) \vee (\bar{a} \wedge b)$
f_7	0111	a oder b	Disjunktion, inklusives ODER	$\vee$, OR, ≥1	$a \vee b = \overline{\bar{a} \wedge \bar{b}}$
f_8	1000	weder a noch b	Pierce´scher Pfeil	$\downarrow$, NOR	$\bar{a} \wedge \bar{b} = \overline{a \vee b}$
f_9	1001	a=b	Äquivalenz	$\Leftrightarrow$, EQ	$(\bar{a} \wedge \bar{b}) \vee (a \wedge b)$
f_{10}	1010	nicht a	Negation	$\neg$, ~, $\bar{\ }$, NOT	$\bar{a}$
f_{11}	1011	nicht a, oder aber b	Implikation	$a \Rightarrow b$	$\bar{a} \vee b = \overline{a \wedge \bar{b}}$
f_{12}	1100	nicht b	wie f_{10}	wie f_{10}	$\bar{b}$
f_{13}	1101	nicht b, oder aber a		$b \Leftarrow a$	$a \vee \bar{b} = \overline{\bar{a} \wedge b}$
f_{14}	1110	nicht gleichzeitig a und b	Scheffer´scher Strich	| , NAND	$\overline{a \wedge b} = \bar{a} \vee \bar{b}$
f_{15}	1111	immer	Konstante 1		$a \vee \bar{a} = \overline{\bar{a} \wedge a}$

Bild 9.1 Elementare boolesche Funktionen

Um tatsächlich allein aufgrund der Spalten 2 und 5 zu rechnen und z.B. die Ausdrücke und Gleichungen (Formeln) der Spalte 6 abzuleiten, bedarf es einer ausgeprägten abstrakt-mathematischen Veranlagung, die den wenigsten Menschen in die Wiege gelegt ist. Die meisten müssen den mühsamen Weg von der Semantik, in der sie zu denken gewohnt sind, zur formalen Semantik des Kalküls gehen. Für diesen Weg soll Spalte 3 eine Hilfestellung geben. Bevor wir sie nutzen, wollen wir uns die Spalte 2 noch etwas genauer ansehen.

Die Tabelle enthält nicht nur die echt 2-stelligen, sondern auch die vier möglichen 1-stelligen und die beiden 0-stelligen Funktionen. Die 1-stelligen Funktionen hängen nur von einer einzigen Variablen ab; das sind zum einen die Funktionen f_5 und f_{10}, die nur von a abhängen (sie ändern ihren Wert *nicht*, wenn b seinen Wert ändert), und zum anderen die Funktionen f_3 und f_{12}, die nur von b abhängen. Die Funktionen f_3 und f_5 sind die Identitätsfunktion, z.B. $f(a)=a$. Die Funktionen f_{10} und f_{12} sind die Negation, z.B. $f(a)=\overline{a}$ (Überstreichung bedeutet Negation). Die Funktionen f_0 und f_{15} hängen weder von a noch von b ab, es sind die Konstanten 0 und 1. Die Konstante 0 ist die Negation der 1 und umgekehrt. Für die Bitfolgen in Spalte 2 bedeutet dies, dass jedem Funktionswert in der Wertetafel von f_0 der entsprechende negierte Funktionswert aus der Tafel von f_{15} stehen muss, wie es tatsächlich der Fall ist. Formal notieren wir $\overline{f}_0 = f_{15}$. Auf die gleiche Weise erkennt man durch bitweisen Vergleich der entsprechenden Bitfolgen, dass $f_1 = \overline{f}_{14}$ und allgemein $f_i = \overline{f}_{15-i}$ gilt.

Die booleschen Ausdrücke in Spalte 6 stellen Kompositoperatoren aus den drei Bausteinoperatoren Negation, Konjunktion und Disjunktion dar. Mit Hilfe dieser drei elementaren Operatoren lässt sich also *jede* elementare Operation beschreiben (komponieren). Wir wollen versuchen, aus den Wertetafeln die Ausdrücke in Spalte 6 abzulesen. Dazu drücken wir die Bedingung dafür, dass die jeweilige Funktion gleich 1 wird, verbal aus. Das Ergebnis ist in Spalte 3 in abgekürzter Form angegeben. Das Vorgehen soll anhand der Konjunktion demonstriert werden. Aus der Wertetafel folgt, dass f_1 genau dann gleich 1 ist, wenn sowohl a als auch b gleich 1 sind. Für die formale Notation in Spalte 6 ist das erste Symbol in Spalte 5 verwendet worden. Der Operationscode AND könnte ein “Aha!” auslösen: AND ist verständlich, es steht für “sowohl...als auch”.

Das Aha-Erlebnis verstärkt sich, wenn man die boolesche Algebra durch die Aussagenlogik interpretiert, d.h. wenn man 0 und 1 als **falsch** und **wahr** und die Variablen als Platzhalter für Aussagen interpretiert, zum Beispiel: a für “Der Lichtschalter ist eingeschaltet”, b für “Die Glühbirne ist funktionstüchtig” und c für “Die Glühbirne leuchtet”; dann gilt $c = a \wedge b$. Das $\wedge$-Zeichen entspricht dem “und” in der verbalen Ausdrucksweise “Die Glühbirne leuchtet (genau dann), wenn sie funktionstüchtig ist *und* der Schalter eingeschaltet ist”. 12

Die angedeutete Interpretation der booleschen Algebra heißt **Aussagenkalkül**, **Aussagenlogik** oder **Aussagenalgebra**. In Kap.11.1 erfahren wir, dass die Aussagenalgebra der booleschen Algebra zeitlich vorausging. Der Weg von der Aussagenalgebra zur booleschen Algebra ist - wie die Entwicklung der Mathematik

überhaupt - der Weg der semantischen Objektivierung (vgl. Kap.5.4). Es ist der Weg der exakten Naturwissenschaften und der Weg jedes einzelnen Menschen, der sein Denken “mathematisiert”. Wir gehen auf die Aussagenalgebra nicht weiter ein und wenden uns wieder der abstrakteren booleschen Algebra zu.

Das bisher zu Bild 9.1 Gesagte soll noch einmal anhand der Funktion f_1 zusammengefasst werden. Die Funktion ist durch ihre Wertetafel gemäß Spalte 2 definiert, sie heißt Konjunktion und der Ausdruck auf der linken Seite der Gleichung in Spalte 6 ist die in der booleschen Algebra übliche Notation. Auf der rechten Seite der Gleichung steht gemäß der Regel $f_i = \overline{f}_{15\text{-}i}$ der Ausdruck für $\overline{f}_{14}$. Der Operationscode von $\overline{f}_{14}$ ist NAND, das Negierte AND. Die Negation des negierten AND ist wieder das AND.

Die analoge Prozedur für die Disjunktion liefert die Gleichung in Spalte 6 der f_7-Zeile. Auf die gleiche Weise ergeben sich alle Gleichungen der Spalte 6. Sie stellen *Formeln* der booleschen Algebra dar. Die Formeln in den Spalten für f_1 und f_7 heißen **morgansche Regeln**. Mit dem Symbol “¬” für die Negation lauten sie:

$$a \vee b = \neg(\neg a \wedge \neg b) \tag{9.5}$$

$$a \wedge b = \neg(\neg a \vee \neg b) \tag{9.6a}$$

Formel (9.6a) soll noch einmal in anderer Notation angeschrieben werden, wobei Konjunktionszeichen unterdrückt und Negations- bzw. Disjunktionszeichen durch `NOT` bzw. `OR` ersetzt werden:

$$ab = \texttt{NOT}(\texttt{NOT}a \ \texttt{OR} \ \texttt{NOT}b). \tag{9.6b}$$

In dieser Form werden wir im Weiteren die Gleichung in Kap.12 anwenden.

Die Richtigkeit der Gleichungen (9.5) und (9.6) kann - wie die Richtigkeit *jeder* Gleichung der booleschen Algebra - durch Nachrechnen geprüft werden, indem man für jedes Argumentwertepaar den Wert der rechten und linken Seite der Gleichung bestimmt. Oft lassen sie sich mit externer Semantik belegen, sodass sie “verständlich” werden, die morgansche Regel z.B. folgendermaßen: “Ich gehe (dann und nur dann) spazieren, wenn ich gesund bin und die Sonne scheint”, mit anderen Worten: “Ich gehe nicht spazieren, wenn ich nicht gesund bin oder wenn die Sonne nicht scheint”.

Geht man alle Zeilen der Tabelle von Bild 9.1 durch, erkennt man, dass Spalte 3 tatsächlich die Bedingungen dafür enthält, dass die jeweiligen Funktionen den Wert 1 annehmen. Die Bedingungen sind jedoch nicht vollständig (sie sind notwendig, aber nicht unbedingt hinreichend), sodass sie keine *exakte* Beschreibung oder gar Definition darstellen. Sie können nach folgender Vorschrift zu einer exakten Beschreibung der jeweiligen Funktion vervollständigt werden: “Schreibe die *vollständige* Bedingung dafür auf, dass f_i gleich 1 wird, notiere die Bedingung als booleschen Ausdruck mit Hilfe der Negation, Konjunktion und Disjunktion und setze diesen Ausdruck gleich f_i.” Das Wort *vollständig* bedeutet, dass *sämtliche* Argumentwerte-

paare anzugeben sind, für die f_i=1 gilt. Nur dann ist gesichert, dass für die übrigen Wertepaare f_i=0 gilt, dass der resultierende Ausdruck also tatsächlich die Wertetafel von f_i exakt beschreibt.

Das Vorgehen soll am Beispiel der Funktion f_6, der *Antivalenz*, demonstriert werden. Die Funktion f_6 ist gleich 1, wenn entweder a=1 und b=0 ist, wenn also $a \wedge \overline{b}$=1 gilt, oder wenn a=0 und b=1 ist, also $\overline{a} \wedge b$=1 gilt. Die vollständige Bedingung dafür, dass f_6 gleich 1 und nicht gleich 0 ist, kann also als Disjunktion zweier Konjunktionen notiert werden. Der resultierende Ausdruck stellt die Komponierung der Funktion f_6 mittels Negation, Konjunktion und Disjunktion dar:

$$f_6 = (a \wedge \overline{b}) \vee (\overline{a} \wedge b). \tag{9.7}$$

Dieser Ausdruck ist in Bild 9.1 für f_6 angegeben. Die Klammern können fortgelassen werden, da in der booleschen Algebra vereinbart wird, dass die Konjunktion den Vorrang vor der Disjunktion hat (in Analogie zum Vorrang der Multiplikation vor der Addition). Auf die gleiche Weise gelangt man zu dem Ausdruck für die Funktion f_8, die *Äquivalenz*. Die Äquivalenz ist gleich der negierten Antivalenz und umgekehrt (die entsprechenden Formeln sind in Spalte 6 nicht angegeben). Wie unschwer zu erkennen ist, erfüllen sämtliche Ausdrücke in Spalte 6 die oben genannte Vollständigkeitsbedingung. Spalte 6 demonstriert, dass sich jede elementare boolesche Funktion als Disjunktion, eventuell als Disjunktion von Konjunktionen darstellen lässt.

Die Methode ist offenbar nicht auf 2-stellige Funktionen beschränkt. Für jede beliebige boolesche Funktion lässt sich aus ihrer Wertetafel der entsprechende Ausdruck ablesen, d.h. eine Disjunktion aus Konjunktionen, wobei allerdings die 2-stellige Konjunktion bzw. Disjunktion eventuell auf eine n-stellige erweitert werden muss und zwar im Sinne des umgangssprachlichen “und..und..und..” bzw. “oder..oder..oder..”. In dieser Verallgemeinerung liefert eine *Konjunktion* den Wert 1, wenn sämtliche Argumentwerte gleich 1 sind, und eine *Disjunktion* liefert den Wert 1, wenn wenigstens ein Argumentwert gleich 1 ist. Offensichtlich lassen sich mehrstellige Konjunktionen in mehrere 2-stellige Konjunktionen dekomponieren. Das Entsprechende gilt für Disjunktionen. Damit ergibt sich der **Satz**: *Jede n-stellige boolesche Funktion lässt sich als Disjunktion n-stelliger Konjunktionen darstellen. Die Darstellung heißt* ***kanonische disjunktive Normalform*** (***KDNF***). Jede Konjunktion der KDNF enthält jede Argumentvariable, entweder negiert oder nichtnegiert.

Das Aufstellen einer KDNF nach obiger Vorschrift soll anhand zweier 3-stelliger Funktionen vorgeführt werden, deren Wertetafeln in Bild 9.2 in einer Tafel zusammengefasst sind. Wenn man sich an die Notationsweise der booleschen Algebra gewöhnt hat, ist es nicht schwer, zu verifizieren, dass es sich um die Wertetafeln der beiden Operationen handelt, die auszuführen sind, wenn zwei Dualzahlen stellenweise addiert werden sollen. In jedem Berechnungsschritt, d.h. für jede Stelle muss aus den jeweiligen Summandenbits x_1 und x_2 und dem Übertragsbit z aus der vorherigen Stelle das jeweilige Resultatbit y und das neue Übertragsbit z' berechnet

13 werden. Ein Operator, der beide Funktionen berechnet, ist ein Dualstellenaddierer; er wird auch **Volladdierer** genannt.

x_1	x_2	z	z'	y
0	0	0	0	0
0	0	1	0	1
0	1	0	0	1
0	1	1	1	0
1	0	0	0	1
1	0	1	1	0
1	1	0	1	0
1	1	1	1	1

Bild 9.2 Wertetafeln des Volladdierers

Die Arbeitsweise des Volladdierers soll anhand der letzten Zeile der Wertetafel demonstriert werden. Es sind drei Einsen zu addieren. Das ergibt die Dezimalzahl 3 und die Dualzahl 11, d.h. sowohl das Resultatbit als auch das Übertragsbit hat den Wert 1. Die Bezeichnung des Übertragsbits mit z soll an den inneren Zustand des Automaten erinnern. Er entspricht ihm genau, wenn die einzelnen *Stellen* einer Summe *iterativ* mit Hilfe des Volladdierers ("*Stellen*addierers") berechnet werden, denn dann muss der Übertrag über einen Schrittverzögerer (Taktverzögerer) auf den Eingang des Volladders zurückgegeben werden, also über einen Speicher, der einen Eingabeoperanden um einen Takt verzögert ausgibt. Die gesamte Schaltung ist ein sogenannter **sequenzieller Addierer**. In Kap.9.5 werden wir sehen, dass sich auch der Schrittverzögerer aus elementaren booleschen Operatoren aufbauen lässt (siehe Bild 9.7), sodass der sequenzielle Addierer als *zirkuläres boolesches Netz* realisierbar ist.

Aus den Wertetafeln von Bild 9.2 lassen sich die Ausdrücke (9.8) für die beiden Funktionen ablesen. Der besseren Lesbarkeit halber ist eine **verkürzte Notation** verwendet worden, bei der die Negation durch Überstreichung notiert wird und die Konjunktionssymbole sowie die Klammern fortgelassen werden. Außerdem wird die Disjunktion ab jetzt stets durch OR bezeichnet.

$$
\begin{aligned}
z' &= x_1x_2\overline{z} \ \texttt{OR}\ x_1x_2z \ \texttt{OR}\ x_1\overline{x_2}z \ \texttt{OR}\ \overline{x_1}x_2z \\
y &= x_1\overline{x_2}\,\overline{z} \ \texttt{OR}\ \overline{x_1}x_2\overline{z} \ \texttt{OR}\ \overline{x_1}\,\overline{x_2}z \ \texttt{OR}\ x_1x_2z
\end{aligned}
\qquad (9.8)
$$

Kanonische disjunktive Normalformen sind häufig redundant und lassen sich vereinfachen. So kann z.B. die erste Disjunktion in dem Ausdruck für z' in (9.8) durch x_1x_2 ersetzt werden, denn in beiden Argumenten der Disjunktion treten x_1 und x_2 nichtnegiert auf, während z einmal negiert und das andere mal nichtnegiert auftritt, sodass der Wert der Disjunktion nicht davon abhängt, welchen Wert z annimmt. In einer reduzierten (nicht mehr kanonischen) disjunktiven Normalform (abgekürzt DNF) müssen die Konjunktionen nicht mehr sämtliche Variablen enthalten. Es sind viele Reduktionsalgorithmen entwickelt worden, die bei der technischen Realisierung boolescher Funktionen zum Einsatz kommen, u.a. um die Produktionskosten der Schaltkreise zu senken.

Die Bilder 9.3 und 9.4 zeigen die Überführung einer KDNF in eine Schaltung (sprich: in ein boolesches Netz) für den sehr einfachen Fall der Addition zweier *ein*stelliger Dualzahlen. Die Summe ist eine ein- oder zweistellige Dualzahl mit den

Stellenbits y_1 und y_2. Ein Operator, der diese Operation ausführt, heißt **Halbaddierer**. Für die Berechnung des *letzten* Bits der Summe zweier Dualzahlen reicht der Halbaddierer aus, weil es keinen Übertrag einer vorangehenden Stellenaddition gibt. Aus der Wertetafel von Bild 9.3 lassen sich die beiden unterhalb der Wertetafel angegebenen KNDF für y_1 und y_2 ablesen, die ihrerseits unmittelbar in die Schaltung von Bild 9.4a überführt werden können. Die drei AND-Operatoren entsprechen den drei Konjunktionen in den booleschen Ausdrücken für y_1 und y_2 (die Konjunktionszeichen sind unterdrückt). Die kleinen Kreise an den Eingängen der AND-Glieder stellen Negatoren dar.

x_1 x_2	y_1 y_2
0 0	0 0
0 1	0 1
1 0	0 1
1 1	1 0

$y_1 = x_1 x_2$
$y_2 = \overline{x}_1 x_2 \text{ OR } x_1 \overline{x}_2$

Bild 9.3 Wertetafel des Halbaddierers und die beiden KNDF zur Berechnung der Resultatbits.

Bild 9.4b zeigt die Schaltung des Halbaddierers mit einer veränderten, matrixförmigen Topologie der Verbindungsleitungen. Die Matrix (der Teil der Schaltung rechts von den AND-Gliedern) besteht aus zwei senkrechten und drei waagerechten Leitern[2]. Die

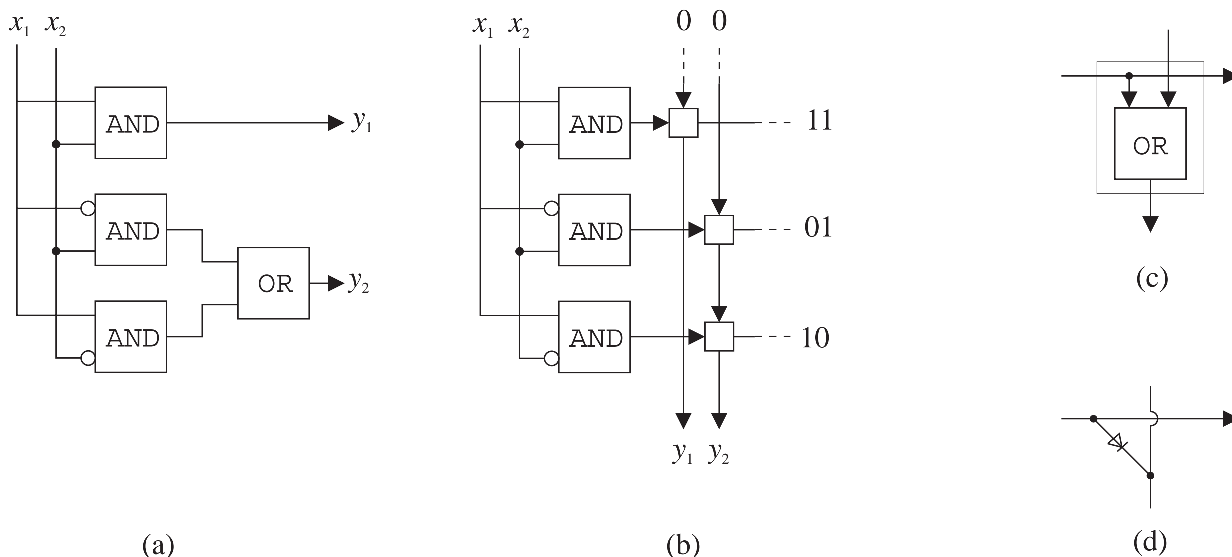


Bild 9.4 Schaltung des Halbaddierers, **(a)** als boolesches Netz, **(b)** als Leitermatrix; **(c)** - Dekomponierung eines Schnittpunktoperators; **(d)** Realisierung eines Schnittpunktoperators mittels Diode. Die kleinen Kreise stellen Negatoren dar.

2 Im Weiteren werden die Wörter *Leitung* und *Leiter* wie Synonyme verwendet, wenn von Leitungs- oder Leitermatrizen die Rede ist. Gemeint ist ein “Stück leitender Draht” bzw. eine entsprechend dotierte Strecke in einem Chip.

punktierten Verlängerungen sollen die Vorstellung hervorrufen, dass die Matrix aus einer größeren Matrix herausgeschnitten ist. An den oberen Enden der senkrechten Leiter liegt stets der Wert 0 an. Die kleinen Quadrate in den Schnittpunkten der senkrechten mit den waagerechten Leitern sind spezielle Flussknoten, die man Schnittknoten nennen könnte, doch ziehen wir in diesem Fall die Bezeichnung **Schnittpunktoperator** vor. Ein Schnittpunktoperator besitzt zwei Eingänge und zwei Ausgänge und lässt sich in die Schaltung von Bild 9.4c dekomponieren. Daraus ist ersichtlich, dass ein Schnittpunktoperator den von links erhaltenen Wert nach rechts weitergibt und dass er nach unten immer dann eine 1 übergibt, wenn er über mindestens einen Eingang eine 1 empfängt. In Kap.12.1 [12.1] wird gezeigt, dass der Schnittpunktoperator durch die in Bild 9.4d gezeigte Diodenschaltung realisiert werden kann.

14

Überzeugen wir uns, dass es sich bei der Schaltung von Bild 9.4b tatsächlich um den Halbaddierer handelt. Man beachte, dass wir es mit einem zirkelfreien Operatorennetz zu tun haben, das keine Weichen, sondern nur starre Flussknoten enthält, also Gabeln und Vereinungen. Das trifft offensichtlich für alle DNF-Schaltungen zu.

Für jedes der Eingabepaare 01, 10, 11 belegt genau eines der drei AND-Glieder seine (waagerechte) Ausgabeleitung mit einer 1. Dadurch ist eine Zuordnung zwischen Eingabepaaren und waagerechten Leitern festgelegt. Die jeweiligen Bitpaare sind rechts neben den Leitern angegeben. Eine Schaltung, die Bitketten (Binärwörtern) Leitungen zuordnet (jedem Wort je eine Leitung), nennen wir **Wort-Leitung-Zuordner**. Sie wird in Kap.12 eine wichtige Rolle spielen. Aus der Funktionsweise der Schnittpunktoperatoren folgt, dass eine senkrechte Leitung eine Disjunktion realisiert, und zwar eine Disjunktion derjenigen Konjunktionen, mit deren Ausgabeleitungen sie über einen Schnittpunktoperator verbunden ist. Eine senkrechte Leitung stellt also zusammen mit den Konjunktionsgliedern die Schaltung einer DNF dar und zwar einer KDNF, denn die Negatoren sind so platziert, dass sämtliche möglichen Eingabetupel berücksichtigt werden. Das AND-Glied mit zwei negierten Eingängen erübrigt sich, denn die Eingabe 00 auf die Schaltung von Bild 9.4b bewirkt die Ausgabe 00 (Taktung mittels Toren vorausgesetzt).

15

Die Matrix enthält zwei senkrechte Leitungen, die Schaltung realisiert also zwei boolesche Funktionen in kanonischer disjunktiver Normalform. Durch Vergleich der Konjunktionen verifiziert man leicht, dass es dieselben Funktionen sind, die durch die Schaltung von Bild 9.4a berechnet werden. Die Darstellung als *Leitermatrix* entspricht der mikroelektronischen Realisierung mittels *Diodenmatrix* bzw. *Transistormatrix* (siehe Kap.12.1 [12.1]).

Zur Bestimmung der KDFN einer Funktion f hat man gemäß obiger Methode die vollständige Bedingung dafür aufzuschreiben, dass f gleich 1 ist. Am Rande sei angemerkt, dass man auch die Bedingung dafür aufschreiben kann, dass f gleich 0, also $\overline{f}$ gleich 1 ist. Wenn man den gesamten sich so ergebenden Ausdruck negiert und die morganschen Regeln mehrmals anwendet, gelangt man zu einem Ausdruck, der eine Konjunktion von Disjunktionen darstellt. Diese Darstellung einer booleschen

Funktion heißt **kanonische konjunktive Normalform (KKNF)**. Auch sie lässt sich in der Regel vereinfachen. Man wird die konjunktive Normalform der disjunktiven dann vorziehen, wenn die Funktion für relativ *wenige* Argumentwertetupel den Wert 0 annimmt. Im entgegengesetzten Fall wird man die KDNF vorziehen.

Damit ist unser erstes Ziel erreicht. Es wurde gezeigt, dass sich jede boolesche Funktion mittels Negation, Konjunktion und Disjunktion ausdrücken lässt. In diesem Sinne sagen wir, dass Negation, Konjunktion und Disjunktion zusammen einen **vollständigen Satz** boolescher Funktionen bilden. Mit diesem vollständigen Satz werden wir vorzugsweise bei der Komponierung der Hardware arbeiten, obwohl er sich reduzieren lässt. Man kann nämlich entweder auf die Konjunktion oder auf die Disjunktion verzichten, weil sich nach den morganschen Regeln die eine durch die andere (und durch die Negation) ausdrücken lässt. Es genügt sogar einzig und allein die NOR-Funktion, um sämtliche booleschen Funktionen komponieren zu können. Der Beweis lässt sich aus Bild 9.1 ablesen. Als Hinweis sei gesagt, dass z.B.

$$x \texttt{ NOR } x = \overline{x}$$

und

$$(x \texttt{ NOR } y) \texttt{ NOR } (x \texttt{ NOR } y) = x \texttt{ OR } y$$

gilt. Analoge Formeln lassen sich für die NAND-Funktion angeben.

Die Schaltungen dieses Kapitels sind zirkelfreie boolesche Netze aus elementaren booleschen Operatoren, also Kombinationsschaltungen. Doch sind sie von einem speziellen Typ, denn ihre Schaltungsstruktur entspricht der Syntax der disjunktiven Normalform. Darum nennen wir sie **disjunktive Kombinationsschaltungen** [3].

Die beschriebene konstruktive Methode zur Überführung der Wertetafel einer Binärwortfunktion in eine disjunktive Kombinationsschaltung ist offensichtlich auf beliebige Funktionstafeln anwendbar. Daraus folge der

Satz: Zu jeder Binärwortfunktion, die durch ihre Wertetafel festgelegt ist, lässt sich eine Kombinationsschaltung (ein zirkelfreies boolesches Netz) zur Berechnung der Funktion (zur Speicherung der Wertetafel) angeben. 16

Das ist ein Resultat von großer praktischer Bedeutung. Unbefriedigend ist allerdings die Beziehungslosigkeit zu den Überlegungen in Kap. 8.4.5, und es stellt sich die Frage, welche Relation zwischen booleschen und rekursiven Funktionen besteht. Die Antwort lautet: Boolesche Funktionen *sind* rekursive Funktionen. Davon kann man sich überzeugen, wenn man sich die Gültigkeit folgender Äquivalenzen klarmacht.

$$a \texttt{ AND } b \Leftrightarrow [a*b = 1] \tag{9.9a}$$

$$a \texttt{ OR } b \Leftrightarrow [a+b > 0] \tag{9.9b}$$

---

3 Diese Bezeichnung ist in der Literatur weniger üblich. Häufig wird der Begriff der kombinatorischen Schaltung ausschließlich für *disjunktive* kombinatorische Schaltungen verwendet.

$$\overline{a} \Leftrightarrow [a = 0] \qquad (9.9c)$$

Das Prädikat auf der rechten Seite jedes der drei Äquivalenzen ist genau dann erfüllt (ist genau dann gleich 1), wenn der boolesche Ausdruck auf der linken Seite gleich 1 ist. Beispielsweise ist das Prädikat in (9.9b) genau dann erfüllt (also gleich 1), wenn a oder b oder beide gleich 1 sind, wenn also a OR b gleich 1 ist. Man beachte, dass a und b auf den linken Seiten der Gleichungen (9.9) boolesche Variablen, auf den rechten Seiten arithmetische Variablen darstellen. Es handelt sich jeweils um unterschiedliche Interpretationen ein und derselben abstrakten Funktion.

Die Prädikate auf den rechten Seiten der Gleichungen legen rekursive Funktionen fest, denn sie besitzen rekursive charakteristische Funktionen. Also sind auch die drei booleschen Funktionen auf den linken Seiten rekursive Funktionen. Da sich jede boolesche Funktion in die disjunktive Normalform überführen lässt, folgt: ***Booleschen*** *Funktionen sind* ***rekursive*** *Funktionen*; und da sich jede Kombinationsschaltung aus booleschen Operatoren komponieren lässt, gilt der

17

Satz: Die durch Kombinationsschaltungen realisierbaren Funktionen sind rekursive Funktionen.

18

Damit beenden wir unseren Streifzug durch die boolesche Algebra. In Kap.11.1 werden wir ihn durch einen historischen Streifzug ergänzen und in großen Schritten den Weg durcheilen, der, in der Antike beginnend, schließlich zur booleschen Algebra und zu den mikroelektronischen Bausteinen eines Computers geführt hat.

9.4 Netzklassen

Die allgemeinen Überlegungen des Kapitels 9.1 und 9.2 über die Komponierung informationeller Operatoren haben unsere Aufmerksamkeit auf Operatorennetze unterschiedlicher Natur (boolesche und neuronale Netze) und unterschiedlicher Struktur (zirkelfreie und zirkuläre) gelenkt, wobei sich die Reihenfolge der betrachteten Netze mehr aus dem Gang unserer Gedanken und den sich dabei erhebenden Fragen und weniger aus irgendeiner Systematik heraus ergab. Dieser Mangel soll nun dadurch behoben werden, dass die bisher erwähnten Operatorennetze nach bestimmten Eigenschaften systematisiert werden. Dabei werden wir auch auf Netzklassen stoßen, die bisher noch keine Erwähnung gefunden haben. Wir beginnen die Systematisierung von Operatorennetzen mit der Frage, welche Folgen die statische Codierung auf die Komponierung von Netzen aus elementaren Operatoren hat, unabhängig davon, wie diese physikalisch funktionieren.

In Kap.9.2 [3] waren wir zu einer wichtigen Einsicht gelangt: Die elementaren Operatoren informationeller Systeme mit statischer Codierung besitzen kein Gedächtnis; sie können “sich nichts merken”. Der Ausgabeoperand (das Resultat) einer Operation liegt nur solange am Ausgang, wie der Eingabeoperand am Eingang liegt. Das gilt für elementare boolesche Operationen und für zirkelfreie Netze aus elementaren booleschen Operatoren, also auch für Kombinationsschaltungen.

Wir wollen nun die Möglichkeiten der Vernetzung elementarer Operatoren systematisch untersuchen. Dazu unterteilen wir alle denkbaren Netze in Klassen, wobei wir drei Eigenschaften als Klassifikationskriterien verwenden wollen. Wir werden Netze danach unterscheiden, ob sie aus booleschen Operatoren oder aus künstlichen Neuronen bestehen, ob sie eine zirkelfreie oder eine zirkuläre Struktur besitzen und ob sie Speicher enthalten oder nicht. Die sich ergebende Klassen und Unterklassen sind in Bild 9.5 aufgelistet. In Klammern sind Beispiele angefügt, von denen diejenigen später ausführlich behandelt werden, welche auf booleschen Netzen basieren.

- Boolesche Netze
  - boolesche zirkelfreie Netze (Kombinationsschaltung, ROM, ALU)
  - boolesche zirkuläre Netze
    - boolesche zirkuläre Netze ohne interne Speicher (Flipflop)
    - boolesche zirkuläre Netze mit internen Speichern (KR-Netz, Prozessor, Prozessorcomputer)
- Neuronale Netze
  - neuronale zirkelfreie Netze (Perzeptron, ADALINE)
  - neuronale zirkuläre Netze
    - neuronale zirkuläre Netze ohne interne Speicher (Hopfieldnetz)
    - neuronale zirkuläre Netze mit internen Speichern (Neurocomputer)

Bild 9.5 Netzklassen

Wir werden die Netzklassen der Reihe nach besprechen.

Zirkelfreie boolesche Netze

Offensichtlich lassen sich elementare boolesche Operatoren ohne weiteres (ohne Einsatz spezieller Speichereinheiten) zu zirkelfreien Netzen verbinden. Die Schaltung des Halbaddierers von Bild 9.4 ist hierfür ein Beispiel. Speichervorrichtungen zwischen den Bausteinoperatoren sind nicht erforderlich. Doch liegt die Summe nur solange am Ausgang der Schaltung, wie die Summanden an seinem Eingang liegen. Der Additionsschaltung muss also ein Speicher für die Summanden vorgeschaltet sein, d.h. eine Schaltung mit statisch stabilen Zuständen für die Codierung der Summanden. So kann der Zustand der Additionsschaltung stabil gehalten werden, sodass am Ausgang die Summe stabil codiert vorliegt.

Diese Schlussfolgerung gilt für beliebige zirkelfreie Netze aus elementaren Operatoren, d.h. für beliebige Kombinationsschaltungen und zirkelfreie neuronale Netze. *Durch das Vorschalten eines Speichers vor einen Operator ohne Gedächtnis wird*

die statische Stabilität des Zustandes des Operators gesichert und damit die statische Codierung des Ausgabeoperanden ermöglicht.

Die beiden Klassen der zirkelfreien booleschen und neuronalen Netze mit internen Speichern fehlen in Bild 9.5. Sie bedürfen keiner besonderen Betrachtung. Denn ihre Realisierung ist an keine zusätzlichen Bedingungen geknüpft im Vergleich zu den zirkelfreien Netzen *ohne* interne Speicher, und sie besitzen diesen gegenüber keine grundsätzlich neuen Eigenschaften, abgesehen von der Möglichkeit, dass sie auf ein Eingabewort nicht mit einem einzigen Ausgabewort, sondern mit einer Folge von Ausgabewörtern reagieren.

Zirkuläre boolesche Netze

Ganz anders liegen die Dinge, wenn ein Netz Rückkopplungen (zirkuläre Verbindungen) enthält. Dass Rückkopplungen *mit* internen Speichern möglich sind, ist offensichtlich. Weniger offensichtlich ist, dass auch Rückkopplungen *ohne* Speicher möglich und sinnvoll sein können. Betrachten wir den denkbar einfachsten Fall einer *booleschen Rückkopplungsschleife*, die Zurückführung des Ausganges eines Negationsgliedes auf seinen Eingang (Bild 9.6a). Die Rückkopplung lässt keinen statisch stabilen Zustand zu. Der Zustand “will” ständig hin und herspringen. (Ob sich die beiden Zustände tatsächlich abwechselnd voll ausbilden können, hängt von den Details des Übergangsprozesses ab). Die *Kopplungszirkularität* führt zu einer eigentümlichen “Widersprüchlichkeit”, die an die Antinomien erinnert, die infolge *referenzieller Zirkularität* auftreten können, wie beispielsweise in der auf sich selbst bezogenen Aussage “Dieser Satz ist falsch” (vgl. Kap.6.2 [6.1]).

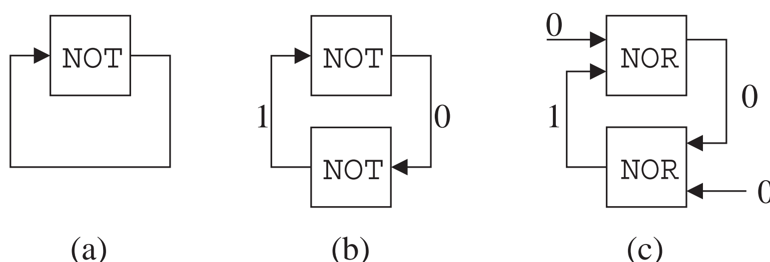


Bild 9.6 Einfache boolesche Rückkopplungsschleifen; **(a)** - widersprüchlicher Zirkel; **(b)** - Zirkel mit zwei stabilen Zuständen; **(c)** - Idee des Flipflop.

Betrachten wir daraufhin noch einmal die Prinzipschaltung des Automaten. Nehmen wir an, das Quadrat in Bild 8.5 stelle eine Kombinationsschaltung dar. Entfernt man nun den Speicher (den mit D bezeichneten Taktverzögerer), ergibt sich eine ähnlich undefinierte Situation wie im Falle der Schleife von Bild 9.6a. Der momentan berechnete Wert z' erscheint unmittelbar am Eingang der Kombinations-

schaltung, die sofort einen neuen z'-Wert berechnet, wodurch wiederum eine Neuberechnung initiiert wird und so weiter. Es liegt ein *widersprüchlicher* Zirkel vor.

Um die Widersprüchlichkeit des Zirkels in Bild 9.6a zu verhindern, muss in Analogie zu Bild 8.5 ein Speicher in die Schleife eingefügt werden. Es genügt jedoch nicht, den z'-Wert auf unbestimmte Zeit aufzubewahren, es muss der Zeitpunkt vorgebbar sein, zu dem der Wert weitergegeben wird. Zu diesem Zweck muss ein Tor in die Schleife eingebaut werden und zwar *vor* dem Speicher, da dieser den Eingabezustand des nachfolgenden Operators aufrechterhalten muss.

Diese Einsicht lässt sich verallgemeinern: *Sowohl boolesche Operatoren und als auch Kombinationsschaltungen lassen sich zu* ***zirkulären booleschen Netzen*** *verbinden, wenn in jeden Zirkel (in jede Rückkopplungsschleife) ein* ***Speicher mit vorgeschaltetem Tor*** *eingebaut wird.* Der Speicher ist jedoch nicht in allen Fällen notwendig, wie sogleich gezeigt wird. 19

Werden alle Tore eines zirkulären booleschen Netzes *synchron* (gleichzeitig) gesteuert, ergibt sich eine *getaktete* Arbeitsweise des Netzes. In jedem Takt stellt sich ein neuer Zustand ein. Während einer Folge von Takten wird eine Folge (*Sequenz*) von Binärwörtern ausgegeben; das Netz zeigt ein *sequenzielles* Verhalten. Fasst man die einzelnen Speicher des Netzes gedanklich zu einem einzigen Speicher zusammen, gelangt man zu der Schaltung von Bild 8.5 und zum Begriff des abstrakten Automaten (vgl. Kap.8.2.3). Fasst man sie hardwaremäßig zu einem zentralen Speicher zusammen, gelangt man zur Prinzipschaltung des Prozessorrechners, wie schon hier vorgreifend auf Kap.13 bemerkt sei.

Wir betrachten nun einen autonomen Automat (oder auch einen Automat mit konstantem Eingabewert), sodass $z' = f(z)$ gilt. Es kann nun der Fall eintreten, dass der momentan berechnete neue Zustand mit dem alten identisch ist, $z' = z$, dass also gilt

$$f(z) = z. \tag{9.10}$$

Dann bleibt der Automat in diesem Zustand “stecken”. Es bildet sich trotz Rückkopplung ein statisch stabiler Zustand aus, sodass der Speicher überflüssig wird. Einen solchen Zustand nennen wir **Eigenzustand**.

Der Begriff des Eigenzustandes ist in der Physik üblich, wo er im Zusammenhang mit *Differenzialgleichungen* verwendet wird. Der zugrunde liegende Tatbestand ist der gleiche: Die Stabilität eines Zustandes setzt die Gleichheit von Eingabe und Ausgabe des beschreibenden Operators voraus. In diesem Zusammenhang sei an den Analogrechner erinnert. Die Rückkopplungschleife in Bild 4.2b erzwingt die Gleichheit von Eingabe und Ausgabe, die in der zu berechnenden Differenzial*gleichung* formal notiert ist. In der Algorithmentheorie wird ein Argumentwert einer Funktion, für den (9.10) gilt, als **Fixpunkt** der betreffenden Funktion bezeichnet.

Zum Begriff des Eigenzustandes waren wir auf formalem Wege gelangt. Es stellt sich die Frage, wie sich Eigenzustände in booleschen Netzen realisieren lassen. Bild 9.6b zeigt das denkbar einfachste Beispiel, eine Schleife über *zwei* Negations-

20

glieder. Die Quelle der Instabilität der Schaltung von Bild 9.6a ist beseitigt, es können sich zwei verschiedene stabile Zustände einstellen (einer von ihnen ist in Bild 9.6b angegeben), die dadurch gekennzeichnet sind, dass auf den beiden Leitungen, welche die Negationsglieder verbinden, entgegengesetzte Werte liegen. Die Schleife in Bild 9.6b stellt also keinen widersprüchlichen Zirkel dar. Logisch entspricht er den beiden Sätzen in Kap.6.2 [6.2], von denen jeder behauptet, der andere sei falsch. Sie stellen *keine* Antinomie dar.

Man kann sich umfangreichere Strukturen ausdenken, in denen Eigenzustände möglich sind, z.B. Schleifen mit einer geraden Anzahl von Negatoren. Es sind auch verzweigte Strukturen (Netze) mit mehr als zwei Eigenzuständen denkbar. Das legt den Gedanken nahe, diese Zustände als *codierende* Zustände und ein solches Netz als Speicher zu verwenden. Damit ein zirkuläres boolesches Netz ohne interne Speicher als **Schreib-Lese-Speicher** dienen kann, muss die Möglichkeit bestehen, den Zustand, in dem sich das Netz gerade befindet, über einen externen Eingang in einen anderen stabilen Zustand zu überführen, d.h. den alten Gedächtnisinhalt mit einem neuen zu überschreiben.

Die Erfindung, die aus der Schleife in Bild 9.6b einen Schreib-Lese-Speicher macht, war für die Rechentechnik von fundamentaler Bedeutung, obwohl er natürlich nur ein einziges Bit speichern kann. Sie besteht in der Ersetzung der Negationsglieder durch NOR-Glieder, sodass sich eine Schleife mit zwei externen Eingängen ergibt (Bild 9.6c). Über sie lässt sich der Zustand verändern. Wenn z.B. an den linken (oberen) externen Eingang eine 1 angelegt wird, bewirkt das eine 0 auf der Ausgabeleitung des oberen NOR-Gliedes, unabhängig davon, welcher Wert vorher auf dieser Leitung gelegen hatte. Das Bemerkenswerte dabei ist, dass sich der Zustand aufrecht erhält, wenn die 1 am Eingang wieder verschwindet und an beiden Eingängen eine 0 liegt. Dieser Zustand ist in Bild 9.6c dargestellt. In entsprechender Weise lässt sich durch eine 1 am rechten (unteren) externen Eingang der umgekehrte stabile Zustand einstellen.

Um die Schleife in einen bestimmten Zustand “einflippen” zu lassen, genügt es also, einen kurzen 1-Impuls auf den entsprechenden Eingang zu geben. Wird anschließend auf den anderen Eingang ein 1-Impuls gegeben, “floppt” die Schleife in den anderen Zustand über. Von daher rührt wohl die lautmalerische Bezeichnung **Flipflop**. Der Flipflop von Bild 9.6c reagiert nur auf Einsen. In Kap.9.5 werden wir uns überlegen, wie sich der Fliflop zu einem Schreib-Lese-Speicher für ein Bit, zu einem **Ein-Bit-Speicher** erweitern lässt, und in Kap.13.2 werden wir aus Ein-Bit-Speichern Register und adressierbare elektronische Speicher komponieren. Unabhängig von der Komponierungsstufe werden wir von *booleschen* Speichern sprechen. *Speicher, deren statisch codierende Zustände Eigenzustände zirkulärer boolescher Netze sind, nennen wir* ***boolesche Speicher***.

Eigenzustände in booleschen Netzen kommen praktisch nur in Ein-Bit-Speichern als den Bausteinen von Registern und adressierbaren Speichern zur Anwendung. Demgegenüber haben Eigenzustände in neuronalen Netzen, auch in sehr großen

neuronalen Netzen, breite technische Anwendung gefunden (s.u.). Die eingeschränkte Nutzung von Eigenzuständen in booleschen Netzen ist verständlich, denn ein solches Netz kann stets durch eine Kombinationsschaltung mit vorgeschaltetem Speicher ersetzt werden.

Aus der Analyse der verschiedenen Möglichkeiten, Kompositoperatoren (Schaltungen) aus booleschen Operatoren zu komponieren, hat gezeigt, dass die Komponierung sowohl zirkelfreier als auch zirkulärer Netze zur Paarung von Speicher und Kombinationsschaltung führt. Die Notwendigkeit, einer Kombinationsschaltung einen Speicher vorzuschalten, ist die Folge der Forderung, dass statische Codierung möglich sein soll. Um diese Einsicht kompakter und aussagekräftiger ausdrücken zu können, vereinbaren wir: *Eine Kombinationsschaltung mit vorgeschaltetem Speicher wird als einfachster Vertreter von Netzen aus Kombinationsschaltungen und Speichern aufgefasst; elementare boolesche Operatoren werden als einfachste Vertreter von Kombinationsschaltungen aufgefasst*. Wenn wir noch berücksichtigen, dass es infolge der Arbitrarität der Codierung stets möglich ist, binär-statische Codierung in irgendeine andere statische Codierung umzucodieren, ohne dadurch die Allgemeingültigkeit der vorangehenden Überlegungen einzuschränken[4] gelangen wir zu folgendem

Satz: Aus booleschen Operatoren komponierte informationelle Operatoren mit statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und booleschen Speichern (oder in solche überführbar), wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegt. 21

Zirkelfreie neuronale Netze

Offensichtlich können Schwellenoperatoren - ebenso wie boolesche Operatoren - ohne Schwierigkeiten zu zirkelfreien Netzen verbunden werden. Wenn mehrere Bausteinoperatoren (Neuronen) eines neuronalen Netzes externe Ausgänge besitzen, liefert das Netz an dem gemeinsamen (“vereinten”) Ausgang ein Binärwort. Legt man an den (vereinten) Eingang ebenfalls ein Binärwort, wird dieses transformiert. Das Netz realisiert eine bestimmte Binärwortfunktion.

Betrachtet man einen derartigen Kompositoperator als schwarzen Kasten, so ist er in seinem Verhalten von einer Kombinationsschaltung nicht zu unterscheiden, solange die Gewichte (Synapsenleitwerte) konstant bleiben. Das bedeutet, dass zu jedem zirkelfreien neuronalen Netz mit definierten Gewichten eine Kombinationsschaltung mit dem gleichen Eingabe-Ausgabeverhalten angeben werden kann, sodass der gegen Ende des Kapitels 9.3 [9.18] für Kombinationsschaltungen formulierte Satz auch für für zirkelfreie neuronale Netze gilt:

---

4 In Kap,12.1 wird gezeigt, dass eine solche Umcodierung mittels zirkelfreier boolescher Netze stets möglich ist.

Satz: Die durch zirkelfreie neuronale Netze realisierbaren Funktionen sind rekursive Funktionen.

Alles, was in Verbindung mit Kombinationsschaltungen hinsichtlich der statischen Codierung gesagt wurde, lässt sich sinngemäß auf zirkelfreie neuronale Netze übertragen. Künstliche Neuronen und zirkelfreie Netze aus künstlichen Neuronen besitzen - ebenso wie elementare boolesche Operatoren und zirkelfreie boolesche Netze - keine inneren stabilen Zustände, also kein Gedächtnis. Wenn sie zur Informationsverarbeitung mit statischer Codierung verwendet werden sollen, müssen ihnen - ebenso wie Kombinationsschaltungen - Speicher vorgeschaltet werden.

Der Umstand, dass neuronale Netze, deren interne Operanden reelle Größen sind, mit binär-statischer Codierung arbeiten können, mag im ersten Augenblick Unverständnis hervorrufen. Er wird verständlich, wenn man sich folgendes klarmacht. Voraussetzung für statische Codierung ist, wie gesagt, das Vorschalten eines Speichers. Ist diese Voraussetzung erfüllt, so führt der statische Zustand des Speichers (d.h. der Speicherinhalt) zur Ausbildung eines statischen Zustandes im nachgeschalteten neuronalen Netz. Dabei entsprechen den booleschen Werten 0 und 1 unterschiedliche reelle Werte der codierenden Zustandsparameter, etwa den Spannungswerten im Netz. Das widerspricht nicht dem Prinzip der statischen Codierung. Die Änderung der codierenden Parameterwerte führt dazu, dass an unterschiedlichen Punkten des Netzes den booleschen Werten 0 und 1 unterschiedliche reelle Werte bzw. Wertebereiche entsprechen.

Ein Blick in die Literatur zeigt, dass vorzugsweise neuronale Netze mit sehr vielen Eingängen untersucht werden und dass diese der Anschaulichkeit halber zweidimensional (in einer Ebene) angeordnet werden, sodass jedem *Eingangsneuron* ein Punkt der Ebene, m.a.W. ein Pixel der *Bild*fläche entspricht. Wenn nun die Pixel der

22

angeregten Neuronen *schwarz* gefärbt werden und die übrigen Pixel *weiß* (untergrundfarben) bleiben, ergibt sich ein gerastertes Schwarz-weiß-Muster, das sog. *Eingabemuster* des Netzes. Häufig werden auch Netze mit sehr vielen *Ausgangsneuronen* untersucht, die ebenfalls zu einem Muster zusammengesetzt werden können. Eventuell kann der Betrachter die Muster interpretieren, also in ihnen irgendwelche Objekte *erkennt*. Erkennt er z.B. in einem Ausgabemuster den Buchstaben A, so ist dies kein Zeichenrealem, denn es trägt keine Symbolfunktion. Das Netz gibt ein Muster aus, das nur "wie ein A aussieht". Darum sprechen wir vom *subsymbolischen* Charakter der Muster [1.4], im Gegensatz zum *symbolischen* Charakter der Ein- und Ausgaben boolescher Netze bzw. traditioneller Rechner. Wenn ein Computer ein A ausgibt, sieht das nicht nur wie ein A aus, sondern es *ist* ein A (das Zeichenrealem A).

Wir wollen uns überlegen, welchen Nutzen man aus dem Umstand ziehen kann, dass die Ausgabeoperanden eines neuronalen Netzes - ebenso wie im Falle der Kombinationsschaltungen - Binärwörter sind, die Eingabeoperanden aber Tupel reeller Zahlen (Vektoren). Dazu fragen wir uns, was eintreten wird, wenn diese Werte stetig verändert werden. Nach Kap.9.2.2 [9] ist klar, dass bei sehr geringen Verän-

derungen des Eingabevektors keine Veränderung des Ausgabewortes zu erwarten ist. Diese tritt erst bei ausreichender Änderung einer oder mehrerer Komponenten des Eingabevektors ein.

Zur Veranschaulichung dieses Verhaltens gehen wir von 2-dimensionalen Vektoren (Eingabepaaren) aus und ordnen jedem Vektor einen Punkt in einem ebenen Koordinatensystem zu. Markiert man nun alle Punkte, die zu ein und demselben Ausgabewort gehören, mit einer bestimmten Farbe, ergibt sich eine Art Landkarte, wobei jedes "Land" ein bestimmtes Ausgabewort besitzt, das als *Name* des jeweiligen Landes dienen kann. (Ein Land kann aus mehreren nicht zusammenhängenden Gebieten bestehen.)

Diese Eigenschaft bleibt auch für 3- und mehrdimensionale Inputvektoren erhalten. Abstrahiert man von dem geometrischen Bild und fasst die Werte der Komponenten des Eingabevektors als Werte *irgendwelcher Merkmale* auf, dann wird die Verhaltensweise des Netzes zu dem, was wir in Kap.5.5 [5.15] mit *Klassifizieren* bezeichnet haben. Das Netz *bildet Klassen*, es funktioniert als Klassifikator; es fasst Mengen von Vektoren zu Klassen zusammen und "bezeichnet" die Klassen mit Ausgabewörtern. Bei Änderung der Gewichte ändern sich die Klassen. Ein neuronales Netz, das alle Eingaben in zwei Klassen einteilt (das eine sogenannte **Dichotomie** ausführt), benötigt nur ein einziges Ausgabeneuron. Das Ausgabebit *codiert* die Klasse. 23

Die Tatsche, dass neuronale Netze klassifizieren können, sollte eigentlich nicht überraschen, denn ihr Aufbau ist, wie gesagt, dem Gehirn abgelauscht, dessen wohl erstaunlichste und vielleicht auch wichtigste Eigenschaft die Fähigkeit zum Klassifizieren ist. Es ist jedoch nicht ohne Weiteres klar, wie sich ein neuronales Netz mit einer *vorgegebenen* Klassifizierungsfunktion konstruieren lässt, oder - im Hinblick auf das Gehirn - wie ein neuronales Netz eine *nützliche* Klassifizierungsfunktion durch synaptische Veränderungen *erlernen* kann (siehe dazu [21.5]). Es sind viele einschlägige Methoden entwickelt und in der Literatur beschrieben worden.[5]

Zirkuläre neuronale Netze

Ebenso wie in booleschen Netzen und in abstrakten Automaten können auch in zirkulären neuronalen Netzen Eigenzustände auftreten. Es sind also stabile Zustände möglich. Solche Netze besitzen ein Gedächtnis. Sie können als Speicher verwendet werden. Neuronale Netze mit Eigenzuständen (internen stabilen Zuständen) spielen in der Theorie und Praxis neuronaler Netze eine hervorragende Rolle. Demgegenüber ist die Bedeutung zirkulärer neuronaler Netze mit *internen* Speichern, die selbstverständlich auch möglich sind, fast zu vernachlässigen. Die Situation ist also umgekehrt

5 Siehe z.B. [Dorffner 91], [Grauel 92], [Nauck 96], [Churchland 97], [Werner 95], [Stöcker 95], [Keller 00]. In Kap.21.3.2 [21.5], wird das Lernen mittels Stichprobe skizziert.

wie im Falle zirkulärer *boolescher* Netze. Sequenziell arbeitende neuronale Netze sind gegenwärtig die Ausnahme.

Das ist merkwürdig, denn die Introspektion in das eigene Denken lässt kaum einen Zweifel daran, dass das *Nachdenken* (das logischen Denken, das Ableiten, das Kopfrechnen) ein sequenzieller Prozess ist. Will man ihn mit Hilfe neuronaler Netze nachbilden, muss man offenbar auf sequenziell arbeitende Netze zurückgreifen, also auf Netze mit internen Speichern. Als Speicher können boolesche oder neuronale Speicher verwendet werden. *Speicher, deren statisch codierende Zustände Eigenzustände zirkulärer neuronaler Netze sind, nennen wir* ***neuronale Speicher***.

Alles, was über zirkuläre boolesche Netze mit internen Speichern gesagt worden ist, kann sinngemäß auf zirkuläre neuronale Netze übertragen werden, und der Satz, mit dem die Behandlung der zirkulären booleschen Netze abgeschlossen wurde, kann verallgemeinert werden. Wir fassen die Ergebnisse dieses Kapitels in dem folgenden **Strukturgesetz informationeller Operatoren mit binär-statischer Codierung** zusammen (der Begriff des zirkelfreien Netzes soll wieder den elementaren Operator als Sonderfall einschließen):

Ein ***Informationeller Operator*** *mit binär-statischer Codierung ist ein Kompositoperator aus elementaren booleschen Operatoren oder künstlichen Neuronen* und zwar

- entweder ein zirkelfreies Netz mit vorgeschaltetem Speicher mit Eingangstor
- oder ein zirkuläres Netz aus zirkelfreien Netzen mit vorgeschalteten (internen) Speichern mit Eingangstoren,
- oder er ist durch einen der beiden Typen ersetzbar, ohne dass die Zuordnungen, die er durchführen kann, verändert werden.

Eine Operationsausführung eines Operators des ersten Typs besteht aus einem einzigen Übergangsprozess, seine kausaldiskrete Beschreibung ist nicht dekomponierbar. Eine Operationsausführung eines Operators des zweiten Typs besteht aus einer (nicht unbedingt vollgeordneten) Folge von Übergangsprozessen, sie ist ein *sequenzieller* Prozess. Ein informationelles System, das sequenziell arbeitet und mindestens ein neuronales Netz enthält, nennen wir **Neurocomputer**. (Zuweilen wird die Forderung der sequenziellen Arbeitsweise nicht gestellt.)

Dass zirkuläre neuronale Netze mit internen Speichern bisher kaum Beachtung gefunden haben, liegt u.a. wohl daran, dass die Informatiker zu sehr von den Möglichkeiten fasziniert waren und sind, die sich aus der relativ großen Anzahl möglicher Eigenzustände großer neuronaler Netze ergeben. Das Interesse, das zirkuläre neuronale Netze mit vielen Eigenzuständen hervorriefen, ist verständlich. Wenn nämlich die Eigenzustände zu *codierenden* Zuständen erklärt werden, wird das Netz als Ganzes zu einem Speicher, obwohl es keine internen Speicher besitzt. Es fragt sich, wie man diese Möglichkeit praktisch nutzen kann und ob die Natur diese Möglichkeit nutzt. Wir werden dieser Frage nicht nachgehen, da sie über den Rahmen des Buches hinausgeht, doch wir erinnern uns, dass wir auf das gleiche Problem bereits hinsichtlich boolescher Netze gestoßen waren und zwar in Verbindung mit

Bild 9.6c. Die Frage war, wie sich das dort dargestellte boolesche Netz als Schreib-Lese-Speicher nutzen lässt. Diese Frage soll im folgenden Kapitel beantwortet werden.

Zum Abschluss dieses Kapitels ist eine Bemerkung darüber am Platze, was mit dem vielleicht etwas unmotiviert erscheinenden Abstecher in das Gebiet der neuronalen Netze bezweckt werden sollte. Der Abstecher war notwendig, um die *Vollständigkeitsforderung* zu erfüllen, die wir uns in Kap.9.2 hinsichtlich des Komponierens informationeller Systeme auferlegt hatten, denn als elementare Operatoren kommen, wie wir gesehen haben, nicht nur elementare boolesche Operatoren, sondern auch künstliche Neuronen in Frage. Ferner sollte die Wegegabel aufgezeigt werden, wo der Weg, der zum Neurocomputer führt, von unserem Weg, der zum Prozessorcomputer führt, abzweigt. Auf dem einen Weg wird der Schwellenoperator als interner Bausteinoperator verwendet, auf dem anderen Wege nicht. Im Zusammenhang mit *Prozessorcomputern* kommen Schwellenelemente ausschließlich in Analog-digital-Konvertern zur Anwendung, die dem Computer vorgeschaltet sind, während sie in neuronalen Netzen und im *Neurocomputer* auch als interne Bausteinoperatoren zur Anwendung kommen.

9.5 Ein-Bit-Speicher

Wir greifen noch einmal das Problem des Ein-Bit-Speichers auf, das sich in Kap.9.4 bei der Behandlung zirkelfreier boolescher Netze ergeben hatte, das wir dort aber nicht vollständig gelöst haben. Wie wir inzwischen erkannt haben, müssen Speicher mit Eingangstoren ausgerüstet sein. Außerdem wissen wir, dass es ausreicht, nur kurzzeitig eine 1 an den einen oder anderen Eingang eines Flipflop zu legen, um zu erreichen, dass dieser sich danach in dem einen oder in dem anderen Eigenzustand befindet.

Auf dieser Grundlage lässt sich ein Ein-Bit-Speicher mit vorgeschaltetem Tor aufbauen. Seine Schaltung ist in Bild 9.7 gezeigt. Sie ist vielleicht zu kompliziert, um sofort verstanden werden zu können. Es ist für das Verständnis des Weiteren nicht unbedingt erforderlich, dass der Leser die Wirkungsweise der Schaltung im einzelnen durchspielt. Das gleiche gilt auch für die Schaltungen in den folgenden Kapiteln, die nicht immer ganz einfach sind. Doch wer die Mühe des Durchspielens auf sich nimmt, der wird belohnt. Denn es ist schon amüsant oder sogar spannend, nachzuempfinden, wie aus elementaren logischen Operatoren ein Ein-Bit-Speicher, ein adressierbarer Speicher, ein Prozessor oder ein Computer wird.

Tatsächlich ist der Nachvollzug der Wirkungsweise der Schaltungen, die in diesem Buch behandelt werden, im Grunde auch für den Unvorbereiteten nicht allzu schwierig. Er sollte sich nicht vom ersten Anblick abschrecken lassen. Ohne Frage handelt es sich um recht komplizierte Produkte menschlicher Erfindungsgabe, und die Entwicklung effektiver und zuverlässiger Schaltungen, beispielsweise der Schal-

tung eines Mikroprozessors oder eines elektronischen Speichers, verlangte und verlangt ausdauerndes, konzentriertes und von Intuition und Phantasie beflügeltes Nachdenken zahlloser Erfinder, Ingenieure, Informatiker, Mathematiker und Physiker.

Ein Vergleich von Bild 9.7 mit Bild 9.6c lässt erkennen, dass die dort dargestellte Schaltung um ein Negationsglied und zwei AND-Glieder erweitert und dass eines der beiden NOR-Glieder um 180° gedreht ist, sodass die Ausgänge beider NOR-Glieder in die gleiche Richtung zeigen.

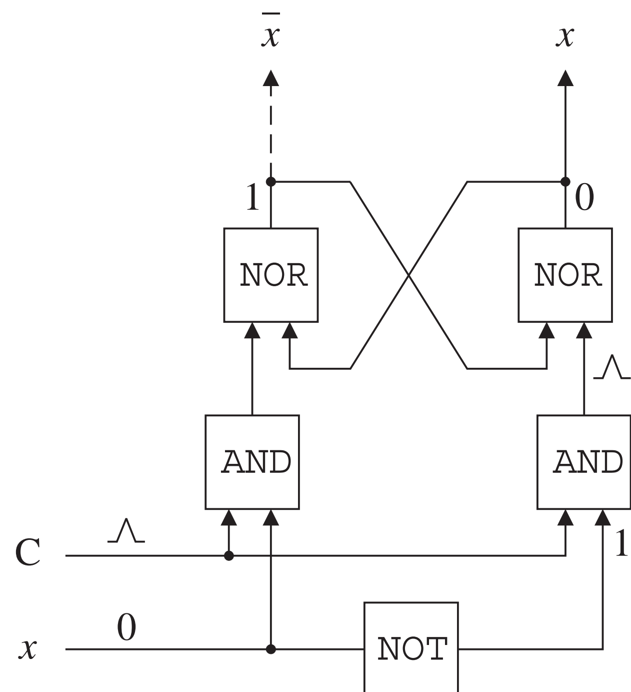


Bild 9.7 Boolesches Netz des Ein-Bit-Speichers.

Die Schaltung besitzt zwei Eingänge, einen *Arbeitseingang*, an dem das zu speichernde Bit (mit x bezeichnet) als statisch stabiler Eingabezustand anliegt, und einen *Steuereingang*, mit C bezeichnet, an dem normalerweise eine 0 liegt, an den aber kurzzeitig eine 1 angelegt werden kann. Diese “kurze 1" nennen wir Steuerimpuls oder **Steuersignal**. Die AND-Glieder fungieren als Tore. Bei $x = 0$, d.h. wenn eine Null gespeichert werden soll, wie in Bild 9.7 dargestellt, passiert das Steuersignal das rechte AND-Glied, und der Flipflop geht in den angegebenen Eigenzustand über bzw. verharrt in ihm. Am Ausgang des rechten NOR-Gliedes kann der Speicherinhalt und am Ausgang des linken NOR-Gliedes der negierte Speicherinhalt ausgelesen werden. Bei $x = 1$ wird bei Erscheinen des Steuersignals der andere Eigenzustand ausgelöst.

Die Schaltung von Bild 9.7 besitzt also diejenigen Eigenschaften, die von einem Ein-Bit-Speicher verlangt werden. Es gibt auch andere Schaltungsvarianten. Ersetzt man z.B. in Bild 9.6c die NOR-Glieder durch NAND-Glieder, bleibt die Flipflop-Eigenschaft erhalten. Eine verbreitete Realisierung des Ein-Bit-Speichers ist der **getriggerte NAND-Flipflop**. Er besteht aus vier NAND-Gliedern (vgl. z.B. [Matschke 86], S.97). Das Wort “*getriggert*” bedeutet, dass der Flipflop ein Eingangstor besitzt. Getriggerte Flipflops werden häufig als **Trigger** bezeichnet.

Über die elektronische Realisierung der booleschen Operatoren, des Flipflop und des Ein-Bit-Speichers ist bisher nichts gesagt worden. Sie wird im folgenden Kapitel behandelt. Aber auch ohne die Realisierungsmöglichkeiten zu kennen, ist es einleuchtend, dass sich durch Zusammenschalten elektronischer Ein-Bit-Speicher elektronisch arbeitende Computerspeicher bauen lassen. Es muss darauf hingewiesen werden, dass das Wort *elektronisch* in diesem Zusammenhang nicht ganz einheitlich

verwendet wird. Der Eindeutigkeit halber vereinbaren wir: Eine technische *Anordnung (eine Schaltung, ein Gerät) heißt* ***rein elektronisch****, wenn ihre Wirkungsweise ausschließlich auf der Bewegung und/oder Positionierung von Elektronen oder Ionen beruht.* 24

In der technischen Umgangssprache wird das Wort *elektronisch* in einem anderen, allgemeineren Sinne verwendet. Beispielsweise spricht man von *elektronischer Datenverarbeitung* auch dann, wenn sie magnetische oder optische Speicherverfahren einsetzt. Da im Weiteren ausschließlich auf rein elektronische Speicher eingegangen wird, sollen an dieser Stelle einige kurze Bemerkungen über magnetische und optische Speicher eingeschoben werden.

9.6 Einschub: Magnetische und optische Speicherverfahren

Große Bedeutung hatten in der Vergangenheit die sog. *Ferritkernspeicher*. Sie bestanden aus vielen kleinen Ringen aus Ferrit, *Ferritkerne* genannt. Ferrit ist ein ferromagnetisches Material. Die beiden Zustände maximaler Magnetisierung eines Ringes (rechtsherum bzw. linksherum) wurden zur Codierung eines Bit genutzt. Das Schreiben und Lesen erfolgte mittels Stromstößen durch Schreib- und Leseleitungen, auf welche die Ferritkerne aufgefädelt waren. Jeder Kern stellte einen Ein-Bit-Speicher dar

Während der Ferritkernspeicher praktisch vollständig durch rein elektronische Speicher verdrängt worden ist, verliert eine andere Variante der magnetischen Speicherung keineswegs ihre Bedeutung, die *Speicherung auf bewegtem Träger*. Die Methode verwendet zur Speicherung einzelner Bits winzige Oberflächenelemente einer dünnen Schicht aus ferromagnetischem Material, das auf Bänder oder Scheiben (Platten) aufgebracht ist. Die Träger werden in schnelle Bewegung versetzt (Rotation beim Magnetplatten- oder Diskettenspeicher, bzw. Umspulen beim Magnetband- oder Kassettenspeicher). Mittels eines Schreib-Lese-Kopfes lässt sich die Magnetisierung eines Elementes durch Induktion verändern (ein Bit einschreiben). Ebenfalls durch Induktion lässt sich die Magnetisierung eines am Kopf sich vorbeibewegenden Oberflächenelements feststellen, d.h. das gespeicherte Bit kann gelesen werden. Hierauf beruht die Arbeitsweise von *Kassetten-* und *Diskettenspeichern*. Ein Vorläufer der Magnetplatte war die *Magnettrommel*; die ferromagnetische Schicht bildete die Oberfläche eines rotierenden Zylinders.

Kassetten- und Diskettenspeicher haben den elektronischen und Ferritkernspeichern gegenüber den Nachteil relativ großer Zugriffszeiten zu einem bestimmten Speicherplatz, da Kopf und Datenträger (Band bzw. Diskette) zueinander positioniert werden müssen, was mechanisch erfolgt. Doch haben sie den Vorteil großer Speicherkapazität bei relativ niedrigen Kosten. Darum werden sie auch als *Massenspeicher* bezeichnet. Außerdem gestatten sie die räumliche Trennung von Datenträger und Laufwerk. Das Laufwerk ist Bestandteil des Computers und bewerkstelligt den

Antrieb und die Positionierung des Datenträgers. Mittels Kassetten oder Disketten können also große Datenmengen zwischen Rechnern über große Entfernungen transportiert werden. Das Laufwerk spielt dann die Rolle eines Ein-Ausgabegerätes. Diese Art des Datentransportes verliert durch die weltweite Computervernetzung zunehmend an Bedeutung.

Seit einigen Jahren setzt sich immer mehr eine optische Speichermethode durch. Dabei wird die magnetische Oberfläche durch eine optisch reflektierende Oberfläche ersetzt. In sie können mittels Laser Punkte eingebrannt werden (zu vergleichen mit dem Einbrennen oder Stanzen von Löchern in Papier), die einfallendes Licht schlechter reflektieren als die nichtverbrannte Oberfläche. Der Zustand der Oberfläche (glatt oder verbrannt) lässt sich durch Abtasten mit Hilfe eines Laserstrahls, der an der Oberfläche reflektiert wird, feststellen. Das Reflexionsvermögen dient als codierender Zustandsparameter. Der **CD-ROM-Speicher** (CD von Compact Disc) arbeitet nach diesem Prinzip. In der Massenproduktion werden CDs gepresst, ähnlich wie Schallplatten. CDs sind - ebenso wie Kassetten - als Musikträger allgemein bekannt. Der CD-ROM-Speicher ist, wie der Name sagt, ein reiner Lesespeicher. Seit einiger Zeit werden auch Schreib-Lese-CDs unter der Bezeichnung **CD-RW** (von ReWritable) angeboten. Die beiden codierenden Zustände einer Schreib-Lese-CD sind nicht "verbrannt" und "unverbrannt", wie bei der nur einmal "brennbaren" CD, sondern es sind zwei Materialzustände mit etwas unterschiedlicher polymorpher Struktur, deren Reflexionsvermögen sich minimal, aber messbar voneinander unterscheiden. Durch genau dosierte Erwärmung und Abkühlung können die Zustände ineinander überführt werden.

Alle Speicher, die als Speichermedium eine rotierende Scheibe verwenden, fassen wir unter der Bezeichnung **Scheibenspeicher** zusammen. Dazu gehört die Festplatte, die Diskette, die CD und CD-RW und als jüngstes Produkt, das auf dem Markt erschienen ist, die **DVD** (Digital Versatile Disk), eine Weiterentwicklung der CD mit bedeutend höherer Speicherkapazität, sodass sie für Videoaufzeichnungen verwendbar ist.

Auf magnetische und optische Speicher werden wir nicht weiter eingehen. Näheres über Speicherverfahren und Massenspeicher siehe z.B. [Werner 95], [Messmer 95], [Biaesch 92]. Wenn im Weiteren von der Komponierung der Speicherhardware die Rede ist, sind rein elektronische Speicher gemeint, soweit nichts Gegenteiliges gesagt ist.

10 Realisierungsprinzip zirkelfreier boolescher Netze

Zusammenfassung der Kapitel 10 bis 12

Die erste Grundidee des elektronischen Rechnens, also die Idee, die Computerhardware auf der booleschen Algebra aufzubauen, ist die technische Fortsetzung eines langen Denkweges der Menschheit. Die über 2000-jährige Entwicklung der Logik von der *Syllogistik* des ARISTOTELES bis zu den *logischen Schaltungen* eines Computers ist ein einzigartiges Beispiel dafür, wie Philosophie *unmittelbar* zu Technik wird. Gegenstand der Logik als eines Zweiges der Philosophie war von jeher das sprachliche Modellieren des deduktiven Denkens.

Die zweite Grundidee des elektronischen Rechnens besteht darin, die beiden Stellungen (statisch stabilen Zustände) eines Stromschalters als Eingabewerte eines booleschen Operators und die beiden (statisch stabilen) Zustände "Strom fließt" und "Strom fließt nicht" als Ausgabewerte des Operators zu interpretieren. Der Schalter ist dann die Realisierung (die "*Interpretation*" im Sinne der Interpretation eines Kalküls) entweder des Identitätsoperators oder des Negators (NOT-Operators) je nachdem, wie die Zuordnungen zwischen den Zuständen und den booleschen Werten getroffen werden. Das Hindernis, das der erfindende Geist überwinden muss, um auf diese Idee zu kommen, ist das Erfassen und geschickte Ausnutzen der Arbitrarität des Codierens.

Ersetzt man den Schalter durch zwei in Reihe geschaltete Schalter, so realisiert die resultierende Schaltung den AND- bzw. NAND-Operator. Ersetzt man ihn durch zwei parallele Schalter, ergibt sich der OR- bzw. NOR-Operator. Es lässt sich also ein vollständiger Satz boolescher Operatoren aus Schaltern realisieren.

Um größere Schalternetze und damit boolesche Kompositoperatoren komponieren zu können, müssen *sämtlichen* booleschen Variablen elektrische Größen entsprechen, deren Werte abgenommen (gemessen, weitergeleitet) werden können. Dieses Problem wurde durch die Erfindung geeigneter *Schaltermechnanismen* gelöst. In den ersten Computern kam der Relais-Mechanismus zur Anwendung, in der *ersten Generation* elektronischer Rechner kam der Röhren-Mechanismus und in der *zweiten Generation* der Transistor-Mechanismus zu Anwendung. Alle drei Mechanismen beruhen darauf, dass durch Anlegen einer geeigneten Spannung an einen "Schalter" die Bewegung von Ladungsträgern durch den Schalter freigegeben bzw. unterbrochen wird.

Beim Relais-Mechanismus wird die Bewegung von Elektronen durch einen metallischen Leiter dadurch unterbrochen, dass der Leiter aufgetrennt ("zerschnitten") wird, indem ein mechanischer Schalter mittels Elektromagneten geöffnet wird. Beim Röhren-Mechanismus wird die Bewegung von Elektronen durch das Vakuum von einer Elektrode zur anderen (von der Kathode zur Anode) dadurch unterbrochen,

dass zwischen den Elektroden eine Potenzialbarriere aufgebaut wird (durch Anlegen einer geeigneten Spannung), die von den Elektronen nicht überwunden werden kann. Der Transistor-Mechanismus unterscheidet sich vom Röhren-Mechanismus dadurch, dass die Ladungsträger (diesmal Elektronen oder Löcher) sich durch einen Halbleiter bewegen und dass die Potenzialbarriere nicht im Vakuum, sondern im Halbleitermaterial aufgebaut wird.

Jeder der drei Schaltermechanismen erlaubt es, Schalternetze ("*Schaltkreise*") zu komponieren. Auf diese Weise lässt sich jeder boolesche Operator und jede Kombinationsschaltung als zirkelfreies Schalternetz aus Relais, Röhren oder Transistoren realisieren. Elektronisch realisierte elementare boolesche Operatoren werden häufig **Gatter** genannt.

In Rechnern der *dritten Generation* sind Schalternetze vorzugsweise mikroelektronisch realisiert. Große Schaltungen werden aus Bausteinen "zusammengesteckt", die oft Dioden- bzw. Transistorenmatrizen darstellen. Eine Diodenmatrix ist ein Leitergitter, genauer ein Tupel paralleler Eingabeleiter, das von einem Tupel paralleler Ausgabeleiter senkrecht geschnitten wird. An einem Schnittpunkt eines Eingabe- und eines Ausgabeleiters kann sich eine Diode befinden, über die das Potenzial des Eingabeleiters (ein Eingabebit) an den Ausgabeleiter übergeben wird.

Zwei Matrixtypen haben besondere Bedeutung, Decodiermatrizen oder Wort-Leitung-Zuordner und Codiermatrizen oder Leitung-Wort-Zuordner. Der *Wort-Leitung-Zuordner* bildet eine Menge von Binärworten auf eine Menge von Leitungen ab. Wenn auf das Eingabeleitungstupel eines der Binärworte gegeben wird, gibt die zugeordnete Ausgabeleitung und nur diese eine 1 aus. Der *Leitung-Wort-Zuordner* bildet eine Menge von Leitungen auf eine Menge von ein Binärworten ab. Wenn auf eine und nur eine der Leitungen eine 1 gegeben wird, gibt das Ausgabeleitungstupel das zugeordnete Binärwort aus. Beide Matrixtypen finden breite Anwendung, beispielsweise beim Aufbau sehr großer Kompositweichen und Kommutatoren, speziell beim Aufbau der Ein- und Ausgangsweichen eines Schreib-Lesespeichers, RAM genannt (von Random Access Memory).

Durch Hintereinanderschaltung eines Wort-Leitung- und eines Leitung-Wort-Zuordners entsteht eine mikroelektronisch realisierte Kombinationsschaltung, die auch *Codeumsetzer* genannt wird. Auf diese Weise können Kombinationsschaltungen mit Hunderten oder gar Tausenden von Ein- und Ausgabeleitungen hergestellt werden. Die Operationstafel der Kombinationsschaltung wird durch "Einlöten" von Dioden (bzw. von Transistoren, die als Dioden arbeiten) in die beiden Leitergitter festgelegt. Dafür sind verschiedene Technologien entwickelt worden. Man spricht auch von "Prägen" oder "Programmieren" der Matrizen.

Ein Codeumsetzer kann nicht nur als realer Operator, sondern auch als Speicher verwendet werden, der nach dem sog. strukturellen Speicherprinzip arbeitet. Dabei stellen die den beiden Matrizen gemeinsamen Leitungen je einen adressierten Speicherplatz dar, auf den über die Dioden der ersten Matrix zugegriffen werden kann und dessen Inhalt durch die Dioden der zweiten Matrix bei deren Herstellung

"eingeprägt" ("einprogrammiert) wird. Ein solcher Speicher heißt (elektronischer) ROM (Read Only Memory), weil er nur einmal beschrieben, aber beliebig oft gelesen werden kann.

10.1 Die zweite Grundidee des elektronischen Rechnens

Über den Aufbau traditioneller Rechner existiert eine umfangreiche Literatur. Es kann nicht unsere Aufgabe sein, den vorhandenen Büchern ein weiteres hinzuzufügen. Unser erklärtes Ziel ist es, die zentralen Ideen aufzuzeigen, die das elektronische Rechnen ermöglichen. Der Leser soll das Gefühl bekommen, dass er, wenn er selber diese Ideen gehabt hätte, eigentlich auch den Taschenrechner oder den PC (Personal Computer) hätte entwerfen können. Wir beschränken uns also auf das Grundsätzliche und überlassen es dem Leser, sich hinsichtlich spezieller Fragen, die ihn interessieren, in der Literatur kundig zu machen[1].

Während unser Gegenstand selber traditionell ist, nämlich die *traditionelle* Rechentechnik und der *traditionelle* Rechner, der sog. **Prozessorcomputer**, ist die folgende Darstellung weniger traditionell. Denn sie wendet durchgängig die Terminologie der USB-Methode an. Das kann sich infolge der Besonderheiten einiger Begriffe hier und da etwas erschwerend auswirken, wenn man bei der Lektüre dieses Kapitels traditionelle Darstellungen zu Rate zieht bzw. die unterschiedlichen Darstellungen miteinander vergleicht. Dann kann das Glossar nützliche Dienste leisten.

Ohne weitere Umschweife kommen wir zum eigentlichen Kern des elektronischen Rechnens. Um das Grundprinzip zu verstehen, auf dem die Funktionsweise eines *elektronischen* Rechners letzten Endes beruht, reicht die Kenntnis des ohmschen Gesetzes aus. Wir wollen es auf die in Bild 10.1 dargestellten Schaltungen (a) und (b) anwenden. Gezeigt sind jeweils zwei fett gezeichnete waagerechte Leiterschienen. An der unteren Schiene liegt die Spannung U_0, an der oberen die Spannung U_1 an. Spannungsquellen und Masse (Leiter mit Nullpotenzial) sind nicht eingezeichnet. Die Schienen sind über einen festen Widerstand R und einen mit ihm in Reihe geschalteten Schalter S miteinander verbunden.

Der Schalter sei über die Eingabespannung x steuerbar, z.B. mit Hilfe eines Elektromagneten, wie es beim Relais der Fall ist. Bei $x{=}x_0$ sei S geöffnet und bei $x{=}x_1$ geschlossen. Man beachte, dass x_0 und x_1 *Namen* (Code) für zwei bestimmte Spannungswerte sind. In geschlossenem Zustand sei der innere Widerstand des Schalters zu vernachlässigen im Vergleich zu R. Die Spannung, die sich an der Verbindungsleitung zwischen dem Schalter und dem Widerstand einstellt, ist mit y bezeichnet und kann am Ausgang gemessen werden.

---

1 Siehe z.B. [Matschke 86], [Schiffmann 92], [Flik 94],[Hennessy 94], [Messmer 95], [Werner 95].

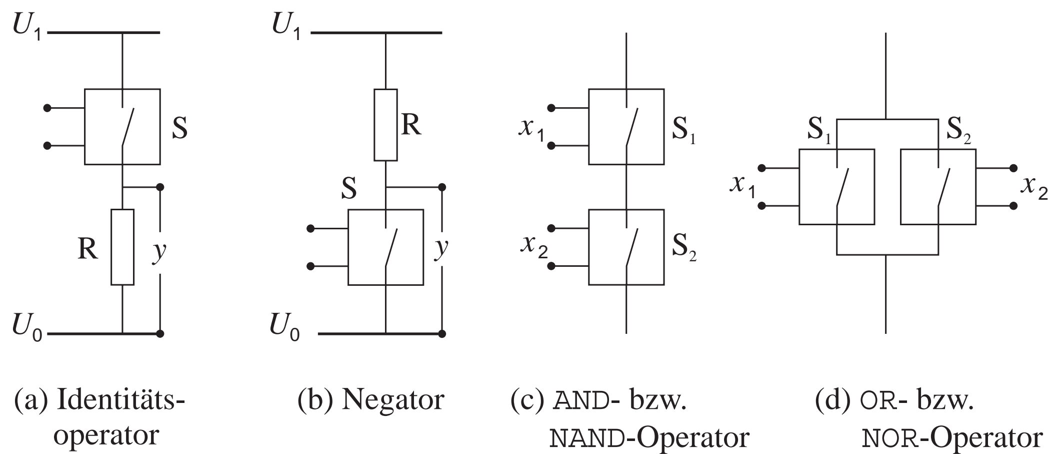


Bild 10.1 Realisierung boolescher Operatoren mit Hilfe von Schaltern. **(a)** Identitätsoperator; **(b)** Negator; **(c)** Kompositschalter des AND- bzw. NAND-Operators; **(d)** Kompositschalter des OR- bzw. NOR-Operators.

Die Schaltungen (a) und (b) stellen Operatoren dar, deren Wertetafeln sich leicht angeben lassen. Zunächst analysieren wir die Verhaltensweise der Schaltungen. Bei geöffnetem Schalter fließt kein Strom, sodass an der Ausgabeleitung in Schaltung (a) die Spannung U_0 und in Schaltung (b) die Spannung U_1 liegt. Bei geschlossenem Schalter kehrt sich die Situation um, denn es fließt ein Strom, welcher bewirkt, dass die Spannungsdifferenz U_1-U_0 am Widerstand R abfällt. Für Schaltung (a) gilt also: $y{=}U_0$, wenn $x{=}x_0$, und $y{=}U_1$, wenn $x{=}x_1$, während Schaltung (b) die umgekehrte Zuordnung realisiert.

Wir nehmen nun eine *Umcodierung* vor und zwar codieren wir x_0 und U_0 in 0 und x_1 und U_1 in 1 um, was wegen der Arbitrarität des Codierens erlaubt ist. Es mag im ersten Augenblick überraschen, dass die Arbitrarität so weit geht und dass man völlig unterschiedliche Inhalte wie z.B. U_0 und x_0 durch das gleiche Codezeichen benennen darf. Tatsächlich liegt dem eine Interpretation der boolesche Werte 0 und 1 durch elektrische Werte zugrunde. Mit der Umcodierung wird vorausgesetzt, dass die Interpretation möglich ist und eingehalten werden kann.

Mit der Umcodierung wird die Wertetafel der Schaltung (a) zu derjenigen des booleschen *Identitätsoperators* (f_3 und f_5 in Bild 9.1), während die Wertetafel der Schaltung (b) zu derjenigen des *Negators* (f_{10} und f_{12} in Bild 9.1) wird[2]. Erklärt man die umcodierten Spannungswerte 0 und 1 zu booleschen Werten, so bedeutet das eine *Interpretation* boolescher Größen durch physikalische Größen, in diesem Fall durch Spannungen.

2 Es gibt noch andere Möglichkeiten, den booleschen Werten Spannungswerte zuzuordnen. Beispielsweise kann U_0 zu 1 und U_1 zu 0 umcodiert werden. Weiterhin muss die Ausgangsspannung nicht gegen die untere, sie kann auch gegen die obere Schiene gemessen werden. Natürlich wirkt sich das auf die Funktionstafel aus.

Mit der elektrotechnischen Realisierung des Negators ist noch nicht viel gewonnen. Um beliebige boolesche Funktionen und somit die boolesche Algebra zu realisieren, muss ein vollständiger Satz boolescher Operatoren in entsprechende Schaltungen überführt werden. Zu diesem Zweck ersetzen wir den Schalter in Schaltung (a) durch einen *Kompositschalter*, der aus zwei Schaltern komponiert ist. Die Komponierung "*in Reihe*" bzw. "*parallel*" erfolgen gemäß der Schaltung (c) bzw. (d) in Bild 10.1. Wir wollen die Funktionsweise der resultierenden Schaltungen analysieren.

Zunächst ersetzen wir den Schalter der Schaltung (a) durch die Reihenschaltung (c) aus zwei Schaltern S_1 und S_2. Es ergibt sich ein Operator mit zwei Eingängen, dessen Wertetafel (bei entsprechender Codierung) mit der des AND-Operators identisch ist, denn Strom fließt nur dann, wenn S_1 *und* S_2 geschlossen sind, m.a.W. am Ausgang liegt nur dann der Spannungswert 1, wenn an *beiden* Eingängen der Spannungswert 1 liegt. Verfährt man in gleicher Weise mit Schaltung (b), ergibt sich der NAND-Operator.

Damit verfügen wir bereits über eine technische Realisierung eines vollständigen Satzes boolescher Operatoren. Aus praktischen Gründen ergänzen wir ihn noch um den OR- und NOR-Operator. Diese erhält man, wenn man den Schalter S der Schaltungen (a) bzw. (b) durch die Parallelschaltung (d) ersetzt. Im Falle der Schaltung (a) ergibt sich der OR-Operator, denn nun fließt immer dann Strom, wenn S_1 *oder* S_2 geschlossen ist (oder wenn beide geschlossen sind). Wird der Schalter in Schaltung (b) durch zwei parallele Schalter ersetzt, ergibt sich der NOR-Operator. Die Idee, boolesche Operatoren mittels Schaltern zu realisieren, bezeichnen wir als **zweite Grundidee des elektronischen Rechnens**.

10.2 Zirkelfreie Schalternetze

Um aus den Schaltungen von Bild 10.1 komplexere boolesche Operatoren zu komponieren, müssen sie in genügender Stückzahl hergestellt und zu **Schalternetzen** miteinander verbunden werden. Dazu ist es notwendig, die Ausgänge der Schaltungen elektrisch an die Eingänge anzupassen, was mit Hilfe von Spannungsteilern und/oder zusätzlichen Spannungsquellen möglich ist. Auf diese Weise lässt sich im Prinzip jede boolesche Funktion und jede Kombinationsschaltung realisieren. Da jede binär codierte Wertetafel in eine Kombinationsschaltung überführt werden kann, ergibt sich folgender Schluss: *Zu jeder **Binärwortfunktion** mit bekannter Wertetafel lässt sich ein **Schalternetz** angeben, das die Funktion realisiert.*

Die *Mikroelektronik* hat Technologien entwickelt, die es ermöglichen, Netze aus Millionen von Schaltern in einem Halbleiterkristall mit einer Oberfläche von größenordnungsmäßig 1 cm^2 zu implementieren. Charakteristisch für solche Netze ist ihre matrixförmige Struktur. Bevor wir auf diese Technologien eingehen, soll ein kurzer historischer Abriß eingefügt werden, der den mühevollen Weg andeuten soll,

den viele Generationen von Philosophen, Naturwissenschaftlern und Technikern zurücklegen mussten, um das Fundament der heutigen Computertechnik zu legen. Wohin der Weg führen würde, davon hatten die Beteiligten entweder gar keine oder nur eine sehr vage Vorstellung. Und auch wir wissen nicht, wohin der Weg geht.

11 Historischer Einschub: Entwicklung der begrifflichen und physikalischen Basis der Rechentechnik

11.1 Von der Syllogistik des Aristoteles zur Schaltalgebra

Die erste Grundidee des elektronischen Rechnens, also die Idee, die Computerhardware auf der booleschen Algebra aufzubauen, setzt einen Jahrtausende langen Denkweg fort. Sie ist der erste Schritt vom *natürlichen* (biologischen, organismischen) Denken zum *künstlichen*, "elektronischen Denken". Die über 2000-jährige Entwicklung der Logik seit der *Syllogistik* des ARISTOTELES (384 - 322 v.u.Z.) bis zu den *logischen Schaltungen* eines Computers ist ein einzigartiges Beispiel dafür, wie Philosophie *unmittelbar* zu Technik wird. Zwar lässt sich die *mittelbare* Einflussnahme der Philosophie auf die Naturwissenschaft und die Mathematik und damit auf die Technik durch die gesamte Geschichte der Wissenschaft verfolgen; dass aber ein Zweig der Philosophie zu Erkenntnissen, Formulierungen und Sätzen gelangt, die geradenwegs durch immer weitergehende Abstraktion und semantische Objektivierung schließlich in ein Kalkül übergehen und durch Interpretation des Kalküls zu technischen Produkten, zu elektronischen Schaltungen werden, das ist wohl einmalig in der kulturellen Evolution.

Die Ursache dafür ist verständlich. Seit Aristoteles ist das Ziel der Logik das Verstehen und Beschreiben des deduktiven Denkens. Inhalt der technischen Informationsverarbeitung ist das maschinelle Rechnen und Schließen, m.a.W. der technische Nachvollzug des deduktiven Denkens. Es ist also ganz natürlich, dass die Rechentechnik sich die Ergebnisse der Logik zunutze macht. In Kap.3 [3.2] hatten wir vereinbart, unter *Denken* das gedankliche Modellieren der Welt zu verstehen, und in Kap.7.1 [7.3] hatten wir die Fähigkeit zum deduktiven Modellieren (zum deduktiven Denken) als deduktive Intelligenz bezeichnet. Damit lässt sich die "Verwandtschaftsbeziehung" zwischen Logik und künstlicher Intelligenz schlagwortartig folgendermaßen charakterisieren: *Die Logik ist die **natürliche** Mutter der **künstlichen** deduktiven Intelligenz.*

Die Beziehung zwischen Logik und künstlicher Intelligenz wird besonders deutlich, wenn man den Gegenstand der Logik folgendermaßen bestimmt: *Gegenstand der Logik ist das sprachliche Modellieren des gedanklichen deduktiven Modellierens.* Aus dieser Formulierung folgt die Vorreiterrolle der Logik nicht nur für die KI-Forschung, sondern für jede exakte Wissenschaft. Außerdem spiegelt sich in ihr der zirkuläre Charakter (Modellieren des Modellierens) der Logik wider. Er entspricht dem zirkulären Charakter der Informatik (vgl. Kap.6.3 [6.5]) und tritt z.B. in dem Umstand zutage, dass die Logiker die Mechanismen der Begriffsbildung nicht

nur als Phänomen (als Operand) untersuchen, sondern auch als Denkmittel (als Operator) benutzen. Das betrifft insbesondere die abstrahierende Begriffsbildung durch Klassifikation und Generalisierung sowie die konkretisierende Begriffsbildung durch Instanzierung und Präzisierung (vgl. Bild 5.4). Ein anschauliches Beispiel dafür ist die Syllogistik des Aristoteles.

Damit beginnen wir einen kurzen Streifzug durch die Geschichte der Logik[1] unter einem bestimmten Aspekt, nämlich im Hinblick auf das uns interessierende Endergebnis: die semantische Objektivierung des Modells des deduktiven Denkens in Form eines Kalküls und dessen Interpretation durch Schalternetze, also durch Hardware. Der Weg dahin ist durch schrittweise Abstraktion und im letzten Schritt durch konkretisierende Interpretation gekennzeichnet. Wir werden einige wenige Marksteine dieses Weges beleuchten.

Die antiken Untersuchungen zum deduktiven Denken beziehen sich natürlicherweise auf das *Schlussfolgern* und weniger auf das *Rechnen*. Das bekannteste Beispiel für das von Aristoteles systematisch untersuchte schlussfolgernde Denken lautet: "*Alle Menschen sind sterblich. Sokrates ist ein Mensch. Also ist Sokrates sterblich*." Das Beispiel ist in vielen Lehrbüchern der Logik zu finden. Es stammt jedoch nicht von Aristoteles und ist für die von ihm analysierte Schlussweise auch nicht ganz zutreffend (s.u.).

Der Schluss beruht auf folgender Voraussetzung. Eine Eigenschaft, die eine Klasse von Individuen charakterisiert, trifft auf jedes einzelne Individuum zu (*Axiom des kategorialen Syllogismus*), oder unter Verwendung des Prädikatbegriffs: Ein Prädikat $P(x)$, das eine Klasse X festlegt, ist für jedes Exemplar (jedes Element, jede Instanz) der Klasse erfüllt. Die formale prädikatenlogische Notation lautet:

$$\forall x \in X \; P(x),$$

zu lesen als "Für alle x aus X ist das Prädikat $P(x)$ erfüllt". In der Terminologie der objektorientierten Programmierung (siehe Kap.18) würde man von Merkmalsvererbung sprechen und sagen: Das Merkmal "Sterblichkeit" wird von der Klasse der Menschen an ihre Instanzen (an jeden einzelnen Menschen) "*vererbt*".

In dem Beispiel beruht das Schließen auf Instanzieren. Zieht man aus der Prämisse, dass alle Menschen sterblich sind, den Schluss, dass alle Griechen sterblich sind, so liegt dem nicht *Instanzieren*, sondern *Präzisieren* zugrunde, denn das Merkmal wird an eine *Unterklasse* vererbt. Aristoteles selber hat das präzisierende, nicht das instanzierende Schließen untersucht.

Diese wenigen Bemerkungen zeigen, dass es sich bei der aristotelischen Syllogistik um eine *Klassenlogik* handelt. Interessanterweise setzt sich die älteste von den Philosophen untersuchte Schlussweise (nämlich die auf Merkmalsvererbung beruhende) erst in dem gegenwärtig jüngsten, dem *objektorientierten* Programmierpara-

---

1 Näheres zur Geschichte der Logik siehe z.B. in [Berka 83] oder [Störig 89].

digma durch. Früher entstandene Paradigmen machen von der Vererbung keinen expliziten Gebrauch. Es ist also eine Art Gegenläufigkeit der Entwicklung zu beobachten, die an die Gegenläufigkeit der Entwicklung der natürlichen und der künstlichen Intelligenz erinnert (vgl. Kap.7.1).

Das Wort "Logik" führten die Stoiker[2] ein. Aristoteles selber nannte die Wissenschaft vom Schlussfolgern *Analytik*. Ein wichtiger Beitrag der Stoa zur Logik ist die Benutzung der "Wenn...dann"-Wendung als eines standardmäßigen Mittels zur Artikulierung des gedanklichen Schließens (des bewussten logischen Schlussfolgerns). Wenn heutzutage ein Mensch erklären soll, was er unter *Schlussfolgern* (*Schließen* oder *Folgern*) versteht, wird er sich sehr wahrscheinlich dieser Wendung bedienen. Mit ihr werden sowohl *logische* Zusammenhänge (*Prämisse - Konklusion*) als auch *kausale* Zusammenhänge (*Ursache - Wirkung*) beschrieben. Der Beziehung zwischen beiden Arten von Zusammenhängen waren wir in Kap.8.2.5 auf den Grund gegangen.

Vorwegnehmend sei gesagt, dass "Wenn...dann"-Wendungen in der Programmierungstechnik eine wichtige Rolle spielen, beispielsweise in Form bedingter Anweisungen (siehe z.B. Bild 15.2b) oder in Entscheidungstabellen, die bei der automatischen Prozesssteuerung zum Einsatz kommen (siehe Kap.12.3.4 [12.5]). Dabei steht i.d.R. der *logische* Aspekt im Vordergrund. Auch in diesem Kapitel haben wir es nur mit logischen "Wenn...dann"-Zusammenhängen zu tun, also mit Zusammenhängen der Art: Wenn die Prämisse erfüllt ist (wenn die *Prämissenaussage* zutrifft), dann ist auch die Konklusion erfüllt (dann trifft auch die *Konklusionsaussage* zu). Man beachte, dass die Gültigkeit der Prämissen eine hinreichende, aber keine notwendige Bedingung der Konklusion ist.

In der antiken Philosophie gab es bereits Ansätze in Richtung eines Formalismus, der Schlussfolgern in Rechnen überführt. Doch erst Gottfried Wilhelm Leibniz (1646-1716) sprach die Idee eines Kalküls auf der Grundlage einer formalisierten Sprache des logischen Schließens explizit aus. Er wollte die *Wahrheit* einer Aussage *berechenbar* machen. Er schrieb: "*Das einzige Mittel, unsere Schlussfolgerungen zu verbessern ist, sie ebenso anschaulich zu machen, wie es die der Mathematiker sind, derart, dass man seinen Irrtum mit den Augen findet und, wenn es Streitigkeiten unter Leuten gibt, man nur zu sagen braucht: 'Rechnen wir!' ohne eine weitere Förmlichkeit, um zu sehen, wer recht hat*"[3].

Es stellt sich die Frage, welcher Zusammenhang zwischen einem solchen "Wahrheitskalkül", dem Schlussfolgern im Sinne von "Wenn...dann"-Zusammenhängen und der aristotelischen Klassenlogik besteht. Um ihn aufzuzeigen, formulieren wir das obige Beispiel folgendermaßen um:

2 Philosophen der als Stoa bezeichneten griechischen Philosophenschule (3. bis 1. Jahrh.v.u.Z.).

3 Zitiert nach [Berka 83].

> Wenn die Aussage “Alle Menschen sind sterblich” zutrifft UND wenn außerdem die Aussage “Alle Griechen sind Menschen” zutrifft, dann trifft auch die Aussage “Alle Griechen sind sterblich” zu.

Die Richtigkeit des Schlusses kann nach den Regeln des *Aussagenkalküls* berechnet werden, eines Kalküls, der den Intentionen von Leibniz wohl ziemlich nahe kommt, aber bedeutend jünger ist. Der Aussagenkalkül - auch *Aussagenlogik* oder *Aussagenalgebra* genannt - ist eine *Interpretation* der booleschen Algebra. Die booleschen Variablen werden als Platzhalter für Aussagen interpretiert und die booleschen Werte 0 bzw. 1 als “falsch” bzw. “wahr” [9.12].

Mit dieser Interpretation *berechnet* sich der Wahrheitswert der gesamten Prämissenaussage in obigem Beispiel aus den beiden mit UND verbundenen Prämissenaussagen nach der Wertetafel der Konjunktion (der AND-Operation). Daraus folgt mit Notwendigkeit die Gültigkeit der Konklusionsaussage, wenn beide Prämissenaussagen zutreffen (beide Bedingungen erfüllt sind). Die Gültigkeit der gesamten Prämisse ist aber keine notwendige Bedingung für die Gültigkeit der Konklusion. Die Griechen könnten auch dann sterblich sein, wenn nicht alle Menschen sterblich wären, sondern beispielsweise die Römer unsterblich. Prämisse und Konklusion sind einander nicht *äquivalent* im Sinne der booleschen Äquivalenz, sondern die Gültigkeit der Prämisse *impliziert* die Gültigkeit der Konklusion. Der “Wenn...dann”-Beziehung entspricht die boolesche *Implikation*.

Die formale Entsprechung zwischen boolescher Algebra und Aussagenalgebra ist ein Beispiel für *isomorphe* Algebren oder Kalküle. Grob gesagt besteht zwischen zwei Algebren (Kalkülen) dann eine **Isomorphie** (eine **isomorphe Abbildung**), wenn sowohl zwischen den Operationen der beiden Kalküle als auch zwischen den Wertemengen der beiden Kalküle *eineindeutige* (in beiden Richtungen eindeutige) Entsprechungen bestehen und wenn einander entsprechende Operationen einander entsprechende Resultate liefern.

Vielleicht sieht mancher Leser in der Isomorphie zwischen Aussagenalgebra und boolescher Algebra eine haarspalterische Selbstverständlichkeit. Eine solche Ansicht würde der geistigen Leistung von George Boole (1815-1864) jedoch nicht gerecht. Sein Verdienst ist es, von der Bedeutung der Wörter “wahr” und “falsch” und von der Bedeutung “Aussage” abstrahiert zu haben und so den Weg zu einer rein formalen, d.h. von jeder Semantik freien Algebra zu öffnen (historisch geht die Aussagenlogik der booleschen Algebra voraus). Einen derartigen Abstraktionsschritt als erster zu tun, erfordert Genialität. Im Nachvollzug scheint er fast selbstverständlich zu sein.

Die obige Umformulierung des Sterblichkeits-Beispiels weist auf eine andere Isomorphie hin, die zwischen Aussagenalgebra und Klassenlogik; üblicherweise spricht man von Isomorphie zwischen *Aussagenalgebra* und **Mengenalgebra** (auch *Mengenlehre* genannt)[4]. Die Mengenlehre wurde von Georg Cantor (1845-1918) entwickelt. Dabei entspricht der aussagenlogischen UND- bzw. ODER-Operation

die mengenlogische *Durchschnitts*-bzw. *Vereinigungs*-Operation. Das sei an folgendem Beispiel illustriert.

Die Menge aller Berlinerinnen ist die *Durchschnittsmenge* der Menge aller Menschen, die in Berlin zu Hause sind, und der Menge aller Menschen weiblichen Geschlechts. Sie ist durch das Prädikat "*ist in Berlin zu Hause* UND *ist weiblichen Geschlechts*" definiert. Ersetzt man das logische UND durch das logische ODER, wird durch das neue Prädikat die *Vereinigungsmenge* festgelegt, das ist die Menge aller Menschen weiblichen Geschlechts erweitert um die (im Vergleich dazu kleine) Menge aller Menschen, die in Berlin zu Hause sind.

Der Inhalt des vorangehenden Absatzes kann auf wenige Zeichen komprimiert werden, wenn man ihn in die Sprache der Prädikatenlogik (des Prädikatenkalküls) übersetzt ("umcodiert"). Bezeichnet man die beiden miteinander geschnittenen bzw. vereinigten Mengen mit M_1 und M_2 und die sie definierenden Prädikate mit P_1 und P_2 und verwendet für die Durchschnitts- bzw. Vereinigungs-Operation das übliche Symbol $\cap$ bzw. $\cup$, so lassen sich die Zusammenhänge des Beispiels folgendermaßen formal notieren ("UND" bzw. "ODER" ist durch "AND" bzw. "OR" ersetzt):

$$M_1 \cap M_2 = \{x\text{: } (P_1 \text{ AND } P_2)\} \text{ bzw.} \tag{11.1a}$$

$$M_1 \cup M_2 = \{x\text{: } (P_1 \text{ OR } P_2)\}. \tag{11.1b}$$

Die beiden geschweift geklammerten Ausdrücke legen je eine Menge fest. Die Zeichenkette $\{x\text{: } (...)\}$ ist zu lesen als: "Menge aller x, für die (...) zutrifft".

Falls dieser oder jener Leser mit der prädikatenlogischen Notation Mühe hat oder wenn er die "Umcodierung" nicht nachvollziehen will, weil er Formeln nicht liebt, verliert er nichts, denn die Umcodierung bringt keinerlei Erkenntnisgewinn. Das heißt nicht, dass die Prädikatenlogik überflüssig ist. Ihr Nutzen liegt, wie der Nutzen jedes Kalküls, in der semantischen Objektivierung (vgl. Kap.5.4 [5.6]) und darin, dass komplizierte Zusammenhänge durch Abstraktion leichter durchschaubar und handhabbar werden. Der Prädikatenkalkül operiert mit Prädikaten, also mit sprachlichen Ausdrücken, welche die Form von Aussagen besitzen und eine oder mehrere Variablen enthalten (vgl. Kap.8.3[8.18]).

Wir übergehen den vollständigen Beweis der Isomorphie zwischen Aussagen- und Mengenalgebra. Mit ihm wäre dann auch die Isomorphie zwischen Mengen- und boolescher Algebra gezeigt. Der Vollständigkeit halber sei nur erwähnt, dass der Negation die **Komplementbildung** entspricht. Die **Komplementmenge** einer Menge M enthält alle Elemente, die *nicht* zu M gehören. Beispielsweise ist die Komplementmenge zur Menge aller Griechen die Menge aller Nichtgriechen. Damit die

---

4 Der Begriff der Menge ist etwas allgemeiner als der der Klasse. Er schließt die Möglichkeit ein, die Elemente einer Menge nicht durch ein Prädikat (*intensional*), sondern durch Aufzählung (*extensional*) festzulegen. Der Klassenbegriff wird hier nicht im Sinne der axiomatischen Mengenlehre verwendet.

Menge der Nichtgriechen eindeutig festgelegt ist, muss eine sog. **Allmenge** existieren, von der die Menge M gewissermaßen "subtrahiert" wird. Die Rolle der Allmenge kann z.B. die Menge aller Menschen spielen.

Es gibt eine weitere Interpretationsmöglichkeit der booleschen Algebra und zwar die für die Rechentechnik ausschlaggebende, von der in Kap.10 die Rede war, die Interpretation durch die *Schaltalgebra*. Die boolesche Algebra wird zur Schaltalgebra, wenn die booleschen Operatoren als Schalter, genauer als Kompositschalter (vgl. Bild 10.1) und die booleschen Werte 0 und 1 als "Schalter geöffnet" bzw. "Schalter geschlossen" interpretiert werden. Auf der Isomorphie zwischen boolescher Algebra und Schaltalgebra beruhten die Überlegungen des Kapitels 10, beziehungsweise - historisch betrachtet - die Überlegungen förderten die Isomorphie zutage.

Aus heutiger Sicht ist es fast selbstverständlich, dass ein Ingenieur, der eine umfangreiche Kombinationsschaltung entwirft (beispielsweise zur Steuerung einer Waschmaschine), sich der booleschen Algebra bedient. Das war keineswegs selbstverständlich, bevor die Isomorphie zwischen boolescher Algebra und Schaltalgebra entdeckt worden war. Findige Ingenieure entwickelten sich ihre eigene "Schaltalgebra", um sich die Arbeit zu erleichtern. Es bedurfte einer erheblichen *intuitiven* Intelligenz, um die genannte Isomorphie zu *sehen*. CLAUDE ELWOOD SHANNON sah sie, derselbe Shannon, der später die Informationstheorie entwickelte.

Der Intuition Shannons muss eine gedankliche Annäherung logischer Begriffe und Relationen an schaltungstechnische Begriffe und Relationen durch Abstraktion bis hin zu ihrer teilweisen Identifizierung vorangegangen sein. Einer ähnlichen Annäherung oder *Konvergenz* unterschiedlicher Begriffe durch Abstraktion waren wir bereits in Kap.8.5 [8.38] begegnet, dort hinsichtlich der Begriffe *Algebra, Kalkül* und *algorithmisches System*. Wir hatten die *begriffliche Konvergenz* (die semantische Konvergenz unterschiedlicher Begriffe) durch Abstraktion als charakteristisches Phänomen des menschlichen Denkens erkannt und als *Konvergenzprinzip des Denkens* bezeichnet.

Betrachtet man die Entwicklung der Wissenschaft im Allgemeinen, fällt einem auf, dass sie ihre entscheidenden Fortschritte offenbar der begrifflichen Konvergenz verdankt. In diesem Sinne kann man auch vom *wissenschaftlichen Konvergenzprinzip* sprechen. Die Fortschritte, von denen hier die Rede ist, beruhen in der Regel auf genialen Ideen, die von dazu geeigneten Gehirnen hervorgebracht werden. Dabei handelt es sich immer um ein *Erfinden*, denn die Bedeutung des Wortes *Idee* schließt ein, dass diese nicht aus Bekanntem abgeleitet werden kann (vgl. die Definition in Kap.7.1 [7.4]). Solche Ideen führen zu neuen Erkenntnissen und zum Forschritt der Wissenschaft.

1

Ein anderes Beispiel für die Wirkungsweise des Konvergenzprinzips entnehmen wir der Physik. Im Denken JAMES MAXWELLS konvergierten durch Abstraktion die Begriffe und Relationen aus den Bereichen der Elektrizität einerseits und der Optik andrerseits. Das Ergebnis war eine einheitliche Theorie, artikuliert in den maxwell-

schen Gleichungen. Später werden wir weiteren Beispielen aus der Informatik begegnen.

Damit sind wir am Ende unseres Streifzugs durch die Logik angekommen, müssen aber noch eine Ergänzung anfügen. Wir haben nämlich von Boole zu Shannon einen allzu großen Sprung gemacht und des Mannes nicht gedacht, der die Idee eines "*Wahrheitskalküls*" als erster tatsächlich verwirklicht hat, nachdem Boole und viele andere Forscher wichtige Vorarbeiten geleistet hatten. Die Rede ist von FRIEDRICH LUDWIG GOTTLOB FREGE (1848-1929). Er hat eine formalisierte Sprache entwickelt, wie sie Leibniz vorgeschwebt haben mag, und **Begriffsschrift** genannt. Sie hat sich jedoch nicht durchgesetzt. Diejenige Algebra, die heute *boolesche Algebra* genannt wird und deren Grundlagen in Kap.9.3 skizziert wurden, hat ihre endgültige Ausformung erst durch die Nachfolger von Frege erhalten.

11.2 Rechnergenerationen

An die historische Skizze des *geistigen* (begrifflichen) Hintergrundes der Rechentechnik soll sich eine entsprechende Skizze der *materiellen* (physikalisch-technischen) Basis der elektronischen Rechentechnik anschließen. Zu diesem Thema existiert eine umfangreiche Literatur[5].

Die ersten *nicht* rein mechanisch arbeitenden Rechner waren *Relaisrechner*. Als Schalter dienten Relais, aus denen das *Rechenwerk* (die Rechenoperatoren) und der *Speicher* (die 1-Bit-Speicher) aufgebaut wurden. Der erste funktionstüchtige programmierbare Relaisrechner mit der Bezeichnung Z3 wurde von KONRAD ZUSE entwickelt und 1941 fertiggestellt. An der Weiterentwicklung von Relaisrechnern wurde etwa noch 10 Jahre hindurch gearbeitet, doch erreichte sie bald ihre Grenzen und zwar aus mehreren Gründen. Relais nehmen zu viel Platz in Anspruch, sie sind zu fehleranfällig, ihre Leistungsaufnahme und Wärmeabgabe ist zu hoch, und sie sind zu teuer. Dadurch, dass es gelang, Schalter zu entwickeln, die in all diesen Eigenschaften das Relais um das Millionenfache übertreffen, wurde die moderne Rechentechnik möglich.

Die Entwicklung vollzog sich in drei großen Schritten, die durch eine Reihe von Erfindungen ausgelöst wurden. Dabei brachte jeder Schritt eine enorme Miniaturisierung der Schalter und Schalternetze mit sich. Im ersten Schritt wurde das Relais durch eine Elektronenröhre, die Triode, und im zweiten durch ein Halbleiterbauelement, den Transistor ersetzt. Der dritte Schritt nutzte die Erfolge der Mikroelektronik, wodurch es möglich wurde, riesige **Schaltkreise** (sprich: Schalternetze) auf kleinen Halbleiterplättchen zu implementieren. Die drei Schritte unterteilen die Geschichte der Rechentechnik in drei Epochen und die Rechner in drei **Generationen**. Jede

5 Siehe z.B. [Hennessy 94] und die dort zitierte Literatur.

Generation	Zeitraum	Technologie	Größe/Aufstellungsart
(0.)	40er Jahre	Relais	Labor
1.	1950-1959	Röhren	Haus-Saal
2.	1960-1968	Transistoren	Saal-Zimmer
3.	1969-1977	Integrierte Schaltungen	Zimmer-Tisch
4.	seit 1978	LSI, VLSI	Tisch-Tasche

Bild 11.1 Rechnergenerationen

Epoche entwickelte ihre charakteristischen Bauelemente und ihre charakteristischen Technologien für den Aufbau von Schalternetzen.

Der erste elektronische, röhrenbestückte Universalrechner wurde während des zweiten Weltkrieges in den USA von J.Presper Eckert und John Mauchly gebaut. Der Rechner wurde ENIAC (Electronic Numerical Integrator and Calculator) genannt und von der Armee für militärische Zwecke verwendet. In den Entwurf der ENIAC flossen wichtige Ideen von John von Neumann ein. Eine Weiterentwicklung, die UNIVAC I, wurde 1951 als erster kommerzieller Rechner auf den Markt gebracht. Bei der folgenden kurzen Behandlung der Rechnergenerationen werden wir uns auf die Grundideen beschränken.

Ein elektrischer Schalter schließt oder unterbricht einen elektrischen Stromkreis, d.h. er ermöglicht bzw. unterbindet das Passieren beweglicher Ladungsträger, in der Regel freier Elektronen. Insofern stellt ein Schalter ein Tor für Elektronen dar.

Das Relais unterbricht den Strom dadurch, dass es den Leiter, durch den die Elektronen fließen, "zerschneidet", indem ein Elektromagnet einen Kontakt mechanisch trennt. Die Schalterfunktion der Triode und des Transistors beruht darauf, dass den sich bewegenden Elektronen bzw. "Löchern" (s.u.) eine Potenzialbarriere in den Weg gestellt werden kann, indem eine ortsfeste negative bzw. positive elektrische Ladung aufgebaut wird, welche die ankommenden Ladungsträger zurückstößt.

Im Falle der Triode bewegen sich Elektronen durch das *Vakuum* von der Kathode zur Anode. Die Barriere wird aufgebaut, indem ein Drahtgitter durch eine äußere Spannung, die sog. *Gitterspannung*, negativ aufgeladen wird. Da der Elektronenstrom auf seinem Wege zur Anode das Gitter passieren muss, kann er durch Anlegen einer ausreichend hohen negativen Gitterspannung unterbrochen werden. Die Triode gelangte in den dreißiger Jahren zur technischen Reife, wozu Walter Schottky wesentlich beitrug.

Im Falle des Transistors bewegen sich Elektronen durch das Kristallgitter eines *Halbleiters*. Als Halbleitermaterial hat sich z.B. Silizium als besonders geeignet erwiesen. Die ortsfeste Potenzialbarriere wird durch gittergebundene Ladungen, i.d.R. durch *Fremdatome* erzeugt, deren Ionisierungswahrscheinlichkeit (d.i. die über die Zeit gemittelte Differenz zwischen Kernladungzahl und Anzahl der Hüllenelektronen) durch eine äußere Spannung, verändert und so der Strom durch den Halbleiter gesteuert und gegebenenfalls unterbrochen werden kann. Das dosierte Einbringen (*Dotieren*) der Fremdatome kann mittels Diffusion bewerkstelligt werden.

Aus praktischen Gründen ist es vorteilhaft, in Halbleitern zwischen negativen und positiven freien Ladungsträgern zu unterscheiden; letztere werden auch *Löcher* genannt, weil sie gewissermaßen "bewegliche Löcher" oder "Blasen" in der strömenden "Elektronenflüssigkeit" darstellen. In diesem Bilde ist zwischen sog. *n*-Leitern, die negative Ladungsströme (Elektronenströme) tragen, und *p*-Leitern, die positive Ladungsströme (Löcherströme) tragen, zu unterscheiden.

In der Grenzschicht zwischen einem *p*- und einem *n*-Leiter bildet sich ein Potenzialsprung aus. Er wird in Halbleiterdioden zur Stromgleichrichtung ausgenutzt. Ein Transistor besteht aus drei Zonen; entweder liegt eine *p*-leitende Zone zwischen zwei *n*-leitenden oder eine *n*-leitende Zone liegt zwischen zwei *p*-leitenden Zonen. Der Transistor wurde in seiner ersten Variante (*Spitzentransistor*) 1947 von John Bardeen und Walter H. Brattain und in seiner zweiten Variante (*Flächentransistor*) 1948 von William B. Shockley erfunden. Alle drei erhielten 1956 gemeinsam den Nobelpreis.

Für die dritte Rechnergeneration lässt sich kaum eine einzelne Erfindung als Auslöser der Entwicklung angeben. Eine wichtige Rolle spielte die Erfindung und ständige Vervollkommnung der *Maskentechnik*. Sie macht es möglich, in einem Siliziumkristall äußerst feine räumliche Strukturen aus Zonen unterschiedlichen Leitungstyps und unterschiedlicher Leitfähigkeit herzustellen. Auf diese Weise lassen sich auf einem einzigen Kristallplättchen (*Chip*) viele Transistoren, Widerstände und andere Schaltelemente zu evtl. sehr großen Schaltkreisen *integrieren*. Die Technologie wurde ständig weiterentwickelt, aus *integrierten* Schaltkreisen wurden *LSI-Schaltkreise* (LSI von Large Scale Integration) und VLSI-Schaltkreise (Very Large Scale Integration).

Zuweilen wird die LSI- und VLSI-Technik als das bestimmende Charakteristikum der *vierten* Rechnergeneration angesehen (siehe Bild 11.1). Ein anderer Vorschlag verbindet die vierte Rechnergeneration mit der Einführung massiv paralleler Verarbeitungsprinzipien, auf die in diesem Buch nicht eingegangen wird. Die Japaner haben den Begriff der *fünften* Generation eingeführt, allerdings in einem Sinne, der von dem ursprünglichen Prinzip abweicht, eine Generation mit einer bestimmten Hardwaretechnologie zu verbinden. Derartige nicht nur an der Hardware orientierten Generationsbezeichnungen haben sich aber nicht durchsetzen können.

In der modernen Mikroelektronik kommt vorzugsweise der sog. MOS-Feldeffekttransitor, kurz MOS-FET zur Anwendung (MOS von Metal-Oxide-Semiconductor). 2

Die Steuerung des elektrischen Stroms durch einen MOS-FET kann mit der Steuerung des Wasserflusses durch einen Kanal mit leichtem Gefälle verglichen werden, der einen Gummiboden besitzt. Die pro Sekunde durchfließende Wassermenge kann dadurch vergrößert oder verkleinert wird, dass der Gummiboden nach unten oder oben ausgewölbt wird. Durch starke Wölbung des Bodens nach oben kann das Wasser vollständig aus dem Kanal verdrängt werden; der Kanal trocknet aus und es gibt nichts mehr, was fließen kann. Dem entspricht im MOS-FET das Verdrängen freier Ladungstäger aus einem leitenden "Kanal" in einem Halbleiter ("Semiconductor"). Wenn keine freien Ladungsträger mehr vorhanden sind, kann kein Strom mehr fließen. Zu erreichen ist dies durch Aufladung einer Elektrode, einer Metallschicht ("Metal"), die entlang des Kanals angebracht (aufgedampft) und gegen den Halbleiter durch eine Oxydschicht ("Oxide") isoliert ist. Dadurch wird ein elektrisches Feld aufgebaut, von dessen Größe die Anzahl der freien Ladungsträger und damit die Leitfähigkeit des Kanals abhängt.

Allgemein heißen Transistoren, deren Leitfähigkeit durch ein *äußeres Feld* gesteuert werden kann, das durch isolierte Ladungen erzeugt ist, *Feldeffekttransistoren*. Vor der Erfindung der Feldeffekttransistoren erfolgte die Steuerung ausschließlich durch ein *inneres* Feld, das durch Anlegen einer Potenzialdifferenz über leitende Verbindungen mit dem Halbleiter in diesem erzeugt wird.

Gemeinsam mit der Miniaturisierung durch die LSI- und VLSI-Technik erhöht sich die Arbeitsgeschwindigkeit elektronischer Schaltungen und speziell des Computers. Der Grund liegt in der Verkleinerung der Laufzeiten der Ladungsträger durch den Halbleiter, insbesondere durch die Bereiche großen Potenzialgefälles in den Grenzschichten. Denn von der Laufzeit hängt die *Bandbreite* des Transistors ab. Die Bandbreite charakterisiert, anschaulich ausgedrückt, die Reaktionsgeschwindigkeit des Transistors auf Änderungen der anliegenden Spannungen. Je schneller die Ladungsträger die Schicht durchqueren , umso kürzer ist die Schaltzeit des elektronischen Schalters, und umso kürzer ist die Zeitdauer, die ein Flipflop benötigt, um in seinen anderen Zustand überzuspringen. Diese Zeitdauer begrenzt die Anzahl der Bits, die ein Flipflop pro Sekunde scharf voneinander trennen kann und damit die maximale Taktfrequenz eines Schaltkreises bzw. Computers.

Für die Leistungsfähigkeit eines Computers ist neben seiner Taktfrequenz die Kapazität seines Arbeitsspeichers entscheidend. Die Arbeitsspeicher moderner Rechner sind in aller Regel VLSI-Schaltkreise. Auf ihre Arbeitsweise wird in Kap.13.2.2 eingegangen. Durch die fortscheitende Miniaturisierung, die noch nicht ihr Ende erreicht hat, steigt die Speicherkapazität elektronischer Speicher ständig. Parallel dazu läuft die Entwicklung neuer Speicherprinzipien. Gegenwärtig werden Speicherchips mit Milliarden von Schaltelementen produziert, und Speicherkapazitäten von über 10^8 Bit werden erreicht.

Neben den rein elektronischen [9.24] Speichern kommen auch magnetische und optische Speicherverfahren zur Anwendung. Die Grundprinzipien ihrer Funktionsweise waren in Kap.9.6 dargelegt worden. In der Speichertechnik der 1. und 2.

Rechnergeneration spielten magnetische Effekte die dominierende Rolle. Damals war die Herstellung rein elektronischer Speicher mit der erforderlichen Speicherkapazität zwar möglich, aber zu teuer. Rechner der 1. und 2. Generation waren mit unterschiedlichen Speichern ausgerüstet, häufig mit Trommelspeichern. Sie alle sind inzwischen aus der Computertechnik verschwunden. Das gilt auch für Ferritkernspeicher, die in Rechnern der 2. und der 3. Generation zum Einsatz kamen. Mit Ausnahme des Trommelspeichers finden Speicher mit bewegtem Träger auch heute noch breite Anwendung, allerdings nicht als *Arbeitsspeicher*, sondern als zusätzliche ("externe") *Massenspeicher* zur Aufbewahrung großer Datenmengen. Die relativ lange Zugriffszeit nimmt man dabei in Kauf[6].

6 Siehe z.B. [Messmer 95], [Werner 95].

12 Technische zirkelfreie boolesche Netze

12.1 Codeumsetzer und elektronischer Festwertspeicher

Nach diesem historischen Einschub nehmen wir den Faden wieder auf, den wir am Ende des Kapitels 10 unterbrochen haben, und wenden uns den technologischen Prinzipien der Mikroelektronik zu. Als Ausgangspunkt wählen wir ein Speicherproblem und fragen: Wie lässt sich die Wertetafel einer Binärwortfunktion maschinell verfügbar machen? Zwei unterschiedliche Lösungen bieten sich an. Zum einen kann die Wertetafel als Kombinationsschaltung realisiert werden [9.16]. Dann sprechen wir von **struktureller Speicherung**, weil die Wertetafel in der *Struktur* der Schaltung "gespeichert" ist. Zum anderen kann die Wertetafel in einem Speicher abgelegt werden, sodass auf die gewünschten Werte über die Adressen der jeweiligen Speicherplätze zugegriffen werden kann. Dann sprechen wir von **adressierter Speicherung**.

Fragt man sich, nach welchem Prinzip das menschliche Gedächtnis arbeitet, wird die Antwort offenbar zugunsten der strukturellen Speicherung ausfallen. In künstlichen neuronalen Netzen erfolgt das Erlernen einer Wertetafel durch langsame Veränderung der Synapsenleitwerte. Fasst man die Leitwerte als Merkmale der Netzstruktur auf, kann man von struktureller Speicherung sprechen. Es ist anzunehmen, dass das Gehirn, das natürliche Vorbild der künstlichen neuronalen Netze, ähnlich arbeitet. Das legt den Verdacht nahe, dass ein entscheidender Unterschied zwischen technischer und biologischer Informationsverarbeitung, zwischen Computer-IV und Human-IV im Speicherprinzip zu suchen ist. Es wird sich jedoch herausstellen, dass der Unterschied zwischen struktureller und adressierter Speicherung gar nicht so grundsätzlicher Art ist, wie es den Anschein haben könnte.

Als Zugang zu der nun zu besprechenden mikroelektronischen Technik des strukturellen Speicherns eignet sich das Problem des Umcodierens. Wir wissen bereits, dass sich die Funktionen eines Taschenrechners, der mit binär verschlüsselten Zahlen rechnet, in Form von Kombinationsschaltungen realisieren lassen. Wir wissen aber noch nicht, wie Dezimalzahlen in Dualzahlen bzw. wie Ziffern (bei ziffernweiser Codierung) in Bitketten umcodiert werden können. Ein Rechner mit dezimaler Ein- und Ausgabe muss über einen *Verschlüsseler* oder **Codierer** verfügen, der Ziffern zu Bitketten "verschlüsselt", d.h. in die computerinterne Darstellung *codiert*, sowie einen *Entschlüsseler* oder **Decodierer**, der Bitketten zu Ziffern "entschlüsselt", d.h. aus der internen in die externe Darstellung *decodiert*.

Wir wollen einen Codierer und einen Decodierer entwerfen. Der Codierer soll den Ziffern (Dezimalzahlen) 0 bis 9 die entsprechenden Dualzahlen zuordnen und der Decodierer soll den Dualzahlen (Bitketten, Binärwörtern) 0000 bis 1001 die jeweilige Dezimalzahl zuordnen. Der Codierer verfüge über 10 Tasten für die Ziffern*eingabe* (man denke an die Zifferntasten der Tastatur eines PC), und der Decodierer verfüge

über 10 Anzeigefelder für die Ziffern*ausgabe*, für jede Ziffer ein Feld. Der Codierer besitzt 4 Ausgabeleitungen und der Decodierer 4 Eingabeleitungen, für jede Stelle der Dualzahl (für jedes Bit des Binärwortes) eine Leitung.

Wir beginnen mit dem Decodierer und fragen, wie seine Schaltung aussehen könnte. Dazu vergegenwärtigen wir uns seine Aufgabe. Der Decodierer muss einem bestimmten Eingabewort eine bestimmte Leitung zuordnen, beispielsweise dem Eingabewort (der Dualzahl) 0101 diejenige Ausgabeleitung, die zum Anzeigefeld der 5 führt. Darum nennen wir eine solche Schaltung auch **Wort-Leitung-Zuordner**. Wer durch diese Bezeichnung an die Schaltung des Halbaddierers von Bild 9.4b erinnert wird, hat die Lösung unseres Problems vielleicht schon erraten (siehe [9.15]).

Bild 12.1a zeigt einen Wort-Leitung-Zuordner mit drei Ein- und drei Ausgabeleitungen. Über das Eingabeleitungstupel können Eingabewörter der Länge 3 Bit eingegeben werden. Jede Ausgabeleitung ist mit einem Anzeigefeld verbunden. Die Schaltung unterscheidet sich von dem Wort-Leitung-Zuordner in Bild 9.4b (dem linken Teil der dort dargestellten Schaltung bis einschließlich der AND-Glieder) in der Anzahl der Eingabeleitungen und darin, dass die NOT-Glieder, die sich ursprünglich vor den Eingängen der AND-Glieder befanden, an den Eingang der gesamten Schaltung vorgezogen sind, wodurch NOT-Glieder eingespart werden. Sie können ganz entfallen, wenn der Zuordner seine Eingaben von Ein-Bit-Speichern erhält, die zu jedem Wert auch dessen Negation liefern (vgl. Bild 9.7).

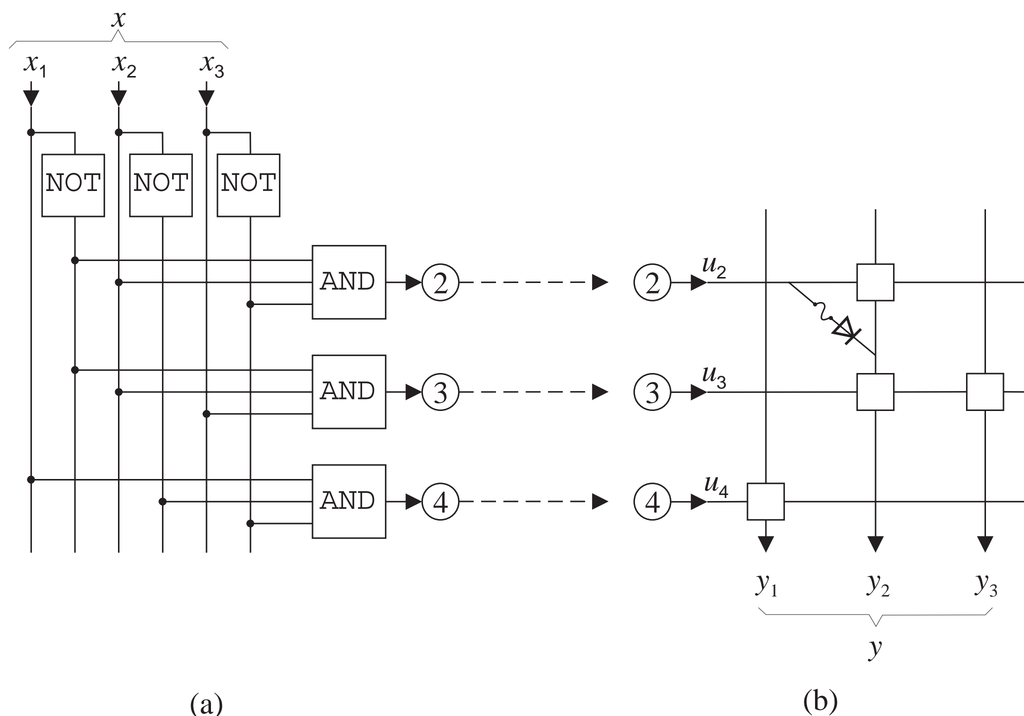


Bild 12.1 **(a)** - Wort-Leitung-Zuordner oder Adressierer; die Kreise stellen Anzeigefelder dar; **(b)** - Leitung-Wort-Zuordner; die Kreise stellen Eingabetasten dar.

Im Weiteren werden wir die durch die Ziffern 2, 3 und 4 gekennzeichneten waagerechten Leiter in Bild 12.1 als 2- bzw. 3- bzw. 4-Leiter bezeichnen[1]. Man verifiziert leicht, dass bei Eingabe des Wortes 010 bzw. 011 bzw. 100 der 2- bzw. 3- bzw. 4-Leiter eine 1 ausgibt, was die Anzeige der Ziffern 2 bzw. 3 bzw. 4 bewirkt. Wenn ein anderes Binärwort der Länge 3 eingegeben wird, führen alle Ausgabeleitungen eine 0. In diesem Sinne sagen wir, dass Bild 12.1a einen *partiellen* Wort-Leitung-Zuordner darstellt.

Ein Codierer hat die entgegengesetzte Aufgabe; er ordnet - verkürzt ausgedrückt - seinen Eingabeleitungen Binärwörter zu. Darum nennen wir eine solche Schaltung auch **Leitung-Wort-Zuordner**. Bild 12.1b zeigt einen Leitung-Wort-Zuordner mit drei Eingabeleitungen. Die Eingabe einer 1 auf den 2- bzw. 3- bzw. 4-Leiter bewirkt die Ausgabe des Wortes 010 bzw. 011 bzw 100. Um das zu erkennen, erinnere man sich an die Funktionsweise des *Schnittpunktoperators*. Sie ist in Kap 9.3 [9.14] erläutert worden, allerdings auf der booleschen Ebene, d.h. unter Bezugnahme auf Bild 9.4c. Die Erläuterung soll jetzt auf der technischen (elektronischen) Ebene wiederholt werden.

Die folgenden Überlegungen entsprechen denen von Kap.10, wo wir die booleschen Operatoren technisch als Schalternetze realisiert haben. Wir übernehmen die dortigen codierenden Zustandsparameter und ordnen der booleschen 0 den (niedrigen) Spannungswert U_0 und der 1 den (hohen) Spannungswert U_1 zu. Im Ruhezustand liege auf allen Leitungen die Spannung U_0. Bei Drücken einer Taste, z.B. der Taste 2 (in Bild 12.1b durch den Kreis mit der 2 im Eingabeleiter dargestellt), wird die Spannung am 2-Leiter von der Ruhespannung U_0 auf die Spannung U_1 angehoben. Die Spannung wird an den (senkrechten) y_2-Leiter weitergegeben. Die Übergabe erfolgt über eine Diode, sodass in der entgegengesetzten Richtung keine Spannungsweitergabe erfolgen kann. Jedes Quadrat in dem Leitergitter stellt eine solche Diodenverbindung dar; an einem der Schnittpunkte ist sie eingezeichnet (die Bedeutung des geschlängelten Leitungsstücks wird später erklärt). Das Gitter mit den eingebauten Dioden heißt **Diodenmatrix** und realisiert eine bestimmte Verbindungsstruktur. In der Praxis wird die Gleichrichterfunktion der Dioden häufig durch Transistoren realisiert, sodass sich eine *Transistorenmatrix* ergibt. Wenn im Weiteren von Diodenmatrizen die Rede ist, sind Transistorenmatrizen eingeschlossen.

Werden die Spannungen U_0 und U_1 als die Binärwerte 0 und 1 interpretiert, liefert die Matrix die genannte Verschlüsselung (Codierung). Davon überzeugt man sich durch "waagerechtes Lesen" der Matrix. Wenn z.B. der 3-Leiter eine 1 führt, übergibt er diese an den y_2- und y_3-Leiter, jedoch nicht an den y_1-Leiter. Es wird also das Binärwort 011 (die duale 3) ausgegeben. Man beachte, dass sich die den waagerechten Leitern zugeordneten Binärwörter unmittelbar aus der Matrix ablesen lassen,

1 Es sei daran erinnert, dass die Wörter *Leitung* und *Leiter* wie Synonyme verwendet werden, wenn von Leitungs- oder Leitermatrizen die Rede ist.

indem ein Leiterschnittpunkt ohne Quadrat als 0 und ein solcher mit Quadrat als 1 gelesen wird.

Man kann die Matrix auch “senkrecht lesen”. Dann ergeben sich die Bedingungen dafür, dass die senkrechten Leitungen eine 1 führen, zum Beispiel: Der y_2-Leiter führt eine 1, wenn der 2-Leiter ODER der 3-Leiter eine 1 führt (oder beide). Ein senkrechter Leiter realisiert also - ebenso wie die senkrechten Leiter des Halbaddierers - eine Disjunktion, in welche diejenigen Eingabevariablen (in Bild 12.1b mit u_2, u_3 und u_4 bezeichnet) eingehen, deren Leiter mit dem betreffenden senkrechten Leiter über eine Diode verbunden sind. Beispielsweise ergibt sich für y_2 die Disjunktion

$$y_2 = u_2 \texttt{ OR } u_3. \tag{12.1}$$

Die Codierungsfunktion des Leitung-Wort-Zuordners von Bild 12.1b ist *strukturell* gespeichert. Die Matrix kann erweitert werden, sodass umfangreichere Funktionstafeln gespeichert werden können. Um sämtliche Ziffern zu verschlüsseln (zu codieren), muss das Gitter mindestens 4, um Ziffern und Buchstaben zu codieren muss es mindestens 6 senkrechte Leiter besitzen. Damit ein Leitung-Wort-Zuordner als Codierer dienen kann, muss die Zuordnung eineindeutig (in beiden Richtungen eindeutig) sein. Diese Bedingung ist für die Matrixschaltung von Bild 9.4b, die ebenfalls einen Leitung-Wort-Zuordner darstellt, *nicht* erfüllt, denn der mittlere und der untere Leiter liefern die gleiche Bitkette.

Wir verbinden jetzt die Ausgabeleiter des Wort-Leitung-Zuordners mit den Eingabeleitern des Leitung-Wort-Zuordners über die gestrichelt gezeichneten Leiter. Das scheint wenig Sinn zu haben, denn die Decodierung hebt die vorangegangene Codierung wieder auf. Die Ausgabewörter sind mit den Eingabewörtern identisch. Die Schaltung berechnet die Identitätsfunktion, aber nur partiell, und zwar für die Argumentwerte 010, 011 und 100. Wird ein anderes Binärwort eingegeben, also 000, 001, 101 oder 111, liefert die Schaltung stets das Ausgabewort 000. Damit der Operator die *totale* (vollständige) Identitätsfunktion für Binärworte der Länge 3 berechnet, müsste die Anzahl der wagerechten Leiter auf 7 erhöht werden (ein Leiter für die Eingabe 000 kann entfallen, Taktung mittels Toren vorausgesetzt).

Der Leser hat den Zweck der soeben durchgeführten “sinnlosen” Verbindung von der linken zur rechten Schaltung in Bild 12.1 vielleicht schon erraten. Wenn nämlich die rechte Schaltung *nicht* die Umkehroperation der linken ausführt, sondern irgendeine andere Zuordnung, dann berechnet der Kompositoperator eine nichtidentische Binärwortfunktion, deren Wertetafel unmittelbar der Struktur der Matrixschaltung entnommen werden kann. Ein nächstes Aha-Erlebnis: Wir haben eine sehr durchsichtige Technologie zur Herstellung von Kombinationsschaltungen nacherfunden. Aber erst eine weitere Idee hat für die Computerherstellung den durchschlagenden Erfolg gebracht.

In Kap.12.2 werden wir erfahren, dass sich Matrizenstrukturen sehr kostengünstig herstellen lassen. Daraus ergibt sich der Wunsch der Hardwareproduzenten, nicht nur Leitung-Wort-Zuordner, sondern auch Wort-Leitung-Zuordner als Matrizen

aufzubauen. Die Idee, wie sich dieser Wunsch verwirklichen lässt, liegt gar nicht so fern, wenn man sich klar macht, dass zu diesem Zweck AND- in OR-Glieder überführt werden müssen, und dass diese Überführung nach der *morganschen Regel* möglich ist (siehe Bild 9.1 und Formel (9.6b). Für einen 3-stelligen AND-Operator lautet sie in verkürzter Notation (die Konjunktionssymbole sind unterdrückt):

$$x_1x_2x_3 = \texttt{NOT}(\texttt{NOT}x_1 \texttt{ OR } \texttt{NOT}x_2 \texttt{ OR } \texttt{NOT}x_3). \qquad (12.2)$$

Wir wollen zeigen, dass durch Anwendung dieser Formel die Schaltung von Bild 12.a in die linke, mit ADR bezeichnete Leitermatrix von Bild 12.2 überführt wird.

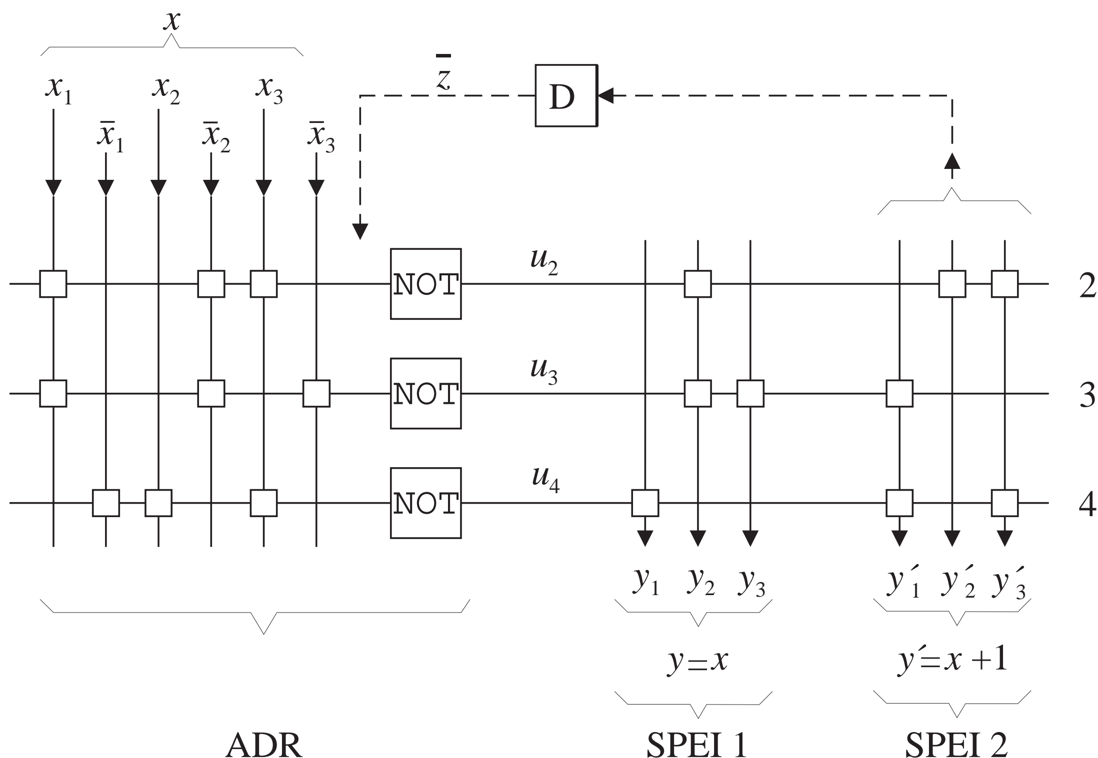


Bild 12.2 Codeumsetzer bzw. Festwertspeicher. ADR - Adressiermatrix; SPEI - Speichermatrix.

Bild 12.2 zeigt eine Schaltung aus drei hintereinandergeschalteten Diodenmatrizen, die mit ADR, SPEI1 und SPEI2 bezeichnet sind. ADR steht für Adressiermatrix, SPEI für Speichermatrix. Die Berechtigung dieser Bezeichnungen wird sich etwas später herausstellen. Die Matrizen besitzen gemeinsame waagerechte Leiter. Der Bitwert auf dem 2- bzw. 3- bzw. 4-Leiter sind mit u_2 bzw. u_3 bzw. u_4 bezeichnet. Die ADR-Matrix (die NOT-Glieder eingeschlossen) ergibt sich aus dem Wort-Leitung-Zuordner von Bild 12.1a bei Anwendung der morganschen Regel (12.2) auf jeden der AND-Operatoren. Dadurch werden die 3-stelligen Konjunktionen in negierte

3-stellige Disjunktionen überführt. Beispielsweise gilt für u_3 gemäß Bild 12.1a $u_3 = (\mathtt{NOT}x_1)x_2x_3$. Daraus wird bei Anwendung von (12.2)

$$u_3 = \mathtt{NOT}(x_1 \ \mathtt{OR} \ \mathtt{NOT}x_2 \ \mathtt{OR} \ \mathtt{NOT}x_3). \tag{12.3}$$

Diese Disjunktion ohne den NOT-Operator vor der Klammer lässt sich, wie wir wissen, als Leiter (diesmal als waagerechter Leiter) mit entsprechend platzierten Dioden (Schnittpunktoperatoren) realisieren, wie es in der ADR-Matrix von Bild 12.2 dargestellt ist. Auf die gleiche Weise ergeben sich die Platzierungen der übrigen Schnittpunktoperatoren der ADR-Matrix. Die bisher nicht berücksichtigten NOT-Operatoren sind in die Ausgabeleitungen der ADR-Matrix (des Wort-Leitung-Zuordners) eingefügt.

Wie man unschwer erkennt, führen bei jeder Eingabe (010, 110 oder 100) zwei der drei waagerechten Leiter vor dem NOT-Operator eine 1. Nach dem NOT-Operator führt eine einzige Leitung eine 1 und zwar diejenige, die auch in Bild 12.1a bei dem gleichen Eingabewort eine 1 führt. Da die SPEI1-Matrix mit der Matrix von Bild 12.1b identisch ist, liefert die Schaltung von Bild 12.2 (ohne die SPEI2-Matrix) das gleiche Ausgabetripel $y = x$ wie die gesamte Schaltung von Bild 12.1. Durch andere Platzierung der Diodenverbindungen können andere Zuordnungen "*eingelötet*" oder "*eingeprägt*" werden. Beispielsweise liefert die SPEI2-Matrix das Ausgabetripel $y' = x{+}1$, wobei die Tripel x und y' als Dualzahlen zu interpretieren sind. Sieht man von der Interpretation ab, kann die Operation, welche die ADR- und SPEI-Matrix gemeinsam ausführen, als *Umcodierung* aufgefasst werden. Aus diesem Grunde werden derartige Schaltungen **Codeumsetzer** genannt.

Obwohl die Bezeichnungen ADR und SPEI bisher nicht erklärt worden sind, haben sie beim Leser vielleicht ein weiteres Aha-Erlebnis ausgelöst. Die waagerechten Leiter in Bild 12.2 können als Speicherplätze aufgefasst werden. Durch das Eingabewort wird ein bestimmter Platz angewählt (*adressiert*). In diesem Sinne kann der Wort-Leitung-Zuordner als *Adressiermatrix* (ADR) aufgefasst werden. Die Eingabe einer Adresse x bewirkt die Ausgabe des "Inhalts" y der angewählten Speicherzelle, sodass der Leitung-Wort-Zuordner als *Speichermatrix* (SPEI) bezeichnet werden kann. Den Inhalt einer Speicherzelle kann man aus der SPEI-Matrix ablesen, indem man die Quadrate (die Symbole der Schnittpunktoperatoren) auf dem betreffenden waagerechten Leiter als Einsen und die übrigen Gitterpunkte als Nullen liest. Danach ist auf dem 3-Leiter das Wort 011 (die Ziffer 3) gespeichert. Der Inhalt eines Speicherplatzes ist durch die "eingelöteten" Dioden der Speichermatrix ein für allemal "strukturell" festgelegt; er kann nur ein einziges mal eingespeichert werden. Mit anderen Worten, die gesamte Schaltung (Adressier- und Speichermatrix) stellt einen **Nur-Lese-Speicher**, einen sogenannten **ROM** (**Read Only Memory**) dar, genauer einen **elektronischen ROM**.

Damit ist gezeigt, dass die Unterscheidung zwischen struktureller und adressierter Speicherung nur bedingte Bedeutung hat. Auch aus der Sicht der Anwendungsmöglichkeiten ist sie unwesentlich. In dieser Hinsicht ist die Unterscheidung zwischen

ROM und RAM wichtiger. **RAM** (**Random Access Memory**) ist die Kurzbezeichnung für **Schreib-Lese-Speicher**, das heißt für Speicher, auf deren Speicherzellen sowohl zum Zwecke des Einschreibens als auch des Auslesens zugegriffen werden kann. Sie werden auch als **Direktzugriffsspeicher** oder **Speicher mit wahlfreiem Zugriff** bezeichnet. In Kap.13.2 werden wir den Aufbau elektronischer RAM nacherfinden.

Zur ROM-Funktion hatte uns "waagerechtes Lesen" der Matrixschaltungen geführt. Waagerechtes Lesen allein der ADR-Matrix und ebenso "Senkrechtes Lesen" der Speichermatrix führt jeweils zu einer booleschen Funktion in Form einer Disjunktion, beispielsweise zur Disjunktion in (12.1), wenn man den y_2-Leiter "senkrecht liest". Ersetzt man in (12.1) die Variablen u_2 und u_3 durch die entsprechenden Konjunktionen, ergeben sich für die Ausgabevariablen boolesche Funktionen in der disjunktiven Normalform (DNF), beispielsweise für y_2

$$y_2 = (\texttt{NOT}x_1)x_2(\texttt{NOT}x_3)\ \texttt{OR}\ (\texttt{NOT}x_1)x_2x_3. \tag{12.4}$$

Wegen der Vorrangregeln (`NOT` vor `AND` vor `OR`) können die Klammern entfallen. Für die übrigen y-Leiter lassen sich entsprechende Ausdrücke angeben. Jeder Leiter liefert den Wert einer booleschen Funktion von drei Variablen. Die gesamte Schaltung stellt eine Kombinationsschaltung mit je drei Eingabe- und Ausgabeleitungen dar, m.a.W. einen Operator, dessen Ein- und Ausgabeoperanden dreistellige Bitketten sind. Wir sind zu einem Ergebnis mit großem praktischen Wert gelangt. Wir haben eine sehr durchsichtige Methode nacherfunden, Kombinationsschaltungen und damit auch elektronische ROM-Speicher mittels Diodenmatrizen zu realisieren.

Es gibt andere Möglichkeiten, reine Lesespeicher herzustellen. Es sei nur an das in Kap.11.2 erwähnte optische Speicherprinzip erinnert, auf dem der sog. CD-ROM basiert. Doch ist ein CD-ROM mit den Nachteilen des bewegten Trägers behaftet. Der rein elektronische ROM hat diesen Nachteil nicht. Dafür ist seine Speicherkapazität niedriger. Doch nimmt sie ständig zu, denn infolge der Perfektionierung der Herstellungstechnologie lassen sich immer größere Matrixschaltungen herstellen. Gleichzeitig sinkt der Preis pro Diode bzw. Transistor und damit pro Speicherplatz. Wir wenden uns nun dieser Technologie zu.

12.2 Herstellung von Diodenmatrizen

Wir beginnen mit einer Aufwandsabschätzung und fragen, wie viele waagerechte und senkrechte Leiter ein ROM etwa enthalten muss, wenn in ihm ein Deutsch-Englisches Wörterbuch mit 1000 Wörtern abgespeichert werden soll. Jedes Wort werde in einer gesonderten Speicherzelle, d.h. auf einem waagerechten Leiter abgespeichert. Der ROM enthält also 1000 waagerechte Leiter. Die Wortlänge sei in jeder Sprache auf 20 Buchstaben, d.h. auf 100 Bit begrenzt (5 Bit pro Buchstabe). Die Adressiermatrix enthält also 200 und die Speichermatrix 100 senkrechte Leiter.

Damit ergeben sich 300*1000 = $3*10^5$ Schnittpunkte, die jedoch bei weitem nicht alle mit Dioden zu besetzen sind.

Die Mikroelektronik ermöglicht die Implementierung eines solchen Wörterbuches auf einem einzigen Chip, die Negatoren in den waagerechten Leitern eingeschlossen. Das Wörterbuch darf auch durchaus noch umfangreicher sein, und es können zusätzliche Schaltungen integriert werden, z.B. ein Eingabecodierer (zur Umcodierung der Buchstaben in Bitketten) und ein Ausgabedecodierer.

Das Wörterbuch steht hier als Beispiel für die unterschiedlichsten Einsatzmöglichkeiten in Industrie, Wirtschaft, Büro und Haushalt. In Kap.9.2 [9.7] war bereits auf die vielfältigen Verwendungsmöglichkeiten von Kombinationsschaltungen in Auskunftssystemen hingewiesen worden. Dadurch wird verständlich, welchen Druck der Markt auf die Entwicklung kostengünstiger Technologien zur Herstellung derartiger integrierter Schaltungen ausübt. Die Ergebnisse sollen in aller Kürze dargestellt werden.

Abgesehen von der Perfektionierung der Bearbeitung des Halbleitermaterials (Chipherstellung, Beschichtung, Maskentechnik, Ätzung, Diffusion u.s.w.) brachte folgende Idee eine erhebliche Kostensenkung. Man stelle zunächst eine vollbesetzte Matrix her, die an jedem Schnittpunkt eine Diode enthält, und erzeuge nachträglich die gewünschte Struktur durch Stillegung der überflüssigen Dioden. Da es für die Herstellungskosten relativ unwesentlich ist, wieviele Bauelemente eine Matrix enthält und andrerseits die Stillegung einfach und billig zu bewerkstelligen ist, ergibt sich die Möglichkeit einer äußert profitablen Massenproduktion bei gesichertem Absatz.

Das Stilllegen (*Ab*schalten) ausgewählter Bauelemente kann z.B. durch Wegätzen von Leitern mittels der Maskentechnik erfolgen oder durch Wegschmelzen durch geeignet dosierte Ströme. In Bild 12.1b ist das zu schmelzende Leiterstück geschlängelt (als Schmelzsicherung) dargestellt. Auch der umgekehrte Weg ist möglich. Es wird eine vollbesetzte Matrix “stillgelegter” Transistoren hergestellt, die in *beiden* Richtungen als *Nichtleiter* wirken. Durch eine geeignete Spannung kann ein innerer Durchschlag hervorgerufen werden, wodurch der Transistor in eine funktionsfähige Diode überführt wird (*an*geschaltet wird).

Das Strukturieren - man sagt auch Prägen oder Programmieren - der Matrix nach dem Durchschmelz- bzw. Durchschlagverfahren erfolgt durch Anlegen einer vorgeschriebenen Spannung an diejenigen Matrixelemente, die ab- bzw. angeschaltet werden sollen. Das Programmieren kann vom Produzenten, aber auch vom Nutzer vorgenommen werden. Man spricht dann von einem **programmierbaren ROM**, abgekürzt: **PROM**.

Nach den genannten Methoden lässt sich ein PROM nur ein einziges mal programmieren. Der Wunsch nach wiederholter Programmierung führte zur Entwicklung des **löschbaren PROM**, abgekürzt: **EPROM** (von erasable PROM). Eine gängige Variante des EPROM verwendet spezielle Transistoren, die sogenannten *Floating-Gate-MOS-FET*. Sie ersetzten die Schmelzsicherung in Bild 12.1b.

Der MOS-FET war in Kap.11.2 [11.3] beschrieben worden. Der Zusatz “Floating-Gate” zeigt an, dass die steuernde Elektrode entlang des Leitungskanals keinen äußeren Kontakt besitzt, sondern sich innerhalb der isolierenden Schicht befindet. Auf diesem isolierten Leiterstück, dem “floating gate”, kann durch Anlegen einer ausreichend hohen Spannung eine Ladung induziert werden (Durchtunnelung der isolierenden Schicht). Ist die Ladung so groß, dass sie alle freien Ladungsträger aus dem Kanal verdrängt, ist die Leitung unterbrochen, was einem geöffneten Schalter oder einer durchgeschmolzenen Sicherung entspricht. Der Isolationswiderstand des isolierenden Materials ist so hoch, dass die Ladung über Jahre bestehen bleibt. Doch kann er durch ultraviolettes Licht so stark herabgesetzt werden, dass die Ladung im Laufe von Minuten abfließt. Auf diese Weise ist es möglich, einen EPROM durch Bestrahlen mit UV-Licht durch ein Fenster in der Verkapselung zu löschen.

Ein Codeumsetzer wird unterschiedlich bezeichnet je nachdem, welche Matrix vom Nutzer programmierbar ist. In einem **ROM**-Baustein ist *keine* Matrix programmierbar; in einem **PROM**- und einem **EPROM**-Baustein ist die Speichermatrix programmierbar; in einem **PAL**-Baustein (von Programmable Array Logic) ist die Adressiermatrix programmierbar und in einem **PLA**-Baustein (von Programmable Logic Array) sind beide Matrizen programmierbar.

12.3 Anwendungen von Diodenmatrizen

12.3.1 Elementare Hardware-Software-Schnittstelle

Wir kehren noch einmal zu dem Begriffspaar *Wort-Leitung* zurück und unterziehen es einer kritischen Analyse. In Kap.12.1 hatten wir uns Schaltungen für den Decodierer und den Codierer überlegt und sie *Wort-Leitung*-Zuordner bzw. *Leitung-Wort*-Zuordner genannt, ohne daran Anstoß zu nehmen, das die Begriffe “Leiter” und “Wort” verschiedenen Bereichen, verschiedenen Ebenen des sprachlichen Modellierens angehören. Leiter sind Elemente der Hardware, Wörter sind Elemente der Software. Diese Ungereimtheit wird noch auffälliger, wenn man “Wort” durch “Adresse” oder durch “Name” ersetzt. Mit “Name” ist gemeint, dass ein Leiter durch ein Binärwort “*benannt*” wird. Tatsächlich zeigt die “Ungereimtheit” an, dass sich in den beiden Schaltungen Hardware und Software treffen, mit anderen Worten: ***Wort-Leitung-Zuordner** und **Leitung-Wort-Zuordner** (Decodierer und Codierer) stellen **Schnittstellen** zwischen Hardware und Software auf einer sehr elementaren Ebene dar*. Diese Feststellung ist unüblich, trifft aber den Kern, wenn man davon ausgeht, dass die Hardware mit Bauelementen und elektrischen Größen, die Software dagegen mit Zeichen zu tun hat, oder allgemeiner, dass die Hardware mit Elektronik und die Software mit Sprache zu tun hat.

Tatsächlich ist die Trennung von Hardware und Software eine Frage des Betrachtungsstandpunktes. Das wird deutlich, wenn man bedenkt, dass die Hardware als *Träger* und die Software als *Beschreibungsmittel* informationeller Prozesse dient und

dass jedem Beschreibungselement Merkmalswerte von Trägerelementen entsprechen, wobei die Entsprechung sehr kompliziert sein kann. Aber wie undurchsichtig sie auch sein mag, letzen Endes wird sie durch Wort-Leitung-Zuordner und Leitung-Wort-Zuordner (Decodierer und Codierer) bewerkstelligt.

Ob man ein informationelles System mehr aus der Sicht des Elektronikers oder der des Programmierers betrachtet, hängt davon ab, wieweit man den *sprachlichen Überbau* der elektronischen Schaltungen bzw. den *elektronischen Unterbau* der sprachlichen Ausdrücke im Auge hat. Je höher man in der Softwarehierarchie aufsteigt, umso mehr abstrahiert man zwangsläufig vom Hardware-Unterbau.

12.3.2 Multiplexer, Demultiplexer und Bus

Mit Hilfe der Adressiermatrix (des Decodierers, des Wort-Leitung-Zuordners) lassen sich Mehrfachweichen aufbauen, das sind Weichen mit mehreren Ein- bzw. Ausgängen. Wie wir wissen, können Weichen aus Toren komponiert und Tore als AND-Glieder verwirklicht werden. Die Bilder 12.3a und b zeigen die Realisierung der einfachen Sammel- bzw. Zweigeweiche mittels AND-Gliedern. Die Sammelweiche gibt in dem Moment, in dem an einer der beiden Steuerleitungen ein Steuerimpuls (d.h. kurzzeitig der Spannungswert, der dem booleschen Wert 1 entspricht) erscheint, den Wert, der auf der durch das Steuersignal "angewählten" Eingabeleitung liegt (der Spannungswert bzw. der ihm entsprechende boolesche Wert) an die gemeinsame

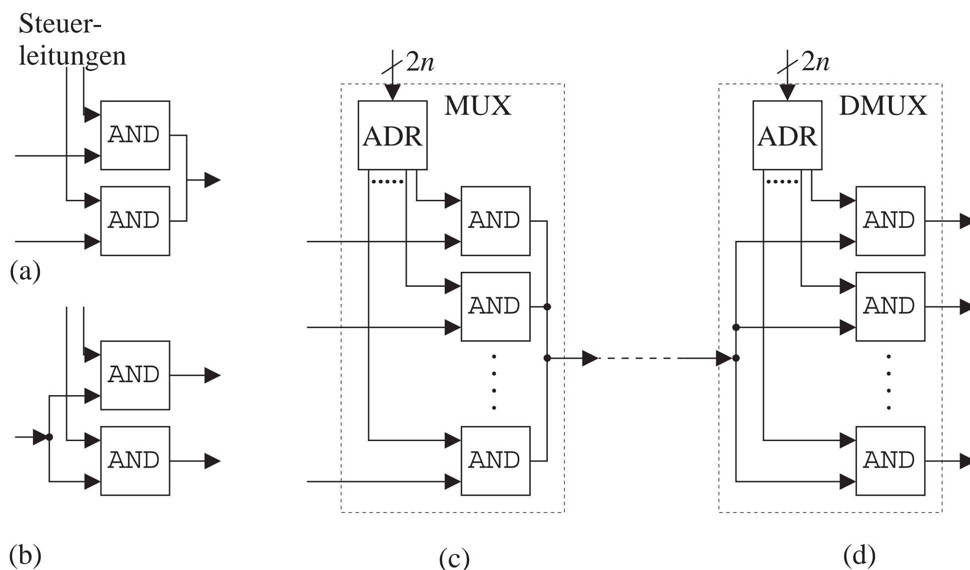


Bild 12.3 Weichen; **(a)** - einfache Sammelweiche; **(b)** - einfache Zweigeweiche; **(c)** - Multiplexer (MUX); **(d)** - Demultiplexer (DMUX)

Ausgabeleitung weiter. Im Falle der Zweigeweiche wird der Wert der gemeinsamen Eingabeleitung an die "angewählte" Ausgabeleitung weitergegeben.

Nachdem wir die Adressiermatrix (im Weiteren ADR-Baustein genannt und mit ADR bezeichnet) nacherfunden haben, fällt es nicht schwer die Schaltungen der Einfachweichen auf Mehrfachweichen zu erweitern. Wenn z.B. eine Zweigeweiche mit m Ausgängen gesteuert werden soll, muss einer von m Leitern "angewählt" werden, wofür ein ADR-Baustein eingesetzt werden kann. Bild 12.3c bzw. d zeigt die Verwirklichung dieser Idee für eine mehrfache Sammel- bzw. Zweigeweiche. Die Schaltungen heißen **Multiplexer** (abgekürzt **MUX**) bzw. **Demultiplexer** (**DMUX**).

Über den Steuereingang (Eingang des ADR-Bausteins) wird ein **Steuerwort** eingegeben, das den anzuwählenden Leiter "öffnet". Das Steuerwort kann als *Name* oder *Adresse* des Leiters aufgefasst werden. Der kurze, schräge Querstrich durch die Steuerleitung mit der Angabe "$2n$" bedeutet, dass es sich um ein Leitungsbündel aus $2n$ Einzelleitungen handelt. Der Faktor 2 zeigt an, dass sowohl die Adresse als auch deren Negation eingegeben wird. Da sich mit einer n-stelligen Adresse 2^n Datenleitungen adressieren lassen, ist bei vorgegebener Anzahl m von Datenleitungen eine Adresslänge $n \geq \mathrm{ld} m$ erforderlich. Multiplexer und Demultiplexer können als **Kompositflussknoten** aufgefasst werden, denn sie lassen sich, wie man leicht erkennt, aus den elementaren Flussknoten von Bild 8.2 komponieren.

Wir verbinden jetzt die Ausgabeleitung des Multiplexers mit der Eingabeleitung des Demultiplexers (in Bild 12.3 durch die gestrichelte Linie angedeutet). Es entsteht eine Konfiguration von Leitungen, über welche Verbindungen zwischen verschiedenen *Sendern* (links) und *Empfängern* (rechts) hergestellt werden können. Die Leitungskonfiguration stellt eine **einkanalige**, **gerichtete Kommunikationsstruktur** dar. *Als* ***Kommunikationsstruktur*** *bezeichnen wir jede Konfiguration von Verbindungen, über welche verschiedene "Teilnehmer" kommunizieren können.* Die vorliegende Kommunikationsstruktur heißt gerichtet, weil Verbindungen nur in einer Richtung (von links nach rechts) möglich sind. Sie heißt *einkanalig*, weil jeweils nur eine einzige Verbindung hergestellt werden kann.

Die Wörter "Nachricht" und "Teilnehmer" hat der Leser sicher richtig interpretiert, obwohl sie etwas unvermittelt verwendet worden sind. Sie zeigen an, dass wir in ein anderes Gebiet übergewechselt sind, in die Kommunikationstechnik. Das ist kein Zufall, denn die Datenübergabe in einem Rechner einerseits und die Nachrichtenübertragung in Kommunikationsnetzen andrerseits stellen die gleichen Probleme, und diese werden nach weitgehend gleichen Prinzipien gelöst. Das gilt auch für die Datenübertragung zwischen Computern, die zu einem sog. **Rechnernetz** miteinander verbunden sind. Demzufolge ist es durchaus möglich und sogar üblich, für die Datenübertragung in Rechnernetzen vorhandene *Tele*kommunikationsnetze zu benutzen.

Die Entwicklung ist soweit fortgeschritten, dass sich kaum noch eine Grenze zwischen Telekommunikationsnetz und Rechnernetz ziehen lässt, denn die "Endstationen", die sog. *Terminals*, d.h. die Geräte, derer sich die Abonnenten eines Kommunkationsnetzes bedienen (den Telefonapparat eingeschlossen) enthalten in

zunehmendem Maße kleinere oder größere Computer. Ein weltumspannendes Kommunikations- und Rechnernetz nimmt immer mehr Gestalt an[2]. Die Menschheit ist dabei, das "Nervensystem" der Informationsgesellschaft aufzubauen. Die inzwischen allgemein bekannte Abkürzung **WWW** für **World Wide Web** ("weltweites Gewebe") ist Nomen und Omen.

Nach dieser Abschweifung kehren wir zu den *gerichteten* Kommunikationsstrukturen der Bilder 12.3c und d zurück. Um Verbindungen in *beiden* Richtungen herstellen zu können, muss an jedem Ende ein MUX-DMUX-Paar vorhanden sein. Ein solches Paar nennen wir **Halbkommutator** (abgekürzt HK). Die sich ergebende Kommunikationsstruktur ist in Bild 12.4a gezeigt, wobei nicht nur die Steuerleitungen, sondern auch die Datenleitungen zu Bündeln zusammengefasst sind. Durch Steuerung der vier als kleine Kreise dargestellten steuerbaren Flussknoten wird die Richtung der Datenübertragung bestimmt.

Bild 12.4b zeigt die Kommunikationsstruktur von Bild 12.4a, wobei ein Halbkommutator als (etwas größerer) Kreis dargestellt ist, der eine **ungerichtete Mehrfachweiche** symbolisiert, die durch eine - evtl. recht lange - Bitkette (Adresse) gesteuert wird. Die Quadrate symbolisieren die kommunizierenden Teilnehmer. Der Begriff des Teilnehmers ist in einem verallgemeinerten Sinne zu verstehen. Ein

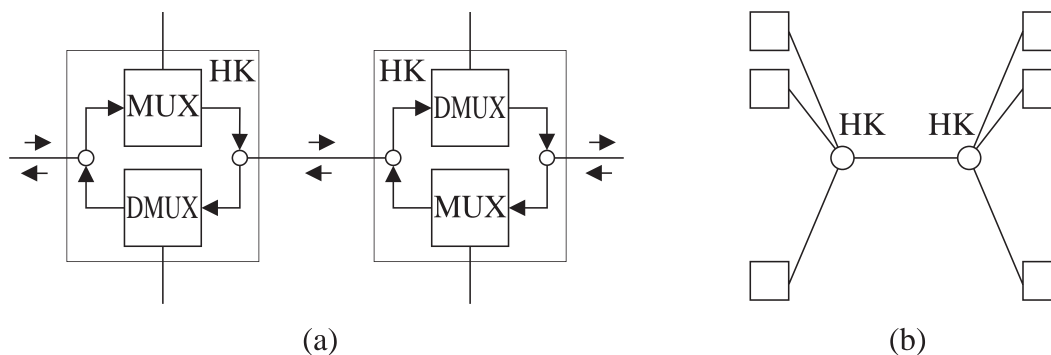


Bild 12.4 Verbindung über Halbkommutatoren (HK)

"Teilnehmer" kann ein Fernsprechteilnehmer sein, er kann ein Computer sein (im Falle eines Rechnernetzes), er kann ein Speicher oder auch ein Speicherplatz sein, wie wir gleich sehen werden.

Geht man davon aus, dass die linke Teilnehmermenge in Bild 12.4b mit der rechten identisch ist, und vereinigt man die Halbkommutatoren zu einem einzigen Knoten, entsteht eine sternförmige Kommunikationsstruktur, die wir **Vollkommutator** nennen (präziser müssten wir Einkanal-Vollkommutator sagen). Über ihn kann jedes beliebige Teilnehmerpaar kommunizieren, aber jeweils nur ein einziges Paar.

2 Näheres siehe z.B. in [Tanenbaum 98],[Lockemann 93].

In konkreten Anwendungsfällen ist es oft nicht erforderlich, dass jedes Paar verbunden werden kann. Aus diesem Grunde führen wir einen verallgemeinerten Kommutatorbegriff ein und vereinbaren:

Ein **Kommutator** *ist eine sternförmige Kommunkationsstruktur, über die zwischen jeweils zwei Teilnehmern Nachrichten übertragen werden können, nicht unbedingt zwischen allen möglichen Paaren und nicht unbedingt in beiden Richtungen.* Wenn im Weiteren von Kommutator die Rede sein wird, ist stets ein *einkanaliger* Kommutator gemeint. 2

Der Begriff des Kommutators und seine Darstellung als Stern kann bei der Analyse und Synthese komplizierter Kommunikationssysteme zunächst sehr hilfreich sein, selbst dann, wenn man den Kommutator (die *Kompositweiche*) nachträglich in *Bausteinweichen* dekomponiert und die Bausteine in Richtung Peripherie verlegt, wodurch eine verästelte Verbindungsstruktur entsteht. Sie ist für Telefon- und Rechnernetze typisch, aber auch für interne Verbindungsstruktur eines Computers. Vorgreifend sei angemerkt, dass die Kommunikationsstruktur von Bild 12.4b derjenigen sehr ähnlich ist, über die der Prozessor eines Prozessorcomputers mit seinem Arbeitsspeicher kommuniziert (siehe Bild 13.7). Dort ist der rechte Halbkommutator durch einen Kommutator ersetzt. Die Quadrate des linken Halbkommutators in Bild 12.4 entsprechen in Bild 13.7 den Speicherplätzen des Arbeitsspeichers. Die Speicherplätze spielen also die Rolle der "Teilnehmer". Größere Computer bestehen aus vielen Einheiten, die untereinander Daten austauschen. Der Austausch kann streckenweise über *gemeinsame* Leitungen gemäß Bild 12.4b erfolgen. Ein solches Verbindungssystem innerhalb eines Computers heißt **Bus**. Ein Bus arbeitet i.d.R. *einkanalig*, d.h. er kann jeweils nur einem einzigen Teilnehmner zur Verfügung gestellt (zugeteilt) werden und diesen mit dem gewünschten Adressaten verbinden. Das Gleiche gilt für Telekommunikationsnetze, auch für Rechnernetze. Ein *Kommunikationskanal* kann jeweils nur einem einzigen Teilnehmer zugeteilt werden. Die Zuteilung kann zentral durch einen speziellen Steueroperator oder auch dezentral erfolgen, indem z.B. der Teilnehmer selber feststellt, ob der Kanal (die Leitung, der Bus) frei oder besetzt ist.

Im Sinne einer Dekomposition des Halbkommutators führen wir nun gedanklich folgende Operation aus. Wir zerschneiden zeilenweise die Diodenmatrizen (die Adressiermatrizen des Multiplexers und Demultiplexers) und verteilen die Zeilen an die zugeordneten Teilnehmer (Adressaten). Jeder Teilnehmer erhält eine MUX- und eine DMUX-Zeile, über welche die Verbindung zum bzw. vom Bus hergestellt werden kann. Die Vorgehensweise wird oft als **Schlüssel-Schloss-Prinzip** bezeichnet, Das hat folgenden Grund. 3

In Kap.12.1 hatten wir die Funktionsweise des Decodierers und des Codierers bzw. der Adressier- und der Speicher-Matrix unter Verwendung der Begriffspaares *Wort-Leitung* bzw. *Speicheradresse-Speicherinhalt* beschrieben. Jetzt wollen wir die Funktionsweise der Adressiermatrix (des Decodierers) unter Verwendung des Be-

griffspaares *Schlüssel-Schloss* beschreiben und greifen damit auf den Beginn des Kapitels 12.1 zurück, wo wir den Decodierer *Entschlüsseler* genannt hatten.

Die Funktionsweise eines Entschlüsselers (z.B. des Decodierers von Bild 12.1a) lässt sich folgendermaßsen beschreiben. Der Ent*schlüssel*er vergleicht das jeweilige Eingabewort (den *Schlüssel*) mit den in der Decodiermatrix (Adressiermatrix) strukturell gespeicherten Wörtern (*Schlössern*). Wenn er Übereinstimmung erkennt, m.a.W. wenn der Schlüssel zum Schloss "passt", belegt er den betreffenden Leiter mit einer 1 und öffnet damit das Schloss, d.h. den Zugang zu dem betreffenden Speicherplatz. An die Stelle des Speicherplatzes tritt jetzt ein Teilnehmer, ein "Adressat", und zwar derjenige, der mit dem zum Schlüssel passenden Schloss versehen ist.

Damit ist es möglich, Nachrichten über eine einzige Leitung, den **Datenbus**, sozusagen "an alle" zu versenden und zwar zusammen mit dem jeweiligen Schlüssel, der *Adresse* des Teilnehmers (des Adressaten). Das erfolgt im Bussystem eines Computers i.d.R. über einen speziellen **Adressbus**. Außerdem müssen die Weichen in Bild 12.4a auf "*Senden*" beziehungsweise, wenn ein Teilnehmer eine Nachricht abstezen will, auf "*Empfangen*" gestellt werden. Dafür wird häufig ein spezieller **Steuerbus** eingerichtet.

Wenn nur eine einzige Leitung zur Verfügung steht (wie z.B. in einem Telefonnetz oder in einem Rechnernetz, dessen Teilnehmer über ein Telefonnetz kommunizieren), muss der Schlüssel gemeinsam mit der Nachricht versendet werden, ähnlich wie die Adresse auf einem Briefumschlag zusammen mit dem Brief versendet wird. [4]

Das Schlüssel-Schloss-Prinzip ähnelt dem Adressierungsmechanismus für Steuerinformationen in lebenden Organismen. Schlüssel und Schloss sind hier molekulare Strukturen (Enzyme, Hormone, Transmitter), die "zusammenpassen" müssen, damit an dem betreffenden Ort ein bestimmter Steuereffekt ausgelöst wird. So wird beispielsweise der Stoffwechsel (Austausch von Baustoffen) im lebenden Organismus gesteuert. Im Gegensatz zum Bus kann sich eine riesige Anzahl von Bauelementen (Eiweißverbindungen) und Steuerenzymen gleichzeitig durch das "Kommunikationssystem" bewegen; man hat es also nicht mit einem *Ein*kanal-, sondern mit einem *Mehr*kanalkommutator zu tun.

Der Mehrkanalkommutator ist auch für die technische Nachrichtenübertragung oft eine unabdingbare Notwendigkeit. Eine naheliegende, aber aufwendige Lösung ist der **Kreuzschienenverteiler** (Bild 12.5). Er enthält pro Teilnehmer je einen senkrechten und einen waagerechten Leiter. In jedem Schnittpunkt (die Schnittpunkte auf der Diagonalen ausgenommen) ist ein Schnittpunktoperator angeordnet, über den eine gerichtete Verbindung nach unten oder nach rechts hergestellt werden kann. Auf diese Weise kann gleichzeitig eine größere, aber doch begrenzte Anzahl von Verbindungen hergestellt werden; man spricht dann von einem **Mehrkanalkommutator**.

Bild 12.5 erinnert an die Leitermatrizen von Bild 12.1b. Doch reichen die einfachen Diodenmatrizen nicht aus, um Verbindungen nach Wunsch herzustellen. Die

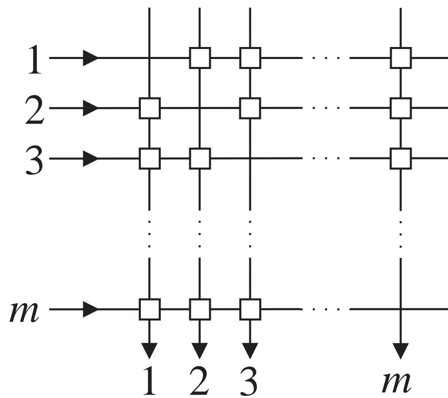


Bild 12.5 Kreuzschienenverteiler

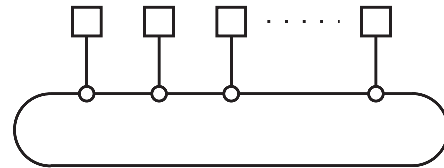


Bild 12.6 Ringverbindung

Schaltungen sind recht aufwendig. Es sind viele weniger aufwendige und dennoch ausreichend flexible Mehrkanalkommutatoren entwickelt worden, die - ebenso wie Diodenmatrizen - letztlich aus *Schaltern* bestehen. Darum werden sie auch *Schalternetze* genannt, eine Bezeichnung, die jedoch, wie wir aus Kap.10.2 wissen, für die gesamte rein elektronische Computerhardware zutreffend ist.

Bild 12.6 zeigt eine andere mögliche Kommunikationsstruktur, eine *Ringverbindung*. Es ist gewissermaßen die der *Sternverbindung* diametral entgegengesetzte Struktur. Alle Teilnehmer benutzen ein und dieselbe Leitung, den **Ringbus**. Jeder Teilnehmer speist seine adressierten Nachrichten in den Ringbus ein, in dem sie solange kreisen, bis der Adressat die Adresse als die seinige erkennt und die Nachricht dem Ring entnimmt. Weitere Methoden, Strukturen und technische Einzelheiten zur Kommunikationstechnik findet der Leser in der Literatur[3].

Bevor auf eine andere wichtige Anwendung der Diodenmatrizen eingegangen wird, soll eine viel elegantere Methode der Mehrkanalkommunikation erwähnt werden, die jeder kennt, die Methode der Senderwahl durch Frequenzabstimmung. Sie kommt in Radio- und Fernsehempfängern zur Anwendung. Während aber die sog. “Radiowellen” oder “Ätherwellen” (die elektromagnetischen Träger der zu übertragenden Signale) den gesamten “Äther” erfüllen, verwendet die moderne Nachrichtentechnik Methoden, welche die Wellen auf engem Raum (in engen “Schläuchen”) zusammenhalten. Das lässt sich entweder mit Lasern bewerkstelligen, die u.a. in Satelliten-Komunikationssystemen eingesetzt werden, oder mit Hilfe spezieller Kabel, in denen sich die Trägerwellen fortbewegen. Die Anzahl der realisierbaren Kanäle pro Kabel und die Anzahl der pro Kanal und Sekunde übertragbaren Bit nimmt mit der Trägerfrequenz zu (vgl. Kap.11.2 [11.3]). Darum ist man bemüht, möglichst hohe Trägerfrequenzen zu verwenden, bis hinauf in den Bereich

3 Siehe z.B. [Tanenbaum 98], [Werner 95].

des sichtbaren Lichts und darüber hinaus. Das ist der Hauptgrund für die Verwendung von *Lichtleitern* in der Nachrichtentechnik.

Diese kurzen Bemerkungen über die Anwendung von Trägerwellen in der Kommunikationstechnik wurden der Vollständigkeit halber eingefügt, obwohl sie nicht zum eigentlichen Thema des Kapitels gehören, dem wir uns nun wieder zuwenden.

12.3.3 Funktionsgeneratoren

In Kap.4.2 war von analog arbeitenden Funktionsgeneratoren, speziell vom Sinusgenerator die Rede gewesen (vgl. Bild 4.2b). Mit Hilfe eines Codeumsetzers lässt sich das digitale Pendant eines solchen Generators aufbauen. Dazu wird zunächst die Funktionstafel in die Matrizen des Codeumsetzers eingetragen (eingprägt), die Argumentwerte in die Adressmatrix und die Funktionswerte in die Speichermatrix. Damit sind die Funktionswerte und deren Adressen gespeichert. Wenn die Möglichkeit bestehen soll, die Wertetafel zu löschen, um eine andere Tafel zu speichern, muss eine PLA verwendet werden, also ein Codeumsetzer mit programmierbarer Adress- und Speichermatrix. Will man die Funktion ausdrucken, muss man die Speicherplätze über die als Adressen dienenden Argumentwerte aufrufen und jeweils beide Werte ausgeben.

Diese Prozedur lässt sich vereinfachen, indem die Speicherplätze durchnummeriert und die Nummern in die Adressmatrix eingetragen werden, während in die Speicherplätze (die Zeilen der Speichermatrix) je ein Argumentwert und der zugeordnete Funktionswert eingetragen werden. In diesem Fall kann ein EPROM-Baustein verwendet werden, denn die Adressiermatrix kann fest programmiert sein.

Um die Argument-Funktionswertepaare der Reihe nach ausgeben zu lassen, kann man einen Zahlengenerator verwenden, der die Nummern sequenziell generiert. Dafür eignet sich ein Taktgeber (z.B. der Taktgeber des benutzten Computers). Durch Zählen und Untersetzen (wiederholtes Halbieren) der Anzahl der Taktimpulse werden die Dualzahlen in steigender Folge generiert. Gibt man sie der Reihe nach auf die PLA, so liefert diese die Wertepaare der Funktion. Diese können gedruckt oder in ein Diagramm überführt werden, indem jedes Paar als Koordinaten je eines Kurvenpunktes interpretiert wird. Auf diese Weise kann der Funktionsverlauf auf dem Bildschirm dargestellt oder durch einen Drucker oder *Plotter* gezeichnet werden. Ein Plotter ist ein Zeichengerät, das an einen Computer als zusätzliche Ausgabeeinheit angeschlossen werden kann.

12.3.4 Steuermatrix

Eine andere Standardaufgabe der Rechentechnik ist die Prozesssteuerung. Betrachten wir als Beispiel die Steuerung des Waschprozesses in einer automatischen Waschmaschine. Die Maschine muss über einen *Steueroperator* verfügen, der eine Folge von *Steuersignalen* generiert, welche die verschiedenen Funktionseinheiten (Wassereinlassventil, Heizung, Trommelmotor, Wasserpumpe) jeweils im richtigen Augenblick ein- bzw. ausschaltet. Als Steuersignal zur Steuerung einer Funktions-

einheit genügt ein einziges Bit, ein *Steuerimpuls*. Wenn gleichzeitig mehrere Einheiten angesteuert werden sollen (z.B, Wasser ab, Heizung an), muss der Steueroperator *Steuerwörter* generieren.

Welches Steuerwort zu einem gegebenen Zeitpunkt generiert werden muss, kann von der Uhrzeit abhängen (z.B. wenn um 10 Uhr gestartet oder wenn 3 Minuten lang geschleudert werden soll) oder vom momentanen *Zustand* des Waschprozesses, der durch Messeinrichtungen, auch *Fühler* genannt, gemessen wird, z.B. durch Temperatur- oder Wasserstandfühler. Die Messwerte müssen dem Steuergenerator *gemeldet* werden, zweckmäßigerweise binär codiert. Die Uhrzeit kann *intern* durch den Steueroperator selbst generiert werden, wenn er über eine Uhr (Impulsgenerator mit Impulszähler) verfügt.

Der Steuersignalgenerator hat also binäre Eingabewörter in binäre Ausgabewörter zu transformieren. Hat man das erkannt, liegt die Idee nicht mehr ferne, mit der M.V.Wilkes 1951 berühmt wurde, die Idee der Steuermatrix. Sie besteht darin, dass die Steuerwörter in die Speichermatrix und die Bedingungen für die jeweiligen Steueraktionen in die Adressiermatrix eines ROM eingetragen werden. Die Bedingungen können Bedienaktionen des Nutzers, die Uhrzeit oder Meldungen sein, d.h. Messwerte, die der gesteuerte Operator (die Waschmaschine) dem Steueroperator "meldet".

Im Falle einer Waschmaschine ohne Automatik ist der "Steueroperator" der Mensch, der die Maschine *bedient* und entsprechend der aktuellen Situation (Bedingung) die richtige *Steuerentscheidung* fällt. Die Bedienungsanleitung (die Vorschrift, der sprachliche Operator, der Algorithmus), nach der er zu verfahren hat, muss jeder möglichen *Bedingung* eine bestimmte *Steueraktion* zuordnen. Eine Tabelle, die in der linken Spalte die Bedingungen und in der rechten die notwendigen Aktionen enthält, heißt **Entscheidungstabelle**. Jede Zeile der Tabelle enthält eine **Regel** der Form 5

$$\text{Wenn } b_1 \text{ und } b_2 \text{ und...und } b_n, \text{ dann } a_1 \text{ und } a_2 \text{...und } a_m, \tag{12.5}$$

worin b_i die Bedeutung hat "Bedingung b_i ist erfüllt" und a_i "Aktion a_i ist auszuführen". Aus der Entscheidungstabelle lässt sich unmittelbar ablesen, wie die Schnittpunktoperatoren (Dioden) in der Adressiermatrix und der Speichermatrix zu platzieren sind.

Es kann zweckmäßig, evtl. sogar notwendig sein, das beschriebene **Matrixsteuerwerk** zu einem *Automaten* zu erweitern. Angenommen, der Waschautomat soll das Spülen, Schleudern und Abpumpen drei mal durchführen. Das kann dadurch erreicht werden, dass drei Regeln programmiert (in den ROM eingeprägt) werden, für jeden Teilvorgang eine Regel. Wenn die drei Teilvorgänge identisch sind, genügt es, eine einzige Regel abzuspeichern und einen Zähler einzubauen. Wenn der Zähler bis 3 gezählt hat, wird der Teilvorgang nicht mehr wiederholt. Der Zählerstand kann als *innerer Zustand* des Steueroperators angesehen werden, sodass dieser zu einem *Automaten* (zu einer Realisierung des endlichen Automaten) wird.

Die Erweiterung einer ROM-Schaltung zu einem Automaten (vgl. Bild 8.5) erfolgt durch Einbau einer Rückkopplung von der Speichermatrix zur Adressiermatrix über ein Verzögerungsglied (D), welches sich das aktuelle Steuerwort merkt und im nächsten Takt auf den Eingang des ROM gibt, der dadurch zu einem *Steuerautomaten* wird. In Bild 12.2 ist eine solche Rückkopplung gestrichelt angedeutet; das zurückgegebene Steuerwort ist (in Analogie zum inneren Zustand des endlichen Automaten) mit z bezeichnet. Die Anzahl der senkrechten Leiter der Adressiermatrix muss entsprechend erhöht werden. Auf diese Weise kann das aktuelle Steuerwort vom Steuerwort des vorangehenden Arbeitstaktes abhängig gemacht werden. Da x und y in dem Beispiel Tripel sind, gibt es 9 verschiedene (x,y)-Paare, die der Decodierer erkennen muss. Er besitzt also 9 waagerechte Leiter.

Der Inhalt einer Steuermatrix stellt ein oder mehrere als Hardware realisierte Programme dar, die i.d.R. vom Nutzer nicht verändert werden können. Darum werden sie nicht als *Software*, sondern als **Firmware** bezeichnet.

Mit der Einführung einer Rückkopplung haben wir das Thema dieses Kapitels, die mikroelektronische Realisierung zirkelfreier boolescher Netze bereits verlassen und den ersten Schritt zum programmierbaren Rechner getan.

13 Von der Kombinationsschaltung zum Von-Neumann-Rechner

Zusammenfassung

Die Realisierung von Funktionen durch Kombinationsschaltungen hat zwei wesentliche Mängel. Zum einen müssen die Funktionswerte vor dem Entwurf der Schaltung bekannt sein. Zum anderen ist es bei einer Erweiterung der Funktionstafel, beispielsweise bei Erhöhung der Genauigkeit der Funktionswerte, nicht mit einer Erweiterung der Schaltung getan, vielmehr muss sie völlig neu entworfen und aufgebaut werden. Beide Mängel werden durch die Verwirklichung der dritten Grundidee des elektronischen Rechnens, der Idee des programmierbaren Rechners behoben.

Unbeschränkt erweiterbare Funktionstafeln (z.B. die Additionstafel) lassen sich bei binär-statischer Codierung nur durch zirkuläre boolesche Netze in Form von KR-Netzen (Netze aus Kombinationsschaltungen und Registern) realisieren. Der Prototyp des programmierbaren Rechners ist ein als *Von-Neumann-Rechner* bezeichnetes KR-Netz. Seine Hauptbestandteile sind ein *Prozessor* und dessen Arbeitsspeicher, *Hauptspeicher* genannt. Der Prozessor kann imperative Algorithmen abarbeiten, die als *Maschinenprogramme*, d.h. als Folge von Maschinenbefehlen formuliert sind. Dabei holt sich der Prozessor die Befehle und die notwendigen Operanden der Reihe nach aus dem Hauptspeicher und liefert die Ergebnisse an den Hauptspeicher zurück. Prozessor und Hauptspeicher liegen in einer (äußeren) Rückkopplungsschleife. Die Datenübergabe in beiden Richtungen erfolgt über eine einzige Verbindung, den sog. *von-neumannschen Flaschenhals*.

Der Prozessor enthält eine innere Rückkopplungsschleife, in welcher die ALU (arithmetisch-logische Einheit), der *Akkumulator* (AC) und ein oder mehrere Datenregister für die schnelle Zwischenspeicherung von Operanden liegen. Die ALU, der Akkumulator, die Datenregister und einige weitere Register sind zu einem KR-Netz verbunden und bilden eine funktionelle Einheit, RALU genannt, die ihrerseits zusammen mit einem Steueroperator den Prozessor bildet. Aus den ALU-Operationen werden durch Steuerung des Operandenflusses, der durch das RALU-Netz fließt, die Maschinenoperationen komponiert. Die Flusssteuerung wird zum Teil von der RALU selbst (dezentral), zum Teil vom Steueroperator (zentral) durchgeführt. Dabei können die Operanden sowohl in der inneren als auch in der äußeren Rückkopplungsschleife zirkulieren.

Der Steueroperator kann als Matrixsteuerwerk ausgebildet sein, d.h. er kann eine oder mehrere ROM-Bausteine enthalten, in denen Programme (als sog. Firmware) gespeichert sind, sodass ein Maschinenbefehl eventuell eine umfangreiche Kompositaktion auslöst. Damit ein Maschinenbefehl vom Prozessor ausgeführt werden kann, muss er das Format des *Befehlsregisters* besitzen, in das der Prozessor jeden

aus dem Hauptspeicher geholten Befehl einliest. Das Format legt fest, welche Teile des Befehls die Operation und welche die Operandenadressen darstellen (codieren). Die Befehle einer *Zwei-Adress-* bzw. *Drei-Adress-Maschine* enthalten zwei bzw. drei Adressen.

Der Prozessor komponiert Operatoren (Operationen) auf zwei Ebenen. Auf der unteren Ebene komponiert er Maschinenoperationen aus den ALU-Operationen und auf der oberen Ebene Kompositoperationen aus den Maschinenoperationen gemäß den Vorgaben von *Maschinenprogrammen*. Ein Maschinenprogramm ist eine Folge von Maschinenbefehlen; es stellt also keinen Datenflussplan, sondern einen *Aktionsfolgeplan* dar, und die Komponierung erfolgt nicht nach den Prinzipien der USB-Methode. Demgegenüber erfolgt die Komponierung der Maschinenoperationen durch Steuerung des Operandenflusses in der RALU nach den Prinzipien der USB-Methode.

Dennoch kann der Von-Neumann-Rechner jeden Datenflussplan realisieren und folglich jede rekursive Funktion berechnen. In diesem Sinne ist er universell. Der Datenflussplan als solcher tritt nicht zutage, weil die Operandenplätze im Hauptspeicher zentralisiert sind (wie im Falle des endlichen Automaten) und die Operationen sequenziell ausgeführt werden. Voraussetzung der Universalität ist die Existenz eines Sprungbefehls zur Simulierung von Zweigeweichen. Ein *Sprungbefehl* veranlasst das Herausspringen aus der normalen Befehlsfolge und die Fortsetzung der Abarbeitung an einer anderen Stelle des Programms.

Mit Hilfe des Sprungbefehls lassen sich Iterationsschleifen programmieren. Theoretisch sind Iterationsschleifen ohne Abbruchkriterium denkbar. Eine Iteration, die nicht terminiert, hat zwar keinen praktischen Sinn und widerspricht dem Realisierbarkeitsprinzip, doch hat sie theoretische Bedeutung. Mit Hilfe nichtterminierender Iterationen lassen sich Funktionen mit abzählbar unendlicher Wertetafel definieren und programmieren. Die Ausführung der Vorschrift ist ein nicht endender Prozess, d.h. eine abzählbar unendliche Folge von Ereignissen.

Eine Funktion, für deren Berechnung eine Vorschrift angebbar ist, welche von einem realen Operator ausgeführt werden kann, der mit statischer Codierung arbeitet, heißt *statisch berechenbare Funktion*. Die Klasse der statisch berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen und der Klasse der mittels Von-Neumann-Rechner berechenbaren Funktionen identisch.

13.1 Die dritte Grundidee des elektronischen Rechnens

Wir sind nun ausreichend vorbereitet, um den Computer nachzuerfinden, genauer den **Prozessorcomputer**, d.h. einen Computer, der aus einem oder mehreren Prozessoren und einem oder mehreren Speichern besteht. Neben diesem Computertyp wird seit längerem an einem ganz anderen Typ gearbeitet, dem **Neurocomputer**. An die Stelle der Prozessoren und Speicher treten Netze aus künstlichen Neuronen, die

sowohl die Verarbeitungsfunktion als auch die Speicherfunktion übernehmen. Wir interessieren uns ausschließlich für den *Prozessor*computer. Vergegenwärtigen wir uns zu diesem Zweck noch einmal unser Ziel sowie die wichtigsten Aussagen und Einsichten, zu denen wir gelangt waren.

Wir hatten in Kap.8.5 unser ursprüngliches Ziel - wir nennen es Ziel 1 - umformuliert und ein scheinbar ganz neues Ziel - Ziel 2 - festgelegt. Wir stellen die beiden Ziele noch einmal einander gegenüber:

Ziel 1: Entwurf eines Gerätes, das jede berechenbare Funktion berechnen kann. 1

Ziel 2: Realisierung eines *Basiskalküls*, in den sich alle Kalküle transformieren lassen, derer sich die natürliche Intelligenz bedient, und Durchführung dieser Transformationen in den Basiskalkül.

Aus Kap.8.5 wissen wir, dass beide Formulierungen einander äquivalent sind.

Die erste Idee zur Erreichung von Ziel 2 (die “erste Grundidee des elektronischen Rechnens”) bestand darin, die boolesche Algebra als den zu realisierenden Basiskalkül zu wählen. Die zweite Grundidee ermöglichte die elektronische Implementierung der elementaren booleschen Operatoren, sodass sich die boolesche Algebra nach der USB-Methode realisieren lässt, soweit das Realisierungsprinzip dies zulässt. Es bleibt die Frage offen, wie sich beliebige Kalküle, derer sich die natürliche Intelligenz bedient, in die boolesche Algebra transformieren lassen. Auf diese Frage kommen wir in Teil 3 zurück. Kapitel 13 ist der Frage gewidmet, wie sich einfache zahlenmäßige Rechnungen in den booleschen Kalkül transformieren und mittels elektronischer Schaltung ausführen lassen. Um an den Ergebnissen der Kapitel 10 und 12 leichter anknüpfen zu können, bleiben wir zunächst bei der Zielstellung, wie sie unter “Ziel 1" formuliert ist.

Aus Kap.9.3 [9.16] wissen wir, dass sich die Wertetafel jeder Funktion in eine Kombinationsschaltung überführen lässt. Damit scheint der Weg zum Ziel 1 frei zu sein. Leider ist dieser Weg aus Aufwandsgründen nicht gangbar. Es müssten nicht nur “unendlich viele” Kombinationsschaltungen gebaut werden, sondern bereits die Größe der einzelnen Schaltungen würde die Grenzen des Machbaren übersteigen. Das haben die Abschätzungen in Kap.9.2 [9.6] gezeigt. Der Grund des Dilemmas liegt in der Starrheit von Kombinationsschaltungen.

Nach dieser Bemerkung liegt die **dritte Grundidee des elektronischen Rechnens** auf der Hand. Sie besteht in der Komponierung *steuerbarer* boolescher Netze. Steuerbarkeit setzt das Vorhandensein steuerbarer Flussknoten (Weichen bzw. Tore) voraus. Die einfachsten steuerbaren Netze sind *steuerbare Kombinationsschaltungen*, also zirkelfreie boolesche Netze, die eine oder mehrere Weichen enthalten. In einem solchen Netz können verzweigte, nichtzirkuläre Operandenflüsse gesteuert werden. Vorgreifend sei erwähnt, dass die ALU ein solches Netz ist.

Die Verwendung von Weichen beim Komponieren von Operatorennetzen eröffnet aber weit größere Möglichkeiten, denn mit ihrer Hilfe können zirkuläre Kopplungsstrukturen aufgebaut und zirkuläre Operandenflüsse gesteuert werden, vorausgesetzt, das Netz enthält Speichereinheiten [9.19]. Die charakteristischen Merkmale

steuerbarer Netze sind bereits aus Bild 8.1 zu erkennen, nämlich die Existenz von Speichern und Weichen und evtl. von Rückkopplungen. Wir wollen uns die weitreichenden Konsequenzen der Verwendung von Weichen klarmachen.

In Kap.8.4.5 hatten wir die Komponierung steuerbarer Operatorennetze auf abstrakter Ebene, ohne Bezugnahme auf boolesche Operatoren untersucht. Dort waren wir auf ganz anderem Wege auf das Komponierungsproblem gestoßen, nämlich bei der Suche nach universellen Methoden der Algorithmenbeschreibung. Die Gemeinsamkeit mit unserer jetzigen Fragestellung ergibt sich aus der Notwendigkeit der Steuerung. Diese ist von einem Steueroperator (Gerät oder Mensch) nach einer bestimmten Steuervorschrift auszuführen. Die Steuervorschrift muss genau festlegen, welche Operationen in welcher Reihenfolge an welchen Operanden auszuführen sind, m.a.W. sie muss ein *imperativer Algorithmus* sein, wie er in Kap.7.2 [7.10] eingeführt worden ist. Wenn nichts anderes gesagt wird, ist in Kapitel 13 unter einem Algorithmus ein imperativer Algorithmus zu verstehen.

Die Steuervorschrift für ein steuerbares boolesches Netz nennen wir **Programm** und das Erstellen (Artikulieren) von Programmen **Programmieren**. In diesem Kapitel ist also Programmieren das Artikulieren imperativer Algorithmen. In Kap.8.4 hatten wir gefragt, ob es eine universelle Methode für die Artikulierung von Algorithmen gibt. Aus jetziger Sicht lautet die Frage: Gibt es eine *universelle* Methode für die *Komponierung* und *Programmierung* steuerbarer boolescher Netze? *Universell* bedeutet - analog zu Kap.8.4 -, dass sich für jede Funktion ein Netz und ein Programm (Algorithmus) angeben lässt nach welchem das Netz die betreffende Funktion berechnet. (Es wird davon ausgegangen, dass das Netz Ein- und Ausgang besitzt, also ein Operator ist.)

In Kap.8.4 musste die Frage abstrakt gestellt und beantwortet werden. Jetzt können wir sie konkret stellen, indem wir von realen Operatoren ausgehen, welche die Programme (Algorithmen) ausführen. Früher war der zugrunde gelegte ausführende Operator der Mensch, und eine konkrete Lösung des Problems (im Sinne des Trägerprinzips) hätte von der Funktionsweise des Gehirns ausgehen müssen. Jetzt besteht die Aufgabe zunächst darin, ein geeignetes universelles steuerbares boolesches Netz zu erfinden, also einen variablen Kompositoperator, der sich auf jede effektiv berechenbare Funktion programmieren lässt. Das wird uns eine ganze Weile beschäftigen, bevor wir uns dem Programmieren und den Sprachen zuwenden können, in denen programmiert wird.

Zuerst muss aber noch die oben gestellte Frage nach den Konsequenzen der Steuerbarkeit vollständig beantwortet werden. Die Konsequenzen sind verstreut in den vorangehenden Kapiteln enthalten und in der folgenden Auflistung zusammengefasst und kurz kommentiert.

1. **Programmierbarkeit**. Ein steuerbares boolesches Netz ist variabel, es lässt sich auf die Berechnung verschiedener Funktionen “programmieren”.
2. **Komponierbarkeit zirkulärer Netze**. Ihr entspricht in Kap.8.4.5 die Möglichkeit der Operatorkomponierung mittels Iteration (vgl. Bild 8.9).

3. **Notwendigkeit von Toren und Speichern**. Sie ergibt sich für zirkuläre Netze aus dem Prinzip der statischen Codierung (siehe Kap.9.4 [9.19]).
4. **Halteproblem**. Im Falle zirkulärer Netze muss das Programm angeben, wann die Operationsausführung zu beenden ist. Das kann durch explizite Vorgabe der Iterationszahl oder durch ein Prädikat erfolgen. Es ist nicht immer feststellbar, ob ein gegebenes Programm terminiert, d.h. ob der programmierte zirkuläre Prozess anhält (siehe Kap.8.3 [8.22]), m.a.W. ob es einen Algorithmus beschreibt oder nicht.
5. **Universalität**. Es besteht die Möglichkkeit, universelle steuerbare boolesche Netze zu komponieren. Dafür ist - neben der Realisierung der Flussknoten - ein einziger elementarer Operatortyp ausreichend, der *Inkrementierer*, also ein Operator, der eine Zahl um 1 erhöht (siehe Kap.8.4.5.). Er kann als zirkelfreier boolescher Operator realisiert werden, denn die Addition einer 1 kann vom Halbaddierer ausgeführt werden (siehe die Bilder 9.3 und 9.4).

Die Punkte 1 und 2 sind unmittelbare Folgen der Steuerbarkeit. Die Punkte 3 und 4 sind Konsequenzen von Punkt 2 und insofern bedingte Konsequenzen der Steuerbarkeit. Sie entfallen für steuerbare zirkel*freie* Netze. Punkt 5 liefert uns den Wegweiser zu unserer Erfindung, denn er enthält de facto das Arbeitsprinzip des Prozessors. Wir müssen "nur" noch die steuerbare Schaltung erfinden, in die der Inkrementierer so eingebettet ist, dass sich mittels Iteration "alles" berechnen lässt. In den Kapiteln 13.4 und 13.5 werden wir sehen, wie sich diese Idee in etwas abgewandelter Form verwirklichen lässt (die Rolle des Inkrementierers wird von der ALU übernommen, die unter anderem auch inkrementieren kann).

Übergangen wurde bislang das Problem der Speicherung. Das zu komponierende steuerbare Netz kann nur dann zu Recht als *boolesches* Netz bezeichnet werden, wenn auch die erforderlichen Tore und Speicher aus elementaren booleschen Operatoren, konkret aus Schaltern komponiert sind. Im folgenden Kapitel werden wir uns überlegen, wie sich dies bewerkstelligen lässt.

Die wichtigste Konsequenz der Steuerbarkeit ist zweifelsohne die Programmierbarkeit. Sie war zunächst von uns gar nicht beabsichtigt, denn auf die Idee der Steuerbarkeit hatte uns das Dilemma mit den Kombinationsschaltungen gebracht, die immense Ausmaße anzunehmen drohten. So betrachtet ist die Steuerbarkeit lediglich als Nebenprodukt anzusehen, was jedoch angesichts ihrer prinzipiellen Bedeutung kaum gerechtfertigt ist. Denn Programmierbarkeit macht erst möglich, was man von einem Rechner erwartet, nämlich die Berechnung von Funktionswerten, die zuvor noch nicht berechnet worden waren. Die Berechnung setzt freilich voraus, dass ein Berechnungsalgorithmus, ein terminierendes Programm vorliegt. Dagegen setzt die Realisierung einer Funktion als Kombinationsschaltung voraus, dass ihre Wertetafel vorliegt.

Sicher hat beides die Erfinder bei der Suche nach dem universellen Rechner beflügelt, sowohl die technische Machbarkeit als auch die Programmierbarkeit. Bei den Pionieren des "maschinellen Rechnens", den Erfinder der ersten universellen

“Rechen-Maschinen”, hat aber wohl doch die Programmierbarkeit an erster Stelle gestanden, d.h. *die dritte Grundidee des elektronischen Rechnens, die* ***Programmsteuerung***. Diese Idee stammt von CHARLES BABBAGE, wenn das Wort “elektronisch” zu “maschinell” verallgemeinert wird.

Zu den Wörtern “Maschine” und “maschinell” ist eine Bemerkung am Platze. Bei dem Wort “Maschine” werden die meisten Menschen zunächst einmal an Werkzeugmaschinen, Baumaschinen oder Kraftmaschinen denken. Man könnte sie alle unter der Bezeichnung “Energie verarbeitende Maschinen”[1] zusammenfassen. Die analoge Wortverbindung “Information verarbeitende *Maschine*” ergibt sich fast “automatisch” als zusammenfassende Bezeichnung für alle “künstlichen Vorrichtungen”, die der Verarbeitung von Zeichen und Zeichenketten dienen; dazu gehören Kassenautomaten ebenso wie mechanische Rechenmaschinen und moderne Computer. Auch sie als *Maschinen* zu bezeichnen, den Computer eingeschlossen, ist durchaus “passend”. Denn das griechische Wort, von dem sich das Wort “Maschine” herleitet, hat die Bedeutung “künstliche Vorrichtung” oder “Werkzeug”. Und schließlich ist die kulturgeschichtliche und philosophische Problematik, die durch die Gegenüberstellung “Mensch und Maschine” angesprochen wird, für Energie verarbeitende und Information verarbeitende Maschinen ein und dieselbe. Man denke an die Wechselwirkung zwischen Mensch und Maschine und an die Rückwirkung der Maschine auf die kulturelle Evolution, konkret die Rückwirkung der Dampfmaschine als Initiator der “*industriellen Revolution*” im 19. Jahrhundert und an die Rückwirkung der Rechenmaschine als Initiator der gegenwärtigen “*informationellen Revolution*”.

Dieser Bemerkung soll eine zweite angefügt werden, jedoch für Leser, die sich in der Entwicklung der Rechentechnik auskennen. Der Prozessor, der in diesem Kapitel beschrieben wird, und manche Begriffe, die dabei verwendet werden, machen auf den Fachmann eventuell einen etwas antiquierten Eindruck. Wer spricht beispielsweise heute noch von Zwei-, Drei- oder Vier-Adress-Rechnern? Der Weg, den wir gehen werden ist durch den Wunsch vorgegeben, als Nichtfachmann den Computer *nach*zuerfinden. Die Vorsilbe “nach” bedeutet, dass wir uns in der *Nach*folge der Pioniere der Computertechnik befinden. Man muss den “ersten”, noch ganz “primitiven” Computer nacherfunden haben, um das Grundprinzip zu verstehen, nach dem alle späteren Computer arbeiten. Dies ist auch der didaktisch beste Weg zum Kern der elektronischen Rechentechnik. In Kap.19.2 werden die wichtigsten zusätzlichen Ideen nachgetragen, welche die Entwicklung von der “ersten elektronischen Rechenmaschine” zum modernen Computer markieren.

1 Richtiger wäre die Bezeichnung “Energie transformierende Maschinen”.

13.2 Elektronische Speicher

13.2.1 Register

Bevor wir den Gedanken der Programmierbarkeit weiter verfolgen, schieben wir einige wichtige Überlegungen zum Problem der Aufbewahrung (Speicherung) von Operanden ein und wenden uns zunächst der Frage zu, wie die erforderlichen Tore und Speicherplätze (die kleinen Quadrate in Bild 8.1a; die fetten Seiten symbolisieren die Eingabetore), über welche die Operanden in einem Operatorennetz von Operator zu Operator weitergegeben werden, aus booleschen Operatoren aufgebaut werden können.

Der Leser ahnt vielleicht schon, wie ein solcher Speicher funktionieren könnte. Er braucht sich nur an den Ein-Bit-Speicher aus Kap.9.5 und an das Schlüssel-Schloss-Prinzip aus Kap.12.3.2 [12.3] zu erinnern, um eine Idee zu haben, wie der Arbeitsspeicher eines Computers aufgebaut werden kann. Doch wollen wir nicht mit dem Endprodukt beginnen, sondern Schritt für Schritt vorgehen. Zunächst muss das boolesche Netz des Ein-Bit-Speichers in ein Schalternetz überführt werden. Zu diesem Zweck sind die booleschen Operatoren durch Schalterkombinationen zu ersetzen, wie in Kap.10.1 dargelegt wurde. Die technische Realisierung der Schalter kann, wie besprochen, mittels Transistoren erfolgen.

Den Speicher für eine Bitkette, die einen Operanden intern codiert, nennen wir **Register**. Die Operandenlänge ist i.Allg. nicht länger als die sog. **Rechnerwortlänge**; das ist die Bitkettenlänge, mit der ein bestimmter Computer standardmäßig hantiert. Sie stellt einen wichtigen technischen Parameter eines Computers dar. Von ihr hängt die Rechengenauigkeit und Rechengeschwindigkeit ab. Beide lassen sich durch Erhöhung der Rechnerwortlänge steigern. Vor gar nicht langer Zeit setzte sich als PC-Standard eine Länge von 32 Bit durch. Aber schon sind 64 Bit im Gespräch. Ältere PCs arbeiteten mit 16 Bit und noch ältere mit 8 Bit.

Ein Speicher (Register) für 32 Bit lässt sich aus 1-Bit-Speichern komponieren, indem man die Schaltung von Bild 9.7 (oder eine andere Flipflop-Variante) 32-mal nebeneinander anordnet, wie in Bild 13.1a dargestellt ist. Jedes Quadrat symbolisiert einen Ein-Bit-Speicher. Die Gabel spaltet die Bits der zu speichernden Bitkette in einzelne Bits auf (gedankliches Aufspalten durch eine Spaltegabel), die Vereinung fasst die Bits wieder zu einem Wort zusammen. Hardwaremäßig handelt es sich um das "Aufbinden" bzw. "Zusammenbinden" der Einzelleitungen eines mehradrigen Kabels bzw. - bei mikroelektronischer Realisierung - eines Leitertupels auf einem Chip. Als Symbol für Register verwenden wir ein Quadrat oder Rechteck mit einer fetten Seite, die das Eingabetor bzw. die Eingabetore symbolisiert.

In das Register von Bild 13.1a werden die einzelnen Bits *gleichzeitig* (**parallel**) ein- bzw. ausgelesen. Unter Umständen kann es aber zweckmäßig oder sogar notwendig sein, sie *nacheinander* (**sequenziell**) ein- oder auszulesen. Das lässt sich auf zweierlei Weise bewerkstelligen, zum einen nach dem FIFO-oder **Silo**-Prinzip,

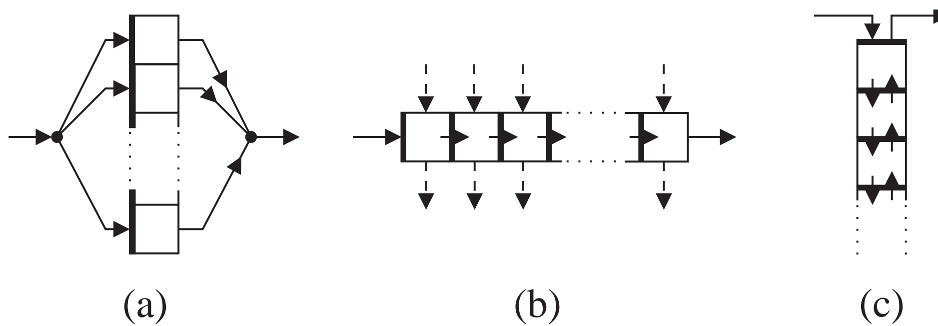


Bild 13.1 Register. **(a)** - Parallelregister; **(b)** - Schieberegister; **(c)** - Stapelregister.

zum anderen nach dem LIFO- oder **Stapel**-Prinzip (FIFO von “First In First Out”, LIFO von “Last In First Out”). Beim Siloprinzip wird, wie beim Getreidesilo, zuerst (beim Silo unten) ausgegeben, was auch zuerst (beim Silo oben) eingespeichert wurde. Das Stapelprinzip entspricht dem Wachsen und Abnehmen eines Aktenstapels, auf den Zugänge obenauf gelegt werden und dem die jeweils oberste Akte zur Bearbeitung entnommen wird.

Für Register, die nach dem Siloprinzip arbeiten, hat sich die Bezeichnung **Schieberegister** eingebürgert, weil die Bits durch das Register “durchgeschoben” werden. Es bietet sich z.B. für die Speicherung der Summanden eines sequenziellen Addierers an, der die einzelnen Stellenwerte der Reihe nach addiert, wie man es in der Schule gelernt hat. Ein solcher Addierer kann aus einem Volladdierer (vgl. Kap. 9.3 [9.13]), einem Verzögerungsglied (vgl. Bild 12.2) und drei Schieberegistern aufgebaut werden. Aus den beiden Summandenregistern wird in jedem Takt je ein Bit herausgeschoben und dem Volladdierer übergeben, der das Summenbit (y in Bild 9.2) in das Summenregister hineinschiebt und den Übertrag (z in Bild 9.2) dem Verzögerungsglied übergibt, das ihn im nächsten Takt auf den Eingang des Volladdierers zurückgibt.

2 Die Addition lässt sich dadurch beschleunigen, dass 32 Volladder parallelgeschaltet werden. Der so komponierte **Paralleladdierer** kann die Addition von Bitketten in einem einzigen Schritt ausführen. Seine Arbeitsregister müssen Parallelregister sein. Der Übergang von der sequenziellen zur parallelen Verarbeitung und der entgegengesetzte Übergang sind Standardoperationen der Hardware. Sie lassen sich mit Registern realisieren, deren Inhalt sowohl parallel (über die gestrichelten Pfeile in Bild 13.1b) als auch sequenziell (über die ausgezogenen Pfeile) ein- und ausgegeben werden kann.

13.2.2 Adressierbarer elektronischer Speicher

Nach dem Prinzip des hierarchischen Komponierens lassen sich Register als Bausteinspeicher für die Komponierung komplexerer Speicherstrukturen einsetzen. Die Register werden dann Speicherplätze genannt. Im Normalfall wird jeweils auf *einem* Speicherplatz *ein* Rechnerwort abgespeichert. Die Speicherplätze werden

durchnummeriert. Die Nummern spielen die Rolle von *Adressen* zur Adressierung der Plätze. Auf diese Weise ist eine vollständige Ordnung der Speicherplätze und damit auch der in ihnen abgespeicherten Rechnerworte festgelegt.

Das Abspeichern und Auslesen einer Folge von Rechnerworten kann wiederum nach dem Silo- oder nach dem Stapelprinzip erfolgen, also gemäß den Bildern 13.1b bzw. c, wo nun aber ein Quadrat ein Register (einen Speicherplatz) darstellt. Die so entstehenden Speicher heißen Silo- bzw. Stapelspeicher. Stapelspeicher werden häufiger **Kellerspeicher** oder - wie im Englischen - **Stack** genannt. Den Kellerspeicher kennen wir bereits aus Kap.8.4.6 (siehe Bild 8.11). Wenn die Speicherung völlig unsystematisch, d.h. ohne jedes Ordnungsprinzip erfolgt, spricht man von **Heap-Speicherung** (heap = Haufen).

Silospeicher sind am Platze, wenn sich Warteschlangen bilden, z.B. eine Folge von Operanden, die auf die Bearbeitung durch einen Operator warten. Der Einsatz von Kellerspeichern kann bei rekursiven Berechnungen zweckmäßig sein. Silo- und Kellerspeicher erlauben keinen unmittelbaren Zugriff auf die einzelnen Speicherplätze.

Wenn auf die Speicherplätze eines Speichers über ihre Adressen direkt zugegriffen werden soll (sog. *Direktzugriff*), muss der Speicher mit einer geeigneten Zugriffshardware ausgerüstet sein. Das ist z.B. notwendig, wenn der Speicher einem Prozessor als Arbeitsspeicher dient. Denn der Prozessor muss während der Abarbeitung eines Programms ständig gezielt auf Speicherplätze zugreifen, um Daten zu holen oder zu speichern. Er muss also mit jeder Speicherzelle über deren Adresse verbunden werden können. Dazu sind Tore in den Ein- und Ausgabeleitungen der einzelnen Speicherzellen erforderlich, die per Adresse geöffnet werden können. Wenn alle Datenein- und -ausgaben über einen gemeinsamen Bus erfolgen, ergibt sich die Kommunikationsstruktur des Halbkommutators (Bild 12.4).

Damit liegt die Idee, wie ein *adressierbarer Speicher* aufgebaut werden kann, auf der Hand. Zuerst wird die Schaltung eines Registers entworfen, das für jedes Bit eines Rechnerwortes je einen Ein-Bit-Speicher enthält. Sodann werden so viele Register zu einem Speicher zusammengefasst, wie zur Realisierung einer geplanten Speicherkapazität erforderlich sind. Um den Speicher zu einem *adressierbaren Speicher* zu machen, muss jeder Speicherplatz mit einem “Eingangsschloss” und einem “Ausgangsschloss” versehen werden, d.h. in seiner Eingabeleitung und Ausgabeleitung muss je ein AND-Glied vorgesehen werden, und zwar das AND-Glied der betreffenden Zeile des Demultiplexers (DMUX) bzw. des Multiplexers (MUX) eines Halbkommutators in Bild 12.4.

All diese “Bausteinschaltungen”, aus denen ein adressierbarer Speicher “komponiert” wird, können auf einem einzigen Chip realisiert werden. Auf diese Weise lassen sich gegenwärtig Speicherchips mit Speicherkapazitäten von mehreren MByte[2] herstellen.

Damit haben wir den elektronischen **Direktzugriffspeicher** oder **RAM** (Random Access Memory) nacherfunden[3]. Neben dem soeben beschriebenen RAM hat sich

der sog. **dynamische RAM**, abgekürzt **DRAM** durchgesetzt. Er nutzt - ebenso wie der Floating-Gate-MOS-FET (siehe Kap.12.2) - den heute erreichbaren sehr hohen Isolationswiderstand von Kondensatoren. Die codierenden Zustände eines Bit sind zwei Ladungszustände eines winzigen Kondensators. Da der Isolationswiderstand aber nicht unendlich groß ist, fließt die vorhandene Ladung mit der Zeit ab und der Speicherinhalt muss periodisch *aufgefrischt* werden, worauf das Wort "dynamisch" hinweist.

In Kap.12.3.2 [12.4] war die Möglichkeit erwähnt worden, die Adresse in Form eines *Schlüsselwortes* mit der Nachricht direkt zu verbinden (voranzustellen oder anzuhängen). Diese Methode lässt sich auch auf die Speicheradressierung anwenden. Dann wird das Schlüsselwort de facto zu einem Teil des Speicherinhalts. Die Festlegung, dass das Schlüsselwort die Adresse (die Nummer) der Zelle ist, kann fallen gelassen werden, denn der Zugriff kann über beliebig vereinbarte Schlüsselwörter erfolgen. Das führt zur Idee des **Assoziativspeichers**. [3]

Die Bezeichnung "*Assoziativ*speicher" bringt zum Ausdruck, dass mit einem Teil des abgespeicherten Inhalts, dem Schlüsselwort oder **Suchargument**, der restliche Inhalt "*assoziiert*" wird. Das Suchargument spielt die Rolle einer Adresse. Damit eröffnen sich neue Möglichkeiten für das Auffinden von Speicherinhalten. Da als Suchargument jedes Binärwort zulässiger Länge erlaubt ist, können geeignet gewählte Bezeichnungen oder Namen als Suchargumente dienen. Das kann z.B. der (codierte) Name eines Mitarbeiters in einer Mitarbeiterdatei sein. Mit ihm könnten alle unter diesem Suchwort abgespeicherten Personaldaten abgerufen (assoziiert) werden. Das Beispiel weist auf die Bedeutung des assoziativen Zugriffs in Datenbanken hin (siehe Kap.16.2[16.6]). Dabei können die "adressierten Speicherplätze" recht umfangreich sein und viele hardwaremäßig realisierte, adressierbare Speicherplätze umfassen.

Eine weitere wichtige Bedeutung der assoziativen Methode liegt in der Möglichkeit, mehrere Speicherplätze gleichzeitig anzuwählen. Wenn beispielsweise in einer Bibliotheksdatei als Suchargument der Autorenname dient, kann durch Eingabe eines bestimmten Namens auf sämtliche Titel aller Autoren dieses Namens zugegriffen werden.

13.3* Einschub: Berechenbarkeits-Äquivalenzsatz

Bevor wir unseren Weg zum Von-Neumann-Rechner fortsetzen, wollen wir uns überlegen, was die Schaltungen, die wir bisher nacherfunden haben, die Kombinati-

---

2 Ein Byte sind 8 Bit; ein MByte (Megabyte) sind 2^{20}, also etwas mehr als 10^6 Byte.

3 Es ist zu beachten, dass "RAM" zuweilen auch als Abkürzung für die Registermaschine (in Kap.8.4.3 mit "URM" bezeichnet) verwendet wird. Ferner ist zu erwähnen, dass auch nicht rein elektronische [9.24] Direktzugriffsspeicher existieren, z.B. Scheibenspeicher oder die heute kaum noch anzutreffenden Ferritkernspeicher.

onsschaltungen und Register, zu leisten in der Lage sind, wenn aus ihnen Netze komponiert werden. In Kap.9.4 [21] waren wir zu einer Aussage gelangt, die zu folgendem Satz verallgemeinert werden kann.

Satz 1. Informationelle Operatoren mit statischer Codierung sind notwendigerweise Netze aus Kombinationsschaltungen und Speichern, wobei in jeder Verbindung (in jedem Operandenweg) zwischen zwei Kombinationsschaltungen ein Speicher mit Eingangstor liegt, oder sie sind in solche Netze überführbar.

Die Verallgemeinerung besteht darin, dass an die Stelle von booleschen Speichern beliebige Speicher treten können. Das bedeutet, dass bei der Überführung, von der in Satz 1 die Rede ist, jeder nicht rein elektronische Speicher des informationellen Operators (Computers), durch einen rein elektronischen zu ersetzen ist, z.B. durch einen elektronischen RAM.

Inzwischen wissen wir, dass Speicher mit Eingangstoren in Form von Registern (geordneten Mengen von Ein-Bit-Speichern) realisiert werden können, sodass sich ein Netz aus Kombinationsschaltungen und Registern ergibt. Ein solches Netz nennen wir KR-Netz. ***KR-Netze** sind **zirkelfreie** oder **zirkuläre** Netze aus Kombinationsschaltungen und Registern, wobei in jeder Leitung für die Bitkettenübergabe zwischen zwei Kombinationsschaltungen ein Register liegt*. Es wird davon ausgegangen, dass Register stets mit Eingabetoren ausgerüstet sind.

Ein KR-Netz mit einem externen Eingang und einem externen Ausgang ist ein steuerbarer Kompositoperator höherer Komponierungsstufe. Wir nennen ihn **KR-Operator**. Der einfachste KR-Operator ist eine Kombinationsschaltung mit vorgeschaltetem Register. Damit kann Satz 1 kompakter artikuliert werden: *Informationelle Operatoren mit binär-statischer Codierung sind KR-Operatoren oder in solche überführbar*. In diesem Satz kann das Adjektiv “binär” ohne Einschränkung der Allgemeinheit gestrichen werden, denn jede statische Codierung kann mittels Kombinationsschaltung in binär-statische umcodiert werden. Es gilt der verallgemeinerte

Satz 2. Informationelle Operatoren mit statischer Codierung sind KR-Operatoren oder in solche überführbar.

Eine Funktion, für deren Berechnung ein KR-Operator angegeben werden kann, nennen wir **KR-berechenbare** Funktion oder **KR-Funktion**. Eine Funktion, die von einem informationellen Operator mit statischer Codierung berechnet werden kann - eine solche Funktion hatten wir *statisch berechenbare* Funktion genannt -, ist also eine KR-Funktion. Damit ergibt sich

Satz 3. Die Klasse der statisch berechenbaren Funktionen ist mit der Klasse der KR-Funktionen identisch.

Enthält ein KR-Operator zwei oder mehrere Kombinationsschaltungen, die über Register hintereinandergeschaltet sind, oder enthält er eine Rückkopplungsschleife, in der ein Register liegt, so arbeitet er zwangsläufig **sequenziell**, d.h. während der Operationsausführung eines solchen Operators werden zwei oder mehrere Bausteinoperationen in einer bestimmten zeitlichen Reihenfolge ausgeführt. Aus diesem Grund werden solche KR-Operatoren auch **Folgeschaltungen** genannt.

Welche Werte ein bestimmter KR-Operator berechnet, hängt davon ab, welche Tore in welcher Reihenfolge geöffnet werden. Die Öffnungsimpulse müssen von einem *Steueroperator* in der erforderlichen Reihenfolge generiert werden. Man betrachte unter diesem Aspekt noch einmal Bild 8.1 und nehme an, dass die Bausteinoperatoren Kombinationsschaltungen und die Operandenplätze Register sind. Das Stellen der Weichen in Bild 8.1a erfolgt durch Steuersignale, welche die Eingangstore der entsprechenden Register öffnen.

4 Man erkennt, dass die Schaltung (die graphische Darstellung) eines KR-Operators nichts anderes ist als ein Operandenflussplan, in welchem die Operatoren als Kombinationsschaltungen und die Operandenplätze einschließlich der Weichen als Register realisiert sind. ***KR-Operatoren*** *sind also Kompositoperatoren, die aus booleschen, d.h. rekursiven Operatoren nach der USB-Methode komponiert sind.* Demzufolge berechnen KR-Operatoren rekursive Funktionen und nur diese. Andrerseits ist jede rekursive Funktion USB-berechenbar und folglich auch KR-berechenbar, m.a.W. für jede rekursive Funktion kann ein KR-Operator für ihre Berechnung angegeben werden. Damit ergibt sich

Satz 4. Die Klasse der KR-berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Aus Satz 3 und Satz 4 folgt der

5 **Berechenbarkeits-Äquivalenzsatz :** *Statische Berechenbarkeit und rekursive Berechenbarkeit sind einander äquivalent*, oder anders ausgedrückt: *Die Klasse der* ***statisch berechenbaren*** *Funktionen ist mit der Klasse der* **rekursiven** *Funktionen identisch.*

Dies ist das Ergebnis einer langen Schlusskette, die bis in das Kapitel 8 zurückreicht. Wir wollen die Aussagen noch einmal Revue passieren lassen, die den Weg markieren. Dabei werden wir das Wort “Funktion” i.Allg. den Wörtern Operator und Operation vorziehen. Als USB-Funktionen hatten wir Funktionen bezeichnet, die nach der USB-Methode komponiert sind.

(1) Rekursive Funktionen sind USB-Funtionen [8.26], denn die rekursiven Komponierungsmittel (funktionale Substitution, Selektion, rekursive Iteration, Minimalisierung) sind mittels der USB-Methode beschreibbar (siehe Kap.8.4.5).

6 (2) Die aus rekursiven Operationen komponierbaren USB-Funktionen sind rekursive Funktionen [8.29], denn die Komponierungsmittel der USB-Methode (die Flussknoten) sind rekursiv beschreibbar, und nichtwohlstrukturierte Operatorennetze lassen sich in wohlstrukturierte überführen [8.28] (siehe Kap.8.4.5).

(3) Folglich sind USB-Funktionen rekursive Funktionen, denn sie werden aus den elementaren booleschen Funktionen komponiert [9.4], also aus rekursiven Funktionen [9.17].

(4) Die Klasse der USB-Funktionen ist mit der Klasse der rekursiven Funktionen identisch (als Folge von (1) und (3)).

(5) Die Klasse der KR-Funktionen ist mit der Klasse der USB-Funktionen identisch, denn KR-Funktionen werden aus den gleichen elementaren Funktionen mit Hilfe der gleichen Komponierungsmittel komponiert wie USB-Funtionen.
(6) Aus (4) und (5) zusammen mit obigem Satz 1 folgt der Berechenbarkeits-Äquivalenzsatz.

Er wird zur These von CHURCH, wenn “statisch berechenbar” durch “effektiv berechenbar” ersetzt wird. Der Äquivalenzsatz folgt also unmittelbar aus der churchschen These, die behauptet, dass die Klasse der *effektiv berechenbaren* (d.h. irgendwie tatsächlich berechenbaren) Funktionen mit der Klasse der *rekursiven* Funktionen identisch ist. Denn der Äquivalenzsatz schränkt die Klasse der *effektiv* berechenbaren Funktionen auf die *statisch* berechenbaren ein und schließt die mittels dynamischer Codierung berechenbaren Funktionen aus, falls es solche gibt. Doch hat der Äquivalenzsatz den Vorteil, dass er keine Hypothese, sondern unter der sehr allgemeinen Voraussetzung statischer Codierung *ableitbar* ist. Die Voraussetzung ist in traditionellen Computern (Prozessorcomputern) erfüllt. Wieweit sie in biologischen und in zukünftigen technischen informationellen Systemen erfüllt ist bzw. erfüllt sein wird, wissen wir nicht. Insofern bleibt die churchsche These eine Hypothese.

Doch jede Funktion, die durch eine “irgendwie” berechnete Wertetafel festgelegt wird, ist eine rekursive Funktion, unabhängig davon, ob das berechnende informationelle System mit statischer oder dynamischer Codierung arbeitet. Denn da die Wertetafel berechnet worden ist, kann sie nicht unendlich sein. Folglich ist sie in eine Kombinationsschaltung überführbar[9.16] , das heißt, sie ist eine rekursive Funktion [9.18].

Hinsichtlich des Exaktheitsanspruchs der Herleitung des Äquivalenzsatzes gilt auch hier die Schlussbemerkung von Kapitel 8. Die Herleitung stellt keinen strengen, mathematischen Beweis dar, doch ist sie logisch folgerichtig.

Die obigen 6 Aussagen können durch zwei weitere ergänzt werden, die schon jetzt *voraussagen*, zu welchem Ergebnis unsere weiteren Bemühungen führen werden. Unser Ziel ist der Prozessorcomputer. Er soll mit statischer Codierung arbeiten. Das hat zwei Konsequenzen. Zum einen folgt aus obigem Satz 1 die Aussage

(7) Der Prozessorcomputer und ist ein KR-Netz oder in ein solches überführbar.

Zum anderen folgt aus dem Berechenbarkeits-Äquivalenzsatz die Aussage

(8) Der Prozessorcomputer und speziell der Von-Neumann-Rechner kann alle rekursiven Funktionen berechnen und nur diese.

Dieser Schluss wird in Kap.13.7 auf anderem Wege bestätigt.

Unser Ziel ist der “universelle” Computer, der *sämtliche* rekursiven Funktionen berechnen kann. Wir müssen also einen “*programmierbaren*” Rechner entwerfen, der sich für die Berechnung jeder rekursiven Funktion konditionieren (programmieren) lässt. Das ist unser nächstes Ziel. Damit kehren wir zum eigentlichen Thema des Kapitels 13 zurück, zur Realisierung der dritten Grundidee des maschinellen Rechnens, der Programmsteuerung.

13.4 Taschenrechner

Zur Verwirklichung der dritten Grundidee des maschinellen Rechnens, der Programmsteuerung, wird ein Steueroperator benötigt, der *Programme interpretieren* kann; wir nennen ihn **interpretierenden Steueroperator**. Bevor wir darangehen, ihn zu entwickeln, wollen wir uns überlegen, wie ein einfacher, *nicht*programmierbarer Taschenrechner aufgebaut ist, der in der Lage ist, eine begrenzte Anzahl arithmetischer Operationen, zumindest die vier Grundrechnungsarten auszuführen. Durch Druck der entsprechenden Funktionstaste soll die gewünschte Funktion (Operation) aufgerufen, d.h. die Operationsausführung ausgeführt werden.

Unser Taschenrechner muss über Codeumsetzer zur Umcodierung zwischen dezimaler und binärer Zahlendarstellung verfügen, über Register zur Abspeicherung der Zahlen (der Argument- und Funktionswerte) und Operatoren, die den Argumentwerten die Funktionswerte zuordnen. Die *Arbeitsoberfläche* (Ein- und Ausgabemittel) ist seit langem standardisiert, sodass viele Leser mit ihr vertraut sein werden.

Zur Realisierung der Operatoren bieten sich zwei Möglichkeiten an:

1. Für jede Funktion wird eine Kombinationsschaltung entworfen.
2. Es wird ein einziger variabler KR-Operator und für jede Funktion ein Steueroperator entworfen, der die Tore des KR-Operators entsprechend steuert.

In beiden Fällen kann der Schaltungsaufwand dadurch verringert werden, dass *variable* Kombinationsschaltungen eingesetzt werden.

Bild 13.2 zeigt einen KR-Operator einschließlich Steueroperator für die Multiplikation durch iterative Addition. Der Addierer `ADD` ist ein Paralleladdierer. Der Übersichtlichkeit halber sind die Register bis auf den Taktverzögerer D nicht eingezeichnet. Der Multiplikator a wird dem Steueroperator, der Multiplikand b dem KR-Operator eingegeben. Dieser führt a-mal eine Addition mit b aus. Im ersten Schritt wird b zu 0 addiert.

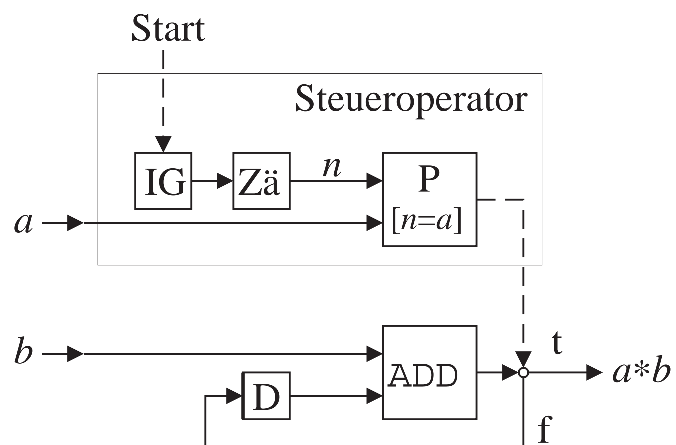


Bild 13.2 Multiplizierer als iterativer Addierer

Der Steueroperator enthält einen Impulsgenerator (IG), einen Zähler (Zä) und einen Vergleichsoperator, der mit P bezeichnet ist, um anzuzeigen, dass es sich um einen Prädikatoperator handelt[4]. Der Impulsgenerator generiert Taktimpulse, die der Zähler zählt, beginnend

mit dem Start der Multiplikation. In jedem Takt wird eine Addition ausgeführt. Der Zähler gibt also die Iterationszahl aus, die mit *n* bezeichnet ist. Der nachfolgende Vergleichsoperator vergleicht *n* mit *a* und gibt das erforderliche Steuersignal zur Steuerung der Zweigeweiche aus. Solange das Prädikat [*n*=*a*] nicht erfüllt ist, wird die vom Addierer berechnete Summe auf den Eingang des Addierers zurückgegeben. Sobald das Prädikat erfüllt ist, wird die Iteration beendet und die Summe, also das Produkt *a***b*, ausgegeben. (Es wird angenommen, dass *a* ganzzahlig ist; Verallgemeinerung auf Dezimalzahlen bereitet keine Schwierigkeiten.) Der Vergleichsoperator hat also das Prädikat [*n*=*a*] zu *entscheiden*.

KR-Operator und Steueroperator sind Kompositoperatoren der zweiten Komponierungsstufe. Die erste Stufe beinhaltet die Komponierung von Kombinationsschaltungen, die zweite die Komponierung eines KR-Operators. Der Steueroperator seinerseits benötigt keinen übergeordneten Steueroperator, sondern lediglich einen Eingang für den Startimpuls, der durch Druck auf die Multiplikationstaste generiert wird.

Wenn der Wert von *a* sehr hoch ist, kann die Multiplikation einige Zeit in Anspruch nehmen. In Bild 13.3 ist eine effektivere Variante gezeigt. Ihre Arbeitsweise beruht auf der Methode, die man in der Schule gelernt hat, nach der zwei Zahlen schriftlich multipliziert werden. Der Faktor *a* ist in dem Parallelregister R1 und *b* ist in dem Schieberegister R2 eingespeichert, wobei das letzte (rechte) Bit der Kette der ersten (höchsten) Stelle von *b* entsprechen muss, sodass sie als erste vom Schieberegister ausgegeben ("eingelesen") wird, wie es bei Taschenrechnern üblich ist. Das Einlesen in R2 kann auch parallel erfolgen. Wenn der Leser die Arbeitsweise nachvollziehen möchte, muss er sich zweierlei klarmachen.

1. Nach dem Schulalgorithmus wird der erste Faktor jeweils mit einer Stelle des zweiten Faktors multipliziert, beginnend mit der ersten Stelle. Diese stellenweise Multiplikation führt der $\underline{\texttt{AND}}$-Operator durch. Die Unterstreichung bedeutet, dass der Operator aus so vielen elementaren AND-Operatoren besteht, wie die Wortlänge des Taschenrechners angibt. Dass der $\underline{\texttt{AND}}$-Operator die Multiplikation von *a* mit jeweils einer Stelle von *b* ausführt, ist leicht einzusehen. Wenn nämlich die laufende Stelle des zweiten Faktors (das laufende, d.h. zuletzt aus dem Schieberegister R2 herausgeschobene Bit der Bitkette, die den Faktor *b* codiert) den Wert 1 besitzt, muss der $\underline{\texttt{AND}}$-Operator die Bitkette des ersten Faktors (*a*) liefern, andernfalls eine Kette von Nullen. Die Bits des Resultats der Stellenmultiplikation ergeben sich demnach aus der Konjunktion der jeweiligen Bits der Bitketten der Faktoren.

2. Nach dem Schulalgorithmus werden die so berechneten Ergebnisse (die Ausgaben des $\underline{\texttt{AND}}$-Operators) untereinander, aber jeweils um eine Stelle nach rechts verschoben aufgeschrieben und dann addiert. Die Verschiebung um eine Stelle entspricht bei Dezimalzahlen einer Multiplikation mit 10, bei Binärzahlen einer

4 Operatoren, die Prädikate entscheiden, hatten wir in Kap.8.4.5 Prädikatoperatoren genannt.

Multiplikation mit 2, in beiden Fällen also dem Anhängen einer 0. Die Multiplikation wird in der Rückkopplungsschleife ausgeführt. In Bild 13.3 sind der Steueroperator und einige weitere Details des KR-Operators unterschlagen, um nicht vom Wesentlichen abzulenken.

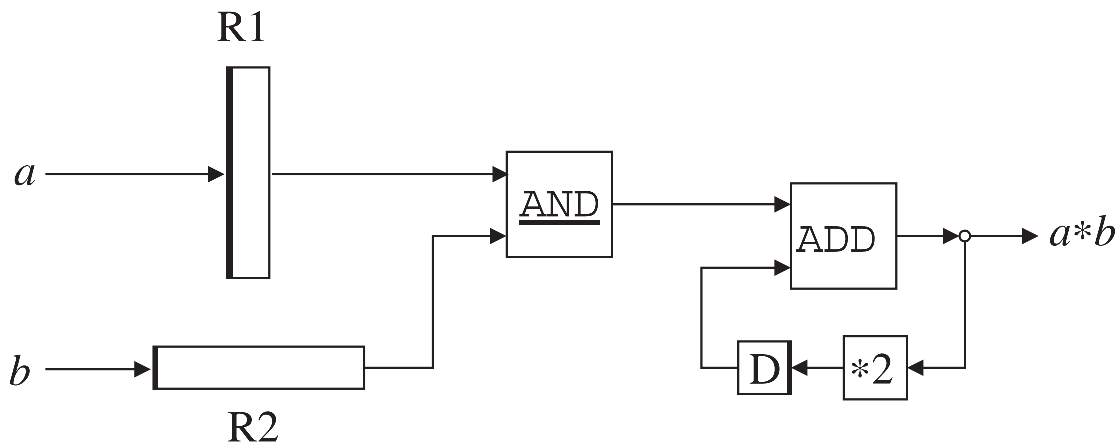


Bild 13.3 Multiplizierer mit Stellenverschiebung

Wir wollen uns überlegen, wie der Steueroperator entworfen und realisiert werden kann. Dazu greifen wir auf Kap.12.3.4 zurück, wo wir uns eine Methode für den Entwurf eines Steueroperators zur Steuerung einer Waschmaschine ausgedacht haben [12.5]. Danach ist zunächst die Entscheidungstabelle des Steueroperators aufzuschreiben und anschließend als ROM zu realisieren. Im Falle des Multiplizierers von Bild 13.2 genügt es, in die Bedingungsspalte der Entscheidungstabelle lediglich die laufende Taktnummer einzutragen, denn der Multiplizierer arbeitet getaktet und der Steueroperator empfängt keinerlei Rückmeldungen vom KR-Operator, sodass die Bausteinoperationen, die der Reihe nach auszuführen sind, nur von der laufenden Taktnummer abhängen. In die Aktionsspalte sind die Steuerwörter einzutragen. Ein Steuerwort enthält die Steuersignale für sämtliche (auch die nicht eingezeichneten) Eingangstore der Register bzw. Ein-Bit-Speicher in der erforderlichen Reihenfolge. Die Entscheidungstabelle, die sich so ergibt, stellt eine Sonderform dar. Es braucht nämlich nicht in jedem Schritt nach der Zeile mit der aktuellen Bedingung *gesucht* zu werden, sondern die Zeilen werden der Reihe nach abgearbeitet.

Es gibt noch viele andere Möglichkeiten, einen Multiplizierer zu entwerfen. Beispielsweise kann der Addierer sequenziell arbeiten. Arbeits- und Steueroperator sind dann Kompositoperatoren der dritten Stufe. Der Steueroperator des Addieres ist dem des Multiplizierers untergeordnet, sodass sich eine *Steuerhierarchie* ergibt.

Auf analoge Weise wie der Multiplizierer lässt sich jede andere Operation realisieren, die unser Taschenrechner “können” soll, d.h. für die er über eine Funktionstaste verfügen soll. Dabei lassen sich sowohl die Arbeitsoperatoren (die Opera-

toren des KR-Netzes) als auch die Steueroperatoren als ROM realisieren. Der Taschenrechner stellt dann eine *ROM-Hierarchie* dar.

13.5 Prozessor

13.5.1 Idee des Prozessors und seiner Programmierung

Da sich für jede berechenbare Funktion ein gesteuerter KR-Operator angeben lässt, könnte man auf die Idee kommen, einen universellen Rechner dadurch zu realisieren, dass man einen universellen KR-Operator und zahllose Steueroperatoren baut. Die Idee stößt auf dieselbe Grenze wie die Idee von den zahllosen Kombinationsschaltungen, die wir bereits in Kap.9.2.1 verworfen hatten. Ihre Verwirklichung ist nur dann sinnvoll, wenn die Anzahl der zu berechnenden Funktionen gering ist, wie im Falle eines Taschenrechners oder eines Steuerrechners, der ein hinsichtlich der Steuerung relativ unkompliziertes Objekt steuert, z.B. eine Waschmaschine, einen Fotoapparat oder einen Automotor.

Eine andere Idee besteht darin, nicht eine Menge, sondern eine Hierarchie von Steueroperatoren zu entwerfen. Dieser Weg war hinsichtlich des Taschenrechners angedeutet worden. Auch er ist möglich, aber wiederum nur für Spezialfälle. Beispielsweise kann es zweckmäßig sein, einen Spezialrechner als ROM-Hierarchie zu konzipieren, wenn er Funktionen berechnen soll, die oft auftreten, aber zu kompliziert sind, um das einfache Taschenrechnerprinzip anwenden zu können. Diese Situation ist z.B. für statistische Auswertungen charakteristisch. Aber mit derartigen *reinen Hardwarelösungen* ist kein universeller Rechner zu bauen, sodass nach einer *Softwarelösung* gesucht werden muss. Gesucht ist ein universeller Rechner in Form eines gesteuerten KR-Operators, der beliebige Operationsvorschriften (Vorschriften zur Berechnung beliebiger rekursiver Funktionen) ausführen kann; gesucht ist ein per Programm beliebig steuerbarer oder *frei programmierbarer KR-Operator*.

Historisch führte der Weg zum programmierbaren Rechner über viele Ideen und realisierte Schaltungen. Relativ schnell hat sich eine Schaltung durchgesetzt, die nachträglich **Von-Neumann-Rechner** genannt worden ist. Sie besteht im Wesentlichen aus einem **Prozessor** und dessen **Arbeitsspeicher**, auch **Hauptspeicher** genannt (siehe Bild 13.7). In seiner einfachsten Form ist der Prozessor ein KR-Netz mit einem einzigen Bausteinoperator, einer steuerbaren Kombinationsschaltung. Sie wird **arithmetisch-logische Einheit** oder kurz **ALU** genannt (U für unit).

Eine ALU kann einige sehr einfache arithmetische Operationen wie Inkrementieren, Dekrementieren, Addieren und Subtrahieren ausführen, ferner Verschiebungen von Bitketten nach rechts und links sowie einige boolesche Operationen. Die Einstellung ("*Konditionierung*") auf eine bestimmte Funktion erfolgt durch Steuerung von Weichen, die in die Kombinationsschaltung eingebaut sind.

Die Beschränkung auf die ALU als einzigen Bausteinoperator des KR-Operators bedeutet den Übergang von *verteilter Verarbeitung* durch mehrere Operatoren eines

Operatorennetzes zur *zentralen Verarbeitung* durch einen einzigen Arbeitsoperator, die ALU, und in dem damit verbundenen Übergang von *verteilter* zu *zentraler Speicherung*. Aus den *lokalen* Operandenspeichern, die über ein KR-Netz *verteilt* sind, werden **Speicherplätze** des Hauptspeichers. Wir erinnern uns, dass die gedankliche Zusammenfassung der Speicherplätze eines Operatorennetzes zu einer einzigen Speichereinheit der Idee des *endlichen Automaten* zugrunde liegt (vgl. Kap.8.2.3 [8.10]). Mit der fundamentalen Bedeutung der Zentralisierung für die Rechentechnik, insbesondere für die Programmierungstechnik werden wir uns in Kap. 13.7 beschäftigen. Im Augenblick interessieren wir uns für ihre Konsequenzen hinsichtlich der Funktionsweise des Prozessors, den wir entwerfen wollen.

Die Beschreibung einer Kompositoperation mittels eines Operandenflussplanes verliert ihren eigentlichen Sinn, da es auf der Ebene der Programmierung des Prozessors kein Netz aus mehreren Operatoren gibt, zwischen denen die Operanden fließen könnten. Vielmehr muss dem Prozessor mitgeteilt werden, welche *Aktion* die ALU als nächste auszuführen hat, also welche Operation mit welchen Operanden. Ein Programm, das eine Folge von Aktionen vorschreibt, nennen wir **Aktionsfolgeprogramm**. Es besteht aus einer Folge von **Befehlen**, je ein Befehl für jede Aktion.

Die graphische Darstellung eines Aktionsfolgeprogramms nennen wir **Aktionsfolgeplan**. Ein Aktionsfolge*programm* ist also die maschinenverständliche sprachliche Artikulierung eines Aktionsfolge*plans*, m.a.W. es ist ein maschinenverständli-

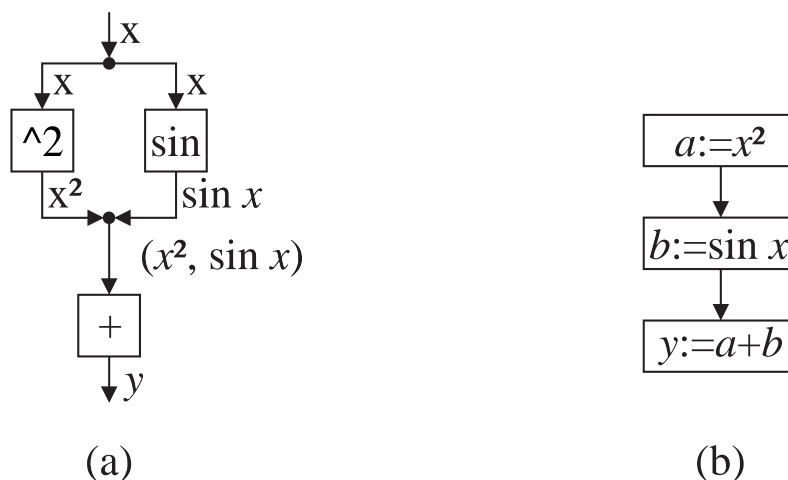


Bild 13.4 Graphische Darstellung der Berechnung der Funktion $y = x^2 + \sin x$. (a) - Datenflussplan; (b) - Aktionsfolgeplan. ^2 ist als “hoch 2" zu lesen.

cher imperativer Algorithmus. Diesem Sachverhalt entspricht der Sprachgebrauch der Informatiker, wonach Aktionsfolgeprogramme als **imperative Programme** bezeichnet werden[5]. Bild 13.4b zeigt den Aktionsfolgeplan zur Berechnung der Funktion $y = x^2 + \sin x$. Die Reihenfolge der ersten beiden Befehle kann umgekehrt werden. Zum Vergleich ist in Bild 13.4a der entsprechende Operandenflussplan

dargestellt. Man beachte die unterschiedliche Bedeutung der Pfeile. Ein Pfeil des Operandenflussplans stellt den Übergabeweg eines Operanden dar; im Aktionsfolgeplan zeigt er auf die nächste Aktion.

Ein charakteristisches Problem der Aktionsfolgeprogrammierung (der imperativen Programmierung) ist der Operandentransport. Infolge der Zentralisierung der Speicherung müssen die Operanden der Bausteinoperationen (der Operationen der ALU) von und zum zentralen Speicher transportiert werden. Ein Steueroperator, der Bestandteil des Prozessors ist, hat dafür zu sorgen, dass die Befehle in der richtigen Reihenfolge ausgeführt und die dabei notwendigen Operandentransporte ausgeführt werden. Wenn wir darangehen einen Prozessor zu entwerfen, müssen wir also unser Augenmerk darauf richten, dass der Steueroperator eine sequenzielle Aktionsfolge steuert und dass er bei jeder Aktion folgende *Aktionsschritte* auszuführen hat:

1. Holen des aktuellen Befehls,
2. Konditionieren der ALU (Einstellung auf die konkrete Operation),
3. Versorgen der ALU mit den aktuellen Operanden und Berechnen des Resultats.
4. Abspeichern des Resultats.
5. Berechnung der nächsten Befehlsadresse

Diese Folge wird aus der abstrakten Sicht des Computerentwurfs **von-neumannsches Operationsprinzip** und aus der konkreten Sicht der internen Semantik (der Prozesse im Computer) **zentrale Steuerschleife** genannt. Der fünfte Aktionsschritt ist der Vollständigkeit halber hinzugefügt. Er wird weiter unten besprochen. In Kap.16.5 [16.14] wird ein sechster Aktionsschritt eingeführt, welcher der Behandlung von Unterbrechungen dient.

Es mag überraschen, dass die zentrale Steuerschleife keinen speziellen Aktionsschritt für die eigentliche Operationsausführung enthält, für die Zuordnung des Resultatwertes zu den Argumentwerten. Der Grund ist folgender. Die Zuordnung wird von der ALU ausgeführt und diese ist eine Kombinationsschaltung, besitzt also keinen Speicher. Der Zuordnungsprozess ist ein Übergangsprozess in der ALU und enthält als solcher keinen Zeitpunkt der kausaldiskreten Prozessbeschreibung (siehe Kap.9.4 [9.2)). Die Zuordnung erfolgt, sobald die Operanden am ALU-Eingang liegen und gehört zum Aktionsschritt 3.

Damit der Prozessor einen Befehl holen und ausführen kann, müssen ihm die **Adressen** des Befehls, der Operanden und des Resultats sowie die auszuführende Operation mitgeteilt werden. Diese Angaben müssen in jedem Befehl enthalten sein, entweder explizit oder implizit (aus den expliziten Angaben ableitbar). Dabei wird vorausgesetzt, dass der Steueroperator auf die Speicherplätze eines Arbeitsspeichers über Adressen zugreifen kann. Außerdem muss jeder Befehl ein Codewort für die auszuführende ALU-Operation enthalten, den sog. **Operationscode** (**OC**).

5 Mit der Einführung des Begriffs des imperativen Algorithmus in Kap.7.2 [7.10] sollte der Begriff des imperativen Programms vorbereitet werden.

An dieser Stelle sei eine Bemerkung zur Speicherorganisation eingeschoben. Ein Computer hantiert mit zwei Objektklassen, mit Programmen (sprachlichen Operatoren) und mit Daten (Operanden technischer sprachlicher Operatoren)[6], sodass es naheliegt, ihn mit zwei Speichern auszurüsten, einem Programmspeicher und einem Datenspeicher. Der ungarisch-amerikanische Mathematiker und Physiker John von Neumann hat vorgeschlagen, Befehle und Daten in einem einzigen Speicher aufzubewahren und einheitlich zu behandeln. Genauer gesagt hat von Neumann mit seiner Autorität diesem, später nach ihm benannten Speicherprinzip zur allgemeinen Anerkennung und zum technischen Durchbruch verholfen. Ursprünglich stammte die Idee von J. Presper Eckert und John W. Mauchly[7].

Die gemeinsame Speicherung von Befehlen und Daten hat zur Folge, dass in einem Befehl anstelle einer Operandenadresse eine Befehlsadresse auftreten kann, sodass eventuell die *Be*arbeitung eines Befehls veranlasst wird (nicht die *Ab*arbeitung, d.h. Ausführung). Das bedeutet eine Relativierung der begrifflichen Unterscheidung zwischen Programmen und Daten oder zwischen Operatoren und Operanden. Wir hatten uns schon früher davon überzeugt, dass die Unterscheidung nicht konsequent durchführbar ist, und es sinnvoll sein kann, einen gemeinsamen Oberbegriff zu bilden, wie es im Lambda-Kalkül gehandhabt wird. Diese generalisierende Abstraktion liegt auch dem von-neumannschen Vorschlag zugrunde.

Wir setzen den unterbrochenen Gedankengang fort und ziehen folgenden Schluss. Damit die genannten Aktionsschritte der zentralen Steuerschleife ausgeführt werden können, muss die ALU in ein KR-Netz eingebettet werden, in dem die erforderlichen Befehls- und Datentransporte stattfinden. Dieses Netz nennen wir **RALU** (**R**egister und **ALU**). Die Weichen der RALU sind von einem Steueroperator zu steuern. Die gedankliche Vereinigung der RALU mit dem Steueroperator führt zum Begriff des **Prozessors** (siehe Bild 13.7)[8].

Bevor wir damit beginnen, die Schaltung eines Prozessors zu entwerfen, überlegen wir uns, wie die Sprache etwa auszusehen hat, die der Prozessor verstehen (interpretieren), d.h. in Steuersignale umsetzen soll. Die Sprache muss die Möglichkeit bieten, maschinenlesbare Aktionsfolgen, also maschinenlesbare *imperative Algorithmen*, d.h. *imperative Programme* zu artikulieren. Eine solche Sprache heißt **imperative Sprache**.

---

6 In einem verallgemeinerten Sinn werden als Daten häufig beliebige (auch beliebig lange) Bitketten bezeichnet, die in Rechnern oder Rechnernetzen transportiert werden.

7 In diesem Zusammenhang ist der Artikel [Bauer 98] sehr aufschlussreich.

8 Die Komponierung des Prozessors aus RALU und Steueroperator entspricht nicht unbedingt den Darstellungen in der Literatur; die Struktur eines Prozessors kann komplizierter sein. Wir begnügen uns mit der sehr einfachen in Bild 13.7 dargestellten Struktur. Sie reicht aus, um die Arbeitsweise eines Prozessors im Prinzip zu verstehen.

13.5.2 Maschinensprache

Damit der zu entwerfende Prozessor einen imperativen Algorithmus ausführen kann, muss dieser in einer Sprache geschrieben sein, die der Prozessor "versteht". Wir nennen sie **Prozessorsprache**. Das lässt sich dadurch erreichen, dass der Aufbau der Befehle, aus denen der Algorithmus besteht, standardisiert und der Prozessor mit einem speziellen Register für die Aufnahme eines standardisierten Befehls ausgerüstet wird. Dieses Register heißt **Befehlsregister**, abgekürzt **BR**. Die Standardisierung schreibt vor, in welcher Reihenfolge die Bestandteile eines Befehls, also der Operationscode und die Adressen zu einem Binärwort (i.d.R. identisch mit dem *Rechnerwort*) zu verketten sind, man spricht von *Befehlsformatierung*. Dem **Befehlsformat** muss der Aufbau (das "Format") des Befehlsregisters genau entsprechen.

Diese *Formatentsprechung* ermöglicht durch sequenzielles Laden der Befehle eines Programms in das Befehlsregister die *direkte* Abarbeitung; "direkt" bedeutet hier: ohne vorherige Übersetzung in eine andere Sprache oder sonstige Bearbeitung. Durch die Formatentsprechung wird die Sprache an die Schaltung "angekoppelt". Man kann auch hier, ähnlich wie im Falle des Wort-Leitung- und Leitung-Wort-Zuordners, von *Schnittstelle* oder *Interface* zwischen Schaltung und Sprache, zwischen Hardware und Software sprechen.

Eventuell muss zwischen Prozessorsprache und Maschinensprache unterschieden werden. Wir vereinbaren: *Eine Programmiersprache, die ein direktes Interface mit einem Computer besitzt, heißt* ***Maschinensprache*** *des Computers. Ein Programm, das in einer Maschinensprache geschrieben ist, heißt* ***Maschinenprogramm***. Die Maschinensprache eines Computers, der einen einzigen Prozessor enthält, ist mit der Prozessorsprache identisch. Soweit von Einprozessorrechnern die Rede ist, können die Wörter *Maschinensprache* und *Prozessorsprache* als Synonyme verwendet werden.

Neben dem Befehlsformat muss auch das **Programmformat** festgelegt werden, d.h. die genaue Anordnung der Befehle innerhalb eines Programms. Wenn ein **Operationsrepertoire** (die Menge der zur Verfügung stehenden Operationscodes), ein Befehlsformat und ein Programmformat vorgegeben sind, ist damit die *Syntax* einer Maschinensprache festgelegt. Man beachte, dass mit der Syntax auch die interne Semantik der Maschinensprache festgelegt ist. Denn aus der Schaltung der Maschine ergibt sich zwangsläufig die interne Semantik eines Befehls bzw. eines Programms, d.h. der Prozess, der bei der Ausführung (Interpretation) des Befehls bzw. Programms in der Maschine abläuft.

Im vorangehenden Kapitel hatten wir uns überlegt, welche Informationen ein Befehl enthalten muss, damit der Prozessor obige Aktionsschritte ausführen kann. Auf dieser Grundlage wollen wir eine Maschinensprache entwerfen.[9] Dazu legen wir fest:

---

9 Es sei an die Bemerkung am Ende des Kapitels 13.1 erinnert, dass die Darlegungen dieses

1. Ein Befehl besteht aus 5 Feldern bestimmter Länge. In die Felder werden der Reihe nach die Befehlsadresse (BA), der Operationscode (OC), die Adressen der Operanden (A1 und A2) und die Adresse des Resultats (A3) eingetragen (siehe die oberste Zeile in Bild 13.5).
2. Ein Programm ist eine Tabelle, deren Zeilen je einen Befehl und deren Spalten jeweils die gleichen Elemente der Befehle enthalten.
3. Das Codewort der Addition ist `ADD`, das der Subtraktion `SUB`.

Damit ist eine Tabellensprache für eine **Vier-Adress-Maschine** definiert, d.h. für einen Rechner, der pro Aktion mit bis zu 4 Adressen hantieren kann. Bild 13.5 zeigt ein Programm, das in dieser Sprache geschrieben ist für die Berechnung des Wertes von r nach der Formel (Ergibtanweisung)

$$r := (a - b) + c \tag{13.1}$$

Dabei sind den Variablen in (13.1) Speicheradressen zugewiesen, beispielsweise der Variablen a die Adresse 3010 (siehe den Kommentar in Bild 13.5).

Das Programm besteht aus zwei Zeilen. Um das Lesen zu erleichtern, ist in der Kopfzeile (sie gehört nicht zum Programm) angegeben, welche Befehlskomponenten

BA	OC	A1	A2	A3
3000	SUB	3010	3011	3012;
3001	ADD	3012	3013	3014;

Kommentar:

3000	-	Adresse des ersten Befehls, Startadresse
3001		Adresse des zweiten Befehls
3010	-	Adresse von a
3011	-	Adresse von b
3012	-	Adresse von d
3013	-	Adresse von c
3014	-	Adresse von r

Bild 13.5 Maschinenprogramm einer Vier-Adress-Maschine zur Berechnung des durch (13.1) festgelegten Ausdrucks. BA - Befehlsadresse, OC - Operationscode, A1, A2 - Operandenadressen, A3 - Resultatadresse.

Kapitels auf den Fachmann eventuell einen antiquierten Eindruck machen. Wenn im Weiteren verschiedene Maschinensprachen vorgeschlagen werden, kommt es darauf an, die Prinzipien, nach welchen Prozessoren arbeiten und programmiert werden, deutlich erkennbar zu machen. Ob der Prozessor, den wir erfinden werden, in genau der beschriebenen Form tatsächlich existiert, ist nebensächlich.

in den einzelnen Spalten eingetragen sind. Auch die Abstände zwischen den Feldern sind wegen der besseren Lesbarkeit eingefügt, obwohl sie nicht zum Programmtext gehören. Die Adressen sind als Dezimalzahlen angegeben. Das setzt voraus, dass der Rechner, der das Programm ausführen soll, über einen Codeumsetzer verfügt, der Dezimalzahlen in Dualzahlen umcodiert. Auch die Operationscodes müssen umcodiert, d.h. in die entsprechenden vorgeschriebenen Bitketten überführt werden.

Die Abarbeitung des Programms beginnt mit dem "Holen" des ersten Befehls d.h. mit dessen Transport aus der HS-Zelle (Speicherplatz des Hauptspeichers) mit der Adresse 3000 in das Befehlsregister BR[10]. Aus dem Kommentar ergibt sich, dass in der ersten Aktion die Differenz *a-b* berechnet werden soll. Zu diesem Zweck müssen zunächst die Werte von *a* und *b* aus den Speicherzellen 3010 bzw. 3011 geholt und in zwei weitere reservierte Register geladen werden, in ein Datenregister (DR) und in ein Register, das wir aus später erkennbaren Gründen *Akkumulator* (AC) nennen. Von da werden sie der ALU zugeführt, die vorher auf Subtraktion konditioniert werden muss. Schließlich wird das Ausgabewort der ALU unter der HS-Adresse 3012 abgespeichert.

Die Ausführung des zweiten Befehls verläuft analog. Über den Speicherplatz mit der Adresse 3012 wird der Wert der Differenz *a-b* an die zweite Aktion übergeben. Diesem Speicherplatz entspricht keine der drei Variablen in (13.1). Die dort gespeicherte Variable stellt eine *Hilfsvariable* dar; wir bezeichnen sie mit *d*. Im Auftreten der Hilfsvariablen *d* spiegelt sich das Grundprinzip des imperativen Programmierens wider. Der Leser erinnere sich an die Definition des imperativen Algorithmus in Kap.7.2 [7.10]. Dort war anhand des gleichen Rechenbeispiels zuerst der Begriff der *Aktion* als Operation an explizit angegebenen Operanden eingeführt worden und anschließend der Begriff des *imperativen Algorithmus* als Sequenz von Imperativsätzen, wobei je ein Imperativsatz eine Aktion vorschreibt. Man beachte, dass die beiden Befehle in Bild 13.5 die beiden Berechnungsschritte darstellen, in die wir den Ausdruck *a-b+c* in Kap. 7.2 [7.11] zerlegt hatten. 11

Unser Programm kann jedem Argumentwertetripel (*a,b,c*) einen Funktionswert *r* zuordnen, es ist ein formaler Operator, der die durch (13.1) festgelegte Funktion berechnet. Die gewünschten Argumentwerte müssen vor dem Start des Programms unter den betreffenden Adressen eingespeichert werden (s.u.). Die beschriebene Vorgehensweise ist ziemlich umständlich und die Frage ist berechtigt, warum man ein Programm schreiben soll, um eine Subtraktion und eine Addition auszuführen. Mit einem Taschenrechner ist die Berechnung einfacher und ökonomischer, jedenfalls wenn man nur wenige Funktionswerte berechnen will. Muss man die Operation

10 Die folgenden Überlegungen bilden die Grundlage für den Entwurf des Prozessors einer Zwei-Adress-Maschine, dessen Schaltung in Bild 13.7 gezeigt ist. Denjenigen Lesern, die gerne Denksportaufgaben lösen, wird vorgeschlagen, sich selber eine Schaltung auszudenken. Wer davon Abstand nehmen will, kann vorblättern und den Prozess der Befehlsausführung anhand des Bildes 13.7 verfolgen.

jedoch hundert- oder gar tausendmal ausführen, so kann sich das Programmieren schon lohnen, zumal es viele Mittel gibt, das Programmieren zu vereinfachen.

Eine bedeutende Erleichterung, die bereits der Taschenrechner gewährt, besteht in der Möglichkeit, mit dem Ergebnis der Subtraktion sofort weiterzurechnen, ohne das Zwischenresultat explizit abzuspeichern. Die Methode, die das ermöglicht, ist uns bekannt; sie heißt *Rückkopplung*. Im vorliegenden Fall besteht sie in der Rückführung des Ausganges der ALU auf ihren Eingang. Das Register in der Rückkopplungsschleife ist der bereits genannte **Akkumulator**. Man beachte, dass sich durch den Akkumulator die explizite Angabe von Operanden teilweise erübrigt. Genau genommen liegt hier eine erste Abweichung vom Prinzip der konsequenten Aktionsfolgeprogrammierung vor.

Der Akkumulator bringt mehrere Vorteile. Er verkürzt das Befehlsformat um eine Operandenadresse, er spart den Hauptspeicherplatz für das Zwischenergebnis ein und er verkürzt die Ausführungszeit, da zwei Zugriffe auf den HS entfallen. Er bringt aber auch einen Nachteil. Es ist ein spezieller Befehl für den Transport von Operandenwerten vom HS zum AC notwendig, der immer dann zum Einsatz kommt, wenn die nächste Aktion nicht mit dem im AC aufbewahrten, sondern einem anderen Wert ausgeführt werden soll. Dadurch kann das Programm länger werden.

Das Beispiel zeigt anschaulich, wie eng Sprache und Hardware miteinander verquickt sind. Beide müssen gemeinsam entworfen werden. Dabei ist ein Kompromiss zwischen mehreren Zielen zu schließen:

- niedriger Hardwareaufwand,
- niedriger Programmieraufwand,
- kurze Ausführungszeiten,
- geringer Speicherplatzbedarf.

Das Suchen nach dem optimalen Kompromiss zieht sich durch die gesamte Hardware-, Software- und Sprachentwicklung.

Neben dem Akkumulator hat sich eine zweite Idee zur Verkürzung des Befehlsformats durchgesetzt, die Einführung eines **Befehlszählers** (**BZ**). Sie geht davon aus, dass die Befehle eines Programms der Reihe nach im HS gespeichert werden, sodass der nächste Befehl die nächst höhere Adresse besitzt (vorausgesetzt, dass ein Befehl nicht mehr als eine Speicherzelle beansprucht). Wenn vor dem Start eines Programms dafür gesorgt wird, dass die Adresse des ersten Befehls in den Befehlszähler geladen wird, können die folgenden Befehlsadressen durch laufende Inkrementierung des Inhaltes des Befehlszählers berechnet werden, sodass sich ihre Angabe in den weiteren Befehlen erübrigt. Die Adresse des ersten Befehls, die sogenannte **Startadresse**, wird i.d.R. automatisch vom Computer (genauer vom sog. Betriebssystem; siehe Kap.19.5) beim “Laden” des Programms in einen freien Speicherbereich festgelegt".

Damit enthält ein Befehl neben dem Operationscode nur noch höchstens zwei Adressen, und aus unserer Vier-Adress- wird eine Zwei-Adress-Maschinensprache. In ihr ist das Programm von Bild 13.6 geschrieben, das die durch (13.1) festgelegte

Funktion berechnet. TVS ist der Operationscode für den oben erläuterten **T**ransport **V**om **S**peicher zum AC. Später werden wir auch einen TNS-Befehl benötigen für den **T**ransport **N**ach dem **S**peicher. Der END-Befehl beendet die Abarbeitung des Programms. Der besseren Lesbarkeit halber sind anstelle der Adressen die Variablenbe-

```
BEGIN
   TVS  <a>
   SUB  <b>
   ADD  <c>  <r>
END
```

Bild 13.6 Programm einer Zwei-Adress-Maschine zur Berechnung von *r* gemäß (13.1). Die spitzen Klammern zeigen an, dass im Programm nicht der Bezeichner, sondern die betreffende Adresse zu stehen hat.

zeichner in spitzen Klammern eingetragen. Beispielsweise stellt `<a>` die Adresse dar, unter der die Werte der Variablen *a* abgespeichert werden.

Nach Ausführung des ersten Befehls (Transport vom Speicher) steht im Akkumulator der Wert von *a*, nach der Subtraktion der Wert der Differenz *a-b* und nach der Addition der Wert von *r*, der dann dem HS übergeben wird. Für die Angabe des Speicherplatzes für *r* steht das zweite Adressfeld des Additionsbefehls zur Verfügung. Nach Ausführung des `ADD`-Befehls steht der Wert von *r* sowohl im HS unter der Adresse <r>, als auch im AC, denn ein Register wird durch Auslesen seines Inhalts nicht “geleert”.

13.5.3 Arbeitsweise der RALU

Vielleicht wird mancher Leser schon ungeduldig angesichts aller möglichen scheinbar kaum zur Sache gehörenden Überlegungen, und es wird Zeit, dass wir aufhören, um den universellen Rechner wie die Katze um den heißen Brei herumzuschleichen. Der Brei ist genügend abgekühlt. Die Aufgabe “Entwerfe eine Zwei-Adressmaschine” können wir “in den Mund nehmen” (artikulieren), ohne Gefahr zu laufen, uns zu verbrennen, d.h. an ihr zu scheitern. Die Aufgabe ist inzwischen so scharf umrissen, dass wohl jeder, dem Denksport Spaß macht, nach einigem Probieren eine Schaltung zusammengebastelt haben wird, die das Programm von Bild 13.6 ausführen kann und die der in Abb.13.7 dargestellten Schaltung mehr oder weniger ähnlich ist.

Wer nicht die Muße hat, *seine* Prozessorschaltung zu erfinden, kann die Arbeitsweise des Prozessors anhand der Schaltung von Bild 13.7 nachvollziehen. Doch auch dies kann er unterlassen; für das weitere Verständnis ist es nicht erforderlich. Er muss sich jedoch merken, was genau der Prozessor “tut”. Wir wiederholen es noch einmal. *Er konditioniert die ALU und versorgt sie mit Operanden (was zur Ausführung der*

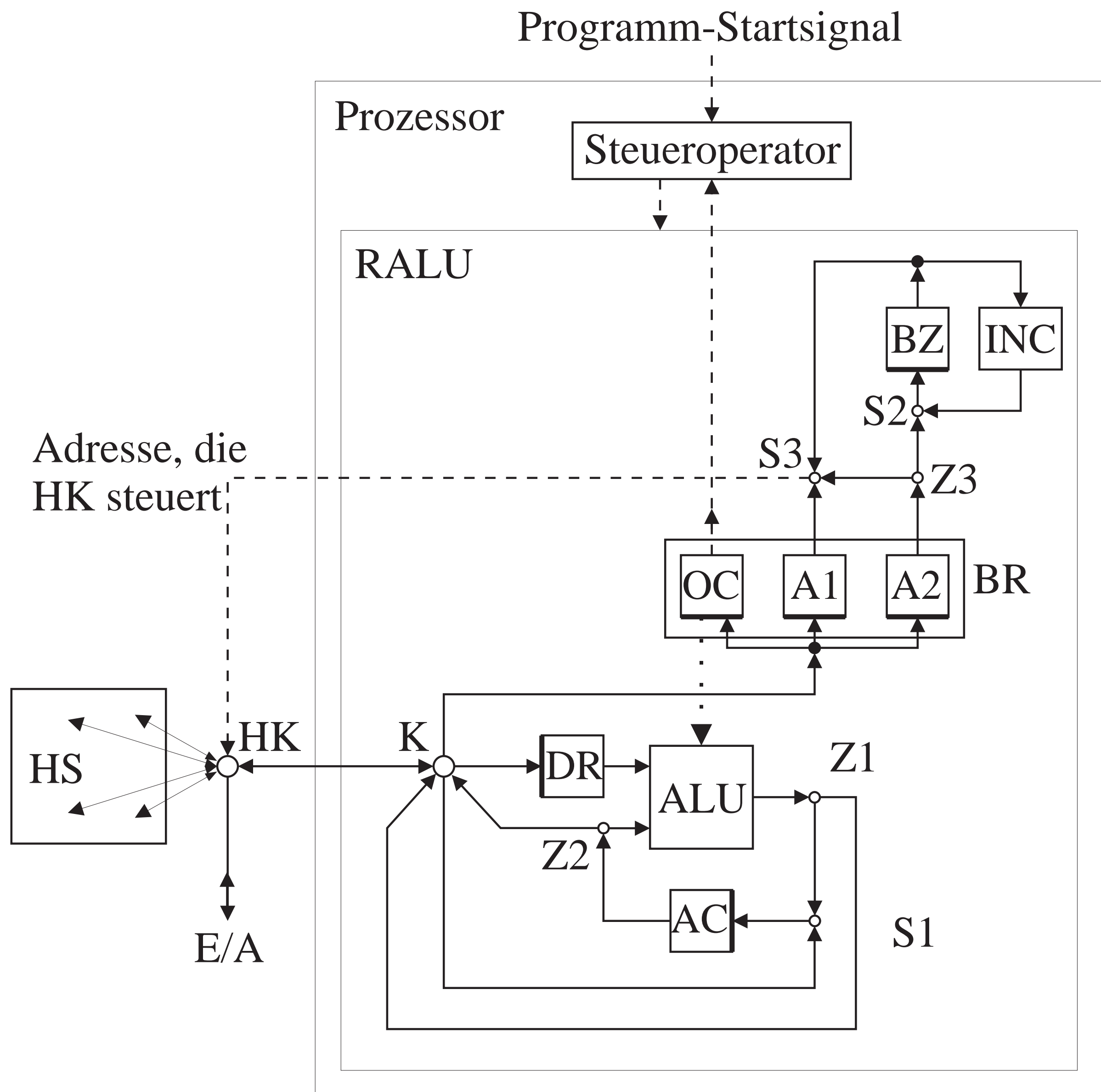


Bild 13.7 Prinzipschaltung eines Einprozessorrechners mit dekomponierter RALU.
Bezeichnungen:

HS	-	Hauptspeicher
ALU	-	Arithmetisch-Logische Einheit
RALU	-	Register+ALU
AC	-	Akkumulator
DR	-	Datenregister
BR	-	Befehlsregister
OC	-	Register für Operationscode
A	-	Register für Adresse
BZ	-	Befehlszähler
INC	-	Inkrementierer
K	-	Kommutator
HK	-	Halbkommutator
S	-	Sammelweiche
Z	-	Zweiwegeweiche
E/A	-	Ein-/Ausgabe

ALU-Operation führt); er bewahrt das Resultat im AC auf und legt es, falls verlangt, im HS ab; und er holt sich den nächsten Befehl.

Für diejenigen Leser, welche die Vorgänge im Prozessor nachzuvollziehen möchten, soll Bild 13.7 näher erläutert werden. Es zeigt die Schaltung eines Einprozessorrechners, dessen RALU in ein KR-Netz dekomponiert ist, das zwei Kombinationsschaltungen (ALU, INC) und vier Register (BZ, BR, AC, DR) enthält. Der HS und - wie wir später sehen werden - auch der Steueroperator lassen sich in KR-Netze dekomponieren. Der ganze Rechner ist also ein KR-Operator. Der Hauptspeicher interessiert hier lediglich unter dem Aspekt der Kommunikation mit dem Prozessor. Sie erfolgt über einen Kommutator (K) [12.2] und einen Halbkommutator (HK; siehe Bild 12.4a). Diese Kommunikationsstruktur ist derjenigen von Bild 12.4b sehr ähnlich. Die Kommunikation mit der Außenwelt erfolgt über den HK. Ein- und Ausgabeeinheiten sind nicht eingezeichnet. Die durchgezogenen Pfeile stellen die Übergabewege der Befehle, der Operanden und der Adressen dar; aus der Sicht des Elektronikers stellen sie elektrische Leitungen (leitende Bahnen auf einem Chip) dar bzw. Leitungsbündel, falls die den Daten entsprechenden Bitketten parallel übertragen werden. Der Adressweg S3-HK ist ausnahmsweise gestrichelt gezeichnet, weil die Adresse die Rolle eines Steuersignals des Halbkommutators HK spielt, sie ist der "Schlüssel", der das "Schloss" des adressierten Speicherplatzes öffnet. Die Wege der Steuersignale zu den Weichen und zum Kommutator K sind nicht einzeln dargestellt, sondern werden gemeinsam durch den gestrichelten Pfeil vom Steueroperator zur RALU symbolisiert. Der Pfeil vom OC-Feld des BR zur ALU ist punktiert gezeichnet, um anzudeuten, dass die ALU durch Übergabe des Operationscodes über den punktierten Pfeil (und Umcodierung in das entsprechende Steuersignal) konditioniert werden *kann*, dass die ALU ihr Steuersignal aber auch vom Steueroperator erhalten kann (der entsprechende Signalweg ist nicht eingezeichnet).

Die Ausführung des im Befehlsregister (BR) befindlichen Befehls beginnt mit der Übergabe (Meldung) des Operationscodes (OC) an den Steueroperator, der daraufhin die Steuersignale in der erforderlichen zeitlichen Reihenfolge und mit den erforderlichen zeitlichen Abständen generiert. Die Signalfolge kann für jeden Befehl ein für allemal festgelegt werden, da bekannt ist, in welcher Reihenfolge die Steueroperationen auszuführen sind, wie viel Zeit die Übergangsprozesse in den Schaltungen beanspruchen und welche zeitlichen Abstände demzufolge zwischen den Steuersignalen einzuhalten sind, um zu gewährleisten, dass die Datenübergaben (Spannungsübergaben zwischen den elektronischen Schaltungen) fehlerfrei erfolgen. Der Steueroperator steuert also die einzelnen Befehlsausführungen autonom (ohne Einfluss von Meldungen). Durch die OC-Meldungen werden die Ausführungen der Befehle eines Programms gestartet, und durch ein Programm-Startsignal wird die Ausführung eines Programms gestartet. Die Startadresse des Programms muss vor oder mit dem Startsignal in den Befehlszähler (BZ) eingetragen werden. In Kap.13.5.5 werden wir uns überlegen, wie sich der Steueroperator schaltungsmäßig realisieren lässt.

1 Aktionsschritt	2 Trans- fer-Nr.	3 Daten	4 Datenweg
1 Befehl holen	1	Befehlsadresse	BZ - S3 - HK
	2	ADD-Befehl	HS - HK - K - BR
2 ALU konditionieren	3	OC der Addition	OC - ALU
3 Operanden an ALU	4	Adresse von c	A1 - S3 - HK
	5	c	HS - HK - K - DR - ALU
4 Resultat speichern	6	Adresse von r	A2 - Z3 - S3 - HK
	7	r	ALU - Z1 - K - HK - HS
5 Adresse des nächsten Befehls berechnen	8	Befehlsadresse	BZ - INC - S2 - BZ

13.8 Registertransfers beim Holen und Ausführen des Additionsbefehls des Programms von Bild 13.6. Bezeichnungen siehe Bild 13.7. In der Spalte “Daten” ist dasjenige Datum (Adressen, Operationscode, Operand) angegeben, das transferiert wird.

Eine weitere Unterstützung beim Nachvollziehen der Programmabarbeitung gibt die Tabelle von Bild 13.8. In ihr sind alle Datenübergaben in zeitlicher Reihenfolge aufgelistet, die während der Ausführung des Additionsbefehls aus dem Programm von Bild 13.6 stattfinden.

Die Spalte 1 enthält die Aktionsschritte [10]. In Spalte 2 sind die einzelnen Übergaben durchnummeriert. Eine Übergabe wird auch als **Transfer** bezeichnet. Wenn in jedem Arbeitstakt der RALU (in jedem Takt des Steueroperators) ein Transfer ausgeführt wird, gibt die Zahl in Spalte 2 die Taktnummer an. In Spalte 3 sind die Daten (Operanden bzw. Adressen) angegeben, die transferiert werden, und in Spalte 4 ihre Wege. Nach dem Laden des ADD-Befehls in das Befehlsregister steht im Feld A1 die Adresse von *c* und im Feld A2 die Adresse von *r*. Im AC steht das Resultat der vorangegangenen Subtraktion, also der Wert der Differenz *a-b*. Zur Illustration sollen die beiden Transfers des dritten Aktionsschrittes ausführlicher beschrieben werden. In diesem Schritt wird die ALU mit Operanden versorgt und das Resultat berechnet [10].

Transfer 4. Der Steueroperator stellt die Sammelweiche S3 so, dass die im Adressfeld A1 befindliche Adresse von *c* dem Halbkommutator HK übergeben wird. Dadurch öffnet sich das “Schloss” des Speicherplatzes mit der Adresse <c>.
Transfer 5. Der Steueroperator stellt HK und K auf Durchgang HS-DR und Z2 in Richtung ALU. Dadurch wird *c* vom HS nach DR transferiert und damit an den oberen Eingang der ALU gelegt. Unmittelbar danach (d.h. nach Ablauf des Übergangsprozesses in der ALU) erscheint am Ausgang der ALU das Resultat *r*, denn am unteren Eingang der ALU liegt die Differenz *a-b*. Weiterer Kommentare wird es kaum bedürfen, um die Abarbeitung des Programms von Bild 13.6 in allen Einzelheiten nachvollziehen zu können.

Eines der Ziele des Entwurfs von Prozessor und Maschinensprache, die gegen Ende des Kapitel 13.5.2 genannt worden waren, ist eine möglichst schnelle Befehlsausführung. Die Ausführungszeit einer Operation kann eventuell dadurch verkürzt werden, dass Datentransfers gleichzeitig ausgeführt werden. Voraussetzung dafür ist, dass die betreffenden Datenwege sich nicht berühren. Geht man unter diesem Gesichtspunkt die Transfers von Bild 13.8 noch einmal durch, erkennt man, dass Transfer 8 mit allen Transfers außer dem ersten gleichzeitig ausgeführt werden kann. Die Adresse des nächsten Befehl kann also berechnet werden, während der laufende Befehl ausgeführt wird. Weiterhin erkennt man, dass die Transfers 3, 4 und 5 in einem einzigen Takt ausgeführt werden können, wenn gewährleistet ist, dass die Steuersignale die Tore solange geöffnet halten, wie es für die fehlerfreie Übergabe der Daten (Spannungen) erforderlich ist. Der aufmerksame Leser wird erkennen, dass die drei Transfers gleichzeitig ausgeführt werden *müssen* oder dass zusätzliche Register erforderlich sind. Ganz anlog verhält es sich mit den Transfers 6 und 7. Die Anzahl der Takte lässt sich also auf 4 herabsetzen. Die Minimierung der Taktzahl pro Befehlsausführung ist eine wichtige Aufgabe der Entwicklungsingenieure.

Der nachfolgende END-Befehl stoppt den Steueroperator (genauer den Taktgenerator des Steueroperators). In bestimmten Fällen wird die Adresse des Befehls, der als nächster auszuführen ist, nicht durch Inkrementieren bestimmt, sondern durch das Programm selbst angegeben. Darauf wird im nächsten Abschnitt eingegangen.

Während der Abarbeitung eines Programms wiederholen sich ständig die fünf (sich evtl. überlappenden) Aktionsschritte der Befehlsausführung: Holen des aktuellen Befehls, Konditionieren der ALU, Versorgen der ALU mit den aktuellen Operanden, Abspeichern des Resultats und Berechnen der Adresse des nächsten Befehls. Diese Folge hatten wir *zentrale Steuerschleife* genannt.

Durch den detaillierten Nachvollzug der Operationsausführung erhält man eine genaue Vorstellung davon, was dieser kleine Kobold, Prozessor genannt, der dabei ist, die Welt zu verändern, tatsächlich macht. Es ist stets das gleiche stereotype Hin- und Herschieben von Bitketten zwischen denselben Registern. Unterwegs können die Ketten transformiert werden, und zwar stets von derselben Kombinationsschaltung, der ALU. Die Übergabe einer Bitkette zwischen zwei Registern wird häufig

Registertransfer genannt, unabhängig davon, ob der Übergabeweg über eine Kombinationsschaltung führt oder nicht.

Der Prozess, den der Leser vielleicht soeben nachvollzogen hat, läuft in der Welt viele Milliarden Male pro Sekunde ab und mischt sich in fast Alles ein, was wir Menschen tun und lassen. Er bringt die Vielfalt der Leistungsmöglichkeiten der technischen Informationsverarbeitung und er bringt "künstliche Intelligenz" hervor. Dieser Übergang von Quantität in Qualität erinnert an den Bau der Materie, an das ständige Herumkreisen der Elektronen auf den gleichen Bahnen um die Atomkerne, die ihrerseits alle aus den gleichen Bausteinen, aus Protonen und Neutronen bestehen. Und dieses stereotype Kreisen bringt die Vielfalt der Welt hervor.

Die Struktur der Materie und die Funktionsweise des Prozessors sind nur zwei Beispiele für das Phänomen, dass durch Wechselwirkung vieler (oft gleicher oder ähnlicher) Objekte (Prozesse) ein neues, kompliziertes Objekt (ein neuer, komplizierter Prozess) mit neuen Eigenschaften entsteht, eventuell eine Hierarchie immer komplizierterer Objekte. Zur Kennzeichnung der komplizierten Struktur derartiger Objekte hat sich das Wort *Komplexität* und für das Hervortreten neuer Eigenschaften das Wort *Emergenz* eingebürgert. Der Begriff der Komplexität wird uns in Kap.21 beschäftigen.

12 Der Leser beachte folgenden wichtigen und vielleicht überraschenden Umstand. Die Signale, die der Steueroperator des Prozessors in Bild 13.7 generiert, sind völlig andere Signale als diejenigen, die ein Steueroperator generiert, der einen realen, nach den Regeln der USB-Methode komponierten Kompositoperator (ein ON) steuert. Angenommen, das Operatorennetz enthält einen Subtrahierer und einen Addierer und es soll der Wert $(a\text{-}b)+c$ berechnet werden. Dann hat der Steueroperator Steuersignale zu generieren, durch welche die Werte von a und b dem Subtrahierer und dessen Ausgabewert und der Wert von c dem Addierer zugeführt werden. Diese Steuersignale haben wenig mit denjenigen zu tun, die der Steueroperator des Prozessors zu generieren hat, damit der Wert von $(a\text{-}b)+c$ berechnet wird (siehe dazu auch Kap.13.7 und die Diskussion zu Bild 19.4).

Das scheint zu bedeuten, dass sich mit Hilfe des Von-Neumann-Rechners das Komponieren von Kompositoperatoren nach der USB-Methode *nicht* von der Hardware auf die Software übertragen lässt. Unsere Idee, sprachliche Kompositoperatoren nach der USB-Methode zu komponieren, d.h. als Operationsvorschrift zu artikulieren, um sie dann vom Computer ausführen zu lassen, scheint zum Scheitern verurteilt zu sein, es sei denn, es gelingt, Operandenflussprogramme in die Maschinensprache zu übersetzen. Um das zu erreichen, müssen wir unsere Maschinensprache offensichtlich erweitern. Wir wollen uns überlegen, welche Erweiterung unbedingt erforderlich ist.

13.5.4 Sprungbefehl

Damit ein Operandenflussplan in eine Maschinensprache übersetzt werden kann, muss diese die Möglichkeit bieten, die Flussknoten des Operatorennetzes *imperativ*,

d.h. durch eine Folge von Befehlen auszudrücken. Insbesondere muss es eine Entsprechung zur Zweigeweiche geben, also einen Befehl, der die Verzweigung der Aktionsfolge steuert, m.a.W. der entscheidet, welcher von zwei möglichen Befehlen als nächster ausgeführt wird. Einem solchen Befehl sind wir bereits im *bedingten Sprung* der unbeschränkten Registermaschine begegnet (siehe Kap.8.4.3).

Die graphische Darstellung eines imperativen Programms, das einen Sprungbefehl enthält, muss offenbar ein *verzweigter* Aktionsfolgeplan sein, denn ein *linearer* Aktionsfolgeplan kann ohne zusätzliche Vereinbarungen keinen bedingten Sprung der Aktionsfolge darstellen. Bild 13.9 zeigt einen verzweigten Aktionsfolgeplan, der die Folge der Aktionen angibt, die bei der sequenziellen Berechnung der durch (8.1) definierten Funktion auszuführen sind (pot-Operator und sin-Operator sind nicht dekomponiert). Dabei ist eine in der Praxis gängige Darstellungform gemäß DIN (Deutsches Institut für Normung) gewählt, der sog. **Programmablaufplan** (**PAP**)[11]. Die Parallelogramme stellen die Dateneingabe bzw. Datenausgabe dar. Jedem rechteckigen Kästchen entspricht eine Aktion. Das rhombische Kästchen stellt eine Zweigeweiche im Aktionsfolgeplan dar. Es entspricht einem Prädikatoperator, der entscheidet, welcher Weg gewählt wird. Wenn das Prädikat [x≤0] erfüllt ist, wird der linke, mit t (von true) bezeichnete, andernfalls der rechte, mit f (von false) bezeichnete Weg eingeschlagen. Das rhombische Kästchen entspricht in Bild 8.1 der Zweigeweiche vor dem Sinusoperator und schließt den Steueroperator der Weiche ein.

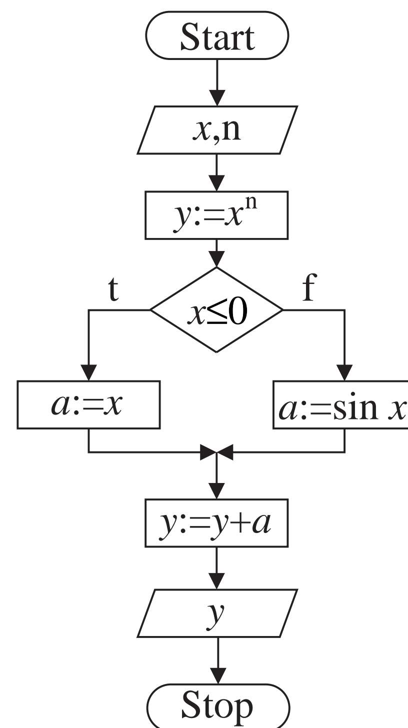


Bild 13.9 Aktionsfolgeplan (Programmablaufplan) für die Berechnung der durch (8.1) definierten Funktion. Der Rhombus stellt eine steuerbare Verzweigung der Aktionsfolge dar.

Zur Ergibtanweisung `y:=y+a` ist eine Bemerkung am Platze. Das Auftreten des Bezeichners `y` auf beiden Seiten der Anweisung könnte einen funktional denkenden Leser irritieren. Doch ist die Ergibtanweisung erlaubt und sinnvoll. Ihre interne Semantik ist das Überschreiben einer Speicherzelle. Um das zu erkennen, muss man sich vergegenwärtigen, dass in dem binär notierten Maschinenprogramm anstelle von `y` die Adresse der Speicherzelle

11 Siehe z.B. [Duden 89].

für den Wert von y steht. Das y auf der rechten Seite der Ergibtanweisung bezeichnet den Inhalt dieser Speicherzelle *vor* der Operationsausführung, das y auf der linken Seite den Inhalt *nach* der Operationsausführung.

Die graphische (zweidimensionale) Operationsvorschrift von Bild 13.9 soll nun in eine eindimensionale ("lineare") Befehlsfolge überführt werden. Wir sprechen von *Linearisierung*. Sie führt zwangsläufig dazu, dass bei der Abarbeitung unter bestimmten Bedingungen innerhalb der Befehlsfolge gesprungen werden muss. Der einfachste Befehl, der dies ermöglicht, heißt **bedingter Sprungbefehl**[12]. Mit seiner Hilfe lässt sich Bild 13.9 in eine Folge von Befehlen, also in einen imperativen Algorithmus überführen (linearisieren). Bild 13.10 gibt einen solchen Algorithmus an. Der dritte Befehl ist folgendermaßen zu lesen: "Springe, falls x nicht größer als 0 ist, nach Befehl 5". In Bild 13.9 entspricht diesem Befehl der Rhombus. Der Befehl setzt die Nummerierung der Befehle voraus, um das Sprungziel "adressieren" zu können.

```
1   y := x^n
2   a := x
3   SPNG x 5
4   a := sin(x)
5   y := y+a
```

Bild 13.10 Linearisierung von Bild 13.9

Wir wollen nun unsere Zwei-Adress-Maschine befähigen, Sprünge in einem Maschinenprogramm auszuführen. Dazu vereinbaren wir, dass in das Adressfeld A2 des Befehlsregisters die **Sprungadresse** eingetragen wird; das ist die Adresse desjenigen Befehls, der *angesprungen* wird, zu dem "verzweigt" werden soll oder, wie man auch sagt, dem die "*Steuerung übergeben*" werden soll (die Steuerung der RALU)[13]. In das Adressfeld A1 wird die Adresse derjenigen Variablen eingetragen, die im Weichenprädikat auftritt. Der Operationscode SPNG zusammen mit dem Bezeichner x legt nach dieser Vereinbarung das Prädikat [x≤0] fest. Wenn es erfüllt ist, soll zum Befehl mit den Nummer 5 gesprungen werden, andernfalls wird Befehl 4 ausgeführt. Hardwaremäßig wird das Springen durch die bisher nicht benutzte Verbindung von Z3 nach S2 in Bild 13.7 ermöglicht. Über sie kann der Inhalt von A2 in den Befehlszähler BZ eingetragen werden. Durch die Eintragung wird die automatische Befehlszählung durch Inkrementieren unterbrochen; nach dem Sprung wird sie fortgesetzt.

Unsere Vereinbarungen hinsichtlich des Sprungbefehls sind willkürlich. Es sind andere Vereinbarungen möglich. Sie müssen in der Sprachdefinition festgelegt sein. Es kann jedes Relationszeichen verwendet werden. Auch die Festlegung, ob bei

12 Es gibt auch komfortablere "Verzweigebefehle".

13 Offenbar liegt hier die Wurzel der Bezeichnung *Steuerfluss* als Synonym von *Aktionsfolge*. Mit der Bezeichnung "Steuerfluss" ist der "Fluss" der Übergabe der Steuerung von Befehl zu Befehl gemeint, m.a.W. die Folge der Befehlsausführungen, der *Aktionen*.

erfülltem oder bei nicht erfülltem Prädikat gesprungen werden soll, ist dem Sprachentwickler überlassen.

Steuerprädikate müssen von der ALU entschieden werden. Dazu muss sie durch den Operationscode (in unserem Beispiel die binär verschlüsselte Buchstabenfolge SPNG) auf die festgelegte Vergleichsoperation konditioniert werden, und ihr müssen die zu vergleichenden Größen (z.B. x und 0) zugeführt werden. Das resultierende Ausgabebit der ALU steuert die Weichen Z3 und S2 (die Steuerleitung ist in Bild 13.7 nicht eingezeichnet).

In Kap.13.7 werden wir uns davon überzeugen, dass unsere so erweiterte Maschinensprache *universell* ist. Sie erinnert sehr an die “Programmiersprache” der “Unbeschränkten Registermaschine” (URM) aus Kap.8.4.3. Offensichtlich ist die URM-Sprache in unmittelbarer Anlehnung an gängige Maschinensprachen definiert worden. Der Sprungbefehl der URM ist etwas komfortabler als unser Sprungbefehl, denn er stellt für die Programmierung der Sprungbedingung zwei Adressen zur Verfügung.

13.5.5 Matrixsteuerwerk

Nach dieser wichtigen Schaltungs- und Spracherweiterung wenden wir uns dem Entwurf des Steueroperators der RALU zu. Der erste Entwurfsschritt (wir führen ihn nicht im Detail aus) besteht darin, die Liste der Aktionsschritte der Addition von Bild 13.8 auf sämtliche Befehle zu erweitern. Dann ist für jeden Befehl bekannt, welche Datenwege in welcher Reihenfolge bei den verschiedenen Befehlsausführungen benutzt werden. Damit ist auch die Reihenfolge bekannt, in der die Tore geöffnet werden müssen. Genau dieses Wissen ist erforderlich, um eine Schaltung zu entwerfen, welche die notwendigen Steuersignalfolgen generiert.

In Kap.12.3.4 sind Entwurfsmethoden und mikroelektronische Realisierungen von Steueroperatoren mittels ROM besprochen worden. Ein Steueroperator dieser Art wurde dort als *Matrixsteuerwerk* und der ROM als *Steuermatrix* bezeichnet. Es wurde zwischen *rückgekoppelter* Steuerung, die aufgrund von Meldungen aus dem Arbeitsoperator (z.B. einer Waschmaschine) erfolgt, und *rückkopplungsfreier* Steuerung unterschieden, die keine Meldungen berücksichtigt. Letztere ist möglich, wenn bei der Steuerung keine Alternativen zu entscheiden sind und wenn die Dauer der Bausteinoperationen von vornherein festliegt, wenn also keine Ende-Meldungen der Bausteinoperatoren erforderlich sind. Das trifft für den Prozessor zu, denn die Zeitdauer, die ein Registertransfer benötigt, ist bekannt, und damit sind die Zeitpunkte bekannt, zu denen die einzelnen Schritte einer Aktion (einer Befehlsausführung) beginnen können. Die “Zeitmessung” (richtiger die Taktzählung) beginnt mit dem Start der Befehlsausführung.

Ein Steueroperator, der keine Meldungen zu berücksichtigen braucht, kann nach dem Funktionsgenerator-Prinzip aufgebaut werden (siehe Kap.12.3.3). Dabei werden (in Analogie zur Waschmaschinensteuerung) in die Adressiermatrix (vgl. Bild 12.2) die Zeitpunkte (Taktnummern) eingetragen (eingeprägt), in jede Zeile der

Matrix eine Taktnummer. In die Zeilen der Speichermatrix wird das jeweilige Steuerwort (Steuersignaltupel) eingetragen (eingeprägt). Das Steuerwort enthält für jeden wählbaren Übergabeweg ein Bit, z.B. für die Steuerung von S3 in Bild 13.7 drei Bit.

Hat der Prozessor aber einen Sprungbefehl mit Sprungbedingung zu interpretieren, hängt die aktuelle Aktion von einer Meldung über die Erfüllung der Sprungbedingung ab (entsprechend der Meldungen des Temperaturfühlers der Waschmaschine). Die Adressmatrix (und die zugrundeliegende Entscheidungstabelle) muss also neben der Taktspalte eine oder mehrere Spalten für Meldungen enthalten. Eine Zeile, die im Adressteil eine Meldung enthält, muss im Speicherteil die "Sprungadresse", d.h. die Taktnummer enthalten, mit der fortzufahren ist. Diese wird über eine Rückkopplungsschleife mit Taktverzögerer auf den Eingang der Adressmatrix zurückgegeben (in Bild 12.2 gestrichelt angedeutet).

Diese Methode kann in *jeder* Spalte angewendet werden, sodass kein Befehlszähler (Taktzähler), sondern nur ein *Taktgenerator* erforderlich ist, d.h. ein Impulsgenerator, der das Weiterspringen von Zeile zu Zeile auslöst. Damit ist grob skizziert, wie ein Matrixsteuerwerk arbeitet.

Damit der Prozessor das Programm von Bild 13.6 abarbeiten kann, muss er für jeden Befehl über eine spezielle Steuermatrix verfügen. Die Steuermatrizen (die ROMs) lassen sich zu einer Matrix vereinigen, indem sie z.B. "untereinander" angeordnet werden. Um die Ausführung eines Befehls zu starten, muss die Anfangszeile der betreffenden Matrix angewählt werden, indem der *OC-Entschlüsseler* (der Decodierer des Operationscodes) den Operationscode in die Nummer der Matrixzeile umcodiert, an der die betreffende Operation beginnt. Das *Operationsrepertoire* bleibt auf die Operationen der ALU beschränkt. Eine Multiplikation muss bereits vom Nutzer programmiert werden. Wir wollen uns überlegen, wie dem abgeholfen werden kann.

13.5.6 Operatorenkomponierung mittels Firmware

Die Verwirklichung einer im Grunde einfachen Idee befreit den Nutzer von der Aufgabe, das Multiplizieren und andere häufig auszuführende Operationen selber programmieren zu müssen. Die Idee überträgt diese Aufgabe dem Computerbauer. Sie wird durch die vorangehenden Überlegungen nahegelegt, sodass mancher Leser sich vielleicht schon gefragt hat, warum man die ALU nicht durch einen (geeignet adaptierten) Taschenrechner ersetzt. Eben darin besteht die Idee. Das Operationsrepertoire wird auf die wichtigsten Operationen wie Multiplizieren, Dividieren, Wurzelziehen u.ä.m. erweitert. Für jede Operation wird eine Steuermatrix entworfen und implementiert. Eine bestimmte Operation wird durch ihren Operationscode gestartet (ensprechend dem Drücken der Funktionstaste des Taschenrechners).

Es ergibt sich eine zweistufige Steuerhierarchie. Der übergeordnete Steueroperator organisiert die Abarbeitung von Maschinenprogrammen und die Kommunikation mit dem HS, jedoch nicht die Abarbeitung der einzelnen Befehle. Diese überträgt er

dem untergeordneten Steueroperator (“Taschenrechner-sop”), indem er ihm den aktuellen Operationscode und die Operanden übergibt. Gegebenenfalls können Operandenwerte unmittelbar über den Akkumulator an die nächste Operationsausführung übergeben werden. Die Erweiterung des Operationsrepertoires der Maschinensprache (Prozessorsprache) verlangt - ähnlich wie die Erweiterung um den Sprungbefehl - eine Hardwareerweiterung um einige Register und Weichen. Zur Bereitstellung der Eingabeoperanden reicht evtl. ein einziges Datenregister (DR in Bild 13.7) nicht aus. Beim Entwurf des untergeordneten Steueroperators ist vom KR-Netz des jeweiligen Kompositoperators auszugehen, im Falle der Multiplikation z.B. von der in Bild 13.3 dargestellten Schaltung.

Die entscheidende Erweiterung des “Taschenrechnerkonzeptes” betrifft das Matrixsteuerwerk, also die Firmware. Um das zu unterstreichen, unterscheiden wir hinsichtlich der Komponierung realer Operatoren zwischen Komponierung durch “*echte Hardware*” und Komponierung durch *Firmware*. Bild 19.1 zeigt die Prinzipschaltung eines Prozessors, der über ein Matrixsteuerwerk zur Operatorenkomponierung verfügt.

13.5.7 Mikroprozessor

Die Integration des Matrixsteuerwerks und der vom ihm gesteuerten Schaltung, der RALU, auf einem Chip liefert ein mikroelektronisches Bauelement von grob gesagt 1 cm^2 Größe, den **Mikroprozessor**. Der HS eines PC besteht i.d.R. aus

Bezeichnung	Jahr	Registerlänge [Bit] Datenreg./Adr.-Reg.	Taktfrequenz [Mhz]
8080	1976	8/16	bis 4
8086/186	1981	16/20	bis 10
80286	1984	16/24	bis 25
80386	1986	32/32	bis 40
80486	1989	32/32	bis 66
Pentium	1993	64/32	bis 233
Pentium II	1997	64/32	bis 450
Pentium III	1999	64/32	600
verschiedene Anbieter	2000		1000

Bild 13.11 Mikroprozessoren der Firma Intel.

mehreren Chips und nimmt eine größere Fläche auf der *Steckkarte* (so heißen die Einschübe eines PC) ein als der Prozessor. Ein wichtiger technischer Parameter eines Mikroprozessors ist neben der Größe und dem Energieverbrauch seine Taktfrequenz.

Sie bestimmt entscheidend die Rechengeschwindigkeit. Der Wettbewerb zwischen den Herstellern bewirkt gemeinsam mit den Wünschen der Nutzer eine ständige Erhöhung der Taktfrequenz. In Bild 13.11[14] sind für mehrer Mikroprozessoren der Firma Intel die für den Nutzer wichtigsten technische Daten und das Erscheinungsjahr aufgeführt.

Die Länge der Adressregister (der Adress-Felder im Befehlsregister) bestimmt die Größe des adressierbaren Speichers. Bei einer Länge von 32 Bit ist die Anzahl der adressierbaren Speicherplätze gleich 2^{32}, sie liegt nach (9.4a) also zwischen 10^9 und 10^{10}. Die Taktfrequenz ist durch die Zeitdauer begrenzt, die ein Registertransfer beansprucht. Diese hängt von den Schaltzeiten der Transistoren ab, also davon, wie schnell sich die erforderlichen Schaltspannungen aufbauen. Damit hängt die Taktfrequenz letzten Endes von der Zeitdauer ab, welche die Ladungsträger brauchen, um die halbleitenden Schichten der Tansistoren zu durchqueren. Letztendlich geht es also darum, Halbleiter mit möglichst hoher Beweglichkeit der Ladungsträger zu verwenden und möglichst dünne p- und n-leitende Schichten zu produzieren. Demzufolge wird weltweit nach neuen Halbleitermaterialien gesucht und die Dotierungstechnologie wird ständig vervollkommnet.

Die geringen Ausmaße von Mikroprozessoren und Arbeitsspeichern, die hohe Taktfrequenz und die Möglichkeit ihrer Massenproduktion (vgl.Kap.12.2) sind entscheidende Voraussetzungen dafür, dass der Computer beginnt, in sämtliche Bereiche menschlicher Tätigkeit einzudringen. Der Hardwareentwurf kann offensichtlich als "gelungen" beurteilt werden.

Nicht so günstig sieht es mit dem Softwareentwurf aus, treffender: mit dem Entwurf und der Implementierung von Programmiersprachen. Der Anwender wünscht sich eine einfache und dennoch ausdrucksstarke Sprache. Aber welche Freiheiten haben die Informatiker beim Entwerfen von Sprachen? Mit dem detaillierten Entwurf des Prozessors ist bereits diejenige Sprache festgelegt, die der Prozessor direkt, d.h. ohne vorherige Anpassung an die Prozessorstruktur (ohne Übersetzung) verarbeiten kann. Diese Sprache hatten wir in Kap.13.5.2 *Maschinensprache*[15] genannt.

Die Maschinensprache unserer Zwei-Adress-Maschine ist im wesentlichen durch den Aufbau des Befehlsregisters in Bild 13.7 und dessen erlaubte Inhalte definiert. Ihre Verwendung zur Beschreibung komplizierter Funktionen kann recht aufwendig werden. Die Suche nach besseren Programmiersprachen ist mit dem Entwurf des Prozessors "vorprogrammiert". Sie dauert bis heute an. In Kap.18 wird ein Überblick über die Entwicklung gegeben. Eine sehr lebendige Darstellung der Softwareent-

---

14 Die Tabelle ist mit einigen Änderungen und Ergänzungen [Märtin 94] entnommen.

15 Genau genommen hatten wir sie *Prozessorsprache* genannt. Da aber in diesem Kapitel nur vom Ein-Prozessor-Rechner die Rede ist, braucht zwischen Prozessor- und Maschinensprache nicht unterschieden zu werden.

wicklung, die Hand in Hand mit der Hardwareentwicklung der Mikroprozessoren vom Intel 8080 bis zum Pentium verlief, findet der Leser in dem Buch “Der Weg nach vorn” von BILL GATES [Gates 97], einem der Hauptakteure auf dem Gebiet der Massenproduktion und Vermarktung von PC-Software.

13.6 Von-Neumann-Rechner

Durch Ankopplung eines Arbeitsspeichers an einen Prozessor entsteht ein *Einprozessorrechner*. Wenn Programme und Daten gleichberechtigt in einem einzigen Speicher abgespeichert werden, nennt man den Rechner **Von-Neumann-Rechner**, weil JOHN VON NEUMANN diesem Strukturprinzip zum Durchbruch verholfen hat. Der Arbeitsspeicher des Von-Neumann-Rechners wird auch *Hauptspeicher* (abgekürzt HS) genannt. Wenn für Programme und Daten getrennte Speicher existieren, wird von **Havard-Computer** gesprochen. Beide Rechnertypen sind Prozessorrechner. Heutzutage wird die getrennte Speicherung praktisch nur noch hinsichtlich des Cache-Speichers (siehe Kap.19.2.1 und Bild 19.1) angewendet.

Die Verbindung vom Prozessor zum Hauptspeicher und wieder zurück zum Prozessor kann als Rückkopplungsschleife aufgefasst werden, in welcher Daten “kreisen”. Die sich ergebende globale Struktur ist diejenige des Automaten von Bild 8.5. Dabei entspricht dem inneren Zustand z des Automaten der gesamte Inhalt des HS. In konsequenter Weiterführung dieser Analogie wird dem Prozessor bei jeder Befehlsausführung (in jedem “Automatentakt”, d.h. jedes mal, wenn das Tor vom Speicher zum Prozessor geöffnet wird) der gesamte Speicherinhalt zur Verfügung gestellt, aus dem sich der Prozessor den aktuellen Befehl und die aktuellen Daten heraussucht. Sodann führt der Prozessor den Befehl aus und generiert den neuen Zustand z', indem er den Inhalt des durch die Resultatadresse gekennzeichneten Speicherplatzes mit dem Ergebnis der Befehlsausführung überschreibt.

Die Vorstellung, dass der Prozessor in jedem Takt mit dem gesamten Speicherinhalt hantiert, aber nur einen kleinen Bruchteil benutzt bzw. verändert, ist durchaus erlaubt. Sie ist verbunden mit der Vorstellung, dass der nichtveränderte Teil gewissermaßen durch den Prozessor “durchgezogen” wird. In Kap.8.4.5 hatten wir dafür den Begriff des *Durchschleppens* eingeführt.

Die Organisation des Befehls- und Datenflusses im Von-Neumann-Rechner führt zu einem ernsthaften Problem, denn es gibt nur eine einzige Verbindung zwischen Prozessor und Speicher, die Verbindung K-HK in Bild 13.7. Über diese eine Verbindung wickelt sich die gesamte Kommunikation zwischen Prozessor und HS ab. Über sie werden sämtliche Befehle und Daten transportiert und zwar in beiden Richtungen und selbstverständlich streng sequenziell.

Dieses Leitungsstück zusammen dem Kommutator und dem Halbkommutator stellt einen sehr einfachen Datenbus dar (vgl. Kap.12.3.2). Sein Durchlassvermögen kann die Arbeitsgeschwindigkeit des Von-Neumann-Rechners evtl. erheblich herab-

setzen. Das Leitungsstück wirkt wie ein Straßenstück, das nur einspurig befahrbar ist und vor dem sich bei stärkerem Verkehr die Fahrzeuge in beiden Richtungen stauen. Die Verbindung zwischen Hauptspeicher und Prozessor wird sehr anschaulich **von-neumannscher Flaschenhals** genannt. Seine Erweiterung oder noch besser Eliminierung beschäftigt die Computerbauer seit Jahrzehnten. Der Entwurf und die Realisierung des Von-Neumann-Rechners stellt das Ergebnis der ersten großen Etappe des Weges "vom Rechnen zum Erkennen" dar.

13.7 Netzparadigma und imperatives Paradigma

In diesem Kapitel soll der begriffliche Hintergrund der Unterscheidung zwischen *Datenfluss*programmierung und *Aktionsfolge*programmierung aufgezeigt werden. Außerdem wird nachgewiesen, dass der Von-Neumann-Rechner alle rekursiven Funktionen und nur diese berechnen kann, womit die Aussage (8) aus Kap.13.3 bestätigt wird, dass die von einem Prozessorcomputer berechenbaren Funktionen rekursive Funktionen sind.

Der Schritt von der Hardware in die Software ist mit einem für die Informatik charakteristischen Wechsel des Denk- und Modellierungsparadigmas verknüpft, dem Übergang aus der Welt der *Operatorennetze* in die Welt der *Algorithmen*, speziell der *imperativen* Algorithmen. Der Paradigmenwechsel ist letzten Endes durch die Zentralisierung der Speicherung verursacht und führt zu einem Verlust an Übersichtlichkeit und Durchschaubarkeit; während die *Sichtbarkeit* der Operandenflüsse in der Beschreibung von Prozessen durch Operatorennetze unmittelbar gegeben ist, geht sie in der Beschreibung durch Algorithmen weitgehend verloren.

Wir haben es mit zwei fundamentalen **Modellierungsparadigmen** zu tun, die wir das **Netzparadigma** und das **imperative Paradigma** nennen. Verallgemeinert könnte man auch von **raum-zeitlichem** und **rein zeitlichem Paradigma** sprechen. Damit knüpfen wir an Kap.8.2.3 [8.11] an. Dort hatten wir bei der Behandlung verschiedener Methoden des kausaldiskreten Modellierens zwischen *raum-zeitlicher* und *rein zeitlicher* Prozessbeschreibung unterschieden. Das heißt nicht, dass für ein Modell in Form eines imperativen Algorithmus oder eines Aktionsfolgeprogramms der Raum nicht existiert und dass es keinerlei räumliche Beziehungen gibt. Doch hat der physische Raum, in welchem Daten im Computer übergeben und gespeichert werden, nichts mit dem Raum zu tun, in dem das *modellierte* Original existiert. Demgegenüber kann der gedachte Raum, in welchem ein Operatorennetz existiert und funktioniert, sehr viel mit dem Raum des Originals zu tun haben. Das "Verschwinden" des "Originalraumes" (des räumlichen Diskursbereiches) aus dem als imperatives Programm formulierten sprachlichen Modells bedeutet aber nicht sein völliges Verschwinden aus dem Modell. Implizit ist der Originalraum auch in einem imperativen Modell enthalten und zwar in den Bezeichnern der Operatoren und Operanden. Durch sie kann der Nutzer (Programmierer) den Raum in ein imperatives Programm

hineininterpretieren. Denn die externe Semantik der Bezeichner kann durchaus räumliche Aspekte besitzen.

Die beiden Paradigmen des Denkens (Modellierens) können alternativ angewendet werden, wenn raumzeitliche Prozesse sprachlich modelliert werden sollen, sei es zum Zwecke ihrer theoretischen Untersuchung, sei es zum Zwecke ihrer Steuerung. Der Kern dieser Wahlfreiheit liegt darin begründet, dass man Prozesse, die mit irgendwelchen Flüssen (Stoff-, Energie-, Informationsflüssen) verbunden sind, aus zwei unterschiedlichen Sichten betrachten kann, entweder aus der Sicht eines mit dem Fluss sich *mitbewegenden* Punktes oder aus der Sicht eines im Raum *fixierten* Punktes ("Standpunktes").

In der Welt der Operatorennetze lässt sich die Menge der verkoppelten Operatoren als Menge kooperierender Akteure auffassen, die sich gegenseitig Daten zur weiteren Verarbeitung zuspielen. Die Akteure denken wir uns ortsfest, die Daten fließend. Für einen Beobachter lassen sich die Daten auf dem Wege ihrer Bearbeitung ohne große Schwierigkeiten verfolgen, vorausgesetzt, das Netz hat eine übersichtliche Struktur. Operationsvorschriften (Programme) für Operatorennetze legen den Datenfluss durch das Netz explizit fest, es sind *Datenflussprogramme*. Der Programmierer eines Datenflussprogramms schwimmt gewissermaßen mit dem Fluss mit. 14

In der Welt der imperativen Algorithmen gibt es nur einen einzigen Akteur, beispielsweise man selber, wenn man etwa die Aktionsfolge eines Kochrezeptes oder die Schritte eines Rechenverfahrens, z.B. des schriftlichen Multiplizierens, der Reihe nach ausführt. Die Operandenwege treten in der Vorschrift nicht explizit auf. Der Akteur muss sich die Operanden (Zutaten), die er gerade benötigt, selber beschaffen. Wenn der Akteur ein Prozessor ist, muss ihm per Programm mitgeteilt werden, wo er die Operanden der aktuellen Aktion findet und wo er das Resultat abspeichern soll. Dafür waren in der Maschinensprache (Kap.13.5.2) die Transportoperationen TVS und TNS eingeführt worden.

Die Rolle des Prozessors eines Von-Neumann-Rechners ist die eines zentralen Akteurs. Sie erzwingt eine Art der Programmierung, die sich von der Beschreibung des Operandenflusses durch ein Operatorennetz wesentlich unterscheidet. Ein Programm für den Prozessor legt keinen Datenfluss, sondern eine Aktionsfolge, d.h. eine Folge von Operationen mit explizit angegebenen Operanden fest. Maschinenprogramme sind *Aktionsfolgeprogramme*. Der "Standpunkt" des Programmierers eines Aktionsfolgeprogramms ist fixiert. Er sieht (mit den Augen des Prozessors) die Daten der Reihe nach "durch sich hindurchfließen".

Betrachtet man daraufhin noch einmal Bild 13.7, erkennt man Folgendes. In das Befehlsregister (BR) werden der Reihe nach die Befehle eines Aktionsfolgeprogramms[16] eingetragen. Nach jedem Befehlseintrag generiert der Steueroperator der

16 Es sei an das Synonym Steuerfluss für Aktionsfolge und Steuerflussprogrammierung für Aktionsfolgeprogrammierung erinnert.

RALU eine Steuerimpulsfolge, die den Datenfluss durch die RALU steuert. Das Befehlsregister bildet die Schnittstelle zwischen imperativem Algorithmus (Maschinenprogramm) und Operatorennetz (RALU und Hauptspeicher). Es sei noch einmal folgender Sachverhalt unterstrichen. Die Maschinensprachen gängiger Computer sind imperative Sprachen. Programme, die mit ihrer Hilfe geschrieben werden können, sind imperative Programme. Der Grund hierfür ist die zentrale Speicherung; die Folge ist das Format der Hardware-Software-Schnittstelle, also der einheitliche Aufbau des Befehlsregisters hardwareseitig und der Befehle softwareseitig mit festen Feldern für den Operationscode und die Adressen.

Im Hinblick auf die beiden Arten des Programmierens werden die genannten *Modellier*paradigmen, das Netzparadigma und das imperative Paradigma, zu *Programmier*paradigmen. Doch besitzen sie, wie gesagt, viel allgemeinere Bedeutung. Das sei an zwei Beispielen aus ganz anderen Bereichen veranschaulicht. Damit soll gleichzeitig der wesentliche Kern und die allgemeine (abstrakte) Bedeutung der Unterscheidung zwischen Operatorennetz und imperativem Algorithmus bzw. zwischen Datenfluss und Aktionsfolge verdeutlicht werden.

15 Das erste Beispiel entnehmen wir dem Straßenverkehr. Wir vergleichen die Sicht eines Autofahrers auf den Verkehr mit derjenigen eines Verkehrspolizisten. Der Autofahrer bewegt sich im Fluss des Verkehrs von Ort zu Ort. Er beobachtet ständig den Fluss und ordnet sich so ein, dass er durch seine Bewegung im Straßen*netz* sein Ziel erreicht. Dabei wird er den zu erwartenden Verkehrsfluss bis hin zum Ziel berücksichtigen. Seine Sicht ist *netz*- bzw. *datenflussorientiert*, wenn man die Autos als Daten bezeichnet. Die netzorientierte Sicht wird noch deutlicher, wenn man an einen Benutzer eines öffentlichen Verkehrs*netzes* denkt.

Demgegenüber ist der Verkehrspolizist auf einer bestimmten Kreuzung fest postiert und hat zu entscheiden, welches Auto im gegenwärtigen Zeitpunkt die Kreuzung passieren soll (welche Aktion ausgeführt werden soll). Die weiteren Wege der Autos (die Ziele ihrer Fahrer) interessieren ihn nicht. Seine Sicht mit Blick auf die Kreuzung (nicht auf das Straßennetz) ist *aktionsfolgeorientiert*. Er selber ist der zentrale Akteur. Das Beispiel hinkt insofern, als der Polizist den Auto*fluss* im benachbarten Straßen*netz* im Auge haben muss und der Autofahrer hinsichtlich seines Wagens eine *Folge* von Steuer*aktionen* ausführt.

16 Das zweite Beispiel entnehmen wir der Physik. Es betrifft die mathematische Beschreibung von Strömungsfeldern durch hydrodynamische Gleichungen. Dabei kann man sich als theoretischer Physiker (als mathematischer Modellierer von Strömungen) entweder sozusagen in ein Flüssigkeitsteilchen als Vehikel hineinsetzen, mit ihm mitschwimmen und seine Bahn und Geschwindigkeit "*datenflussorientiert*" beobachten bzw. berechnen (LAGRANGE'sche Herangehensweise), oder man kann an einem festen Punkt Aufstellung nehmen und die Strömungsparameter an diesem Punkt "*aktionsfolgeorientiert*" beobachten bzw. berechnen (EULER'sche Herangehensweise).

Beide Herangehensweisen, beide Sichten müssen letztendlich zu den gleichen Resultaten führen, im Strömungsbeispiel zu den gleichen Modellaussagen, im Verkehrsbeispiel zu dem gleichen angestrebten Endzustand, in dem alle Autofahrer ihre Ziele erreicht haben. Auf unser Problem übertragen heißt dies, dass sich ein und dieselbe Funktion sowohl datenfluss- als auch aktionsfolgeorientiert beschreiben und berechnen lässt. Die Folge ist die Herausbildung von zwei *Programmierparadigmen* oder *Programmierwelten*, wie zuweilen gesagt wird.

Mit den Konsequenzen der Spaltung der Programmierungstechnik in zwei vom Ansatz her unterschiedliche Bereiche werden wir konfrontiert, wenn wir versuchen, die Universalität des Von-Neumann-Rechners nachzuweisen. Zu diesem Zwecke ist zu zeigen, dass sich jedes Datenflussprogramm in ein Aktionsfolgeprogramm überführen lässt, denn jede rekursive Funktion ist eine USB-Funktion und lässt sich durch einen Datenflussplan beschreiben. Der Von-Neumann-Rechner kann also nur universell sein, d.h. jede rekursive Funktion berechnen, wenn sich jeder Datenflussplan in einen Aktionsfolgeplan überführen lässt.

Um den Beweis zu erbringen, kann man sich auf Kap.8.4.3 berufen. Unter der Annahme, dass sich der Speicher beliebig erweitern lässt, stellt der Von-Neumann-Rechner eine unbegrenzte Registermaschine (URM) dar, wobei die Register der URM den Registern der RALU und den adressierbaren Speicherplätzen des HS entsprechen. Vergleicht man die Befehle der Registermaschine mit denen der in Kap.13.5. definierten Maschinensprache, stellt man fest, dass das Befehlsrepertoire der Maschinensprache - es wird **Befehlssatz** genannt - den Befehlssatz der Registermaschine in sich einschließt. Das ist nicht verwunderlich, denn die fiktive Registermaschine ist im Grunde zu dem Zweck erdacht worden, die Universalität des Von-Neumann-Rechners nachzuweisen.

Damit könnten wir uns zufrieden geben, vorausgesetzt, wir vertrauen auf die Schlussfolgerung der Algorithmentheoretiker, dass die Registermaschine genau alle rekursiven Funktionen berechnen kann. Der Beweis dafür ist in Kap.8.4.3 nicht geführt worden. Wir wollen ihn auf unsere Weise nachholen.

Dazu genügt es zu zeigen, dass sich mit den Mitteln der Maschinensprache die Komponierungsmittel der USB-Methode, also die Datenübergabe von Aktion zu Aktion (an die Stelle der Bausteinoperationen treten jetzt Aktionen), die Gabel, die Vereinung und die Zweigeweiche darstellen (programmieren) lassen. In Kap.8.1 [8.1] hatten wir uns davon überzeugt, dass auf die Sammelweiche ohne Verlust der Universalität verzichtet werden kann, vorausgesetzt, es finden keine parallelen Prozesse in dem Netz statt, sodass keine Sammelweiche als Sequenzierer fungieren muss. Wir betrachten die Komponierungsmittel der Reihe nach.

1. Eine **Datenübergabe** zwischen zwei Aktionen[17], sagen wir eine Übergabe von Aktion A an Aktion B, lässt sich dadurch programmieren, dass die Resultatadresse

---

17 Mit anderen Worten ein Pfeil in einem PAP; vgl. Bild 13.9.

des Befehls zu Aktion A in den Befehl zu Aktion B als Operandenadresse eingetragen wird. Wenn zwischen den beiden Befehlen andere Befehle liegen, evtl. ein längeres Programmstück, muss gesichert sein, dass der Prozessor den Speicherplatz, in den er das übergebene Datum eingetragen hat, bis zu dessen Verwendung nicht überschreibt. Für die Sicherung des Speicherplatzinhaltes hat der Programmierer Sorge zu tragen. Hier haben wir es mit einem sehr einfachen Fall einer - zuweilen recht unangenehmen - Begleiterscheinung der Aktionsfolgeprogrammierung zu tun, die in ihrer Verallgemeinerung als **Seiteneffekt** (Effekte "an der Seite" oder "vonseiten" anderer Programmteile) bezeichnet wird.

2. Die Programmierung von **Vereinungen** ist durch das Befehlsformat gesichert, da es die Angabe zweistelliger Operationen samt ihrer beiden Operanden ermöglicht. Bei der Abarbeitung werden die beiden Operanden zu einem Operandenpaar *vereinigt*, gerade so, wie die Operanden (Summanden) des Additionsoperators in Bild 8.1 in dem Flussknoten vor dem Additionsoperator vereinigt werden.

3. Gabeln lassen sich dadurch programmieren, dass zwei Befehle ein und dieselbe Operandenadresse enthalten. In Analogie zur einfachen Datenübergabe muss garantiert sein, dass die Inhalte der beteiligten Speicherplätze nicht vorzeitig durch andere Befehlsausführungen überschrieben werden.

4. Der **Zweigeweiche** scheint der bedingte Sprungbefehl zu entsprechen. Genau genommen ist das jedoch nicht der Fall, denn der Sprungbefehl verzweigt keinen *Daten*fluss, sondern einen *Steuer*fluss (eine Aktionsfolge), also die Übergabe der Steuerung an den nächsten Befehl. Dennoch lässt sich eine Zweigeweiche mit Hilfe eines bedingten Sprungbefehls beschreiben. Dabei obliegt es dem Programmierer zu sichern, dass der Sprungbefehl die richtige (verlangte) *Daten*flussverzweigung bewerkstelligt, d.h. dass in den Befehlen, zu denen der Sprungbefehl verzweigt, die richtigen Operandenadressen eingetragen sind. Dass es dabei leicht zu Programmierfehlern kommen kann, insbesondere bei stark verzweigten Programmen, ist verständlich.

Damit haben wir uns überzeugt, dass sich sämtliche Komponierungsmittel der USB-Methode und damit auch der Datenflussprogrammierung in Aktionsfolgeprogrammen darstellen lassen. Die Überlegungen zeigen gleichzeitig, dass auch umgekehrt die Komponierungsmittel der Aktionsfolgeprogrammierung (also die Wiederholung von Operanden- bzw. Resultatadressen und der Sprungbefehl) in Datenflussprogrammen dargestellt werden können. Damit kommen wir zu folgendem Ergebnis: *Aktionsfolgepläne lassen sich in Datenflusspläne überführen und umgekehrt*.[18] Aus der wechselseitigen Überführbarkeit ergibt sich der

18 Bedeutend kürzer hätte der Nachweis mit Hilfe der *Durchschleppmethode* erbracht werden können.

Satz: Die Klasse der USB-Funktionen, d.h. derjenigen Funktionen, die sich nach Datenflussprogrammen berechnen lassen, ist mit der Klasse der imperativ berechenbaren Funktionen, d.h. derjenigen Funktionen, die sich nach Aktionsfolgeprogrammen berechnen lassen, identisch. Aktionsfolgeprogramme (Aktionsfolgepläne) berechnen alo rekursive Funktionen. 17

Da sich *jedes* Maschinenprogramm (jedes Aktionsfolgeprogramm) des Von-Neumann-Rechners in ein Datenflussprogramm überführen lässt und demzufolge eine USB-Funktion berechnet, ist jede mittels Von-Neumann-Rechner berechnete Funktion eine rekursive Funktionen. Da sich andrerseits jedes Datenflussprogramm in ein Aktionsfolgeprogramm und jedes Aktionsfolgeprogramm in ein Maschinenprogramm des Von-Neumann-Rechners überführen lässt, kann für jede rekursive Funktion ein Maschinenprogramm des Von-Neumann-Rechners geschrieben werden. Der Von-Neumann-Rechner kann also *jede* rekursive Funktion berechnen kann. Damit ergibt sich die Schlussfolgerung, dass die Klasse der durch den Von-Neumann-Rechner berechenbaren Funktionen mit der Klasse der rekursiven Funktionen identisch ist.

Diese Aussage hatten wir in Kap.13.3 unmittelbar aus dem *Berechenbarkeits-Äquivalenzsatz* [5] hergeleitet (vgl. Aussage (8) am Ende von Kap.13.). Die dortige Herleitung basierte auf dem Umstand, dass der Von-Neumann-Rechner ein KR-Netz darstellt oder in ein solches überführbar ist. Die jetzige Herleitung basiert auf dem Umstand, dass alle Komponierungsmittel der USB-Methode mit Hilfe der Maschinensprache beschreibbar sind. Da dies auch für die Sprache der unbeschränkten Registermaschine URM gilt, wie man unschwer erkennt, ist damit die Richtigkeit der Behauptung aus Kap.8.4.3 nachgewiesen, dass die Klasse der URM-berechenbaren Funktionen mit der Klasse der rekursiven Funktionen identisch ist. Außerdem folgt, dass *jedes Aktionsfolgeprogramm* ***wohlstrukturierbar*** ist[19], denn die Überführung eines Datenflussprogramms in ein Aktionsfolgeprogramm zerstört die Wohlstruktur nicht. Dass sich Operandenflusspläne wohlstrukturieren lassen, wissen wir aus Kap.8.4.5. [8.28]

Die oben getroffene Aussage, dass für jede rekursive Funktion ein Maschinenprogramm für ihre Berechnung geschrieben werden kann, ist gleichbedeutend mit folgendem

Satz: Der Maschinenkalkül eines Prozessorcomputers ist universell.

Der Begriff des Maschinenkalküls war in Kap.8.6 eingeführt worden. Der **Maschinenkalkül** eines Computers ist seine Maschinensprache zusammen mit ihrer internen Semantik (den semantischen Regeln, nach denen der Computer die Befehle der

19 Der ausführliche Beweis ist in [Böhm 66] erbracht worden.

Maschinensprache ausführt). Wir hatten einen Kalkül *universell* genannt, wenn in ihm jede rekursive Funktion berechnet werden kann.

Abschließend sollen einige besonders wichtige der eingeführten Begriffe zusammengestellt und die in der Literatur gängigen Bezeichnungen hinzugefügt werden.

- Die graphische Darstellung eines Kompositoperators mittels der Symbole der USB-Methode ohne Angaben zur Steuerung von Flussknoten heißt *Operandenflussgraph*, im Falle eines sprachlichen Operators auch *Datenflussgraph*.
- Ein Operandenflussgraph/Datenflussgraph, der mit allen erforderlichen Angaben zur Steuerung von Flussknoten beschriftet ist, heißt *Operandenflussplan/Datenflussplan*.
- Ein maschinenlesbarer Operandenflussplan/Datenflussplan heißt *Operandenflussprogramm/Datenflussprogramm*. Operandenflussprogramme sind *linearisierte* Operandenflusspläne, es sei denn der Rechner versteht die USB-Sprache oder eine andere, äquivalente zweidimensionale Sprache.
- Die graphische Darstellung einer Aktionsfolge mittels der Symbole der USB-Methode ohne Angaben zur Steuerung von Flussknoten heißt *Aktionsfolgegraph*.
- Ein Aktionsfolgegraph, der mit allen erforderlichen Angaben zur Steuerung der steuerbaren Flussknoten beschriftet ist, heißt *Aktionsfolgeplan*, in der Informatikliteratur meistens als *Steuerflussplan* bezeichnet.
- Ein nach DIN-Norm[20] dargestellter Aktionsfolgeplan heißt *Programmablaufplan* (abgekürzt PAP; siehe z.B. Bild 13.9).
- Ein maschinenlesbarer (i.d.R. also linearisierter) Aktionsfolgeplan heißt *Aktionsfolgeprogramm* oder *imperatives Programm*.
- Ein *Maschinenprogramm* ist eine Folge von Maschinenbefehlen. Ein *Maschinenbefehl* ist die Beschreibung einer Computeraktion. Ein terminierendes Maschinenprogramm ist ein imperativer Algorithmus.

20 DIN - Deutsches Institut für Normung.

Teil 3

Von der Maschinensprache zur künstlichen Intelligenz

14 Zwischenbilanz

Zusammenfassung

Mit dem Einprozessorrechner ist die boolesche Algebra als derjenige Kalkül realisiert worden, in den jeder andere Kalkül, dessen sich die natürliche Intelligenz bedient, zu transformieren ist. Einige Transformationsschritte sind bereits ausgeführt worden und zwar die Überführung der Maschinenoperationen in boolesche Operationen. Um uns nicht in der unübersehbaren Vielfalt der Vorgehensweisen und Methoden der natürlichen Intelligenz zu verlieren, konzentrieren wir unsere Versuche, künstliche Intelligenz zu verwirklichen, auf drei markante Rollen, die der Computer als intelligentes Werkzeug des Menschen übernehmen soll:

- der Computer als Helfer beim Lösen mathematischer Probleme,
- der Computer als Helfer beim Lösen nichtmathematischer Probleme,
- der Computer als Unterhalter, als Gesprächs- und Spielpartner.

Die Unterscheidung zwischen mathematischen und nichtmathematischen Problemen ist aus der Sicht des Menschen mit seinen alltäglichen Aufgaben, nicht aus der Sicht des Computers zu verstehen. Für den Computer gibt es keine nichtmathematischen Aufgaben. Anhand dieser Rollen wird die KI-Problematik in Teil 3 abgehandelt. Auf diese Weise kommen die wesentlichen Probleme der künstlichen Intelligenz zur Sprache.

Es versteht sich von selbst, dass der Rechner mit Zahlen, oder, wie man auch sagt, dass er *numerisch* rechnen kann. Nicht ganz so selbstverständlich ist, dass der Computer auch mit Variablen oder, wie wir im Weiteren auch sagen werden, dass er *analytisch* rechnen kann[1]. Noch merkwürdiger mag es erscheinen, dass der Computer auch nichtmathematische Probleme lösen kann. Das scheinbare *Paradoxon der KI* besteht darin, dass der "Rechner", der zum *Rechnen* gemacht ist, "*nichtmathematisch denken*" kann. Doch hat es nur den Anschein. Tatsächlich kann der Computer nur "kalkülisiert denken", d.h. rechnen. Er "rechnet " auch dann, wenn er an Spielen oder Gesprächen teilnimmt.

Der Weg zur künstlichen Intelligenz hat kein Ende; zumindest kann er kein Ende haben, solange wir nicht wissen, was *natürliche* Intelligenz "wirklich" ist, was sie vermag. Voraussetzung dafür wäre die vollständige Kenntnis der Funktionsweise des Gehirns. Und selbst dann bliebe das prinzipielle Problem der Zirkularität der Fragestellung bestehen: Kann der Mensch erkennen, was er selber kann?

---

1 Zuweilen wird das Rechnen mit Variablen auch als *symbolisches* Rechnen bezeichnet.

14.1 Probleme des Softwareweges zur KI

Da es aus Aufwandsgründen unmöglich ist, rein hardwaremäßig einen universellen Rechner zu bauen, sehen wir uns gezwungen, das Komponieren von Operatoren, aufbauend auf einer geeigneten Hardware, softwaremäßig fortzusetzen. Als Hardware wählen wir den Einprozessorrechner. Wir müssen also oberhalb der Maschinenebene, d.h. unter Verwendung der Maschinensprache, eine Softwarehierarchie komponieren. Zunächst fixieren wir noch einmal das Ziel, das erreicht werden soll. Dazu erinnern wir uns an das Ziel 2, wie es zu Beginn von Kap.13 [13.1] formuliert worden ist: *Realisierung eines* ***Basiskalküls****, in den sich alle Kalküle transformieren lassen, derer sich die natürliche Intelligenz bedient, und Durchführung der Transformationen in den Basiskalkül.* Als Basiskalkül haben wir die boolesche Algebra gewählt und auf ihrer Grundlage den Maschinenkalkül entwickelt.

Die Aufgabe von Teil 3 muss offenbar darin bestehen, verschiedene Kalküle in den Maschinenkalkül zu transformieren. Es fragt sich, um welche Kalküle es sich handeln kann. In Kap.8.6 [8.41] hatten wir das "Fernziel der KI-Forschung" folgendermaßen formuliert: *Kalkülisierung und Implementierung des folgerichtigen Denkens des gesunden Menschenverstandes.* Dieses Ziel ist recht einfach zu erreichen, wenn eine Art des Denkens simuliert werden soll, die bereits kalkülisiert ist. Das gilt z.B. für das mathematische Denken. Dieser Aufgabe ist Kapitel 15 gewidmet. Schwieriger wird es, das menschliche Denken zu simulieren, wenn es nicht oder zumindest nicht deutlich sichtbar kalkülisiert ist. Diesem Problem wenden wir uns in Kap.16 zu. Wir werden uns anhand von zwei Beispielen überlegen, wie das Kalkülisieren erfolgen kann. Das Implementieren eines Kalküls, also das Übersetzen der Kalkülsprache in die Maschinensprache, wird in Kap.16.4 behandelt.

Von den speziellen Problemen der Programmierung und der Softwaretechnologie wird in Teil 3 kaum die Rede sein. Darüber findet der Leser einiges in Teil 4. Im Augenblick kommt es uns darauf an, eine Zwischenbilanz zu ziehen, die wichtigsten Feststellungen und Ergebnisse der vorangehenden Kapitel zusammenzufassen und zu verallgemeinern und auf dieser Grundlage unser Ziel noch schärfer und unseren weiteren Weg deutlicher zu bestimmen.

Der Computer, den wir in Kap.13 entworfen haben, kann unter Verwendung seiner Maschinensprache programmiert, d.h. zur Ausführung unterschiedlicher Operationen konditioniert werden. Wie wir wissen, kann für jede rekursive Funktion ein Maschinenprogramm zu ihrer Berechnung geschrieben werden. Unser Computer ist universell, das heißt, er kann jede rekursive Funktion berechnen, wenn er über ein entsprechendes Programm verfügt, vorausgesetzt, Rechenzeit und Speicherkapazität stehen in ausreichendem Maße zur Verfügung.

Man könnte hinzufügen: Sonst kann der Computer nichts. Dieser Satz ist jedoch irreführend. Denn es gibt "fast nichts", was er nicht kann. Doch dieses "fast nichts" ist etwas sehr Wichtiges. Der Computer scheint nicht imstande zu sein, die Zeichenketten, mit denen er hantiert, mit externer Semantik zu belegen, zumindest nicht in

dem Sinne, wie der Mensch dies tut, wenn er über die Welt nachdenkt oder wenn er sich mit anderen Menschen unterhält. Anders ausgedrückt, der Computer kann den Zeichenrealemen keine Ideme zuordnen, es sei denn, man würde ihm ein Bewusstsein zuschreiben. Diesen Gedanken wollen wir nicht weiterverfolgen. Er würde uns vom Wege abbringen. Denn unser Weg soll nicht zum künstlichen Bewusstsein, sondern "nur" zur künstlichen Intelligenz führen, d.h. zur Fähigkeit des Computers, zu wahren Aussagen über die Wirklichkeit zu gelangen, aber eben zu Aussagen, die erst im Bewusstsein des Menschen mit externer Semantik belegt werden müssen.

Der Weg, auf dem wir das erreichen wollen, ist durch die USB-Methode vorgezeichnet. Er besteht in der Fortsetzung des in der Hardware begonnenen Weges in der Software, m.a.W. in der Komponierung von Operationsvorschriften, d.h. sprachlicher Operatoren, nach den gleichen Regeln, nach denen wir bisher reale Operatoren komponiert haben. Das bedeutet, dass die Flussknoten der Hardware ihr Pendant in der Software haben müssen, dass also die verwendete Programmiersprache über Sprachelemente zur Beschreibung aller Flussknotentypen verfügen muss, wenn sie universell sein soll. Für die Maschinensprache trifft das zu, wovon wir uns in Kap.13.7 überzeugt haben. Das bedeutet, dass mit ihrer Hilfe jede KR-berechenbare Funktion in Form eines imperativen Programms beschrieben, d.h. softwaremäßig aus den Sprachelementen der Maschinensprache "komponiert" werden kann. Es bedeutet aber noch nicht, dass ihre Ausdrucksmittel ausreichen, um Netze und Hierarchien sprachlicher Operatoren zu komponieren.

Hier liegt ein Problem der Programmierungstechnik, das den Informatikern erheblich zu schaffen gemacht hat und das auch uns noch beschäftigen wird. Im Augenblick begnügen wir uns mit folgender Feststellung. Damit der Programmierer sprachliche Kompositoperatoren nach den Regeln der USB-Methode komponieren kann, müssen die Bausteinoperatoren ebenso selbständige, in sich abgeschlossene Einheiten darstellen, wie Hardwareoperatoren, sodass der Programmierer mit ihnen hantieren und sie zu Netzen verknüpfen kann. Dieser Forderung genügen Maschinensprachen, wie wir sie in Kap.13 entworfen haben, leider nicht, denn sie dienen ausschließlich der Artikulierung von Aktionsfolgeprogrammen, nicht aber von Datenflussprogrammen. Dass das Problem im Prinzip lösbar sein muss, folgt aus der Tatsache, dass sich jeder Aktionsfolgeplan in einen Datenflussplan überführen lässt. In Kap.15.4 werden wir die Idee zur Lösung des Problems kennen lernen und die Lösung selber als *prozedurale Abstraktion* bezeichnen.

Wenn der Schritt von der Aktionsfolge zum Datenfluss geschafft ist, sodass mit Datenflussplänen weitergearbeitet (weiterkomponiert) werden kann, ist der Weg, der zu dem gesuchten intelligenten informationellen System führen soll, durch die USB-Methode klar vorgezeichnet. Wenn wir als Ziel unserer Bemühungen die künstliche Intelligenz deklarieren, ergibt sich eine merkwürdige Situation: Der Weg ist klar, das Ziel dagegen nicht. Denn es ist durchaus nicht klar, worin die natürliche Intelligenz, die simuliert werden soll, genau besteht, und wieweit sie geht, m.a.W. wieweit die Fähigkeit des Menschen geht, die Welt sprachlich zu modellieren. Der

Begriff der Modellierbarkeit der Welt ist ein intuitiver Begriff, und als solcher nicht exakt definierbar. Wir erinnern uns, dass das Gleiche für den Begriff der Berechenbarkeit gilt. Die Ursache ist in beiden Fällen ein und dieselbe: die Zirkularität, die in der Frage nach den eigenen Fähigkeiten liegt (vgl. Kap.8.3 [8.23]).

14.2 Drei Rollen des Computers als eines intelligenten Partners des Menschen

Die Unmöglichkeit, das Ziel der KI-Forschung explizit, extensional zu bestimmen, lässt sich nicht dadurch aus dem Wege räumen, dass man das "Fernziel" der KI-Forschung als Simulation des gesunden Menschenverstandes definiert und die beiden Schritte dahin festgelegt, das Kalkülisieren und das Implementieren. Die extensionale Undefinierbarkeit der Intelligenz bedeutet aber nicht, dass es überhaupt keinen Weg zur KI gibt; sie bedeutet lediglich, dass der Weg kein Ende hat. Die Überschrift des Teils 3 bezeichnet nicht Anfang und Ende, sondern Anfang und Richtung des Weges. Aus diesem Grunde lassen wir es bei obiger Konkretisierung des Fernziels der KI bewenden, geben ihr aber eine Form, die der öffentlichen Diskussion zur künstlichen Intelligenz besser gerecht wird, die oft als Diskussion um die "Rolle des Computers als Partner der Menschen" geführt wird.

Wir konzentrieren uns auf drei Rollen, die der Computer als intelligentes Werkzeug des Menschen spielen soll:

1. der Computer als Helfer beim Lösen mathematischer Probleme,
2. der Computer als Helfer beim Lösen nichtmathematischer Probleme,
3. der Computer als Unterhalter, als Gesprächs- und Spielpartner.

Nur in der letzten Rolle tritt der Computer tatsächlich als Partner auf. In den ersten beiden Rollen tritt er eher als Werkzeug auf. Wenn sich jedoch Mensch und Computer beim Problemlösen gegenseitig helfen, kann der Computer als *Kooperationspartner* auch in den ersten beiden Rollen auftreten. Damit der Computer diese beiden Rollen spielen kann, muss er offenbar in der Lage sein, aus gegebenen Prämissen logisch richtige Schlüsse zu ziehen, und zwar je nach Aufgabe auf mathematischem oder auf (scheinbar) nichtmathematischem Wege. In diesem Sinne liegt es nahe, zwischen mathematischer und nichtmathematischer KI zu unterscheiden. Doch ist die Wortverbindung "nichtmathematische KI" missverständlich. Denn der Computer kann nur in seinem Maschinenkalkül "denken", d.h. rechnen, bzw. in irgendeinem anderen Kalkül, falls dieser durch ein Übersetzerprogramm in seinen eigenen Kalkül transformiert worden ist. Nach dem Kalkültransformationssatz [8.39] ist das stets möglich. Im Gegensatz zum Menschen ist der Computer also nicht in der Lage, nichtmathematisch zu denken. Darum gibt es aus der Sicht des Computers keine nichtmathematischen Probleme, sondern nur aus der Sicht des Menschen und es wäre richtiger, anstelle von nichtmathematischer KI von simulierter natürlicher nichtmathemati-

scher Intelligenz zu sprechen. Das ist zu bedenken, wenn im Weiteren der Kürze halber von Nichtmathematischer KI die Rede ist.

Auch wenn man weiß, dass der Computer das Vorgehen des Menschen beim Lösen nichtmathematischer Probleme nur simuliert, ist es - selbst für den Fachmann - immer wieder faszinierend zu erleben, was der Computer mit den Operationen der ALU anstellen kann. Es erscheint geradezu paradox, dass er "vernünftig denken" kann, obwohl er letztlich nur Bitketten vergleicht und transformiert. Diesen erstaunlichen Umstand bezeichnen wir als das "**Paradoxon der KI**". In Kap.16 werden wir uns überlegen, wann und wie es möglich ist, den Computer zu befähigen, nichtmathematisches Denken zu simulieren.

Die Reihenfolge, in der die drei Rollen des Computers aufgezählt sind, spiegelt etwa die Entwicklungsetappen der KI wider. Als erstes konnte der Rechner rechnen. Dagegen ist er bis heute kaum im Stande, ein "Allerweltsgespräch" zu führen und über alles Mögliche sinnvoll zu plaudern. Die Entwicklung der natürlichen Intelligenz des Menschen vom Kleinkind bis zum Erwachsenen verläuft entgegengesetzt. Diesen merkwürdigen Gegensatz hatten wir in Kap.7.1 [7.1] mit dem Wort "Gegenläufigkeitsphänomen" charakterisiert.

Die drei Rollen, die der Computer spielen soll, verschieben in der angegebenen Reihenfolge die Grenze der KI immer weiter in den Bereich hinein, den viele Menschen naturgemäß als ihre eigene "menschliche" Domäne empfinden und den sie gegen das Eindringen der KI aus den unterschiedlichsten, teilweise emotionalen oder ethischen Gründen zu verteidigen sich veranlasst fühlen. Die Folge ist, dass die Frage nach den Grenzen der KI breit und heiß diskutiert wird.

Angesichts der Unbegrenztheit des Weges zur KI ist die Frage nach den Grenzen der KI, also nach dem Ende des Weges zu ihr, inhaltlos. Doch spielt sie in der öffentlichen Diskussion eine derartige Rolle, dass wir sie nicht übergehen dürfen. Wir werden sie immer wieder stellen. Wir werden sie aber nie allgemein, sondern nur von unserem momentanen Standpunkt aus stellen und beantworten. Wir werden stets bemüht sein, die Grenze dessen, was wir auf unserem Weg (in den einzelnen Kapiteln) erreicht haben, klar zu benennen.

Unser Weg wird gemäß der Überschrift des dritten Teils die Richtung der technischen Entwicklung "vom Rechnen zum Erkennen" (vgl.[7.1]) einschlagen und mit der "mathematischen KI" beginnen, genauer mit dem Rechnen "unmittelbar oberhalb" der Hardware, also mit dem Rechnen, das über die Operationen der ALU und des Prozessors, m.a.W. über die Operationen der Maschinensprache hinausgeht, zumindest ein klein wenig, indem es diese Operationen als Bausteinoperationen verwendet. Dabei werden diese jedoch nicht hardwaremäßig miteinander verbunden, sondern die Verbindungen werden in einem Programm beschrieben, das vom Prozessor abzuarbeiten ist.

Zum Begriff der mathematischen KI, wie er hier verwendet wird, ist folgende wichtige Bemerkung zu machen. Wir werden uns bei ihrer Realisierung zunächst auf deduktive Intelligenz beschränken und intuitive Intelligenz ausschließen. Dies be-

deutet für das im Weiteren zu besprechende mathematische Modellieren, dass die Erstellung eines ersten mathematischen Modells Aufgabe des Menschen ist, denn sie erfordert Intuition (oder Kreativität, wie zuweilen gesagt wird). Der Rechner soll ausschließlich beim deduzierenden Arbeiten mit dem Modell, also beim Rechnen in dem Kalkül helfen, aus dem sich das Modell durch Interpretation ergibt, d.h. durch Ersetzen kalkülinterner Bezeichner durch externe Bezeichner.

Abschließend sei darauf hingewiesen, dass hinter der Zielstellung der KI, den natürlichen Menschenverstand zu simulieren, andere Ziele stehen können, beispielsweise die Herausbildung besserer gesellschaftlicher Organisationsformen, um so die anstehenden sozialen Probleme zu lösen und die Überlebenswahrscheinlichkeit der Menschheit zu erhöhen. Das Ergebnis wäre ein Evolutionsschritt, bewirkt durch den (bewussten) Willen zur Arterhaltung. Auf derartige Sekundärziele der KI werden wir nicht eingehen. Doch sei eine eher philosophische Bemerkung zu den "Zielen" der KI erlaubt. Das Ergebnis der "zielstrebigen" Bemühungen der Menschen ist der "Fortschritt" der kulturellen Evolution. Die aber hat keine Ziele, auch nicht, wenn der Mensch sich einbildet, sie stellen zu können - eine paradoxe Situation. Die Volksweisheit hat sie in die Worte gekleidet: " Der Mensch denkt, Gott lenkt".

15 Lösen mathematischer Probleme

Zusammenfassung

Aus den Maschinenoperationen, letztendlich also aus den ALU-Operationen, können unter Verwendung der Maschinensprache beliebige arithmetische Operationen komponiert (programmiert) werden, m.a.W. der arithmetische Kalkül kann in den booleschen transformiert werden. Das Programmieren im internen Binärcode des Computers (in der *intern codierten Maschinensprache*) ist unangenehm und aufwendig. Es wird durch die Definition und Implementierung von *Assemblersprachen* und von *höheren Programmiersprachen* erleichtert.

Eine Assemblersprache kann als "*gehobene Maschinensprache*" aufgefasst werden, denn sie stellt im Gegensatz zu höheren Programmiersprachen keine Erweiterung der intern codierten Maschinensprache dar, erlaubt aber die Verwendung von Bezeichnern für Operanden und Operationen, auch für Kompositoperationen, also für Programme. Das Übersetzerprogramm, das die Übersetzung von Programmen aus der Assemblersprache in den internen Code ausführt, heißt *Assembler*.

Höhere Programmiersprachen enthalten ausdrucksstärkere Sprachelemente für die Komponierung sowohl von Kompositoperanden, beispielsweise Vektoren oder Matrizen, als auch von Kompositoperationen. So sind für die Programmierung von Iterationen verschiedene Sprachelemente entwickelt worden, z.B. die WHILE-Anweisung. Ein Programm, das in einer höheren Programmiersprache geschrieben ist, muss in die Maschinensprache des Computers übersetzt werden, der das Programm abarbeiten soll.

Assemblersprachen und höhere Programmiersprachen dienen der Überwindung der "semantischen Lücke" zwischen der Sprache, in welcher der Programmierer normalerweise denkt und sich ausdrückt, und der Sprache, die der "sprachbegabte" (d.h. mit Übersetzerprogrammen ausgerüstete) Computer versteht. Damit dienen sie der Lösung des *technischen Semantikproblems*, d.h. der Anbindung der Nutzersemantik, also der externen oder formalen Semantik, in welcher der Nutzer denkt, an die interne Computersemantik, an die Prozesse im Computer. Die Anbindung lässt sich verhältnismäßig leicht bewerkstelligen, wenn der Nutzer (beispielsweise ein Mathematiker) in einem Kalkül denkt. Ihm fällt der Sprung über die semantische Lücke umso leichter, je verwandter seine Kalkülsprache der Sprache des Computers (der Programmiersprache) ist. Der Wunsch, die semantische Lücke zu verringern, ist der Motor der Entwicklung neuer Programmiersprachen.

Der Computer kann nicht nur numerische (zahlenmäßige), sondern auch analytische Rechnungen (Rechnungen mit Variablen) ausführen. Analytisches Rechnen besteht im Überführen analytischer Ausdrücke in andere Ausdrücke, z.B. $(a+b)(a-b)$ in (a^2-b^2) oder $\sin x/\cos x$ in $\tan x$. Der Computer bedient sich beim analytischen Rechnen - ebenso wie gegebenenfalls der Mensch - einer Formelsammlung. Sie kann

beispielsweise die “Formel” $\sin x/\cos x = \tan x$ enthalten. Das Gleichheitszeichen in einer Formel kann als zweiseitige Ergibtgleichung gelesen werden; die rechte Seite ergibt sich aus der linken und umgekehrt. Das Hantieren des Computers mit Formeln heißt *Formelmanipulation*.

Die wichtigsten Bausteinoperationen der Formelmanipulation sind das *Suchen* (es schließt *Vergleichen* ein) und das *Substituieren*. Um einen analytischen Ausdruck umzuformen, wird zunächst durch Vergleich nach einer Formel gesucht, die auf den umzuformenden Ausdruck anwendbar ist. Das ist dann der Fall, wenn der Ausdruck eine Teilzeichenkette enthält, die eine vom Kontext unabhängig ausführbare Anweisung darstellt und die mit der linken oder rechten Seite der Formel identisch ist oder durch Bezeichnerabgleich (geeignete Änderung der Bezeichner) identisch gemacht werden kann, sodass die Teilzeichenkette durch die andere Seite der Formel substituiert werden kann. Existiert eine solche Formel, wird die Substitution ausgeführt. Wenn mehrere Formeln gleichzeitig anwendbar sind, muss der Lösungsweg durch Versuchen gefunden werden, sodass das Rechnen zu einem nichtdeterministischen Prozess wird. Die Berechnungsvorschrift stellt einen *nichtdeterministischen Algorithmus* dar. Durch eine zusätzliche Vorschrift für die eindeutige Wahl der anzuwendenden Formel kann ein nichtdeterministischer Algorithmus in einen deterministischen überführt werden.

Programme der Formelmanipulation und des analytischen Rechnens werden vorteilhafterweise in funktionalen Programmiersprachen geschrieben. Das Übersetzen in eine Maschinensprache ist immer möglich. Auf diese Weise lässt sich jeder analytische Kalkül in den booleschen transformieren.

Der Mechanismus der Formelmanipulation ist auch dann anwendbar, wenn nicht analytische Ausdrücke gemäß mathematischer Formeln, sondern wenn beliebige Zeichenketten (Wörter) gemäß einer Liste von Substitutionsregeln umgeformt werden. Dann spricht man von *Wortmanipulation*. Durch einen deterministischen Manipulationsalgorithmus (samt Regelliste) wird eine eindeutige Abbildung aus einer gegebenen Eingabewortmenge in eine Ausgabewortmenge, also eine Funktion festgelegt. Die Klasse der durch deterministische Wortmanipulation berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Nicht nur das analytische, sondern auch das numerische Rechnen ist im Grunde ein Substituieren. Beispielsweise können die Zeilen des kleinen Einmaleins als Substitutionsregeln aufgefasst werden. Die Tätigkeit eines Computers ist *immer* ein Substituieren. Dabei arbeitet er niemals mit Bedeutungen, niemals mit externer Semantik, sondern ausschließlich mit Zeichenketten, er arbeitet nicht mit Idemen, sondern mit Realemen.

Mathematisches Operieren ist stets Deduzieren in einem Kalkül, d.h. numerisches oder analytisches Rechnen. Durch entsprechende Software kann der Computer befähigt werden, jede beliebige numerische oder analytische Rechnung und damit jede mathematische Operation auszuführen.

15.1 Funktionales Programmieren

Wir wenden uns der ersten der drei Rollen des Computers zu, die wir in Kap.14.2 ins Auge gefasst haben, der Rolle eines Helfers beim Lösen mathematischer Aufgaben. Beim Nachdenken darüber, wie der Computer befähigt werden kann, diese Rolle zu spielen, stößt man auf einen scheinbar fatalen Widerspruch zwischen den Fähigkeiten des Computers und der ihm zugedachten Rolle. Der Einprozessorrechner von Bild 13.7 kann nur seine Maschinenprogramme, d.h. speziell notierte imperative Algorithmen ausführen. Die übliche Mathematik, wie sie in Naturwissenschaft und Technik, nicht selten aber auch im Alltag zur Anwendung kommt, bevorzugt formalisierte Sprachen, die nicht die imperative, sondern die funktionale Notation verwenden. Man denke an die Gleichung der Geraden, an das Fallgesetz oder an die Funktion (8.1), die durch das Operatorennetz von Bild Abb.8.1 berechnet wird. Zur Erinnerung sei (8.1) noch einmal angegeben, diesmal jedoch unter der Annahme, dass der Exponent n kein konstanter Parameter, sondern eine Variable ist:

$$y = f_1(x,n) = x^n + x \quad \text{für } x \leq 0 \tag{15.1a}$$

$$y = f_2(x,n) = x^n + \sin x \quad \text{für } x > 0, \tag{15.1b}$$

wobei n eine ganze Zahl ≥ 0 ist.

Es wäre eine verlockende Idee, die Sprache der Mathematik als Programmiersprache zu verwenden, sodass man nur die Formeln (15.1) über die Tastatur in seinen PC einzugeben brauchte, um ihn zu befähigen, die Funktion $f(x,n)$ zu berechnen. Dass dies im Prinzip möglich ist, folgt aus dem Kalkül-Tansformationssatz [8.39]. Erforderlich wäre ein Übersetzerprogramm, dass die funktionale Sprache der Mathematik in die imperative Maschinensprache des PC übersetzt. In Kap.15.9 werden wir uns überlegen, wie das im Prinzip möglich ist. Im Augenblick gehen wir davon aus, dass das Übersetzungsproblem gelöst ist. Die Lösung verlangt aber eine "*computerangepasste*" funktionale Notation der zu übersetzenden Sprache, d.h. eine Notation, die vollkommen eindeutig ist und die ein effizientes Übersetzen ermöglicht. Wenn wir darangehen, eine funktionale Programmiersprache zu entwickeln, die diese Forderungen erfüllt, bleibt die interne Semantik funktionaler Programme im Dunkeln, denn wir wissen nicht, wie sie übersetzt werden und welche Prozesse bei der Ausführung der übersetzten Programme im Computer auslöst werden. Darin liegt ein Verstoß gegen das Trägerprinzip, den wir in Kauf nehmen, um den begonnenen Gedankengang nicht unterbrechen zu müssen.

Ein kritischer Blick auf die Ausdrücke (15.1) zeigt, dass die Notation nicht sehr geeignet ist, um unmittelbar als Computerprogramm zu dienen. Man erkennt nämlich, dass für die Notation von Operationen drei unterschiedliche Syntaxregeln zu Anwendung kommen. Bei der Addition steht der Operationsbezeichner *zwischen* den beiden Operanden, bei der Sinusoperation steht er *vor* dem Operanden und bei der Potenzierung tritt gar kein Bezeichner auf, sondern die Operation ist durch Hochstel-

lung des Exponenten gekennzeichnet. Diese Uneinheitlichkeit der Syntax geht zu Lasten der Übersetzung. Hinzu kommt, dass die Hochstellung die Linearität (Eindimensionalität) der Sprache verletzt; die Eingaben sind keine reine Zeichen*ketten*. Wir vereinheitlichen die Notationsweise und gewährleisten die Linearität, indem wir festlegen, dass das Operationssymbol (der Bezeichner der Operation oder Funktion) stets explizit angegeben und den Operanden vorangestellt wird.

In Kap. 8.4.7 [8.34] waren zwei spezielle derartige funktionale Notationsweisen eingeführt worden, die *Präfixnotation* in (8.15) und die *Listennotation* in (8.16). Danach sind f_1 und f_2 folgendermaßen zu notieren, in Präfixnotation

$$f_1 = +(\mathrm{pot}(x,n), x)$$

$$f_2 = +(\mathrm{pot}(x,n), \sin(x)) \qquad (15.2)$$

und in Listennotation

$$f_1 = (+\ (\mathrm{pot}\ x\ n)\ x))$$

$$f_2 = (+\ (\mathrm{pot}\ x\ n)\ (\sin\ x)) \qquad (15.3)$$

Diese Notationen erinnern wenig an einen Algorithmus oder eine Aktionsfolge, eher an einen Datenflussplan. Beispielsweise kann man die zweite Formel in (15.3) folgendermaßen lesen. Der x-Wert wird sowohl dem sin-Operator als auch dem Potenzoperator als Eingabeoperand übergeben; sodann werden die Ausgabeoperanden beider Operatoren an den Additionsoperator als Eingabeoperanden weitergegeben, der als Ausgabeoperand schließlich den Funktionswert liefert. Die Eingabeoperanden des Addierers treten nicht explizit auf, sie werden vielmehr durch (pot x n) und (sin x) dargestellt. Eine solche Notations- und Interpretationsweise hatten wir *funktional* genannt.

Hierin liegt der Kern der funktionalen Notation und des funktionalen Programmierens. *In einem* ***funktional notierten Programm*** *steht der Bezeichner eines Operators (einer Operation, einer Funktion) zusammen mit dem Eingabeoperanden (Eingabeoperandentupel) für den Wert, den die Operationsausführung (die Funktionsberechnung) liefert.*

Danach stellt jeder Klammerausdruck in (15.3) einen Wert dar. Syntaktisch stellt er eine Liste von Elementen dar. Das erste Element der Liste bezeichnet die Operation (Funktion), die folgenden Elemente bezeichnen die Argumente. Sie können ihrerseits wiederum Listen darstellen, sodass Listen hierarchisch strukturiert oder geschachtelt sein können. Für (15.3) trifft dies zu. In der USB-Terminologie wäre eine geschachtelte Liste als *Kompositliste* zu bezeichnen, die aus *Bausteinlisten* komponiert ist. Komponieren ist in diesem Fall ein Aneinanderreihen, eine sog. *Konkatenation*. Die Möglichkeit der listenförmigen Notation war offenbar ein Anlass für McCarthy, die listenorientierte Programmiersprache Lisp zu entwickeln.

Aus der Sprache Lisp sind viele Abkömmlinge hervorgegangen, u.a. die Sprache CommonLisp. In dieser Sprache nimmt (15.3) folgenden Form an (vgl. Bild 20.3 Zeile 3):

```
(defun f2(x n)(+(pot x n)(sin x))).
```

Die Zeichenkette `defun f2(x n)` bedeutet, dass die Funktion `f2(x n)` durch den nachfolgenden Ausdruck definiert wird. Dem entspricht die Bedeutung des Gleichheitszeichen in (15.3), wo es ein Definitionszeichen darstellt. Weitere Kommentare erübrigen sich.

Der Umstand, dass in der funktionalen Notation Zwischenresultate nicht explizit benannt werden, hat zur Folge, dass Iterationen *rekursiv* formuliert werden müssen. Darauf war bereits in Kap.8.4.6 [8.32] hingewiesen worden und auch darauf, dass die Definition der rekursiven Iteration in Kap. 8.4.5 eine entsprechende Notation in Form von (8.9) anbietet. Speziell ist in (8.11) für das Potenzieren durch iteratives Multiplizieren eine funktionale Notation angegeben. Ersetzt man f durch pot, geht (8.11) in

$$\mathrm{pot}(x,n) = x * \mathrm{pot}(x,n-1) \quad \text{für } n>0$$

$$\mathrm{pot}(x,0) = 1 \tag{15.4a}$$

über. Die funktionale Notation wird dadurch möglich, dass die zu definierende Funktion (in (8.9) und (8.11) mit f und in (15.4a) mit pot bezeichnet) in dem definierenden Ausdruck auftritt, dass also die Funktion sich gewissermaßen durch sich selbst definiert. Eben in diesem Sinne wird das Wort Rekursion in der Programmierungstechnik benutzt [8.30]. In CommonLisp wäre folgendermaßen zu notieren (vgl. Bild 20.3 Zeile 4):

```
(defun pot(x n)(if(= n 0)1(* x(pot x(- n 1)))))
```
(15.4b)

Die if-Funktion war in Kap.8.4.7 [8.36] eingeführt und im Anschluss an (8.21) erklärt worden.

Die Darstellung der rekursiven Berechnung durch ein Operatorennetz in den Bildern 8.1 und 8.9 und in den Formeln (8.9) und (8.11) demonstriert die enge Verwandtschaft zwischen funktionaler Programmierung und Operandenfluss- bzw. Datenflussprogrammierung (in der Literatur oft als datenflussorientierte Programmierung bezeichnet). Beide Programmierungsarten beruhen auf der Vorstellung eines Operandenflusses durch ein Operatorennetz. Insofern ist funktionales Programmieren datenflussorientiert zu nennen. Auch bei der Operandenflussprogrammierung erübrigt sich die explizite Benennung von Zwischenergebnissen, da deren Weg durch den Operandenflussplan unmittelbar vorgegeben ist, im Unterschied zur Aktionsfolgeprogrammierung, wo alle Wege über den zentralen Arbeitsspeicher führen, sodass die jeweiligen Speicherlätze über die Namen der Zwischenergebnisse explizit angegeben werden müssen [13.8]. 1

Die vorangehenden Überlegungen verstoßen, wie gesagt, gegen das Trägerprinzip, weil die interne Semantik funktionaler Programme völlig im Dunkeln geblieben

ist. Wir wollen etwas Licht in das Dunkel bringen, indem wir uns überlegen, wie das Maschinenprogramm aussehen könnte, in das ein funktionales Programm übersetzt wird. Das Übersetzen selber bleibt zunächst außer Betracht (siehe Kap.15.9).

15.2 Imperatives Programmieren

Unsere Maschinensprache von Kap.13.5.2 ist offensichtlich keine funktionale Sprache, denn jeder Operand wird explizit benannt; die Konstruktion des Prozessorrechners verlangt dies, denn das Resultat einer Rechnung steht dem Prozessor nicht mehr unmittelbar zur Verfügung, nachdem es im Hauptspeicher abgespeichert und der AC (Akkumulator) mit einem anderen Wert überschrieben worden ist. Wenn der Prozessor es benötigt, muss er es benennen und anfordern. Das bedeutet, dass Maschinenprogramme (falls sie terminieren) imperative Algorithmen [7.10] sind, wie wir bereits wissen. Es sei an die folgenden, in Kap.13.5.1 [13.9] eingeführten Begriffe erinnert: *Ein maschinenverständlich notierter imperativer Algorithmus heißt* ***imperatives Programm****. Programmiersprachen, die das Schreiben imperativer Programme unterstützen, heißen* ***imperative Sprachen****.*

Wir wollen nun ein Maschinenprogramm zur Berechnung der durch (15.1b) definierten Funktion $y = f_2(x,n)$ schreiben. Wir nehmen an, dass die Maschinensprache den Operationscode `SIN` für die Sinusfunktion und den Operationscode `POT` für das Potenzieren enthält. Ihre Komponierung aus den ALU-Operationen kann beispielsweise durch ein Matrixsteuerwerk realisiert werden (siehe Kap.13.5.6). Die vier Befehle im Programm von Bild 15.1 stellen den zentralen Abschnitt eines möglichen Maschinenprogramms dar. Anstelle der Variablenadressen sind (nach dem Vorbild von Bild 13.6) die in spitze Klammern gesetzte Variablenbezeichner eingetragen. Beispielsweise stellt <x> die Adresse dar, unter der die Werte der Variablen x abgespeichert werden. Die Befehle haben folgende Wirkungen im Computer (folgende interne Semantik):

```
1  TVS <x>
2  POT <n> <y>
3  SIN <x>
4  ADD <y> <y>
```

Bild 15.1 Fiktives Maschinenprogramm zur Berechnung der Funktion $y = x^n + \sin x$.

Befehl 1: Transport des Wertes der Variablen x vom Speicher in den Akkumulator.

Befehl 2: Holen des Exponenten (d.h. Transport des Wertes von n vom Speicher in das dafür vorgesehene Datenregister), Potenzieren und Speichern der Potenz unter der Adresse der Variablen y (das ist erlaubt, obwohl das Ergebnis noch nicht der Wert von f_2 ist).

Befehl 3: Holen des Wertes der Variablen x und Berechnen von $\sin(x)$. Das Ergebnis bleibt nur im AC stehen, es wird nicht im HS gespeichert. Wenn der Befehl gar keine Adresse enthielte, würde nach seiner Ausführung im AC der Wert von $\sin(x^n)$ stehen.

Befehl 4: Holen des unter <y> gespeicherten Wertes, Addieren dieses Wertes mit dem im AC stehenden Wert von $\sin(x)$ und Abspeichern der Summe (des Wertes von f_2) in der Speicherzelle der Variablen y.

Wenn die Maschinensprache die POT-Operation nicht zur Verfügung stellt, obliegt es dem Programmierer, das Potenzieren auf das Multiplizieren zurückzuführen (vorausgesetzt die Maschinensprache enthält die MUL-Operation). Im Programm von Bild 15.2a ist dies geschehen. Es wird davon ausgegangen, dass die Werte von x und n bereits im Hauptspeicher stehen. Der dafür zuständige *Deklarationsteil* des Programms, der die Variablen *deklariert* (dem Computer "erklärt") und die entsprechenden Wertzuweisungen sind unterschlagen. Ferner wird angenommen, dass die Konstanten 0 und 1 unter den Adressen der Bezeichner `NULL` und `EINS` fest abgespeichert sind.

```
BEGINN 4000                PROGRAMM f2;
 1  TVS <EINS>
 2  TNS <y>                   y := 1;
 3  TVS <NULL>
 4  TNS <z>                   z := 0;
 5  TVS <x>
 6  MUL <y> <y>           ZYK y := x*y;
 7  INK <z>                   z := z+1;
 8  SPK <n> 5                 Wenn z kleiner als n
                               dann springe nach ZYK
                             sonst
 9  SIN <x>
10  ADD<y> <y>                  y := y+sin(x);
ENDE                       ENDE
```

(a) **(b)**

Bild 15.2 Maschinenprogramm zur Berechnung der Funktion $y = x^n + \sin x$. (a) - Programm; (b) - Kommentar.

Der Leser wird das Programm ohne Schwierigkeiten verstehen, wenn er zum Verständnis der linken Zeilen von Bild 15.2 (der Zeilen von Bild 15.2a) die rechten Zeilen (die Zeilen von Bild 15.2b) als Kommentare zurate zieht. Die Bedeutungen (die interne Semantik) der Operationscodes `TVS` und `TNS` (Transport vom bzw. nach dem Speicher) sind aus Kap. 13.5.3 bekannt. Offenbar bewirken die linken Befehle

der Zeilen 1 und 2 gemeinsam die Abspeicherung einer 1 unter der Adresse <y>, also die auf der rechten Seite angegebene Wertzuweisung. Analoges gilt für die Befehle 3 und 4. Der MUL-Befehl entspricht intersemantisch dem ADD-Befehl, nur wird die ALU nicht auf Addition, sondern auf Multiplikation konditioniert. Zeile 7 befiehlt die Inkrementierung von z.

Befehl 8 ist auf der rechten Seite ziemlich ausführlich kommentiert. Der Operationscode SPK ist die Abkürzung von "SPringe, wenn Kleiner". Der Sprungbefehl ist folgendermaßen zu lesen: "Wenn der Inhalt des Akkumulators kleiner ist als z, springe nach Befehl 5". Auf der rechten Seite steht zwischen Wenn und dann die Bedingung, unter der gesprungen werden soll. Der Kommentar zu Befehl 8 artikuliert einen *bedingten Sprungbefehl* in einer unmittelbar verständlichen Form. Das Sprungziel (Befehl 5) ist im Kommentar *markiert*, d.h. die Zeile beginnt mit einer *Marke*, in diesem Fall mit der Marke ZYK. Die Markierung ist notwendig, da die Anweisungen (Kommentare) nicht nummeriert sind.

Das Programm bewerkstelligt das Potenzieren mit Hilfe eines *Zyklus*, d.h. durch wiederholtes Zurückspringen von Zeile 8 zu Zeile 5. Dem Zyklus entspricht die Rückkopplungschleife in Bild 8.1. Das Potenzieren erfolgt durch iteriertes Multiplizieren. Nach der Inkrementierung (Zeile 7) steht unter der Adresse <z> die aktuelle Iterationszahl, also die Anzahl der bis dahin ausgeführten Multiplikationen und unter der Adresse <y> die entsprechende Potenz von x. Wenn beispielsweise der Wert von f_2 für x=2 und n=3 berechnet werden soll, wird die ursprüngliche 1 unter der Adresse <y> der Reihe durch die Zahlen 2, 4, 8 überschrieben. Sobald unter der Adresse <z> die Zahl 3 steht, wird die Iteration abgebrochen. Die Zeilen 5 bis 8 bilden gemeinsam die imperative Notation einer funktionalen Iteration [8.32]. Die Befehle 9 und 10 sind auf der rechten Seite in einer einzigen Anweisung zusammengefasst.

Der bedingte Sprungbefehl 8 unterscheidet sich vom Sprungbefehl der Registermaschine in Kap. 8.4.3. Es sind viele bedingte Sprungbefehle denkbar, von denen in der Regel aber nur einige wenige in einer konkreten Maschinensprache realisiert sind. Das Grundprinzip der Iteration ist, unabhängig vom speziellen Sprungbefehl, stets das gleiche. Es ist in Kap.8.4.5 behandelt worden.

Damit die rechten Zeilen in Bild 15.2 die Rolle von Kommentaren spielen können, müssen sie die Semantik der linken Zeilen "mit einfacheren Worten", d.h. sinnfälliger artikulieren. Die für die weitere Entwicklung der Rechentechnik entscheidende Idee bestand darin, den Kommentar so exakt und vollständig auszuformulieren, dass seine Übersetzung in die Maschinensprache algorithmierbar und implementierbar wird. Diese *Exaktheitsforderung* verlangt, dass die Kommentarsprache zu einer Kalkülsprache formalisiert wird. Besonders wichtig für die Lesbarkeit der Kommentare ist die Ersetzung der Adressen durch Bezeichner, beispielsweise <y> durch y. In Kap.15.4 werden wir uns überlegen, wie der Computer befähigt werden kann, Bezeichner zu "verstehen".

Die Kommentare von Bild 15.2b erfüllen die Exaktheitsforderung an eine Programmiersprache bereits im Wesentlichen (abgesehen von dem fehlenden Deklara-

tionsteil). Das ist durch Befolgung bestimmter Syntaxregeln erreicht worden. Dazu gehört das Abschließen jeder Anweisung mit einem Semikolon und das "Einklammern" des Programms in die Wörter `PROGRAMM` und `ENDE`. Diese Regeln gewährleisten, dass das Übersetzerprogramm Anfang und Ende der einzelnen Anweisungen sowie des ganzen Programms richtig erkennt. Die Nummerierung der Anweisungen erübrigt sich.

Das `Programm f2` in Bild 15.2b ist offensichtlich ein *imperatives* Programm, denn es besteht, ebenso wie das Maschinenprogramm, aus Befehlen, die üblicherweise nicht als Befehle, sondern als *Anweisungen* bezeichnet werden. Die Artikulierung der Kommentare ist recht willkürlich. Ebenso willkürlich ist die Definition einer Programmiersprache, die aus derartigen Kommentaren hervorgegangen ist. Abgesehen von der Exaktheitsforderung sind der Phantasie der Spracherfinder kaum Grenzen gesetzt. Es ist also nicht verwunderlich, dass die unterschiedlichsten Sprachen entwickelt worden sind. Ob sich eine Sprache durchsetzt, hängt zum einen von ihrer Nutzerfreundlichkeit ab, vor allem von der Lesbarkeit, der Ausdrucksstärke und der Fehleranfälligkeit der artikulierbaren Programme, und zum anderen davon, wie effektiv sich die Sprache implementieren lässt, d.h. welcher Aufwand für die Übersetzung in eine Maschinensprache getrieben werden muss. In Kap.20 werden einige Sprachen anhand von Programmbeispielen vorgestellt.

Bei der Definition höherer Programmiersprachen ist es üblich, als Sprachelemente, die keine Bezeichner sind, englische Wörter oder mathematische Symbole zu verwenden. Anstelle des Kommentars zu Befehl 8 könnte beispielsweise `IF z<0 THEN GOTO ZYK ELSE` notiert werden, in Übereinstimmung mit der Syntax einiger gängiger Programmiersprachen.

Wir wollen nun das `PROGRAMM f2` von Bild 15.2b dahingehend erweitern, dass es von einem Computer ausgeführt werden kann, dessen Maschinensprache den Operationscode SIN nicht enthält. Die Sinusfunktion muss also ausprogrammiert werden. Für kleine Winkel x bietet sich die Reihenentwicklung (15.5) an[1].

$$\sin x = x - x^3/3! + x^5/5! - \ldots + (-1)^n * x^{(2n+1)}/(2n+1)! \ldots. \qquad (15.5)$$

2

Wie unschwer zu erkennen ist, ergibt sich das n-te Glied der Reihe aus dem vorangehenden durch dessen Multiplikation mit $-x^2/(2n(2n+1))$. Dieses Bildungsgesetz eignet sich für die Formulierung eines Programms zur iterativen Berechnung der Sinusfunktion. Bild 15.3 zeigt zwei Möglichkeiten. Beide Programme enhalten eine sog. **Laufanweisung** (die Zeilen zwischen `FOR` und `ENDFOR` bzw. zwischen `WHILE` und `ENDWHILE`), das linke Programm in Form einer **STEP-UNTIL-Anweisung**, das rechte in Form einer **WHILE-Anweisung**. Beide Programme sind unvollständig; u.a. sind die Deklarationen unterdrückt. Nur die Laufanweisungen mit ihren Anfangswertbelegungen sind vollständig wiedergegeben.

1 Für Winkel um 90° wäre eine Entwicklung für $\cos(90° - x)$ günstiger.

```
s := x;
y := x;


FOR n=2 STEP 1 UNTIL 20 DO
  s := -s*x^2/(4*n^2+2*n);
  y :=  y+s;

ENDFOR
```

(a)

```
s := x;
y := x;
n := 0;

WHILE ABS(s)>0,00001  DO
  n := n+1;
  s := -s*x^2/(4*n^2+2*n);
  y := y+s;
ENDWHILE
```

(b)

Bild 15.3 Programme zur Berechnung der Sinusfunktion gemäß (15.5), (a) - unter Verwendung einer STEP-UNTIL-Anweisung, (b) - unter Verwendung einer WHILE-Anweisung. Das hochgestellte Dach ist das Operationssymbol des Potenzierens.

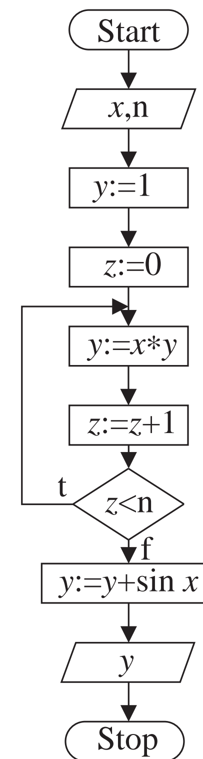
In der zweiten Zeile des linken Programms wird für die Variable `n`, *Laufvariable* genannt, ein Wertebereich von 1 bis 20 festgelegt, den die Laufvariable aufwärts in Einer-Schritten durchläuft. In jedem Schritt wird für den jeweiligen Wert von `n` der Wert des Summanden `s` (des laufenden Gliedes der Reihe) gemäß dem Bildungsgesetz der Reihe berechnet und zur bisherigen Teilsumme hinzuaddiert bzw. von ihr subtrahiert (in der Zeile vor `ENDFOR`). Die Iteration wird bei n=20 beendet. Die Summanden werden mit wachsendem n immer kleiner. Je mehr Glieder der Reihenentwicklung berücksichtigt werden, umso genauer fällt die Gesamtsumme mit dem exakten Wert der Sinusfunktion zusammen.

Oft ist es wünschenswert, das Abbruchkriterium der Iteration so festzulegen, dass der letzte Summand das Endergebnis nicht mehr als um eine vorgegebene Schranke verändert, sodass eine geforderte Ganauigkeit des berechneten Wertes gewährleistet ist. Das rechte Programm trägt diesem Wunsche Rechnung. Die Iteration wird fortgesetzt, solange ("while") das Prädikat `ABS(s) > 0.00001` erfüllt ist, solange also der Absolutwert des laufenden Summanden größer als 0,00001 ist. Sobald er kleiner oder gleich 0,00001 wird, bricht die Iteration ab.

Man beachte folgenden Unterschied zwischen den Abbruchprädikaten der beiden Programme von Bild 15.3. Im linken Programm ist die maximale Iterationszahl, bei der die Iteration abzubrechen ist, von vornherein festgelegt. Im rechten Programm ist das nicht der Fall. Vielmehr ergibt sich der Wert der Variablen im Abbruchprädikat, also der Wert von `ABS(s)`, aus dem Ergebnis der laufenden Iteration. Die Entscheidung, ob abgebrochen wird oder nicht, wird nach der in Kap.8.4.5 besprochenen Methode der *Minimalisierung* getroffen. Das Programm mit der WHILE-Anweisung definiert demzufolge eine μ-*rekursive* Funktion, während das Programm mit der STEP-UNTIL-Anweisung eine *primitiv rekursive* Funktion definiert.

Nach diesen Bemerkungen ist der Leser vielleicht bereits imstande, das Pascal-Programm von Bild 20.1 für die Berechnung der durch (15.1) definierten Funktion zu verstehen.

Mit dem Programm in Bild 15.2b ist zwar eine erheblich bessere Lesbarkeit erreicht im Vergleich zum Programm in Bild 15.2a, doch gibt die lineare Notationsweise als Folge von Anweisungen die Verzweigungsstruktur (die Rückkopplungschleife) nicht sehr anschaulich wieder. Das kann sich bei stark verzweigten Programmen recht unangenehm bemerkbar machen, insbesondere bei verschachtelten Rückkopplungsschleifen und Alternativmaschen. Um die Struktur leichter erkennbar zu machen, ist eine graphische (zweidimensionale) Darstellung sinnvoll. Dafür bietet sich der Aktionsfolgeplan oder der Programmablaufplan (PAP) an, dessen Symbole in Kap.13.5.4 erklärt worden sind (vgl. Bild 13.9). Bild 15.4 zeigt den PAP für das Programm von Bild 15.2b. Jedem rechteckigen Kästchen entspricht eine Aktion, also eine Ergibtanweisung in Bild 15.2b. Das rhombische Kästchen beschreibt den Sprungbefehl. Solange die aktuelle Iterationszahl z kleiner ist als die maximale Iterationszahl n, d.h. solange das Steuerprädikat im Rhombus erfüllt ist, erfolgt die Übergabe der Steuerung längs des mit t (true) beschrifteten Zweiges zurück zur nächsten Multiplikation. Sobald das Steuerprädikat nicht mehr erfüllt ist, erfolgt die Übergabe längs des mit f (false) beschrifteten Zweiges und der Zyklus wird verlassen.



15.4 Programmablaufplan (PAP) zum Programm von Bild 15.2

15.3 Näherungsverfahren

Aus unseren abstrakten Überlegungen über die Berechenbarkeit von Funktionen wissen wir zwar, dass sich für jede rekursive Funktion ein Maschinenprogramm des Von-Neumann-Rechners erstellen lässt, kurz, dass sie *von-Neumann-berechenbar* ist. Dennoch erhebt sich die Frage, wie die Berechnung in jedem Einzelfall möglich ist. Wir stellen die Frage konkreter. Die Sinusfunktion kann durch verschiedene Prädikate definiert werden, z.B. durch die Differenzialgleichung (4.2) oder durch die Faustregel “Gegenkathete durch Hypothenuse”. Wie lässt sich dieses Verhältnis mit Hilfe der Grundrechenarten rein digital aus dem Winkel, d.h. ohne Längenmessung berechnen? Die Antwort gibt die Reihenentwicklung (15.5).

Da alle rekursiven Funktionen von-Neumann-berechenbar sind, liegt es nahe anzunehmen, dass sich für alle Funktionen wenn nicht exakte, so doch stets Nä-

herungsformeln angeben lassen, die nur die Grundrechenarten enthalten. Dass die Vermutung stimmt, zeigt ein Blick in einschlägige Mathematikbücher, in denen die Reihenentwicklung von Funktionen behandelt wird, d.h. die Zerlegung von Funktionen in Summanden, die nur die Grundrechenarten enthalten und deren Absolutwerte monoton abnehmen. Da die Reihenentwicklung irgendwann abgebrochen werden muss, handelt es sich um eine Näherung. Das braucht uns nicht zu beunruhigen, denn aus Kap.4 wissen wir, dass Rundung eine immanente Notwendigkeit des digitalen Rechnens ist.

15.4 Assembler, Binder, Lader

Ein Programm, das man mit Mühe geschrieben hat, möchte man jederzeit zur Verfügung haben. Man möchte es bequem aufrufen und auf beliebige Eingabewerte anwenden können. Das ist durchaus möglich. Doch muss man sich dabei an die Anfangsadresse des Programms und an die Adressen aller Ein- und Ausgabewerte erinnern. Viel bequemer wäre es, wenn man die Variablenbezeichnungen unmittelbar verwenden könnte und wenn man das Programm über einen Namen, der sich leicht merken lässt, aufrufen könnte. Ein Programm, das über seinen Namen gerufen werden kann, wird **Prozedur** genannt.

Beide Wünsche erfüllt ein Programm, das die vom Programmierer gewählten **Bezeichner** (die Namen von Programmen oder Variablen) durch geeignete Adressen ersetzt. Variablenbezeichner werden auch **Identifikatoren** genannt. Das Ersetzen von Identifikatoren durch Adressen heißt **Assemblieren** und ein assemblierendes Programm heißt **Assembler**. Ein Assembler kann auch dafür eingesetzt werden, geeignet gewählte Operationcodes, also Bezeichner für die Maschinenoperationen, in Interncodes zu übersetzen. Das Assemblieren stellt eine sehr einfache Art des Übersetzens dar und lässt sich ohne besondere Schwierigkeit automatisieren, d.h. programmieren. Ein Übersetzerprogramm überführt einen Programmtext in einen neuen, *äquivalenten* Text, d.h. in einen Text mit gleicher interner Semantik.

Unabhängig davon, aus welcher Sprache in welche Sprache ein Programm übersetzt wird, ist es üblich, das ursprüngliche Programm als **Quellprogramm** oder **Quellcode** und das übersetzte Programm als **Zielprogramm** oder **Zielcode** zu bezeichnen. Dementsprechend heißen die verwendeten Sprachen **Quellsprache** und **Zielsprache**. Im Falle des Assemblers wird das Quellprogramm **Assemblerprogramm** genannt. Es enthält Identifikatoren (Variablenbezeichnern) und Operationscodes (Operationsbezeichner). Da es nur Operationscodes für diejenigen Operationen enthalten darf, die von der "Maschine" (vom Prozessor) angeboten werden, bezeichnen wir Assemblerprogramme ebenfalls als Maschinenprogramme und unterscheiden zwischen *höheren* und *internen Maschinenprogrammen*. Ein internes Maschinenprogramm enthält nur Adressen und intern codierte Operationscodes. Es ist "lauffähig", d.h. es kann ohne weitere Bearbeitung in den Hauptspeicher "gela-

den" und abgearbeitet werden. Darum wird es auch *ladbares Maschinenprogramm* oder kurz **ladbares Programm** genannt. Auch die Quell- und Zielsprache des Assemblers bezeichnen wir beide als Maschinensprachen und unterscheiden zwischen **höherer** und **interner Maschinensprache**.

Wir wollen uns überlegen, wie der Assembler seine Aufgabe lösen kann. Offenbar muss er zwei Arbeitsgänge ausführen, *Phasen* genannt. In der ersten Phase muss er alle Bezeichner erkennen und "sammeln" ("assemblieren"). Dazu muss er aus dem Assemblerprogrammtext diejenigen Bitketten heraussuchen, die keine Lexeme der internen Maschinensprache sind. Als **Lexeme** werden die kleinsten Einheiten einer Programmiersprache bezeichnet, die interne Semantik tragen.[2] Die Suchprozedur wird **lexikale Analyse** genannt. Sie kann dadurch vereinfacht werden, dass im Assemblerprogramm die verwendeten Variablenbezeichner in einem sog. *Deklarationsteil* aufgelistet, man sagt *deklariert* werden. Die Syntaxregeln vieler Sprachen schreiben vor, dass ein Programm mit einem Deklarationsteil beginnt. 3

Wenn der Assembler einen neuen Variablenbezeichner erkannt hat, legt er für ihn in der sog. **Symboltabelle** eine Zeile an, in welche später die Attribute (Merkmale) der bezeichneten Variablen eingetragen werden wie Speicherplatzbedarf und Adresse. Der Bezeichner eines Programms ist zweckmäßigerweise durch ein geeignetes vorgestelltes Codewort, z.B. `PROGRAM` oder `PROG`, zu kennzeichnen und an die Spitze des Programms zu setzen. Die Syntax eines Programms könnte in der *Backus-Naur-Form* [5.3] folgendermaßen festgelegt werden, wobei die geschweiften Klammern anzeigen, dass ein Programm beliebig viele Befehle enthalten kann:

Programm → `PROG` *Programmbezeichner Deklarationsteil* {*Befehl*} `ENDE` (15.6)

Um die vollständige Sprachsyntax zu definieren, muss die Syntax jeder der drei kursiv gedruckten Programmglieder - wir hatten sie *metasprachliche Variable* genannt - festgelegt werden. Für "*Befehl*" könnte das im Falle einer Dreiadressmaschine durch die Syntaxregel (5.2) (Kap.5.3) geschehen. Es ergibt sich eine hierarchische Syntaxdefinition, die sich als Syntaxbaum darstellen lässt, ähnlich wie für natürliche Sprachen (vgl.Bild 5.2).

Der Leser führe sich noch einmal Folgendes vor Augen. Nach dem Start eines Programms "liest" der Prozessor "Wort für Wort" den Speicherinhalt, beginnend bei der Anfangsadresse. Sobald er `ENDE` erkennt, bricht er ab. "Lesen" bedeutet "Kopieren in das Befehlsregister (BR)", und "Wort für Wort" bedeutet Adresse für Adresse oder Befehl für Befehl. Das setzt voraus, dass ein Befehl sowohl in eine Speicherzelle als auch in das Befehlsregister hineinpasst. (Dies ist keine prinzipielle Forderung an die Schnittstelle zwischen Hard- und Software.)

2 In diesem Zusammenhang wird häufig auch die Bezeichnung *Morphem* verwendet.

Der Aufbau des Befehlsregisters bestimmt das Format des Befehls und legt damit die Lexemgrenzen fest. Demzufolge sind Leerzeichen oder andere Trennzeichen in einem ladbaren Programm überflüssig. In einem Assemblerprogramm sind sie notwendig, falls Bezeichner unterschiedlicher Länge zugelassen sind. Die notwendige Formatanpassung der Software an die Hardware (der Befehle an das Befehlsregister) erfolgt automatisch bei der Adresszuweisung. Das erleichtert die Arbeit des Assemblers.

Die Einfachheit des Assemblers hat noch einen weiteren Grund, der in der Syntaxdefinition liegt. Die Syntax ist so definiert, dass sich eine syntaktische Analyse [5.4] [16.13] erübrigt. Denn für jedes Lexem ist die Lexemklasse (die metasprachliche Variable) durch das Programm- und Befehlsformat eindeutig festgelegt. Es erübrigt sich also auch der Aufbau eines Syntaxbaumes, wie wir ihn in Kap.5.3 für Aussagesätze der deutschen Sprache durchgeführt hatten. Das ändert sich bei höheren Programmiersprachen, wo die Syntaxanalyse eventuell recht aufwendig werden kann.

In der zweiten Phase muss der Assembler das Quellprogramm noch einmal durchgehen und dabei die Bezeichner durch *relative Adressen* (relativ zur Anfangsadresse des Programms) ersetzen, die er den Bezeichnern zuweist. Damit ist die Assemblierung abgeschlossen. Das assemblierte Programm ist ein *verschiebbares* (als Ganzes im Speicher durch Änderung der Anfangsadresse verschiebbares), in einen beliebigen freien Speicherbereich ladbares Programm.

Ein Computer verfügt gewöhnlich über eine große Zahl ladbarer Programme, die in einem peripheren Speicher, z.B. auf einer Festplatte gespeichert sind. Bevor ein ladbares Programm geladen und gestartet werden kann, muss es "gebunden" werden. Das **Binden** besteht i.Allg. aus zwei Schritten. Falls das Programm ein anderes ladbares Programm als Unterprogramm ruft, müssen zunächst die relativen Adressen des Unterprogramms an die relativen Adressen des Hauptprogramm *gebunden* werden. Das erledigt ein Programm namens **Binder**[3]. In diesem ersten Schritt des Bindens wird u.a. die sog. *Parameterübergabe* (Operandenübergabe) vom Hauptprogramm an das Unterprogramm (von der rufenden an die gerufene Prozedur) realisiert. Im zweiten Schritt werden alle relativen Adressen in absolute Adressen umgerechnet, d.h. die Befehle und Variablen des Programms werden an Speicherplätze des Hauptspeichers gebunden. Das erledigt ein Programm namens **Lader**. Der Lader lädt außerdem das Programm in den Hauptspeicher.

Das Binden vor dem Laden muss nicht unbedingt sämtliche Befehle und Variablen eines ladbaren Programms betreffen. In den Kapitel 19 und 20 werden wir Situationen kennen lernen, die ein Binden während der Laufzeit zweckmäßig oder sogar notwen-

3 Anstelle der Wörter *binden* und *Binder* werden unter Informatikern häufiger die eingedeutschten Wörter *linken* und *Linker* verwendet.

dig machen. *Binden vor dem Programmstart wird **statisches Binden**, Binden während der Laufzeit wird **dynamisches Binden** genannt.* 5

15.5 Semantische Lücke

Die Einführung des Assemblers hat bedeutsame Konsequenzen. Sie stellt den ersten, wenn auch zaghaften Schritt zur Lösung eines Grundproblems der Programmierungstechnik und der künstlichen Intelligenz dar, das aus dem Alltag als Sprachbarriere bekannt ist.

Als *Sprachbarriere* wird die Schwierigkeit bezeichnet, mit der jeder konfrontiert ist, der sich mit einem Menschen unterhält, der "eine andere Sprache spricht". Das kann eine Fremdsprache oder auch eine Fachsprache sein, die man nicht kennt. Es kann auch die Sprache einer anderen Kultur, einer anderen Weltanschauung oder einfach die Ausdrucksweise einer völlig anderen Lebensart sein.

Die Aufzählung zeigt, dass die Wurzel der Sprachbarriere entweder in der unterschiedlichen *Artikulierung* von Idemen liegt, also in der Benutzung unterschiedlicher Ausdrucksweisen bzw. Sprachen, oder in einem - möglicherweise sehr tiefliegenden - Unterschied der Ideme der Gesprächspartner, also der Bewusstseinsinhalte, mit denen ihr Denken operiert. Es liegt eine Inkompatibilität der den benutzen Zeichenrealemen zugeordneten Bedeutung (Semantik) vor. Wir nennen sie **semantische Lücke**. Diesen Begriff hatten wir bereits in Kap.5.4 [5.8] im Zusammenhang mit den drei Semantiken eingeführt, ohne näher auf ihn einzugehen.

In Kap.5.4 [5.5] hatten wir vereinbart, die Bedeutung (das Idem) eines Kompositzeichens immer dann als dessen Semantik zu bezeichnen, wenn man annehmen kann, dass sie für alle Beteiligten die gleiche oder zumindest ausreichend ähnlich ist. Wenn das nicht der Fall ist, liegt eine semantische Lücke vor. Sie kann das gegenseitige Verständnis zweier Gesprächspartner unmöglich machen.

Ersetzen wir den einen der Gesprächspartner durch einen Computer, so scheint die Überwindung der semantischen Lücke ein hoffnungsloses Unterfangen zu sein, denn die Lücke ist eher ein Abgrund, der zwischen zwei radikal unterschiedlichen Semantiken klafft, der *externen* Semantik des Menschen und der *internen* Semantik des Computers (Bild 5.3). Dennoch muss eine Brücke über den Abgrund geschlagen werden, sonst könnten Mensch und Computer sich nicht "verstehen", und der Computer könnte dem Menschen nicht helfen.

In Kap.8.6 hatten wir vereinbart, die Semantik, in welcher der Nutzer eines Computers denkt, **Nutzersemantik** zu nennen. Demgegenüber werden wir die Semantik, "in welcher der Computer denkt", **Computersemantik** nennen oder auch **Maschinensemantik**, wenn unterstrichen werden soll, dass die Semantik der Maschinensprache gemeint ist. Man könnte dann auch von der *Semantik des Maschinenkalküls* sprechen. Damit lässt sich das Problem der Schließung der semantischen Lücke prägnanter formulieren: *Die **Nutzersemantik** ist partiell* (soweit es erforder- 6

lich ist) *an die **Maschinensemantik** zu binden*. Dies Problem hatten wir in Kap.5.4 das *technische Semantikproblem* genannt. Das "*semantische Anbinden*" besteht darin, dass an bestimmte Zustände oder Vorgänge im Computer Externsemantik "angebunden" wird, dass ihnen Ideme zugeordnet werden. Das Wort "partiell" bringt zum Ausdruck, dass die externe Semantik nur zum Teil, und zwar i.Allg. zu einem sehr geringen Teil an Hardwarezustände gebunden wird. Der größte Teil der "Kontextsemantik", mit der das Gehirn hantiert (des aktivierten Idemkomplexes) bleibt außerhalb der semantischen Bindung, besitzt also keine Entsprechung im Computer (siehe dazu Kap.17.1).

Damit externe Semantik an Maschinensemantik gebunden werden kann, muss sie durch *Kalkülisierung* in *formale* Semantik überführt werden. Davon war in Kap.8.6 die Rede. Der Nutzer muss also parallel in externer und in formaler Semantik denken. Der Begriff der Nutzersemantik schließt externe und formale Semantik ein. Das gleiche gilt für die Semantik, in der ein angewandter Mathematiker denkt, der z.B. ein Statikproblem in mathematische Formeln fasst.

Es ist sicher nicht übertrieben, das technische Semantikproblem als das Kernproblem des Programmierens, des Sprachentwurfs, der künstlichen Intelligenz und schließlich auch der weltweiten Diskussion um die künstliche Intelligenz und um die Informatik überhaupt zu bezeichnen. Ingenieure, Naturwissenschaftler, Linguisten und Philosophen haben sich mit diesem Problem beschäftigt und werden es auch in Zukunft tun.

Auch wir werden uns noch lange und intensiv mit dem Semantikproblem und seiner Überwindung beschäftigen müssen (siehe Kap.18.3). Doch schon jetzt lässt sich Folgendes sagen. Der Assembler erleichtert das Anbinden der Nutzersemantik an die Maschinensemantik dadurch, dass er dem Nutzer erlaubt, Variable zu verwenden und für sie mnemotechnisch geeignete Bezeichner zu benutzen. Im nächsten Kapitel werden wir das semantische Binden im Falle des numerischen Rechnens im Detail untersuchen.

Zuvor wollen wir versuchen noch deutlicher zu verstehen, warum das Semantikproblem so schwierig ist und wo seine Wurzeln liegen. Dazu knüpfen wir noch einmal an die Vorstellung zweier Gesprächspartner an, von denen einer durch einen Computer ersetzt ist. Diese Vorstellung darf auf keinen Fall zu der Annahme verleiten, der Computer "verstehe" seinen Partner in der umgangssprachlichen Bedeutung, d.h. in der auf die Umgangssprache bezogenen Bedeutung des Wortes "verstehen". Er kann ihn nicht verstehen, weil ihm nur ein ganz geringer Teil der Semantik zur Verfügung steht, in der sein Partner denkt. Der gesamte Kontext, innerhalb dessen der Mensch dem Computer etwas mitteilt, existiert für den Computer nicht und bleibt von der Kommunikation ausgeschlossen.

Der Kontext eines Wortes oder Satzes kann enorm groß sein, so z.B., wenn von den newtonschen Axiomen oder von Beethovens Neunter die Rede ist. Verstehen setzt dann ein entsprechend großes Wissen beim Hörer (Interpretierer) voraus. Der Computer hat ein solches Wissen nicht, denn er verfügt über keine externe Semantik,

er “weiß” nichts von der Welt, zumindest nicht auf dem gegenwärtigen Stand des Computerwissens.

Diese Behauptung kann bei demjenigen Widerspruch provozieren, dem bekannt ist, dass der Computer zur “Wissensverarbeitung” eingesetzt wird. Es fragt sich, wie er Wissen über die Welt verarbeiten kann, ohne über externe Semantik zu verfügen. Um die Frage zu beantworten, muss zunächst der Wissensbegriff definiert werden. Primär handelt es sich um einen personenbezogenen Begriff. *Das **Wissen** eines Menschen ist die Gesamtheit aller Aussagen, an deren Wahrheit er nicht zweifelt. Das Wissen einer Gruppe von Menschen ist das gemeinsame, objektivierte individuelle Wissen der Gruppenmitglieder*.

Wissen ist also eine Menge von Aussagen. Nach unseren grundsätzlichen Überlegungen über den Begriff der Information in den Kapiteln 1 und 2 liegt damit die Antwort auf die gestellte Frage auf der Hand. Mitgeteiltes Wissen ist Information, besteht also aus Realemen und Idemen. Physisch übertragen und im Computer verarbeitet werden nur die Realeme. Das Gehirn fügt ihnen Ideme (Bewusstseinsinhalte, externe Semantik) hinzu. Der Computer leistet *Realemverarbeitung*, genau genommen also keine Wissensverarbeitung oder Informationsverarbeitung, wenn man, wie in diesem Buch, unter Information - und entsprechend unter Wissen - die Gesamtheit von Realem und Idem versteht. Nichtsdestoweniger kann der Computer durchaus den Eindruck erwecken, als denke er in externer Semantik, in den Idemen seines Nutzers. Dieser Eindruck kann mit einem einfachen Trick hergerufen werden, was an einem kleinen Beispiel demonstriert werden soll.

Angenommen, ein Physiker lässt sich bei der Auswertung seiner Formeln (beim numerischen Modellieren) vom Computer helfen. Er hat ein Programm zur Berechnung der Fallgeschwindigkeit einer Kugel in Flüssigkeiten geschrieben. Weil er vergesslich ist, möchte er sich die Resultate so kommentiert ausgeben lassen, dass er ihre Bedeutung auch noch morgen und in einem Monat sofort versteht. Eine Möglichkeit ist die Ausgabe in Aussagesätzen, beispielsweise “Die maximale Geschwindigkeit der fallenden Kugel beträgt in Wasser 87 cm pro Sekunde”. Ein solcher Satz kann tatsächlich den Anschein erwecken, als arbeite der Computer mit externer Semantik, was natürlich eine Illusion ist. Die Illusion lässt sich durch eine sehr einfache *Zeichenkettenoperation* hervorbringen. Die Operation besteht darin, dass der Computer in eine vorgegebene Zeichenkette an einer bestimmten Stelle eine aktuelle Zeichenkette (z.B. 87) einfügt. Dies ist ein sehr einfaches Beispiel von **Textverarbeitung**. Noch primitiver ist das Täuschungsmanöver, wenn der Computer nach dem Einschalten seinen Partner mit “Hallo” begrüßt.

15.6 Numerisches Rechnen

Für den programmierenden Computeranwender besteht der Zwang zum semantischen Binden konkret darin, dass er seine Gedanken, genauer seine i.d.R. umgangs-

sprachlich artikulierten Wünsche an den Computer in einer Programmiersprache artikulieren muss, kurz, er muss eine Sprache an eine andere semantisch binden. Das ist oft sehr mühevoll, wenn nicht gar unmöglich. Es gibt jedoch einen Anwendungsbereich, in dem Denken und Programmieren (Algorithmieren) so nahe beieinander liegen, dass sich die semantische Bindung relativ mühelos erreichen lässt, nämlich dann, wenn der Partner (Nutzer) des Computers mathematisch denkt, beispielsweise ein Naturwissenschaftler, der mathematische Modelle der Welt aufstellt und sich dabei vom Rechner helfen lassen will.

Mathematisches Modellieren erfolgt auf zwei Ebenen, auf einer analytischen und einer numerischen. Auf der analytischen Ebene werden Gleichungen aufgestellt und umgeformt, z.B. die Gleichung der Erdbewegung um die Sonne. Dabei wird mit Variablen und Formeln hantiert, der Informatiker spricht von *Formelmanipulation*. Auf der numerischen Ebene werden die Gleichungen numerisch ausgewertet. Dabei werden arithmetische Operationen ausgeführt, d.h es wird *mit Zahlen gerechnet*. Zuweilen wird das Manipulieren mit Variablen (Bezeichnern, Symbolen) *symbolische* Informationsverarbeitung und das Rechnen mit Zahlen *Kalkulation* genannt. Diese Sprechweise werden wir *nicht* übernehmen, sondern vereinbaren:

8 Das Rechnen mit Bezeichnern für Variablen oder Funktionen heißt **analytisches Rechnen**. Das Rechnen mit Werten (Konstanten oder Werten von Variablen) heißt **numerisches Rechnen**. Zahlen sind spezielle Bezeichner von Werten. Analytisches Rechen kann numerisches Rechnen enthalten.

Wir haben die Begriffe des analytischen und numerischen Rechnens bewusst so definiert, dass mit ihnen alle Arten des mathematischen Operierens erfasst sind. Wenn sich zeigen lässt, dass der Computer zur Ausführung jeder numerischen und jeder analytischen Rechnung befähigt werden kann, so folgt daraus, dass er zur Ausführung *jeder* mathematischen Operation befähigt werden kann.

Die Bedeutung der so eingeführten Begriffe fällt nicht in jedem Falle mit der Bedeutung zusammen, in der sie zuweilen in der Literatur verwendet werden. So wird unter numerischem Rechnen häufig ausschließlich das Rechnen mit Zahlen verstanden. Wir dagegen verwenden den Begriff beispielsweise auch dann, wenn in der Prädikatenlogik mit Individuenkonstanten gerechnet wird oder in der booleschen Algebra mit booleschen Konstanten. Die Berechnung des Wertes von $1 \wedge 0$ ist nach obiger Definition eine numerische Rechnung, eine Umformung nach der morganschen Regel hingegen eine analytische Rechnung. Auch das Differenzieren, das Integrieren und das Lösen von Differenzialgleichungen ist analytisches Rechnen. Man beachte, dass das analytische Rechnen mehr umfasst als das Rechnen im Rahmen der Analysis, wenn unter Analysis, wie üblich, dasjenige Gebiet der Mathematik verstanden wird, das sich mit Funktionen auf der Grundlage der Infinitesimalrechnung beschäftigt, also u.a. mit dem Differenzieren, dem Integrieren und dem Lösen von Differenzialgleichungen. Die Bezeichner, mit denen beim analytischen Rechnen gerechnet wird, können auch Wahrscheinlichkeiten, Wahrscheinlichkeitsverteilungen oder Funktionen von Wahrscheinlichkeitsverteilungen benennen. Vor-

aussetzung, dass mit einem Bezeichner gerechnet werden kann, ist, dass er Element einer Kalkülsprache ist.

Je nachdem, ob mathematisches Modellieren auf analytischem oder auf numerischem Rechnen beruht, m.a.W. ob es auf analytischer oder numerischer Ebene erfolgt, sprechen wir von **analytischer** bzw. **numerischer Modellierung**. Numerische Modellierung, die ein Rechner ausführt, wird häufig Computersimulation genannt. Das ist besonders dann üblich, wenn mit dem numerischen Modell "experimentiert" wird. Wir werden im Weiteren unter **Computersimulation** ganz allgemein das *"Nachmachen" irgendwelcher Prozesse oder Handlungen durch den Computer* verstehen, das "Nachmachen" von Denkprozessen eingeschlossen.

Charakteristisch für die Simulation ist, dass der Computernutzer (der Experimentator) mit Bezeichnern für variierbare Parameter arbeitet und durch Zuweisung verschiedener Werte an die Parameter mit dem Simulationsmodell experimentiert. Wenn beispielsweise das Fallen einer Kugel in einer Flüssigkeit simuliert wird, können Durchmesser und spezifisches Gewicht der Kugel und die Viskosität der Flüssigkeit variierbare Parameter sein.

Wir wollen uns überlegen, wie ein Physiker vorzugehen hat, der sich beim numerischen Modellieren helfen lassen will, der z.B. das Simulationsexperiment mit der Kugel ausführen will. Dabei wollen wir davon ausgehen, dass unser Experimentator über ein analytisches Modell der Kugelbewegung verfügt. Zuerst muss er diesem Modell die Form berechenbarer Funktionen geben, genauer die Form arithmetischer Ergibtanweisungen. Dabei wird er sich in der Regel geeigneter Näherungsformeln bedienen müssen. Sodann muss er allen Variablen (Parametern und Resultaten) geeignete Bezeichner zuordnen und schließlich ein Programm schreiben, d.h. die Bezeichner deklarieren und die Funktionen in Befehlsfolgen überführen.

Der Tätigkeit des Programmierens entspricht beim physikalischen Experiment das Aufbauen einer Versuchsapparatur. Diese besteht im Fallbeispiel aus einem Gefäß mit Flüssigkeit und Geräten zum Messen von Längen und Zeitintervallen. Das physikalische Experiment beginnt mit dem Fallenlassen der Kugel, das Simulationsexperiment mit dem Programmstart. Die Resultate werden von der physikalischen Apparatur zunächst in analoger Form geliefert und dann - durch den Experimentator oder durch die Apparatur - in die digitale Form konvertiert. Der Computer liefert sie unmittelbar in digitaler Form.

Die Resultate des Rechners muss der Experimentator *interpretieren*, d.h. er muss ihnen seine eigene Semantik, die *Nutzersemantik* zuordnen. Das kann er, auch wenn das Resultat nicht in Form eines Satzes ausgedruckt wird, denn er weiß, welche Bezeichner was bedeuten. Der Rechner weiß das nicht. Er kennt die Nutzersemantik nicht, sondern nur seine interne Semantik, die *Computersemantik*, d.h. er "weiß" in jedem Augenblick nur, was mit der gerade eingelesenen Zeichenkette zu geschehen hat. Der Computer kennt *nicht* die Bedeutung, die der Nutzer mit den Zeichenketten verbindet, er *operiert **nicht** mit **Bedeutungen*** (vgl. das *Bedeutungsprinzip* [Einleitung.2]).

Die physikalische Apparatur arbeitet nicht mit Zeichen, also auch nicht mit Semantik. Die Apparatur und ihr Verhalten treten unmittelbar als Ideme von Urrealemen in das Bewusstsein des Beobachters. Semantische Bindung erübrigt sich. Dennoch kann auch hier eine Art Interpretieren notwendig sein, nämlich dann, wenn der Experimentator sich im Grunde gar nicht für das Verhalten der Apparatur, sondern für ein ganz anderes Phänomen interessiert, das er mit Hilfe der Apparatur modelliert und zwar analog modelliert. Dabei kann das Original völlig anderer Natur sein. Das "Interpretieren" besteht dann darin, dass der Experimentator physikalische Größen des Modells als physikalische Größen des Originals deutet.

Eine besondere Art von Versuchsapparatur hatten wir in Kap.4.2 kennen gelernt, den Analogrechner. Bei einem Analogrechnerexperiment besteht das "Interpretieren" darin, dass elektrische Größen des Analogrechners als Größen des Originals gedeutet werden, z.B. die Ausgangsspannung als Fallgeschwindigkeit der Kugel, wenn das Fallexperiment mit einem Analogrechner modelliert wird. Diese Art des Interpretierens beinhaltet nicht das "Anbinden" von Bedeutungsinhalten an Zeichen, von Idemen an Realeme, ist also etwas anderes als das, was in Kap.2 als Interpretieren definiert und in Bild 2.1 als Pfeil 5 dargestellt ist. Es beinhaltet dementsprechend auch nicht das Binden interner an externe Semantik, sondern externer an externe Semantik.

Voraussetzung des *analogen* (physikalischen) Modellierens ist, dass das Verhalten von Original und Modell durch dieselben Gleichungen beschrieben wird. Voraussetzung des *digitalen* Modellierens (des Simulierens) ist dagegen, dass die Gleichungen, die das Original beschreiben, implementiert sind. Das Arbeiten mit Gleichungen setzt in jedem Falle die Bindung externer an formale Semantik voraus, also das Interpretieren eines Kalküls (vgl.Kap.5.4 [5.11]).

Nach diesem kleinen Ausflug in das analoge, d.h. nichtsprachliche Modellieren wenden wir uns der eingangs aufgestellten Behauptung zu, wonach semantisches Binden im Falle mathematischer Modellierung relativ einfach ist. Zunächst ist festzustellen, dass ein Maschinenprogramm Operatoren (Operationscodes), Operanden (Variablenbezeichner) und eventuell Weichen (bedingte Sprungbefehle) enthält und dass diese drei Sprachelemente ihre expliziten oder impliziten Entsprechungen sowohl im mathematischen Modell, das als Maschinenprogramm implementiert ist, als auch im Original besitzen. Wir werden das semantische Binden für die drei genannten Sprachelemente getrennt behandeln und beginnen mit der Frage: Wie erfolgt das *semantische Binden der Operatoren* beim numerischen Modellieren?

Die Antwort liegt auf der Hand. Das semantische Binden von Operatoren erfolgt dadurch, dass die arithmetischen Operatoren, mit denen der Programmierer gedanklich arbeitet, im Prozessor implementiert und über Operationscodes aufrufbar sind, die mehr oder weniger genau dem mathematischen Sprachgebrauch entsprechen. Das hat zur Folge, dass das Programmieren zu einem rein syntaktischen Transformieren wird, das lediglich die Substitution von Zeichenketten beinhaltet, beispielsweise der Zeichenkette a+b durch `ADD a b` oder `(+ a b)`.

Daraus darf allerdings nicht der allgemeine Schluss gezogen werden, die Semantik einer Sprache sei durch ihre Syntax gegeben. Das wäre sicher zu kurz gegriffen. Denn beim Simulieren und bei jeder Verhaltensmodellierung mittels Computer müssen Prozesse, die in dem zu modellierenden Original ablaufen, in Prozesse, die im Computer ablaufen, überführt werden. Wenn das auf rein syntaktische Transformationen zurückgeführt werden kann, ist das sicher die Ausnahme.

Kommen wir nun zur *semantischen Bindung der Operanden*, d.h. der Variablen in den arithmetischen Ausdrücken, aus denen das mathematische Modell besteht. Auch sie macht keine Schwierigkeiten, und wir kennen das Vorgehen bereits. Sie besteht aus zwei Schritten, dem Binden externer Größen (z.B. der Geschwindigkeit der fallenden Kugel aus obigem Beispiel) an Variablen arithmetischer Ausdrücke (das ist die Umkehrung der Interpretation eines Kalküls) und dem Binden der arithmetischen Variablen an Speicherplätze durch den Assembler. Es ist zu beachten, dass Variablenbezeichner vor der Programmausführung durch Zahlen ersetzt werden. Es wird also mit Zahlen gerechnet und nicht mit Variablen, m.a.W. analytisches Modellieren ist auf diese Weise nicht möglich. Wie es möglich wird, werden wir uns in Kap.15.8 überlegen.

Dem dritten Sprachelement, dem *bedingten Sprungbefehl*, entspricht nutzersemantisch das Denken in Alternativen und in Wenn-dann-Konstruktionen. Am Beispiel der Zyklusorganisation in Bild 15.2 haben wir gesehen, dass es ziemlich schwierig sein kann, die "Bedeutung" eines Sprungbefehls zu erkennen, ihn semantisch an das Denken zu binden. Wir haben das Problem dadurch gelöst, dass wir Sprachelemente eingeführt haben, die dem menschlichen Denken und Sprachgebrauch näher liegen. Dies sind die bedingte Sprunganweisung und im Falle von Iterationen die Laufanweisung (Bild 15.3). Damit haben wir aber bereits die Grenze zwischen Maschinensprachen und höheren Programmiersprachen überschritten. Man beachte, dass der Programmierer, der seine Wenn-dann-Artikulierungen in Sprungbefehle überführt, im Endeffekt wiederum nur Zeichenketten substituiert, also syntaktische Transformationen durchführt. Es müsste also möglich sein, diese Operation dem Computer zu übertragen, d.h. ein geeignetes Übersetzerprogramm zu schreiben. Auf diesen Gedanken werden wir in Kap.16.4 zurückkommen. Bis dahin gehen wir von der Annahme aus, dass für jedes Nutzerprogramm, das in einer beliebigen formalen Sprache geschrieben sein darf, ein Übersetzerprogramm erstellt werden kann, welches das Nutzerprogramm in ein äquivalentes Maschinenprogramm übersetzt.

15.7 Programmierung von Operatorenhierarchien

Mit der Möglichkeit, Variablenbezeichner zu verwenden, sind die Vorteile, die der Assembler dem Nutzer bietet, nicht erschöpft. Neben den Bezeichnern von Variablen "versteht" der Assembler auch Bezeichner (Namen) von Programmen, d.h.

er erkennt einen Programmnamen und ersetzt ihn durch die Anfangsadresse des betreffenden Programms. Wenn er außerdem einen Sprungbefehl zu dieser Adresse einfügt, sodass sie in den Befehlszähler geladen wird, reagiert der Prozessor auf den Programmnamen genauso wie auf einen Operationscode eines Programms, das im ROM des Matrixsteuerwerks als Firmware abgespeichert ist (vgl. Kap.13.5.5). Diese geringfügige Erweiterung des Assemblers erlaubt dem Programmierer, ein Programm, das er selber erstellt, benannt und im Hauptspeicher abgelegt hat, ebenso zu benutzen wie die Programme der Firmware, d.h. er darf den Programmbezeichner beim Schreiben weiterer Programme praktisch in der gleichen Weise verwenden, wie die Operationscodes der Maschinensprache.

10 Damit ist die Möglichkeit gegeben, Operatorennetze und Operatorenhierarchien softwaremäßig zu komponieren. Dem schrittweisen Komponieren von Operatoren entspricht ein verschachteltes Aufrufen von Prozeduren (Unterprogrammen). Auf diese Weise lassen sich selbst sehr große Programme effektiv - evtl. in Arbeitsteilung durch ein Programmierteam - und übersichtlich programmieren, sodass sie machbar und lesbar werden. Ein Programm ist gut lesbar, wenn seine Funktionsweise, d.h. wenn seine *interne* Semantik leicht zu erkennen ist. Für jeden, der sich in einem Programm zurechtfinden muss, also auch für denjenigen, der es im praktischen Einsatz wartet, ist seine Lesbarkeit von großer Bedeutung.

Für den Anwender eines Programms hingegen ist nicht die *interne*, sondern die dem Programm zuzuordnende *externe* Semantik für das Arbeiten mit dem Programm ausschlaggebend. Dem Anwender kommt es weniger auf die *Lesbarkeit*, als vielmehr auf die *Verstehbarkeit* des Programms an. Ein Beispiel soll den Unterschied zwischen Lesbarkeit und Verstehbarkeit illustrieren.

Angenommen, ein Unternehmer hat ein Programm in Auftrag gegeben, das seinen Betrieb modelliert. Es soll ihm helfen, die Produktion besser zu organisieren. Um das Programm sinnvoll einsetzen zu können, muss er die dem Programm zugeordnete *externe* Semantik *verstehen*. Er muss die Entsprechungen zwischen seinem gedanklichen, eventuell textlich und graphisch niedergelegten Modell des Betriebes und den Berechnungen, die das Programm ausführt, und den Bezeichnungen, die in den Ein- und Ausgaben des Computers auftreten, kennen. Das muss ihm vom Programmierer erklärt werden. Er muss aber nicht das Programm *lesen* (interpretieren) können, die *interne* Semantik braucht ihm nicht erklärt zu werden. Etwas verkürzt können wir zusammenfassend sagen: *Die **Lesbarkeit** eines Programms betrifft die **interne**, die **Verstehbarkeit** die **externe** Semantik.*

Gute Lesbarkeit ist weitgehend eine Frage des *Programmierstils*. Der Stil eines Programmierers ist - ähnlich wie der Stil eines Schriftstellers - die typische Art und Weise, sich in einer bestimmten Sprache auszudrücken. Er ist das Ergebnis von Gewohnheit, gegebenenfalls auch von Übereinkünften innerhalb eines Teams, aber auch von Geschmack und Mode.

Der besseren Lesbarkeit halber sollten kurze, in sich abgeschlossene Programmbausteine angestrebt und komplizierte Abhängigkeiten zwischen den Bausteinen,

d.h. komplizierte Operandenflüsse vermieden werden. Besonders beeinträchtigt wird die Lesbarkeit durch Überlappungen von Maschen und/oder Schleifen. In Kap.8.4.5 war anhand des Bildes 8.10 gezeigt worden, dass Überlappungen immer eliminiert werden können. Dort hatten wir Kompositoperatoren ohne Überlappungen wohlstrukturiert genannt. Die Erstellung wohlstrukturierter Programme ist eine Grundforderung der sogenannten *strukturierten Programmierung*.

Durch eine gute Programmstruktur kann nicht nur die Lesbarkeit, sondern auch die Verstehbarkeit eines Programms erhöht werden. Dazu muss sie die Struktur des Originals widerspiegeln. Das soll am obigen Beispiel der Betriebsmodellierung demonstriert werden. Wir gehen davon aus, dass der Unternehmer seinen Betrieb hierarchisch organisiert hat und dass die Funktionsweise der Hierarchie und ihrer Bausteine algorithmisch beschreibbar ist. Wenn nun das modellierende (simulierende) Programm eine Operatorenhierarchie darstellt, die der Hierarchie des Betriebes entspricht, wird der Unternehmer die für den Nutzer sichtbare Funktionsweise des Programms schnell erfassen. Wenn zudem die Programmbezeichner mit den Bezeichnungen der Betriebsteile, der Bereiche und Abteilungen der Produktion und der Verwaltung übereinstimmen bzw. an sie erinnern, werden Unternehmer und Angestellte die Ein- und Ausgaben des Programms sehr bald verstehen und sich mit der Anwendung des Programms anfreunden und den Computer als *bequemen* Helfer akzeptieren. Die Bezeichnungen müssen sich gewissermaßen selber "dokumentieren". 11

Das reicht allerdings nicht aus, um ein Programm leicht verstehbar zu machen; dazu muss es ausführlich dokumentiert werden, am besten dadurch, dass in den Programmtext Kommentare für den Nutzer, aber auch für den Programmierer (nicht für den Computer) eingetragen werden, die wichtigen Programmzeilen ihre *externe* Semantik zuordnen. Es kommt darauf an, dass die "konkrete" (externe) Bedeutung von Programmzeilen, die Bezeichner eingeschlossen, sofort erkennbar ist, auch noch nach Jahren.

Aus abstrakter Sicht kommt es darauf an, dass Computer und Anwender ähnliche Begriffe verwenden, genauer gesagt, dass sie bei der *komponierenden Begriffsbildung* einheitlich vorgehen. In Kap.5.5 war die komponierende Begriffsbildung als spezielle Form der Begriffsbildung eingeführt und am Beispiel des Anzuges (bestehend aus Rock, Hose, Weste) illustriert worden. Dort [5.16] war auch bereits darauf hingewiesen worden, dass komponierende Begriffsbildung vorliegt, wenn im Rahmen einer Komponierungshierarchie ein neues Komposit gebildet wird, das im Weiteren die Rolle eines selbständigen Denkobjektes spielt.

Wir kommen zu folgender Aussage: *Bei der Computermodellierung hierarchisch organisierter Originale kann die semantische Lücke zwischen dem Denken des Anwenders und dem des Programmierers durch* ***einheitliche komponierende Begriffsbildung*** *teilweise überwunden werden.* Der Effekt wird noch verstärkt, wenn der Anwender für die Bausteine des Originals (z.B. des Betriebes) und der Programmierer für die entsprechenden Programmbausteine dieselben Namen (Bezeichner,

Fachausdrücke) verwenden. Die semantische Lücke zwischen dem Programmierer und seinem Auftraggeber verschwindet weitgehend. Beide verstehen sich auf einer abstrakten Ebene, auf der ein Name nur noch ein bestimmtes Verhalten, eine Funktionsweise bezeichnet, abstrahiert von allen technischen, auch programmtechnischen Datails.

12 In der Programmierungstechnik wird dieses Vorgehen **prozedurale Abstraktion** genannt, was zum Ausdruck bringen soll, dass ein Operator bzw. eine Operationsvorschrift (eine Prozedur) als "schwarzer Kasten" behandelt wird und von den Einzelheiten der Prozesse, die im Operator bzw. die bei Ausführung der Vorschrift (der Prozedur) ablaufen, abstrahiert wird. Diese Abstraktion gewinnt praktische Bedeutung, wenn die Vorgänge, die bei einer Operationsausführung ablaufen, für die Umgebung, d.h. für andere Operatoren *unsichtbar* sind, m.a.W. wenn die Operatoren einer Hierarchie (bzw. die in ihnen ablaufenden Prozesse) gegeneinander *abgekapselt* sind.

Das verlangt eine programmierungstechnische Realisierung, die garantiert, dass jeder Baustein der Operatorenhierarchie eine relativ abgeschlossene Einheit darstellt, dessen Verhaltensweise die anderen Bausteine zwar kennen, dessen Innenleben sie jedoch nicht kennen und auch nicht beeinflussen können. Dadurch können unerwünschte Wechselwirkungen zwischen den Prozessen, sog. **Seiteneffekte**, weitgehend ausgeschlossen werden. (In der betriebliche Hierarchie entsprechen Seiteneffekte z.B. der nichtkompetenten Einflussnahme auf fremde Produktionsbereiche.)

Zur Verwirklichung dieser Forderung haben sich die Sprachentwickler viele Varianten ausgedacht. Für einen Programmbaustein, der die Forderung erfüllt, hat sich die Bezeichnung Programmmodul eingebürgert. In diesem Sinne wird auch kurz von **Modul** und **Modulhierarchie** gesprochen. Das am weitesten gehende Konzept der Kapselung von Operatoren und Prozessen verwendet den Begriff des *Objekts* (präziser: des *informatischen* Objekts), auf den in den Kapiteln 18 und 19 eingegangen wird. Bild 20.8 zeigt ein Programm für den Softwareentwurf von Netzen aus gekapselten Operatoren und seine Anwendung auf das Operatorennetz von Bild 8.1. Es ist *objektorientiert* programmiert, wodurch trotz des Umfangs des Programms gute Verstehbarkeit und gute Lesbarkeit erreicht werden.

15.8 Analytisches Rechnen

Wie bereits erwähnt, vollzieht sich mathematisches Modellieren auf zwei Ebenen, auf der numerischen, von der in Kap.15.6 die Rede war, und auf einer analytischen, der wir uns nun zuwenden.

Wer erinnert sich nicht an die unbeliebten eingekleideten Mathematikaufgaben? Nach der verbalen Beschreibung irgendeiner Problemsituation wurde nach dem Wert einer oder mehrere Größen gefragt, z.B. nach dem Treffpunkt zweier sich entgegenfahrender Autos oder nach dem Winkel, unter dem zwei Kirchtürme von einem

bestimmten Punkt aus gesehen werden. Zur Lösung musste man in einem ersten Schritt den richtigen Ansatz finden in Form einer oder mehrerer Gleichungen. Diese mussten in einem zweiten Schritt umgeformt werden, um für jede der gefragten Größen einen Ausdruck aus bekannten Größen zu erhalten. In einem dritten Schritt mussten die gefragten Werte berechnet werden, wofür die Schüler heutzutage eventuell ihre Taschenrechner benutzen dürfen. 13

Wie ein Taschenrechner oder ein programmierbarer Rechner numerische Aufgaben löst, haben wir ausführlich besprochen. Es stellt sich die Frage, ob man den Rechner auch die beiden ersten Lösungsschritte ausführen lassen kann. Auf die Delegierung des ersten Schrittes (Ansatzfindung) an den Rechner muss offenbar verzichtet werden, wenn der Ansatz sich nicht formal aus der verbalen Aufgabenstellung ableiten ("berechnen") lässt, wenn er vielmehr auf Intuition beruht, also intuitive (kreative, erfindende) Intelligenz erfordert [7.3]. Um die Frage hinsichtlich des zweiten Schrittes, des Umformens, zu beantworten, knüpfen wir an Kap.15.6 an, wo wir zwischen numerischem und analytischem Modellieren unterschieden hatten, und präzisieren die dortigen Vereinbarungen.

*Ein **analytisches Modell** besteht aus einem oder mehreren Sätzen (wahren Aussagen oder Prädikaten) eines Kalküls über eine oder mehrere Variablen (evtl. auch über Funktionen), die das Original beschreiben (durch das Original interpretierbar sind). Das Umformen der Sätze nach den Regeln des Kalküls ist **analytisches Rechnen*** [8]. Die Sätze können die Form von Ergibtgleichungen (Ergibtanweisungen) oder Relationen (relationale Gleichungen oder Ungleichungen) besitzen [8.20]. 14

Aus der Sicht der praktischen Anwendungen werden von einem mathematischen Modell in der Regel numerische Aussagen verlangt. Diese können unmittelbar nur aus Ergibtgleichungen berechnet werden. In den meisten Fällen liegt das Modell zunächst jedoch in Form von Relationsgleichungen vor, meistens von algebraischen Gleichungen oder von Differenzialgleichungen. Das Überführen in Ergibtgleichungen nennt man **Lösen** der Gleichung. Wenn überhaupt Lösungen existieren, können diese eventuell durch analytisches Rechnen gefunden werden. Ist das nicht möglich, lassen sich die gesuchten Werte unter Umgehung der analytischen Lösung mittels eines geeigneten Näherungsverfahrens numerisch berechnen. Hiermit beschäftigt sich die sog. *numerische Mathematik*.

Am Rande sei erwähnt, dass Differenzialgleichungen als primäre Beschreibung dann zu erwarten sind, wenn physikalische Eigenschaften modelliert werden sollen, denn die Gleichungen der Physik sind in aller Regel Differenzialgleichungen.

In Kap.4.2 war dargestellt, wie *Analog*rechner zur "Lösung" von Gleichungen eingesetzt werden können. In diesem Kapitel wollen wir uns überlegen, wie der *Digital*rechner befähigt werden kann, Gleichungen analytisch zu lösen und ganz allgemein mit analytischen Ausdrücken zu hantieren. Ziel muss dabei nicht unbedingt die Überführung einer Relationsgleichung in eine Ergibtgleichung sein. Ein anderes denkbares Ziel des analytischen Rechnens kann der Beweis sein, dass zwei gegebene Ausdrücke einander gleich sind oder dass ein bestimmter Satz einer formalisierten

Sprache wahr ist. Ein Programm, das derartige Aufgaben löst, wird **Theorembeweiser** genannt.

Der Entwurf eines Programms, das den Computer befähigen soll analytisch zu rechnen, geht, wie im Grunde jede “Beschriftung der tabula rasa” (Kap.7.1), von der Introspektion aus, unabhängig vom Aufgabentyp. So werden auch wir vorgehen. Zuerst überlegen wir uns, wie wir selber verfahren, und dann, ob bzw. wie sich das Verfahren implementieren lässt.

Erinnern wir uns genauer an den Mathematikunterricht. Die meisten Leser werden das Lösen linearer und quadratischer Gleichungen oder das Rechnen mit trigonometrischen Funktionen in mehr oder weniger freundlicher oder auch unfreundlicher Erinnerung haben. Manch einer wird auch das Differenzieren und Integrieren gelernt haben, vielleicht auch, wie einfache Differenzialgleichungen analytisch gelöst werden. All das ist analytisches Rechnen. Mit der Erinnerung daran taucht vielleicht eine Formelsammlung vor dem geistigen Auge auf, in der man nach irgendeiner Formel sucht, entweder nach einer bestimmten, die man vergessen hat, oder nach einer geeigneten, die einem bei der Lösung einer Aufgabe weiterhelfen könnte.

Die Erwähnung der Formelsammlung hat beim Leser, der sich noch an den *Markovalgorithmus* aus Kap.8.4.4 erinnert, möglicherweise ein Aha-Erlebnis ausgelöst, verbunden mit einer Idee, wie sich analytisches Rechnen programmieren lässt. In Kap.8.4.4 war das Vorgehen des Markovalgorithmus mit der Benutzung einer Formelsammlung verglichen worden. Ein Markovalgorithmus verfügt nämlich - in Analogie zur Formelsammlung - über eine *Regelliste*, die angibt, welche Zeichenketten durch welche ersetzt (substituiert) werden dürfen. Der Algorithmus legt die Reihenfolge der Regelanwendungen eindeutig fest, man sagt: das Verfahren ist (der Algorithmus arbeitet) **deterministisch**.

Beim Lösen einer Gleichung, d.h beim Überführen einer relationalen Gleichung in eine Ergibtgleichung geht man analog vor; man substituiert Zeichenketten durch andere. Angenommen, in einer Gleichung tritt der Ausdruck $(1\text{-}\cos^2 x)^{1/2}$ auf. Dann weiß man - z.B. aus einer Formelsammlung - , dass er durch eine ganze Reihe anderer Ausdrücke substituiert werden darf, z.B. durch sin$x$ oder tan$x$*cos$x$. Im Gegensatz zum Markovalgorithmus hat man die Wahl, mit welcher Substitution man die Rechnung fortsetzt.

15 *Ein Algorithmus (eine Rechenvorschrift, ein Verfahren), der Wahlmöglichkeiten offen lässt, heißt* ***nichtdeterministisch*** oder indeterministisch. Analytisches Rechnen ist in zweifacher Hinsicht nichtdeterministisch. Zum einen kann man vor der Wahl stehen, welchen Teil der gesamten Zeichenkette man substitutieren will, zum anderen, welche konkrete Substitution (welche Formel) man anwenden will. Der Markovalgorithmus beseitigt den zuerst genannten Indeterminismus durch die Vorschrift, dass stets der am weitesten linksstehende substituierbare Teilausdruck substituiert wird. Ein Algorithmus, der im Zweifelsfalle vorschreibt, welcher Teilausdruck zu substituieren ist, heißt **kanonisch**. Es lässt sich zeigen, dass Kanonisierung möglich ist, ohne die Funktion, die der Algorithmus berechnet, zu verändern.

Beim Suchen nach einer passenden Formel kommt es auf die Bezeichner der Variablen nicht an (Arbitrarität der Codierung). Beispielsweise kann in den obigen trigonometrischen Ausdrücken der Winkel auch mit y oder mit irgendeinem griechischen Buchstaben bezeichnet sein. Nur muss bei der Verwendung einer Formel die Zuordnung der Bezeichner eindeutig sein. Diese Zuordnung, die man beim Benutzen einer Formelsammlung ganz automatisch macht, nennen wir **Bezeichnerabgleich**. 16

Wenn man sich für eine aus mehreren möglichen Formeln entschieden hat, muss man damit rechnen, dass der eingeschlagene Weg nicht zum Ziel führt und man einen anderen probieren muss. Dazu wird man im Rechengang bis an den Punkt zurückgehen, wo man sich für den falschen Weg entschieden hat, gerade so wie ein Wanderer, der auf einen Abweg geraten ist. In unwegsamem Gelände verfolgt er die eigene Spur zurück bis zur Wegegabel, wo er falsch gegangen ist. Dieses Bild liegt der Bezeichnung **Backtracking** zugrunde, die sich in der Informatik für das beschriebene Zurückgehen beim Suchen eingebürgert hat. 17

Allgemein kann festgestellt werden: *Das Lösen eines Problems führt bei Anwendung eines nichtdeterministischen Verfahrens auf ein* ***Suchproblem***. Als Suchmethode bietet sich das Backtracking an, bei dem das Suchen aus wiederholten Versuchen mit eventuellen Rücksprüngen besteht. Wenn die Lösungssuche nicht systematisch erfolgt, sondern in mehr oder weniger zufälligem Probieren besteht, spricht man von **Trial-and-Error-Methode**.

Wie problematisch Suchen sein kann, hat jeder erfahren, der einmal in einer Mathematikarbeit unter Zeitdruck nach dem richtigen Lösungsweg gesucht hat. Natürlich bedarf es nicht derartiger Erinnerungen, um sich die Bedeutung und Problematik des Suchens vor Augen zu führen. Analysiert man das eigene Alltagsverhalten, erkennt man, dass das Suchen eine ziemlich häufige Beschäftigung ist und dass man dabei meistens die Trial-and-Error-, seltener die Backtracking-Methode anwendet. Darum nimmt es nicht wunder, dass Suchen eines der zentralen Probleme nicht nur des analytischen Rechnens, sondern der künstlichen Intelligenz überhaupt ist. Das zeigt ein Blick in die Inhaltsverzeichnisse einschlägiger Lehrbücher[4].

Ein tieferer Blick in ein KI-Lehrbuch lässt den Weg erkennen, den die Informatiker gehen, um den Computer zum Suchen zu befähigen. Es ist der *Standardweg der Introspektion*: Wie ich es mache, so soll es der Computer machen. Dieser Weg hatte schon beim Addieren erfolgreich Pate gestanden. Er hat auch zum Backtracking geführt. Er wird aber problematisch, wenn beim menschlichen Verhalten *Intuition* ins Spiel kommt, wie es beim Suchen häufig der Fall ist. Das Suchproblem berührt also denjenigen Bereich der Intelligenz, den wir zunächst ausschließen wollten. Wir kommen darauf in den Kapiteln 16.3 und 21.3 zurück.

Damit der Computer dem Menschen beim analytischen Rechnen helfen kann, muss sein Speicher mit geeigneter Software gefüllt werden. Dabei wird es sich um

4 Z.B. [Russell 95],[Schefe 87].

ein mehr oder weniger umfangreiches Programmpaket handeln, je nachdem wie "gescheit" der Computer sein soll, wobei intuitive Intelligenz ausgeklammert bleiben soll. Wir nennen das Programmpaket **Analytiksystem**. *Sein zentraler Teil, der sog.* ***Formelmanipulator****, muss drei Bestandteile enthalten, eine* ***Regelliste*** *(Formelsammlung), ein* ***Suchprogramm*** *und einen* ***Substituierer****.*

Die Regelliste muss an die zu lösende Aufgabenklasse angepasst sein. Sie kann, wenn das Analytiksystem universell sein soll, den Umfang eines dicken mathematischen Nachschlagewerkes annehmen. Ob dieses sich implementieren lässt, ist in erster Linie eine Frage des Speicherplatzes und der Speicherorganisation. Von dem an sich unproblematischen Substituieren war in Kap.8.4 wiederholt die Rede in Verbindung mit dem Markovalgorithmus, der funktionalen Substitution und dem Lambda-Kalkül. Am problematischsten ist das Suchen. Wenn man es nicht dem Zufall überlassen will, auf welchem Wege der Computer ein analytisches Problem zu lösen versucht, müssen die vorwiegend indeterministischen Verfahren, die in Mathematikbüchern angeboten werden, in deterministische Verfahren überführt werden.

18 Die bekannte Suche im Labyrinth illustriert sehr anschaulich, wie sich Suchen deterministisch durchführen lässt. Gegen das Verlaufen im Labyrinth wird der Ariadnefaden empfohlen. An ihm kann man sich stets "zurückhangeln" (Backtracking). Außerdem ist es zweckmäßig, beim Suchen systematisch vorzugehen, sodass man nicht zweimal in dieselbe Sackgasse gerät, aber auch keine Möglichkeit unversucht lässt. Folgender Algorithmus, der aus zwei Wenn-Dann-Regeln besteht, leistet dies.

Regel 1. Wenn du an eine Gabel kommst (Einfach- oder Mehrfachgabel), dann wähle den am weitesten links liegenden noch offenen (d.h. nicht als Sackgasse markierten) Weg.

Regel 2. Wenn du in eine Sackgasse geraten bist, dann gehe den Weg, den du gekommen bist, bis zur nächsten Gabel mit einem noch offenen Weg zurück, markiere den Weg, auf dem du steckengeblieben bist, als Sackgasse und gehe gemäß Regel 1 weiter.

Das ist ein deterministischer Algorithmus (in jedem Moment ist der nächste Schritt eindeutig festgelegt), der auf Backtracking basiert (Regel 2). Es liegt nahe, ihn auch beim analytischen Rechnen anzuwenden. Dazu muss die Reihenfolge, in welcher die Formeln der Formelsammlung in jedem Rechenschritt durchzuprobieren sind, festgelegt werden, z.B. in der Reihenfolge, wie sie in der Formelsammlung bzw. im Speicher aufgelistet sind. Das würde heißen, dass nach jeder Formelanwendung die Liste von Anfang an durchmustert werden muss. In dieser Weise verfährt der Markovalgorithmus, wie wir aus Kap.8.4.4 wissen. Er stellt einen deterministischen Sonderfall des allgemeinen Formelmanipulators dar. Natürlich sind intelligentere Verfahren denkbar als das "stupide" Durchmustern. Auf eins von ihnen werden wir gleich zu sprechen kommen.

Es ist zu beachten, das sich das analytische Rechnen insofern deutlich vom Suchen im Labyrinth unterscheidet, als sich praktisch immer eine anwendbare Formel findet. Trotzdem kann Backtracking sinnvoll sein, nicht, weil man in eine Sackgasse geraten ist, sondern weil man sich deutlich vom Ziel entfernt hat. Die Methode, nach der man merkt, dass man sich vom Ziel entfernt hat, lässt sich oft durch Selbstbeobachtung erkennen und demzufolge auch implementieren.

Beim Theorembeweisen kann man beispielsweise versuchen, ein Maß für den Unterschied zwischen dem aktuellen Ausdruck und dem Zielausdruck zu finden und als **Zielabstand** zu definieren. Dieser sollte durch eine Substitution nicht oder nur unerheblich vergrößert, nach Möglichkeit jedoch verkleinert werden. Die Einführung eines Zielabstandes ist ein häufig begangener Weg, den Computer intelligenter suchen zu lassen, d.h. ihn zu befähigen, effektiver zu suchen als mittels phantasielosen Durchmusterns. Der Erfindungsreichtum der Informatiker hat dazu geführt, dass der Computer beim Problemlösen, das auf Suchen beruht, dem Menschen schon heute ebenbürtig, nicht selten sogar überlegen ist. Das betrifft - innerhalb bestimmter Grenzen - z.B. das analytische Rechnen und das Schachspielen. Viele Strategien sind erdacht worden, die den Lösungsweg minimieren sollen. Die Informatiker nennen eine solche Strategie *Auswertungsstrategie*. 19

Wenn es dem Unerfahrenen so scheint, als wähle der Mathematiker “intuitiv” die richtige Formel oder der Schachprofi den richtigen Zug, so kann der entsprechend programmierte Computer den gleichen Eindruck des Intuitionsbegabten auf den Ahnungslosen machen; doch muss ihm sehr viel Expertenwissen einprogrammiert sein, von dem der Ahnungslose keine Ahnung hat. Damit soll nicht behauptet werden, dass mathematische oder schachspielerische Fähigkeiten lediglich eine Frage des verfügbaren Expertenwissens sind. 20

Es ist an der Zeit, auf eine scheinbar widersprüchliche Schlussfolgerung einzugehen. Wer sich nach allem Bisherigen davon hat überzeugen lassen, dass der Rechner einerseits nichts kann, als rekursive Funktionen berechnen, dass er andrerseits aber analytisch rechnen kann, der muss schlussfolgern, dass analytisches Rechnen nichts anderes ist, als das Berechnen rekursiver Funktionen. Die Schlussfolgerung ist scheinbar widersprüchlich. Denn was haben diese beiden Tätigkeiten miteinander gemein?

Die Frage provoziert den Verdacht, dass die beiden Tätigkeiten vielleicht ebensoviel miteinander gemein haben, wie der Markovalgorithmus mit den rekursiven Funktionen. Denn in Kap.8.4.4 war (ohne Beweis) gesagt worden, dass der Markovalgorithmus rekursive Funktionen berechnet, und in diesem Kapitel haben wir festgestellt, dass er ein Sonderfall der Formelmanipulation ist.

Der kritische Leser wird sich damit nicht zufrieden geben, denn die Behauptung in Kap.8.4.4, der Markovalgorithmus berechne rekursive Funktionen, hat sich auf Autoritäten berufen und war nicht das Resultat eigenen Nachdenkens. Wir werden die Richtigkeit der Behauptung auch jetzt nicht mathematisch beweisen. Doch wollen wir versuchen, sie plausibel zu machen.

Zunächst überzeugen wir uns, dass sich eine konkrete analytische Rechnung durch einen *Aktionsfolgeplan* beschreiben lässt. Zentraler Akteur, der die Ausführung des Plans steuert, ist der Prozessor eines Computers. Der gesamte Inhalt des Arbeitsspeichers übernimmt jetzt die Rolle der Zeichenkette, die schrittweise verändert wird. Einer *Substitution* entspricht eine *Aktion* des Aktionsfolgeplans und damit eine Operation des Prozessors, bei der ein Speicherbereich überschrieben, d.h. sein Inhalt *substituiert* wird. Wenn bei der Formelmanipulation mehrere Formeln anwendbar sind, entspricht das einer Weiche im Aktionsfolgeplan. Die Weichenstellung (Entscheidung der Alternative bzw. Fallauswahl) nimmt der Prozessor gemäß Suchvorschrift vor.

Auf diese Weise lässt sich jede konkrete analytische Rechnung als Aktionsfolgeplan darstellen. Da nun aber jeder solche Plan (jedes Aktionsfolgeprogramm) eine rekursive Funktion berechnet [13.17], folgt, dass beim deterministischen analytischen Rechnen rekursive Funktionen berechnet werden.

Die Argumentation behält ihre Gültigkeit, wenn die mathematische Formelliste durch eine Liste beliebiger Substitutionsregeln ersetzt wird. Dadurch wird die Formelmanipulation zur *Zeichenketten-* oder *Wortmanipulation*. Die Darstellbarkeit als Aktionsfolgeplan bleibt dadurch unberührt und auch ihre Konsequenzen:

Konsequenz 1: Jede von einem deterministischen Wortmanipulator berechnete Funktion ist eine rekursive Funktion.

Diese Konsequenz ist offensichtlich auch in umgekehrter Lesart richtig:

Konsequenz 2: Jede rekursive Funktion kann von einem deterministischen Wortmanipulator berechnet werden,

denn erstens lässt sich zu jeder rekursiven Funktion ein Aktionsfolgeplan für ihre Berechnung angeben, und zweitens lässt sich die Ausführung jedes Aktionsfolgeplans als deterministische Wortmanipulation darstellen. Daraus folgt die Konsequenz 2. Sie wird fast zu einer Trivialität, wenn man bedenkt, dass jede Berechnung durch einen Computer letzten Endes von der ALU ausgeführt wird und dass die ALU Bitketten substituiert, also Wortmanipulationen vornimmt. Aus den Konsequenzen 1 und 2 folgt der

Satz: Die Klasse der durch deterministische Wortmanipulation berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Dieser Satz gilt auch für den speziellen Fall der Markovfunktionen (der nach dem Markovalgorithmus berechenbaren Funktionen), denn der Markovalgorithmus ist ein spezieller deterministischer Wortmanipulator. Damit ist die Richtigkeit der Behauptung aus Kap.8.4.4 nachgewiesen, dass die Klasse der Markovfunktionen mit der Klasse der rekursiven Funktionen identisch ist.

Unsere Schlussfolgerungen gelten auch für numerische Rechnungen, denn arithmetische Funktionen sind rekursive Funktionen. Folglich müssen auch sie sich als Wortmanipulation auffassen lassen, d.h. auch sie müssen Substitutionsregeln befolgen. Eine von ihnen lautet beispielsweise: “2+2 ist durch 4 zu ersetzen”. Jede Zeile

des kleinen Einmaleins ist eine Regel. Die vollständige Regelliste des numerischen Rechnens ist offenbar ziemlich lang, tatsächlich ist sie unbeschränkt.

Die Überlegungen haben noch einmal verdeutlicht, was der Computer letzten Endes tatsächlich macht: er substituiert. Zudem haben sie uns einen Einblick in Zusammenhänge zwischen unterschiedlichen Methoden der Algorithmenbeschreibung gewährt, die in Kap.8.4 ziemlich zusammenhanglos dargelegt worden sind und auf den ersten Blick wenig miteinander zu tun zu haben schienen. Folgende Bemerkungen mögen diese Einsicht noch ein wenig vertiefen.

Wir haben gesehen, dass sich jedes Rechnen, numerisches wie analytisches, auf ein und dieselbe Grundoperation, auf das Substituieren zurückführen lässt. Das gilt nicht nur für das Kopfrechnen und das Rechnen mit Papier und Schreibstift, wie wir es in der Schule gelernt haben, sondern auch für den Computer. Wenn der Prozessor eine TNS-Operation ausführt (Transport Nach dem Speicher), substituiert er den alten Speicherplatzinhalt durch einen neuen. Dabei wird der neue Inhalt entweder einem anderen Speicherplatz entnommen oder er wird von der ALU berechnet.

Es war wohl die zunächst intuitive Überzeugung, dass die Zeichenkettensubstitution als einzige Grundoperation ausreicht, um jedes beliebige algorithmische Verfahren zu beschreiben, die MARKOV zur Definition seines Normalalgorithmus animiert hat und CHURCH zur Definition des Lambda-Kalküls [8.33]. Vielleicht haben ähnliche Vorstellungen TURING zur Konstruktion seiner Maschine angeregt, wobei er nur ganz elementare Operationen zuließ. Dementsprechend begrenzte er das Substituieren auf das Ersetzen eines einzigen Zeichens. Die Regelliste und die Vorschrift für die Weichensteuerung (Bewegung des Schreib-Lesekopfes) schloss er in die Automatentabelle ein.

Damit beenden wir unsere Untersuchung zur Frage, wie der Computer zum "mathematischen Assistenten" qualifiziert werden kann, d.h. wie man ihn befähigen kann, dem Menschen beim Lösen mathematischer Aufgaben zu unterstützen. Die Untersuchung betraf lediglich die prinzipielle Seite der Fragestellung. Was konkret getan werden muss, um den Computer zu qualifizieren, m.a.W. um die tabula rasa mit der notwendigen Software zu beschriften, davon war bisher nicht Rede. Auch die Frage der Nutzersprache ist kaum diskutiert worden. Wir verschieben sie auf die Kapitel 16.4, 18 und 20.

Wir hatten in Kap15.6 [15.8] die Begriffe des analytischen und numerischen Rechnens so definiert, dass mit ihnen das Rechnen in jedem beliebigen Kalkül, m.a.W. alle Arten des mathematischen Operierens erfasst sind. Wir haben gesehen, dass jedes numerische und jedes analytische Rechnen ein Manipulieren mit Formeln ist und dass sich Formelmanipulation algorithmieren und damit implementieren lässt. Damit lautet das Resümee dieses Kapitels: *Der Computer (Prozessorrechner) kann zur Ausführung* ***jeder*** *mathematischen Operation befähigt werden.*

15.9 Bemerkung zur Programmübersetzung

Wenn ein Programm ausgeführt werden soll, das nicht in der Maschinensprache des ausführenden Computers programmiert worden ist, muss es zunächst übersetzt werden. Auf das Übersetzungsproblem wird in Kap. 16.4 eingegangen. Wir wollen uns aber schon jetzt fragen, wie im Prinzip vorzugehen ist, um das Programm von Bild 15.3b in das von 15.3a zu übersetzen. Die Antwort wird durch die einheitliche Grobstruktur (Folge von Aktionsvorschriften) beider Programme nahegelegt. Sie lautet: Die Aktionsvorschriften (Anweisungen) auf der rechten Seite von Bild 15.3 sind durch die entsprechenden Aktionsvorschriften (Befehle oder Befehlsfolgen) auf der linken Seite zu ersetzen. Dies ist offensichtlich das Grundprinzip, nach dem eine imperative Sprache in eine Maschinensprache zu übersetzen ist. Die beiden Sprachen unterscheiden sich im Wesentlichen nur in der Syntax der Aktionsvorschriften.

Fragt man aber, wie eine funktionale Sprache in eine Maschinensprache übersetzt werden kann, gibt es keine so einfache Antwort. Wir wollen versuchen, die Frage für den funktionalen Ausdruck (15.4b) zu beantworten. Er lautete

```
(defun pot(x n)(if(= n 0)1(* x(pot x(- n 1)))))
```
(15.7)

Wir erinnern daran, dass durch einen Ausdruck der Form `(defun f(..)(...))` die Funktion `f(..)` durch den nachfolgenden Klammerausdruck definiert wird, im Falle von (15.7) durch die If-Funktion

```
(if(= n 0)1(* x(pot x(- n 1))))).
```

Das Prinzip der aktionsweisen Übersetzung ist auf funktionale Programme nicht anwendbar. Ein anderes Prinzip ist gesucht. Um es zu finden, erinnern wir uns an die Wurzeln des funktionalen Programmierens, an den Lambda-Kalkül. In Kap.8.4.7. waren Funktionen mit Hilfe des Lambda-Operators durch funktional notierte Ausdrücke (Lambda-Ausdrücke) definiert worden. Wir hatten uns überlegt, wie eine so definierte Funktion "ausgewertet" werden kann, m.a.W. wie die Funktion berechnet werden kann. Die Idee liegt nahe, die dortige Methode auf unser Übersetzungsproblem anzuwenden. Das bedeutet, dass zunächst alle Substitutionen, die möglich sind, ausgeführt werden, denn die "Auswertung" eines Lambda-Ausdrucks begann mit der *Lambda-Eliminierung* [8.35] mittels Substitution. Damit das Prinzip der Methode deutlich sichtbar wird, substituieren wir zunächst nur eine der beiden Variablen durch einen Wert und zwar `n` durch denWert `3`. Dann geht (15.7) in (15.8) über:

```
(defun pot(x 3)(if(= 3 0)1(* x(pot x 2))).
```
(15.8)

Das Prädikat `(= n 0)` ist nicht erfüllt. Folglich ist die If-Funktion (laut Sprachdefinition von CommonLisp) durch den zweiten Ausdruck nach dem Prädikat, also

durch `(* x(pot x 2))` zu substituieren (der erste Ausdruck nach dem Prädikat ist der Wert `1`). Das ergibt

```
(defun pot(x 3)(* x(pot x 2))).
```

Nach dem gleichen Vorgehen ist der Reihe nach `(pot x 2)`, `(pot x 1)` und `(pot x 0)` zu substituieren. Die letzte Substitution liefert für die If-Funktion den Wert `1`. Das Resultat aller Substitutionen lautet

```
(defun pot(x 3)(* x(*x(*x 1)))).
```
(15.9)

In der bisherigen Prozedur wird der Leser den ersten Teil einer rekursiven Berechnung erkannt haben. In Kap.8.4.6 hatten wir sie am Beispiel der Fakultät-Funktion durchexerziert. Es folgt nun der zweite Teil, die *Wertberechnung*, wie wir ihn in Kap.8.4.7 [8.35] genannt hatten. Er bereitet keine Schwierigkeiten, denn jeder Ausdruck der Form `(* x u)` steht für das Resultat `r` einer Ergibtanweisung `r := x*u)`. Demzufolge lässt sich (15.9) in eine Folge von Multiplikationsaktionen überführen:

```
a := x*1
b := x*a
y := x*b.
```

Darin sind `a` und `b` Hilfsvariablen und `y` bezeichnet den Wert `pot(x 3)`. Damit ist der funktionale Ausdruck (15.8) imperativ notiert. Der nächste Schritt ist die Übersetzung in die Maschinensprache, die Zuweisung eines Wertes an `x` und die Abarbeitung.

Nach diesem Prinzip werden funktionale Programme übersetzt und ausgeführt, man sagt *interpretiert*. Das interpretierende Programm heißt **Interpreter**. Auch logische Programme werden interpretiert, worauf in Kap.20.2.4 eingegangen wird. Das Interpretieren eines funktionalen oder logischen Programms beginnt stets mit dem Substituieren. Wenn in dem Programm keine rekursiven Funktionsdefinitionen auftreten (wie beispielsweise die Definition der Potenzfunktion), enthält die Substitutionsprozedur keine rekursiven Iterationen, sondern besteht aus einer Folge einzelner Substitutionen.

16 Lösen nichtmathematischer Probleme

Zusammenfassung

Mit der Wortmanipulation ist ein erster Schritt in den “nichtmathematischen” Bereich getan, denn die umzuformenden Zeichenketten müssen aus der Sicht des Menschen, der sich vom Computer helfen lässt, nicht Ausdrücke eines mathematischen Kalküls sein. Im nächsten Schritt werden Substitutionsregeln mit externer Semantik zugelassen, die dem alltäglichen logischen Schlussfolgern zugrunde liegen, beispielsweise der Schlussfolgerung, dass der Sohn der Schwiegermutter meiner Mutter entweder mein Vater oder mein Onkel ist. Menschliches *Schlussfolgern* im gängigen Sinne des Wortes ist ein Ableiten von Schlüssen aus gegebenen Fakten nach den Regeln des logischen Denkens, wobei aus introspektiver Sicht des denkenden Menschen das Ableiten nicht explizit formalisiert ist.

Um Schlussfolgern zu implementieren, muss es zunächst kalkülisiert werden. Dazu sind die Fakten und Regeln in der formalen Sprache eines Kalküls, beispielsweise in der Sprache des Prädikatenkalküls (der Prädikatenlogik) zu formulieren. Das Schlussfolgern kann dann auf formalem Wege nach den Regeln des Prädikatenkalküls erfolgen. Dabei wird von jeder externen Semantik abstrahiert, nachdem sie an die formale Semantik des Prädikatenkalküls angebunden worden ist. Kalkülisierung betrifft immer einen speziellen Bereich des Denkens (des Nachdenkens), den sog. *Diskursbereich*, auf den sich das externsemantische *Fakten-* und *Regelwissen* bezieht. Insofern stellt das Ergebnis der Kalkülisierung einen *speziellen Denkkalkül* dar.

Auf diese Weise wird das Schlussfolgern zum formalen Ableiten. Es besteht aus Syntaxvergleichen, Substitutionen und Transformationen gemäß Regeln. Diese Art des Schlussfolgerns wird *Inferenzieren* genannt. Eine formale Schlussfolgerung, eine *Inferenz*, kann von einem Computer vollzogen werden, der über ein einschlägiges Inferenzprogramm und über das erforderliche *Fakten-* und *Regelwissen* verfügt.

Ein Inferenzprogramm zusammen mit dem Fakten- und Regelwissen eines bestimmten Anwendungsgebietes (z.B. der Buchführung oder der Konstruktion von Motoren) heißt *Expertensystem* für das betreffende Gebiet. Ein Expertensystem muss über ein ausreichend universelles und effizientes Inferenzprogramm, über eine geeignete Dialogsprache und über ein ausreichendes und leicht abrufbares Wissen über das Fachgebiet (über den Diskursbereich) verfügen.

Ein Expertensystem kann Leistungen vollbringen, die als intuitive Leistung oder als Erfindung bezeichnet werden können. Beispielsweise könnte ein ausreichend “qualifiziertes” Expertensystem für Chemie in der Lage sein, aus dem Wissen, das Kekulé besaß, die Formel des Benzolringes abzuleiten, d.h. die Formel zu “erfinden”, wie Kekulé sie erfunden (intuitiv gefunden) hat. Wenn eine Aussage auf intuitivem

Wege gefunden wird, obwohl sie durch Ableitung hätte gefunden werden können, sprechen wir von *reduzibler Intuition*.

Voraussetzung für die Implementierung irgendeines speziellen Denkkalküls ist die automatische Übersetzung der betreffenden Kalkülsprache in die Maschinensprache des verwendeten Rechners, m.a.W. die Verwirklichung der vierten Grundidee des elektronischen Rechnens. Aus der Sicht des übersetzenden Menschen, beispielsweise eines Dolmetschers, ist Übersetzen eine nichtmathematische Leistung menschlicher Intelligenz. Doch ist sie kalkülisierbar und kann vom Computer erbracht werden.

Das Übersetzen kann durch einen Interpreter oder einen Compiler ausgeführt werden. Ein *Interpreter interpretiert* die Sätze (die interpretierbaren Programmstücke) der Sprache des speziellen Denkkalküls, d.h. er übersetzt die einzelnen Programmstücke und führt sie aus. Ein *Compiler* übersetzt ein Programm im Ganzen aus der Eingabesprache, auch *Quellsprache* genannt, in die Ausgabesprache, auch *Zielsprache* genannt. Zielsprache ist die Maschinensprache, der sog. Objektcode für den Binder.

Ein Compiler (Übersetzerprogramm) besteht aus einem Scanner, einem Parser und einem Codegenerator. Der *Scanner* führt die *lexikale Analyse* durch, d.h. er klassifiziert die einzelnen Wörter (die Lexeme) des Quellprogramms. Beispielsweise können alle Variablenbezeichner zu einer Klasse, alle Operationssymbole zu einer anderen Klasse zusammengefasst werden. Der *Parser* führt die *syntaktische Analyse* durch, d.h. er erkennt und klassifiziert syntaktische Konstrukte und benennt sie mit metasprachlichen Klassennamen. Beispielsweise erkennt er eine Folge Bezeichner-Ergibtzeichen-Ausdruck als Ergibtanweisung. Aus den metasprachlichen Klassen baut der Parser eine hierachische Struktur auf, welche die syntaktische Struktur des Quellprogramms vollständig und ohne Rest wiedergeben muss. Der *Codegenerator* überführt diese Struktur in die Zielsprache, indem er die Konstrukte der Quellsprache durch Konstrukte der Zielsprache gemäß einer Regelliste substituiert.

Das Vorgehen ist dem nichtmaschinellen Übersetzen zwischen natürlichen Sprachen abgeschaut und kann auf diese angewendet werden. Die syntaktische Strukturierung, mit anderen Worten die Verwendung grammatikalischer Regeln, ist das entscheidende Mittel natürlicher wie künstlicher Sprachen zur Erhöhung der Aussagekraft einer Sprache, zur Steigerung der semantischen Dichte. Bei Verwendung *generativer Grammatiken* kann die aufwendige syntaktische Analyse sehr effektiv gestaltet werden, und es lassen sich Regeln angeben, nach denen Programmiersprachen zu entwerfen sind, um sie übersetzerfreundlich zu machen.

16.1 Schlussfolgern

Wir waren zu der Einsicht gelangt, dass ein Problem, das der Computer lösen soll, in kalkülisierter Form vorliegen muss, damit eine Lösungsvorschrift programmiert

werden kann. Eingedenk dieser Einsicht wird der Leser die Überschrift "Lösen nichtmathematischer Probleme" richtig verstehen. Um dennoch mögliche Missverständnisse zu vermeiden, soll die Überschrift ausführlicher artikuliert werden: "Lösen von Aufgaben durch den Computer, die vom Menschen auf nichtmathematischem Wge gelöst werden". Nur aus der Sicht des Menschen hat es Sinn, ein Problem als "nichtmathematisch" zu bezeichnen. Für den Computer gibt es keine nichtmathematischen Probleme (siehe Kap.14.2).

In Kap.7 hatten wir verschiedene Wege herausgearbeitet, auf denen der Mensch zu neuen Aussagen über die Welt und zu neuen sprachlichen Modellen der Welt gelangen kann, den deduktiven, den assoziativen und den intuitiven Weg. Dementsprechend hatten wir zwischen *deduktiver, assoziativer* und *intuitiver Intelligenz* (Fähigkeit zum sprachlichen modellieren) unterschieden. Deduzieren (ableiten) kann entweder *Rechnen* oder *Schlussfolgern* sein. In Kap.15 haben wir uns überlegt, wie der Computer rechnen kann, und zwar rechnen in einem *beliebigen* Kalkül. Jetzt werden wir uns überlegen, ob der Computer auch zum Schlussfolgern und zum Assoziieren befähigt werden kann. Wir werden sogar die Frage aufwerfen, ob er mit intuitiver Intelligenz ausgestattet werden kann. Der Leser erkennt, das wir im Sinne des Gegenläufigkeitsprinzips vorgehen.

Auf dem Wege zur künstlichen Intelligenz hat uns der Prozessorrechner mit seinem - wenn auch recht primitiven - Sprachverständnis bisher nicht im Stich gelassen. Er scheint allen gängigen mathematischen Problemen gewachsen zu sein. Nur muss er, genauso wie der Mensch, "gelernt haben, mit den Problemen umzugehen". Die Art und Weise des Lernens ist allerdings kaum mit der des Menschen zu vergleichen. Dass die Wiederholung die Mutter der Wissenschaft (des Wissens und Könnens) ist, trifft auf den Computer nicht zu. Es genügt, wenn ihm der "Lehrstoff" einmal vermittelt ("eingetrichtert") wird, allerdings genau und bis in alle Einzelheiten. Wiederholen und Üben erübrigt sich. Daran wird sich nichts ändern, wenn wir nun die Gefilde der Mathematik verlassen, d.h. den Bereich, in dem der Mensch durch Rechnen zu neuen Aussagen gelangt, und die "Reise zur KI" fortsetzen, auf der Suche nach der Grenze der künstlichen Intelligenz. Wir bleiben, getreu unserer Zielstellung, auf der symbolischen Betrachtungsebene [1.4], und unser Werkzeug ist der Prozessorcomputer, nicht der Neurocomputer.

Es mag zweifelhaft erscheinen, dass der Computer, der "eigentlich doch nur" rekursive Funktionen berechnen kann, imstande ist, logische Schlussfolgerungen zu ziehen, dass er sinnvolle Assoziationen haben oder irgendetwas Sinnvolles intuitiv finden oder erfinden kann. Es wird sich zeigen, dass der Zweifel zum Teil unseren "Denkgewohnheiten über das Denken" entspringt und dass auch intuitives und erfinderisches Denken in gewissem Grade durchaus simulierbar ist. Außerdem werden wir erkennen, dass das "Feld der Mathematik" ein viel weiteres ist, als dasjenige, welches einem in der Schule gezeigt wurde. Bevor wir dieses Feld betreten, vergegenwärtigen wir uns noch einmal unser Anliegen.

Wir wollen verstehen, was es mit der künstlichen Intelligenz auf sich hat, was man von ihr erwarten kann und was sie offenbar nicht zu leisten vermag. Wir begnügen uns damit, zu erkennen (nachzuerfinden), wie es *im Prinzip* möglich ist, dass der Computer auf der Grundlage ihm bekannten Wissens zu neuen Einsichten gelangt, die sich scheinbar nicht durch numerisches oder analytisches Rechnen ableiten lassen. Der Zusatz "im Prinzip" bedeutet, dass wir unsere Überlegungen abbrechen, sobald wir den "prinzipiellen" Weg erkannt haben. Bekanntlich liegt der Teufel im Detail, d.h. in den Schwierigkeiten, die im konkreten Fall zu überwinden sind. Sie erfordern eventuell umfangreiche theoretische Untersuchungen, die über das Anliegen und den Rahmen des Buches hinausgehen. Wir brechen also genau da ab, wo es für die Theoretiker anfängt interessant zu werden[1].

In der Hoffnung, "auf den ersten Blick" Grenzen der künstlichen Intelligenz zu erkennen, betrachten wir eine Eigenschaft, die als *Findigkeit* (geistige Wendigkeit, Pfiffigkeit, Einfallsreichtum u.ä.m.) bezeichnet wird. Das ist eine Eigenschaft, die Menschen in mehr oder weniger hohem Grade besitzen, über die der Computer aber offenbar nicht verfügt. Rätsel sind ein beliebtes Mittel, die Findigkeit eines Menschen auf die Probe zu stellen, seine "Intelligenz" zu testen. Was liegt näher, als diese Methode auf den Computer anzuwenden, wenn es darum geht, die Grenzen seiner
1 Intelligenz zu erkennen? Nehmen wir das Rätsel aus dem Ödipusmythos: "Am Morgen geht es auf vier Beinen, am Mittag auf zwei, am Abend auf drei. Was ist das?" Wie kann der Computer befähigt werden, die Antwort (der Mensch) zu "erraten"? Eine triviale Methode bestünde darin, die Lösung "vorzusagen". Würde man sämtliche Rätsel der Welt samt ihrer Lösungen abspeichern, könnte der Computer alle Rätsel "lösen".

Diese Methode, die im Grunde ein Nachplappern oder Nachschlagen ist, hat mit Intelligenz scheinbar nicht viel zu tun. Tatsächlich ist sie genauso "intelligent", wie die maschinelle Berechnung einer booleschen Funktion, deren Wertetafel im Computer abgespeichert ist. Man ist geneigt, derartiges einfaches "Nachschlagen" nicht als intelligente Methode anzuerkennen. Unwillkürlich meint man, eine intelligente Problemlösungsmethode müsste mit Nachdenken, mit logischem Schließen oder mit Ableiten, mit Deduzieren zu tun haben. Für das Lösen mathematischer Aufgaben durch numerisches oder analytisches Rechnen trifft das offensichtlich zu. Darum ist es verständlich, dass der Computer sie lösen kann. Doch wie kann er ohne "Vorsagen" *nicht*mathematische Aufgaben lösen, für die es keine Rechenregeln zu geben scheint? Wie kann der "Rechner nichtmathematisch denken"? Dass er es kann, hatten wir das "Paradoxon der KI" genannt.

Wir stehen vor der Frage, wie sich die Grenze der simulierbaren Intelligenz über das in Kap.15 besprochene Rechnen hinaus verschieben lässt. In den vorangehenden

---

1 Dem interessierten Leser steht eine umfangreiche Literatur zur Verfügung, u.a. [Schefe 87], [Schöning 89], [Russel 95].

Kapiteln haben wir angedeutet, wie sich die Kalküle der Arithmetik und der Analysis[2] und überhaupt jeder analytische Kalkül in den Maschinenkalkül, also letzten Endes in den booleschen Kalkül abbilden lässt. Jetzt erhebt sich die Frage, wieweit sich menschliches Denken, das scheinbar nicht, zumindest nicht bewusst, nach den Regeln irgendeines Kalküls abläuft , kalkülisieren lässt, sodass es simulierbar wird.

Kehren wir zu unserem Rätsel zurück und suchen nach irgendwelchen Anhaltspunkten, wie der Mensch die Lösung möglicherweise finden könnte. Dabei stellt sich heraus, dass eine Art Findigkeit erforderlich ist, über die auch die natürliche Intelligenz nur selten verfügt. Wenn jemand das Rätsel ohne Hilfe "errät", hat er es wahrscheinlich gekannt. Möglich wäre allerdings, dass einem hellen Kopf durch *Assoziation* "einfällt", dass der Mensch *vier* Extremitäten besitzt, die er zunächst alle vier, später aber nur zwei von ihnen zur Fortbewegung benutzt. Das Finden der Antwort scheint aber nicht kalkülisierbar zu sein, zumindest nicht so ohne Weiteres. Auf jeden Fall muss das Problem in einer geeigneten verallgemeinerten, abstrakteren Form beschrieben werden.

Im Augenblick wollen wir den "assoziativen Weg" nicht weiter verfolgen, sondern nach Rätseln suchen, deren Lösungen zwar nicht "mathematisch" (im üblichen Sinne des Wortes), aber doch "irgendwie" ableitbar zu sein scheinen, die sich in gewissem Sinne kalkülisieren lassen. Das ist immer dann möglich, wenn dem Ableiten ein Schlussfolgern nach bestimmten Regeln zugrunde liegt. Das ist für das Lösen von Denksportaufgaben charakteristisch. Wir dürfen erwarten, dass sich in diesem Fall das menschliche Vorgehen simulieren lässt, freilich nur soweit, wie es auf Ableiten aus vorhandenem Wissen beruht. Versuchen wir es mit einer bekannten Aufgabe.

Denksportaufgabe 1: Verwandtschaftsproblem 2

Eine gedachte Person stellt eine andere mit folgenden Worten vor: "*Ist doch dieses Mannes Mutter meiner Mutter Schwiegermutter*." Frage: In welchem Verwandtschaftsverhältnis steht der Vorgestellte zum Vorstellenden?

Man muss schon sehr fix im Denken sein, um die Antwort (Vater oder Onkel) sofort parat zu haben. Sogar einige Minuten des Nachdenkens reichen eventuell nicht aus. Das Nachdenken artet leicht in ein Probieren aus, eventuell in ein ständig wiederholtes Neuanfangen, wobei die Systematik des Suchens nach wenigen Schlussschritten verloren geht. Offenbar reicht das Kurzzeitgedächtnis nicht aus, um alle versuchten Wege ständig klar gegenwärtig zu haben. Der Ariadnefaden fehlt.

Beim Schachspiel ist es ähnlich. Die Anzahl der Züge, die ein Spieler vorausdenken kann, ist durch sein Kurzzeitgedächtnis begrenzt[3]. Aus dem gleichen Grunde ist die Länge von Kopfrechnungen begrenzt. Ist das Kurzzeitgedächtnis überfordert,

---

2 Analysis ist nicht mit analytischem Rechnen zu verwechseln.

3 Ob der Begriff des Kurzzeitgedächtnisses den Sachverhalt genau trifft, bleibt dahingestellt.

greift man zu Papier und Stift und ergänzt das Gedächtnis durch einen "externen Speicher".

Wenn man nicht zu den Schnelldenkern gehört und sich an die Lösung obiger Denksportaufgabe macht, beginnt man vernünftigerweise damit, sich das Wissen über Verwandtschaftsbeziehungen zu vergegenwärtigen. Dabei handelt es sich um Sprach"*regeln*" (Bezeichnungsregeln), die z.B. festlegen, unter welchen Bedingungen welche Verwandtschaftsbeziehung bestehen oder welche Beziehungen welche anderen nach sich ziehen. Zweckmäßigerweise notiert man sich alles, was für die Lösung der Aufgabe brauchbar zu sein scheint. Das notierte Wissen könnte folgende "*Regeln*" enthalten:

1. Wenn x Schwiegermutter von y ist, dann existiert eine Person z, die Kind von x ist und die mit y verheiratet ist (Schwiegermutterregel).
2. Wenn x mit y verheiratet und die Mutter von z ist, dann ist y der Vater von z (Vaterregel).

Beim anschließenden eigentlichen Lösen der Aufgabe kann man dann - in Analogie zum Nachschlagen in einer Formelsammlung - die Liste auf anwendbare Regeln durchsuchen. Ebenso wie in einer Formelsammlung haben gleiche Variablenbezeichner, die in verschiedenen Formeln/Regeln auftreten, an sich nichts miteinander zu tun; erst der Kontext, in dem sie auftreten, legt Beziehungen zwischen den Variablen fest.

Die beiden genannten Regeln haben die Form von Wenn-dann-Sätzen; sie stellen *Implikationen* dar (man erinnere sich an die Regeln in Entscheidungstabellen [12.5]). Der Wenn-Teil enthält die **Prämisse**(n), der Dann-Teil die **Konklusion**(en). Die Schwiegermutterregel gilt stets, die Vaterregel nur "in der Regel". Wenn wir im Weiteren eine Regel anwenden, nehmen wir zunächst an, dass sie im betrachteten konkreten Fall tatsächlich gilt (z.B. dass y der Vater und nicht der Stiefvater von z ist). Die erste Regel ist für die Lösungsfindung sicher erforderlich. Ob das auch für die zweite Regel gilt und ob noch andere Regeln benötigt werden, ist nicht ohne Weiteres zu erkennen.

Um mit unserem Wissen besser hantieren zu können, überführen wir es in eine übersichtliche, standardisierte Kurznotation. Dazu wählen wir die Sprache des **Prädikatenkalküls**[4]. Wir vereinbaren vier zweistellige Prädikate und interpretieren sie, d.h. wir ordnen ihnen ihre externe Semantik zu:

`ehe(x,y)` - x ist mit y verheiratet,
`va(x,y)` - x ist Vater von y,
`mu(x,y)` - x ist Mutter von y,
`smu(x,y)` - x ist Schwiegermutter von y.

4 Anstelle von Prädikatenkalkül wird auch Prädikatenlogik gesagt. Wenn im Weiteren von Prädikatenkalkül die Rede sein wird, ist darunter stets der sogenannte Prä*dikatenkalkül erster Ordnung* zu verstehen. In der englischsprachigen Literatur wird er häufig als *First-Order Logic* bezeichnet.

Die Zeichenketten `ehe, va, mu, smu` sind Namen (Bezeichner) von Prädikaten. Die Variablen eines Prädikats hatten wir Individuenvariablen genannt [8.19]. Bild 16.1 stellt die Verwandtschaftsverhältnisse in Form eines Graphen dar.

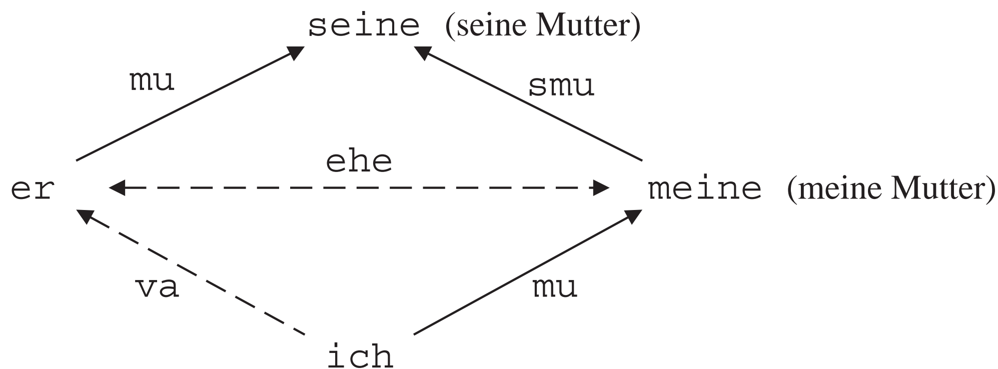


Bild 16.1 Verwandtschaftsgraph zu Denksportaufgabe 1 (Verwandtschaftsproblem). Bezeichnungen siehe Text. Die Pfeile zeigen jeweils zum ersten Prädikatargument, z.B. der Pfeil des Prädikats `va` auf den Vater. Die Beziehungen der durchgezogenen Pfeile sind durch die Aufgabenstellung gegeben.

Mit den vereinbarten Bezeichnungen ist folgende Notation der ersten der oben genannten Regeln naheliegend:

`smu(x,y)` ⇒ `mu(x,z) AND ehe(y,z).`

Intuitiv ist klar, dass die Konklusion für alle x und y gilt, für welche die Prämisse `smu(x,y)` gilt. Sie gilt aber nicht für jedes z, doch existiert ein solches z mit Sicherheit. Um das zum Ausdruck zu bringen, verwendet der Prädikatenkalkül den sogenannten **Existenzquantor**. Man schreibt $\exists x$:..., liest dies als “Es existiert ein x, für das gilt:...” und sagt, dass x durch den Existenzquantor $\exists$ **gebunden** ist. Wenn dagegen ein Prädikat für alle Werte einer Individuenvariablen x wahr ist, schreibt man $\forall x$:..., liest dies als “Für alle x gilt” und sagt, dass x durch den **Allquantor** $\forall$ gebunden ist. Damit können wir unser **Regelwissen** (die beiden Regeln, die wir uns gemerkt haben) folgendermaßen notieren:

Regel A: ∀`x` ∀`y: smu(x,y)` ⇒ ∃`z: (mu(x,z) AND ehe(y,z)),`

Regel B: ∀`x` ∀`y` ∀`z: mu(x,z) AND ehe(x,y)` ⇒ `va(y,z).`

Um auch unser **Faktenwissen** über den Diskursbereich in Kurzform notieren zu können, vereinbaren wir für die beteiligten Personen folgende Namen (Bezeichner von Individuen*konstanten*):

`ich` - der Vorstellende,
`er` - der Vorgestellte,
`meine` - Mutter des Vorstellenden,
`seine` - Mutter des Vorgestellten.

Damit lautet das Faktenwissen:

Fakt 1: `smu(seine,meine),`
Fakt 2: `mu(meine,ich) ,`
Fakt 3: `mu(seine,er) .`

Die Frage könnte folgendermaßen notiert werden:

Frage: `?(er,ich).`

Das Fragezeichen bezeichnet das gesuchte Prädikat (die gesuchte Beziehung). Fakt 1 ist die Kurznotation desjenigen Satzes, mit dem in der Aufgabenstellung “er” vorgestellt wird. Die Fakten 2 und 3 ergeben sich als selbstverständlicher Kontext aus dem Satz des Vorstellenden, sodass sie kaum der Erwähnung wert zu sein scheinen. Dennoch dürfen sie nicht fehlen, wenn der Computer in der Lage sein soll, die Lösung aus dem ihm mitgeteilten Wissen abzuleiten. Er kann nämlich auf keinerlei externe Semantik zurückgreifen. Er kann nur “blindlings”, ohne jedes Verständnis, mechanisch, automatisch - oder wie immer man das “Denken” des Computers bezeichnen will - vorgehen. Unter externer Semantik (siehe Bild 5.3) ist das gesamte Kontextwissen zu verstehen, das sich auf die konkrete Situation bezieht bzw. mit ihr in Verbindung gebracht werden kann und das jedem Menschen, der sich in der betreffenden Situation befindet, “automatisch” zur Verfügung steht, d.h. das er mit der Situation *assoziiert*. Das Problem der Anbindung der *externen* Semantik (Nutzersemantik, Humansemantik) über die *formale* Semantik (Kalkülsemantik) an die *interne* Semantik des Computers (Maschinensemantik, die Prozesse in der Hardware) hatten wir das technische Semantikproblem genannt.

Für das *formale* Schlussfolgern, d.h. für das Schlussfolgern nach den Regeln des Prädikatenkalküls unter Abstraktion von jeglicher externen Semantik, wird das Wort *Inferenzieren* verwendet. Wir vereinbaren: *Das formale Schlussfolgern nach den Regeln des Prädikatenkalküls heißt* ***Inferenzieren****. Eine durch Inferenzieren hergeleitete Schlussfolgerung heißt* ***Inferenz****. Die dem Inferenzieren zugrunde liegenden Regeln bzw. Fakten bilden das sog.* ***Regel-*** bzw. ***Faktenwissen***.

Das gesteckte Ziel besteht also in der Lösung der Denksportaufgabe durch Inferenzieren. Wenn man dabei die obigen Regeln A und B verwendet, stellt man fest, dass der Ableitungsprozess etwas umständlich wird, dass er sich aber durch Umformulierung dieser Regeln zu

Regel 1: `smu(x,y) AND mu(x,z)` $\Rightarrow$ `ehe(y,z),`
Regel 2: `mu(u,v) AND ehe(u,w)` $\Rightarrow$ `va(w,v)`

vereinfachen lässt.

Die Richtigkeit der beiden Regeln ist unschwer zu verifizieren. Regel 2 unterscheidet sich von Regel B durch andere Variablenbezeichner und durch das Fehlen der Allquantoren. Die Wahl neuer Bezeichner ist keine Notwendigkeit; sie soll die Gefahr ausschließen, dass der Leser die Regeln falsch interpretiert, indem er annimmt, dass gleiche Bezeichner in den beiden Regeln dieselben Variablen bezeichnen. Es wäre genauso richtig gewesen, auch in Regel 2 die Bezeichnungen *x, y* und *z* zu verwenden. Wenn in Regel 1 dieselben Bezeichner auftreten, führt das zu keiner

Fehlinterpretation durch den Interpreter (durch das Übersetzerprogramm), denn, wie bereits erwähnt, haben Variablenbezeichner in verschiedenen Regeln an sich (ohne Berücksichtigung des Kontextes, in dem sie auftreten) nichts miteinander zu tun; beispielsweise hat das x in Regel A nichts mit dem x in Regel B zu tun. Dieser Sachverhalt sei wegen seiner Wichtigkeit noch einmal herausgestellt. *Der **Gültigkeitsbereich** eines Variablenbezeichners, d.h. die Bindung eines Bezeichners an eine Variable und letzten Endes an einen Speicherplatz beschränkt sich auf die jeweilige Formel*. Das Fehlen von Allquantoren in den Regeln 1 und 2 beruht auf der stillschweigenden Vereinbarung, dass Allquantoren nicht geschrieben zu werden brauchen, wenn sie für alle Variablen gelten.

3

Die Regeln 1 und 2 sind sogenannte Hornklauseln. *Eine Implikation, deren Prämisse eine Konjunktion aus beliebig vielen elementaren Prädikaten ("Bausteinprämissen") und deren Konklusion ein elementares Prädikat ist, wird als **Hornklausel** bezeichnet*. Die Überführung prädikatenlogischer Ausdrücke in Hornklauseln ist ein wichtiger "Normalisierungsschritt" des maschinellen Inferezierens.

Nach diesen längeren Vorbereitungen können wir mit dem Inferenzieren beginnen. Es besteht in einem "systematischen Versuchen", die Regeln durch die Konstanten (durch die beteiligten Personen) unter Berücksichtigung der Fakten zu befriedigen. Wir beginnen mit dem ersten Prädikat in der Prämisse von Regel 1 und suchen im Faktenwissen einen "passenden" Fakt, d.h. einen solchen, dessen syntaktische Struktur mit der des Prädikats übereinstimmt. Das trifft für Fakt 1 zu, der sich von `smu(x,y)` nur darin unterscheidet, dass er Konstante anstelle der Variablen enthält. Demzufolge substituieren wir versuchsweise x durch `seine` und y durch `meine`. Für die Beschreibung einer Substitution werden verschiedene Notationsweisen verwendet. Eine gängige Notation ist z.B. [`x/seine`] und entsprechend [`y/meine`]. Da im vorliegenden Fall Variablen durch Konstante substituiert werden, handelt es sich um "Wertzuweisungen" in einem verallgemeinerten Sinne, sodass das Ergibtzeichen (Wertzuweisungszeichen) := gerechtfertigt ist (siehe Bild 16.2).

Man beachte die Ähnlichkeit zum Vorgehen nach dem Markovalgorithmus oder zum Vorgehen beim analytischen Rechnen. Ganz im Sinne des Markovalgorithmus nehmen wir nun das zweite Prädikat in der Prämisse von Regel 1 und suchen nach einem *passenden* Fakt. Zuerst stoßen wir auf Fakt 2, stellen aber fest, dass er mit der ersten Substitution nicht "zusammenpasst", denn der Variablen y ist bereits die Person (die Konstante) `meine` zugewiesen. Die weitere Suche führt auf Fakt 3, und man stellt fest, dass er mit der vorherigen Zuweisung `x:=seine` zusammenpasst.[5] Wir substituieren also `z:=er`.

5 Im Fachjargon hat sich eine sprachästhetisch unschöne "neudeutsche" Sprechweise eingebürgert: Für "passen" wird das eingedeutschte Wort "*matchen*" (vom englischen Verb to match = passen, zusammenpassen) verwendet. Statt "Fakt 3 passt" würde "Fakt 3 matcht" gesagt werden.

Mit diesen Substitutionen ergibt sich der
Schluss 1: `ehe(meine,er)`

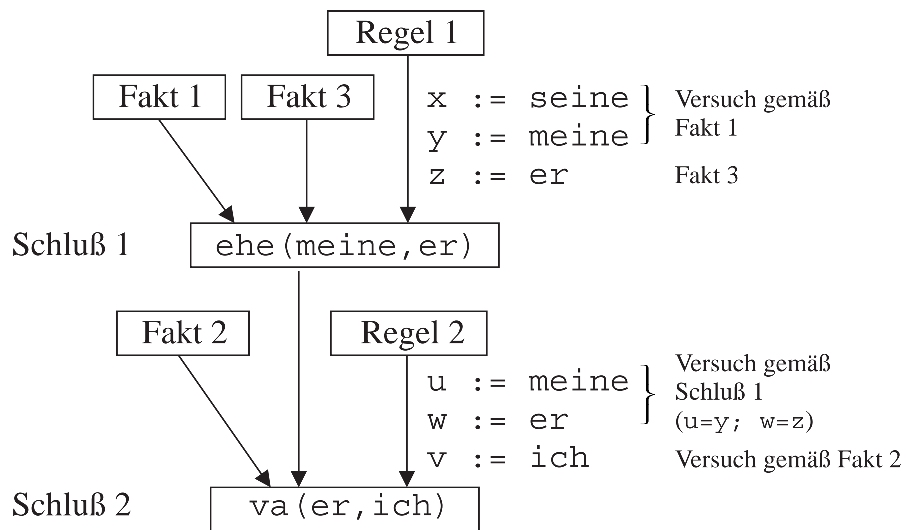


Bild 16.2 Inferenzbaum zum Verwandtschaftsproblem

Der Weg zu Schluss 1 ist in Bild 16.2 graphisch dargestellt. Die Pfeile zeigen von der Regel 1 und den verwendeten Fakten zur Konklusion, zum Schluss 1. Rechts neben dem Pfeil, der von Regel 1 ausgeht, sind die dabei durchgeführten Substitutionen angegeben, sowie die Begründungen der Substitutionen. Nach dem gleichen Muster ist der Weg von Schluss 1 zu Schluss 2 dargestellt. In Klammern ist die Entsprechung zwischen Variablenbezeichnern hinzugefügt. Beispielsweise bedeutet w=z, dass den Variablen w und z derselbe Wert (dieselbe Person), nämlich `er` zugewiesen wird.

Die Richtigkeit des zweiten Schlussfolgerungsschrittes ist so offensichtlich, dass der gedanklich schlussfolgernde Mensch ihn “automatisch” (unbewusst, reflektorisch) vollzieht. Wenn man weiß, dass `er` der Mann meiner Mutter ist, weiß man auch (ohne nachdenken zu müssen), dass `er` mein Vater ist.

Wir haben eine Lösung der Denksportaufgabe durch formales Schlussfolgern, durch Inferenzieren gefunden, also mit Hilfe einer Methode, die sich implementieren lässt, sodass auch der Computer befähigt werden kann, die Lösung zu finden. Die Aufgabe ist aber noch nicht vollständig gelöst. Wie man leicht verifiziert, kann der Vorgestellte auch der Onkel des Vorstellenden sein. Die Lösung `va(er,ich)` ist nur dann mit Sicherheit richtig, wenn `er` keinen Bruder hat, d.h. unter der Voraussetzung, dass das benutzte Faktenwissen alle relevanten Personen umfasst und insofern *vollständig* ist. Falls `er` einen Bruder hat, muss dieser als fünfte Person

(abgekürzt `p5`) im Faktenwissen vertreten sein. Ausserdem wäre eine zusätzliche Regel notwendig. Die Wissensbasis müsste durch

Fakt 4: `bru(p5,er)` und

Regel 3: `mu(x,y) AND mu(x,z)` $\Rightarrow$ `bru(y,z).`

ergänzt werden.

Die Notwendigkeit, den Diskursbereich ausreichend vollständig zu erfassen, ist ein charakteristisches Problem des formalen Schlussfolgerns. Neben der Vollständigkeit muss die *Widerspruchsfreiheit* der Wissensbasis gewährleistet sein. Weder die Fakten noch die Regeln dürfen sich widersprechen. Die Wissensbasis muss in sich **konsistent** sein. Konsistenz muss auch für alle ableitbaren Aussagen (Fakten) gefordert werden. Ihre Gewährleistung kann problematisch sein, insbesondere dann, wenn im Diskursbereich Veränderungen eintreten. Neue Fakten müssen nicht unbedingt mit den alten konsistent sein oder sie können alte Inferenzen in Frage stellen. Wenn beispielsweise `er` ein zweites mal heiratet, kann aus Regel 2 nicht mehr mit Sicherheit der Schluss 2 gezogen werden.

Damit ist die Aufzählung der Probleme, die beim Inferenzieren auftreten können, nicht beendet. Im folgenden Kapitel werden einige weitere aufgezeigt. Manche der Probleme vereinfachen sich oder entfallen vollständig, wenn eine andere Strategie des Inferenzierens angewendet wird, die jeder aus dem Alltagsleben kennt, das Probieren, genauer das Verifizieren von Annahmen.

Beim Nacherfinden des maschinellen Inferenzierens hatten wir uns vom Vorgehen eines systematisch veranlagten Menschen inspirieren lassen, der zunächst relevantes Wissen, Regeln und Fakten sammelt und dann versucht, mit Hilfe des gesammelten Wissens die Frage zu beantworten. Ein Mensch, der mehr zum Probieren neigt, wird wahrscheinlich anders vorgehen. Der erste Schluss, den wohl die meisten Menschen zunächst einmal ziehen werden, die Systematiker wie die Probierer, ist der, dass die Schwiegermutter eine Generation älter ist als `er` und dass sie zwei Generationen älter ist als `ich`, dass folglich `er` eine Generation älter ist als `ich`. Das trifft z.B. für den Vater von `ich` zu. Der Probierer wird nun von der *Hypothese* ausgehen, dass `va(er,ich)` die Lösung ist und prüfen, ob in diesem Fall das Prädikat `smu(seine,meine)` erfüllt ist. Die Prüfung fällt positiv aus, womit der Probierer sich eventuell zufrieden gibt.

Das Inferenzieren aufgrund einer Hypothese und deren Prüfung wird Inferenzieren durch **Rückwärtsverkettung** oder kurz **Rückwärtsinferenz** genannt. Es wird von der Lösung *rückwärts*, d.h. in Richtung vorgegebener Fakten geschlossen. Das Vorgehen ohne Hypothese, bei dem, ausgehend von den gegebenen Fakten *vorwärts* (in Richtung Lösung) geschlossen wird, heißt Inferenzieren durch **Vorwärtsverkettung** oder kurz **Vorwärtsinferenz**.

Im Falle vieler Fakten und weniger Hypothesen ist Rückwärtsinferenz offensichtlich einfacher zu realisieren und zu implementieren als Vorwärtsinferenz, denn der Suchraum wird durch Hypothesen erheblich eingeschränkt. Darum ist es verständ-

lich, dass viele Inferenziersysteme mit Rückwärtsverkettung arbeiten. Das gilt beispielsweise für Prolog-Systeme (siehe Bild 16.3).

Wir kehren noch einmal zu unserem Probierer zurück. Wenn er gründlich ist, wird er weitersuchen und andere Hypothesen prüfen. Die naheliegendste Hypothese ist `onk(er,ich)`. Die Prüfung zeigt, dass auch sie mit Fakt 1 vereinbar ist. Damit lautet die Lösung `va(er,ich) ODER onk(er,ich)`. Ein Prologsystem würde die Hypothesen (Anfrage) `?va(er,ich)` und `?onk(er,ich)` bestätigen. Die Hypothese `?bru(er,ich)` würde es ablehnen.

Damit ist die Frage, ob es noch andere Personen geben könnte, für die Fakt 1 zutrifft, immer noch nicht aus der Welt geschafft. Zum Nachweis der **Vollständigkeit der Antwort** kann wiederum die Rückwärtsverkettung angewendet werden. Dabei ist zu beweisen, dass die Annahme irgendeiner anderen verwandtschaftlichen Beziehung, also irgendeines anderen Prädikats `P(er,ich)` zu einem Widerspruch mit Fakt 1 führt. Dazu reicht das Suchen und die Wertzuweisung als "*Bausteinoperationen* des Inferenzierens" nicht aus. Um den Widerspruch herzuleiten, bedarf es zum einen eines umfangreicheren Regelwissens, und zum anderen müssen eventuell Prädikate transformiert werden, m.a.W. es müssen *Formelmanipulationen*, konkret *Prädikattranformationen* gemäß den verfügbaren Regeln durchgeführt werden, bis man zu einem Widerspruch gelangt, also zu einem Prädikat, das stets falsch ist.

Beim Inferenzieren wechseln sich *Suchen* von Wissen mit *Anwenden* von Wissen ab. Gesucht wird nach *anwendbarem* Wissen (Regeln, Fakten). Wissen anwenden ist stets ein *Substituieren*. Im einfachsten Fall werden Variable durch Konstante substituiert (Wertzuweisung). Es können auch Variable durch Variable oder Prädikate durch Prädikate substituiert, d.h. Prädikate oder Prädikatverbindungen (Klauseln) *transformiert* werden. Die Transformation (Formelmanipulation) stellt also - ebenso wie die Wertzuweisung - eine spezielle Art der Substitution dar (vgl. die Überlegungen am Ende des Kapitels 15.8.). In jedem Fall setzt Substitution die *syntaktische Übereinsimmung* der involvierten Prädikate voraus. Der Syntaxvergleich ist Bestandteil des Suchprozesses.

Das Substituieren beim Inferenzieren erfolgt ganz analog zum Substituieren beim analytischen Rechnen, wie es in Kap.15.8 dargelegt wurde. Dort bestand die Anwendung einer Formel (z.B. aus einer Formelsammlung) auf einen gegebenen Ausdruck aus zwei Schritten, dem *Bezeichnerabgleich* und der *Substitution*, in völliger Analogie zum Inferenzieren. Nur wird der Bezeichnerabgleich in Falle des Inferenzierens **Unifikation** genannt. Damit setzt sich ein Inferenzprozess aus drei *Bausteinoperationen* zusammen:

- Suche (einschließlich Syntaxvergleich),
- Unifikation,
- Substitution.

In der Sprechweise der USB-Methode können wir also abschließend feststellen: ***Inferenzieren*** *ist eine Kompositoperation aus Such-, Unifikations- und Substitutionsoperationen.*

16.2 Wissensverarbeitung

Im vorangehenden Kapitel haben wir das formale Schlussfolgern anhand einer Denksportaufgabe nacherfunden. Das ist ein *theoretischer* Erfolg. Die *praktische* Bedeutung der "Erfindung" hängt davon ab, wieweit sich das Vorgehen verallgemeinern und in den verschiedensten Situation anwenden lässt, in denen mit Wissen "hantiert", Wissen "verarbeitet" wird. Aus dieser Sicht drängt sich eine etwas andere, allgemeinere Frage auf: *Lässt sich Wissensverarbeitung automatisieren*? Zur Wissensverarbeitung gehören

- **Wissenserwerb**, d.h. das Sammeln und Abspeichern von Wissen, auch *Wissensakquisition* genannt,
- **Wissenszugriff**, d.h. das *Wiederauffinden* von abgespeichertem Wissen,
- **Inferenzieren**, d.h. das *Produzieren* von abgeleitetem aus dem gespeicherten Wissen.

Ein **Wissensverarbeitungssystem** muss über Programme für alle drei Tätigkeiten verfügen, vom Sammeln abgesehen, obwohl auch das in gewissen Grenzen automatisiert werden kann. Wenn die Fähigkeit zum Inferenzieren fehlt, spricht man i.Allg. von **Datenbanksystem**. Ebenso wie ein Mensch, kann auch ein Wissensverarbeitungssystem nicht *alles* wissen. Wenn seine Wissensbasis auf ein bestimmtes Gebiet spezialisiert ist, wird es **Expertensystem** für dieses Gebiet genannt. Ein Expertensystem kann nur dann die Rolle eines echten Experten übernehmen, wenn es über

- ein ausreichend umfangreiches und schnell und zuverlässig abrufbares Wissen über den Diskursbereich,
- ein ausreichend vollständiges und effizientes Inferenzierprogramm und
- eine geeignete Dialogsprache

verfügt. Wir wollen uns überlegen, ob bzw. wie diese Bedingungen erfüllt werden können.

Vollständigkeit des Inferenzierers

Wir beginnen mit der zweiten Bedingung und fragen, ob bzw. wieweit sich das in Kap.16.1 beschriebene Schlussverfahren verallgemeinern lässt, ob sich vielleicht sogar ein universeller Inferenzoperator, ein "*universeller Inferenzierer*" (in Analogie zum universellen Rechner) realisieren lässt, mit anderen Worten, ob ein Programm geschrieben werden kann, das in der Lage ist, aus jedem beliebigen vorgegebenen Regel- und Faktenwissen "alle nur möglichen" Schlüsse zu ziehen.

Man könnte den Eindruck haben, als würden wir in eine ähnlich "bodenlose" Frage verstrickt, wie es die Frage nach der Berechenbarkeit von Funktionen war.

Denn Inferenzieren ist de facto nichts anderes als das Berechnen von Funktionswerten, wobei die Funktionen durch *Prädikate* [8.17] festgelegt sind und die Funktionswerte *Merkmalswerte von Individuenvariablen* darstellen.

Dennoch ist die Frage nach dem “universellen Inferenzierer” nicht “bodenlos”, denn wir befinden uns bereits auf “formalem Boden”, auf dem Boden des Prädikatenkalküls, und die Frage kann formal beantwortet werden, wenn man davon ausgeht, dass Inferenzieren eine Kompositoperation aus den Bausteinoperationen *Suchen, Unifizieren* und *Substituieren* ist. Man kann auch auf den Algorithmenbegriff (genauer auf den Begriff des *imperativen* Algorithmus) zurückgreifen und definieren:

Ein ***Inferenzieralgorithmus*** *ist eine Vorschrift zur Ableitung von Konklusionen aus Prämissen in endlich vielen Such-, Unifizierungs- und Substitutionsschritten.* Durch Artikulierung des Algorithmus in einer Programmiersprache und Implementierung wird der Computer zu einem (realen) Inferenzierer.

Damit ist der Inferezierer formal definiert, und seine “Universalität” kann formal (mit den Mitteln des Prädikatenkalküls) untersucht werden, wobei an die Stelle des unscharfen Begriffs der Universalität der scharfe Begriff der **Vollständigkeit** tritt. *Ein Inferenzierer ist* ***vollständig*** *hinsichtlich einer gegebenen Wissensbasis, wenn er aus den Fakten der Wissensbasis (den Prämissen)* ***alle*** *wahren Konklusionen (neue Fakten) ableiten kann.*

Der Nachweis, ob ein Inferenzierer vollständig ist oder nicht, stellt ein theoretisch anspruchsvolles Problem dar, auf das wir nicht näher eingehen werden. Wir begnügen uns mit dem Verweis auf die angegebene Literatur (insbesondere auf [Russell 95]) und mit der Erwähnung einiger charakteristischer Schwierigkeiten theoretischer Natur.

Auf zwei Probleme wurde bereits aufmerksam gemacht, das Problem der Vollständigkeit eines gegebenen Faktenwissens hinsichtlich konkreter Anfragen und das Problem der Widerspruchsfreiheit oder Konsistenz der Wissensbasis. Darüber hinaus können alle diejenigen Probleme auftreten, mit denen wir beim analytischen Rechnen konfrontiert wurden. Denn Inferenzieren ist - ebenso wie das analytische Rechnen - ein Rechnen mit Variablen, jedoch nicht nach den Regeln der Algebra oder Analysis, sondern nach den Regeln des Prädikatenkalküls. Hinsichtlich der grundsätzlichen Vorgehensweise sowie der auftretenden Schwierigkeiten wäre hier alles zu wiederholen, was in Kap.15.8 gesagt worden ist, nur treten an die Stelle der Formeln der Algebra und Analysis die Formeln des Prädikatenkalküls. Zu den Einzelheiten dieser Art des Rechnens muss auf die Literatur verwiesen werden[6]. Wir überlassen es dem Leser, sich noch einmal die Gedankengänge und Aussagen von Kap.15.8 zu vergegenwärtigen und auf das Inferenzieren zu übertragen.

Es besteht stets die Möglichkeit, dass ein Inferenzprozess nicht terminiert. Dann gibt es keinen “Schluss” in zweifachem Sinne, es kann kein “Schluss” *gezogen* (keine

---

6 Z.B. [Russell 95],[Schöning 89].

Antwort gegeben) werden, weil das Programm zu keinem “Schluss” (Ende) *kommt*. Das kann verschiedene Ursachen haben. Beispielsweise kann der Inferenzierer in einen unendlichen Suchzyklus geraten oder er kann auf ein Prädikat stoßen, das nicht entscheidbar ist, z.B. weil es eine *widersprüchliche Zirkularität* enthält oder weil es ein *Wahrsageprädikat* ist.

Das Versagen eines Inferenzierers muss also durchaus nicht die Folge von Mängeln in der Hard- oder Software sein oder eine Folge prinzipieller Grenzen der Intelligenz des Computers, sondern es kann ein Mangel des sprachlichen Modellierens an sich vorliegen. Zwar dient Sprache der Modellierung der Realität, also dessen, was *der Fall*, was *wahr* ist. Doch lässt Sprache beliebig freien Raum für das Artikulieren unwahrer, widersprüchlicher, sinnloser oder nichtentscheidbarer Aussagen. Mit den Ursachen und Folgen hatten wir uns in Kap.6 auseinandergesetzt. Beim sprachlichen Modellieren durch den Rechner kann die Folge darin bestehen, dass ein Programm nicht terminiert.

Effizienz des Inferenzierens

Es gibt heute leistungsfähige Inferenzprogramme. Praktisch für jeden Beruf, in welchem regelbasiertes Schlussfolgern eine Rolle spielt, kann ein Expertensystem erstellt werden, das einen in dem Beruf Tätigen unterstützen kann, vorausgesetzt, es verfügt über die erforderliche Wissensbasis (sprich: Fachwissen). Auf dem Wege zu diesem Erfolg waren viele Schwierigkeiten zu überwinden, von denen einige bereits genannt wurden. Die Aufgabe, mit der die Informatiker konfrontiert waren, als sie versuchten, das menschliche Schlussfolgern zu simulieren, ist hinsichtlich der Höhe des Anspruchs vergleichbar mit der Aufgabe, numerisches Rechnen zu simulieren, d.h. *Rechen*maschinen im ursprünglichen Sinne des Wortes zu bauen.

Das numerische Rechnen hat uns den gesamten zweiten Teil bis einschließlich Kapitel 15.6 beschäftigt. Der Weg zu einer technisch brauchbaren Lösung führte über die Normalisierung von Ausdrücken und die Standardisierung der Bausteinoperationen. Ausgangspunkt waren die elementaren booleschen Operatoren, aus denen Kompositoperatoren komponiert wurden. Eine zentrale Rolle spielte die KNDF, die kanonische disjunktive *Normal*form und ihre mikroelektronische Realisierung, die zum *Standard*baustein der verschiedensten Operatoren höherer Komponierungsstufe wurde, insbesondere verschiedener Matrizenschaltkreise.

In der technischen Handhabung des Prädikatenkalküls und in der “Technologie” des Inferenzierens spielt die Hornklausel eine ähnliche “normalisierende” Rolle wie die KNDF in der Technologie der booleschen Algebra und des numerischen Rechnens. Auch die Hornklausel ist aus booleschen Operatoren komponiert, doch deren Operanden sind keine Aussagen über Konstanten, keine Fakten, sondern Aussagen über Variablen.

Durch die Normalisierung der Ausdrücke zu *Implikationen* in der Form von Hornklauseln wird die *Standardisierung* des Inferenzierens zum sogenannten **Resolutionsverfahren** möglich. Es besteht aus einzelnen Schritten. Bild 16.2 stellt einen

konkreten Inferenzprozess nach dem Resolutionsverfahren graphisch dar. Die Ableitung eines Knotenprädikats heißt **Resolution**, das Ergebnis des letzten Inferenzschrittes (der letzten Resolution) heißt **Resolvente**. Es lässt sich Folgendes zeigen: *Das Resolutionsverfahren ist hinsichtlich falscher Prädikate stets vollständig*. Damit kann die Wahrheit jedes Prädikats dadurch nachgewiesen werden, dass aus seiner Negation ein Widerspruch, also die Resolvente `wahr⇒falsch` folgt.

Das Ziel ist erreicht. Wir haben erkannt, wie der Computer schlussfolgern kann. Doch aus dem Schlussfolgern ist durch Kalkülisierung des Problems Rechnen geworden. Der beschriebene Lösungsprozess des Verwandtschaftsproblems ist Deduzieren im Rahmen eines (nichtaxiomatisierten) Kalküls, der durch Sprachvereinbarungen (die sich an das Prädikatenkalkül anlehnen) und zwei Regeln definiert ist. Das bedeutet nicht, dass auch der Rätselrater, der vom Prädikatenkalkül nichts weiß, die Lösung des Verwandtschaftsproblems durch Rechnen findet. Für ihn ist das Problem nichtmathematischer Natur.

Es entspricht dem gängigen Sprachgebrauch, bei der kalkülisierten Beschreibung eines Sachverhaltes von *mathematischer* Modellierung zu sprechen, unabhängig davon, ob der Kalkül axiomatisiert ist oder nicht. Der Begriff des mathematischen Modells ist an die Existenz eines Kalküls gebunden, doch muss der Kalkül nicht unbedingt axiomatisiert sein. Die Verwendung eines nichtaxiomatisierten Kalküls birgt die Gefahr in sich, dass eine versuchte Ableitung ein falsches oder widersprüchliches Resultat oder auch gar kein Resultat liefert. Ein Blick in die Geschichte der Naturwissenschaft zeigt, dass zur Beschreibung neuer Phänomene i.Allg. zunächst nichtaxiomatische Theorien entwickelt und erfolgreich angewendet werden, deren Axiomatisierung oft erst viel später gelingt. (Man erinnere sich: Eine physikalische Theorie ist die Interpretation eines Kalküls durch Objekte der Wirklichkeit). Die Entwicklung der klassischen Mechanik oder der Quantenmechanik sind Beispiele hierfür.

Wenn das Wort “mathematisch” nicht unbedingt “axiomatisch”, unbedingt aber “kalkülisiert” beinhaltet, ist es gerechtfertigt, das Schlussfolgern insoweit dem Bereich der Mathematik zuzuordnen, wie es kalkülisiert ist, d.h. wie es Inferenzieren ist. Unter dieser Bedingung wird Schlussfolgern zu einer mathematischen Operation. Ungerechtfertigt dagegen wäre es, das alltägliche Schlussfolgern mathematisch zu nennen, es sei denn, es verwendet *bewusst* einen Kalkül. Der Schluss vom Blitz auf den bevorstehenden Donner verlangt keine Mathematik, weder jetzt, noch vor hunderttausend Jahren. In diesem Zeitmaßstab gemessen ist bewusstes Kalkülisieren und Inferenzieren sicher ein ziemlich junges Produkt der kulturellen Evolution.

Anfragesprache

Expertensysteme wären nutzlos, wenn ein Anwender, der den Computer eine Inferenz ausführen lassen will, dafür selber ein Inferenzprogramm schreiben müsste. Diese Arbeit nimmt ihm das System ab, richtiger der Systemprogrammierer. Der Wert von Expertensystemen und der Wert der KI-Technologie ganz allgemein

besteht gerade darin, dass der Nutzer lediglich das für sein Problem spezifische Wissen und seine Fragen oder Hypothesen in einer geeigneten Programmiersprache zu artikulieren braucht. Diese Sprache nennen wir **Anfragesprache**.

Anfragesprachen sind i.d.R. nicht dafür geeignet, eine Berechnungsvorschrift zu artikulieren. Vielmehr dienen sie der *Beschreibung der Bedingungen*, unter denen Schlussfolgerungen gezogen (Vorschläge gemacht, Expertisen erstellt) werden sollen. In diesem Sinne spricht man von **deskriptiven** oder **deklarativen** Sprachen, im Gegensatz zu den **prozeduralen** Sprachen, die der Artikulierung von Operationsvorschriften und damit unmittelbar der Steuerung von Prozessen dienen. Die Bezeichnungen *prozedural* und *deklarativ* werden auch auf Programme angewendet. Ein deklaratives Programm artikuliert Prädikate (Eigenschaften und Relationen) und *deklariert* sie als *wahr*. Danach bietet sich als Anfragesprache die Sprache des Prädikatenkalküls an, denn sie ist eine sehr allgemeine formale Sprache, die mit Prädikaten hantiert. Bei der Lösung der Denksportaufgabe in Kap.16.1 haben wir sie ohne weitere Begründung verwendet.

Auch die Sprache Prolog ist eine deklarative Sprache, die sich an die Sprache der Prädikatenlogik anlehnt. Prolog ist als "PROgrammieren in LOgik" zu lesen, oder genauer "Programmieren in Prädikatenlogik". Sie unterstützt das sog. *logische Programmieren*. In den Kapiteln 18 und 20 werden nähere Ausführungen über logische Sprachen gemacht. Prolog ist als Anfragesprache für Inferenziersysteme geeignet und als solche konzipiert. Bild 16.3 zeigt ein Prolog-Programm zum Lösen des Verwandtschaftsproblems.

```
((ehe y z)(smu x y)(mu x z))
((va w v)(mu mu u v)(ehe u w))
((smu seine meine))
((mu meine ich))
((mu seine er))
? (va er ich)
```

Bild 16.3 Prolog-Programm zum Lösen des Verwandtschaftsproblems.

In den letzten drei Programmzeilen, vor der eigentlichen Anfrage, die mit einem Fragezeichen beginnt, wird der Leser das Faktenwissen erkennen (oben als Fakt 1, Fakt 2 und Fakt 3 bezeichnet), wobei anstelle der Präfixnotation die Listennotation verwendet ist. Die ersten beiden Zeilen stellen das Regelwissen dar. Sie entsprechen den Regeln 1 und 2, die wir bei der Lösung des Verwandschaftsproblems verwendet haben; es sind *Hornklauseln*, allerdings in einer speziellen Syntax notiert. Der jeweils erste Klammerausdruck, `(ehe y z)` bzw. `(va w v)`, ist die Konklusion, die nachfolgenden Klammerausdrücke sind die Prämissen (Bausteinprämissen). Der Implikationspfeil ist unterdrückt. Wäre er notiert, müsste er nach links gerichtet sein.

Das Programm demonstriert die Nutzerfreundlichkeit der Sprache und die Leistungsfähigkeit ihres Interpreters. Die letzte Zeile ist eine Hypothese, die der Programmierer vorschlägt. Sie kann bestätigt oder abgelehnt werden. Das Inferenzieren erfolgt also nach der Methode der Rückwärtsverkettung.

Die Frage nach der Sprache, in welcher der Computer seine Ausgaben "*artikuliert*", stand und steht nicht zur Debatte, weil sie selten definiert ist, jedenfalls nicht intensional durch Vorgabe von Syntaxregeln, sondern höchstens extensional durch Aufzählung konkreter Ausgabezeichenketten, die der Systementwickler vorprogrammiert und der Computer gegebenenfalls ergänzt. Man erinnere sich an die Ausgabe des Simulationssystems an den Experimentator hinsichtlich der Fallgeschwindigkeit der Kugel in Kap.15.5 [15.7]. Normalerweise gibt es also gar keine syntaktisch definierte "Dialog"-Sprache. Die einzige Forderung an die Ausgabe ist die leichte Interpretierbarkeit durch den Nutzer, das leichte Anbinden der *Nutzersemantik* an die Ausgabezeichen.

Die Bedeutung des Prädikatenkalküls ist nicht auf die Anfragesprache beschränkt, sondern jede Art von Wissensverarbeitung verwendet mehr oder weniger explizit den Prädikatenkalkül. Das ist verständlich, denn die Aufgabe des Computers besteht darin, *richtige Aussagen über die Welt zu produzieren*, also erfüllte Prädikate. Der Computer "weiß" davon nichts, aber Systementwickler und Nutzer wissen es und der Nutzer eines Computers interpretiert dessen Ausgaben als erfüllte Prädikate, als wahre Aussagen über die Welt. Eben das entspricht dem Anliegen des *technischen aktiven sprachlichen Modellierens*.

Diesem Ziel dienten alle unsere Überlegungen. In Kap.15 haben wir im Einzelnen verfolgt, wie der Computer das sprachliche Modellieren im Bereich der exakten Naturwissenschaften, insbesondere der Physik, also das *mathematische* Modellieren unterstützen kann. Dort werden die Aussagen über die Welt formalisiert, indem sie in der Sprache eines mathematischen Kalküls formuliert werden, sodass sie als Interpretationen des Kalküls aufgefasst werden können. Durch das Implementieren physikalischer oder anderer formaler Theorien wird es möglich, neue richtige Aussagen nach den Regeln eines Kalküls abzuleiten, und zwar durch numerisches oder analytisches Rechnen. Die *Wissensverarbeitung*, d.h. das "nichtmathematische" Modellieren durch den Computer, verfährt nach dem gleichen Rezept, nur verwendet sie den Prädikatenkalkül (die Prädikaten*logik*), sie "rechnet logisch". Im Sinne unserer Definition des analytischen Rechnens [15.8] [15.14] führt die Wissensverarbeitung *analytische Rechnungen* durch.

An dieser Stelle ist ein kurzer Rückblick angebracht. Wenn man sich noch einmal die Bilder 5.3 und 8.6 und die Überlegungen zum technischen Semantikproblem [15.9] und zur semantischen Bindung von Operanden vergegenwärtigt, erkennt man, dass die Einführung der obigen Bezeichnungen für Verwandte und für Verwandtschaftsrelationen dem ersten Schritt der semantischen Bindung, nämlich der Anbindung der *externen* Semantik (der "Verwandtschaftssemantik") an die *formale* Semantik des Kalküls dient. Für die Richtigkeit dieses *ersten* Schrittes, also dafür, dass

die externe Semantik logisch richtig und in ausreichendem Umfang an die Kalkülsemantik angebunden wird, ist derjenige verantwortlich, der das Regel- und Faktenwissen sowie die Anfrage artikuliert, wofür zweckmäßigerweise eine implementierte Programmiersprache verwendet wird, die eine effiziente Artikulation verwandtschaftlicher Beziehungen gestattet, beispielsweise Prolog. Für den *zweiten* Schritt, für die Anbindung der Kalkülsemantik an die Maschinensemantik sind der Entwickler der verwendeten Sprache und ihr Implementierer, der das Übersetzerprogramm schreibt, verantwortlich.

In beiden Schritten können Fehler unterlaufen. Sie sind i.d.R. schwer zu erkennen. Oft machen sie sich erst in offensichtlich falschen Schlussfolgerungen bemerkbar. Die Richtigkeit der Schlussfolgerungen, die der Computer bei der Lösung des Verwandtschaftsproblems durch *abstraktes* Schlussfolgern (Inferenzieren) zieht (siehe Bild 16.2), werden viele Leser durch "*konkretes*" Schlussfolgern anhand der *externen* Semantik verifiziert haben. Um dem Leser das zu ermöglichen, wurde ein Beispiel "aus dem Leben" mit durchschaubarer externer Semantik gewählt. Die Schlussfolgerungen können aber, wie gezeigt, auch rein formal, also ausschließlich aufgrund der formalen Semantik, d.h. nach den Regeln des Prädikatenkalküls gezogen werden. Das erfordert jedoch ein anders geartetes, abstrakteres Denken, das dem Nichtmathematiker in der Regel ungewohnt ist.

Wissenszugriff

Das Zugreifen auf gespeichertes Wissen (das Wiederauffinden von Wissen) hätte aus historischen Gründen *vor* dem Inferenzieren behandelt werden müssen, aber auch aus systematischen Gründen, denn es ist Bestandteil des Inferenzierens; nach einem "passenden" Fakt oder nach einer passenden Regel muss eventuell in einer umfangreichen Wissensbasis gesucht werden. Dabei hilft die **Datenbanktechnologie** , auf die ein kurzer Blick geworfen werden soll.

Im Falle der *adressierten* Speicherung wird das Abspeichern und Wiederfinden mit Hilfe von Speicherplatzadressen realisiert. Die üblichen Datenbanken arbeiten nach diesem Prinzip. Wie der adressierte Zugriff hardwaremäßig realisiert wird, ist uns bekannt. Die technischen Einzelheiten sind in Kap.13.2 besprochen worden. Das Problem, das jetzt vor uns steht, liegt in der Anbindung der (externen) Nutzersemantik an die (interne) Maschinensemantik. Welche Prozesse müssen im Computer ablaufen, wenn beispielsweise ein Nutzer nach der Telefonnummer eines Bekannten fragt oder nach den Zugverbindungen von Berlin nach Wien?

4

Nach altbewährter Methode könnte man versuchen, das Problem durch Introspektion zu lösen. Bei der Simulation des Kopfrechnens sind wir so vorgegangen. Das war insofern möglich, als man selber weiß, wie man beim Rechnen verfährt. Aber was genau "*tut* man" (was passiert im Kopf), wenn man sich etwas merkt oder sich an etwas erinnert? Die Methode scheint zu versagen. In Kap.7.2 [7.9] hatten wir gesagt "*Gedächtnisleistungen sind Leistungen der reproduktiven, intuitiven Intelli-*

genz" und in Kap.7.1 [7.7] "*Intuition beruht auf nichtbewusster Verarbeitung von nichtbewusstem Wissen*".

Danach scheint uns die Introspektion nicht weiterhelfen zu können. Dennoch kann sie es. Denn wenn man sich an etwas nicht erinnern kann, es aber dennoch versucht, beginnt ein Suchprozess, dessen man sich mehr oder weniger deutlich bewusst ist. Offenbar handelt es sich um ein Weitertasten ("Navigieren") in einem "Meer" von Erinnerungen oder - unter Verwendung früher eingeführter Begriffe - um das Verketten von *Denkobjekten* mittels *Assoziation*.

In Kap.5.5 [5.19] hatten wir Assoziieren (als Methode des Zugreifens auf Gedächtnisinhalte) folgendermaßen definiert (der besseren Lesbarkeit halber ist überall "Merkmalswert" durch "Merkmal" ersetzt):

Assoziieren ist entweder
- das Zugreifen auf Objekte mittels eines oder mehrerer ihrer Merkmale oder
- das Zugreifen auf ein oder mehrere Merkmale mittels eines Objekts, das diese Merkmale besitzt, oder
- eine Kombination dieser beiden Zugriffsmethoden.

Eine Kombination liegt z.B. vor, wenn mit einem Merkmal eines Objekts ein anderes Merkmal desselben Objekts assoziiert wird oder wenn mit einem Objekt ein anderes Objekt über ein gemeinsames Merkmal assoziiert wird. Die zuletzt genannte Art des Assoziierens liegt der Verwendung *mnemotechnischer Hilfsmittel* zugrunde, wobei das gemeinsame Merkmal (oft eine gemeinsame Silbe in den Bezeichnern) die "Eselsbrücke" zum gesuchten Objekt bildet (vgl. auch die Beispiele in Kap.5.5 [5.19]).

In den zitierten Sätzen steht "Objekt" stets für "*Denkobjekt*". Erinnert man sich nun noch an die Begriffsbestimmung des Denkobjekts als Tupel von Merkmalswerten und bedenkt, dass das Faktenwissen hinsichtlich eines Objekts ein Merkmalswertetupel ist, erkennt man einen Weg zum maschinellen Erinnern. Es ist ein Mechanismus zu implementieren, der es erlaubt, in einer großen Menge von Merkmalswerten bestimmte (gesuchte) Merkmalswerte mit Hilfe anderer (vorgegebener) Merkmalswerte aufzufinden.

5 Zu diesem Zweck wird für das Merkmalswertetupel eines jeden Objekts des Diskursbereichs ein Speicherbereich vorgesehen, der in Felder unterteilt ist, für jedes Merkmal ein Feld. *Ein gespeichertes Tupel wird* ***Datensatz*** *genannt. Ein Datensatz enthält für jedes Merkmal eines Merkmalstupels ein* ***Feld****. Die Datensätze der verschiedenen Objekte können in einer* ***Datei*** *verbunden werden. Die Zusammenfassung einer oder mehrerer, eventuell auch sehr vieler Dateien und eines sog.* ***Datenbankbetriebssystem*** *wird* ***Datenbank*** *genannt. Die Dateien einer Datenbank bilden die sog.* ***Datenbasis*** *der Datenbank.* Die Datenbasis ist also die Gesamtheit aller Daten, die in einer Datenbank abgespeischert sind. Die Daten einer Datenbank lassen sich auch nach anderen Gesichtspunkten zu Dateien verbinden. Beispielsweise können die Inhalte (Werte) eines bestimmten Feldes (Merkmals) aller Datensätze (Objekte) in einer Datei zusammengefasst werden, und es kann für jedes Merkmal

eine spezielle Datei eingerichtet werden. Die Struktur der Datenbasis einerseits und die Strategien des Speicherns und Suchens andrerseits müssen so aneinander angepasst sein, dass sich minimale Speicher- und Suchzeiten ergeben. Das Speichern und Suchen ist eine der Aufgaben des Datenbankbetriebssystems.

Das Suchen in einer Datei soll am Beispiel eines “maschinellen Telefonbuchs”, einer “*Abonnentendatei*” demonstriert werden. Die Datei bestehe aus Datensätzen, die den Eintragungen eines *gedruckten* Telefonbuchs zu einem bestimmten Abonnenten entsprechen. Um eine Rufnummer in einem gedruckten Telefonbuch zu finden, sucht man nach dem Familiennamen, meistens zusätzlich nach dem Vornamen und, wenn notwendig, auch noch nach der Straße oder nach dem Beruf. Diese Suchmethode kann auf die Abonnentendatei übertragen werden. Ein Merkmalswert bzw. ein Teiltupel, das einem Datensatz eindeutig zugeordnet ist (das nur ein einziges mal vorkommt), heißt **Schlüssel**. Um auf einen Datensatz (und damit auf die einzelnen Felder) über seinen Schlüssel zugreifen zu können, muss ein “Adressbuch” implementiert sein, das jedem Schlüssel die Adresse des betreffenden Datensatzes zuordnet. Will man die Telefonnummer eines Teilnehmers erfahren, muss man dessen Schlüssel eingeben. Daraufhin sucht das Datenbankbetriebssystem im Adressbuch nach dem Schlüssel. Wenn es ihn findet, kennt es die Adresse des betreffenden Datensatzes und kann die gewünschte Telefonnummer auslesen. Als Außenstehender kann man den Eindruck gewinnen, als “*assoziiere*” der Computer wie der Mensch mit dem Schlüssel (z.B. dem Namen) die Telefonnummer.

Durch die Wörter *Assoziieren* und *Schlüssel* wird der Leser vielleicht an Kap.13.2.2 [13.3] erinnert. Dort war vom *Assoziativspeicher* die Rede und vom *Schlüsselwort*, das einen Teil eines Speicherplatzinhalts bildet und gleichzeitig als “Adresse” (*Suchargument*) des Speicherplatzes dient. Dort war auch schon auf die Analogie des “*assoziativen Zugriffs*” in einem Assoziativspeicher einerseits und in einer Datenbank andrerseits hingewiesen worden. Die Schlüsselwörter in einer Datei (Datenbank) spielen dieselbe Rolle wie die Suchargumente in einem Assoziativspeicher oder wie der gemeinsam mit einer Nachricht verschickte Schlüssel bei Anwendung des Schlüssel-Schloss-Prinzips [12.4].

6

Wie nahe das skizzierte Zugriffsprinzip über Merkmalswerte dem menschlichen Erinnerungsvermögen kommt, wird durch folgendes Beispiel verdeutlicht. Auf einem Spaziergang im August sieht man in der Ferne ein Kornfeld mit blauen Farbflecken. Mit den Merkmalswerten Farbe, Standort und Jahreszeit wird ein Naturliebhaber wahrscheinlich sofort (ohne nachzudenken) Kornblumen assoziieren. Hat er noch nie eine Kornblume gesehen, kann er sie mit Hilfe eines Blumenerkennungsbuches bestimmen, indem er nach den genannten Merkmalen sucht. Das Vorgehen lässt sich simulieren. Für jede Blume wird ein Datensatz abgespeichert. Das Anfragetupel (blau, Acker, August) wird der Computer wahrscheinlich mit mehreren Namen beantworten, z.B. “Ackerskabiose, Wegwarte, Kornblume”. Für eine eindeutige Antwort bedarf es weiterer Merkmale, was evtl. schwieriger zu verwirklichen ist, am anschaulichsten mit Hilfe einer Zeichnung.

Das Blumenbeispiel hat vielleicht den einen oder anderen Leser auf eine weitere Idee gebracht, das Vorgehen des Menschen zu simulieren und zwar das "Heranpirschen" an das gesuchte Objekt durch immer genauere Beschreibung, durch *Präzisierung*, d.h. durch Absteigen in einer Klassenhierarchie (siehe Bild 5.4). Zunächst wird die Klasse "Blau" aufgeschlagen. In ihr wird nach der Unterklasse "Acker" und schließlich nach "August" gesucht. In jedem Präzisierungsschritt wird ein weiteres Merkmal in die Suche einbezogen, sodass der *Suchraum* immer enger begrenzt wird.

7

Diese Suchmethode setzt eine *begriffliche Strukturierung* der Wissensbasis voraus, genauer eine *klassifikatorische* Strukturierung. Grundlage sind die begriffsbildenden Operationen *Klassifizieren* und *Generalisieren*. Hier zeigt sich, dass Begriffsbildung nicht nur der Erhöhung der Ausdruckskraft (der *semantischen Verdichtung*) der Sprache dient, sondern auch dem Sicherinnern, dem Navigieren im eigenen Wissen. Es stellt sich die Frage, ob sich dafür noch andere begriffsbildende Operationen eignen.

Ein Blick auf Bild 5.4 zeigt, das dies für das *Komponieren* (*Aggregieren*) offensichtlich der Fall ist. Beispielsweise kann bei der Nutzung der Wissensbasis eines Fertigungsbetriebes das *dekomponierende* Suchen angezeigt sein. Bei der Suche nach dem Geburtstag eines Mitarbeiters könnte in der Verwaltungshierarchie des Betriebes "abgestiegen" werden, bei der Suche nach den Maßen einer Welle eines Motors in der Hierarchie der Bauelemente der Fertigungsprodukte.

Diese wenigen Hinweise sind im Grunde ausreichend, um vieles von dem nachzuerfinden, was auf dem Gebiete der Datenbanktechnologie entwickelt worden ist[7]. Allerdings darf nicht der Eindruck entstehen, dass die Datenbanktechnologie auf das Wiederauffinden gespeicherter Daten beschränkt ist. Sie umfasst weit mehr, denn eine Datenbank besteht aus der **Datenbasis** und dem **Datenbankbetriebssystem**. Letzteres ist die zum *Betreiben*, d.h. zur Nutzung und Wartung der Datenbasis erforderliche Software. Die *Datenbasis* ist die Gesamtheit aller Daten, also sämtlicher gespeicherten Merkmalswerte. Zu ihrer Wartung gehört u.a. das Erweitern und Aktualisieren der Datenbasis sowie deren Konsistenzerhaltung.

So eindruckvoll der Stand der Technik auch ist, mit der Leistungsfähigkeit des menschlichen Gehirns kann sich die Technik nicht messen. Man überlege sich, wie viele Felder der Datensatz zum Denkobjekt "Erde" oder "Meine Mutter" enthalten müsste und welches Assoziationsfeld durch diese Worte berührt wird. Der volle Umfang externer Semantik passt in keinen Computer. Auf diese Grenze der KI kommen wir in Kap.17 zurück.

---

7 Aus der umfangreichen Datenbankliteratur seien [Wedekind 91], [Wedekind 89][Lockeman 93] und [Kemper 97] genannt.

16.3 Erfinden

Wir haben uns überzeugen können, dass nicht nur die natürliche, sondern auch die künstliche Intelligenz die Fähigkeit zum Schlussfolgern besitzt. Damit ist aber die Frage nach ihren Grenzen nach wie vor nicht beantwortet. Die Grenze der natürlichen Intelligenz ist jedenfalls *nicht* erreicht. Der Mensch kann nämlich nicht nur rechnen und schlussfolgern, er kann auch erfinden, m.a.W. er besitzt die Fähigkeit, neue richtige Aussagen nicht nur auf deduktivem Wege (durch Schließen oder Rechnen), sondern auch auf *intuitivem* Wege, d.h. durch *unbewusste Wissensverarbeitung* zu finden. Diese Art des Findens hatten wir **Erfinden** genannt.

Der so definierte Begriff des Erfindens schließt das Erfinden technischer Funktionsprinzipien (z.B. der Dampfmaschine) insofern ein, als diese Art des Erfindens die gedankliche Vorwegnahme der Erfindung voraussetzt, m.a.W., dass er das "Erfinden der Aussagen" voraussetzt, die der technischen Erfindung zugrunde liegen (vgl. Kap.7.1 [7.4]).

Ursprünglich schien uns die Simulation intuitiver Intelligenz ein hoffnungsloses Unterfangen zu sein. Dennoch wollen wir versuchen, das Erfinden zu simulieren. Bestärkt werden wir von unserem Erfolg, Assoziation und Erinnerungsvermögen in gewissem Umfange zu simulieren, obwohl beide vorwiegend auf *unbewusster* Wissensverarbeitung beruhen.

Wir gehen von einer konkreten Erfindung aus und überlegen uns, ob auch der Computer sie hätte machen können. Wir wählen dafür die Erfindung der Struktur des Benzolringes durch August Kekulé. Offensichtlich handelt es sich dabei um eine *Erfindung* im Sinne unserer Definition. Sicher hat Kekulé keine Entdeckung gemacht. Zwar war das Benzol etwas bereits Vorhandenes. Gesucht ist aber dessen molekulare Struktur, und die war nicht "zu sehen", sie konnte nicht experimentell "*entdeckt*" werden. Auch scheint Kekulé die gesuchte Struktur nicht abgeleitet zu haben, denn die Lösungsidee ist ihm in einem Tagtraum gekommen, den er selber folgendermaßen beschreibt[8].

"Ich drehte meinen Stuhl zum Feuer und fiel in einen Dämmerschlaf. Wieder tanzten die Atome vor meinen Augen. [...] lange Ketten, oft enger miteinander verbunden, waren alle in Bewegung, ineinander verschlungen wanden sie sich wie Schlangen. Aber Achtung, was war das? Eine der Schlangen hatte ihren eigenen Schwanz gepackt und drehte sich vor meinen Augen spöttisch im Kreis. Blitzartig schreckte ich hoch [...]."

Offenbar war der Traum die Fortsetzung des Suchens in den Halbschlaf hinein. Es stellt sich die Frage, worin der Unterschied zwischen der erfolg*losen* Suche im

8 Entnommen aus [Ortoli 98]

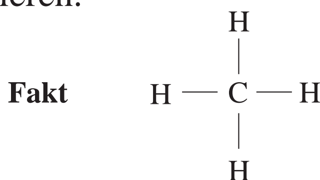
Wachen und der erfolg*reichen* Suche im Halbschlaf bestand. Eine einleuchtende Antwort auf diese Frage könnte helfen, Erfinden zu simulieren.

Im wachen Zustand hat Kekulé sicher alle aus seiner Sicht denkbaren Verkettungen von Wasserstoff- und Kohlenstoff-Atomen (H- und C-Atomen) durchprobiert, in einer Art Puzzlespiel. Wir wollen versuchen, das Puzzeln zu systematisieren und zu algorithmieren. Dazu gehen wir ganz ähnlich vor wie beim Implementieren des Schlussfolgerns. Zunächst suchen wir Fakten und Regeln, nach denen die Molekülbildung erfolgen kann. Dabei gehen wir vom chemischen Allgemeinwissen hinsichtlich unseres Problems aus, das wir in 6 Axiomen aufschreiben.

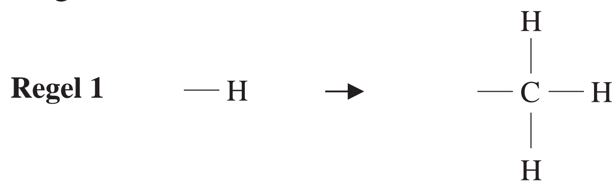
8

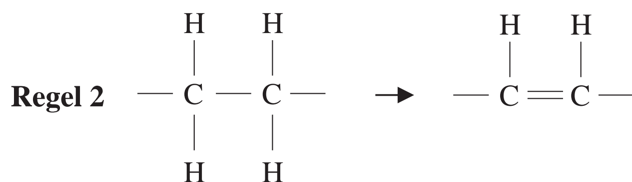
Axiom 1: H-Atome sind einwertig.
Axiom 2: C-Atome sind vierwertig.
Axiom 3: Ein C-Atom kann über jede seiner Valenzen ein H-Atom binden.
Axiom 4: Zwischen zwei C-Atomen sind Einfach- und Doppelbundungen möglich.
Axiom 5: Andere Bindungen gibt es nicht[9].
Axiom 6: Ein Molekül hat keine freien Valenzen.

Beim Suchen nach möglichen (den Axiomen nicht widersprechenden) Verbindungen wollen wir etwa so vorgehen, wie auch Kekulé aufgrund seines Wissens vorgegangen sein könnte. Wir beginnen mit einer bekannten Verbindung und verändern sie durch Substitution gemäß “vernünftiger” Regeln. Vernünftig heißt, dass die Regeln dem Wissen und den Vorstellungen der damaligen Zeit entsprechen. Als Ausgangsverbindung wählen wir das Methan (CH_4). Seine Existenz stellt einen “Fakt” dar. Wir notieren:



Als Regeln wählen wir





9 Die Möglichkeit von Dreifachbindungen wird außer Betracht gelassen, um die weiteren Überlegungen nicht unnötig zu komplizieren.

Wahrscheinlich wird auch Kekulé diese oder entsprechende Regeln bewusst oder unbewusst verwendet haben.

Ausgehend von diesem Wissen können C- und H-Atome zu linearen und verzweigten Ketten verbunden werden. Regel 1 erlaubt die Substitution eines H-Atoms durch eine CH_3-Gruppe. Regel 2 erlaubt die Substitution einer C_2H_4-Gruppe (Teilkette) durch eine C_2H_2-Gruppe. Das Puzzeln mit den Strukturformeln geht folgendermaßen vor sich. Auf das Methanmolekül lässt sich zunächst nur Regel 1 anwenden. Auf das Ergebnis (C_2H_6, Äthan) lässt sich auch Regel 2 anwenden, was C_2H_4 (Äthylen) ergibt. Auf beide Produkte lässt sich wieder Regel 1 und anschließend Regel 1 oder Regel 2 anwenden und so weiter. Durch (n-1)-malige Awendung von Regel 1 auf Methan wird die Verbindung C_nH_{n+2} "produziert". (Das Wort "produziert" ist mit Hinblick auf Kap.16.4 der Sprechweise der Theorie generativer Grammatiken entlehnt). Hinter dieser Formel verbirgt sich eine Vielfalt molekularer Strukturen. Es ist nämlich nicht gleichgültig, an welcher Stelle das H-Atom substituiert wird, an einem endständigen oder an einem nichtendständigen C-Atom einer Kohlenwasserstoffkette. Dabei können unterschiedliche, verzweigte Strukturen, entstehen, was zu unterschiedlichen chemischen Eigenschaften führen kann.

Der Generierungsalgorithmus ist also doppelt indeterministisch. Es kann zwischen verschiedenen Regeln und zwischen verschiedenen Stellen gewählt werden, an denen eine Regel angewendet werden soll, ganz analog zu den Algorithmen des analytischen Rechnens und des Inferenzierens. Das überrascht sicher nicht sehr. Offensichtlich gilt ganz allgemein: Regelbasierte Algorithmen der *Zeichenmustertransformation* mittels Substitutionsregeln sind zweifach indeterministisch, solange die Wahlmöglichkeiten nicht durch zusätzliche Vorschriften eingeschränkt werden. Im Falle der zweidimensionalen Sprache der chemischen Strukturformeln sind Kompositzeichen i.Allg. keine Zeichenketten, sondern Zeichenmuster aus den elementaren Zeichen (Buchstaben für Atome und Strecken für Valenzen).

Stillschweigend haben wir soeben die Generierung chemischer Verbindungen als *Zeichenmusterverarbeitung* aufgefasst. Wir dürfen sie als *Informationsverarbeitung* auffassen, wenn die Zeichenmuster semantisch belegt, d.h. interpretiert sind. Im Diskursbereich "Chemie" ist das der Fall; sie sind als (durch) Chemikalien interpretiert. Das gilt allerdings nicht für alle Verbindungen, die algorithmisch generierbar sind, denn viele von ihnen existieren nicht, weil sie nicht stabil sind. Stabilitätsbedingungen berücksichtigt unser Algorithmus nicht, sodass er eine teilweise nicht existente (virtuelle) Welt modelliert. Das ist, wie wir aus Kap.6.1 wissen, eine charakteristische Freiheit des sprachlichen Modellierens.

Die vorangehenden Überlegungen weisen den Weg, der zu gehen ist, um den Computer zum Puzzeln zu befähigen. Man erkennt nämlich, dass das Puzzeln in ganz ähnlicher Weise mit Formeln manipuliert, wie das analytische Rechnen in Kap.15.8 und das Inferenzieren in Kap.16.1. Das Regel- und Faktenwissen spielt dabei eine etwas andere Rolle als beim Inferenzieren. Das Faktenwissen, die Strukturformel des Methan, entspricht vielmehr dem Ansatz und das Regelwissen der Formelsammlung

beim Lösen eingekleideter Aufgaben im Mathematikunterricht [15.13]. Von der Fragestellung her entspricht das Puzzeln dem Theorembeweisen bzw. dem Inferenzieren, wenn entschieden werden soll, ob eine Hypothese zutrifft, beispielsweise die Hypothese “Der Vorgestellte ist mein Vater”.

Um den Computer zum Puzzeln zu befähigen, muss ein Formelmanipulator implementiert werden, der mit zweidimensional notierten Formeln manipuliert. Wir werden auf die Einzelheiten des “Puzzlealgorithmus” nicht eingehen, sondern nehmen an, dass er implementiert ist, und beauftragen den Computer, solange zu suchen (zu puzzeln), bis er die Verbindung C_6H_6 gefunden hat. Die Aufgabe lässt sich auch anders artikulieren, beispielsweise: Der Rechner soll *entscheiden*, ob die Zeichenkette C_6H_6 durch den Algorithmus *generierbar* ist. Wir geben drei weitere Artikulierungen dieser Frage an:

- Ist der Ausdruck C_6H_6 in dem durch die Regeln festgelegten Kalkül *ableitbar*?
- Ist das *Theorem* “C_6H_6 ist existent” wahr?
- Ist die Zeichenkette C_6H_6 in der *Sprache* enthalten, die durch die Regeln festgelegt ist?

Die letzte Frage ist ein Vorgriff auf das nächste Kapitel. Sie wird verständlich, wenn man weiß, dass in der Theorie formaler Sprachen eine Sprache *extensional* als Menge aller zugelassenen Zeichenketten definiert wird.

Die drei Fragen sollen die Zusammenhänge zwischen verschiedenen Gebieten der KI verdeutlichen, dem *Schließen*, dem *Theorembeweisen* und dem *Übersetzen* oder allgemeiner der formalen *Analyse und Synthese sprachlicher Ausdrücke*. Auf einer ausreichend hohen Abstraktionsebene betreffen die verschiedenen Fragen ein und denselben Sachverhalt; sie werden identisch miteinander (ein Beispiel für begriffliche Konvergenz [8.38]). Infolgedessen ist es auch nicht verwunderlich, dass überall die gleichen Arten von Indeterminismus auftreten, dass überall gesucht werden muss und dass die Methoden des Suchens und seine Darstellung durch Graphen, insbesondere durch Bäume, in den verschiedenen Bereichen der KI ein und dieselben sind.

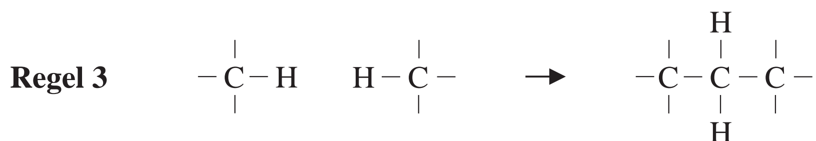
Wenn Kekulé nach unserem Algorithmus vorgegangen ist, musste seine Suche erfolglos bleiben, solange er sich nicht von dem Algorithmus lösen konnte. Denn die Verbindung C_6H_6 ist nicht generierbar, sie liegt - so die Sprechweise der Mathematiker - außerhalb der *Suchraumes*, der durch die beiden Regeln *aufgespannt* ist[10]. Offenbar beschreibt der Algorithmus die Wirklichkeit nur unvollständig. Ein besserer Algorithmus ist erforderlich, m.a.W. das Regelwissen muss erweitert werden.

Wer die Strukturformel von Benzol kennt, wird eine Regel suchen, die Ringverbindungen produziert. Kekulé wusste es nicht. Der Computer weiß es auch nicht. Kann man ihn trotzdem dazu befähigen, sich eine neue Regel “auszudenken”? Offensichtlich ist das möglich. Es ist nicht sehr schwierig, ein Programm zu entwer-

---

10 Wenn Dreifachbindungen zugelassen werden, ist C_6H_6 generierbar, allerdings nur formal, nicht tatsächlich im Labor.

fen, das mit Hilfe eines Zufallszahlengenerators Verbindungen aus C- und H-Atomen mit einer oder mehreren freien Valenzen generiert, die den Axiomen nicht widersprechen. Man beachte, dass auch der Programmierer, der das Programm schreibt, nichts von Ringverbindungen zu wissen braucht. Angenommen der Computer verfügt über ein solches Programm und er generiert die Regel,



nach der zwei H-Atome durch eine CH_2-Gruppe substituiert wird. Wenn der Computer diese Regel seinem Regelwissen hinzufügt und mit dem erweiterten Wissen puzzelt, wird er früher oder später den Benzolring generieren. Man beachte, dass auch Regel 1 und Regel 2 hätten "erwürfelt" werden können. Tatsächlich lassen sich alle denkbaren Regeln aus den Axiomen *herleiten*, ähnlich wie sich die Rechenregeln eines axiomatisierten Kalküls aus den Axiomen herleiten lassen.

Es stellt sich heraus, dass der puzzelnde Computer "in einem Kalkül rechnet". Das überrascht nicht, denn zu irgendetwas Anderem kann er nicht befähigt werden. Er rechnet sogar in einem axiomatisierten Kalkül. Allerdings ist der Kalkül nicht ordnungsgemäß formal definiert, es sei denn, man akzeptiert die graphische Beschreibung der Zeichenmuster und Regeln als Definition der Syntax der "CH-Sprache" (Sprache der Kohlen-Wasserstoff-Verbindungen). Wenn der Computer die CH-Sprache verstehen soll, muss sie implementiert werden, was die formale Definition ihrer Syntax einschließt.

Kekulé war, wie seine Zeitgenossen, offenbar nicht in der Lage, Regel 3 zu formulieren und gemeinsam mit den anderen Regel systematisch anzuwenden, zumindest nicht im wachen Zustand, weil seiner Phantasie durch Denkgewohnheiten Grenzen gesetzt waren. Denn durch Regel 3 wird eine Kette aus z.B. 5 C-Atomen zu einem Ring aus 6 C-Atomen "zusammengebogen". Wenn sich in dem Ring Einfach- und Doppelbindungen abwechseln, was sich mittels Regel 2 erreichen lässt, ergibt sich der Benzolring. Das lag offenbar jenseits der Grenzen damaliger Vorstellungsgewohnheiten. Im Halbschlaf verschwinden diese Grenzen. Im Traum ist das Ungewohnte, des Unerlaubte, das Ungesetzliche und das Unmögliche erlaubt und möglich. Der *Phantasie* ist freier Lauf gelassen. Hier tritt die gemeinsame Wurzel der Begriffe Intuition, Kreativität, Findigkeit und Phantasie deutlich zutage.

Der Computer verfügt über den Speicherinhalt hinaus über keinerlei Erfahrung, die das Suchen einschränken könnte. Darauf beruht sein Erfolg. Man könnte sagen: Er ist klug, weil er "dumm" ist, d.h. weil er wenig weiß. Dieser Satz kann auch auf Menschen zutreffen. Beispielsweise wäre es gar nicht ausgeschlossen gewesen, dass ein intelligenter Chemielaborant die Strukturformel von Benzol schneller hätte finden können, weil das Denken eines Laboranten weniger durch Spezialwissen eingeschränkt ist als das eines Professors.

Man überzeugt sich leicht, dass auch Regel 3 den durch die Axiome aufgespannten Suchraum noch einengt. Beispielsweise lässt sich gedanklich ein Molekül konstruieren, das aus einem Ring von C-Atomen besteht, die über Doppelbindungen miteinander verbunden sind. Ein solcher Ring enthält kein einziges H-Atom. Er widerspricht nicht den Axiomen

Zweifellos ließe sich ein Programm schreiben, nach dem der Computer schrittweise *sämtliche* Strukturen generiert, die sich mit den Axiomen im Einklang befinden. Der Computer könnte also beispielsweise beauftragt werden, aus *m* Kohlenstoffatomen alle erlaubten Strukturen aufzubauen, beginnend mit einer minimalen Anzahl von H-Atomen.

Bei Ausführung dieses Programms würden zunächst alle von den Axiomen her erlaubten reinen C_n-Moleküle generiert werden mit $n \leq m$. Derartige Verbindungen sind vor gar nicht langer Zeit im Labor gefunden worden; sie heißen **Fullerene**. Beispielsweise ist das Molekül C_{60} ein bekanntes Fulleren[11]. Der Computer würde auch die (topologische) Struktur des Diamanten komponieren. Allerdings könnte er nicht erkennen, dass es sich um ein *räumliches Gitter* handelt. Dazu müsste er mehr wissen. Irgendwann würde der Computer auf das Benzolmolekül stoßen. Die drei Verkettungsregeln benötigt er dafür nicht. Hätte es zu Kekulés Zeiten schon die KI gegeben, dann hätte ihm der Computer zuvorkommen können.

Das Benzolbeispiel zeigt, dass Findigkeit - sei es die eines Forschers oder die eines Rätselraters - entscheidend von der Fähigkeit abhängt, im “*richtigen*” *Suchraum* zu suchen. Dieser darf nicht zu eng, aber auch nicht zu weit sein, weil sonst das Suchen entweder erfolglos oder zu langwierig ist. Als sehr anschauliche Illustration eines zu engen Suchraumes sei folgende bekannte Denksportaufgabe angeführt.

Denksportaufgabe 2: Problem der Verbindungslinie

Die 9 Punkte in Bild 16.4a sind durch einen einzigen Streckenzug (einen Zug von vier geraden Linien ohne abzusetzen) zu verbinden. Solange man auf den Bereich innerhalb der Punkte fixiert ist, bleiben alle Versuche erfolglos, wie z.B. der in Bild 16.4a dargestellte. Erst wenn man über diesen Bereich hinausdenkt, kann man die in Bild 16.4b gezeigte Lösung finden. Der zu enge Suchraum enthält als Kandidaten für die Teilstrecken, aus denen sich der gesuchte Streckenzug zusammensetzt, nur diejenigen Geraden, auf de-

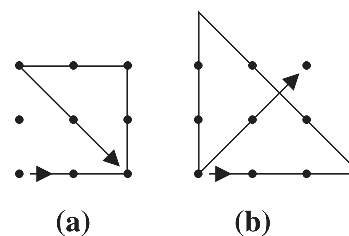


Bild 16.4 Denksportaufgabe 2.

11 Die Bezeichnung ist von dem Namen des amerikanischen Architekten R. Buckminster Fuller abgeleitet, dessen Kuppelbauten den C_n-Molekülen ähnlich sehen.

nen je 3 der 9 Punkte liegen. Der erweiterte Suchraum enthält außerdem diejenigen, auf denen 2 Punkte liegen.[12]

Es stellt sich die Frage, ob für die Lösungsfindung tatsächlich Intuition erforderlich ist, ob es sich also um *Erfinden* handelt oder im Grunde doch "nur" um *Schlussfolgern*. Das Benzolbeispiel legt folgende Antwort nahe: Das *regelbasierte Suchen* erfordert keine Intuition, während das *Formulieren der Regeln* nicht ohne Intuition möglich zu sein scheint.

Aber der Schein trügt, wie wir bereits wissen. Denn beispielsweise ergeben sich die Regeln, Regel 3 eingeschlossen, unmittelbar aus den Axiomen, worauf bereits hingewiesen wurde. Man braucht sich bei ihrer Formulierung nicht vorzustellen, zu welchen Konfigurationen sie führen können, insbesondere braucht man sich keine Ringverbindung vorzustellen (das wäre *externe* Semantik). Der Computer liefert sie automatisch, wenn er "stupide" die Axiome - aber eben nur diese - anwendet. Demnach ist zum Finden der Struktur des Benzolringes keine Intuition erforderlich. Andrerseits ist es ganz sicher gerechtfertigt, zu sagen, dass Kekulés Leistung auf Intuition beruhte. Seine Intuition war die *Erweiterung des Suchraumes*.

Um diesen Gedanken klarer zu artikulieren, führen wir den Begriff des Beschreibungsraumes ein. *Der **Beschreibungsraum** eines Originalbereiches ist die Menge aller Aussagen über den Originalbereich, die einer Menge von Grundaussagen nicht widersprechen. Die Menge der Grundaussagen nennen wir **Grundwissen**.* Das Grundwissen spannt den Beschreibungsraum auf, ebenso wie das **Suchwissen** den **Suchraum** aufspannt. Das Suchwissen ist die Gesamtheit von Faktenwissen und Regelwissen. Wenn Grundwissen und Suchwissen zusammenfallen, fallen auch Beschreibungsraum und Suchraum zusammen. 10

Betrachtet man nun noch einmal kritisch das Benzolbeispiel, erkennt man, dass die Regeln 1 und 2 bzw. 1, 2 und 3 einen kleineren bzw. einen größeren Unterraum des Beschreibungsraumes festlegen, der seinerseits durch das Grundwissen (H ist einwertig; C ist vierwertig) festgelegt ist. Die drei Räume sind gewissermaßen ineinandergeschachtelt. Der Übergang in den nächst größeren Raum kann aus den Grundaussagen *abgeleitet*, also *ohne* Intuition gefunden werden.

Wenn der gedankliche Zugang in einen übergeordneten Suchraum durch Denkgewohnheiten versperrt ist, aber dennoch gefunden wird, ist offensichtlich Intuition erforderlich. Doch muss diese Art der Intuition von der "unabdingbar notwendigen Intuition" unterschieden werden. Darum vereinbaren wir: *Wenn eine Aussage auf intuitivem Wege gefunden worden ist, obwohl sie durch Ableitung hätte gefunden werden können, wenn also die intuitive Leistung auf Ableiten zurückgeführt werden kann, sprechen wir von **reduzibler Intuition**.* 11

Ob jede Suchraumerweiterung ableitbar ist, erscheint zweifelhaft. Vielmehr setzt sie oft hohe intuitive Intelligenz voraus. Die Fähigkeit zum Erweitern des Suchrau-

12 Das Beispiel und seine formale Darstellung ist in [Schefe 87] behandelt.

mes durch Überwindung von Denkgewohnheiten war für die Fortschritte der Physik von hervorragender Bedeutung. Das ist verständlich, denn der Beschreibungsraum, in dem alle Beobachtungen auffindbar sein müssen, ist durch “eherne Naturgesetze” festgelegt, an deren Gültigkeit man sich gewöhnt und die man deswegen für absolut gültig hält. Doch kann sich herausstellen, dass sie relativiert werden müssen, um neue Beobachtungen erklären zu können. An einige Beispiele sei erinnert.

Nicolaus Kopernikus überwand die Vorstellung, dass die Sonne um die Erde kreist, und postulierte den umgekehrten Sachverhalt, wodurch sich die beobachtete Bewegung der Planeten am Himmel einfacher erklären ließ. Unter Erweiterung des Suchraums für die Planetenbahnen von Kreisen auf Ellipsen konnte Johannes Kepler aus dem von Tycho de Brahe bereitgestellten Faktenwissen seine Gesetze ableiten. Diese Leistung vollbringt heute bereits der Computer, abgesehen von der Festlegung des Beschreibungsraums. Wir kommen darauf noch einmal in Kap. 21.4 zurück.

Max Planck überwand die Denkgewohnheit, dass die Energie einen kontinuierlichen Wertebereich besitzt, und Albert Einstein machte sich von der Vorstellung frei, dass Geschwindigkeiten sich einfach vektoriell addieren, z.B. die Geschwindigkeit einer Person oder eines Gegenstandes *in* einem Zug (relativ zum Zug) und die Geschwindigkeit des Zuges (relativ zu einem ruhenden Bezugssystem, z.B. zur Erdoberfläche).

Die Frage ist erlaubt, ob bzw. wann der Computer zu derartigen Leistungen fähig ist. Die Antwort ist eigentlich trivial: Wenn er im Suchraum keine Lösung findet, aber über einen Beschreibungsraum verfügt, der über den Suchraum hinausgeht, kann man ihn per Programm veranlassen, den Suchraum (“seinen Horizont”) zu erweitern. Das “Erfinden” neuer Regeln wird zum Ableiten, denn es erfolgt seinerseits *regelbasiert*, und zwar auf der Basis des Grundwissens. Dem entspräche im Benzolbeispiel das Ableiten der Regel 3. Das wäre die Simulation reduzibler Intuition, zu der Kekulé nur im Halbschlaf fähig war.

In diesem Kapitel ist es uns im Grunde nicht gelungen, die Grenzen der KI zugunsten des Computers zu verschieben. Wir haben zwar gesehen, dass bestimmte Leistungen, von denen gesagt wird, dass sie Intuition erfordern, simulierbar sind. Die genaue Analyse hat jedoch gezeigt, dass es sich dabei um Suchraumerweiterungen handelt, die auch ohne “echte” Intuition hätten gefunden werden können.

Oft ist aber nicht *reduzible*, sondern *echte* Intuition gefragt, die durch keinerlei Schlussfolgern ersetzt werden kann. Das gilt beispielsweise für das Ödipusrätsel. Wie ließe sich in diesem Fall der Beschreibungsraum und wie ließen sich Suchräume festlegen? Offenbar versagt die Methode. Bisher haben wir kein Rezept gefunden, diejenige Intelligenz zu simulieren, die notwendig ist, um derartige Rätsel zu lösen.

Es ist an der Zeit, die Ergebnisse, zu denen wir gelangt sind, zu resümieren. Ein kritischer Blick auf das bisherige Kapitel 16 zeigt, dass es sich im Grunde um eine Ergänzung des Kapitels 15.8 handelt. Denn in Kap.16 wurden Sonderfälle des analytischen Rechnens behandelt, für das in Kap.15.8 eine allgemeine Methode auf der Grundlage der Formelmanipulation erarbeitet worden ist. Der Unterschied zwi-

schen den Kapiteln 15 und 16 besteht darin, dass in Kap.15 Probleme behandelt wurden, mit deren Stellung auch der Kalkül vorgegeben war, in dessen Rahmen sie zu lösen sind. Dagegen musste für die Lösung der in Kap.16 gestellten Probleme ein passender Kalkül erst gefunden werden. Aus diesem Grunde werden die Probleme des Kapitels 15 "automatisch" als *mathematische* Probleme klassifiziert, während die Probleme des Kapitels 16 von unvoreingenommenen Lesern natürlicherweise als *nichtmathematische* Probleme klassifiziert werden.

Der tiefere Grund dieser Unterscheidung liegt darin, dass die Werte der Variablen in den Problemen des Kapitels 15 Zahlen sind, was auf die Probleme des Kapitels 16 nicht zutrifft. Mit anderen Worten, die sprachlichen Modelle in Kap.15 beruhen auf *Messen* und *Zählen*, in Kapitel 16 dagegen auf *Klassifizieren* ohne Bezugnahme auf Zahlen, d.h. ohne Quantifizierung. In Kapitel 16 sind wir in gewissem Sinne "zu den Wurzeln" der Rechentechnik zurückgekehrt, zur "Klassenlogik" des ARISTOTELES (vgl. Kap.11.1). Aus dieser Sicht hätte die Behandlung der boolesche Algebra in Kap.16 eingeordnet werden müssen. Sie musste vorgezogen werden, weil wir die boolesche Algebra als hardwaremäßig zu realisierenden Kalkül ausgewählt hatten.

Für drei der vier Grundideen des elektronischen Rechnens [8.42] haben wir uns Realisierungsmöglichkeiten überlegt; es fehlt die vierte, das automatische Übersetzen. Aus der Sicht des übersetzenden Menschen, beispielsweise eines Dolmetschers, ist Übersetzen eine nichtmathematische Leistung menschlicher Intelligenz. Um das Übersetzen zu automatisieren, muss es kalkülisiert werden. Insofern gehört das Übersetzungsproblem, dem wir uns nun zuwenden, in das Kapitel "Lösen nichtmathematischer Probleme".

16.4 Übersetzen und die vierte Grundidee des elektronischen Rechnens

Bei unseren Überlegungen darüber, wie der Computer nichtmathematische Aufgaben lösen kann, ging es um die Kalkülisierung verschiedener Denkprozesse, speziell des Schlussfolgerns und des Erfindens. Dabei bestand das Kalkülisieren darin, dass eine geeignete formalisierte Sprache definiert wurde und der Denkprozess auf das Ableiten von neuen aus alten (als wahr vorausgesetzten) Sätzen dieser Sprache nach vorgegebenen Regeln zurückgeführt wurde. Das Ergebnis ist ein spezieller Kalkül, genauer eine spezielle Beschreibung von Denkprozessen in Form eines Kalküls. Wir haben ihn *speziellen Denkkalkül* genannt.

Damit ist der erste Teil von Ziel 2, wie es zu Beginn von Kap.13 [13.1] formuliert wurde, erfüllt. Der zweite Teil steht noch aus, die Überführung des speziellen Denkkalküls in den Maschinenkalkül, m.a.W. die Übersetzung der speziellen Kalkülsprache in die Maschinensprache unter Erhaltung der Semantik. Die Idee, die Übersetzung vom Computer selber ausführen zu lassen, nennen wir die **vierte**

Grundidee der elektronischen Rechnens. In diesem Kapitel wollen wir uns überlegen, wie sich die Idee verwirklichen lässt.

Es wurde wiederholt darauf hingewiesen, dass das schwierigste Problem die Erhaltung der Nutzersemantik ist (der Semantik, "in" oder "mit" welcher der Nutzer denkt). Wenn ein Nutzer dem Computer einen Auftrag in der Sprache eines speziellen Denkkalküls erteilt, muss gewährleistet sein, dass der Auftrag das Beabsichtigte bewirkt, m.a.W. dass der Computer den Auftrag "richtig versteht". Wir wollen uns noch einmal klarmachen, was das bedeutet.

Im Falle der Denksportaufgabe 1 (Verwandtschaftsproblem) lautet die Frage "Welche Verwandtschaftsbeziehung besteht zwischen `Er` und `Ich`?" Dafür haben wir die Kurznotation `?(er,ich)` vereinbart. Im Benzolbeispiel lautet die Frage "Wie ist die Strukturformel von C_6H_6?", in Kurznotation z.B. `struk?(6,6)`. Man könnte auch die Frage stellen "Welche Kohlenwasserstoffmoleküle sind denkbar?" in Kurznotation z.B. `moleküle?(C,H)`.

12 Die Sprache, in der dem Computer derartige Fragen gestellt werden, hatten wir *Anfragesprache* genannt. Allgemein werden wir eine Sprache, in der einem Computer ein Auftrag erteilt wird (das kann auch eine Frage oder der Auftrag zur Prüfung einer Hypothese sein), **Auftragssprache** nennen. Im Falle der Verwandtschaftsaufgabe war die Auftragssprache an die Sprache des Prädikatenkalküls angelehnt. Im Falle eines Expertensystems für Chemie wird man sie an die Fachsprache für chemische Verbindungen und Reaktionen anlehnen. Der Computer muss die Ausdrücke der vereinbarten Auftragssprache *verstehen*.

In der Alltagskommunikation zwischen Menschen sagt man, dass eine Person B verstanden hat, was eine Person A gesagt hat, wenn B ausreichend genau weiß, was A gemeint hat, oder in der Sprechweise von Kap.2 und Bild 2.1, wenn B einem von A empfangenen Zeichenrealem ein Idem zuordnet, das ausreichend genau mit dem Idem übereinstimmt, das A artikulieren wollte. Ob die Genauigkeit der Interpretation ausreichend war, zeigt sich im Verhalten (im Tun und Sagen) von B. Wenn A von B nicht ausreichend genau "verstanden" worden ist, kann sich das sofort, eventuell aber auch erst nach langer Zeit oder nie herausstellen. A kann die Genauigkeit des Verstehens kontrollieren, wenn B die Nachricht mit seinen eigenen Worten wiederholt.

Die Interpretation einer Nachricht durch den Empfänger, die *Empfängersemantik* der Nachricht, manifestiert sich also letztlich im Handeln (Tun und Sagen) des Empfängers. Sehr deutlich wird das in den Worten der Mutter: "Hast du mich *verstanden*?!" Die Frage ist im Grunde eine nachdrückliche Mahnung: "*Tu*, was ich dir gesagt habe!"

Im Hinblick auf den Computer ist Verstehen definitionsgemäß identisch mit Handeln, denn als *trägerinterne* Semantik hatten wir denjenigen Prozess bezeichnet, der von einem Zeichenrealem (z.B. einem Befehl oder Programm) im Träger (im Computer) ausgelöst wird [5.7]. Interpretieren durch den Computer, d.h. Zuordnen

der internen Semantik, ist identisch mit Abarbeiten. *Der Computer "**versteht**" einen Auftrag, indem er ihn **ausführt**. Er versteht ihn **richtig**, wenn er ihn richtig ausführt.*

Hiergegen kann eingewendet werden, dass der Computer einen Auftrag "verstanden" haben muss, *bevor* er mit seiner Ausführung beginnen kann. Der Einwand enthält einen wahren Kern. Denn wenn dem Computer beispielsweise die Frage `struk?(6,6)` gestellt wird, kann er dieses Zeichenrealem zunächst einmal nur Zeichen für Zeichen lesen. In der eingelesenen Zeichenfolge kann er dann ihm bekannte Teilfolgen erkennen. Dazu muss er eine **lexikale Analyse** durchführen (wie der Assembler in Kap.15.4). Dabei erkennt er z.B., dass die Buchstabenfolge `struk` ein Lexem der Auftragsprache darstellt. Damit hat er aber noch nicht den eigentlichen Sinn des Auftrags erfasst, den der Mensch in ihn hineinlegt.

Die lexikale Analyse führt zwar zu einem gewissen "Verständnis" insofern, als die Analyse *metasprachliche* Aussagen liefert, also Aussagen *über* den betreffenden Satz, jedoch nicht zu einem Verständnis auf *objektsprachlicher* Ebene; die Bedeutung des Satzes kann nicht erschlossen werden. Was das heißt, wollen wir uns an einem Beispiel - wieder aus der Schule, diesmal aus dem Englischunterricht - klarmachen. Max soll folgenden Satz übersetzen, obwohl er kein Wort Englisch kann:

Peter understands this sentence. (16.1)

Max ist ein findiges Bürschchen und übersetzt: "Peter unterstand die Sentenz". Nach kurzem Nachdenken berichtigt er: "Peter *ver*stand die Sentenz". Was mag dabei alles in Maxens Kopf vorgegangen sein? Das ist ein weites Feld, das wir im Augenblick nicht betreten wollen. Nachdem Max ein Wörterbuch erhalten und einige Zeit darin gesucht hat, liefert er die richtige Übersetzung, wobei er ganz selbstverständlich das Objekt "Satz" in den Akkusativ setzt. Wenn Max nichts über englische Grammatik weiß und weniger findig ist, braucht er ein morphologisches Wörterbuch, aus dem er z.B. erfährt, dass "understands" die Wortform für die dritte Person Präsens von "understand" ist.

Die Vokabelsuche hätte der Computer eventuell schneller erledigt. Er hätte aber - selbst mit einem morphologischen Wörterbuch - nicht ohne Weiteres feststellen können, in welchem Fall die Wörter "Peter" und "sentence" stehen und welches von ihnen Subjekt und welches Objekt ist. Vielmehr muss er eine **syntaktische Analyse** durchführen. Auch der Mensch kommt zuweilen nicht ohne bewusstes Analysieren der Syntax aus, insbesondere, wenn er die Sprache schlecht beherrscht. 13

Angenommen, Max hat in der letzten Stunde die syntaktische Konstruktion des AcI (Akkusativ mit Infinitiv) und dessen Übersetzung ins Deutsche gelernt. Nun soll er folgenden Satz übersetzen:

I heard him go away. (16.2)

Da ihm der AcI noch nicht in Fleisch und Blut übergegangen ist, übersetzt er ihn nicht "automatisch" richtig, sondern er führt zunächst eine *syntaktische Analyse*

durch, wobei er die gelernte *syntaktische Übersetzungsregel* anwendet. Die Regel sagt aus, dass dem englischen syntaktischen Konstrukt “AcI” das deutsche Konstrukt “Dass-Satz” entspricht, also ein Nebensatz, der mit “dass” beginnt. Dabei wird das Akkusativ-Lexem des AcI zum Subjekt und der Infinitiv zum Prädikat (im grammatikalischen Sinne) des Dass-Satzes.

Wir wollen uns nun überlegen, welche weiteren Hilfsmittel Max benötigt, um den Satz (16.2) richtig übersetzen zu können, *ohne* vorher den AcI gelernt zu haben, und wie er dabei vorzugehen hat. Offensichtlich benötigt Max neben dem morphologischen ein syntaktisches Wörterbuch, in dem zu allen englischen syntaktischen Konstrukten die deutschen Äquivalente angegeben sind, u.a. die obige AcI-Regel. Die syntaktischen Quell- und Zielkonstrukte müssen in einer geeigneten Metasprache artikuliert sein.

Damit ist das Übersetzungsproblem aber noch nicht gelöst. Max kann nämlich nicht sicher sein, ob ein erkanntes Akkusativ-Lexem mit nachfolgendem Infinitiv-Lexem ein AcI ist. Dafür müssen noch andere syntaktische Voraussetzungen erfüllt sein. Grundsätzlich muss die Lexemfolge des *gesamten* Satzes als syntaktisches Konstrukt erkannt sein, das den Syntaxregeln entspricht. Die Grundidee der syntaktischen Satzanalyse ist in Kap.5.3 am Beispiel einfacher deutscher Sätze dargelegt worden. Es besteht darin, dass Teile des Satzes zu *metasprachlichen Variablen* zusammengefasst werden (präziser: zu Klassen, die mit metasprachlichen Variablenbezeichnern benannt werden), die ihrerseits zu metasprachlichen Variablen höherer Ebene zusammengefasst werden, sodass ein Syntaxbaum entsteht (vgl. Bild 5.2). Wenn die Zusammenfassungen der metasprachlichen Variablen durch geeignete Klammerung gekennzeichnet wird, kann die zweidimensionale Beschreibung der sytaktischen Struktur in eine lineare überführt werden, z.B. in die Backus-Naur-Form [5.3]. Zur Illustration betrachten wir Maxens Vorgehen bei der Analyse des Satzes (16.2). Die lexikale Analyse ergibt die Lexemfolge

Pronomen im 1.Fall, Verbform, Pronomen im 4.Fall, Infinitiv.

Diese nichtformalisierte Darstellung ist sicher jedem verständlich. Um die syntaktische Struktur des Satzes zu erkennen, muss Max sich anhand des syntaktischen Wörterbuches davon überzeugen, dass ein Pronomen im 1. Fall die Rolle eines Subjekts und eine Verbform die Rolle eines Prädikats spielen können, dass ein Pronomen im 4. Fall mit nachfolgendem Infinitiv ein AcI sein kann und dass die Folge Subjekt, Prädikat, AcI ein Satz sein kann. Es ergibt sich folgende syntaktische Struktur der Lexemfolge (16.2):

Pronomen im 1.Fall, Verbform, (Pronomen im 4.Fall, Infinitiv),

wobei der AcI in runde Klammern gesetzt ist.

Es fragt sich nun, ob bzw. wie Maxens Methode automatisiert d.h. ob und wie Übersetzen kalkülisiert werden kann. Dass dies im Prinzip möglich ist, erkennt man, wenn man sich das Grundprinzip klarmacht, nach dem Max vorgeht. Es besteht darin,

dass er schrittweise Teilzeichenketten des Quelltextes durch andere Zeichenketten *substituiert*. In jedem Schritt sucht er zuerst nach einer passenden Teilzeichenkette (Lexemfolge) und anschließend substituiert er sie durch eine andere Zeichenfolge gemäß Wörterbuchregel. Genau so geht der Markovalgorithmus, das analytische Rechnen und das Schlussfolgern vor. Damit ist zwar noch nicht die Frage beantwortet, wie man in dem speziellen Fall des automatischen Übersetzens konkret zu verfahren hat, doch das Prinzip ist klar, und wir sind ausreichend vorbereitet, um in großen Zügen verstehen zu können, wie ein Übersetzerprogramm arbeitet.

Die Suche nach effektiven Methoden der Sprachübersetzung hat zur Herausbildung eigenständiger Gebiete der theoretischen und der praktischen Informatik geführt, nämlich der Theorie formaler Sprachen und der Übersetzerprogrammtechnik, speziell des Compilerbaus. Wir werden uns nicht in diese Gebiete vertiefen, sondern verweisen auf die Literatur[13] und begnügen uns mit der Herausarbeitung der zentralen Ideen sowie einer Andeutung ihrer Realisierungsmöglichkeiten.

Die Grundidee übernehmen wir von Max. Übersetzen ist eine Kompositoperation aus drei nacheinander auszuführenden Bausteinoperationen, der lexikalen Analyse, der syntaktischen Analyse und der Substitution. Die drei Operationsvorschriften (Programme) heißen **Scanner**, **Parser** und **Codegenerator**. Letzterer generiert den *Zielcode* (das Zielprogramm).

Bevor wir auf den Scanner eingehen, wollen wir uns überlegen, wie der Parser arbeitet, um zu erkennen, welche Informationen er vom Scanner erhalten muss. Wir beginnen mit folgender Definition: *Eine* ***Sprache*** *ist die Gesamtheit ihrer Sätze* oder abstrakter: *Eine Sprache ist die Menge aller zulässigen Zeichenketten*. Welche Zeichenketten (allgemeiner welche Zeichenkörper oder konkreter welche Wörter und welche Sätze[14]) erlaubt sind, wird von den Syntaxregeln (von der Syntax[15]) der Sprache bestimmt.

Wer durch das Wort “Syntax*regeln*” an die Regeln eines Kalküls und speziell an die Substitutionsregeln des Schlussfolgerns erinnert wird, ist der Lösung unseres Problems schon ein Stück näher gekommen. Wenn man nun noch an die *Generierung* der CH-Sprache (der Formeln von Kohlenwasserstoffen) denkt, steht man unmittelbar vor der Erfindung der sogenannten **generativen Grammatiken**. In Kap.16.3 war von der *Generierung* der chemischen Formel des Benzols, also der Zeichenkette C_6H_6 die Rede. Dabei standen dem Computer drei Substitutionsregeln zur Verfügung, durch die alle erlaubten Molekülketten und damit alle “erlaubten Zeichenketten” festgelegt sind, deren Gesamtheit die CH-Sprache bildet.

---

13 Z.B. [Schöning 95], [Schmitt 92], [Aho 92].

14 In der theoretischen Literatur wird statt “Satz” oft “Wort” gesagt und zwischen Wort und Satz nicht unterschieden.

15 In der Sprachlehre wird die Lehre vom Satzbau als Satzlehre oder Syntax bezeichnet (vgl. die Definition zu Beginn von Kap.5.3). Sie ist Bestandteil der Grammatik.

Wir abstrahieren nun von jeglicher Semantik und definieren eine fiktive Sprache ganz formal, beispielsweise durch folgende Regeln:

```
   w  → aa
  aa  → aaaa
aaaa  → b;
```

`w` steht für "Wort" und stellt das **Spitzensymbol** dar. Gemäß diesen Regeln lassen sich z.B. folgende Wörter generieren: `aa, aab, aaaa, b, bb, aabb`. Die so definierte "*Sprache*" (Menge von Zeichenketten) enthält alle "Wörter", die aus Teilketten aus einer geraden Anzahl des Zeichens `a` und Teilketten aus einer beliebigen Anzahl des Zeichens `b` bestehen. In einer der Backus-Naur-Form ähnlichen Notation könnte dieser Sachverhalt folgendermaßen notiert werden: $[[aa]^*[b]^*]^*$, wobei der Stern beliebig häufige, auch 0-malige Aneinanderreihung der geklammerten Kette bezeichnet. Man beachte, dass diese Notation keine syntaktische (im Sinne der Syntax einer Sprache), sondern lediglich eine *lexikale Struktur* beschreibt. Man kann die einzelnen Zeichen auch als Lexeme auffassen. Dann beschreiben die Regeln alle Lexemfolgen der betreffenden Sprache.

Auf diese Weise lassen sich sehr einfache Sprachen beschreiben, generieren oder definieren (welches Verb hier passt, hängt vom Standpunkt des Betrachters ab), z.B. die Sprache der arabischen oder der römischen Zahlen. Anhand der Regeln lässt sich auch feststellen, ob eine gegebene Zeichenkette ein Wort bzw. eine Lexemfolge der betreffenden Sprache ist. Für komplexere Sprachen scheint die generative Methode ungeeignet zu sein. Dennoch ist sie auf Grund einer weiteren Idee sogar für natürliche Sprachen erfolgreich einsetzbar.

Um uns dieser Idee zu nähern, vergegenwärtigen wir uns noch einmal, dass das Artikulieren, d.h. das Codieren von Idemen (von Semantik) bewusst oder unbewusst auf zwei Ebenen abläuft. Auf der objektsprachlichen Ebene wird der gedachten und zu artikulierenden Bedeutung (dem Bewusstseinsinhalt, dem Idem) eine Zeichenkette explizit zugeordnet, die eine *lexikale* (in der Regel hierarchische) Struktur besitzt (vgl. Bild 5.1). Dabei werden auf der metasprachlichen Ebene implizit den lexikalen Bestandteilen der Zeichenkette *metasprachliche Variable* zugeordnet wie Subjekt, Prädikat, Objekt. Dadurch erhält die Zeichenkette eine *syntaktische* Struktur.

Die entscheidende Idee besteht nun darin, den Generierungsmechanismus nicht nur auf die lexikale, sondern auch auf die syntaktische Struktur anzuwenden. Damit ist das Prinzip klar, nach dem der Parser vorzugehen hat. Voraussetzung ist, dass für die Quellsprache und für die Zielsprache je ein System von Syntaxregeln existiert, nach dem sich jeder Satz der Sprache syntaktisch beschreiben lässt. Für die deutsche Sprache wird das System u.a. die Regeln (5.1), (5.3) und (5.4) und für eine Maschinensprache die Regel (5.2) enthalten. Für die in Bild 15.3b verwendete Sprache gelten u.a. die Regeln[16]

16 Es ist hier nicht die allgemeine Definition der WHILE-Anweisung angegeben.

WHILE-anweisung	→	`WHILE` *ausdruck rel zahl* `DO` {*ergibtanweisung*} `ENDWHILE`
ergibtanweisung	→	*id* := *ausdruck* | *zahl*
ausdruck	→	*id* | *op(id)* | *id op id*

Im Gegensatz zu (5.2) sind die metasprachlichen Variablen nicht in spitze Klammern gesetzt, sondern kursiv gedruckt. Das Zeichen | steht für "oder" (alternative Möglichkeiten).

Die Lexeme `WHILE`, `DO` und `ENDWHILE` (Ende der WHILE-Anweisung) gehören zur Objektsprache und heißen **Schlüsselwörter**. Variablenbezeichner sind mit *id* (von **Identifikator**) abgekürzt, *op* steht für Operationszeichen und *rel* für Relatopnszeichen. Es handelt sich um Namen für Lexemklassen, z.B. *id* für die Klasse der Identifikatoren. Man kann das Dekomponieren als "Rechnen mit Variablen" auffassen, wobei die Klassennamen die Rolle von Variablen speziell von *metasprachlichen* Variablen spielen, denen Werte (z.B. konkrete Bezeichner) zugeordnet werden. Darum sind sie, wie beim analytischen Rechnen, kursiv gedruckt. Die geschweiften Klammern bedeuten diesmal, dass der geklammerte Teil sich beliebig oft wiederholen darf, aber mindestens einmal auftreten muss.

Die Regelliste ist nicht vollständig. Zum einen fehlen die Regeln für *zahl* und *id*, zum anderen ist die Regel für *ausdruck* unvollständig, wie der Vergleich mit Bild 15.3b zeigt. Die meisten Zeilen des dortigen Programms werden zwar richtig beschrieben, die Ergibtanweisung für `s` jedoch nicht. Wer versucht, die Regel für *Ausdruck* um weitere Alternativen zu ergänzen, wird feststellen, dass dies gar nicht so ganz einfach ist. Wir verzichten auf die notwendige Erweiterung, da die Regel in der verkürzten Form für die weiteren Überlegungen ausreicht.

Um an Hand derartiger Regeln die syntaktische Struktur (die "Syntax") eines Programms zu erkennen, kann der Parser folgendermaßen vorgehen. Beginnend mit dem Spitzensymbol (mit der metasprachlichen Variablen *Program*) "dekomponiert" er schrittweise jede auftretende metasprachliche Variable, indem er sie durch die rechte Seite einer passenden Syntaxregel substituiert. Der Einfachheit halber nehmen wir an, dass ein Programm laut Syntaxregel eine Folge von Anweisungen ist (vgl. (15.6) in Kap.15.4; den Deklarationsteil unterschlagen wir). Die Dekomponierung des Quelltextes in Anweisungen macht keine Schwierigkeiten, wenn die Syntaxregel vorschreibt, dass Anweisungen durch Trennzeichen (z.B. Semikolons) oder durch Schlüsselworte (z.B. ENDWHILE) abzuschließen sind.

Im nächsten Schritt sind die Anweisungen zu dekomponieren. Die entsprechende Syntaxregel stellt eine Reihe spezieller Anweisungen zur Wahl (Ergibtanweisung, bedingte Anweisung, WHILE-Anweisung u.a.m.). Der Parser muss solange probieren, bis er eine Dekomponierung gefunden hat, die mit dem Quelltext vereinbar ist. Wenn ihm das gelungen ist, substituiert er die betreffende metasprachliche Variable durch die rechte Seite (bzw. durch die passende Alternative der rechten Seite) der

betreffenden Regel. Wenn er auf ein `WHILE` stösst, weiß er, dass er die metasprachliche Variable *WHILE-Anweisung* wählen und durch die rechte Seite obiger Syntaxregel, also durch `WHILE` *ausdruck relop zahl* `DO` {*ergibtanweisung*} `ENDWHILE` substituieren muss. Die Analogie zum analytischen Rechnen (Suchen in einer Formelsammlung und nachfolgendes Substituieren) ist offensichtlich.

Der Parser fährt mit dem schrittweisen Dekomponieren der metasprachlichen Variablen fort, bis er die elementaren metasprachlichen Variablen erreicht hat, die sich nicht mehr dekomponieren, sondern nur noch instanzieren, d.h. durch Lexeme der Objektsprache substituieren lassen, beispielsweise *op* durch `ADD` oder *id* durch den betreffenden Bezeichner.

Häufig sind in einem Dekomponierungsschritt mehrere Regeln oder mehrere Alternativen in einer Regel anwendbar, die alle mit dem Quelltext vereinbar sind, sodass mehrere Dekomponierungswege offen stehen. Dann ist derjenige Weg zu wählen, der mit der grössten Wahrscheinlichkeit zum Quellprogramm führt. Wenn eine Bewertung der Wege nicht möglich ist, muss entweder gewürfelt oder nach einer geeigneten Vorschrift verfahren werden, z.B..nach der Vorschrift “Wähle die in der Regel zuerst genannte Alternative”. Wenn keine solche Vorschrift existiert, ist die Syntaxanalyse ein nichtdeterministischer Prozess, der nach einem *nichtdeterministischen Algorithmus* abläuft. In jedem Fall ist die Analyse mit Suchen verbunden und alle diesbezüglichen Überlegungen, die wir beim analytischen Rechnen [15.15] und beim Schlussfolgern angestellt haben, wären hier zu wiederholen. Erinnert sei an das *Backtracking* [15.17], das oft angewendet wird, wenn der Parser in eine Sackgasse geraten ist.

Während der syntaktischen Analyse entsteht Schritt für Schritt der *Syntaxbaum* des zu analysierenden sprachlichen Ausdrucks, z.B. eines Programms oder eines Satzes (siehe Bild 5.2). Dabei wächst der Baum “von oben nach unten”, beginnend mit dem Spitzensymbol. Dieses auf *De*komponierung beruhende Verfahren wird **Abwärts-** oder **Top-down-Analyse** genannt.

Der Parser kann auch in umgekehrter Richtung vorgehen und “unten”, genauer z.B. “unten links” mit der Analyse beginnen und schrittweise die metasprachlichen Variablen “komponieren”. Im ersten Schritt der Aufwärtsanalyse bestimmt der Parser das kürzeste Anfangsstück des Quelltextes (die kürzeste Anfangslexemfolge), das mit der rechten Seite einer Syntaxregel vereinbar ist und substituiert es durch die metasprachliche Variable, die auf der linken Seite der Regel steht. So fährt er schrittweise von links nach rechts und von unten nach oben fort, bis er das Spitzensymbol erreicht hat. Die Syntaxregeln dienen diesmal als Substitutionsregeln, in denen die Pfeile von rechts nach links zeigen. Diese Vorgehensweise heißt **Aufwärts-** oder **Bottom-up-Analyse**.

Wir wissen nun, welche Informationen der Parser vom Scanner erhalten muss. Mit denjenigen Lexemen, die nicht in den Syntaxregeln auftreten, kann er wenig anfangen. Er muss aber wissen, ob ein solches Lexem ein Identifikator oder ein Operator ist. Diejenigen Lexeme, die in den Syntaxregeln auftreten und zur syntak-

tischen Strukturierung der Lexemkette beitragen, müssen dem Parser explizit übergeben werden. Dazu gehören Trennzeichen (z.B. das Semikolon), Klammern und klammernde Schlüsselwörter (z.B. `WHILE...ENDWHILE`), mit deren Hilfe der Programmierer das Quellprogramm syntaktisch strukturieren kann. Je konsequenter er davon Gebrauch macht bzw. die Sprachdefinition ihn dazu zwingt, umso einfacher ist die syntaktische Analyse.

Wer sich an Kap.15.4 erinnert, wird bemerkt haben, dass die Tätigkeit des Scanners identisch ist mit der Tätigkeit des Assemblers in der ersten Phase des Assemblierens [15.3]. Die Vorgehensweise des Assemblers kann also übernommen werden, das Anlegen einer *Symboltabelle* eingeschlossen. Für jeden Bezeichner legt der Scanner eine Zeile in der Symboltabelle an, wo er selber und später der Parser und der Codegenerator alle relevanten Attribute eintragen, z.B. den Gültigkeitsbereich des Bezeichners innerhalb des Programms, den Typ der bezeichneten Variablen (z.B. `real` oder `integer`), den Speicherplatzbedarf und die Adresse des Speicherplatzes, an den die Variable gebunden ist.

Nachdem der Scanner ein Lexem analysiert hat, übergibt er dem Parser entweder den Namen der Klasse, zu der es gehört, oder, falls es ein Lexem ist, das in den Syntaxregeln auftritt, das Lexem selber. Die Symbole, die er übergibt, also Zeichenketten wie WHILE oder id, werden unter der Bezeichnung **Token** zusammengefasst, und die Tokenfolge, die vom Scanner zum Parser "fließt", wird **Tokenstrom** genannt. Während der lexikalen Analyse der Zeilen 3 und 4 des Programms von Bild 15.6 könnte der Scanner den Tokenstrom

id := zahl ; `WHILE` op (id) rel zahl `DO` (16.3)

generieren.

Angenommen, der Parser arbeitet nach dem Aufwärtsverfahren. Dann sind in der Regel mehrere Token erforderlich, um die nächst höhere syntaktische Einheit gemäß Komponierungsregeln bilden und ihre metasprachliche Klasse bestimmen zu können. Zum Treffen der richtigen Entscheidung benötigt der Parser eventuell viele Token, sodass er weit voraus- und zurückschauen, d.h. viel *Kontext* berücksichtigen muss, m.a.W. auch er muss sich, wie der Mensch, einen gewissen, wenn auch im Vergleich zum Menschen sehr begrenzten Überblick verschaffen. Beim Abwärtsverfahren gilt Entsprechendes.

Der Aufwand, den der Parser treiben muss, insbesondere die Zeitdauer, die er für die Analyse braucht, hängt wesentlich davon ab, wie viel Kontext in den einzelnen Analyseschritten berücksichtigt werden muss. Das aber hängt davon ab, wie *übersetzerfreundlich* die Lexik und die Syntax der Sprache festgelegt sind. Von der Sprachdefinition hängt andererseits die *Nutzerfreundlichkeit* der Sprache ab. Die Forderungen hinsichtlich Übersetzerfreundlichkeit und Nutzerfreundlichkeit können sich widersprechen, sodass beim Sprachentwurf ein Kompromiss zwischen ihnen gefunden werden muss.

Mit der syntaktischen Analyse ist der *Analyseteil* des Übersetzens abgeschlossen und es beginnt der *Syntheseteil*, die Codegenerierung. Sie erfolgt nach dem gleichen Prinzip, das auch Max anwandte, als er einen AcI durch einen Dass-Satz aufgrund eines Wörterbucheintrags substituierte. Auch der Codegenerator muss über ein Wörterbuch verfügen, d.h. eine Liste von Regeln, welche die Rolle von Substitutionsregeln spielen. Auf der linken Seite einer Regel steht ein syntaktisches Konstrukt der Quellsprache, auf der rechten eine Tokenfolge der Zielsprache, beispielsweise für die Generierung eines Inkrementierungsbefehls (vgl. die Zeile 7 der Programme von Bild 15.2)

$$\text{id1} := \text{id1+1} \;\rightarrow\; \texttt{INC}\ \text{id1}. \tag{16.4}$$

Der Leser wird erkennen, dass die rechte Seite keine Lexemfolge, sondern eine Tokenfolge darstellt. Die Richtigkeit derartiger Regeln ist aus den Syntaxdefinitionen der höheren Programmiersprache (Quellsprache) und der Assemblersprache (Zielsprache) formal zu beweisen.

Die prinzipielle Vorgehensweise des Codegenerators ist uns inzwischen von anderen Substitutionsprozessen her gut bekannt. Er substituiert schrittweise Stücke des Quellcodes durch Stücke des Zielcodes gemäß den Substitutionsregeln und ersetzt die syntaktischen Bezeichnervariablen (z.B. *id1*) durch die betreffenden Bezeichner. Damit ist der Zielcode erstellt.

Man beachte, dass die Vorgehensweisen von Scanner, Parser und Codegenerator an keine bestimmte Zielsprache gebunden sind, dass sie also nicht auf die Assemblersprache beschränkt sind (man lasse sich nicht durch das Beispiel (16.4) irritieren).

Wenn ein *ladbares* Programm erzeugt werden soll, das in den Hauptspeicher geladen werden kann, um ausgeführt zu werden, ist die Zielsprache eine binär codierte Maschinensprache, und an die Übersetzung muss sich das Assemblieren und das Binden anschließen [15.4]. Im ersten Schritt des Bindens werden relative Adressen eines oder mehrerer gerufener Prozeduren an relative Adressen der rufenden Prozedur gebunden und dadurch die verschiedenen Bausteinprozeduren zu einer Kompositprozedur verbunden. Dieses “Verbinden” durch den Binder wird vom Codegenerator vorbereitet. Wenn eine Prozedur eine andere ruft, muss bei der Abarbeitung der Prozessor die rufende Prozedur unterbrechen und die gerufene Prozedur “anspringen”. Dazu ist ein Sprungbefehl zur Anfangsadresse der gerufenen Prozedur erforderlich. Diesen baut der Codegenerator in den Zielcode ein, wenn der Scanner einen Prozedurruf erkannt hat. Der Befehl bewirkt eine **Unterbrechung** der Abarbeitung der rufenden Prozedur. In Kap.19.5 werden wir eine andere Methode des Verbindens von Prozeduren (ladbaren Programmen) kennen lernen, wobei zunächst ein Systemprogramm gerufen wird, welches das Verbinden einleitet.

Unterbrechungen sind auch in anderen Situationen notwendig, so bei Fehlermeldungen oder wenn Daten von oder zur Peripherie transportiert werden sollen, beispielsweise wenn Daten vom Plattenspeicher benötigt werden. Für die Bearbei-

tung von Unterbrechungen ist i.Allg. ein spezieller Aktionsschritt der zentralen Steuerschleife [13.10] vorgesehen. 14

Hiermit wollen wir die Überlegungen zur Arbeitsweise eines Übersetzerprogramms abschließen. Wir haben uns nur für Grundprinzipien interessiert. Sehr viele, auch wichtige Einzelheiten sind nicht zur Sprache gekommen. Darum sollen einige Ergänzungen in mehr oder weniger zufälliger Reihenfolge angefügt werden[17].

In Fachbüchern zur Übersetzerprogrammtechnik findet man nach der lexikalen und syntaktischen Analyse in der Regel Ausführungen zur sogenannten **semantischen Analyse**. In der Sprechweise dieses Buches könnte man für "semantische Analyse" auch "*internsemantische Kontrolle*" sagen. Es handelt sich nämlich im Wesentlichen um die Überprüfung der Anbindbarkeit programmiersprachlicher Ausdrücke an die Hardware. Beispielweise wird geprüft, ob ein syntaktisch richtiger Ausdruck wie z.B. `a+b` vom Mikroprozessor auch tatsächlich ausgeführt werden kann. Angenommen, durch das Pluszeichen wird ein Mikroprogramm aufgerufen, das entweder zwei Integerzahlen *oder* zwei Realzahlen addiert. Wenn nun `a` eine Intergerzahl und `b` eine Realzahl ist, kann der Prozessor die Addition nicht ausführen und der Ausdruck wird als falsch zurückgewiesen. Um das zu vermeiden, wird er beim Übersetzen "semantisch" überprüft und nötigenfalls eine sog. **Typanpassung**, in diesem Fall die Überführung einer Integerzahl in eine Realzahl vorgenommen.

Bei der Generierung des Maschinencodes wird oft zunächst ein **Zwischencode** erzeugt, meistens in Form des Maschinencodes einer (fiktiven) Drei-Adress-Maschine. Häufig wird der Zwischencode hinsichtlich des für die Abarbeitung erforderlichen Aufwandes *optimiert*, insbesondere hinsichtlich der Ausführungsdauer.

Ein Übersetzerprogramm, das vollständige Programme geschlossen übersetzt, heißt **Compiler**. Erfolgt die Übersetzung und Ausführung stückweise (in einzelnen übersetz- und ausführbaren Stücken) spricht man von **Interpretieren**. Ein interpretierendes Übersetzerprogramm heißt **Interpreter**. Das Wort "Interpretieren" hat den gleichen Sinn wie in Bild 2.1, nur tritt an die Stelle der *externen* Semantik (der Nutzersemantik, des Idems) die *interne* Semantik (Computersemantik). Die einzelnen Stücke des Quellprogramms, die der Interpreter interpretiert, müssen internsemantisch abgeschlossen sein, d.h. sie müssen eine *vollständige* Beschreibung dessen sein, was im Computer bei der Abarbeitung des Programmstücks vor sich gehen soll. Interpreter kommen vorwiegend in **interaktiven** Systemen, z.B. in **Dialogsystemen** zur Anwendung. In einem *interaktiven* System wechseln sich die Aktivitäten von Computer und Nutzer ab.

Abschließend sei noch einmal auf den enormen Vorteil hingewiesen, den der Mensch dem Computer gegenüber bei der (i.Allg. unbewussten) Analyse und Synthese sprachlicher Ausdrücke besitzt. Dem Menschen macht das Dekomponieren bzw. Komponieren syntaktischer Einheiten kaum Schwierigkeiten, weil er in der

---

17 Näheres siehe z.B. in [Aho 92].

Lage ist, Sätze (z.B. den Satz in Bild 5.2) “*in Gänze*” zu erfassen. Bei Benutzung der Muttersprache entfällt eine bewusste Syntaxanalyse von Sätzen, zumindest solange es sich nicht um Sätze mit sehr kompliziertem Satzbau handelt. Dem Computer fehlt die Übersicht. Er kann nur Zeichen für Zeichen lesen. Das ist der Grund für den großen programmtechnischen Aufwand des maschinellen Übersetzens. Welchen Aufwand das Gehirn beim Übersetzen treibt, wissen wir nicht.

16.5* Theorie formaler Sprachen

Der Titel von Kap.16.4 könnte die Erwartung geweckt haben, dass vom Übersetzen zwischen natürlichen Sprachen die Rede sein wird, einem sehr aktuellen Gebiet der Informatikforschung. Dass diese Erwartung nicht erfüllt worden ist, hat zwei triftige Gründe, einen negativen und einen positiven. Der negative Grund wird in Kap.17 klar werden, wenn wir uns überlegen, woran das Vorhaben scheitert, dem Computer die Rolle eines ernsthaften oder witzigen Gesprächspartners zu übertragen. Es scheitert am technischen Semantikproblem, dem Problem des Anbindens der *Humansemantik* (*Nutzersemantik*) an die *Maschinensemantik*. Hier liegt die Hauptschwierigkeit des qualifizierten Übersetzens aus einer natürlichen Sprache in eine andere, etwa das Übersetzen eines Gedichtes von Goethe ins Chinesische.

Der positive Grund besteht in der universellen Anwendbarkeit generativer Grammatiken. Sieht man vom technischen Semantikproblem ab, besteht der Kern jedes Übersetzens in der Wörterbucharbeit der lexikalen Analyse und der Codegenerierung und in der analytischen Arbeit des Erkennens der syntaktischen Struktur. Nun ist unschwer einzusehen, dass die Methode der generativen Sprachdefinition bzw. Sprachbeschreibung, die wir bisher nur auf Programmiersprachen angewendet haben, ohne grundsätzliche Veränderungen auch auf natürliche Sprachen anwendbar ist. Das ist nicht verwunderlich angesichts der Ähnlichkeit des syntaktischen Aufbaus natürlichsprachlicher und programmiersprachlicher Konstrukte. Diese Ähnlichkeit liegt der didaktischen Methode zugrunde, die stets zur Anwendung kam, wenn es sich anbot, und die darin besteht, syntaktische Sachverhalte zunächst an natürlichen Sprachen zu demonstrieren und dann auf Programmiersprachen zu übertragen. Man erinnere sich an die Gegenüberstellung der syntaktischen Struktur von Aussagesätzen (5.1) und Befehlen (5.2) in Kap.5.3., oder an die Begriffe der metasprachlichen Variablen und des Syntaxbaumes, die in Kap.5.3 parallel für natürliche Sprachen und für Programmiersprachen eingeführt wurden, und auch an Max, wie er versucht, einen englischen Satz zu übersetzen.

Tatsächlich haben die Informatiker bei der Entwicklung von Übersetzern auf die Ergebnisse der Linguisten zurückgegriffen. Die Idee der generativen Grammatik entstammt nicht der technischen Informatik, sondern der Sprachwissenschaft. Doch erst in der Hand der theoretischen Informatiker entstand eine ausformulierte Theorie, auf die wir einen kurzen Blick werfen wollen. Das ist insofern lehrreich, als an diesem

Beispiel Gemeinsamkeiten und Unterschiede zwischen informatischen Theorien und Theorien der exakten Naturwissenschaften sichtbar werden.

Die linguistischen Begriffe, die wir bisher verwendet haben, waren inhaltlich (d.h. über externe Semantik) eingeführt worden. Sie sind semantisch geladen, z.B. der Satzbegriff mit der Vorstellung konkreter Sätze und deren Aufbau. Jetzt abstrahieren wir von derartiger Semantik und definieren ganz formal:

Eine **Grammatik** ist ein 4-Tupel $G = (V,\Sigma,P,s)$. V und Σ sind zwei endliche Mengen ohne gemeinsame Elemente. V ist die Menge der sog. **Nichtterminalsymbole** und Σ ist die Menge der **Terminalsymbole**. P ist eine endliche Menge von *Regeln*, den sog. **Produktionsregeln**. Die Elemente aus P sind Substitutionsregeln der Form $u \rightarrow v$, worin u und v Ketten von Terminal- und Nichtterminalsymbolen sein können. Das Zeichen s heißt **Startsymbol**; es ist Element aus V. Die Menge aller Terminalsymbolketten, die sich aus s durch Substitutionsfolgen erzeugen lassen, ist die von G erzeugte (definierte, beschriebene) Sprache $L(G)$.

Es ist unschwer zu erkennen, dass sich die Beschreibungs- bzw. Analysemethode konkreter sprachlicher Ausdrücke mit Hilfe semantisch belegter metasprachlicher Begriffe als Interpretation des eingeführten Formalismus auffassen lässt. Die Elemente von V sind Zeichen (Buchstaben, Symbole) der *Metasprache*, in der über die *Objektsprache* gesprochen wird; darum heißen sie *Nichtterminal*symbole. Die Elemente von Σ sind Zeichen (Buchstaben, Symbole) der Objektsprache, die beschrieben (analysiert) wird; darum heißen sie *Terminal*symbole.

Der interessierte Leser kann sich den Inhalt der Grammatikdefinition an folgenden Beipielgrammatiken verdeutlichen, die beide [Schöning 95] entnommen sind, wo sie näher erläutert werden[18]. Der Pfeil in den Produktionsregeln (in den Elementen von P) entspricht dem Pfeil in Syntaxregeln, z.B. in (5.2).

Beispiel 1

$G_1 = (\{E,T,F\}, \{(,),a,+,*\}, P, E)$, wobei:
$P = \{E \rightarrow T,\ E \rightarrow E+T,\ T \rightarrow F,\ T \rightarrow T*F,\ F \rightarrow a,\ F \rightarrow (E)\}$

Mit dieser Grammatik lassen sich die korrekt geklammerten arithmetischen Ausdrücke darstellen. Beispielsweise ist die Zeichenkette $a+a*a*(a+a)$ ein Satz ("Wort") der Sprache $L(G_1)$. Anstelle von a könnten wir auch *id* schreiben. Es steht für beliebige arithmetische Variablen. Mit den Symbolen *E, T, F können* die Begriffe *Expression* (Ausdruck), *Term* und *Faktor* (im Sinne der üblichen Definition formalisierter Sprachen) assoziiert werden, d.h. ihnen *kann* metasprachliche Semantik zugeordnet werden. Dadurch wird die Grammatik "verständlicher", sie bekommt

18 Wir behalten die in [Schöning 95] verwendete Notationsweise bei, d.h. die Buchstaben aus V werden groß, die aus Σ werden klein geschrieben.

Sinn. Doch sind diese Assoziationen aus der Sicht der abstrakten Definition der Grammatik überflüssig. Die Situation ähnelt der in Kap.16.1, als wir uns das abstrakte Inferenzieren durch konkretes Inferenzieren verständlicher machten.

Beispiel 2

$G_2 = (V,\Sigma,P,\mathrm{s})$, wobei:
$V = \{\mathrm{s},B,C\}$
$\Sigma = \{a,b,c\}$
$P = \{\mathrm{s}\to a\mathrm{s}BC,\ \mathrm{s}\to aBC,\ CB\to BC,\ aB\to ab,\ bB\to bb,\ bC\to bc,\ cC\to cc\}$
Durch diese Grammatik wird z.B. das Wort $aaabbbccc = a^3b^3c^3$ generiert. Allgemein besitzen die Wörter der Sprache $L(G_2)$ die Form $a^nb^nc^n$ mit $n\geq 1$. Die Regeln sind - ebenso wie beim analytischen Rechnen und beim Schlussfolgern - Substitutionsregeln, und die dargestellte Formalisierung der Syntaxanalyse stellt eine Kalkülisierung dar, sodass einer Implementierung nichts im Wege steht.

Vergleicht man die Substitutionsregeln der beiden Grammatiken miteinander, stellt man fest, dass alle Regeln von G_1 auf der linken Seite ausschließlich Nichtterminalsymbole enthalten; jede Regel substituiert genau ein Nichtterminalsymbol. Demgegenüber können die linken Seiten einiger Regeln von G_2 mehrere Nichtterminalsymbole und eventuell zusätzlich ein Terminalsymbol enthalten. Das bedeutet in unserer früheren, semantisch beladenen Sprechweise ausgedrückt, dass zur Analyse ein mehr oder weniger umfangreicher Kontext erforderlich ist. Dementsprechend wird die Grammatik G_2 und die durch sie definierte Sprache **kontextabhängig** oder **kontextsensitiv** genannt, während G_1 **kontextfrei** genannt wird. Die Arbeit mit G_2 demonstriert unsere frühere Einsicht - diesmal rein formal -, dass Kontextabhängigkeit die Analyse erschwert. Die schrittweise Generierung von Terminalketten entspricht der Top-down-Analyse.

Man kann sich andere Grammatiken mit anderen Substitutionsregeln ausdenken, auch mit anderen Vorschriften für den Aufbau der Substitutionsregeln, also mit anderen *Metaregeln*. Um die Analyse (dem Parser die Arbeit) zu erleichtern, könnte man z.B. neben der Kontextfreiheit verlangen, dass die rechte Seite einer Regel mit einem oder mehreren Terminalsymbolen beginnt, gefolgt von einem oder mehreren Nichtterminalsymbolen. Eine solche Grammatik heißt **regulär**.

Im Sinne dieser Überlegungen hat der amerikanische Sprachwissenschaftler NOAM CHOMSKY in den 50er Jahren seine generative Transformationsgrammatik entwickelt und auf dieser Grundlage eine Hierarchie von Sprachen aufgebaut, wobei eine in der Hierarchie höhere Sprache aus der darunter liegenden durch *Präzisierung* hervorgeht, d.h. durch Hinzunahme zusätzlicher Forderungen an den Aufbau der Substitutionsregeln.

Chomsky hatte sich das Ziel gestellt, den Aufbau natürlichsprachiger Ausdrücke mit Hilfe der mathematischen Logik zu beschreiben. Der weltweite Erfolg seiner Ergebnisse bezog sich jedoch weniger auf natürliche als vielmehr auf Programmier-

sprachen, auf deren Definition und Analyse. Die heute gängigen Programmiersprachen sind fast durchweg kontextfreie, oft sogar reguläre Sprachen.

Es ist das natürliche Bestreben der Theoretiker, jede neue formale Beschreibung irgendeines Tatbestandes daraufhin zu untersuchen, ob sich auf sie vielleicht die eine oder andere bereits erprobte mathematische Methode anwenden lässt, um den neuen Formalismus besser handhaben zu können. Der Formalismus von Chomsky machte da keine Ausnahme. In diesem Fall war es die Automatentheorie, die der generativen Transformationsgrammatik ein solides mathematisches Fundament bescherte und damit die Möglichkeit, den Formalismus mit einer Menge von Theoremen zu untermauern und in Richtung Axiomatisierung weiterzuentwickeln.

Auf diese Weise entstand das Gebäude der heutigen **Theorie formaler Sprachen**. Ohne auf Einzelheiten einzugehen, sollen einige Ideen und Begriffe der Theorie erläutert werden. Der Begriff des Automaten, genauer des *abstrakten endlichen Automaten* im Sinne der Automatentheorie, ist in Kap.8.3 eingeführt worden. Wir erinnern uns, dass ein Automat ein sprachlicher Operator mit Gedächtnis ist. Das Gedächtnis dient der Aufbewahrung des inneren Zustandes, den der Automat selber berechnet. Der neue innere Zustand z' ist eine Funktion $f(x,z)$ - Folgefunktion genannt - des momentanen Eingabezeichens x und des alten Zustandes z (vgl. (8.4)). Einer der Zustände ist als *Anfangszustand* ausgezeichnet.

Wenn man die Begriffe Automat und Sprache in ihren ursprünglichen Bedeutungen einander gegenüberstellt, ist kaum zu verstehen, wie sich zwischen ihnen eine Brücke schlagen lässt. Was hat z.B. die Arbeitsweise eines Fahrkartenautomaten mit den Regeln der deutschen Sprache gemeinsam? Zwar haben die Definitionen des endlichen Automaten und der formalen Sprachen bereits zu einer erheblichen Annäherung auf abstrakter Ebene geführt, doch sind weitere Schritte erforderlich.

Der erste Schritt besteht darin, dass wir einige innere Zustände als *Endzustände* markieren. Wenn ein Endzustand erreicht wird, hält der Automat an, in Analogie zur Turingmaschine und zum Markovalgorithmus. Wenn dem Automaten irgendeine Zeichenkette eingegeben wird, geht er *möglicherweise* in einen Endzustand über. Die theorienverbindende Idee ist nun folgende. Die Gesamtheit aller Eingabeketten, die den Automaten aus dem Anfangszustand in einen Endzustand überführen, wird als Sprache aufgefasst (im Sinne einer extensionalen Definition der Sprache als Menge ihrer Wörter).

Man kann nun umgekehrt eine konkrete Sprache vorgeben und einen konkreten Automaten daraufhin untersuchen, ob er die Sprache *akzeptiert*, d.h. ob er nach Eingabe eines beliebigen Wortes der Sprache in einen Endzustand übergeht. Ist das der Fall, wird der Automat **akzeptierender Automat** der betreffenden Sprache genannt. Damit ist die gesuchte Brücke zwischen Automaten- und Sprachtheorie gefunden. Es zeigt sich jedoch, dass der Automatenbegriff einiger Erweiterungen und Präzisierungen bedarf, um effektiv und umfassend bei der Sprachanalyse eingesetzt werden zu können. Sie sind der Inhalt des zweiten Schrittes.

Der zweite Schritt in Richtung Vereinigung (*Konvergenz*) beider Theorien zielt auf die automatentheoretische Erfassung des evtl. nichtdeterministischen Charakters der Syntaxanalyse. Zu diesem Zweck ist der Begriff des **nichtdeterministischen**
15 **Automaten** erforderlich. Für ihn ist die Folgefunktion $f(x,z)$ keine eindeutige, sondern eine mehrdeutige Abbildung. Im Falle des *Turingautomaten* (der Turingmaschine) ist das gleichbedeutend mit einer Mehrdeutigkeit der Automatentabelle.

Durch diese Erweiterung verliert die Theorie nicht an Wert, genauso wie die Methoden des Rechnens und Schlussfolgerns nicht durch den inhärenten Indeterminismus an Wert verlieren, der sich stets durch zusätzliche Vorschriften überwinden lässt, beispielsweise durch eine deterministische Suchvorschrift. Ebenso lässt sich ein nichtdeterministischer Automat in einen deterministischen überführen. Auch der Indeterminismus der syntaktischen Analyse muss beim Implementieren durch geeignete Vorschriften ausgeräumt werden (wenn kein Zufallszahlengenerator zum Einsatz kommen soll). Dabei spielen ähnliche Optimierungsüberlegungen eine Rolle wie beim analytischen Rechnen nach einem nichtdeterministischen Algorithmus.

Ein anderer wichtiger Begriff der Theorie ist der des Kellerautomaten. Er wurde, ebenso wie der des Kellerspeichers, bereits in Kap.8.4.6 eingeführt. Einen Automaten, dessen Gedächtnis als Kellerspeicher organisiert ist, hatten wir Kellerautomaten genannt (vgl. die Bilder 8.12 und 13.1c). Um die Bedeutung des Kellerautomaten für die Theorie formaler Sprachen zu verstehen, verfolgen wir die Arbeit eines Computers bei der Berechnung des Wertes des Ausdrucks $3+2*(1+(5-4))$. Dieser Ausdruck ist nach den Regeln der kontextfreien Grammatik G_1 aufgebaut. Bei seiner Berechnung muss der Computer nach Erkennen des Multiplikationszeichens die begonnene Addition unterbrechen und zunächst die Multiplikation ausführen, da sie vor der Addition Vorrang hat. Dazu muss er den bereits gelesenen Teil des Additionsbefehls speichern und mit der Berechnung des zweiten Summanden beginnen. Nach Erkennen der ersten öffnenden Klammer muss er wiederum unterbrechen, das inzwischen Gelesene speichern und mit der zweiten Addition beginnen. Beim Erkennen der zweiten öffnenden Klammer wiederholt sich das Spiel zum dritten Mal. Danach kann er vom Ende her mit dem Rechnen beginnen: $5-4=1$; $1+1=2$; $2*2=4$; $3+4=7$.

Dieses Vorgehen, das mit wiederholtem Unterbrechen und Speichern beginnt, bevor die eigentliche Rechnung ausgeführt werden kann, erinnert an die rekursive Berechnung von 5! in Kap.8.4.6. Ein Blick auf Bild 8.11 legt eine wichtige Idee des Computerentwurfs nahe. Beim Berechnen geklammerter Ausdrücke ist die Verwendung eines Kellerspeichers zweckmäßig, denn dann erübrigt sich das aufwendige Speichern und Lesen über Adressen. Der Kellerspeicher liefert bei der abschließenden Berechnung (durch op_2 in Bild 8.11) die Daten in derjenigen Reihenfolge, in der sie benötigt werden. Dieser Mechanismus war bereits in Kap.8.4.6 bei der Berechnung der Fakultät besprochen worden.

Nach dieser Überlegung wird folgendes Resultat der Theorie nicht überraschen: *Kontextfreie Sprachen werden von Kellerautomaten akzeptiert*. Das ist verständlich,

denn ein Kellerspeicher ist in der Lage, das erforderliche Unterbrechen und Speichern zu erledigen, obwohl er nicht über den Adressiermechanismus verfügt.

Damit sind die Einsatzmöglichkeiten des Kellerprinzips nicht erschöpft. Die Vorteile, die es bei der Ausführung eines konsequent *funktional* formulierten Programms bringt, sind offensichtlich, denn dieses stellt einen einzigen geschachtelten Klammerausdruck dar. Aber auch bei der Ausführung imperativer Programme ist es ein wertvolles Hilfsmittel, denn auch hier gibt es das "Klammern" im weiten Sinne des Wortes (ganz abgesehen von geklammerten mathematischen Ausdrücken, die von vielen imperativen Sprachen zugelassen sind). Beispielsweise stellt das Schlüsselwortpaar `WHILE - ENDWHILE` in Bild 15.3b ein Klammerpaar dar, das die Befehle "einklammert", die bei jedem Iterationsschritt der Berechung der Sinusfunktion gemäß (15.5) auszuführen sind.

Ein weiteres Beispiel für den Einsatz des Kellerspeichers ist der *Unterprogrammruf* (*Prozedurruf*). In Kap.15.7 [15.10] hatten wir uns überlegt, dass sich das Komponieren von Kompositoperatoren dadurch programmtechnisch realisieren lässt, dass für die Bausteinoperationen Unterprogramme (Prozeduren) geschrieben werden, die vom Hauptprogramm (dem Programm der Kompositoperation) aufgerufen werden. Wenn bei der Abarbeitung des Hauptprogramms ein Unterprogrammruf erkannt wird, muss das Hauptprogramm unterbrochen und alles abgespeichert werden, was zu seiner späteren Fortführung erforderlich ist. Bei tiefer Schachtelung der Unterprogrammrufe (bei hohem Komponierungsgrad) erspart ein Kellerspeicher viel organisatorische Arbeit, denn er liefert nach Beendigung eines Unterprogramms durch Entkellerung stets die gerade erforderlichen Daten zur Fortführung des jeweils aktuellen Programms.

Der Vollständigkeit halber sei noch einmal darauf hingewiesen, dass ein konsequent funktional geschriebenes Programm eine Operatorenhierarchie nicht durch Unterprogramm-Schachtelung, sondern mittels gewöhnlicher Klammerung realisiert. Eben darin besteht die Kernidee des funktionalen Programmierens. Ein geklammerter Ausdruck stellt für den übergeordneten Klammerausdruck denjenigen Wert dar, der sich bei der Berechnung des untergeordneten Klammerausdrucks ergibt.

Betrachtet man nun noch einmal die Tätigkeit eines Computers, der ein Programm zuerst übersetzt und dann ausführt, erkennt man die Rolle des Kellerns in beiden Tätigkeiten. Beim Übersetzen eines Ausdrucks, der Klammern (im weiten Sinne) enthält, muss der Parser die Analyse auf der aktuellen Komponierungsebene, z.B. des Hauptprogramms (im natürlichsprachlichen Fall des Hauptsatzes) unterbrechen, die notwendigen Daten kellern und mit der Analyse des geklammerten Ausdrucks, z.B. des gerufenen Unterprogramms (eines Nebensatzes oder Satzteiles) beginnen. An denjenigen Stellen, an denen der Parser beim Erkennen eines Unterprogrammrufs die Analyse des Hauptprogramms unterbricht, muss der Prozessor die Abarbeitung des übersetzten Hauptprogramms unterbrechen. Er realisiert den Unterprogrammruf dadurch, dass er in den Befehlszähler nicht die Adresse des nächsten Maschinenbefehls, also nicht die inkrementierte alte Adresse, sondern die Anfangsadresse des

Unterprogramms einträgt. In Bild 13.7 entspricht dem die Übergabe der neuen Adresse aus dem Befehlsregister (BR) an den Befehlszähler (BZ). Um später bei der Absprungadresse (bei dem Befehl, an dem das Hauptprogramm verlassen wurde) fortfahren zu können, müssen alle erforderlichen Daten gespeichert, d.h. zweckmäßigerweise gekellert werden.

Schließlich soll noch eine für die KI besonders charakteristische Anwendung des Kellerspeichers erwähnt werden, das Suchen in einem Baum (z.B. in der Form des Backtracking), von dem wiederholt im Zusammenhang mit indeterministischen Verfahren die Rede war (siehe z.B. Kap.15.8). Beim Abstieg im Suchbaum muss der momentane "Zustand" des Suchprozesses in jedem Knoten (an jeder "Wegegabel") abgespeichert werden, um gegebenenfalls später die Suche an diesem Punkt, aber in anderer Richtung wieder aufnehmen zu können. Offensichtlich ist auch hier das Speichern nach dem Kellerprinzip zweckmäßig.

Die vielseitige Verwendung des Kellerprinzips hat uns etwas vom Thema abgelenkt. Wir kehren noch einmal zur Theorie formaler Sprachen zurück und nennen einige ihrer Ergebnisse. In Bild 16.5 ist zusammengefasst, welche Sprachtypen durch welche Automaten akzeptiert werden. In der zweiten Spalte ist die Bezeichnung der generierenden Grammatik bzw. des generierten Sprachtyps und in der dritten Spalte der akzeptierende Automatentyp angegeben. Die vier Sprachtypen bilden eine

Typ	Grammatik,Sprache	akzeptierender Automat
Typ 3	regulär	endlicher Automat
Typ 2	kontextfrei	Kellerautomat
Typ 1	kontextsensitiv	linear beschränkter Turingautomat
Typ 0	ohne Beschränkung	Turingautomat

Bild 16.5 Chomsky-Hierarchie. Spalte 2 - Eigenschaft der generierenden Grammatik; Spalte 3 - Klasse des akzeptierenden Automaten.

Hierarchie, die sog. **Chomsky-Hierarchie**. Jeder Hierarchieebene entspricht ein Sprachtyp. Dieser wird aufsteigend nummeriert (erste Spalte). Sprachen höheren Typs sind in Sprachen niederen Typs enthalten.

Eine Sprache, deren Grammatik den Produktionsregeln keinerlei Beschränkungen auferlegt, heißt vom Typ 0. Sprachen vom Typ 0 werden von Turingautomaten akzeptiert. Kontextsensitive Sprachen (Typ-1-Sprachen) werden von *linear beschränkten* Turingautomaten akzeptiert, das sind solche, die denjenigen Teil des Bandes, auf dem die Eingabe steht, niemals verlassen. Man erkennt, dass ein Aufsteigen von niederen zu höheren Sprachtypen einem Präzisieren entspricht. Das Präzisieren ist nach Bild 5.4 ein begriffsbildende Operation, die den Abstraktionsgrad herabsetzt. Im Gegensatz zu Bild 5.4 entspricht dem Abstrahieren in Bild 16.5 also die Abwärtsrichtung. Weitere Einzelheiten zur Theorie formaler Sprachen können

z.B. in [Schöning 95] nachgelesen werden. Wir wollen es mit der Nennung eines Satzes genug sein lassen, der das sog. Wortproblem löst. Es geht um die Frage, ob man einer Zeichenkette ansehen kann, ob sie ein Satz (ein Wort) einer bestimmten Sprache ist oder nicht.

Die Sprachtheoretiker haben diese Frage das **Wortproblem** genannt und im Rahmen der Theorie formaler Sprachen beantwortet. Die Antwort lautet: Es gibt einen Algorithmus, der die Frage beantwortet, ob eine gegebene Zeichenkette ein Wort einer von einer gegebenen kontextsensitiven Grammatik definierten Sprache ist. Mit anderen Worten, das Wortproblem ist für Typ1-Sprachen (und damit auch für Typ2- und Typ3-Sprachen) entscheidbar. Für Typ0-Sprachen ist das Wortproblem nicht allgemein entscheidbar.

Das Wenige, was über die Theorie formaler Sprachen gesagt worden ist, sollte zum einen eine Vorstellung davon geben, mit welchen theoretischen Problemen sich die Theoretiker u.a. beschäftigen und aus welchen praktischen Aufgabenstellungen diese Probleme erwachsen sind. Zum anderen sollte ein weiteres Beispiel für die Wirkungsweise des wissenschaftlichen Konvergenzprinzips [8.38] [11.1] angeführt werden. Die Wirkungsweise des Prinzips besteht hier darin, dass eine neue Theorie dadurch entsteht (*erfunden* wird), dass Begriffe und Relationen aus ganz unterschiedlichen Bereichen auf höherer Abstraktionsebene "konvergieren", dass sie miteinander kompatibel oder sogar identisch werden (vgl. Kap.8.5 und 11.1). Im vorliegenden Fall handelte es sich um Begriffe und Relationen aus dem Bereich der Sprachen einerseits und dem der Automaten andrerseits. Im Falle der in Kap.11.1 erwähnten "Erfindung" von SHANNON hatte es sich um Begriffe und Relationen aus der Schaltungstechnik und der Aussagenlogik gehandelt. In beiden Fällen gehören die konvergierenden Bereiche der Informatik an, und die Abstraktionen erfolgten innerhalb des Denkgebäudes der Informatik.

Eine nicht weniger große Rolle spielt das Konvergenzprinzip im Bereich der Physik. Beispielsweise sind die maxwellschen Gleichungen das Ergebnis einer Konvergenz zwischen elektrodynamischen und optischen Begriffen und Relationen. Es besteht aber ein chartakteristischer Unterschied in der Wirkungsweise des Prinzips in der Physik einerseits und in der Informatik andrerseits. Die Naturwissenschaften modellieren *die Natur* sprachlich, während die Informatik *das sprachliche Modellieren* sprachlich modelliert. Insofern betrifft das Konvergenzprinzip einmal das Zusammenwachsen (das Identischmachen) unterschiedlicher Begriffe von Objektsprachen, das andere mal unterschiedlicher Begriffe von Metasprachen.

Danach ist zu erwarten, dass eine "*naturwissenschaftliche*" Theorie der Informatik, die gegenwärtig noch nicht existiert, die Konvergenz objektsprachlicher und metasprachlicher Begriffe und die Bildung entsprechender Oberbegriffe zur Voraussetzung haben wird. Das würde ein Abstraktionsniveau erfordern, das oberhalb des Abstraktionsniveaus sowohl gegenwärtiger naturwissenschaftlicher als auch informatischer Theorien liegt.

17 Unterhaltung

Zusammenfassung

Der Computer kann befähigt werden, an umgangssprachlicher Konversation teilzunehmen, zur Zeit allerdings nur unter Verzicht auf die “Beliebigkeit” menschlichen Unterhaltens (Plauderns). Es gibt zwei Wege, den Computer dialogfähig zu machen: semantische Spezialisierung und semantische Verarmung. *Semantische Spezialisierung* besteht in der Beschränkung auf ein enges Spezialgebiet, auf einen kleinen Diskursbereich, sodass die Definition eines speziellen Denkkalküls und damit die Kalkülisierung des Dialogs möglich wird.

Semantische Verarmung beruht auf der Beschränkung auf Nichtssagendes. Viele Alltagsunterhaltungen zeichnen sich hierdurch aus. Der Computer kann an ihnen teilnehmen, ohne dass seine “unnatürliche” Intelligenz auffällt. Wenn er über eine größere Menge von Floskeln verfügt und vielleicht sogar noch über Regeln zur Auswahl passender Floskeln als Reaktion auf das Gerede seines Gesprächspartners, dann hat er Aussicht den *Turingtest* zu bestehen.

Inhaltsvolle Gespräche setzen *rationale Konsensfindung*, d.h. Einigung auf ein Thema, auf einen bestimmten Diskursbereich und dessen spezifische Sprache voraus und beruhen oft auf *emotionaler Konsensfindung*, auf Einstimmen auf gefühlsmäßige Gemeinsamkeiten. Letztere ist dem Computer verschlossen, wenn man davon ausgeht, dass er weder Bewusstsein noch Emotionen besitzt.

Es ist immer möglich, einen Computer in einen bild- oder klanggebenden Übertragungskanal einzubinden und nicht nur zur Umcodierung, sondern auch zur Manipulation derjenigen Bilder und Klänge zu benutzen oder zu missbrauchen, die den Übertragungskanal passieren. Der Computer kann auch am Anfang der Übertragungskette stehen und Klänge und Bilder erzeugen. Er kann fiktive, nichtexistierende Welten hervorbringen und “*virtuelle Realitäten*” schaffen.

Das *technische Semantikproblem* spielt bei der Unterhaltung durch Bilder und Klänge kaum eine Rolle, denn es werden nicht die *Bedeutungen* von Bildern oder Klängen codiert, sondern die Bilder und Klänge *selber* und zwar durch einfache Diskretisierung ohne Anwendung intelligenter Verfahren wie Abstraktion, Klassifikation oder Standardisierung.

Ursache dafür, dass die Informationsgesellschaft kräftige Wurzeln schlägt, ist weniger die künstliche Intelligenz, als vielmehr das massenhafte Eindringen von Computern in unser Leben oder, noch treffender, die Unterwanderung unseres Daseins durch ein Heer von Prozessoren.

Im Computerschach tritt ein charakteristischer Unterschied zwischen natürlicher und künstlicher Intelligenz zutage. Der Mensch begegnet der Komplexität des Schachspiels durch komplexes *Sehen* und komplexes *Denken*. Der Computer begegnet ihr sehr erfolgreich durch hohe Rechengeschwindigkeit. Dabei wird er kaum

"klüger". Der Mensch kann seine Spielqualität durch Übernahme fremder und durch Sammeln eigener Erfahrungen steigern. Die Schacherfahrungen eines Menschen sind vorwiegend globaler Natur und betreffen ganze Klassen von Spielsituationen und deren Vor- und Nachteile. Auch Erkennen und Denken des Schachspielers sind in vieler Hinsicht global. Er erfasst eine Situation und ihre Vorzüge und Gefahren global ("auf einen Blick"). Der Computer dagegen begnügt sich im wesentlichen mit dem systematischen, aber phantasielosen Durchspielen aller möglichen Fortsetzungen einer Partie und dem Vergleich der eigenen Verluste mit denen des Gegners während der verschiedenen Zugfolgen.

Darin spiegelt sich ein grundsätzlicher Unterschied zwischen dem Denken (dem intern codierten, aber nicht extern artikulierten sprachlichen Modellieren) des Menschen und dem "Denken" des Computers wider. Der Mensch ist in der Lage, gleichzeitig viele Fakten und Zusammenhänge im Bewusstsein zu halten und in seinem Denken zu berücksichtigen. Er denkt global. Er erfasst die Dinge im Überblick und trifft Entscheidungen gewissermaßen von einem höheren Gesichtspunkt aus. Der Mensch denkt in komplexen Vorstellungen, in Bildern, in Gestalten. Er *gestaltet* eine Schachpartie, er *gestaltet* einen Prozess, er *gestaltet* sein Leben.

Ein Computer kann eine Schachstellung nur feldweise und ein Bild nur pixelweise, genauer nur computerwortweise betrachten, denn er denkt algorithmisch. Kurz: *Der Computer denkt in Computerwortfolgen, der Mensch kann anschaulich und in globalen Zusammenhängen denken.*

17.1 Der Computer als Gesprächspartner

Wir verlassen nun den sicheren Boden erprobter und anerkannter Methoden der künstlichen Intelligenz und wenden uns Problemen zu, für deren Lösung zwar auch ein umfangreiches Arsenal methodischer und programmtechnischer Hilfsmittel existiert, doch sind die Verfahren oft noch unzureichend theoretisch untermauert und praktisch wenig ausgereift. Die Überlegungen des Kapitels 17 können sich in weit geringerem Maße auf Autoritäten berufen als die bisherigen Überlegungen. Es haftet ihnen ein gewisser spekulativer Zug an. Doch wäre es anders nicht möglich, die Grenzen des gesicherten Wissens und der sicheren Methoden zu überschreiten. Wir müssen sie überschreiten, wenn wir den Computer befähigen wollen, sich *mit uns* zu unterhalten oder *uns zu unterhalten* im ganz alltäglichen Sinne des Wortes "Unterhaltung". Das Kapitel 17 lässt die Exaktheit und logische Folgerichtigkeit der Gedankengänge vorangehender Kapitel vermissen. Doch fällt es nicht nur in dieser inhaltlichen Hinsicht aus dem Rahmen. Das Wort "Unterhaltung" charakterisiert nicht nur den Gegenstand, über den gesprochen wird, sondern auch den Stil, der eher unterhaltsam als streng wissenschaftlich zu nennen ist.

Als Charakterisierung des Gegenstandes hat das Wort "Unterhaltung" zwei Bedeutungen, eine aktive und eine passive. Ein Mensch kann sich, selber aktiv, mit

einem anderen unterhalten; dann spricht man auch von *Konversation* zwischen zwei oder mehreren Gesprächspartnern. Oder ein Mensch kann sich, selber passiv, von einem anderen unterhalten lassen. Falls ein Mensch viele Zuhörer unterhält, z.B. im Fernsehen, werden häufig die englischen Wörter *Entertainer* und *Entertainment* benutzt.

Wenn die Rolle des Computers als Unterhaltungspartner (in beiden Bedeutungen) in einem Umfange behandelt werden soll, der dieser seiner Rolle in der Informationsgesellschaft entspricht, wären dafür mehrere Bücher erforderlich. Wir beschränken uns auf eine kurze Beantwortung der Frage, was Informatik mit Unterhaltung zu tun hat und welche prinzipiellen Probleme zu lösen sind, damit der Computer als Unterhaltungspartner fungieren kann. Wir beginnen mit der Frage, unter welchen Bedingungen und in welchem Maße ein Computer die Rolle eines Gesprächspartners übernehmen kann.

Im Jahre 1950 veröffentlichte der englische Wissenschaftler ALAN M.TURING, ein Pionier der Informatik, der Erfinder der Turingmaschine, einen Artikel unter dem Titel "Computing Machinery and Intelligence" [Turing 50]. Darin schlug er einen Test vor, der nach ihm **Turingtest** genannt wird. Der Test soll auf die Frage Antwort geben, ob bzw. wann es sinnvoll ist zu sagen, dass der Computer *denken* kann. Der Test läuft folgendermaßen ab.

Eine Versuchsperson ist über ein Kommunikationsgerät, z.B. über einen Fernschreiber, mit einem Partner verbunden, den er weder sehen noch hören kann. In einem "Gespräch" soll er feststellen, ob sein Partner ein Mensch oder ein Computer ist. Wenn er in einer Reihe unterschiedlicher Gespräche den Computer nicht als Computer erkennt, sondern annimmt, dass er mit einem Menschen spricht, ist es offenbar gerechtfertigt, zu sagen, dass der Computer denken kann.

Im Jahre 1966 veröffentlichte ein anderer Pionier der Informatik, der in den USA lebende Wissenschaftler J.WEIZENBAUM, ein Programm namens *Eliza* [Weizenbaum 66]. Dieses Programm befähigt einen Computer die Rolle eines Arztes zu spielen, der sich mit einem neuen Patienten über dessen Anliegen, Beschwerden und persönliche Besonderheiten unterhält, so wie es zu Beginn eines Diagnosegesprächs üblich ist. Der Computer spielt seine Rolle überraschend gut, zumindest auf den ersten Blick, sodass man geneigt sein könnte, zu sagen, dass der Computer denkt, da er offenbar in der Lage ist, den Turingtest zu bestehen.

Analysiert man ein Gespräch mit Eliza genauer, kann man den Eindruck gewinnen, dass Weizenbaum sich mit seinem Programm den Spaß gemacht hat, den Turingtest ad absurdum zu führen. Zunächst stellt man fest, dass der Diskursbereich und die Rolle des Computers so definiert sind, dass der Computer im wesentlichen Fragen stellt, wofür ein sehr begrenztes Wissen ausreicht. Freilich muss er seine Fragen "intelligent" stellen, d.h. vernünftig aus der Sicht des Patienten. Um das zu erreichen, wendet das Programm einen einfachen Trick an. Es formuliert die nächste Frage in der Regel durch Konkretisierung oder Verallgemeinerung der vorangehenden Antwort des Patienten.

Auf die Aussage des Patienten “Ich leide sehr an Kopfschmerzen” könnte der Computer beispielsweise fragen “Seit wann haben Sie die Beschwerden?” oder “Wann treten die Beschwerden vorzugsweise auf?” oder “Sind Ihnen ähnliche Fälle in Ihrer Verwandtschaft bekannt?” Für derartige Fragen lassen sich “Produktionsregeln” implementieren, nach denen sich Fragen aus vorherigen Aussagen produzieren lassen, ganz ähnlich wie im Falle der Produktion neuer Aussagen beim Schlussfolgern oder bei der Produktion von Wörtern oder Sätzen einer Sprache nach den Produktionsregeln einer generativen Grammatik.

Zum Produzieren der Fragen oder Aussagen des Arztes benutzt der Computer Wissen, das beispielsweise in Form von Tafeln (in diesem Fall Frage-Antwort-Tafeln) oder von Regeln und Fakten abgespeichert sein kann. Damit der Computer auf möglichst viele Aussagen des Patienten “vernünftig” reagieren kann, muss er sie in eine geeignete Standardform bringen, sodass die Regeln und Tafeln anwendbar werden. Andrerseits sollte er für seine eigenen Standardantworten mehrere Artikulationsmöglichkeiten bereithalten, damit der Patient keinen Verdacht schöpft. Auf diese Weise kann das “Denken und Sprechen” des Arztes kalkülisiert werden, wenn auch nur in sehr begrenztem Umfange.

Im Laufe des Gesprächs kann *Eliza* (d.h. das Programm, das so tut, als sei es ein Arzt) immer besser auf den Patienten eingehen und immer verblüffendere Fragen stellen, denn während des Gesprächs erweitert das Programm Eliza sein einprogrammiertes allgemeines und medizinisches Faktenwissen um spezifisches Wissen, das ihm vom Patienten mitgeteilt wird. Es baut gewissermaßen eine gemeinsame Wissensbasis auf und damit eine gemeinsame externe Semantik, die es so artikulieren kann, dass der Patient Elizas Artikulationen richtig versteht und sie durchaus für vernünftig hält.

Nach dieser Analyse mag es scheinen, das Programm Eliza sei allzu erkünstelt und seine verblüffenden Fähigkeiten seien wenig aussagekräftig hinsichtlich der Möglichkeiten der KI. Dennoch ist der Wert des Programms nicht zu unterschätzen, denn es macht deutlich, wo die entscheidenden Schwierigkeiten liegen, denen die künstliche Intelligenz nicht gewachsen ist, und welches die Tricks sind, mit deren Hilfe sie sich scheinbar überwinden lassen, aber eben nur scheinbar. Bei diesen Tricks handelt es sich um zwei ernst zu nehmende Vereinfachungen, die nicht übersehen werden dürfen, wenn die Möglichkeiten der künstlichen Intelligenz beurteilt werden sollen. Wie wir gleich sehen werden, handelt es sich bei den beiden Tricks um *semantische Spezialisierung* und *semantische Verarmung*.

In der medizinischen Praxis kommen zunehmend interaktive Diagnosesysteme zur Anwendung. Ein solches System schlägt dem Arzt geeignete Fragen an den Patienten vor. Die Antwort wird über ein Eingabegerät (evtl. über einen akustischen Eingang einschließlich Spracherkenner) eingespeichert. Die Antworten werden klassifiziert, d.h. jede Antwort wird einer abgespeicherten Standardantwort zugeordnet. Das System stellt seine Diagnose. Gegebenenfalls stellt es mehrere Diagnosen mit Wahrscheinlichkeitsangaben. Die Hilfe des Diagnosesystems ist umso wertvoller, je

weitgehender das Denken des Arztes kalkülisiert (algorithmisiert) ist und je mehr Wissen und Erfahrungen abgespeichert sind. Zweifellos spielt die Erfahrung beim Diagnostizieren eine hervorragende Rolle.

1

Die Verwendbarkeit des Computers als Dialogpartner des Arztes beruht - ebenso wie im Falle von Eliza - auf Kalkülisierung, die durch *Klassifizierung* und *Standardisierung* ermöglicht wird. In erster Linie werden die Patientenantworten standardisiert. Das medizinische Wissen wird klassifiziert, um es schnell abrufbar einspeichern zu können. Ferner können auch Erfahrungen des Arztes klassifiziert und gespeichert werden. Das Kalkülisieren betrifft das Ableiten von Diagnosen aus den (standardisierten) Aussagen des Patienten, aus dem medizinischen Wissen und möglicherweise aus der Erfahrung des Arztes. Voraussetzung für eine sinnvolle Standardisierung ist inhaltliche, d.h. **semantische Spezialisierung**, d.h. die Beschränkung auf ein enges Spezialgebiet, auf einen kleinen Diskursbereich.

Derartige computerunterstützte Diagnosesysteme können als Erweiterung von Eliza aufgefasst werden. Der Kern beider Systeme ist - wie der aller schlussfolgernden Systeme - das regelbasierte Produzieren neuer Aussagen (bzw. Fragen) aus bekannten Aussagen. Voraussetzung ist, wie gesagt, Klassifizierung und Standardisierung der Aussagen, m.a.W. Festlegung einer endlichen Menge zugelassener Aussagen. Die Endlichkeit der Aussagemengen ist im Grunde nichts anderes als die Endlichkeit der zu berechnenden Funktionstafeln.

An dieser Stelle ist eine Zwischenbemerkung hinsichtlich der Standardisierung am Platze, die wir als Voraussetzung für den Mensch-Maschine-Dialog erkannt haben. Tatsächlich setzt jeder Dialog zwischen Menschen und überhaupt jede sprachliche Kommunikation Standardisierung voraus. Sie beginnt mit dem Hervorbringen von *Standardlauten* durch Tiere z.B. bei Gefahr. Der Mensch begann, Standardlaute zu Wörtern und Sätzen zu komponieren. Später erfand er *Standardzeichen*, die er den Standardlauten zuordnete. Davon war in Kap.5.1 [5.1] im Zusammenhang mit der Idemobjektivierung die Rede.

Die Herausbildung der Buchstabenschrift aus ursprünglichen bildlichen, d.h. analogen (in zweifachem Sinne des Wortes: "entsprechenden" und "nichtsprachlichen") Darstellungen über Bilderschriften zeigt, dass zwischen analoger nichtsprachlicher und digitaler sprachlicher Artikulierung von Bewusstseinsinhalten, m.a.W. zwischen Urrealemen und Zeichenrealemen ein kontinuierlicher Übergang besteht (siehe Bild 2.1 und 3.1). Die Grenze zwischen sprachlicher und nichtsprachlicher "Codierung" ist also nicht scharf. Beispielsweise ist die Frage berechtigt, ob die malerischen Motive bei Chagall (Eselskopf, Geiger) oder die klanglichen Motive bei Richard Wagner (Ringmotiv, Siegfriedmotiv) analoge Darstellungen oder codierende Zeichen sind.

Nach diesem Exkurs über die Wurzeln des Standardisierens kehren wir zurück zum Computer als Gesprächspartner. Auf Alltagsunterhaltungen, die jeden Diskursbereich berühren können, ist das Verfahren der Klassifizierung und Standardisierung kaum anwendbar. Nichtsdestoweniger kann der Computer auch hier als Dialogpart-

ner auftreten, allerdings aus einem ganz anderen Grund. Tatsächlich kommt Elizas Methode im Alltag gar nicht so selten zum Einsatz. Beispielsweise kann sie erfolgreich von jemandem angewendet werden, der verheimlichen will, dass er nicht weiß, wer die Person ist, die ihn soeben als alten Bekannten begrüßt und in ein Gespräch verwickelt hat. Er wird möglichst nichtssagende Antworten geben und selber Fragen stellen, deren Beantwortungen seinem Gedächtnis auf die Sprünge helfen soll. Ähnlich nichtssagend können Gespräche sein, mit denen nichts weiter bezweckt wird, als das Redebedürfnis zu befriedigen oder Zeit totzuschlagen, z.B. auf Kaffeekränzchen oder am Stammtisch. In solchen Gesprächen ist der Wert dessen, was gesagt wird, oft vernachlässigbar gering, die Gespräche sind kaum "informativ". Ein Computer könnte sich nach der *Eliza-Methode* prächtig an solchen Gesprächen beteiligen.

2 Die Verwendbarkeit des Computers als Gesprächspartner beruht diesmal auf einer relativen Sinn*losigkeit*, das heißt, auf einer Armut an Sinn, an Semantik, auf **semantischer Verarmung**. *Damit haben wir zwei Wege erkannt, den Computer dialogfähig zu machen:* ***semantische Spezialisierung*** *und* ***semantische Verarmung***. Das eine wie das andere ermöglicht die notwendige Standardisierung. Keiner der beiden Wege kann die Lösung sein, wenn man den Computer befähigen will, sinn*volle* (inhaltsvolle) Gespräch zu führen. Einem solchen Unterfangen stehen nach wie vor zwei miteinander verkoppelte Schwierigkeiten entgegen. Die Gesprächspartner müssen eine gemeinsame Wissensbasis besitzen und eine gemeinsame Sprache sprechen, in der sie sich unterhalten. Auf beide Schwierigkeiten soll in aller Kürze eingegangen werden.

Zunächst soll am Beispiel von Homonymen (Wörtern mit mehreren Bedeutungen) illustriert werden, wie wichtig die Gemeinsamkeit des Diskursbereiches ist, über den sich zwei Menschen unterhalten. Den Satz "Der Absatz gefällt mir" wird jedermann sofort richtig verstehen, allerdings völlig unterschiedlich, je nachdem, ob von einem Buch oder einem Schuh die Rede ist. Ähnliche Beispiele lassen sich für andere Homonyme anführen. Die Richtigkeit der Interpretation beruht auf dem Umstand, dass Gesprächspartner sich automatisch auf eine, dem Diskursbereich entsprechende "gemeinsame Sprache" einigen, genauer gesagt auf die Bedeutungen (Bewusstseinsinhalte, Ideme), die durch die Sprache artikuliert werden.

Da zwischen den Idemen der Partner ausreichende Entsprechung vorausgesetzt werden muss, können wir von Semantik sprechen [5.5]. Gesprächspartner müssen sich semantisch aufeinander *einstimmen*, sie müssen einen *semantischen Konsens* finden ("semantisch" als Adjektiv zu "Konsens" ist genau genommen überflüssig). Die Konsensfindung betrifft ausschließlich die Partner und die Dauer des Gesprächs. Insofern unterscheidet sie sich von *semantischer Objektivierung*, die universellen Charakter hat.

Wenn der Computer die Rolle eines Dialogpartners beim Lösen mathematischer wie nichtmathematischer Aufgaben spielen kann, so liegt der Grund dafür in der semantischen Objektivierung durch Anbindung an eine formale Semantik, wodurch

der "semantische Konsens" zwischen Mensch und Maschine gewährleistet ist. Andrerseits ist die Notwendigkeit der semantischen Konsensfindung gerade der Grund dafür, dass der Computer in der Regel *nicht* in der Lage ist, die Rolle eines Gesprächspartners zu spielen.

Man beachte, dass der Konsens nicht die Sprache selber betrifft, sondern das Wissen über den Diskursbereich. Allerdings kann die gegenseitige Einstimmung über den Bereich des *Wissens* hinausgehen und auch das *Gefühl* betreffen, beispielsweise in einem Gespräch über die Schönheit der Natur, in einem Gespräch zwischen Streithähnen oder zwischen Liebenden. Hier muss der Computer wohl vollständig seine Segel streichen. Wenn durch Sprache Emotionen "mitgeteilt" werden, wollen wir von *emotionaler Semantik* sprechen. Auch hinsichtlich emotionaler Semantik müssen Gesprächspartner einen Konsens finden, um sich zu verstehen. Man könnte von "Konsensibilisierung" sprechen. Wir vereinbaren: *Wenn Konsensfindung das gemeinsame Wissen von Gesprächspartnern betrifft, sprechen wir von* ***rationaler Konsensfindung****, wenn sie gemeinsame Gefühle betrifft, von* ***emotionaler Konsensfindung***. Letztere ist dem Computer verschlossen, wenn man davon ausgeht, dass er weder Bewusstsein noch Emotionen besitzt. 3

Es ist aber nicht nur die Semantik, sondern auch die Syntax der Sprache, die es dem Computer eventuell sehr schwer macht, an Alltagsgesprächen teilzunehmen. Das hat vor allem zwei Ursachen, die relativ komplizierte Grammatik natürlicher Sprachen und Verstöße gegen sie.

Natürliche Sprachen als Produkte der Evolution richten sich nicht nach Chomskys Erkenntnissen, sie sind leider nicht von Typ 3, 2 oder 1 (siehe Bild 16.5 in Kap. 16.5). Die syntaktische Analyse natürlichsprachiger Sätze stößt auf erhebliche Schwierigkeiten. Doch gibt es keinen stichhaltigen Grund, der ihre Möglichkeit prinzipiell ausschließt, abgesehen von dem notwendigen Aufwand an Speicherplatz und Rechenzeit. Die technische Entwicklung hat immer wieder scheinbar Unmögliches möglich gemacht.

Um den Computer konversationsfähig zu machen, könnte man auf die Idee kommen, eine Umgangssprache mit einfachen Syntaxregeln zu entwickeln. Doch erscheint es zweifelhaft, ob auf diese Weise die Ausdrucksstärke natürlicher Sprachen erreicht werden kann. Gegen diesen Zweifel sprechen die Erfolge der sogenannten *Plansprachen*. Das sind planmäßig konstruierte Umgangssprachen mit relativ einfacher Grammatik, sodass sie leicht analysierbar und erlernbar sind.

Die bekannteste Plansprache ist **Esperanto**. Sie wurde vor etwa 100 Jahren von dem polnischen Augenarzt LUDWIG ZAMENHOF entwickelt mit dem Ziel, der Welt eine internationale Kommunikationssprache zur Verfügung zu stellen. Dieser Gedanke kann heute beim Zusammenwachsen der Welt, was eine einheitliche Sprache erforderlich macht, politische Bedeutung erlangen. Denn die Einsetzung einer ethnischen Sprache, z.B des Englischen, als internationale Verkehrssprache könnte nationale Gefühle verletzen und das Zusammenwachsen behindern, während Esperanto es eher fördern würde.

Shakespeare, Goethe und Puschkin sind in Esperanto übersetzt worden. Die Syntax lässt das trotz ihrer Einfachheit zu. Sogar ein in Esperanto übertragenes Gedicht kann etwa so verstanden werden, wie das Original. Offenbar hängt die Breite der Interpretationsmöglichkeit nicht oder nur wenig vom Kompliziertheitsgrad der Syntax der verwendeten Sprache ab. Das ist insofern verständlich, als die Interpretation auf der Grundlage rationaler und emotionaler Konsensfindung des Lesers mit dem Dichter erfolgt, was nicht Angelegenheit der Sprache, sondern des interpretierenden Menschen ist.

Ein anderer Weg zu einer breiten internationalen Kommunikation wäre die maschinelle Übersetzung zwischen allen beteiligten ethnischen Sprachen. Bei n beteiligten Sprachen wären $2n(n\text{-}1)$ Übersetzer erforderlich. Diese Zahl lässt sich erheblich herabsetzen, wenn eine einheitliche Zwischensprache eingeführt wird. Ein Quelltext wäre zunächst in die Zwischensprache und anschließend aus dieser in die Zielsprache zu übersetzen. In diesem Fall wären nur $2n$ Übersetzer erforderlich. An der Verwirklichung dieser Idee wird gearbeitet, wobei auch Esperanto als Zwischensprache benutzt wird.

Angenommen, alle grammatikalischen und lexikalischen Probleme des maschinellen Sprachverstehens seien gelöst. Dann bleibt immer noch eine große Schwierigkeit bestehen, von rationaler und emotionaler Konsensfindung ganz abgesehen. Im alltäglichen Gespräch kümmert man sich nämlich wenig um grammatikalische Regeln. Auf Schritt und Tritt wird gegen sie verstoßen, allerdings vorwiegend gewohnheitsmäßig, sodass auch für die Verstöße Regeln aufgestellt und implementiert werden können. Nehmen wir als Beispiel die Replik: “Das kannst du doch nicht machen” - “Doch” - “Nein”. Diese Worte reichen für die Verständigung in einer bestimmten Situation aus, auch wenn nur Bruchstücke ganzer Sätze artikuliert werden. Die Antwort “Doch” steht für den vollständigen Satz: “Das kann ich *doch* machen”, der gedanklich ergänzt wird. Was das erste “Das” bedeutet, geht aus der Situation (aus dem Kontext) hervor und wird ebenfalls gedanklich ergänzt. Die richtigen Ergänzungen sind aufgrund der gemeinsamen Einstimmung auf einen bestimmten Diskursbereich und aufgrund der Kenntnis der vorangegangenen Replik möglich.

Es gibt keinen Grund anzunehmen, dass der Computer prinzipiell nicht in der Lage ist, derartige Ergänzungen vorzunehmen, freilich wieder unter der Voraussetzung, dass ausreichend Zeit und Speicherplatz zur Verfügung steht. Wie groß der Aufwand werden kann, wird klar, wenn man bedenkt, dass jedes Wissenselement unter einer Adresse abgespeichert werden muss, und dass der Prozessor nur über Adressen auf sein Wissen zugreifen kann.

Die Evolution hat weit effektivere Methoden hervorgebracht, die es dem Menschen erlauben, nach Belieben und scheinbar unmittelbar und uneingeschränkt mit seinem gesamten Wissen zu hantieren. Die Hantierung mit Gedächtnisinhalten scheint der entscheidende Punkt zu sein, in dem sich Mensch und Maschine hinsichtlich ihrer Fähigkeiten zur Konversation unterscheiden. Es stellt sich die Frage, ob

dies vielleicht überhaupt der entscheidende Punkt ist hinsichtlich der unterschiedlichen intellektuellen Fähigkeiten des Menschen und des Computers. Wir kommen darauf in Kap.17.3 zurück.

17.2 Der Computer als Unterhalter

Die Rolle des Computers als Unterhalter, als Spielgefährte oder einfach als Zeittotschläger ist schon heute groß und sattsam bekannt. Sie wird sicher noch erheblich zunehmen, bis der menschliche Selbsterhaltungstrieb sich ernsthaft dagegen zur Wehr setzen wird. Wir werden uns nicht für die vielen Facetten dieser Rolle des Computers interessieren, sondern lediglich für die Frage, was Unterhaltung per Computer mit Informatik zu tun hat.

Wir hatten die Informatik als Lehre vom aktiven sprachlichen Modellieren definiert und erinnern an den Auskunftsautomat, der gewünschte Informationen schriftlich oder mündlich erteilt, als Beispiel für aktive (d.h. vom System selbst vollzogene) Artikulierung von Aussagen, m.a.W. für aktives sprachliches Modellieren eines bestimmten Diskursbereiches.

Wir erweitern das Beispiel um eine CD, auf der beispielsweise ein Gesundheitslexikon oder Goethes Faust abgespeichert ist. Der Besitzer der CD samt eines entsprechenden Computers kann in dem Lexikon oder im Faust herumblättern und nach Belieben lesen. Dabei gibt der Computer Zeichen aus, die der Nutzer interpretieren kann. Dieser empfängt also *Information*, genauer: er empfängt Zeichenketten, die Bewusstseinsinhalte aktivieren (Ideme auslösen) und dadurch zu Information werden [1.3]. Die Tätigkeit des Computers (die technische "Informationsverarbeitung") beschränkt sich dabei im wesentlichen auf das Auffinden der gewünschten Textstellen und das Umcodieren aus dem Binärcode der CD in den Buchstabencode des Bildschirms. (Wir nehmen an, dass die CD nicht analog sondern digital geprägt ist. Beides ist möglich.)

Was aber findet statt, wenn der Computer stehende oder bewegte Bilder zeigt oder wenn er Musik macht, wenn also seine Mitteilungen nicht sprachlicher, sondern analoger (nichtsprachlicher) Natur sind? Um das Problem in seiner ganzen Breite zu erkennen, gehen wir davon aus, dass unser Computer Video- und Audio-Ausgänge besitzt. Er kann also Bitketten ausgeben, welche die Frequenzen und Amplituden von Licht- und Schallwellen codieren. Wenn diese Informationen auf einem peripheren Speicher des Computers gespeichert sind, z.B. auf einer CD, und wenn der Computer mit einem Fernsehgerät verbunden wird, spielt er die Rolle eines Videorekorders.

Wir interessieren uns zunächst für die visuelle Information und fragen, welche Formen sie auf dem Wege von der CD bis zum Nervensystem annimmt. Genauer gesagt, wir fragen nach der Transformation der Realeme, die letzten Endes vom Menschen interpretiert und dadurch zu Information werden. Auf der CD und im

Computer ist die Information binär codiert. Auf dem Bildschirm ist sie durch die Lage der beteiligten Bildpunkte (*Pixel*), durch diskrete Wellenlängenbereiche und Helligkeitswerte, mit denen die Pixel leuchten und - im Falle bewegter Bilder - durch diskrete Zeitpunkte codiert, zu denen sie leuchten. Eine Fernsehkamera zerlegt einen aufgenommenen Vorgang in eine diskrete Folge von Bildern und jedes Bild in eine diskrete Folge von Pixeln, die zeilenweise das Bild überdecken. Man nennt diese Art der Diskretisierung *Rasterung*. Farbe und Helligkeit der Pixel werden i.d.R. sequenziell übertragen und auf dem Bildschirm des Empfängers wieder zu dem gesendeten Bild zusammengefügt.

Sämtliche bildgebenden Größen (Zeitpunkt, Ort, Farbe, Helligkeit) werden so fein, in so kleinen Schritten diskretisiert (digitalisiert), dass der Betrachter sie als kontinuierliche (analoge) Größen empfindet, er nimmt kontinuierlich verlaufende Linien, Färbungen, Schattierungen und Bewegungen wahr. Das Entsprechende gilt für die Wiedergabe von Musik. Die Introspektion sagt uns, dass unsere auditiven und visuellen Eindrücke und Empfindungen in jedem Fall kontinuierlicher Natur sind, einerlei, wie sie zustande kommen und welcher Natur ihre Quellen (z.B. die reale Umgebung oder der Fernseher) sind.

Die Kontinuität der Empfindungen entspricht aber nicht der physiologischen Realität. Das kontinuierliche Bild der Außenwelt oder auch das scheinbar kontinuierliche Bild des Fernsehers, das die Linse auf die Netzhaut projiziert, wird durch die Struktur der Netzhaut diskretisiert. Es wird ein Bild perzipiert, das ähnlich wie das des Fernsehers aus Pixeln besteht. Jedem Stäbchen bzw. Kolben der Netzhaut entspricht ein Pixel. Die "Digital-Analog-Konvertierung" zu einer kontinuierlichen Empfindung ist das Produkt der Gehirntätigkeit. Ob eine solche Konvertierung neurophysiologisch tatsächlich stattfindet, ist zu bezweifeln. Angesichts der Funktionsweise der Neuronen ist es wahrscheinlich, dass eine *Umcodierung* stattfindet, wobei das Gehirn sowohl statisch als auch dynamisch codiert [9.1]. Experimentell ist dies gegenwärtig nicht eindeutig belegbar. Mit anderen Worten, es fehlt der experimentelle Beweis, dass Interpretieren (die Pfeile 1 und 5 in Bild 2.1) mit Codieren verbunden ist, dass also Urideme materielle Träger in Form neuronaler Zustände besitzen.

Unversehens ist unsere Frage zu einer sehr grundsätzlichen geworden: *Entsprechen* ***mentalen*** *Zuständen* ***nervale*** *codierende Zustände?* Im Nachwort werden einige philosophische Aspekte dieser Frage beleuchtet, ohne sie zu beantworten. Wir wissen also nicht, ob es sich bei den Pfeilen 2, 4 und 5 in Bild 2.1 um Verarbeitung von Zeichenrealemen handelt. Hingegen beinhaltet Pfeil 2 mit Sicherheit Codieren und Pfeil 3 Zeicheninformationsverarbeitung. Anders ausgedrückt, der internen wie externen Codierung liegt Standardisierung zugrunde. Die zu artikulierenden Bewusstseinsinhalte werden durch Abstraktion in Begriffe, in *Standardbedeutungen* und diese durch Benennung in *Standardzeichen* überführt. Mit diesem "Trick" löst die Natur den "*Widerspruch zwischen der kontinuierlichen Natur des Gedachten und der nichtkontinuierlichen Natur des Denkens*" (teleonomisch gesprochen).[1]

Ebenso wie in Kap.17.1 [1] kommen wir zu dem Ergebnis, dass Standardisierung Voraussetzung der Kommunikation ist. Früher hatten wir die Idemobjektivierung [5.1] als Voraussetzung genannt. Offenbar handelt es sich um zwei Seiten ein und desselben Sachverhaltes. Aus der Sicht der Kommunikation zwischen Menschen lag es näher, von Idemobjektivierung bzw. von semantischer Objektivierung [5.6] zu sprechen. Aus der Sicht der *technischen* Kommunikation liegt der Begriff der Standardisierung näher. Im Falle *menschlicher* Kommunikation schließt sich an die interne codierte Verarbeitung das externe Codieren (Pfeil 3 in Bild 2.1) an, z.B. durch Sprechen oder durch Schreiben mit der Hand, also durch kontinuierliche (analoge) Prozesse.

Die letzten Überlegungen zur Kommunikation gehen bereits über die ursprüngliche Fragestellung hinaus, die das Artikulieren nicht einschloss, wie z.B. das Nacherzählen einer gehörten CD oder einer Sendung. Doch demonstriert die Einbeziehung des Artikulierens besonders anschaulich den wiederholten Wechsel zwischen analoger und digitaler Informationsdarstellung, ja sogar die gleichzeitige Verwendung beider Darstellungsformen. Handschriftlicher Text ist z.B. eine kontinuierliche Darstellung von sprachlich codierten Bewusstseininhalten (Idemen), und das gerasterte Fernsehbild ist für den Zuschauer eine kontinuierliche Darstellung.

Der Wechsel zwischen analoger und digitaler Darstellung war bereits am Ende von Kap.4.2 diskutiert worden, jedoch aus rein technischer Sicht. Dort war auch der Begriff der Konvertierung zwischen den beiden Darstellungsformen eingeführt worden. Wir sahen, dass Konvertierung (in beiden Richtungen) das Zusammenschalten von Analog- und Digitalrechnern und so den Einsatz von Digitalrechnern bei der Lösung analoger Steuerungsaufgaben ermöglicht. Wir wollen uns nun überlegen, welche Möglichkeiten das Konvertieren dem Computer als Unterhalter eröffnet.

Die entscheidende Einsicht ist diese: Es ist immer möglich, einen Computer in einen bild- oder klanggebenden Übertragungskanal einzubinden. Diese Einsicht legt den Gedanken nahe, den Computer nicht nur zum Umcodieren und Suchen zu benutzen, sondern auch zur Manipulation derjenigen Bilder und Klänge, die ihn passieren. Per Programm lässt sich alles machen, aus einem X ein U, aus gelb grün, aus schön häßlich, aus Bach Jazz, aus meinem Gesicht das eines Papagein und aus seiner Stimme meine.

Der Computer kann auch am Anfang der Übertragungskette stehen und “selber”, durch Abarbeitung entsprechender Programme, Klänge und Bilder erzeugen. Er kann fiktive, nichtexistierende Welten hervorbringen. Die Illusion, dass es sich um *wirkliche* Welten handelt, kann dadurch bedeutend gesteigert werden, dass dem Betrachter die Möglichkeit gegeben wird, auf die vom Computer produzierte Welt einzuwirken, mit ihr in Kontakt zu treten und sich in ihr zu bewegen. Man spricht dann von **virtueller Realität**, und der Raum, in dem man sich bewegt, wird **Cyberspace**

1 Zitat aus Kap.4.1 [4.1].

genannt. Bereits der selbstvergessene Computerspieler befindet sich in einer anderen "Wirklichkeit", selbst wenn die Möglichkeiten, auf diese einzuwirken, sehr begrenzt sind.

Die virtuelle Realität kann noch erheblich realer erscheinen, wenn auch die anderen Sinnesorgane einbezogen werden. Hinsichtlich der taktilen Rezeption hat man es zu einigen Erfolgen und in bestimmter Hinsicht sogar zu beachtlichen Erfolgen gebracht. So kann der Computer in bereitwilligen Versuchspersonen unter Verwendung geeignet konstruierter Vorrichtungen den Orgasmus auslösen, natürlich auf optimierte Weise. Ob das eine positive Entwicklung ist und ob der Mensch sich auf diesem Wege dem Ebenbild Gottes nähert, ist fragwürdig.

Mit diesen Bemerkungen sollte nur ein kleines Schlaglicht auf das geworfen werden, was der Menschheit bevorstehen könnte, wenn sie alle Unterhalterpotenzen des Computers ausschöpfen würde.

So interessant die Unterhalterrolle des Computers in psychologischer und sozialer Hinsicht auch sein mag, aus der Sicht der Informatik als Wissenschaft vom aktiven sprachlichen Modellieren ist sie relativ uninteressant, denn die zentralen Fragen werden kaum berührt, weder die der Semantik noch die der künstlichen Intelligenz. Das technische Semantikproblem spielt bei der Unterhaltung durch Bilder und Klänge keine Rolle, denn es werden nicht die *Bedeutungen* von Bildern oder Klängen codiert, sondern die Bilder und Klänge *selber* und zwar durch einfache Diskretisierung ohne Anwendung intelligenter Verfahren wie Abstraktion, Klassifikation oder Standardisierung. Der Programmierer teilt dem Computer kaum etwas von seiner eigenen Intelligenz mit, im Gegensatz zu den in den Kapiteln 15 und 16 dargelegten KI-Methoden, in denen der Computer durch *Intelligenztransfer* per Programm zum Helfer beim Lösen von Aufgaben befähigt wurde.

Das Gesagte beinhaltet eine gewisse Abwertung des Computers als Unterhalter, zumindest eine Abwertung der wissenschaftlichen Bedeutung diesbezüglicher Erfolge. Daraus darf jedoch nicht der Schluss gezogen werden, die Bearbeitung von Bildern und Klängen sei grundsätzlich für die Informatik nicht von wissenschaftlichem Interesse. Innerhalb der Informatik hat sich ein selbständiger Wissenschaftszweig unter der Bezeichnung "Bildverarbeitung" etabliert, der sich mit der computerinternen Darstellung und Bearbeitung von Bildern befasst. Allein die Erwähnung der automatischen Auswertung von Lichtbildern macht die praktische Bedeutung dieser Arbeitsrichtung deutlich. Auch wird sofort klar, welch große Rolle bei der Bildverarbeitung das Klassifizieren und Standardisieren spielt, beispielsweise bei der rechnergestützten Überführung von Luftaufnahmen in Landkarten. In diesem Zusammenhang ist auch das Komponieren von Musik durch den Computer zu nennen, ein gerade aus *wissenschaftlicher* Sicht höchst interessanter Prozess.

Die vorangehenden Überlegungen haben gezeigt, dass der Computer allerhand kann, ohne besonders intelligent zu sein. Gerade darum können die gewonnenen Einsichten helfen, sich ein begründetes Urteil hinsichtlich Ursachen, Möglichkeiten und Gefahren der Informationsgesellschaft zu bilden. Unsere Einsichten lassen sich

in folgendem Sastz zusammenfassen. *Ursache dafür, dass die Informationsgesellschaft kräftige Wurzeln schlägt, ist weniger die künstliche Intelligenz, als vielmehr das massenhafte Eindringen von Computern in unser Leben oder noch treffender:* ***die Mitgestaltung unseres Daseins durch ein Heer von Prozessoren***.

Prozessoren mischen sich nachdrücklich in unseren Alltag ein. Am "aufdringlichsten" im sogenannten **Multimediabereich**, d.h. im Bereich des computergestützten Kommunizierens und Informierens unter Verwendung verschiedener Kommunikations*medien*, wie Druck, Bild und Ton. Oft weniger auffällig mischen sich Prozessoren in all unser Tun ein, primär in das weniger intelligente Tun wie das Ausführen häufig sich wiederholender manueller Tätigkeiten (z.B. das Bedienen von Geräten und Maschinen) oder das Anstellen häufig notwendiger Überlegungen und Rechnungen (z.B. in Verbindung mit Haushalt und Einkauf oder mit dem Ausfüllen von Formularen, wie Anträgen oder Steuererklärungen). Vor allem aber mischen sie sich in die Unterhaltung ein und vervielfachen die nutzlos verbrachte Zeit, obwohl sie eigentlich dazu bestimmt sind, Zeit zu sparen.

Dass der Computer dabei eine unerhörte Penetranz entwickelt und keinen Lebensbereich auslässt, ist eine Folge der Ausnutzung menschlicher Grundeigenschaften durch die Marktwirtschaft. Denn Hard- und Software bringt Geld, und das offenbar umso mehr, je "niederer" die Bedürfnisse sind, die sie befriedigen, d.h. je mehr diesen Bedürfnissen Instinkte zugrunde liegen.

17.3 Der Computer als Schachpartner

Wir wollen nun noch eine sehr spezielle Rolle des Computers untersuchen, die Rolle eines Schachpartners. Man kann sie als Kombination der Rolle eines Unterhaltungspartners mit der eines Problemlösungspartners auffassen. Anhand des Schachspielens wollen wir noch einmal der immer wieder gestellten Frage nach den Grenzen der künstlichen Intelligenz nachgehen, diesmal konkret in der Form:

Wieweit lässt sich Schachintelligenz simulieren?[2]

Diese Frage ist im Grunde ebenso sinnlos wie die Frage nach den Grenzen der künstlichen Intelligenz überhaupt. Dennoch wird sie uns weiterhelfen. Zunächst rüsten wir das Problem ab, indem wir, ebenso wie früher bei der Behandlung anderer Partnerrollen des Computers, pragmatisch vorgehen und uns mit dem Erfinden (Nacherfinden) einiger konkreter Möglichkeiten, den Computer schachintelligent zu machen, begnügen. Wir wollen "schlaue" Algorithmen erfinden. Dabei gilt in besonderem Maße das zu Beginn des Kapitels 17.1 Gesagte: Es handelt sich weitgehend um spekulative Überlegungen in dem Sinne, dass wir uns auf keine Autoritäten

2 Es sei auf den sehr aufschlussreicher Artikel über das Computerschach von JÜRG NIEVERGELT [Nievergelt 96] hingewiesen.

und auf keine ausgebauten Theorien stützen können. Solche existieren nur in sehr begrenztem Umfang. Auch die Spieltheorie kann uns kaum helfen. Wir werden uns auch nicht auf existierende Schachprogramme stützen, denn die “wollen gewinnen”, während wir erkennen wollen. Wir wollen erkennen, worin *natürliche* Schachintelligenz besteht, um sie kalkülisieren und implementieren zu können. Warum existierende Schachprogramme uns nur wenig helfen können, wird später begründet.

In Kap.21.3 werden die Mechanismen (Algorithmen) der Schachintelligenz, die wir uns anschicken zu erfinden, auf andere Arten des sprachlichen Modellierens verallgemeinert. In diesem Zusammenhang sei an zwei Vereinbarungen erinnert.

Erste Vereinbarung. Wir hatten jedes *codierende* Modellieren, also auch gedankliches Modellieren, *sprachliches* Modellieren genannt und die Fähigkeit zum sprachlichen Modellieren hatten wir *Intelligenz* genannt. Schachspielen stellt einen fortlaufenden *Problemlösungsprozess* dar. Das Nachdenken über den nächsten Zug (Lösen des aktuellen Problems) ist eine spezielle Form des sprachlichen Modellierens. Die spezielle Fähigkeit, über eine Schachpartie *nachzudenken*, d.h. die Partie sprachlich (gedanklich) zu modellieren, ist eine spezielle Art von Intelligenz. Wir nennen sie **Schachintelligenz**. Sie befähigt den Spieler, das *Schachproblem* so gut er kann zu lösen, d.h. die für ihn besten Züge zu erkennen.

Zweite Vereinbarung. Unter *Simulieren* hatten wir das Modellieren von Eigenschaften und Verhaltensweisen irgendwelcher Objekte oder Systeme auf einem Computer verstanden, also sprachliches Modellieren mittels Computer. In diesem Kapitel wird ein spezielles Simulieren behandelt, das Modellieren (“Nachmachen”) menschlicher Schachintelligenz auf einem Computer, genauer auf einem *Prozessor*computer. Wir benutzen das Wort Simulieren sowohl hinsichtlich des Programmierers, der das *Simulations*programm schreibt, als auch hinsichtlich des Computers, der das Programm ausführt.

Nach diesen Rückerinnerungen und Vereinbarungen wenden wir uns der gestellten Frage zu, wollen sie aber zuspitzen auf die Frage: *Kann der Computer Schachweltmeister werden?* Die Frage ist nicht, ob es möglich ist, dass ein Computer in der Lage ist, eine Schachpartie gegen den Weltmeister zu gewinnen, sondern ob künstliche Schachintelligenz produziert werden kann, die grundsätzlich der natürlichen überlegen ist. Diese Frage ist sinnvoll, denn es wird nicht nach prinzipiellen Grenzen der Intelligenz gefragt, sondern nach dem Unterschied zwischen natürlicher und künstlicher Intelligenz hinsichtlich eines ganz bestimmten Problems.

Den Sieg eines Computers über einen Schachweltmeister könnte ein Journalist mit der Schlagzeile kommentieren: “Maschinelle Intelligenz besiegt menschliche Intelligenz”. Das wäre ein typisches Beispiel für Furoremachen durch Irreführung, selbst dann, wenn die Aussage an sich stimmt. Die Irreführung ist eine doppelte. Zum einen ist die Schlagzeile eine Binsenwahrheit. Denn seit langem gibt es Schachprogramme, die in der Lage sind, mittelmäßige Schachspieler zu besiegen. Zum anderen provoziert sie die unerlaubte Verallgemeinerung, der Computer sei intelligenter als der Mensch, während er lediglich “*schach*intelligenter” ist. Genauer gesagt bedeutet

der Sieg des Computers lediglich, dass er in einem bestimmten Fall schachintelligenter gewesen ist als derjenige Mensch, der vom Computer geschlagen wurde. Betrachtet man die Ergebnisse der Schachturniere “Mensch contra Computer” und setzt den Trend in die Zukunft fort, muss man allerdings zu dem Schluss kommen, dass schließlich wohl der Computer als Sieger aus dem Duell hervorgehen wird

Wodurch zeichnet sich Schachintelligenz aus? Wie spielt man Schach? Welche intelligenten Operationen führt man aus? Diese Fragen wird derjenige stellen und zu beantworten versuchen, der ein Schachprogramm schreiben will. Denn offenbar muss man das Vorgehen des Menschen genau *verstanden* haben, bevor man es simulieren, d.h. kalkülisieren und algorithmieren kann[3].

Die erste Idee, die sicher jedem kommt, der Schachspielen simulieren will, ist das gedankliche *Vorausspielen*, das Suchen nach dem besten Zug durch *Probieren*. So sind auch zwei Pioniere der Informatik und des Computerschach vorgegangen, TURING und SHANNON (siehe [Nievergelt 96]). Es ist ein *Suchproblem* zu lösen. Auf ein ganz anderes und dennoch im Prinzip ähnliches Suchproblem waren wir in Kap.15.8 im Zusammenhang mit dem analytischen Rechnen gestoßen. Beim analytischen Rechnen sucht sich der Computer in einem *Suchgraphen* (im Labyrinth aller möglichen Ableitungswege) seinen Weg. Die Knoten des Graphen waren damals mathematische Ausdrücke; jetzt sind es Stellungen auf dem Spielbrett. Die Kanten waren dort Transformationen gemäß Formeln; jetzt sind es Züge gemäß Regeln. Ein wesentlicher Unterschied besteht darin, dass Schach ein Zweipersonenspiel ist, und dass man die Züge des Gegners nicht vorhersehen kann.

Angesichts dieser Unbestimmtheit könnte man bei der Spieltheorie Hilfe suchen. So verfährt jedoch kein Schachspieler. Der benutzt vielmehr seine Intelligenz, d.h. seine Fähigkeit zum sprachlichen (gedanklichen) Modellieren dazu, die Partie *in Gedanken* weiterzuspielen und dabei verschiedene Züge des Gegners ins Kalkül zu ziehen.

Für ein solches Vorgehen scheint der Computer wegen seiner Schnelligkeit prädestiniert zu sein. Bevor er einen Zug ausführt, probiert er “in Gedanken” alle möglichen Züge durch, alle möglichen Antwortzüge des Gegners, alle möglichen eigenen Antworten auf die Züge des Gegners und so fort. Diese Methode des *vollständigen Durchmusterns* lässt sich zwar ohne besondere Schwierigkeiten implementieren, doch ist die Anzahl der Züge einer vorausgespielten Zugfolge, die sogenannte *Tiefe* des Vorausspielens, auf einige wenige Züge begrenzt, wie folgende grobe Abschätzung zeigt.

Beim Eröffnungszug hat Weiß 20 Möglichkeiten. Danach hat Schwarz 20 Möglichkeiten, sodass nach zwei Zügen, Zug und Gegenzug), 20^2=400 verschiedene Stellungen möglich sind. Nach drei Zügen wären näherungsweise 20^3 und nach n Zügen 20^n verschiedene Stellungen möglich. Tatsächlich liefert diese Extrapolation

3 Der Einsatz neuronaler Netze wird hier nicht betrachtet.

für kleine n (Eröffnungsspiel) zu niedrige Werte, da die Anzahl der Zugmöglichkeiten im Mittel mit n zunimmt, für große n hingegen zu hohe Werte, zum einen infolge des Ausscheidens von Figuren, zum anderen weil jede Stellung auf vielen Wegen erreicht werden kann. Im Endspiel verliert die Näherung ihren Sinn. Die Anzahl der möglichen Züge kann vom Gegner gezielt herabgesetzt werden, im Grenzfall auf Null (vollständiges Blockieren).

Bereits für relativ kleine n ergibt die Abschätzung eine riesige Zahl (man spricht von "kombinatorischer Explosion"), für $n = 10$ beispielsweise $20^{10} \approx 10^{13}$ mögliche Stellungen. Allein um sie aufzuzählen benötigt ein Computer, der pro Sekunde, sagen wir, eine Million Stellungen generieren kann, etwa 4 Monate. Dabei ist die Näherung 20^n nach 10 Zügen eher zu niedrig als zu hoch. Eine in [Reischuk 90] angegebene Abschätzung besagt, dass bei einer Generierungsgeschwindigkeit von einer Milliarde Stellungen pro Sekunde 100 Millionen Jahre nicht ausreichen, um alle legalen Spielstellungen (das sind nach Reischuk über 10^{24}) aufzuzählen.

Die Frage nach dem richtigen Schachzug kann offensichtlich weder der Mensch noch der Computer durch vollständiges Durchmustern, d.h. durch vollständiges Vorausspielen der Restpartie lösen. Das Problem (der Suchgraph) ist *zu komplex*. In Kap.21.2 wird ein Problem, dessen Lösungsaufwand mit einem charakteristischen Parameter, dem sog. Problemumfang (in unserem Fall mit n) exponentiell zunimmt, als Problem mit *exponentieller Komplexität* bezeichnet. Derartige Probleme werden schon für relativ kleinen Problemumfang unlösbar, d.h. in zulässiger Zeit nicht berechenbar. Die sog. *Berechnungskomplexität* ist zu hoch.

4

Wir werden den Begriff der Berechnungskomplexität (im Weiteren kurz Komplexität genannt) schon in diesem Kapitel benutzen, obwohl er erst in Kap.21.2 definiert wird. Dabei wird ein intuitives Verständnis des Begriffs beim Leser vorausgesetzt. Damit können wir unser Ergebnis in folgende Worte fassen: Das vollständige Durchmustern (das Vorausspielen aller Restpartien) scheitert an der Komplexität des Problems. Dabei handelt es sich nicht um *prinzipielle*, sondern um *praktische* Unmöglichkeit. Je schneller die Computer werden, umso weiter (tiefer) können sie in zulässiger Zeit vorausrechnen. Es ist ein Kampf *Rechengeschwindigkeit contra Komplexität* im Gange.

Die Komplexität des Problems wird noch offensichtlicher, wenn man bedenkt, dass jede Stellung nicht nur generiert (gestellt), sondern auch bewertet werden muss. Der Frage nach der Bewertung waren wir bereits beim analytischen Rechnen begegnet. An jeder Wegegabel im Labyrinth (an jeder Verzweigung im Suchgraph) musste beurteilt werden, welcher Weg (die Anwendung welcher Formel) eher zum Ziel führt. Zu diesem Zweck wurde der *Zielabstand* [15.19] eingeführt. Diese Methode ist beim Schach kaum anwendbar (abgesehen vom Endspiel). Wenn die Intelligenz des Spielers (des Menschen oder des Computers) einzig und allein im "stupiden" Suchen besteht, kann eine Stellung nur dadurch bewertet werden, dass die Partie bis zum "bitteren Ende" durchgespielt wird, was jedoch aus Zeitgründen unmöglich ist.

Der Spieler könnte sein Vorausspielen beim Erreichen einer Stellung abbrechen, in der er ohne eigenen Verlust eine Figur des Gegners schlagen kann (Erreichen eines Teilziels). Doch kann auch das schon zu lange dauern. Abgesehen davon ist diese Taktik aus zwei weiteren Gründen fragwürdig. Zum einen kann das Schlagen einer gegnerischen Figur zu einer für den Schlagenden ungünstigen, vielleicht sogar tödlichen Stellung führen. Zum anderen kann das Opfern einer eigenen Figur zu einem Stellungsvorteil führen, der den Verlust überwiegt. Das Vorausspielen muss also fortgesetzt werden.

Der Spieler muss sich entscheiden, warauf er mehr Zeit verwendet, auf das Vorausspielen, also auf das Generieren möglicher Folgestellungen, oder auf die Analyse der generierten Stellungen hinsichtlich ihres taktischen und strategischen Wertes. Das gilt auch für den Computer. Dabei tritt ein auffallender Unterschied zutage. Hinsichtlich der Generierung möglicher Stellungen ist der Computer dem Menschen überlegen, während bei ihrer Bewertung der Mensch dem Computer überlegen ist. Wenn Mensch und Computer ihre Kräfte messen, ist es für den Computer angesichts seiner Schnelligkeit sicher vorteilhafter, das Vorausspielen zu bevorzugen, für den Menschen dagegen das Analysieren und Bewerten. Die Praxis des Computerschach bestätigen dies. Schachprogramme sind i.d.R. auf schnelles Vorausspielen gezüchtet. Aber nicht nur von der Software, sondern auch von der Hardware hängt die Rechengeschwindigkeit des Computers ab. In den vergangenen Jahren hat gerade die Hardware die Spielqualität des Computers erheblich gesteigert. Die heutigen Hochleistungs-Schachcomputer sind Spezialcomputer, die über viele Prozessoren verfügen. In Kap.19.5.4 wird darauf kurz eingegangen.

Weil der Mensch langsamer ist als der Computer, muss er “intelligenter” sein; er muss effektiver suchen und den Wert einer Stellung “auf einen Blick”, ohne langes Analysieren, also *intuitiv* erkennen können. Tatsächlich kann er das; aber *wie macht er das*? Wir erinnern uns an die Bemerkung in Kap.15.8 [15.20], dass ein ahnungsloser Zuschauer den Eindruck haben kann, dass der erfahrene Schachspieler, den er beobachtet, *intuitiv* die richtigen Züge macht, weil er dermaßen schnell zieht, dass zum Nachdenken kaum Zeit bleibt. Wieder die Frage: *Wie macht er das*? Wir wollen versuchen, die Antwort durch Introspektion zu finden. Wir werden dem Denken und Trainieren des Schachspielers nachgehen und uns überlegen, wieweit sich beides simulieren lässt.

Eine sehr einfache Antwort auf die Frage nach der Wurzel der Intuition des Spielers wäre, dass er die Partie auswendig kennt. Für einen erfahrenen Spieler kann das zutreffen, zumindest hinsichtlich des Eröffnungsspiels, das er unzählige Male und in allen Varianten durchexerziert hat. Aus dem gleichen Grund kann er auch in der Lage sein, das Endspiel “automatisch” zu absolvieren. Wenn in solchen Fällen ein Spieler *intuitiv* zu spielen scheint, so verbirgt sich hinter der Intuition weiter nichts als *Erfahrung* und zwar *bewusste* Erfahrung. In diesem Fall sprechen wir von **scheinbarer Intuition**. 5

Auch im Mittelspiel kann Erfahrung helfen, dort allerdings mehr punktuell, hinsichtlich bestimmter Stellungen, die der Spieler sich eingeprägt hat mitsamt dem jeweils günstigsten folgenden Zug. Tatsächlich ist das Einprägen von Stellungen und sogar ganzer Partien eine Methode, die eigene Spielqualität zu erhöhen. Je mehr Partien ein Spieler im Kopf hat, umso öfter wird er eine aktuelle Stellung *wiedererkennen* und *wissen*, welches der nächste "richtige" Zug ist, d.h. welcher Zug in der aktuellen Situation schon einmal mit Erfolg ausgeführt worden ist. Ein solches Vorgehen, das sich an konkreten Stellungen, an konkreten "*Fällen*" orientiert, wird fallorientiert oder **fallbasiert** (englisch: case based) genannt.

6 Ein fallbasierter und infolgedessen schneller Schachzug wird auf denjenigen, der die Erfahrung des Spielers nicht kennt, den Eindruck der *Intuition* machen. Dieser Eindruck wird verstärkt, wenn der Spieler selber seinen Zug nicht genau erklären kann, weil die zugrundeliegende Erfahrung nicht in sein Bewusstsein getreten ist, was durchaus verständlich wäre. Wenn ein Spieler wiederholt mit ein und derselben Stellung (oder einer ähnlichen) konfrontiert wird, liegt die Annahme nahe, dass sich - in entfernter Analogie zu PAWLOVS Hundeexperiment - so etwas wie ein bedingter Reflex ausbildet in dem Sinne, dass der Entscheidungsprozess nicht mehr ins Bewusstsein tritt, dass er "*interiorisiert*" wird. Es stellen sich folgende Fragen:

- Ist Schacherfahrung auf den Computer übertragbar?
- Ist das Sammeln von Erfahrung simulierbar?
- Ist das Anwenden von Erfahrung simulierbar?

Wir werden die Fragen ausführlich beantworten.

Ein Anfänger wird nicht dadurch Schachspielen lernen, dass er Großmeister kopiert, sondern in erster Linie dadurch, dass er übt, also spielt und am eigenen Spiel *lernt*. Er sammelt Erfahrung. Er merkt sich typische Fehler und Erfolge; er merkt sich typische *Fälle*. Er prägt sich nicht fremde sondern eigene Partien ein. Er lernt Fälle, und er lernt aus Fällen. Letzteres bedeutet, dass er seine Erfahrungen aus vielen Fällen zu **Erfahrungsregeln** verdichtet, indem er aus einer Reihe von Fällen, die in bestimmter Hinsicht einander *ähnlich* sind, eine Erfahrungsregel extrahiert, z.B. die Regel, dass man einen Springer nicht auf ein Randfeld stellen soll. Diese Regel hat einen so durchsichtigen Grund, dass man sie auch ohne Erfahrung unmittelbar aus den *Spiel*regeln ableiten kann. Auf den Randfeldern hat ein Springer nämlich höchstens halbsoviele Zugmöglichkeiten, wie auf den meisten anderen Feldern (vorausgesetzt, die Zielfelder sind nicht von eigenen Figuren besetzt).

Die *Ähnlichkeit* aller Stellungen, für welche die eben genannte Erfahrungsregel gilt, besteht darin, dass sie alle durch das *Merkmal* "eigener Springer auf einem Randfeld" charakterisiert sind. Wenn man zwei Stellungen (allgemein zwei Objekte) einander *ähnlich* nennt, wird damit zum Ausdruck gebracht, dass sie in einem oder mehreren Merkmalen übereinstimmen. In je mehr Merkmalen sie übereinstimmen, umso ähnlicher sind sie sich. Wir kommen darauf später genauer zurück.

Es gibt viele derartige Erfahrungsregeln für einander ähnliche Stellungen. Der Anfänger muss sie nicht alle selber finden. Sie sind in Schachlehrbüchern zusam-

mengestellt. *Wenn ein Spieler sich aufgrund einer Erfahrungsregel für einen Zug entscheidet, sprechen wir von* ***regelbasiertem Entscheiden*** *im Gegensatz zum* ***fallbasierten Entscheiden***. Eine Regel ist in vielen Spielsituationen anwendbar, sie gilt für viele *Fälle*.

Wir wiederholen das Gesagte noch einmal mit anderen Worten. Nachdem sich der Anfänger (der *Lernende*) die *Spielregeln* angeeignet hat, stehen ihm folgende Wege offen, seine Spielqualität zu erhöhen (das *Lernen* fortzusetzen): Übernahme fremder Erfahrung (*Reproduktion von Wissen*) und Sammeln eigener Erfahrung (*Produktion von Wissen*). Das Wissen betrifft sowohl *Fälle* (ganz bestimmte Stellungen) als auch *Erfahrungsregeln*. Die Qualifizierungswege (die Lernmethoden) setzen *reproduktive* bzw. *produktive Intelligenz* voraus [3.1].

Es sind also vier Arten des Wissenserwerbs, vier Qualifikationswege zu unterscheiden, *die* ***Produktion*** *von* ***Fall-*** *und* ***Regelwissen*** *und die* ***Reproduktion*** *(Übernahme) von* ***Fall-*** *und* ***Regelwissen****. Die Interiorisierung von Wissen führt zum Erscheinungsbild der* ***Intuition***. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter *Regelwissen* nicht *Spiel*regelwissen, sondern *Erfahrungs*regelwissen zu verstehen.

Es erhebt sich die Frage, welche Arten des Wissenserwerbs dem Computer offen stehen, m.a.W. welche diesbezüglichen Fähigkeiten der natürlichen Intelligenz simulierbar sind. Da Implementieren Algorithmieren und Algorithmieren Kalkülisieren voraussetzt, lässt sich die Frage ganz allgemein, aber ziemlich nichtssagend folgendermaßen beantworten: *Die vier Qualifikationswege lassen sich - wie jede Fähigkeit der natürlichen Intelligenz - soweit simulieren, wie man sie kalkülisieren und in einen Algorithmus überführen kann.* Ob die Anwendung simulierter Fähigkeiten zum Erscheinungsbild der Intuition führt, hängt in erster Linie von der Rechenzeit ab, die der Computer für eine Entscheidung benötigt. Es bleibt allerdings dahingestellt, ob das Problem der Simulierbarkeit der Intuition damit aus der Welt geschafft ist. Doch lässt es sich, nachdem wir die vier Qualifizierungswege herausgearbeitet haben, konkretisieren. Unter diesem Gesichtspunkt wollen wir die vier Wege der Reihe nach untersuchen.

1. Übernommenes Fallwissen. Um fremdes Wissen (fremde Erfahrung) anwenden zu können, muss es zunächst abgespeichert werden. Wir gehen davon aus, dass Fallwissen bestimmten Spielstellungen bestimmte Züge zuordnet, sodass es eine Funktion darstellt. Jeder Fall (jede Spielstellung) stellt einen Argumentwert und der dazugehörige Zug den entsprechenden Funktionswert dar. Das Fallwissen stellt also eine *Funktionstafel* dar und kann *strukturell* (z.B. als Kombinationsschaltung) oder *adressiert* (z.B. in einem peripheren Speicher) abgespeichert werden. Fallwissen kann auf positiver wie auf negativer Erfahrung beruhen, d.h. auf erfolgreichen Zügen, die zu empfehlen sind, sowie auf Zügen, die zum Misserfolg geführt haben und darum besser zu unterlassen sind.

Um das abgespeicherte Fallwissen anwenden zu können, muss der Computer über ein Programm verfügen, das ihn befähigt, eine aktuelle Stellung *wiederzuerkennen*,

das heißt zu erkennen, dass sie im Erfahrungswissen (in der Funktionstafel) enthalten ist. Dazu muss er Stellungen miteinander Feld für Feld vergleichen können. Das zu realisieren bereitet keine prinzipiellen Schwierigkeiten, abgesehen vom Zeitaufwand. Je mehr Fälle die Tafel (das Fallwissen) enthält, umso besser spielt der Computer, umso höher ist seine Schachintelligenz, aber umso länger braucht er für jeden Zug. Der Zeitaufwand lässt sich u.a. dadurch herabsetzen, dass nicht die vollständige Stellung (das ganze Schachbrett), sondern nur ein Teil in Betracht gezogen wird. Diese Möglichkeit werden wir weiter unten aufgreifen.

Der Computer würde perfekt spielen, wenn die Funktionstafel sämtliche denkbaren Fälle umfasste, also - gemäß oben genannter Näherung - mindestens $10^{24} \approx 2^{80}$ Fälle (siehe Formel (9.4b)). Für ihre adressierte Abspeicherung wäre eine Adresse von mindestens 80 Bit Länge und für die strukturelle Abspeicherung eine Kombinationsschaltung mit 80 Eingängen erforderlich. Die Schaltung müsste eine 80-stellige boolesche Funktion berechnen. All das ist praktisch nicht zu verwirklichen.

Schnelles fallbasiertes Spielen würde als intuitives Spielen erscheinen. Es scheitert, ebenso wie das reine Durchsuchen, an der Komplexität des Problems. Dabei handelt es sich wiederum nicht um ein prinzipielles, sondern um ein praktisches Scheitern. Auf diese Weise lässt sich der perfekte Schachcomputer also *nicht* realisieren, nicht nur wegen der Adress- bzw. Wortlänge, sondern auch wegen der Notwendigkeit, zuvor für jede der 10^{24} Stellungen den günstigsten Zug zu ermitteln. Fallwissen kann also nur eine sehr kleine Auswahl aller möglichen Stellungen umfassen. Die Auswahl muss der Programmierer treffen, oder er muss den Computer befähigen, selber Erfahrung zu sammeln.

2. Selbstproduziertes Fallwissen. Damit der Computer eigene Erfahrung sammeln kann, muss er über ein Programm verfügen, das ihn befähigt, Züge zu bewerten und entsprechend der Bewertung zusammen mit der Spielstellung als positive oder negative Erfahrung abzuspeichern. Damit stoßen wir wieder auf das bereits besprochene Problem der Bewertung und ihrer Simulierbarkeit. Offensichtlich ist es günstiger, das vom Menschen akkumulierte Fallwissen zu übernehmen und zu nutzen.

3. Übernommenes Regelwissen. Regelwissen unterscheidet sich von Fallwissen dadurch, dass nicht das ganze Spielbrett betrachtet wird, sondern nur ein Ausschnitt, nur ein Teil des Brettes und nur ein Teil der Figuren. Einen solchen Ausschnitt nennen wir **Konfiguration**. Eine Konfiguration kann eine, zwei oder mehrere Figuren enthalten. Eine Konfiguration aus einer Figur ist z.B. “weißer König steht auf a1.” (Wir nehmen uns die Freiheit, von *Kon*figuration aus *einer* Figur zu sprechen, obwohl die Bezeichnung paradox ist.) Zunächst überlegen wir uns, wie der Mensch Erfahrungsregeln produziert, was sie beinhalten und wie sie in Worte gefasst werden können.

8

Zu einer **Erfahrungsregel** gelangt man durch Zusammenfassung von Konfigurationen zu einer *Klasse* und zwar solcher Konfigurationen, die einander so ähnlich sind, dass sie zu ein und derselben Empfehlung führen, einen Zug auszuführen bzw. zu unterlassen. Eine Empfehlung kann auch mehrere gute und schlechte Züge

enthalten, wodurch die Entscheidung nichtdeterministisch wird. Umgangssprachlich wird eine Konfigurationsklasse i.Allg. durch einen Aussagesatz beschrieben, beispielsweise durch den Satz “Springer steht auf Randfeld” oder “König ist auf Grundlinie eingemauert”, d.h. die eigenen Figuren hindern ihn daran, die Grundlinie zu verlassen. Die Empfehlung wird umgangssprachlich natürlicherweise als Imperativsatz artikuliert, z.B. “Verlasse das Randfeld!” oder “Öffne die Mauer!”

Die “Mauerregel” kann durch Bewegen einer von drei (am Rande von zwei) Figuren befolgt werden, vorausgesetzt, es stehen keine weiteren eigene Figuren im Wege. Welcher konkrete Zug auszuführen ist, hängt von dem Feld ab, auf dem der König steht, und von den mauernden Figuren, aber auch von der Gesamtstellung. Diese kann den Spieler sogar veranlassen, die Regel zu negieren.

Wie lassen sich Erfahrungsregeln und ihre Anwendung implementieren? Hinsichtlich der Form der Abspeicherung der Regeln steht einem, wenn man keine bessere Idee hat, die *extensionale Notlösung* offen, d.h. die Aufzählung aller Konfigurationen, die zu der betreffenden Klasse gehören, und aller Züge der betreffenden Empfehlung. Doch wird für umfangreichere Konfigurationen die Anzahl der Elemente einer Klasse sehr schnell so groß, dass deren Abspeicherung auf Schwierigkeiten stößt oder sogar unmöglich wird. Wieder ist es die Komplexität des Problems, an der eine allgemeine Lösung, d.h. die Implementierung eines *universellen* Algorithmus, der in jedem Falle ausreichend schnell terminiert, scheitert (kombinatorische Explosion). Es handelt sich also auch hier um ein praktisches, nicht um ein prinzipielles Scheitern.

Gegen die kombinatorische Explosion kann man sich eventuell dadurch zur Wehr setzen, dass man Variable für Felder und/oder Figuren einführt und eine Konfigurationsklasse durch eine konkrete Konfiguration und ein Prädikat festlegt und zwar durch eine Formel, nach der die Konfigurationen der Klasse berechnet werden können. Auf diese Weise lassen sich z.B. alle Konfigurationen generieren, die sich durch *Translation* (z.B. durch Verschieben des Königs auf der Grundlinie) oder durch *Spiegelung* einer gegebenen Konfiguration an einer senkrechten oder waagerechten Gitterlinie des Schachbretts entstehen. Aber auch diesem Weg ist bei zunehmender Komplexität früher oder später eine Grenze gesetzt.

Das Anwenden einer Regel muss offensichtlich ähnlich beginnen, wie das Anwenden von Fallwissen, nämlich mit dem Vergleich der aktuellen Stellung mit den Bedingungen der Regeln, diesmal mit dem Ziel, herauszufinden, ob eine Regel anwendbar ist, d.h. ob die aktuelle Stellung eine Konfiguration enthält, die eine konkrete Realisierung (ein *Element* in der Sprechweise der Mathematik) der Konfigurationsklasse einer Regel ist. Dieser Satz weckt beim Leser möglicherweise eine Assoziation. Das Suchen nach einer passenden Regel war der entscheidende Schritt im Markovalgorithmus und beim analytischen Rechnen. Die aktuelle Stellung enspricht einer zu transformierenden Zeichenkette und eine Konfiguration einer zu substituierenden Teilkette. Weiter unten kommen wir auf diese Analogie zurück.

Das Suchen nach einer passenden Regel kann als Klassifizieren aufgefasst werden, genauer als *Klassieren*, denn die aktuelle Stellung wird in eine *bekannte* Klasse eingeordnet und zwar in die Klasse derjenigen Stellungen, die der Bedingung der Regel entsprechen, d.h. die eine Konfiguration enthalten, die in die Konfigurationsklasse der Bedingung der Regel fällt. In diesem Sinne kann man die Konfigurationsklasse als *Merkmal* der ihr entsprechenden *Stellungsklasse* auffassen.

Das Klassi*fizieren* (das "Machen" der Konfigurations- und Stellungsklassen) ist Aufgabe desjenigen, der die Regel erstellt. Er legt die Konfigurationsklassen fest. Die entsprechenden Stellungsklassen ergeben sich durch Vervollständigungen der Konfigurationen zu allen möglichen Stellungen. Insofern beruht das Erstellen einer Regel auf *Abstraktion*. Es werden nur Ausschnitte von Stellungen betrachtet, vom Rest der jeweiligen Stellung wird abstrahiert.

Hier bietet es sich an, noch einmal auf den Begriff der Ähnlichkeit zurückzukommen. Zwei Schachstellungen sind einander umso ähnlicher, in je mehr Merkmalen sie übereinstimmen. Da die Konfigurationsklassen und die Empfehlungen aufgrund taktischer Überlegungen festgelegt werden und ihre Befolgung gewissermaßen die Taktik des Spielers bestimmt, ist es gerechtfertigt, von *taktischer Ähnlichkeit* zu sprechen.

Hat der Computer eine anwendbare Regel gefunden, muss er entscheiden, ob er sie anwendet oder nicht und gegebenenfalls, welchen der angebotenen Züge er ausführt. Dazu muss er die möglichen resultierenden Stellungen bewerten, was, wie wir wissen, nur sehr bedingt möglich ist. Wiederum ist es die Komplexität, an der eine allgemeine, praktikable Lösung scheitert. Die Güte machbarer Regelanwendungen hängt von der Rechengeschwindigkeit des Computers ab.

Die Regelanwendung kann dadurch beschleunigt werden, dass die Regeln nicht adressiert, sondern strukturell gespeichert werden. Eine Regel hat die uns von der Implikation und von den Entscheidungstabellen her bekannte Form eines Wenn-dann-Satzes: "Wenn die und die Bedingungen gegeben sind, dann führe die und die Aktion aus". Regelwissen kann also als *Entscheidungstabelle* formuliert und beispielsweise als ROM-Schaltung realisiert werden (vgl.Kap.12.3.4 und Formel (12.5)). Wenn die Empfehlung der Erfahrungsregel (die Aktion der Entscheidungsregel) mehrere Züge enthält, sind sie alle dem entsprechenden Leiter der ROM-Schaltung einzuprägen, doch nur eine ist jeweils auszuwählen, eventuell nach Bewertungsmerkmalen.

9 Sowohl den Empfehlungen als auch den Konfigurationsklassen können *Bewertungsmerkmale* zugeordnet werden, z.B. das Merkmal "vorteilhaft" oder "gefährlich". Der eingemauerte König war ein Beispiel für eine gefährliche Konfiguration. Gefährliche Konfigurationen können zu einer Klasse zusammengefasst werden. Jede Stellung, die Element dieser Klasse ist, wäre als gefährlich zu bewerten. Nach diesem Rezept können *Bewertungsalgorithmen* entworfen werden.

Wir wollen nun auf den oben angedeuteten gedanklichen Brückenschlag zum Markovalgorithmus und zum analytischen Rechnen eingehen. Die hervorstechende

Gemeinsamkeit, das Suchen in einem Suchgraph, war bereits erwähnt worden. Jetzt kommt es uns in erster Linie auf die Verschiedenheiten der drei Methoden an.

Wenn eine Regel/Formel auf eine Zeichenkette (einen analytischen Ausdruck, eine Spielstellung) anwendbar ist, kann eine Transformation der Zeichenkette (des Ausdrucks, der Stellung) vorgenommen werden, die in der Regel/Formel angegeben ist. Dabei darf man im Falle des analytischen Rechnens aus allen anwendbaren Formeln eine beliebige auswählen. Diese Freiheit lässt einem der Markovalgorithmus nicht. Dagegen geht die Freiheit beim Schachspielen noch weiter als beim analytischen Rechnen. Man ist nicht an die Regeln gebunden, sondern hat bei der Wahl seines Zuges volle Entscheidungsfreiheit und darf seiner *Phantasie* völlig freien Lauf und seine eigene *Intuition* "zum Zuge" kommen lassen.

Der Schachspieler ist also nur an die Spielregeln gebunden, nicht an die Erfahrungsregeln. Blinde Befolgung der Erfahrungsregeln könnte sogar gefährlich werden, denn die Vernachlässigung aller nicht betrachteten Figuren birgt die Gefahr in sich, dass die Anwendung einer Regel zu einer ungünstigen Stellung führt. Der erfahrene Spieler unterliegt dieser Gefahr kaum, denn er hat das gesamte Brett im Blick und erkennt, ob es zweckmäßig ist, eine Regel anzuwenden oder nicht.

Die Annahme liegt nahe, dass die Überlegenheit des Menschen über den Schachcomputer und allgemein der natürlichen über die künstliche Intelligenz in der Entscheidungsfreiheit, also letzten Endes in der *Willensfreiheit* des Menschen liegt, und dass die Überlegenheit eine prinzipielle ist, weil der Computer keine Willensfreiheit besitzt. Er kann nur Programme ausführen.

Die "Unfreiheit" des Computers kann durch Einbau eines Zufallszahlengenerators aufgehoben werden. Man lässt den Computer den nächsten Zug "würfeln". Doch werden seine Gewinnchancen dadurch eher verringert als erhöht. Der Computer wird dadurch nicht intelligenter, denn die gewonnene Freiheit ist nichts anderes als *Unabhängigkeit*, sie ist keine *Willensfreiheit* , denn sie kann nichts wollen, sie kann sich keine Ziele stellen. Der Computer gewinnt keine *intuitive* Intelligenz, denn ***Intuition** ist immer **zielgerichtet***.

Menschliche Intuition hat aber nicht nur mit der *Existenz* eines Zieles zu tun, sondern auch damit, dass der Mensch sich seine Ziele frei wählen kann. Er kann auch verlieren wollen. Er kann "wollen, was er will", alles, was ihm seine *Phantasie* eingibt. Damit öffnet sich neben der interiorisierten Erfahrung eine weitere Quelle der Intuition, die *Phantasie*, und das Wort erlangt diejenige Bedeutung, in der es umgangssprachlich vorwiegend benutzt wird, beispielsweise hinsichtlich des künstlerischen Schaffens, aber auch hinsichtlich des Schachspielens. Schach ist eine *Kunst*. Damit geht unsere Frage in eine andere über: *Lässt sich Phantasie simulieren?* Wir lassen diese Frage im Raume stehen, fügen aber noch eine Bemerkung an. Die Begründung der Überlegenheit des Menschen über den Computer durch die Willensfreiheit wird gegenstandslos, wenn man in der Willensfreiheit keinen objektiven Tatbestand, sondern eine subjektive Überzeugung sieht.

4. Selbstproduziertes Regelwissen. Im Vergleich zum übernommenen Regelwissen spielt selbstproduziertes Regelwissen - ebenso wie selbstproduziertes Fallwissen - für das Computerschach eine untergeordnete Rolle. Bis ein bestimmter Computer (ein Computer mit installiertem Schachprogramm) das im Laufe von Jahrhunderten angesammelte Regelwissen selber produziert haben könnte, ist er lange "gestorben", d.h. sein Programm ist zum alten Eisen geworfen worden. Dennoch wollen wir uns der Vollständigkeit halber überlegen, wie sich dieser Weg simulieren ließe und welche Schwierigkeiten dabei auftreten könnten, denn für die KI ganz allgemein spielt selbstproduziertes Regelwissen durchaus eine Rolle.

Wir fragen: Wie kann aus Fallwissen Regelwissen abgeleitet werden? Das Problem ist zwar schwieriger als die zuvor besprochenen Qualifikationswege, doch spielt die Zeit, die benötigt wird, um aus Fallwissen Regelwissen abzuleiten, eine zweitrangige Rolle, da neue Regeln nicht online (während des Spiels) abgeleitet werden müssen. Der Computer kann sich damit beschäftigen, wenn er "Zeit hat". Das Gleiche gilt für das Aneignen fremden Wissens.

10 Wir wollen einen Algorithmus entwerfen, der aus einer Menge von Fällen eine Regel extrahiert oder "abstrahiert". Wir werden sehen, dass das Wort *abstrahieren* hier durchaus am Platze ist. Beim Entwurf gehen wir davon aus, dass eine Regel einer Reihe von Stellungen ein und dieselbe Empfehlung zuordnet. Demzufolge beginnt der Algorithmus zweckmäßigerweise mit der Wahl einer Empfehlung aus dem Fallwissen. Sodann muss er diejenigen Stellungen, die zu dieser Empfehlung führen, auflisten und aus ihnen diejenigen herausfinden, die eine bestimmte Konfiguration enthalten. Wenn ihm das gelingt, hat er eine Regel gefunden. Aber nach welcher Konfiguration soll er suchen?

Bei der Beantwortung dieser Frage kann der Programmierer dem Computer dadurch helfen, dass er typische Bestandteile von Konfigurationen vorgibt, z.B. "Eigener König steht in einer Ecke" oder "Eigener König und gegnerischer Turm stehen auf eigener Grundlinie" oder "Eigener König ist eingemauert". Es sind solche Teilkonfigurationen auszuwählen, die zwar noch zu arm sind, um eine Empfehlung angeben zu können, die aber für den erfahrenen Spieler ein Achtungszeichen darstellen.

Ausgerüstet mit einer Liste derartiger *beachtenswerter* (gefahren- oder erfolgsträchtiger) Teilkonfigurationen kann der Computer nun das Fallwissen nach diesen durchsuchen, zunächst ohne die Empfehlungen zu berücksichtigen. Findet er für eine Teilkonfiguration mehrere Fälle mit ein und derselben Empfehlung, kann er sie zu einer Regel zusammenfassen. Findet er viele Fälle, aber mit unterschiedlichen Empfehlungen, kann er die Teilkonfiguration um ein Feld oder eine Figur vergrößern und untersuchen, ob in der zuvor gefundenen Untermenge des Fallwissens eine kleinere Untermenge von Fällen enthalten ist, welche die vergrößerte Teilkonfiguration enthalten und außerdem alle zu ein und derselben Empfehlung führen. Ist die Suche erfolgreich, kann eine Regel formuliert werden.

Nebenbei sei darauf hingewiesen, dass die Hinzunahme weiterer Felder oder Figuren als Hinzunahme weiterer Merkmale aufgefasst werden kann, dass es sich also um *Präzisieren* im Sinne von Bild.5.4 handelt, d.h. um eine begriffsbildende Operation.

Man könnte nun die Idee des Vorausspielens mit der Idee der beachtenswerten Konfigurationen kombinieren, indem man das Vorausspielen auf solche Züge beschränkt, die zu einer erfolgsträchtigen Konfiguration führen bzw., falls die eigene Stellung eine gefahrenträchtige Konfiguration enthält, diese entschärfen. Damit würde man dem Vorgehen des Menschen sicher näher kommen. Man könnte sich viele andere Suchstrategien einfallen lassen. Doch wollen wir es genug sein lassen und an dieser Stelle das Erfinden von Schachalgorithmen abbrechen.

Unsere algorithmischen Erfindungen können als Bestätigung folgender oft geäußerten und intuitiv plausiblen Behauptung aufgefasst werden: *Was man verstanden hat, kann man simulieren*. Es bedarf allerdings der Präzisierung, was die Worte "Etwas verstehen" genau bedeuten. Bezüglich der Computerschachintelligenz und der künstlichen Intelligenz überhaupt bedeuten sie soviel wie *"Zurückführung einer intelligenten Leistung auf einen Mechanismus"*. Dabei bezeichnet das Wort Mechanismus eine *endliche Kette diskreter* (genauer *kausaldiskreter) Schritte, die von einem Computer ausgeführt werden können*. Mit diesen Präzisierungen ist die ursprüngliche Behauptung nicht nur plausibel, sondern zu einer Selbstverständlichkeit geworden.

Es würde naheliegen, die Behauptung durch ein "nur" zu verschärfen: *Nur was man verstanden hat, kann man simulieren*. Dieser Satz folgt aus keiner unserer Überlegungen, und tatsächlich trifft er in dieser allgemeinen Form nicht zu. Das folgt bereits aus dem Wenigen, was über neuronale Netze gesagt wurde. Wir kommen darauf noch einmal in Kap.21 zurück.

Unser Erfinden von Schachalgorithmen veranlasst den einen oder anderen Leser vielleicht, selber erfinderisch zu sein. Sicherlich wird er nachfühlen können, welchen Spaß es machen muss, Schachprogramme zu entwickeln, und wie *unheimlich* (wörtlich gemeint) spannend es sein muss, an einem Schachprogramm zu basteln, das versierte Schachspieler besiegt, vielleicht sogar eines Tages Schachweltmeister wird. Ein Sieg eines Computers über einen Meister des Schachspiels ist immer ein Sieg folgsamer Pedanterie und exakter Befehlsausführung über die freie Phantasie. Darin liegt nichts Ungewöhnliches. Die Evolution scheint sogar solche Siege zu lieben, speziell die kulturelle Evolution.

Aus der langen Geschichte des Computerschach zieht Nievergelt in dem oben erwähnten Artikel [Nievergelt 96] folgende Lehre: *"Aber eine Weisheit möge man sich merken. Hunderte von äußerst kompetenten, hochmotivierten Hackern und Forschern haben ein halbes Jahrhundert gebraucht, um künstliche Schachexperten zu bauen, die an menschliche Spitzenleistung herankommen. Dabei ist der Wissensbereich "Schach" eher kleiner, homogener als derjenige vieler "Expertensysteme" (oder "Novizensysteme"?) für kommerzielle Anwendungen. Man darf also nicht*

erwarten, dass ein kleines Team von Programmierern in kurzer Zeit ein nützliches Expertensystem hervorzaubern kann."

Diese Gegenüberstellung von Aufwand und erreichter sehr partieller künstlicher Intelligenz eines Schachcomputers ist ein Schlag gegen alle Euphorie hinsichtlich Expertensystemen und hinsichtlich der KI überhaupt. Nievergelts Schlussfolgerung zeigt, wie weit wir von einer künstlichen Intelligenz entfernt sind, die es mit der menschlichen Intelligenz in voller Breite aufnehmen kann.

Abschließend soll noch einmal der entscheidende Punkt herausgestellt werden, in dem der Mensch dem Schachcomputer überlegen ist, ohne dabei auf so schwierige Begriffe wie Phantasie und Willensfreiheit Bezug zu nehmen. Der Mensch besitzt die Fähigkeit, beim Denken sehr viele Fakten gleichzeitig im Bewusstsein zu halten und zu berücksichtigen. Er denkt *global*, sein Denken ist *komplex*; es erfasst die Dinge im Überblick und trifft Entscheidungen gewissermaßen von einem höheren Gesichtspunkt aus. *Der Mensch denkt in Komplexen.* Vielleicht gewinnt dieser Satz an assoziativer Kraft, wenn das Wort "Komplex" durch das vielschichtige deutsche Wort "Gestalt" ersetzt wird: *Der Mensch denkt* ***in Gestalten****, er denkt gestalthaft*. Er *gestaltet* eine Schachpartie, er *gestaltet* eine Konfiguration und erkennt eine Konfiguration *gestalthaft*, als Gestalt. Im Gegensatz dazu kann der Computer eine Schachstellung nur feldweise "betrachten" und nur computerwortweise bearbeiten, denn er [12] denkt algorithmisch. Verkürzt aber prägnant formulieren wir: *Der Computer denkt in* ***Computerwortfolgen****, der Mensch kann* ***anschaulich*** *und* ***in globalen Zusammenhängen*** *denken.*

Aus dieser Feststellung könnte der Schluss gezogen werden, dass die künstliche Intelligenz für alle Zeiten weit unter dem Niveau der natürlichen Intelligenz bleiben wird. Tatsächlich gibt es für eine solche Schlussfolgerung keinen zwingenden Grund. Zum einen haben wir gesehen, zu welchen erstaunlichen Leistungen der Prozessorcomputer durch die ständige Weiterentwicklung seiner Hard- und Software bereits [13] befähigt worden ist; und diese Entwicklung ist in keiner Weise abgeschlossen. Zum anderen wissen wir aus Kap.9.2.2, dass der genannte Unterschied zwischen dem Denken des Menschen und dem "Denken" des Computers zwar hinsichtlich des Prozessorcomputers, nicht aber hinsichtlich des Neurocomputers besteht. Insbesondere die Lernfähigkeit des Neurocomputers [9.11] gibt Anlass zu einer optimistischeren Sicht auf die Zukunft der KI.

18 Evolution der Programmiersprachen

Zusammenfassung

Sprachen, in denen sich Menschen sowohl mündlich als auch schriftlich zum Zwecke des Informationsaustausches artikulieren können, heißen *Humansprachen*. Humansprachen sind Laut- und meistens gleichzeitig Schriftsprachen. Sie sind bidirektional, d.h. zwischen zwei Kommunikationspartnern in beiden Richtungen verwendbar. Programmiersprachen sind unidirektionale Schriftsprachen. Humansprachen wie Programmiersprachen sind *lineare* (eindimensionale) Sprachen, d.h. als komposite Zeichenrealeme werden ausschließlich Zeichenketten verwendet. Die Linearität der Humansprachen ist die Ursache für eine Diskrepanz zwischen Denken und Sprechen, denn der Mensch denkt vorwiegend bildhaft und netzorientiert. Das entspricht der geometrischen Struktur und kausalen "Vernetzung" der Welt. *Der Mensch denkt netzorientiert und spricht satzorientiert*.

Zwischen dem vernetzten Original und dem vom modellierenden Menschen gedachten vernetzten Modell, dessen Träger ein *neuronales Netz* (das Gehirn des modellierenden Menschen) ist, liegt eine linearsprachliche, satzorientierte Schicht. Dieser Sachverhalt liegt auch beim Modellieren mit Hilfe des Computers vor, nur ist der Träger des vernetzten Modells kein neuronales, sondern ein *boolesches Netz*, die zentrale Computerhardware. Die satzorientierte Schicht zwischen Original und Modell ist in diesem Fall die Maschinenebene, wobei die Sätze der linearen Sprache die Befehle der Maschinensprache sind. Die satzorientierte Schicht heißt *satzorientierte Schnittstelle* oder anschaulicher *linearsprachlicher Flaschenhals des sprachlichen Modellierens*. Durch den engen Flaschenhals muss sich die Artikulierung einer netzorientierten (z.B. räumlichen) Vorstellung satzweise "hindurch zwängen", wobei die Schnittstelle den Informationsfluss zwischen Denken und Sprechen in ähnlicher Weise behindern kann wie der von-neumannsche Flaschenhals den Informationsfluss zwischen Hauptspeicher und Prozessor behindert.

Eine starke Triebkraft der Evolution der Programmiersprachen ist das Bestreben der Programmierer, sich von den Einschränkungen der Maschinensprache zu befreien und die semantische Lücke zwischen Gedachtem und Programmiertem zu schließen. Das sichtbarste Ergebnis ist die Herausbildung des *funktionalen* und des *logischen* (relationalen) *Programmierparadigmas*. Funktionale bzw. logische Programmiersprachen erleichtern das "Umcodieren" funktional bzw. relational formulierter Modelle, die von funktional bzw. relational denkenden Modellierern erstellt (erdacht, evtl. aber noch nicht artikuliert) worden sind, in eine Programmiersprache.

Eine zweite Triebkraft der Entwicklung ist der Wunsch nach Erhöhung der Ausdrucksstärke, der semantischen Dichte von Programmiersprachen. Die *internsemantische Dichte* eines programmiersprachlichen Eingabetextes ist das Verhältnis

der Anzahl der auszuführenden Maschinenbefehle zur Länge des Textes (Anzahl der Bits oder Lexeme). Damit lässt sich die Idee der *Begriffsbildung* als Mittel der semantischen Verdichtung auf Programmiersprachen übertragen, wobei es sich um *internsemantische* Verdichtung handelt.

Die semantische Verdichtung von Programmiersprachen und Programmen beruht auf komponierender und klassifizierender Abstraktion hinsichtlich Operationen (*prozedurale Abstraktion*) und Operanden (*Datenabstraktion*). Ein wichtiger Aspekt der prozeduralen Abstraktion ist das "data hiding", das Schützen prozedureigener Daten gegen den Zugriff durch andere Prozeduren. Das Wort "*Kapselung*" bringt dieses Anliegen anschaulich zum Ausdruck. Speziell auf *klassifizierender* Abstraktion beruht die Idee der "*Vererbung*". Sie besteht darin, in einer Klassenhierarchie von Prozeduren auf jeder Hierarchieebene nur dasjenige explizit zu programmieren, was beim Abstieg in der Hierarchie hinzukommt, also die zusätzlichen (als Vorschrift formulierten) "Merkmale", die beim Übergang von der Oberklasse zur Unterklasse die Präzisierung ausmachen. Die Merkmale (Vorschriften) der Oberklasse werden an alle Unterklassen vererbt.

Die Vereinigung von Kapselung und Vererbung haben zur Herausbildung des Sprachelements "Objekt" und des *objektorientierten* Programmierparadigmas geführt. Ähnlich wie ein Denkobjekt (Idem) ein relativ abgeschlossener Bewusstseinsausschnitt ist, so ist ein *Objekt* ein relativ abgeschlossener Programmteil, der mit anderen Objekten in Wechselwirkung treten, z.B. Aufträge (*Direktiven*) ausführen kann, ohne seine Identität zu verlieren. Die gemeinsame Eigenschaft von Objekten und Operatoren ist die Fähigkeit, Netze zu bilden, Netze kooperierender Partner oder Akteure. Betrachtet man ein aus dem Netz herausgelöstes Objekt, einen isolierten Akteur, so führt dieser eine Folge von Direktiven aus, die den "Imperativen" (Befehlen) imperativer Sprachen entsprechen. Das objektorientierte Paradigma vereinigt in sich Vorteile der Datenflussprogrammierung einerseits und der Aktionsfolgeprogrammierung andrerseits.

18.1 Der Flaschenhals des linearsprachlichen Modellierens

Zunächst vergegenwärtigen wir uns noch einmal, was in diesem Buch unter "Sprache" und "sprachlichem Modellieren" verstanden wird. Im Untertitel des Buches ist mit sprachlichem Modellieren jedes *codierende* Modellieren gemeint. *Ein* ***sprachliches Modell*** *eines Originals ist dessen Beschreibung mit Hilfe codierender Zeichen*, wobei "Beschreibung" nicht auf *Schreiben* und "sprachlich" nicht auf *Sprechen* eingeschränkt ist. Sprachliche Modelle können Sätze einer natürlichen oder künstlichen Sprache sein, z.B. Formeln, es können auch zweidimensionale, man sagt auch *ebene* (in der "Zeichenebene" dargestellte) Gebilde sein wie Graphen, chemische Strukturformeln oder Bilder.

Eine Sprache dient der Artikulierung von *Aussagen über die Welt*. Eine "Aussage" sagt etwas über ein oder mehrere Objekte aus, indem sie ihnen Merkmalswerte (Attribute) zuordnet. Im Falle von Bildern kann die Zuordnung quasi uncodiert, d.h. anschaulich erfolgen. Extensional ist eine Sprache als Menge von auditiven oder visuellen *Kompositzeichen* definiert, die aus elementaren Zeichen nach den syntaktischen Regeln der Sprache gebildet werden. Im Rahmen der Sprachlehre und auch umgangssprachlich wird der Sprachbegriff i.Allg. enger gefasst, indem als Kompositzeichen nur Zeichenketten zugelassen werden. Eine Zeichenkette ergibt sich durch *Verkettung* (durch Voranstellen oder Anhängen, d.h. durch Komponieren "entlang einer gedachten *Linie*") von Zeichen oder Zeichenketten. Aus diesem Grunde spricht man von *eindimensionalen* oder **linearen** Sprachen.

Nach dieser Wiederholung führen wir für die folgenden Überlegungen den Begriff der "Humansprache" ein und vereinbaren: *Als* ***Humansprache*** *wird eine Sprache bezeichnet, in der sich Menschen sowohl mündlich als auch schriftlich zum Zwecke des Informationsaustausches artikulieren können, die also als* ***Lautsprache*** *(****auditive*** *Sprache) und als* ***Schriftsprache*** *(****visuelle*** *Sprache) verwendet wird*. Eine Humansprache dient denjenigen, die sie beherrschen, als zweiseitiges oder **bidirektionales** Kommunikationsmittel , d.h. jeder kann Mitteilungen *senden* und *empfangen*, jeder kann sich in der Sprache *artikulieren* und jeder kann sie *interpretieren*.

Demgegenüber ist eine *Programmiersprache* ein einseitiges oder **unidirektionales** Kommunikationsmittel. In ihr artikulieren Menschen sprachliche Operatoren (Operationsvorschriften). Der Computer interpretiert Programme, die ihm in einer Programmiersprache, die er "versteht", zur Ausführung übergeben werden. Er artikuliert sich aber nicht in der Programmiersprache.

Der Leser hat sicherlich richtig verstanden, was hier mit "Interpretieren durch den Computer" gemeint ist, nämlich das Zuordnen *maschineninterner* Semantik zu einer empfangenen Zeichenkette. Ursprünglich hatten wir den Begriff des Interpretierens als "Interpretieren durch den Menschen" definiert, und zwar als Zuordnen von *externer* Semantik (Zuordnen eines *Idems*) zu einer empfangenen Zeichenkette (einem *Zeichenrealem*) und das "Artikulieren durch den Menschen" als Zuordnen eines Zeichenrealems zu einem Idem (vgl. Bild 2.1).

Dementsprechend ist konsequenterweise unter "*Artikulieren* durch den Computer" das Zuordnen einer Ausgabezeichenkette zur *maschineninternen* Semantik, d.h. zu einem codierenden internen Zustand zu verstehen, was natürlich möglich ist. Insofern könnte man sagen, dass der Computer sich artikulieren kann. Doch wäre das irreführend. Der Computer artikuliert sich nicht in dem Sinne, dass er "seinen", mit *externer* Semantik behafteten Idemen Zeichenrealeme zuordnet. Sein "Artikulieren" ist lediglich ein Überführen aus seiner internen in eine externe *Darstellung*. Was der Computer ausgibt, legt i.d.R. der Programmierer fest. Wenn dieser einen umgangssprachlichen Satz (mit externer Semantik) einprogrammiert, kann die Illusion entstehen, der Computer "denke in externer Semantik" und artikuliere *sich* umgangssprachlich (vgl. das Fallexperiment in Kap.15.5 [15.7]).

Trotz dieses gravierenden Unterschiedes zwischen Human- und Programmiersprachen besitzen sie eine fundamentale Eigenschaft gemeinsam. *Humansprachen wie Programmiersprachen sind* ***lineare Sprachen***. Auf den ersten Blick scheint das eher eine Selbstverständlichkeit als eine "fundamentale" Eigenschaft zu sein. In Kap.18.3 wird sich zeigen, dass die Linearität den wohl wichtigsten Motor für die Evolution sowohl der Humansprachen als auch der Programmiersprachen darstellt. Zunächst fragen wir nach dem *Grund* der Linearität.

Aus der Sicht des Menschen ist die Linearität von *Programmiersprachen* notwendig, damit er bei der Kommunikation mit dem Computer seine humansprachlichen Artikulierungsgewohnheiten beibehalten kann. Diese Forderung lag letzten Endes unserer sehr speziellen Zielstellung zugrunde, ein Gerät zur Verarbeitung von *Bitketten* zu konstruieren. Warum aber sind *Humansprachen* linear? Der Grund ist offenbar in der Funktionsweise des Hörapparates zu suchen. Dieser registriert den "*eindimensionalen*" Verlauf, des Druckes, den die Luft auf das Trommelfell ausübt. Die "einzige Dimension" ist die Zeit. Auf dieser Grundlage kann sich nur eine *eindimensionale* Sprache entwickeln. Demgegenüber registriert der Sehapparat eine zeitliche Folge ebener (genauer sphärischer, jedenfalls zweidimensionaler) "Bilder", die von der Linse auf die Netzhaut projiziert werden.

Das Fehlen geometrischer Dimensionen bei der auditiven Perzeption der Welt (abgesehen von einem sehr groben Stereohören) ist nicht etwa ein Mangel des Hörapparates, sondern hat prinzipielle physikalische Ursachen, die mit der großen Länge von Schallwellen (im Vergleich zur Länge von Lichtwellen) zusammenhängen.

Mit dem Menschen, seinen Sinnesorganen, seinem Bewegungsapparat und seinem Gehirn hatte die Evolution die Möglichkeit der Entwicklung von Sprachen geschaffen. Infolge der physikalischen und physiologischen Voraussetzungen entwickelten sich zunächst auditive (gesprochene) Sprachen und einfache visuelle, in erster Linie gestische Zeichensprachen. Aus den *eindimensionalen* auditiven Sprachen gingen später (unter Einbeziehung des visuellen und motorischen Apparates) *eindimensionale* Schriftsprachen hervor, indem Folgen gedachter Objekte (Idemfolgen) und später Phonemfolgen in Zeichenfolgen "übersetzt" wurden, wobei die einzige Dimension nicht mehr die *zeitliche*, sondern *eine räumliche* Dimension ist. Da die Zeichen selber i.d.R. ebene (zweidimensionale) Gebilde (Schriftzüge auf einer Schreibebene) darstellen, bezeichnet man Schriftsprachen besser als *linear* und nicht als eindimensional.

Hier soll eine Zwischenbemerkung eingeschoben werden, die über den eigentlichen Gegenstand des Buches hinausgeht. Eine Humansprache kann als Lautsprache und als Schriftsprache verwendet werden. Daran scheint nichts Verwunderliches zu sein. Tatsächlich ist dieser Doppelcharakter ein erstaunliches Phänomen. Denn um es zu erklären, muss man annehmen, dass das Gehirn sowohl mit statischer als auch mit dynamischer Codierung umgehen kann [9.1]. Denn *auditive Sprachen verwenden* ***dynamische*** *Codierung, während Schriftsprachen* ***statische*** *Codierung verwenden.*

Die Linearität der Humansprachen ist die Ursache für eine Diskrepanz zwischen Denken und Sprechen, auf die etwas ausführlicher eingegangen werden soll, da sie dem Menschen nur relativ selten zum Bewusstsein kommt, nämlich dann, wenn er sich “nicht ausdrücken kann”.

Die hervorragende Bedeutung der visuellen Perzeption der Welt durch den sehenden Menschen hat dazu geführt, dass die *Bildhaftigkeit* des “Eindrucks”, den die Welt auf die Netzhaut macht, sich auf das Denken übertragen hat. *Der Mensch denkt vorwiegend* ***bildhaft***, d.h. in zweidimensionalen, eventuell auch in dreidimensionalen Vorstellungen.

Eine erste Verarbeitung der Sinneseindrücke durch das Gehirn führt zur Heraustrennung einzelner Objekte, so dass sich die Welt dem Menschen als Menge von Objekten darstellt, zwischen denen räumliche und zeitliche Beziehungen bestehen. Infolge einer weiteren Verarbeitung, in die gewissermaßen die gesamte Menschheitserfahrung eingeht, stellt sich die Welt dem Menschen, seinem analytischen Verstande, als *Netz* von Objekten dar, zwischen denen Ursache-Wirkungs-Beziehungen bestehen. Die Welt ist *kausal vernetzt*. Mit “kausal vernetzt” soll zum Ausdruck gebracht werden, dass die Objekte der Welt gegenseitig aufeinander *wirken*, sodass die Ereignisse in der Welt *zeitliche Ursache-Wirkungsfolgen* bilden. Der Mensch “denkt” die Welt (modelliert die Welt gedanklich) als räumliches und kausales Netz. Das bringen wir mit dem Satz zum Ausdruck: *Der Mensch denkt* ***netzorientiert***. Die [1] kausale Modellierung impliziert die Dimension der Zeit.

Ob die kausale Vernetzung eine Eigenschaft der Welt oder eine Eigenschaft des menschlichen Denkens ist, sei dahingestellt. Jedenfalls sind wir Menschen gewohnt, die Welt als kausales Netz anzusehen. Das Denken in Ursache-Wirkungs-Zusammenhängen liegt nicht nur dem naturwissenschaftlichen, sondern jedem Modell der Welt, jedem “*Weltbild*” zugrunde, auch einem religiösen, mystischen, magischen oder esoterischen.

Das Wort “vernetzt” steht als *bildhafter* Ausdruck für “*miteinander in Beziehung stehend*”. Im vorangehenden Kapitel haben wir das Denken des Menschen *gestalthaft* genannt. In diesem vielsinnigen Wort finden die Wörter “bildhaft” und “netzorientiert” eine geglückte inhaltliche Synthese mit hoher praktischer Relevanz, wie die Bezeichnung “*Gestaltpsychologie*” zeigt.

Während der Mensch netzorientiert denkt, spricht er “linear”. Da die *semantische Einheit* von Humansprachen der Satz ist, kann man prägnant, aber nicht ganz exakt (s.u.) sagen: *Der Mensch denkt* ***netzorientiert*** *und spricht* ***satzorientiert***. Zwischen Denken und Sprechen besteht also ein charakteristischer Unterschied und die *interne* Codierung des Denkens unterscheidet sich wesentlich von der *externen* Codierung des Sprechens.

Die Überführung von dem einem in den anderen Code ist Aufgabe des Sprachzentrums. Dem externen Artikulieren von Vorstellungen muss ein i.Allg. unbewusstes satzorientiertes internes Artikulieren, also ein Denken in Sätzen vorausgegangen sein. Es ist also nicht verwunderlich, dass sich mit der Zeit auch ein *bewusstes Denken*

und *Nachdenken in Sätzen* herausgebildet hat. Der Mensch kann also nicht nur netzorientiert, sondern auch satzorientiert denken. Der Überführung interncodierter (gedachter) in externcodierte (gesprochene oder geschriebene) Sätze entspricht der Pfeil 3 in Bild 2.1, wobei "Zeichenidem" und "Zeichenrealem" zu "Satzidem" und "Satzrealem" zu konkretisieren sind.

Die Ergebnisse psychologischer Experimente sprechen dafür, dass der Träger des gestalthaften, netzorientierten Denkens vorzugsweise in der rechten und der Träger des satzorientierten Denkens vorzugsweise in der linken Gehirnhälfte liegt, wo sich normalerweise das Sprachzentrum befindet. Die Überführung des gestalthaft Gedachten in gesprochene Sprache kann auf Schwierigkeiten stoßen, mit denen das im Unterbewusstsein arbeitende Sprachzentrum nicht fertig wird und die es "an das Bewusstsein delegiert". Man *merkt* dann, dass man "sich nicht ausdrücken kann", dass es schwierig ist, seine Vorstellungen (das gestalthaft Gedachte) in Worte zu fassen, d.h. linearsprachlich zu artikulieren. In solchen Fällen bedient man sich gerne einer Zeichensprache, etwa indem man Stift und Papier nimmt und "zeichnet" oder indem man den Körper, z.B. die Hände "sprechen" lässt. Man denke an die Wendelbewegung, die ein Mensch mit seiner Hand ausführt, um zu verdeutlichen, was eine Wendeltreppe ist.

Dass sich die lineare Umgangssprache eventuell als recht unbeholfen erweist, ist nicht verwunderlich, insbesondere beim Artikulieren komplexer Vorstellungen. Denn das vorgestellte Objekt, z.B. eine Wendeltreppe, wird zwar gedanklich als Ganzes erfasst, muss aber sprachlich als Folge von Sätzen beschrieben werden. Bildlich gesprochen muss beim *Artikulieren* eine komplexe ganzheitliche Vorstellung, ein gestalthaftes ("voluminöses") Objekt zu einem langen dünnen Strang, zu einer Folge von Sätzen verformt werden, die ihrerseits Folgen von Morphemen oder Buchstaben sind.

Dieses Bild vor Augen sprechen wir metaphorisch vom *Nadelöhr* oder noch anschaulicher vom **Flaschenhals des linearsprachlichen Modellierens**. Das Wort "Flaschenhals" soll - ähnlich wie im Falle des von-neumannschen Flaschenhalses - die Vorstellung wecken, dass der Inhalt einer Flasche durch den Hals hindurch *geschleust* werden muss, um "nach außen" treten und verwendet werden zu können, und dass er dabei aus einer voluminösen Form zu einem dünnen Strahl, Strang oder Strom verformt werden muss. Der "lange, dünne Strom" der Rede verlässt den Sprechenden durch den "Hals" (den physischen "Flaschenhals" Kehlkopf und Mund) und tritt durch das Ohr des Hörenden (den physischen Flaschenhals des Empfängers) in den Flaschenbauch (das Gehirn), wo es wieder eine "voluminöse" Form annimmt, d.h. durch Interpratation zu einer ganzheitlichen, gestalthaften Vorstellung (z.B. zur Vorstellung einer sich wendelnden Treppe) wird. Aus einem "Rinnsal" von Worten wird ein "Meer" von Vorstellungen.

2 Neben der *räumlichen* (geometrischen) *Komplexität* des Gedachten gibt es einen zweiten Grund, Gedanken mehrdimensional, speziell zweidimensional zu artikulieren, die *logische Komplexität*. Das sogenannte logische Denken, z.B. Rechnen oder

Schlussfolgern, vollzieht sich vorzugsweise linearsprachlich, auch dann, wenn es *nicht* extern, sondern gedanklich artikuliert wird. Dies ist der Grund dafür, dass logisches Denken zuweilen als schwierig oder unangenehm (weil unanschaulich) empfunden wird.

Hinzu kommt, dass logisches Denken in größeren Zusammenhängen eine große Kapazität des Kurzzeitgedächtnisses erfordert. Denn alle Objekte und alle Beziehungen zwischen ihnen müssen zugriffsbereit im Bewusstsein gehalten werden, und der Zugriff erfolgt nicht über räumliche Vorstellungen, wie beim vorstellenden Denken, sondern über gedachte Namen (interne Codierungen; vgl. Bild 2.1). Wenn dies allzu schwierig wird, benutzt man gerne einen externen Speicher, z.B. Papier und Stift und codiert extern, wobei die Papierebene netzorientiertes Codieren erlaubt. Ein Beispiel hierfür ist der Verwandtschafts*graph* von Bild 16.1. Er unterstützt das Zurechtfinden in der Verzwicktheit oder Vertracktheit (in der logischen Komplexität) der linearsprachlichen Beschreibung des verwandtschaftlichen Beziehungs*netzes*.

Schließlich gibt es einen dritten Grund für die Verwendung einer ebenen Sprache, die *kausale Komplexität*. Auch hier leisten graphische Darstellungen gute Dienste. Ein Beispiel sind die Operatorennetze der USB-Methode. Die Methode dient der "anschaulichen" zweidimensionalen, sprachlichen (codierten) Modellierung von Ursache-Wirkungsbeziehungen, die auch dann, wenn sie hohe Komplexität aufweisen, "überschaubar" bleiben. Die netzorientierte *kausale* Vorstellungs- und Beschreibungsweise war an den unterschiedlichsten Beispielen illustriert worden, in Bild 8.3 am Beispiel eines Fertigungsprozesses, in Kap.13.7 am Beispiel des Straßenverkehrs [13.15] und der Strömung von Flüssigkeiten [13.16], und in Kap.15.7 am Beispiel eines Unternehmens [15.11], das man sich als Operatorenhierarchie, also als übereinander geschichtete Netze vorstellen kann. In Kap.8.2.2 hatten wir eine Methode kennen gelernt, die es gestattet, die kausale Vernetzung der Welt auf sehr abstrakter Ebene mit Hilfe von *Petrinetzen* zu beschreiben.

Bisher war zwischen dem von-neumannschen Flaschenhals und dem Flaschenhals des linearsprachlichen Modellierens lediglich eine rein metaphorische Analogie zu erkennen. Es besteht aber eine ganz konkrete Analogie. Um sie herauszuarbeiten, geben wir zunächst eine genaue Bestimmung des bereits verwendeten Begriffs der *semantischen Einheit*. *Eine Zeichenkette, die als Objekt-Merkmals-Zuordnung, also als Aussage interpretierbar ist, heißt* ***semantische Einheit****. Eine semantische Einheit heißt* ***syntaktisch vollständig****, wenn alle Objekte, die an der Merkmalszuordnung beteiligt sind, explizit genannt werden.* Die semantischen Einheiten von Humansprachen sind Sätze. Die semantischen Einheiten von Maschinensprachen sind Befehle.

Hiergegen könnte eingewendet werden, dass die Definition der semantischen Einheit nur auf Aussagesätze zutrifft. Darauf wäre zu erwidern, dass auch Imperativsätze bzw. Maschinenbefehle und auch Fragesätze Objekt-Merkmals-Zuordnungen, also Aussagen artikulieren. Imperativsätze bzw. Maschinenbefehle *schreiben Zuordnungen vor,* Fragesätze *fragen nach Zuordnungen.* Insofern trifft die Definition auf jeden Satz und jeden Maschinenbefehl zu.

Die Analogie, auf die wir hinauswollen, beruht darauf, dass die semantischen Einheiten sowohl maschinensprachlicher als auch natürlichsprachlicher Ausdrücke syntaktisch vollständig sind. Ein Befehl nennt die beteiligten Objekte, also die Ein- und Ausgabeoperanden, durch Angabe ihrer Bezeichner bzw. Adressen. Ein Satz nennt sie durch entsprechende Satzglieder; es sind i.d.R. Substantive oder Pronomen. Sowohl Humansprachen als auch Maschinensprachen befolgen die Regel, dass semantische Einheiten syntaktisch vollständig sein müssen[1]. Angesichts dieser Gemeinsamkeit vereinbaren wir: *Syntaktisch vollständige semantische Einheiten, also humansprachliche Sätze und Maschinenbefehle werden unter der Bezeichnung* ***Satz im weiten Sinne (i.w.S.)*** *zusammengefasst.*

Die Forderung syntaktischer Vollständigkeit mag überraschen, lässt sich aber plausibel erklären. Eine ihrer Ursachen liegt in beiden Fällen in der Speicherung, im Falle der Humansprachen in der Kapazität des Kurzzeitgedächtnisses und im Falle der Maschinensprachen in der Zentralisierung der Speicherplätze im Hauptspeicher, die hardwaremäßig zu der Verbindung zwischen HK und K in Bild 13.7, d.h. zum von-neumannschen Flaschenhals führt und die softwaremäßig das Programmieren in Aktionsfolgen verlangt. Damit ist die zunächst rein metaphorische Analogie zwischen dem von-neumannschen und dem linearsprachlichen Flaschenhals zu einer konkreten Analogie geworden, die im materiellen Träger ihre Ursache hat.

Auch die Metapher vom “Flaschen*bauch*” lässt sich vom materiellen Träger her deuten, in welchem aus dem eintretenden “Rinnsal” von Sätzen bzw. Befehlen ein “Meer” von Wirkungen wird. Die Wirkungen im “Flaschenbauch” sind die Prozesse, die im Gehirn bzw. in der Zentraleinheit (CPU) ausgelöst werden. Beide “Flaschenbäuche” sind *Netze*, entweder *Neuronennetze* oder *boolesche Netze*.

Hinsichtlich der syntaktischen Vollständigkeit sei daran erinnert, dass auch für Algorithmen im ursprüngliche Sinne, d.h. für *imperative* Algorithmen, die syntaktische Vollständigkeitsforderung gilt. Schließlich sei noch erwähnt, dass in Humansprachen Pronomen die Rolle von Subjekten und Objekten übernehmen können. Das setzt eine eindeutige Zuordnung zwischen Nomen und Pronomen und ein entsprechendes Kurzzeitgedächtnis voraus. Ähnlich verfahren Maschinensprachen. Die Rolle des Kurzzeitgedächtnisses können der Akkumulator oder andere CPU-Register übernehmen. Zeiger auf diese Register können mit Pronomen in humansprachlichen Sätzen verglichen werden.

Unsere ursprüngliche Idee war es, das Komponieren von Operatoren von der Hardware in die Software zu übernehmen, d.h. in die Software hinein fortzusetzen. Der linearsprachliche Flaschenhals vereitelt die unmittelbare Verwirklichung dieser Idee. Das hatten wir bereits in Kap.13.5.3 erkannt und mit dem bedingten Sprungbefehl einen Ausweg gefunden, der die Fortsetzung des Komponierens ermöglicht,

1 Dabei ist von sog. *Ellipsen* abgesehen; das sind Auslassungen von Redeteilen humansprachlicher Sätze, die der Interpretierer “automatisch” ergänzt.

wenn auch nicht in Form von Operatorennetzen, so doch in Form verzweigter Aktionsfolgen, konkreter in Form von Maschinenprogrammen, welche bedingte Sprungbefehle enthalten, oder allgemeiner in Form imperativer Programme mit bedingten Anweisungen. Der Preis für die softwaremäßige Fortsetzung des Komponierens ist die Aufgabe des Netzparadigmas zugunsten des imperativen Paradigmas. Dieser Paradigmenwechsel ist in Kap.13.7 ausführlich besprochen worden.

Hinsichtlich des Modellierens "der vernetzten Welt" steht der Programmierer vor der gleichen Schwierigkeit wie jeder Mensch, der Zusammenhänge der "vernetzten Welt" in Worte fassen will. Zwischen dem vernetzten Original und dem vernetzten Modell, dessen Träger entweder ein boolesches Netz oder ein neuronales Netz sein kann, liegt eine linearsprachliche, und zwar eine *satzorientierte* Schicht. Das Wort "satzorientiert" bedeutet hier, dass die Bausteine dieser Schicht Sätze im weiten Sinne sind. Die Evolution der Programmiersprachen ist von dem - als Selektionsdruck wirkenden - Bestreben der Programmierer geprägt, sich von den Einschränkungen zu befreien, die ihm das satzorientierte Artikulieren auferlegt. Diese Entwicklung wollen wir in großen Zügen nachvollziehen. Dabei können wir auf den Einsichten dieses Kapitels aufbauen, die wir noch einmal zusammenfassen.

Es hat sich folgendes Bild ergeben. Zwischen der *externen* Netzstruktur der Außenwelt und der *internen* Netzstruktur des Computers bzw. des Gehirns liegt eine *linearsprachliche, satzorientierte* Schicht. Sie ist für die humansprachliche Modellierung der Welt ebenso notwendig wie für die Modellierung durch Prozessorrechner (Simulation). Wir nennen sie ***satzorientierte Schnittstelle*** oder anschaulicher ***linearsprachlichen Flaschenhals*** *des sprachlichen (d.h. codierenden) Modellierens.*

18.2 Semantische Verdichtung

In Kap.5.1 hatten wir uns Gedanken über die Evolution natürlicher Sprachen gemacht. Der Versuch liegt nahe, unsere dortigen Einsichten auf Programmiersprachen zu übertragen. Dies ist angesichts der Unterschiede in den Ursachen und in den Mechanismen der Evolution der beiden Sprachklassen sicher nur sehr bedingt möglich. Humansprachen (natürliche Laut- und Schriftsprachen) sind unter dem Evolutionsdruck in Richtung höherer Überlebenschancen entstanden, ohne dass der Mensch sie *bewusst* "konstruiert" hätte. Programmiersprachen sind das Produkt *bewusster*, *zielgerichteter*, *konstruktiver* Tätigkeit der Menschen, wobei auch hier ein Druck in Richtung höherer "Überlebenschancen" eine Rolle gespielt hat und weiterhin spielt, besser gesagt, in Richtung höherer Durchsetzungschancen im wirtschaftlichen oder beruflichen Wettbewerb.

Die Evolution von Sprachen ist ein Beispiel dafür, wie sehr die Evolution durch die Teilnahme der menschlichen Intelligenz am Evolutionsprozess beschleunigt wird. Das betrifft nicht nur die Programmiersprachen, sondern jede Sprache, an deren Entwicklung der Mensch bewusst teilnimmt. Dabei handelt es sich um eine opera-

tionale Zirkularität. Die Intelligenz, also die Fähigkeit zum sprachlichen Modellieren, schmiedet sich ihr eigenes Werkzeug, die Sprache. Dieser Prozess ist im Laufe der Menschheitsgeschichte immer mehr aus dem unbewussten in den bewussten Bereich menschlicher Tätigkeit aufgestiegen.[2]

Der Umstand, dass ein und derselbe physiologische Apparat (das Gehirn) Sprache *benutzt* und Sprache *schafft*, lässt erwarten, dass trotz aller Unterschiede zwischen der Evolution von Humansprachen und Programmiersprachen Ähnlichkeiten bestehen, denn ein Sprachentwickler denkt über die Programmiersprache und deren Syntax, die er definieren will, in “*seiner eigenen*” (durch lebenslange Benutzung “zu eigen” gewordenen, interiorisierten) Humansprache nach.

Es gibt noch andere Gründe, die Ähnlichkeiten erwarten lassen. So wird von universellen Programmiersprachen ebenso wie von Humansprachen verlangt, dass ihr Umfang endlich ist und dass ihre Artikulationen räumlich und zeitlich endlich sind, dass mit ihrer Hilfe aber trotzdem “unendlich viel” ausgedrückt werden kann, dass “unendlich viele” Originale sprachlich modelliert werden können. Schließlich müssen programmiersprachliche Ausdrücke ebenso wie humansprachliche eindeutig und schnell interpretierbar sein.

In Kap.5.1 hatten wir die “Reaktion” der Evolution von Humansprachen auf die genannten Forderungen durch drei Evolutionsprodukte charakterisiert:

- Begriffsbildung,
- Grammatik,
- Idemobjektivierung.

Da programmiersprachliche Ausdrücke an sich (ohne menschlichen Interpretierer) keine externe, sondern nur maschineninterne Semantik besitzen, entfällt die Idemobjektivierung, denn interne Semantik *ist* durch den technischen Träger objektiviert. Die Rolle der Grammatik von Programmiersprachen ist in den Kapiteln 16.4 und 16.5 behandelt worden. Es stellt sich die Frage, ob die Begriffsbildung für Programmiersprachen eine ähnliche Bedeutung besitzt wie für Humansprachen.

Es scheint so, als müsse die Frage verneint werden, denn humansprachliche Begriffsbildung ist an externe Semantik gebunden. Ein wesentlicher “Zweck” der Begriffsbildung ist die Erhöhung der Ausdrucksstärke, der *semantischen Dichte*. Die ***semantische Dichte*** eines Textes war in Kap.5.5 [5.14] (nicht sehr exakt) definiert worden als *Umfang des im Mittel pro Buchstaben (oder auch pro Wort) assoziierbaren Gedächtnisinhaltes* (der assoziierbaren Denkobjekte oder Ideme). Man erinnere sich an die Wörter “Tierreich”, “Erde”, “Gott”, die in Kap.5.5 als Beispiele für Wörter mit hoher semantischer Dichte genannt wurden.

Da die Begriffe “Idem” und “Denkobjekt” nicht exakt definiert sind, gilt dies auch für den Begriff der semantischen Dichte. Das ändert sich, wenn man den Begriff auf Programmiersprachen überträgt. Das ist möglich, wenn unter semantischer Dichte

2 Siehe z.B. [Klix 80].

nicht *extern*semantische, sondern *intern*semantische Dichte verstanden wird. Die ***internsemantische Dichte*** eines programmiersprachlichen Eingabetextes lässt sich definieren als die *mittlere Anzahl der Maschinenbefehle pro Bit (oder auch pro Lexem) des Eingabetextes*, die bei dessen Abarbeitung (Interpretation) ausgeführt werden. Damit lässt sich auch die Idee der Begriffsbildung auf Programmiersprachen übertragen. Sie dient der *internsemantischen Verdichtung*.

Zu diesem Zwecke werden durch Abstraktion neue Begriffe gebildet, die sich nun aber nicht auf die zu modellierende Welt, sondern auf die modellierende Sprache beziehen und diese beschreiben. Die Begriffsbildung betrifft also, ebenso wie die Begriffsbildung der Schulgrammatik, *metasprachliche* Begriffe. In Kap.5.5 war das Vorgehen am Beispiel des Syntaxbaumes eines deutschsprachigen Satzes (siehe Bild 5.2) veranschaulicht worden. In der Backus-Naur-Form der Syntaxregeln einer Sprache treten die metasprachlichen Begriffe als *metasprachliche Variablen* auf, z.B. "*Satz*" in (5.1) oder "*Befehl*" in (5.2). Jetzt wollen wir den Weg verfolgen, den die Entwickler von Programmiersprachen beim Erfinden neuer Sprachkonstrukte und neuer metasprachlicher Begriffe gegangen sind.

In einem humansprachlichen oder programmiersprachlichen Text entsprechen den metasprachlichen Begriffen bzw. Variablen konkrete Wörter oder Konstruktionen (Sprachkonstrukte), z.B. der AcI in (16.2) [16.13] oder die WHILE-Anweisung in Bild 15.3b. Der Weg zum abstrakten metasprachlichen Begriff beginnt mit der *Erfahrung*, dass gewisse komplexe Operationen oder Operanden häufig in ähnlicher Weise auftreten. Wenn derartige Fälle erkannt sind, kann es zweckmäßig sein, ein kompakteres (ausdrucksstärkeres, *semantisch dichteres*) Sprachelement zu definieren, beispielsweise eine Laufanweisung anstelle eines Zyklus, wie die Programme der Bilder 15.2 und 15.3 demonstrieren. Wenn ein auf diese Weise entstandenes Sprachkonstrukt als Sprachelement in eine Programmiersprache aufgenommen wird, erscheint in der Syntax der Sprache eine neue *metasprachliche Variable*.

Es ist aber auch eine semantische Verdichtung durch den Programmierer selbst (den Benutzer einer gegebenen Programmiersprache) möglich. Viele Sprachen erlauben dem Programmierer, sich für den eigenen Gebrauch Konstrukte, die von der Sprache nicht zur Verfügung gestellt werden, selber zu definieren und Bezeichner für sie zu vereinbaren. Das gilt insbesondere für Kompositoperationen (Operationsvorschriften). Wenn der Bezeichner einer solchen selbstdefinierten Operation in einem Programm seinerseits als Name eines *Programms* auftritt, sprechen wir von *Unterprogramm*. Dann kann auf das Resultat der Operation über einen speziellen Bezeichner zugegriffen werden. Wenn der Bezeichner in einem Programm (evtl. auch in einem an sich imperativen Programm) als Name einer *Funktion* auftritt, sprechen wir von *Unterfunktion*. Der Bezeichner stellt dann den Wert der Unterfunktion, also das Resultat der Operation dar. Unterprogramme und Unterfunktionen können über ihre Namen eventuell auch von anderen Anwendern aufgerufen werden.

Die beschriebene Art der semantischen Verdichtung von Operationsvorschriften beruht auf *komponierender Abstraktion* (vgl. Bild 5.5) bezogen auf das Komponieren 3

sprachlicher Operatoren. In Kap.15.7 [15.12] war dafür der Begriff der **prozeduralen Abstraktion** eingeführt worden. Diese Bezeichnung beruht auf folgendem Sachverhalt. Ein Programmierer, der für eine Operation, beispielsweise für die Multiplikation von Matrizen, ein Programm (eine Prozedur) geschrieben hat, darf für die Prozedur einen Bezeichner, z.B. MATRMUL, vereinbaren und im Weiteren wie einen Operationscode verwenden. Wenn er davon beim Programmieren Gebrauch macht, *abstrahiert* er von den Datails der Operation. Der Bezeichner steht für einen "schwarzen Kasten", dessen Inhalt nicht interessiert, sondern nur das, was er ausgibt.

Der Leser vergleiche das Vorgehen mit Bild 5.2. Dort handelt es sich um die Einführung metasprachlicher Variablen für die metasprachliche Beschreibung von Sätzen der deutschen Sprache. In Kap.5.5 hatten wir erkannt, dass die grammatikalische Begriffsbildung (die Einführung metasprachlicher Begriffe) auf *komponierender* und *klassifizierender* bzw. *generalisierender* Abstraktion beruht. Die komponierende Abstraktion ist in Bild 5.2 in senkrechter, die klassifizierende (generalisierende) in waagerechter Richtung dargestellt. Weiter unten werden wir sehen, dass auch bei der Entwicklung von Programmiersprachen nicht nur die komponierende, sondern auch die klassifizierende Abstraktion eine wichtige Rolle spielt. Zunächst wollen wir unser Verständnis für die Rolle der komponierenden Abstraktion erweitern und vertiefen.

Auf einer sehr hohen Komponierungs- und Abstraktionsebene kommen *Anwenderprogrammpakete* zur Anwendung, die dem Computeranwender fachspezifische Unterstützung anbieten. Beispielsweise kann ein Dachdeckermeister seinen Computer beauftragen, alle Berechnungen durchzuführen, die mit der Deckung eines Daches zusammenhängen, von der Materialbestellung bis zum Ausdruck der zu bezahlenden Rechnung. Dabei benutzt er eine *ebene* (zweidimensionale) Sprache. In einem *Fenster* auf dem Bildschirm bietet der Computer *Aktionen* in Form von Symbolen (meist kleinen Bildchen) an. Die Gesamtheit der angebotenen Aktionen heißt **Menü**. Die gewünschten Aktionen ruft der Anwender durch "*Mausklick*" auf, beispielsweise die Berechnung der Dachfläche und anschließend die Berechnung der erforderlichen Anzahl an Dachziegeln. Dazu "klickt" er mit der Maus der Reihe nach die betreffenden Symbol an, d.h. er führt mit der Maus den Mauszeiger, einen mit der Maus verschiebbaren Lichtpunkt, auf eines der Symbole und drückt die Maustaste. Auf diese Weise kann er ein "Aktionsfolgeprogramm programmieren". Ganz ähnlich verfährt ein Statistiker, der ein Statistikprogramm benutzt, oder ein Schriftsteller, der ein Textverarbeitungsprogramm benutzt.

Das *Fensterprinzip* als Kommunikationsmethode zwischen Mensch und Maschine hat sich in weiten Bereichen der Computeranwendung durchgesetzt, nicht zuletzt durch das *Windows-Betriebssystem* (Window = Fenster). In der Regel vereinigt ein Fenster in sich die Funktion eines Menüangebots und eines Fragebogens, in den der Nutzer über die Tastatur seine Eingaben "einträgt".

Wenn die Erteilung eines Auftrags an den Computer über ein Fenster durch Mausklick oder durch Eintragen einer Antwort besteht, ist es kaum noch gerechtfer-

tigt, von einer *Programmier*sprache zu reden, denn der Nutzer programmiert nicht den Computer, sondern er wird vom Computer *abgefragt*. Auch die Bezeichnung *Dialog*sprache für die Kommunikation nach dem Fensterprinzip zwischen Mensch und Computer ist unpassend, weil sie die Vorstellung wecken kann, dass beide Partner ihre Repliken *selber artikulieren*, was für den Computer nicht zutrifft. Passender wäre die Bezeichnung *Abfragesprache* oder, falls die Eingabe ausschließlich per Mausklick erfolgt, *Menüsprache*. Es existiert allerdings keine scharfe Grenze zwischen Programmier- und Abfragesprache, und häufig kommen sie gemeinsam zur Anwendung. Beide Sprachklassen fassen wir unter dem Begriff "**Auftragssprache**" zusammen. Diese Bezeichnung ist bereits in Kap.16.4 [16.12] eingeführt worden.

Auftragssprachen und speziell Programmiersprachen können nach ihrem *Komponierungsgrad* klassifiziert werden, genauer nach dem maximalen Komponierungsgrad der Operationen, für welche die Sprache einen Bezeichner bzw. ein Menüsymbol enthält. Beispielsweise muss der Komponierungsgrad einer Auftragssprache für die Arbeit mit einem umfassenden Mathematiksystem, das u.a.. Differenzialgleichungen lösen kann, sehr hoch sein (siehe dazu Kap.19.4 [19.4] und das Kapitel "Computeralgebrasysteme" in [Bronstein 95]). Im Laufe der Jahre sind Programmiersprachen mit immer höheren Komponierungsgraden entwickelt worden. In diesem Zusammenhang wurde der Begriff der *Sprachgeneration* in Analogie zur Rechnergenerationen vorgeschlagen. Er beruht auf einer sehr abstrakten Analogie und hat sich nicht durchsetzen können.

Dennoch enthält diese abstrakte Generationenanalogie zwischen Hardware- und Softwarekomponierung einen *realen* Kern und zwar die Tendenz zur Spezialisierung. Sie geht naturgemäß mit der Herausbildung immer komplexerer Kompositoperatoren einher. Softwaremäßig erfolgt sie durch fortschreitende prozedurale Abstraktion auf immer höherer Komponierungsebene, hardwaremäßig durch die Realisierung immer komplexere Operationen als mikroelektronische Schaltungen. Jeder Operator, der sich *sprachlich* komponieren lässt, kann auch *real*, d.h. als Schaltung komponiert werden (man denke an die ROM-Hierarchie [13.7]). Viele Rechner verfügen z.B. über eine Schaltung für die Multiplikation von Matrizen. Sowohl für das reale (schaltungsmäßige) als auch für das sprachliche Komponieren gilt, dass mit dem Komponierungsgrad in der Regel auch die Spezialisierung zunimmt, dass also der Anwendungsbereich des Computers bzw. der Sprache eingeschränkt wird. Beispielsweise wird einem Statistiker wahrscheinlich wenig mit einem Vektorrechner gedient sein, der auf das Lösen partieller Differenzialgleichungen spezialisiert ist. Ebenso wenig wird einem Elektriker ein Programmpaket nützen, das für Dachdecker konzipiert ist.

Besonders sinnfällig ist die Generationenalogie im Falle der Bezeichnung "vierte Generation", die sich aber weder für Rechner noch für Sprachen durchgesetzt hat. In den 70er Jahren wurde vorgeschlagen, Computer, die Operatoren hoher Komponierungsstufe hardwaremäßig zur Verfügung stellen, als Rechner der vierten Generation

zu bezeichnen. Ein solcher Rechner stellt ein Operatorennetz aus vielen Prozessoren oder kompletten Computern (sog. Workstations) mit variabler Kopplungsstruktur des Netzes dar. Er arbeitet massiv parall. Dieser Computertyp kommt gegenwärtig wegen seines günstigen Preis/Leistungsverhältnisses zunehmend zum Einsatz. Es werden Rechenleistungen bis nahe an 1 Tflops (10^{12} Gleitkommaoperationen pro Sekunde) erreicht. Derartige Systeme werden als Spezialrechner für ganz bestimmte Aufgaben entworfen, beispielsweise für die Wettervorhersage. In Analogie dazu wurde ebenfalls in den 70er Jahren vorgeschlagen, Sprachen, die für bestimmte Anwendungsgebiete (Diskursbereiche) spezielle Operationen hoher Komponierungsstufe zur Verfügung stellen, als Sprachen der vierten Generation zu bezeichnen.

Von den Maschinensprachen über die soeben charakterisierten "Sprachen der vierten Generation" bis zu den Menüsprachen erstreckt sich ein Spektrum von Sprachen mit steigendem Komponierungsgrad. Man beachte, dass der Begriff des Komponierungsgrades nicht an ein bestimmtes Programmierparadigma gebunden ist und nicht nur für imperative Sprachen Sinn hat. Zwar stellen die gängigen funktionalen Sprachen - abgesehen von einigen Ausnahmen - keine hochkomponierten Funktionen zur Verfügung, dafür aber elegante sprachliche Mittel zur Komponierung von Funktionen, die sich mehr oder weniger deutlich an die Idee des Lambda-Kalküls von A. CHURCH anlehnen (siehe Kap.8.4.7). Das gilt insbesondere für die Sprache *Lisp*. Eine Ausnahme bildet u.a. die Sprache APL, eine funktionale Programmiersprache relativ hohen Komponierungsgrades.

Demgegenüber existieren viele *logische* Sprachen mit hohem Komponierungsgrad, z.B. Anfragesprachen von Datenbanksystemen, die gleichzeitig die Möglichkeit demonstrieren, dass nicht nur die Operationen, sondern auch die Operanden hoch komponiert sein können. Das trifft z.B. für Merkmalswertetupel, Datensätze und Dateien zu, die alle die Rolle von Operanden spielen können.

Man könnte nun erwarten, dass die Sprachevolution nicht nur für Operationen, sondern auch für Daten metasprachliche Begriffe auf der Grundlage komponierender Abstraktion hervorgebracht hat. Tatsächlich hat sich als Pendant zur prozeduralen Abstraktion der Begriff der **Datenabstraktion** herausgebildet. Wir wollen uns überlegen, worum es sich dabei handeln könnte.

Aus der Sicht eines "Rechners", der mit Zahlen rechnet, kann es sich offensichtlich nur um "Zahlenabstraktion" handeln. Jeder, der sich ein wenig mit Mathematik oder Programmieren beschäftigt hat und der sich fragt, worin Abstraktion bezüglich Zahlen bestehen kann, wird wahrscheinlich sofort an die *Klassen* der ganzen Zahlen, der rationalen und der reellen Zahlen denken. Der "Datentyp integer" (Klasse der ganzen Zahlen) und der "Datentyp real" (Klasse der reellen Zahlen; "real" wird ebenso wie "integer" englisch ausgesprochen) sind so ziemlich das Erste, was ein Programmieranfänger lernt.

Wie man sieht, handelt es sich bei dieser Art von Datenabstraktion nicht um komponierende, sondern um *klassifizierende* Abstraktion, also um diejenige Abstraktion, die in Bild 5.2 in waagerechter Richtung dargestellt ist. Als Klassifizieren

hatten wir das Zusammenfassen verschiedener Objekte zu einer *Klasse* bezeichnet, die in bestimmten Merkmalen übereinstimmen. Bei der klassifizierenden Abstraktion hinsichtlich Daten spricht man i.Allg. nicht von Klassen, sondern von *Typen*, genauer von **Datentypen**. In der Typangabe ist alles enthalten, was der Computer wissen muss, um mit "*typisierten*" (oder "*getypten*") Daten umzugehen, u.a. um für sie den erforderlichen Speicherplatz bereitzustellen. Typisierung bewirkt semantische Verdichtung.

Bevor wir das Zusammenspiel klassifizierender und generalisierender Abstraktion in der Evolution der Programmiersprachen genauer analysieren, fassen wir unsere bisherigen Einsichten in folgender Feststellung zusammen: *Die **semantische Verdichtung** von Programmiersprachen und Programmen kann durch komponierende Abstraktion hinsichtlich Operationen (Prozeduren) und klassifizierende Abstraktion hinsichtlich Operanden (Daten) erreicht werden*. Doch wird damit die Komplexität der Sprachevolution nicht voll erfasst. An der *prozeduralen* Abstraktion kann nämlich auch *Klassifizieren* und an der *Datenabstraktion* kann auch *Komponieren* beteiligt sein. Die Ursache hierfür liegt in der inhaltlichen Verknüpfung von Operationen und Operanden. Dieser Sachverhalt tritt bei mathematischen Operationen sehr anschaulich zutage.

Wenn der Computer als "hochqualifizierter Rechner", als mächtiges mathematisches Werkzeug benutzt werden soll, ist es zweckmäßig, eine Sprache mit hohem Komponierungsgrad zur Verfügung zu stellen, die es gestattet, nicht nur Operationen mit booleschen oder numerischen Werten (Zahlen) und Variablen, sondern auch mit Vektoren, Matrizen, Tensoren, Tupeln, Aussagen, Prädikaten oder Mengen in kompakter Form auszudrücken, wie es in der Mathematik üblich ist. Die dafür notwendige prozedurale Abstraktion kann freilich nur dann vom Anwender effektiv genutzt werden, wenn sie durch eine entsprechende Datenabstraktion begleitet wird. Die erforderlichen Datentypen stellen häufig *Kompositdaten* dar, die gemeinsam mit den Kompositoperationen definiert werden, wie z.B. die komplexen Zahlen gemeinsam mit ihrer Addition und Multiplikation.

Das Gleiche gilt für die Komposition *nicht*mathematischer Operationen und Daten, wie das Beispiel der Datenbanken zeigt. Datensätze und Dateien sind *Kompositdaten*, die ohne entsprechende Kompositoperationen (Suchen, Aktualisieren) wenig Sinn hätten. Doch obwohl sie durch Komponierung entstehen, beruht die Definition entsprechender syntaktischer Variablen, also die Definition des Datentyps "*Datensatz*" oder "*Datei*", auf *klassifizierender* Abstraktion. Das gleiche gilt für die Definition des Datentyps "*komplexe Zahl*", "*Array*" oder "*Menge*". Das klassische Beispiel ist der Datentyp "*record*", den NIKLAUS WIRTH mit der Definition seiner Sprache *Pascal* eingeführt hat und der nichts anderes darstellt als einen Datensatz-Typ.

Die Möglichkeit, Komposit*daten* zu Klassen (Typen) zusammenzufassen, wirft die Frage auf, ob sich in Analogie dazu auch aus Komposit*operationen* Klassen (Typen) bilden lassen. Offenbar ist das möglich. Beispielsweise können die Additio-

nen ganzer bzw. reeller bzw. komplexer Zahlen, hinter denen sich unterschiedliche konkrete Operationen verbergen, zur “*Klasse der Additionen*” zusammengefasst werden. Wo es Sinn hat, kann man auch die Addition von Vektoren und von Matrizen hinzunehmen, eventuell sogar auch die “Addition” (Disjunktion) boolescher Werte oder die “Addition” (Vereinigung) von Mengen.

Genau genommen handelt es sich dabei nicht um Klassifizieren im Sinne von Bild 5.5 (Übergang vom Exemplar zur Klassen), sondern um *Generalisieren* (Übergang Unterklasse - Oberklasse), denn die Addition stellt bereits eine Klasse dar, deren Exemplare die Additionen konkreter Summanden sind. Die Generalisierung kann fortgesetzt werden, indem beispielsweise alle Operationen mit ganzen oder reellen Zahlen zur Klasse der *arithmetischen Operationen* zusammengefasst werden. Auf diese Weise kann sich eine *Hierarchie* von Operationstypen ergeben. Hinsichtlich *Daten*typen kann man analog verfahren. Ganze Zahlen, reelle Zahlen und komplexe Zahlen stellen drei Datentypen mit zunehmendem **Generalisierungsgrad** dar. Die allgemeinste Klasse von Operationen bzw. von Daten, der wir begegnet sind, hatten wir mit *op* bzw. mit *od* oder *id* bezeichnet; im Rahmen der USB-Methode hatten wir die Bezeichnung *od* und im Zusammenhang mit der syntaktischen Analyse (Kap.16.4) die Bezeichnung *id* (von Identifikator) verwendet.

Generalisierte Klassen (Typen) werden in der Programmierungstechnik - in Abhängigkeit von Realisierungsnuancen - auch als **generische** oder **polymorphe Typen** bezeichnet. Ihre Verwendung bewirkt eine zusätzliche semantische Verdichtung, eine Erhöhung der Ausdrucksstärke der Sprache. Bei der Übersetzung eines Programms, das derartige Typen verwendet, muss das Übersetzerprogramm bei der syntaktischen Analyse feststellen, welcher konkrete Typ im gegebenen Fall vorliegt.

Interessanterweise ist die Evolution der Programmiersprachen *nicht* in Richtung zunehmender Generalisierung von Operationstypen einerseits und Datentypen andrerseits verlaufen, sondern in Richtung ihrer Kombination. Dabei verfolgten die Sprachentwickler zwei Ziele, eine noch effizientere Programmierung durch zusätzliche semantische Verdichtung und gleichzeitig die Sicherung einer fehlerfreien Programmausführung. Das Prozedurkonzept birgt nämlich die Gefahr in sich, dass die Abarbeitungsprozesse verschiedener Prozeduren sich gegenseitig stören. Diese Gefahr besteht immer dann, wenn zwei oder mehrere Prozeduren dieselben Daten benutzen, wenn sie also während ihrer Abarbeitung auf dieselben Speicherplätze zugreifen. Dieses Problem führt uns in die Domäne des *Betriebssystems*, das für die Prozessorganisation zuständig ist. Darum verschieben wir eine ausführlichere Behandlung auf Kap.19.5. Doch soll schon jetzt der Gang der weiteren Entwicklung skizziert werden, die durch zwei Ideen geprägt ist, die mit den Worten *Kapselung* und *Vererbung* benannt werden.

Kapselung. Die erste der beiden Ideen richtet sich gegen die genannte Gefahr der gegenseitigen Störung von Prozessen. Die Idee wird durch den oben erwähnten Umstand nahegelegt, dass Operationen und ihre Operanden nur *gemeinsam* Sinn ergeben und insofern eine Einheit bilden. Wenn dafür gesorgt wird, dass auf dieje-

nigen Daten, mit denen eine Prozedur arbeitet, nur diese Prozedur selber zugreifen kann, lassen sich störende Wechselwirkungen zwischen den Ausführungsprozessen verschiedener Prozeduren vermeiden. Das ist allerdings nicht restlos zu verwirklichen, da zwischen den Prozeduren, die als Bausteine eines oder auch mehrerer Programme dienen, eventuell Daten übergeben werden müssen. Dennoch ist es möglich, dass ein Prozedur ihre Daten weitgehend vor anderen Prozeduren "*verbirgt*" (sog. **data hiding**).

Diese Idee führte zur Erfindung neuer Sprachelemente durch Zusammenfassung von Operanden und Operationen (Daten und Prozeduren) zu programmierungstechnischen Einheiten. Für diese Einheiten wurden je nach Sicht des Erfinders und in Abhängigkeit von Realisierungsnuancen verschiedene Bezeichnungen vorgeschlagen wie **abstrakter Datentyp**, **Datenkapsel** und **Modul**. Der Kern ist in jedem Fall das "data hiding", also das *Schützen* prozedureigener Daten gegen den Zugriff durch andere Prozeduren. Das Wort "*Kapselung*" bringt dieses Anliegen anschaulich zum Ausdruck.

Vererbung. Die zweite Idee besteht darin, in einer *Klassenhierarchie* von Prozeduren auf jeder Hierarchieebene nur dasjenige explizit zu programmieren, was beim Abstieg in der Hierarchie hinzukommt, also die zusätzlichen (als Vorschrift formulierten) "*Merkmale*", die beim Übergang von der Oberklasse zur Unterklasse die *Präzisierung* ausmachen. Die Merkmale (Vorschriften) der Oberklasse werden an alle Unterklassen *vererbt*, wodurch Programmtext eingespart und so ein Verdichtungseffekt erzielt wird. Das Vorgehen soll am Beispiel der Implementierung des Operatorennetzes von Bild 8.1 (der Funktion (8.1)) erläutert werden. 4

Jeder Operator, jeder Flussknoten und jeder Operandenplatz hat seine Ein- und Ausgänge. In jedem Abarbeitungstakt des zu erstellenden Programms muss festgestellt werden, wo welche Operanden übergeben werden, m.a.W. ob und gegebenenfalls welcher Operand einen bestimmten Eingang bzw. Ausgang passiert. Diese Operation kann für die "*Klasse aller Bausteine des Netzes*" einheitlich programmiert werden. Diese allgemeinste *Klasse* wird im folgenden Präzisierungsschritt in die *Klassen* der Operatoren, Flussknoten und Plätze zerlegt. Im nächsten Präzisierungsschritt kann z.B. die Klasse der Operatoren in arithmetische und trigonometrische Operatoren und die Klasse der Flussknoten in starre und steuerbare Flussknoten zerlegt werden.

In jedem Präzisierungsschritt werden jeweils die notwendigen Programmergänzungen programmiert. Im Falle der Präzisierung der Operatoren sind die Ergänzungen Vorschriften zum Addieren, Multiplizieren bzw. zur Berechnung der Sinusfunktion. In Kap.20.3 ist das Vorgehen vollständig ausprogrammiert.

Den entscheidenden Anstoß zur weiteren Sprachevolution lieferte die Idee, Kapselung und Vererbung zu vereinigen und ein entsprechendes neues Sprachkonstrukt, einen neuen *Typ* zu definieren, der die Bezeichnung **Objekt** erhielt. Diese Wortwahl mag ungeschickt erscheinen, denn das semantisch hochgeladene Wort "Objekt" kann irritieren und missverstanden werden. Gegebenenfalls werden wir der Eindeutigkeit 5

halber von *informatischen* bzw. von *umgangssprachlichen* Objekten sprechen. In Kap.18.3 wird sich die Berechtigung der Wortwahl “Objekt” herausstellen. Eine Programmiersprache, die den Typ “*Objekt*” zur Verfügung stellt, heißt **objektorientierte Sprache**.

Ein Objekt ist ein relativ abgekapselter Programmbaustein, der über eigene Prozeduren, **Methoden** genannt, und über einen privaten Speicher verfügt und damit über eigene Daten, mit denen die Methoden des Objekts operieren. Ein Objekt kann als Spezialist aufgefasst werden, der bestimmte Dinge *kann* (bestimmte Methoden beherrscht). Objekte können sich gegenseitig um **Dienstleistungen** bitten. Die einzelnen Objekte eines objektorientierten Programms bilden gewissermaßen ein Netz von Kooperationspartnern, die sich gegenseitig Aufträge erteilen können. Für diese Art der Auftragserteilung wird auch das Wort *Direktive* verwendet, und es wird von *direktivem* statt von *objektorientiertem* Programmieren gesprochen [Schefe 87].

Durch Verwendung einer objektorientierten Sprache kann nicht nur die *Sicherheit*, sondern eventuell auch die *Effizienz* der Ausführung von Programmen erhöht, d.h. die Laufzeit verkürzt werden, zumindest bei geschickter Programmierung. Da ein Objekt seine Methoden und seinen Speicher selbst verwaltet, führt das objektorientierte Programmieren zu einer Dezentralisierung der Steuerung und alle damit verbundenen Vorteile können genutzt werden. Darüber hinaus bedeutet die Einführung des Typs (der metasprachlichen Variablen) “*Objekt*” einen weiteren Schritt zur Lösung des *technischen Semantikproblems*, wie im folgenden Kapitel gezeigt wird.

18.3 Technisches Semantikproblem und Programmierparadigmen

Die charakteristische Schwierigkeit, vor der jeder Nutzer eines Computers steht, liegt in der Notwendigkeit, das eigene Denken, die eigenen Vorstellungen in einer Sprache auszudrücken, die der Computer “versteht”. Den Kern dieser Schwierigkeit hatten wir in Kap.15.5 [15.6] in der *semantischen Anbindung* erkannt, d.h. in der Anbindung der *Nutzersemantik* (der *externen* Semantik, in welcher der Nutzer denkt) über eine *formale Semantik* (die Semantik eines Kalküls bzw. einer Programmiersprache) *an die interne Maschinensemantik*, also an die Prozesse, die im Computer ablaufen. Das Problem der semantischen Anbindung hatten wir das “*technische Semantikproblem*” genannt. Bild 18.1 veranschaulicht die Problematik.

Bild 18.1 enthält die gleichen semantischen Ebenen wie die Bilder 5.3 und 8.6. Doch zeigt Bild 18.1 einen anderen Weg auf, der zu den drei Ebenen führt. Ausgangspunkt sind diesmal verschiedene *Sprach*ebenen. In der linken Spalte sind drei Sprachklassen definiert, denen in der rechten Spalte die jeweilige Semantik zugeordnet ist. Eine Sprache, in der ein Analytiker oder ein Modellierer über das Original nachdenkt und ein erstes Modell entwirft, nennen wir **Modelliersprache**. Eine Modelliersprache trägt die *externe* Semantik des Nutzers, d.h. die Interpretationen

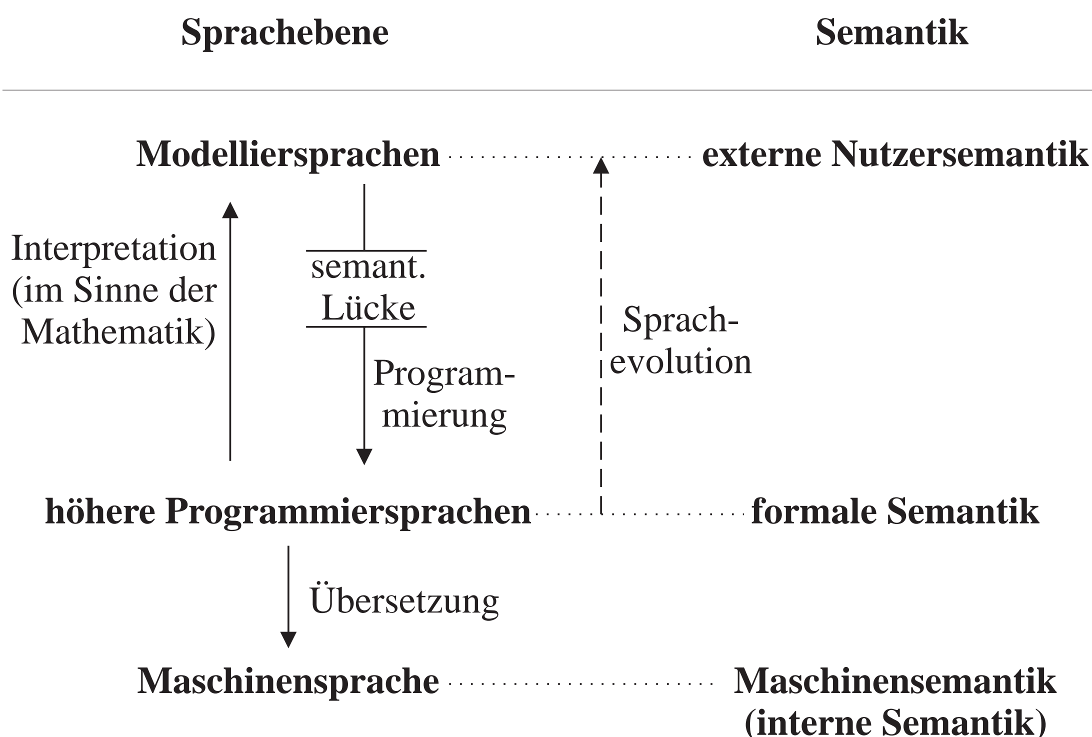


Bild 18.1 Zum technischen Semantikproblem

modelliersprachlicher Ausdrücke sind Vorstellungen (Gedanken, Ideme) des Nutzers. Die Anbindung der externen Nutzersemantik an die interne Maschinensemantik erfolgt normalerweise in zwei Schritten. Im ersten Schritt wird das Modell von einem Programmierer in einer höheren Programmiersprache ausprogrammiert, wodurch das Modell von der oberen in die mittlere Ebene überführt wird. Dabei erfolgt die Anbindung der externen Semantik des Nutzers an die formale Semantik der Programmiersprache. Im zweiten Schritt wird das Programm von einem Übersetzerprogramm in die Maschinensprache übersetzt, wodurch das Modell von der mittleren in die untere Ebene überführt wird. Dabei erfolgt die Anbindung der formalen Semantik an die interne Semantik der Maschine.

In beiden Schritten können Fehler auftreten, die sehr häufig durch unkorrekte semantische Anbindung verursacht sind. Die Folge ist, dass der Computer nicht das "tut" (rechnet), was der Modellierer "will", was er seiner Meinung nach in der Modelliersprache artikuliert hat. Eine korrekte semantische Anbindung der externen Semantik an die Maschinensemantik kann nur dann garantiert werden, wenn sowohl das Modell der obersten Ebene (und damit die externe Realität) als auch das Maschinenprogramm der untersten Ebene (und damit die computerinterne Realität) Interpretationen des durch die Programmiersprache der mittleren Ebene definierten Kalküls sind. Jeder Programmierer wird zwar bemüht sein, dieser Forderung auf mehr oder weniger systematische Weise nachzukommen, der Nachweis, dass die Bedingung tatsächlich erfüllt ist, wird aber nur selten erbracht.

Beim Übergang von der obersten zur untersten Ebene müssen nicht unbedingt alle drei Schritte explizit ausgeführt werden. Der zweite Schritt entfällt, wenn der Programmierer in der Maschinensprache programmiert. Der erste Schritt entfällt, wenn Modelliersprache und Programmiersprache zusammenfallen, wenn also die Sprache, in welcher der Modellierer gewohnt ist zu denken und das Gedachte zu artikulieren, implementiert ist. Für ein mathematisch formuliertes Modell ist das durchaus möglich. Denn es existieren umfangreiche Mathematiksysteme, deren Eingabesprachen (Auftragssprachen) weitgehend an mathematische Sprachen angelehnt sind, sodass der Modellierer sein Modell praktisch in der ihm gewohnten Sprache "programmieren" und unmittelbar vom Rechner ausführen lassen kann. Falls eine *physikalische Theorie* implementiert ist, erübrigen sich möglicherweise beide Schritte des semantischen Anbindens.

Auch der umgekehrte Weg ist möglich, der darin besteht, dass nicht die Modelliersprache an die implementierte Kalkülsprache angelehnt wird, sondern dass die Kalkülsprache an die - evtl.verbale - Modelliersprache angepasst wird. Der **Fuzzy-Kalkül** macht diesen Weg gangbar. Er erlaubt es, dem Computer teilweise verbale (man sagt auch "linguistische") Aussagen mit unscharfer Semantik anzubieten. In Kap.21.3.2 wird die Idee der Methode skizziert.

Unsere Überlegungen zeigen, dass der Begriff des sprachlichen Modells in recht unterschiedlichen Bedeutungen verwendet wird. Wir hatten ihn sehr allgemein als eine *durch idealisierende Abstraktion vereinfachte Beschreibung eines Originals* eingeführt [2.1], wobei die Beschreibung aus Aussagen über Merkmale des Originals besteht. Um die Unterschiede in der Verwendung dieses Begriffs deutlicher herauszuarbeiten, wollen wir die Rolle der mittleren Ebene beim Modellieren ins Auge fassen und untersuchen, was die Kalkülisierung genau beinhaltet und wie der Kalkül in das Modell eingeht.[3] Um uns nicht im Abstrakten zu verlieren, betrachten wir die folgenden konkreten Modelle.

1. Computermodell eines Produktionsbetriebes (Kap.15.7)
2. Modell verwandschaftlicher Beziehungen (Kap.16.1)
3. Modell der Kohlenwasserstoff-Moleküle (Kap.16.3)
4. Mathematisches Modell des Planetensystems (Kap.4.2).

Geht man die vier Modelle der Reihe nach durch, erkennt man, dass der jeweils verwendete Kalkül eine immer grundsätzlichere Rolle im Modell spielt; aus einer *punktuellen* Rolle im ersten Modell wird eine *globale* im letzten. Dabei ist unter "punktuell" die unzusammenhängende Interpretation des Kalküls durch einzelne Modellaussagen zu verstehen und unter "global" die ganzheitliche Interpretation des Kalküls durch das Modell und damit durch das Original. Mit der schrittweisen

---

3 Die Anregung zu den folgenden Überlegungen gab der Artikel "Softwaretechnik und Erkenntnistheorie" von PETER SCHEFE [Schefe 99], in dem der Unterschied zwischen naturwissenschaftlichen und programmiersprachlichen Modellen aus erkenntnistheoretischer Sicht diskutiert wird.

Globalisierung werden die kausalen Zusammenhänge des Originals durch das Modell immer tiefer erfasst. Durch die Interpretation des Kalküls (durchgezogener Aufwärtspfeil in Bild 18.1) werden Aussagen der formalen Ebene mit externer Semantik belegt. Das Gesagte soll anhand der vier genannten Modelle illustriert werden.

7

Das Computermodell eines Produktionsbetriebes möge der *quantitativen Beschreibung* des Produktionsablaufs dienen, also der zahlenmäßigen (numerischen) Berechnung von Werten interessierender Merkmale. Die prozedurale Abstraktion bzw. die objektorientierte Programmierung gibt die Möglichkeit, das Modell als Operatorenhierarchie zu formulieren, wobei die hierarchische Struktur des Modells derjenigen des Originals entsprechen sollte [15.11]. Darin liegt ein wichtiger Schritt zur Überwindung der semantischen Lücke. Interessierende Merkmale können u.a. Durchlaufzeiten von Werkstücken durch bestimmte Bearbeitungsabschnitte (z.B. durch das Operatorennetz von Bild 8.3.), die Auslastung von Maschinen, benötigte Materialmengen oder der Gewinn sein. Das Modell verwendet den arithmetischen Kalkül. Dieser wird "punktuell" interpretiert und angewendet, d.h. für die Berechnung der verschiedenen interessierenden Größen werden spezielle Programme geschrieben ("erfunden"). Tiefer liegende kausale Zusammenhänge werden nicht erfasst. In diesem Sinne ist die Kalkülisierung "*oberflächlich*".

Im Modell des Planetensystems kommt der Kalkül der Analysis zur Anwendung. Er ist bedeutend "*tiefer*" in das Modell des Diskursbereiches eingebunden. Das Modell erfasst die kausalen Zusammenhänge vollständig, in ganzer *Tiefe*. In diesem Sinne sagen wir, dass der **Kalkülisierungsgrad** des Modells maximal ist. Auch in diesem Modell können einzelne Aussagen berechnet werden, z.B. die Umlaufzeit der Erde um die Sonne. Doch muss dafür nicht eine spezielle Rechenvorschrift (Formel) "ad hoc erfunden" werden; vielmehr lässt sie sich aus dem Modell nach den Regeln des Kalküls "ableiten". Das Modell produziert neue Modellaussagen. Das Original (der Diskursbereich) wird "als Ganzes" oder "global" durch das formale Modell erfasst. Grundlage sind die *Erkenntnisse*, die Newton in seinen Prinzipien der Mechanik formuliert hat, die Hypothese der Gravitationskraft und die Erfindung der Differenzialrechnung. Das Modell ist eine *physikalische Theorie*.

Ähnliche Verhältnisse liegen im Modell der Kohlenwasserstoff-Moleküle vor. Grundlage sind die chemischen *Erkenntnisse*, die als Grundwissen bzw. als Axiome [16.8] formuliert wurden. Aus ihnen können alle interessierenden Aussagen über den Diskursbereich, d.h. "alle möglichen" (von den Axiomen erlaubten) Moleküle aus C- und H-Atomen *abgeleitet* werden. Im Gegensatz zum Planetenmodell stellt dieses Modell keine physikalische Theorie dar. Um das zu erreichen, müsste das Faktenwissen über die Wertigkeiten aus der elektromagnetischen Wechselwirkung zwischen C- und H-Ionen abgeleitet werden.

Auch im Modell der Verwandtschaftsbeziehungen können neue Aussagen aus bekanntem Wissen *abgeleitet* werden. Doch beruht das Wissen nicht, wie in den letzten beiden Modellen, auf der experimentellen Bestimmung (Messung) von Merk-

malswerten, sondern auf dem Umstand, dass jeder Mensch Vater und Mutter hat. Im Übrigen beruht es auf Vereinbarungen.

8 Im Modell eines Produktionsbetriebes beruht das Faktenwissen sowohl auf Messungen (z.B. Messung der Dauer einer Operation) als auch auf Vereinbarungen (z.B. Festlegung von Preisen). Das Wissen hat aber in jedem Falle "punktuellen" Charakter, es betrifft bestimmte Merkmale bestimmter Objekte, z.B. bestimmter Werkstücke oder bestimmter Operationen. In den anderen drei Beispielen hat das Wissen "globalen" Charakter für den gesamten Diskursbereich. Beispielsweise gelten die Bewegungsgleichungen für sämtliche Planeten und die Axiome der C-H-Verbindungen gelten für beliebige Kohlenwasserstoffmoleküle.

Aus den Beispielen lässt sich folgende verbale Begriffsbestimmung extrahieren. *Der **Kalkülisierungsgrad** eines sprachlichen Modells ist der Grad der Durchgängigkeit und Geschlossenheit der Kalkülisierung des Modells.* Je umfassender und geschlossener das Modell kalkülisiert ist, umso allgemeiner sind die Modellaussagen. Das bedeutet, dass mit zunehmendem Kalkülisierungsgrad die Aussagen Variablen betreffen, z.B. Aussagen über (irgendwelche) Planeten und deren Koordinaten oder Aussagen über (irgendwelche) Atome und deren Verbindungen oder Aussagen über (irgendwelche) Personen und deren verwandtschaftliche Beziehungen. Das Modell wird also zunehmend *analytisch*. Damit ergibt sich die Möglichkeit, ein Maß für den

9 Kalkülisierungsgrad anzugeben. *Der **Kalkülisierungsgrad** eines Modells ist das Verhältnis von analytischem zu numerischem Rechenaufwand (gemessen in elementaren Rechenschritten) beim Ableiten von Modellaussagen ohne Berücksichtigung des numerischen Lösens von Gleichungen.*

Die Analyse des Grades der Kalkülisierung stellt eine Ergänzung zur Klassifikation von Modellen in Bild 3.1 dar. Dort hatten wir zwischen formalisierten und nichtformalisierten sprachlichen Modellen unterschieden. Inzwischen wissen wir, dass formalisiertes Modellieren stets die Interpretation eines Kalküls durch das modellierte Original beinhaltet. In Bild 3.1 kann also "formalisiert" durch "kalkülisiert" ersetzt werden. Aufgrund unserer Analyse erkennen wir nun, dass die Unterteilung in zwei Modellklassen, formalisierte und nichtformalisierte, der Vielfalt möglicher sprachlicher Modelle nicht voll gerecht wird, dass vielmehr verschiedene Grade der Formalisierung möglich sind. Es läge nahe, von Formalisierungsgrad zu sprechen, doch ziehen wir den ausdrucksstärkeren Begriff des Kalkülisierungsgrades vor.

Damit beenden wir den Abstecher in die Problematik der Modellklassifikation und wenden uns dem Pfeil "Programmierung" in Bild 18.1 zu. Wie bereits erwähnt, entfällt die Programmierschicht, wenn die Modelliersprache implementiert ist. Das ist jedoch die Ausnahme. Normalerweise liegt zwischen der Modelliersprache und der Programmiersprache die in Kap.15.5 ausführlich behandelte *semantische Lücke*, die der Programmierer "überspringen" muss, um die externe *Nutzersemantik* an die Semantik einer maschinenverständlichen Sprache anzubinden. Jede Verringerung der semantischen Lücke erleichtert ihm die Arbeit. Dadurch entsteht ein Selektions-

druck in Richtung des gestrichelten Pfeils in Bild 18.1, der die Sprachevolution vorantreibt.

Das Wort "Selektionsdruck" ist durchaus berechtigt, denn die Phantasie der Spracherfinder brachte eine unübersehbare Menge von Programmiersprachen hervor, die immer noch wächst. In der praktischen Arbeit mit den Sprachen haben sich manche bewährt und sind weiterentwickelt worden, andere sind ausgestorben. Die Entwicklung ging in verschiedene Richtungen, denen unterschiedliche Konzepte und Denkschemata zugrunde liegen. So entstanden unterschiedliche **Sprachparadigmen**. Mit dem Wort "Paradigma" ist in diesem Zusammenhang die konzeptionelle Grundlage einer Sprache gemeint, die einem bestimmten *Denkschema* des Programmierers angepasst ist, z.B. dem Denken in Algorithmen, in Funktionen oder in Operatorennetzen.[4]

Der abstrakte Begriff des Programmierparadigmas hat sich als "Nebenprodukt" der Bemühungen herausgebildet, die semantische Lücke zu veringern und die Programmiersprachen immer besser den Denkgewohnheiten der Programmierer und, wenn möglich, auch der Programmnutzer anzupassen. Dieses Ziel war die Motivation für viele Überlegungen und Ideen der Kapitel 15, 16 und 17. Die Entwicklung *funktionaler* oder *logischer* (relationaler) Programmiersprachen erleichtert das "Umcodieren" *funktional* bzw. *relational* formulierter Modelle (Modelle, die von "funktional" bzw, "relational" denkenden Modellierern erstellt sind) in eine Programmiersprache.

Angesichts der weiten Verbreitung des funktionalen Denkens in Naturwissenschaft und Technik ist die große Bedeutung des *funktionalen* Programmierparadigmas neben dem ursprünglichen, maschinenorientierten *imperativen* Paradigma nicht verwunderlich. Im Alltagsleben denken die Menschen aber doch relativ selten in Funktionen und weit öfter in Wenn-Dann-Zusammenhängen, d.h. sie denken *relational* oder "*logisch*" im Sinne des *logischen* Programmierparadigmas. Auch wir haben bei der Lösung der Verwandschaftsaufgabe relational gedacht.

Wegen der Netzorientiertheit des menschlichen Denkens hätte man erwarten können, dass die genannten Paradigmen zunehmend durch ein "*datenflussorientiertes Paradigma*" verdrängt werden, d.h dass zumehmend Programme als *Datenflusspläne* formuliert werden. Denn ein Datenflussplan kommt dem netzorientierten Denken am nächsten. Es gab Ansätze in dieser Richtung, die auf der Konzeption entsprechender Hardware in Form sogenannter *Datenflussmaschinen* basierten. Es haben sich aber weder die Datenflussmaschinen noch das datenflussorientierte Paradigma durchsetzen können. Doch ist das datenflussorientierte Paradigma in zwei andere Paradigmen eingeflossen, in das funktionale und in das *objektorientierte* Paradigma.

---

4 Zu den begrifflichen und philosophischen Hintergründen dieser Entwicklung findet der Leser in den Artikeln [Pflüger 94] und [Pflüger 97] interessante Ausführungen.

In Kap. 15.1 [15.1] hatten wir festgestellt, dass dem funktionalen Programmieren die Vorstellung eines Datenflusses zugrunde liegt. Das tritt in dem Umstand zutage, dass in einem funktionalen Programm ebenso wie in einem Datenflussprogramm (Datenflussplan) Zwischenresultate nicht explizit in Erscheinung treten, da sie gewissermaßen automatisch durch den Datenfluss weitergetragen werden. Insofern ist das funktionale Paradigma *datenflussorientiert* zu nennen.

Seit der Mitte der achtziger Jahre hat es sich zunehmend eingebürgert, diejenige Programmierweise, die sich infolge der Verwendung von Objekten (des Datentyps "Objekt") herausgebildet hat, als eigenständiges Paradigma aufzufassen und vom **objekorientierten Programmierparadigma** zu sprechen. Es vereinigt in sich Charakteristiken sowohl des imperativen als auch des funktionalen Paradigmas. Um das zu erkennen, lenken wir zunächst unsere Aufmerksamkeit auf eine spezifische Eigenschaft, die Objekte und Operatoren gemeinsam besitzen. Wir erinnern uns an die in Kap.18.2 [6] eingeführten Begriffe des Objekts und des objektorientierten Programms und zitieren:

"*Die einzelnen Objekte eines objektorientierten Programms bilden gewissermaßen ein Netz von Kooperationspartnern, die sich gegenseitig Aufträge erteilen können*."

Ihm stellen wir einen Satz aus Kap.13.7 [13.14] gegeüber:

"*In der Welt der Operatorennetze lässt sich die Menge der verkoppelten Operatoren als Menge kooperierender Akteure auffassen, die sich gegenseitig Daten zur weitern Verarbeitung zuspielen*."

Die gemeinsame Eigenschaft ist die Fähigkeit, Netze zu bilden, Netze kooperierender Partner oder Akteure. Betrachtet man ein aus dem Netz herausgelöstes Objekt, einen isolierten Akteur, so führt dieser eine Folge von Direktiven (Aufträgen, Befehlen) aus. Hierin liegt die *imperative* Komponente der objektorientierten Programmierung. Demgegenüber stellt die kooperative Bearbeitung eines Auftrages durch die Objekte eines objektorientierten Programms eine *datenflussorientierte* und damit auch eine *funktionale* Komponente dar.

10

Die auffälligste Ähnlichkeit besteht jedoch zwischen dem (nicht existierenden) datenflussorientierten Paradigma und dem objektorientierten Paradigma, konkreter zwischen Datenflussplänen und objektorientierten Programmen. Das zeigen die beiden zitierten Sätze. Fast könnte man geneigt sein, beide Paradigmen zu identifizieren und "objektorientiert" durch "datenflussorientiert" zu ersetzen. Doch würde damit ein wesentlicher Aspekt des Objektbegriffs verloren gehen, auf den ausführlicher eingegangen werden soll.

Wie bereits bemerkt, mag die Wahl des Wortes "Objekt" ungeschickt erscheinen, weil dieses Wort bereits umgangssprachlich belegt ist, wobei die Bedeutung des umgangssprachlichen Objektbegriffs wenig mit der des informatischen Objektbegriffs, dem die Vorstellung der Kapselung zugrunde liegt, zu tun zu haben scheint. Bei genauerer Betrachtung erkennt man jedoch, dass es gerade die Vorstellung der

relativen Kapselung ist, die den umgangssprachlichen Objektbegriff mit dem informatischen verbindet.

In Kap.5.5 [5.17] hatten wir festgestellt, dass sich im Denken des Menschen durch Beobachtung der Welt einzelne Objekte dadurch "herauskristallisieren", dass sie mit Merkmalswerten in Verbindung gebracht und im Endeffekt durch diese "definiert" und dadurch zu *Denkobjekten* werden [5.18]. Hierauf beruht das "Begreifen" der Welt durch das Neugeborene und dessen intellektuelle Entwicklung. Hierauf beruhen auch die begriffsbildenden Operationen von Bild 5.4, z.B. das Generalisieren und Präzisieren, also diejenigen Operationen, auf denen die Vererbung von Eigenschaften informatischer Objekte beruht. Entscheidend für die intellektuelle und kulturelle Evolution ist die relative Abgeschlossenheit und Stabilität der Denkobjekte. Im Denken treten Denkobjekte miteinander in Wechselwirkung, ohne sich gegenseitig zu stören (zu *zer*stören). Genau diese Eigenschaft bildet die Grundlage des informatischen Objektbegriffs. Hierin liegt eine nicht sofort erkennbare, aber umso stichhaltigere Rechtfertigung der Wortes "Objekt" als Name des so benannten Datentyps.

Eine zweite, sehr sinnfällige Rechtfertigung ergibt sich daraus, dass auf einem ausreichend hohen Abstraktionsniveau der informatische mit dem umgangssprachlichen Objektbegriff verschmilzt (begriffliche Konvergenz). In der Losung "*Jedem Objekt in der Wirklichkeit sein Objekt im Programm!*" hat die Verschmelzung ihren treffenden Ausdruck gefunden. Dahinter verbirgt sich eine neue Möglichkeit der Annäherung zwischen Programmier- und Modellierebene. Ein weiterer Schritt in Richtung des gestrichelten Pfeils in Bild 18.1, also in Richtung der Schließung der semantischen Lücke konnte getan werden. Das Ergebnis dieses Schrittes besteht darin, dass Denkobjekte, mit denen der Modellierer hantiert, in Objekte eines objektorientierten Programms überführt werden können. 11

Das Neue und Bedeutende des objektorientierten Ansatzes im Vergleich zum datenflussorientierten Ansatz (zur Operatorennetzmethode) besteht darin, dass Objekte keine Operatoren, keine Akteure sein müssen, dass sie vielmehr programmiersprachliche Repräsentationen beliebiger Denkobjekte sein können, die in Beziehung zueinander stehen und die sowohl im Denkprozess als auch im Abarbeitungsprozess eines objektorientierten Programms "ungestört" miteinander wechselwirken können.

Dadurch kommt eine neue Dynamik in den Abarbeitungsprozess hinein. Denn die Reaktion eines Objekts auf eine Direktive (auf eine Bitte um eine Dienstleistung) ist nicht durch den Programmierer vollständig vorherbestimmt, sondern kann von den momentanen Gegebenheiten abhängen, insbesondere vom inneren Zustand (vom "Wissenstand") des Objekts. Diese Dynamisierung ist ein entscheidender Vorteil des objektorientierten Paradigmas. Programmierungstechnisch wird sie durch *dynamisches Binden* realisiert [15.5].

Wie man sieht, besitzen Objekte im Vergleich zu den Prozeduren imperativer Programme mehr "Freiheiten", den Prozessverlauf mitzubestimmen, obwohl Objekte - ebenso wie Prozeduren - durch Direktiven (Befehle), die sie empfangen, aktiviert werden. Die Idee liegt nahe, auch *diese* "Unfreiheit" zu beseitigen und dem Objekt

die “Freiheit” zu geben, von sich aus aktiv zu werden, m.a.W. das Objekt mit *Eigeninitiative* auszustatten. Diese Idee ist bereits realisiert. Das Produkt, ein Objekt oder besser ein Akteur, der nicht nur auf Direktiven wartet, sondern selber aktiv werden kann, wird **Agent** genannt. Vertieft man sich in diese Idee, öffnet sich ein weites Feld neuer Perspektiven, die hier nur angedeutet werden sollen.

Wenn die Aktivität eines Agenten sinnvoll sein soll, darf sie nicht rein zufälligen Charakter haben, sondern muss irgendwelche Ziele verfolgen. Ein Agent muss seine Aktivität aus der Aktivität der Umgebung gemäß seiner *eigenen Intentionen* ableiten und sich in diesem Sinne der Umgebung anpassen. Der Unterschied zwischen Objekt und Agent besteht also schlagwortartig ausgedrückt darin, dass ein Objekt sich durch sein Wissen und Können auszeichnet, während ein Agent sich durch sein Wissen, Können *und Wollen* auszeichnet. Alle drei Eigenschaften liegen in der den Agenten implementierenden Software begründet. Sie können vom Programmierer vorgegeben werden oder sich im Wechselwirkungsprozess mit der Umgebung, d.h. mit anderen Agenten entwickeln. In letzterem Fall muss der Programmierer die Fähigkeit des Agenten, sich zu entwickeln, zu lernen, d.h. Erfahrungen zu sammeln und zu nutzen, einprogrammieren.

Was aber kann der Agent wollen? Wenn wir bei unserer ursprünglichen Zielstellung bleiben, ein Gerät zu bauen, das dem Menschen als *Denkassistent* dient, lautet die Antwort: *Ein Agent muss wollen, was sein Nutzer will*, d.h. was der Nutzer des betreffenden “*agentenorientierten*” Programms will. Das können die verschiedensten Aktionen sein. Ein Agent kann die Aufgabe haben, seinen “Herrn” an Termine zu erinnern, die Eingangspost durchzusehen und zu filtern, Schreiben zu verfassen oder seinen Herrn auf Fehler hinzuweisen. Der Bürger der Informationsgesellschaft kann *persönliche Assistenten* “einstellen” und im Internet aktiv werden lassen, um gezielt Informationen einzuholen bzw. zu verteilen oder um sich auf Chancen oder Gefahren aufmerksam machen zu lassen. Aus einem Expertensystem wird ein Agent oder ein Kollektiv von Agenten, von denen jeder ein Experte ist, der die Ziele des Nutzers verfolgt.

Die Eigenschaften und Fähigkeiten eines Agenten in der Form des persönlichen Assistenten überschreiten nicht den Rahmen der künstlichen Intelligenz, wie sie bisher besprochen worden ist. Der Schritt in unbekanntes Terrain wird dann vollzogen, wenn ein Agent “*wollen kann, was er will*”. Das bedeutet, dass der Agent über einen freien Willen und damit über Selbstbewusstsein verfügt.

Es wird den Leser nicht überraschen, dass hier der Punkt erreicht ist, an dem wir unsere Wanderung zu den Grenzen der künstlichen Intelligenz abbrechen. Wir wollten verstehen, wie sich auf der Grundlage des gegenwärtigen Wissensstandes künstliche *Intelligenz* produzieren, nicht aber, wie sich künstliches *Selbstbewusstsein* produzieren lässt. Wie immer brechen wir da ab, wo es besonders interessant, wo es geradezu spannend wird. Den Wanderer ergreift eine Spannung, die durch die Dunkelheit und Ungewissheit erzeugt wird, die über dem Gelände liegt, das er betritt,

falls er die Wanderung fortsetzt. Wir halten uns an den wittgensteinschen Imperativ: *"Wovon man nicht sprechen kann, darüber muss man schweigen."*

Das Ziel des Buches ist erreicht. Der letzte Teil bringt Ergänzungen, die zum einen in die Tiefe technischer Details gehen und zum anderen den eigentlichen Themenkreis des Buches überschreiten. Die Kapitel 19 und 20 richten sich an Leser, die etwas mehr über die technischen Schwierigkeiten erfahren möchten, die auf dem Weg zum gegenwärtigen Stand der Computertechnik und der künstlichen Intelligenz überwunden werden mussten. In Kapitel 21 wird ein Blick auf ein Gebiet geworfen, das die Brücke von der traditionellen Informatik zur *technischen Neuroinformatik*, vom Prozessorcomputer zum *Neurocomputer* schlägt. Im Zentrum steht der Begriff der Komplexität. Es werden Methoden erläutert, die es erlauben, auch solche Systeme und Prozesse zu simulieren, die infolge ihrer Komplexität nicht durchschaubar sind. In Kap.21.4 werden einige in Teil 3 offen gebliebene Fragen hinsichtlich der Simulierbarkeit des menschlichen Denkens noch einmal einer kritischen Analyse unterzogen. Dabei wird an Kap.17.3 angeknüpft.

Diejenigen Leser, welche den langen Weg bis zu diesem Punkt mitgegangen sind, sich aber für die genannten vertiefenden bzw. erweiternden Themen weniger interessieren, können die Kapitel 19 bis 21.3 überspringen. Das Schlusswort enthält einige persönliche Meinungsäußerungen des Autors zur Bedeutung der Informatik und der künstlichen Intelligenz für die Zukunft der menschlichen Gesellschaft.

Teil 4

Vertiefende Ergänzungen

19 Ergänzungen zum Problemlösungsweg der Rechentechnik

Zusammenfassung

Hand in Hand mit der in Teil 3 dargestellten Kalkülisierung und Algorithmierung immer neuer Bereiche des menschlichen Denkens vollzog sich eine ständige Perfektionierung der Hard- und Software. Das betraf in erster Linie die Erhöhung der Arbeitsgeschwindigkeit und der Zuverlässigkeit des Computers, die Erweiterung des vom Computer bearbeitbaren Datenmaterials und die Einfachheit der Nutzung des Computers. Dabei schälten sich drei Hauptproblembereiche und entsprechende Spezialdisziplinen heraus: *Rechnerarchitektur*, *Betriebssysteme* und *Programmiersprachen*.

Die Arbeitsgeschwindigkeit wurde u.a. durch Optimierung des Befehlssatzes der CPU, Beschleunigung der Datenbereitstellung und Parallelisierung in Form von *Pipelining*, *Vektorverarbeitung* und *Array*-Anordnungen erhöht (ALU-Array, systolisches Array). Das Hantieren mit großen Datenmengen bei relativ schnellem Zugriff wurde durch Aufbau einer Hierarchie von Speichern zunehmender Speicherkapazität ermöglicht.

Die Abarbeitung der sprachlichen Operatoren der Softwarehierarchie wird durch das Betriebssystem organisiert. Das *Betriebssystem* eines Computers ist die Gesamtheit aller Organisationsprogramme und aller Programme für die Steuerung der peripheren Geräte. Ein *Organisationsprogramm* ist eine Vorschrift für die Zuweisung eines Betriebsmittels an eine Anwendungsoperationsausführung, an einen sog. Anwendungsprozess. Ein *Prozess* stellt aus der Sicht des Nutzers die Ausführung einer Operation durch den Prozessor, aus der Sicht des Systemprogrammierers eine Folge von CPU- und Speicherzuständen dar.

Die Organisationsprogramme haben jeden Prozess rechtzeitig mit den notwendigen *Betriebsmitteln* (Hardwarekomponenten, Programmen, Daten) zu versorgen unter Berücksichtigung verschiedener Vorrang- und Optimierungskriterien. Eine wichtige Aufgabe ist die Verhinderung von Störungen und die Lösung von Konflikten zwischen Prozessen, die ein und dieselben Betriebsmittel verwenden, z.B. dieselben Daten oder dieselben Operatoren. Im Falle des Einprozessorrechners sind sämtliche Prozesse auf die Dienste des einzigen Prozessors angewiesen. Um zu gewährleisten, dass Prozesse sich nicht gegenseitig stören, ist es zweckmäßig, jeden Ruf eines Anwendungs- oder Systemprogramms von einer einzigen Zentrale, dem *Betriebssystemkern* überwachen zu lassen.

Oberhalb der Prozessorebene werden reale Kompositoperatoren nicht mehr als KR-Netze (Netze aus Kombinationsschaltungen und Registern), sondern als *PS-Netze* (Netze aus Prozessoren und Speichern) komponiert. Die Organisation von Prozessen in PS-Netzen, z.B. in Mehrprozessorrechnern oder in Rechnernetzen, kann die

Installation eines übergeordneten Betriebssystems erforderlich machen. PS-Netze werden auch als *verteilte Systeme* bezeichnet.

19.1 Vorbemerkung

Die unkonventionelle Darstellung der Informatik in den Teilen 1 bis 3 kann manchen Leser enttäuscht haben, weil er nicht das gefunden hat, was er erwartet hatte, eine Einführung in die *Praxis* der Computertechnik, in die Programmierung und Nutzung des Computers, in den Umgang mit der Hard- und Software, mit Betriebs- und Nutzersystemen. Ein solcher Leser wird auf die Lektüre dieses Buches hoffentlich nicht viel Zeit verwendet haben, sondern in der fast unübersehbaren Menge einschlägiger Informatikbücher das Richtige gefunden haben[1]

Das Material, das den Inhalt konventioneller Informatikbücher ausmacht, spielt hier die Rolle einer *Ergänzung*. Ergänzt wird jedoch nicht ein "Extrakt praktischen Wissens", sondern eine Aufzählung *praktischer* Probleme, die auf dem Wege zum gegenwärtigen Stand der Computertechnik gelöst werden mussten, sowie einige Ideen ihrer Lösung. Im Vergleich zu den Problemen der vorangehenden Kapitel sind die nun zu behandelnden Probleme von weniger grundsätzlicher Bedeutung und ihre Lösungen haben einen Zug von Zufälligkeit, sie hätten oft auch anders gelöst werden können, und viele von ihnen werden in Zukunft sicher neu gelöst werden. Die ständige Weiterentwicklung von Hardware und Software bestätigt diese Erwartung.

Welche Lösungen sich durchsetzen, ist in der Regel mehr das Ergebnis ökonomischer als wissenschaftlicher Argumente und Entscheidungen. Der Marktführer hat das Sagen. Die Forderung nach wirtschaftlicher Effizienz sorgt dafür, dass sich die Entwicklung der Produkte, die auf dem Markt angeboten werden, in relativ kleinen Schritten vollzieht, denn das Vorhandene muss ökonomisch ausgeschöpft werden. Dabei ist nicht wichtig, was die Wissenschaft, sondern was der Konkurrent anbietet.

Man kann einwenden, dass sich die Entwicklung beispielsweise der Multimediatechnik und die kommunikative Vernetzung der Welt ganz und gar *nicht* in kleinen Schritten vollzieht. Das ist sicherlich richtig. Aber derartige *qualitativen* Sprünge sind die Folge einer langsamen *quantitativen* Entwicklung, nämlich der schrittweisen Steigerung der Arbeitsgeschwindigkeit des kleinen Kobolds, des Prozessors, und eine schrittweise Verkleinerung seiner Abmaße und der Abmaße seines Arbeitsspeichers.

In dieser Hinsicht, d.h. hinsichtlich Taktfrequenz und Miniaturisierung, ist die Informatik auf Kooperation mit anderen Zweigen der Wissenschaft und Technik angewiesen. Es geht letzten Endes um die Herstellung von elektronischen Bauele-

---

1 Einige neuere Lehrbücher seinen genannt: [Appelrath 98], [Broy 98], [Goos 98] [Rembold 99], [Balzert 99], [Gumm 00], [Ernst 00].

menten mit minimalen Abmaßen, minimalem Energieverbrauch, maximaler Bandbreite und ausreichend niedrigem Rauschen. Der Zusammenhang zwischen Miniaturisierung, Bandbreite und Taktfrequenz war in Kap.11.2 [11.3] skizziert worden. Es gibt eine nicht zu überschreitende Grenze hinsichtlich der Taktfrequenz, die durch die Bewegungsgeschwindigkeit der Ladungsträger in Halbleitern gegeben ist, die erheblich unter der Lichtgeschwindigkeit liegt, sowie durch die minimale, noch funktionierende Dicke von *pn*- bzw. *np*-Übergängen in Halbleitern.

Um dieser Grenze näher zu kommen, bedarf es der Zuarbeit der Physiker und Chemiker. Darauf gehen wir nicht ein. Uns interessieren die Möglichkeiten der *Informatiker*, die Leistungsfähigkeit des Computers zu steigern und zwar nicht nur die Leistungsfähigkeit der Hardware, sondern die des Computers samt seiner Software. Wir werden die folgenden drei Bereiche betrachten

- Rechnerarchitektur,
- Betriebssysteme,
- Programmiersprachen,

die eng miteinander verflochten sind. Hinter diesen drei Stichwörtern verbirgt sich "ein weites Feld", eine unübersehbare Menge technischer Details. Wir erleichtern uns die Aufgabe dadurch, dass wir die genannten Bereiche relativ isoliert voneinander betrachten werden, obwohl ihre organische Einheit dadurch ziemlich rücksichtslos zerschnitten wird. Die vielfältigen wechselseitigen Abhängigkeiten werden weitgehend unterschlagen. Zudem werden wir die einzelnen Gebiete unter ganz bestimmten Aspekten, d.h. einseitig betrachten.

Abweichend von dem bisher befolgten Grundsatz, Fachbegriffe nach Möglichkeit mit deutschen Wörtern zu benennen, werden wir uns im Weiteren oft der gängigen englischen Bezeichnungen bedienen. Wir beginnen mit der Rechnerarchitektur.

19.2 Hardwarearchitektur unterhalb der Prozessorebene

19.2.1 Befehlssatz und Speicherhierarchie

Der Aufbau eines Computers von der Ebene der booleschen Operatoren bis zur Maschinenebene wird üblicherweise als **Rechnerarchitektur** bezeichnet. Dabei handelt es sich um die Architektur der Hardware (Firmware eingeschlossen), also um die Struktur einer Hierarchie realer Operatoren. Zuweilen wird das Wort auch in einem weiteren Sinne verwendet und schließt die Architektur oberhalb der Maschinenebene ein. Dabei kann es sich sowohl um Hardware als auch um Software handeln, also um die Struktur einer Hierarchie aus realen und sprachlichen Operatoren. Gegenstand dieses bzw. des nächsten Kapitels ist die Hardwarearchitektur *unterhalb* bzw. *oberhalb* der *Prozessorebene* ("unterhalb" und "oberhalb" im Sinne der Operatorenkomponierung "von unten nach oben").

In diesem Kapitel werden Weiterentwicklungen der Architektur von Rechnern betrachtet, die einen einzigen Prozessor besitzen. Die Entwicklungen gehen in

verschiedene Richtungen und können sich auf die Prozessorsprache, die im Falle eines Einprozessorrechners mit der Maschinensprache identisch ist, auswirken. Im nachfolgenden Kapitel 19.3 wird die Architektur von Computern mit mehreren Prozessoren behandelt, deren Architekturen und Prozessorsprachen als gegeben angenommen werden. Die Prozessoren stellen die Bausteinoperatoren der **Prozessorebene** dar. Untersucht wird die Architektur zwischen der *Prozessorebene* und der *Maschinenebene*, auf der die Maschinensprache definiert ist.

Zunächst interessieren uns, wie gesagt, mögliche Verbesserungen des *Ein*prozessorrechners von Bild 13.7. In Anlehnung an die einschlägige Literatur wird der Prozessor im Weiteren auch als **CPU** (Central Processing Unit) bezeichnet. Der Begriff der CPU ist jedoch allgemeiner; eine CPU kann aus mehrere Prozessoren bestehen. Wir gehen zunächst davon aus, dass die CPU nur einen einzigen Prozessor enthält.

Wir werden die Architektur eines Computers ausschließlich unter dem Gesichtspunkt der Arbeitsgeschwindigkeit betrachten. Andere Parameter wie beispielsweise Volumen und Kosten bleiben unberücksichtigt. *Geschwindigkeit* ist zwar nicht identisch mit *Leistung*, aber beide Größen stehen - ebenso wie beim Auto - in enger Beziehung zueinander, und in der Regel wird die Leistung eines Computers durch eine Geschwindigkeitsangabe charakterisiert. Dazu bietet sich z.B. die Taktfrequenz der CPU an oder die Anzahl der pro Sekunde ausführbaren Gleitkommaoperationen, gemessen z.B. in MFLOPS (Megaflops, Million FLoating point Operations Per Second). Eine andere Möglichkeit wäre, die Leistung eines Computers durch die Anzahl der Befehle zu messen, die er im Mittel pro Sekunde während der Ausführung eines Programms abarbeitet, also eine Folge *unterschiedlicher* Operationen. Die so gemessene Leistung kann allerdings erheblich vom Charakter des Programms (wissenschaftliche Aufgabe, Textverarbeitung, Bankcomputer u.ä.m.), von der verwendeten Programmiersprache, von der Speicherorganisation, vom Betriebssystem und von weiteren Faktoren abhängen.

Aus der Sicht des Nutzers ist es nicht unbedingt nur die Geschwindigkeit des Computers, die seine *Leistungsfähigkeit als Assistent des Nutzers* bestimmt. Oft steht die *Nutzerfreundlichkeit* an erster Stelle. Es können auch mehrere Nutzer sein, die gleichzeitig "freundlich bedient" sein wollen. Auf derartige Wünsche kommen wir später zurück. Im Augenblick konzentrieren uns auf drei Möglichkeiten der Leistungssteigerung, die bei den Bemühungen um eine "schnelle Architektur" im Vordergrund standen und stehen:

1. Optimierung des Befehlssatzes der CPU
2. Beschleunigung der Datenbereitstellung,
3. Parallelisierung des Verarbeitungsprozesses.

Ganz allgemein werden Maßnahmen, die zur Erreichung einer Leistungserhöhung in erster Linie in Frage kommen, von einer (nicht auf allen Gebieten des Lebens gültigen) Gesetzmäßigkeit diktiert: *Je kleiner desto schneller*. Sie gilt fast durchweg für die beteiligten Hardwarebausteine (Operatoren, Speicher). Hinsichtlich der ele-

mentaren elektronischen Bausteine, der Transistoren, ist die Gültigkeit der Regel offensichtlich: je dünner die *p*- bzw. *n*-Schicht, umso kürzer die Laufzeiten der Ladungsträger durch die Schicht und umso höher die maximale Taktfrequenz. Für die darüber liegenden Schichten gilt die Regel zum einen aus dem analogen Grunde: je kleiner die Baueinheiten, desto kleiner die Wege, desto kürzer die Übertragungszeiten. Zum anderen gilt die genannte Gesetzmäßigkeit infolge des wachsenden Steueraufwandes bei zunehmender Kompositgröße. Dieses Phänomen ist sicher jedem aus eigener Erfahrung mit Verwaltungshierarchien gut bekannt.

Andrerseits sind die Computerentwerfer bestrebt, die Leistungsfähigkeit nicht nur hinsichtlich der Geschwindigkeit zu verbessern, sondern beispielsweise auch hinsichtlich der Maschinensprache und der Speichergröße, um Nutzerwünsche besser zu befriedigen. Beides führt aber unweigerlich zu Zeitverlusten infolge steigenden Steueraufwandes. Die Suche nach dem optimalen Kompromiss war und ist bestimmend für die Entwicklung neuer Architekturprinzipien. Der Kompromiss, der sich gegenwärtig durchgesetzt hat, soll anhand von Bild 19.1 erläutert werden.

Gegenüber Bild 13.7 zeigt Bild 19.1 einige Veränderungen. Die Details der RALU sind fortgelassen. Die Steuerung der RALU hat das Matrixsteuerwerk übernommen (siehe Kap.13.5.5), und für die Speicherung der Bitketten, die von der ALU transformiert werden können, sind weitere Speichereinheiten hinzugekommen, die gemeinsam eine "Speicherhierarchie" bilden (in Bild 19.1 von der ALU aus *abwärts* dargestellt). Den Teil von Bild 19.1 oberhalb der strichpunktierten Linie nennen wir **zentrale Hardware**, den Teil unterhalb der Linie **periphere Hardware**. Zur peripheren Hardware, auch kurz **Peripherie**[2] genannt, gehört alles, was nicht zur zentralen Hardware (oberhalb der strichpunktierten Linie) gehört, also auch Festplatten und sog. **externe Geräte**, die "extern" an den Computer angeschlossen werden, u.a. die (nicht eingezeichneten) **E/A-Geräte** wie Bildschirm, Tastatur, Drucker u.ä.m. Speicher mit herausnehmbarem Speichermedium (Scheibe, Band) stellen de facto E/A-Geräte für große Datenmengen dar und werden zuweilen auch als solche bezeichnet. Anhand des Bildes 19.1 sollen die ersten beiden der drei oben genannten Möglichkeiten der Leistungssteigerung erläutert werden.

Befehlssatz. Abgesehen von der Variabilität der ALU ist der Befehlssatz der CPU durch das Matrixsteuerwerk festgelegt. Man könnte geneigt sein, das Matrixsteuerwerk sehr umfangreich zu gestalten und eventuell sogar aus mehreren Schichten zu komponieren. Dann würde die Maschinensprache viele Operationen anbieten, wodurch der Programmieraufwand herabgesetzt werden könnte. In dieser Richtung gab es ernsthafte Bemühungen. In den letzten Jahren ist die Entwicklung jedoch in die entgegengesetzte Richtung gegangen, und es hat sich eine relativ sparsame Variante unter der Bezeichnung **RISC**-Prozessor (Reduced Instruction Set Computer) durch-

2 Die Bezeichnung "Peripherie" stammt aus den Zeiten vergangener Rechnergenerationen. Damals wurde das Wort jedoch in einer etwas anderen Bedeutung verwendet.

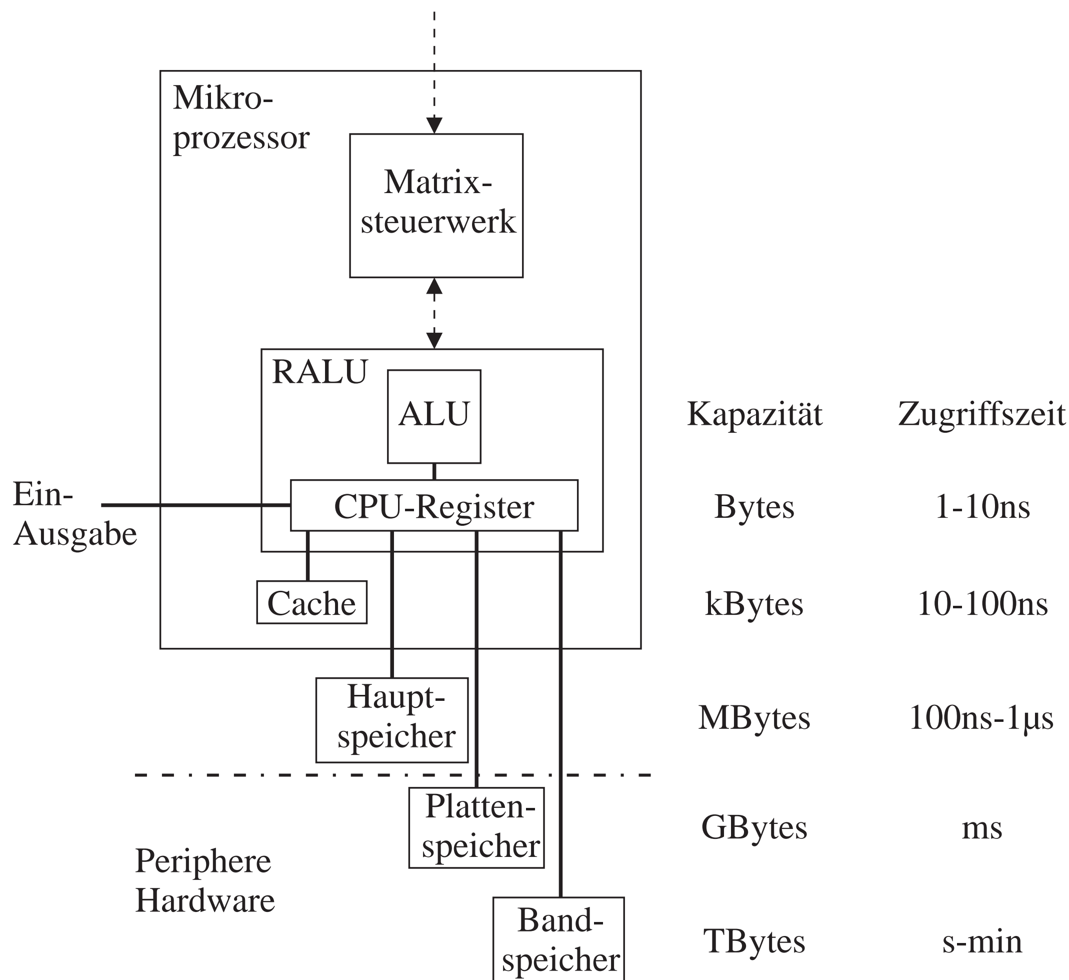


Bild 19.1 Mikroprozessor mit Speicherhierarchie; erweiterte Architektur von Bild 13.7. - - - - Steuerleitungen; ▬▬ Datenleitungen. Auf der rechten Seite sind die Speicherkapazitäten und Zugriffszeiten angegeben. Die skalierenden Buchstaben vor den Maßeinheiten sind zu lesen als

k	Kilo	10^3	m	milli	10^{-3}
M	Mega	10^6	µ	Mikro	10^{-6}
G	Giga	10^9	n	Nano	10^{-9}
T	Tera	10^{12}.			

gesetzt. Die Erfahrung zeigt, dass die höhere Arbeitsgeschwindigkeit des RISC-Prozessors den höheren Softwareaufwand mehr als aufwiegt[3].

Speicherhierarchie. Alle Befehle und Daten, mit denen der Prozessor bei der Abarbeitung eines Programms hantiert, müssen den von-neumannschen Flaschenhals, also die Strecke HK-K in Bild 13.7 passieren. Jedes Zwischenresultat wird in den HS transportiert und von da wieder in ein Register der CPU geholt.

3 Ausführlich ist die RISC-Architektur und der RISC-Befehlssatz in [Hennessy 94] behandelt, wo der Leser eine umfassende Einführung in das Gesamtgebiet der Rechnerarchitektur findet.

Dermaßen umständlich verfährt in analogen Alltagssituationen kein vernünftiger Mensch, keine Hausfrau, kein Handwerker, kein Ingenieur. Vielmehr sucht man sich zuerst alles, was ständig zur Hand sein muss, aus der Vorratskammer, dem Geräte- oder Bücherschrank zusammen und legt es unmittelbar am Arbeitsplatz bereit. Auch Zwischenprodukte legt man nicht erst weit weg, sondern behält sie in der Nähe. Eventuell wird sogar mit einer ganzen Hierarchie von Ablagen (Speichern) gearbeitet, beispielsweise Küchentisch - Küchenschrank - Vorratskammer - Keller. Man könnte noch den Supermarkt als "externen Speicher" hinzufügen.

Nach diesem Muster verfährt die CPU von Bild 19.1. Die dargestellte Architektur enthält eine Speicherhierarchie[4] mit den fünf Ebenen:

- CPU-Register
- Cache
- Hauptspeicher
- Plattenspeicher oder allgemeiner Scheibenspeicher
- Bandspeicher.

Unter der Bezeichnung *Scheibenspeicher* sind alle magnetischen und optischen Speicher (Plattenspeicher, CD-ROMs) zusammengefasst, die mit rotierenden Scheiben (Disketten, CDs, DVDs) als Speichermedium ausgerüstet sind.

Die ALU kann unmittelbar mit einer mehr oder weniger großen Menge von Registern kommunizieren, den sogenannten **CPU-Registern**. Sie stellen eine Erweiterung des Akkumulators (AC) und des Datenregisters (DR) von Bild 13.7 dar. Die CPU-Register können sowohl Silo- (FIFO-)Register als auch Keller- (LIFO-)Register sein.

Der **Cache** ist ein schneller Zwischenspeicher zwischen CPU und Hauptspeicher. Die Computerbauer haben ihn als Reaktion auf die sog. 90/10-Regel entwickelt. Messungen haben gezeigt, dass im Mittel ein Programm 90% seiner Ausführungszeit für 10% des Codes verbraucht. Es wird also ein relativ kleiner Teil eines Programms bedeutend häufiger abgearbeitet als der Rest. Das Messergebnis mag überraschen, wenn auch eine gewisse Konzentration auf einige Programmabschnitte infolge von Zyklen durchaus plausibel ist. Für die Zugriffe auf den Speicher bedeutet die 90/10-Regel, dass sie auf relativ eng begrenzte Bereiche lokalisiert sind. Diese **Lokalität des Zugriffs** ist sowohl für Befehle als auch für Daten beobachtet worden, d.h. auf einen kleinen Teil aller Daten, mit denen ein Programm arbeitet, wird bedeutend häufiger zugegriffen als auf den großen Rest. Das regte die Ingenieure zum Entwurf eines schnellen Zwischenspeichers an. Das Ergebnis war der sogenannte *Cache*. Er dient der Aufbewahrung oft benutzter Befehle und Daten. Dabei kommt häufig nicht das von-neumannsche, sondern das *Havardprinzip* zur Anwendung, d.h. für Befehle und Daten werden zwei getrennte Speicherbereiche vorgesehen.

---

4 Das Wort "Hierarchie" hat hier nicht die Bedeutung von Komponierungshierarchie.

Nach der Regel “je kleiner desto schneller” wird die Kapazität des Cache möglichst klein gehalten. Damit er nicht überläuft, wird alles, was längere Zeit unbenutzt “herumliegt” in den Hauptspeicher eingeordnet (ebenso wie in der Küche oder in der Werkstatt). Zusätzlich kann der Zugriff durch Techniken beschleunigt werden, die *assoziativ* genannt werden, obwohl sie i.Allg. *nicht* nach dem assoziativen Zugriffsprinzip im eigentlichen Sinne des Wortes erfolgen, also *nicht* über den Speicherplatzinhalt selber (genauer über einen bestimmten Teil der abgespeicherten Bitkette).

Ob die CPU das gerade Benötigte im Cache findet, ist mehr oder weniger Glücksache; man spricht von *Trefferwahrscheinlichkeit*. Die Anschaulichkeit dieses Wortes wollen wir ausnutzen, um die Arbeitsweise der Speicherhierarchie anhand folgenden Bildes zu verdeutlichen. Wir blicken von oben, von der ALU her, auf die Hierarchie. Die einzelnen Speicher stellen wir uns als Scheiben vor, die konzentrisch aufeinander liegen und nach oben (zur ALU) hin kleiner werden entsprechend der jeweiligen Speicherkapazität. Färben wir nun den obersten Kreis schwarz und die sichtbaren Ringe der übrigen Speicher weiß, erblicken wir eine Schießscheibe. Das Schwarze sind die CPU-Register. Der nächste Ring ist der Cache und so weiter.

Wenn die CPU ein Datum, das sie gerade benötigt in den CPU-Registern findet, hat sie ins Schwarze getroffen. Befindet sich das Datum im Cache, hat sie auch noch Glück gehabt. Ist das Datum auch dort nicht auffindbar, muss es im Hauptspeicher gesucht werden, was bereits merklich länger dauert. Ist es auch da nicht auffindbar, muss auf die peripheren Speicher zugegriffen werden, was die Programmabarbeitung erheblich aufhält.

Der Einbeziehung eines peripheren Speichers in die Arbeit der CPU kann beispielsweise erforderlich werden, wenn mit einer großen Datenbank gearbeitet wird oder wenn das Programm, das die CPU ausführt, so umfangreich ist, dass es im Hauptspeicher keinen Platz hat. Das kann dazu führen, dass zwischen Plattenspeicher und Hauptspeicher relativ häufig größere Programmabschnitte hin und hertransportiert werden müssen. Um dieses “*Umschaufeln*” zu beschleunigen, geht man folgendermaßen vor. Der Inhalt eines Teils des Plattenspeichers wird in Abschnitte einheitlicher Größe, **Seiten** genannt, segmentiert. Eine solche Seite ist die Transporteinheit, und für jeweils eine Seite wird im Hauptspeicher Platz zur Verfügung gestellt. Der Zugriff zu den Seiten erfolgt über Tabellen (Inhaltsverzeichnisse), die dem Beginn der einzelnen Programmsegmente die Anfangsadresse der betreffenden Seite im Hauptspeicher bzw. im peripheren Speicher zuordnen.

Die Methode des *seiten*weisen Transportierens und Speicherns wird **Paging** genannt. Es erhebt sich die Frage, wer das Paging organisiert. Wer legt die Tabellen an? Wer sucht in den Tabellen die aktuelle Adresse? Wer organisiert den Transport? Offenbar muss ein spezielles *Paging-Programm* geschrieben werden, das die Organisation vornimmt, das also bei seiner Abarbeitung die Steuersignale generiert, die den Datentransport zwischen peripherem Speicher und Hauptseicher steuern. Der Prozessor, der das Pagingprogramm ausführt, spielt dabei die Rolle des realen

Steueroperators. Der Nutzer sollte von den organisatorischen Maßnahmen der Datenbereitstellung möglichst wenig merken. Vielmehr sollte er den Eindruck haben, als stünde ihm ein sehr großer Arbeitsspeicher zur Verfügung. In diesem Sinne spricht man von **virtuellem Speicher**.

Zur Speicherperipherie eines PC gehören außer der Festplatte i.Allg. weitere Scheibenspeicher wie Disketten- und CD-Speicher, die gleichzeitig als E/A-Geräte dienen. Eventuell kann auch ein Kassettenspeicher angeschlossen werden.

Damit beenden wir unsere Überlegungen zu den ersten beiden oben genannten Möglichkeiten der Leistungssteigerung von Computern und wenden uns der dritten Möglichkeit zu, der Parallelisierung von Verarbeitungsprozessen.

19.2.2 Pipelining und Vektorrechner.

Ein Nachteil des Einprozessorrechners besteht darin, dass die Maschinenoperationen (die Operationen der Maschinenebene) *sequenziell* ausgeführt werden müssen, auch dann, wenn eine *parallele* (gleichzeitige, simultane) Ausführung möglich wäre, falls mehrere reale Operatoren (mehrere Prozessoren) zur Verfügung stünden. Aber selbst wenn sie zur Verfügung stehen, ist eine parallele Ausführung zweier oder mehrerer Operationen nur dann möglich, wenn die Operationen voneinander kausal unabhängig sind, d.h. wenn zwischen ihnen keine Operandenübergaben stattfinden. Solche Operationen werden **nebenläufig** genannt. Nebenläufige Operationen können parallel ausgeführt werden, wenn die notwendigen realen Operatoren zur Verfügung stehen. Das spart Zeit, erfordert aber zusätzliche organisatorische Maßnahmen.

Im Zusammenhang mit den Operatorennetzen der Bilder 8.1 und 8.3 wurde bereits festgestellt, dass zum Zwecke der Zeiteinsparung Operationen in den Ästen einer starren Masche parallel ausgeführt werden können (im Gegensatz zur Alternativmasche), z.B. das Potenzieren und das Berechnen der Sinusfunktion in Bild 8.1 oder das Polieren und Bohren in Bild 8.3. In beiden Fällen liegt ein Paar *nebenläufiger* Operationen vor.

Eine andere Art von Parallelität bietet sich an, wenn ein Kompositoperator, der aus einer *Kette* von Bausteinoperatoren besteht, eine *Folge* von Operanden bearbeitet. Dann lässt sich die Operationsausführung oft so organisieren, dass die Operanden unmittelbar hintereinander die Kette durchlaufen, sodass sich mehrere Operanden gleichzeitig im *Komposit*operator befinden, ähnlich wie Autos, die auf einem Fließband montiert werden. Die Informatiker sprechen allerdings nicht von Fließband, sondern von **Pipeline** und von *Pipelineverarbeitung* oder **Pipelining**. Jeder einzelne Bausteinoperator führt seine Operationen *sequenziell* aus, während der Kompositoperator seine Operationen *zeitlich überlappend*, also stückweise *parallel* ausführt.

Die Computerbauer sind auf die Idee gekommen, das Pipeline-Prinzip auf die Operationsausführung (sprich: Befehlsausführung) durch einen Einprozessorcomputer anzuwenden. Das mag unsinnig erscheinen. Doch erinnere man sich an den Kommentar zu Bild 13.8., wonach während einer Befehlsausführung die Berechnung

der Adresse des nächsten Befehls erfolgen kann, zumindest dann, wenn die neue Befehlsadresse durch Inkrementieren der alten bestimmt wird. Das ist möglich, weil der *Prozess* des Inkrementierens seine *privaten Betriebsmittel besitzt*.[5] Damit der Inkrementierungsprozess *laufen* kann, müssen ihm folgende *Betriebsmittel zugeteilt* sein: der Operator INC, der Befehlszähler (BZ), die Sammelweiche S2 und die Leitungen der Inkrementierungsschleife. Der Befehlszähler spielt während der Inkrementierung die Rolle eines *privaten Speichers*. Kein anderer Transfer mit Ausnahme des ersten benötigt die genannten Betriebsmittel.

Die Berechnung der Adresse des nächsten Befehls gleichzeitig mit der Ausführung des vorangehenden Befehls ist im Grunde genommen ein ganz einfacher Fall von Pipelining. Weitere Überlappungen der Ausführungen zweier aufeinander folgender Befehle ist nach Bild 13.8 nicht erlaubt. Der Flaschenhals HK-K verbietet sie. Diese Beschränkungen, die durch die Architektur von Bild 13.7 dem Pipelining auferlegt werden, ist für die Architektur von Bild 19.1 teilweise aufgehoben. Die wesentliche Ursache hierfür liegt in dem Umstand, dass auf den Cache und die CPU-Register *parallel* zugegriffen werden kann, denn die CPU-Register und der Cache liegen *diesseits* des Flaschenhalses (aus der Sicht der CPU), sodass die Einschränkungen, die der Flaschenhals dem Pipelining auferlegt, zum Teil entfallen.

Das Pipelining ist im Laufe der Jahre ständig vervollkommnet worden. Dabei wuchs die Anzahl der Befehle, die überlappend ausgeführt werden, die sog. **Pipelinetiefe**, immer weiter an. Wenn eine Pipeline beispielsweise “fünf Befehle *tief*” ist, muss jeder Maschinenbefehl in fünf Segmente (Takte oder Taktgruppen) unterteilt sein, die betriebsmittelmäßig voneinander unabhängig sind. Dafür besteht in obigem Beispiel keine Möglichkeit. Pipelining wird erst dann effektiv, wenn die Ausführung eines Befehls bedeutend mehr Registertransfers beinhaltet, als die Addition in Bild 13.8. Das ist beispielsweise der Fall, wenn **Adressrechnung** notwendig ist, d.h. wenn eine Datenadresse, die zur Befehlsausführung erforderlich ist, nicht *explizit* im Befehl enthalten sind, sondern berechnet werden muss. Eine Adressrechnung muss z.B. dann ausgeführt werden, wenn die Adressen der Variablen nicht absolut, sondern relativ zur Anfangsadresse des Programms angegeben werden. Das Arbeiten mit **relativen Adressen** ist notwendig, wenn das Programm keine feste Anfangsadresse besitzt, sondern im Hauptspeicher verschoben werden kann oder wenn es sich in einem peripheren Speicher befindet und von dort an einen freien Platz im Hauptspeicher geholt werden muss. Adressrechnungen können bei der Benutzung höherer Programmiersprachen viel Zeit in Anspruch nehmen.

Der Vergleich des Pipelining mit der Fließbandproduktion von Autos hinkt insofern, als bei der Automontage in der Regel ein Fließband viele Autos des gleichen Typs produziert, also, im Gegensatz zum Pipelining, gewissermaßen ständig ein und

5 Wir benutzen hier die Terminologie und Redeweise der Betriebssystemtechnik, um schon jetzt die Bedeutung der kursiv gedruckten Wörter und Wendungen zu verdeutlichen.

denselben Befehl ausführt. Dieser Sonderfall kann aber auch beim Pipelining vorliegen, beispielsweise, wenn zwei Vektoren addiert werden. Dann wird nämlich ständig der gleiche Befehl (Additionsbefehl) an einer Reihe von Operanden ausgeführt. Die Operanden sind Summandenpaare und zwar Paare der sich entsprechenden Komponenten der beiden Vektoren. Beim Rechnen mit Vektoren lässt sich die Rechengeschwindigkeit dadurch erhöhen, dass die Komponenten der zu verarbeitenden Vektoren gemeinsam bereitgestellt werden, indem sie aus dem Speicher, in dem sie aufbewahrt sind, in spezielle **Vektorregister** innerhalb der CPU transportiert werden. Wenn die Vektorregister als Schieberegister organisiert sind, können ihnen die jeweiligen Komponenten direkt (ohne Adressierung) in der richtigen Reihenfolge entnommen werden, die das Pipelining verlangt. Man spricht dann von **Vektor-Pipelining**. Ein Computer mit der Möglichkeit zum Vektor-Pipelining wird **Vektorrechner** genannt. Leistungsfähige Vektorrechner verfügen über mehrere spezialisierte Pipelines, z.B. eine für Festkommaadditionen, eine andere für Gleitkommamultiplikationen und weitere. Dafür bedarf es nicht unbedingt mehrerer Prozessoren.

Natürlich lässt sich die Rechengeschwindigkeit durch Zusammenschalten mehrerer Prozessoren zu einem **Mehrprozessorrechner** weiter erhöhen, auch ohne Pipelining. Die Idee des Mehrprozessorrechners ist im Grunde einfacher und naheliegender als die des Pipelining. Wir kommen auf sie in den Kapiteln 19.3 und 19.5.4 zurück. Hier sei nur erwähnt, dass Computer (auch PCs) relativ häufig einen zweiten Prozessor besitzen, einen sog. **Koprozessor**, der spezielle, zeitaufwendige Operationen, z.B. Gleitkommamultiplikationen durchführt, während der zentrale Prozessor die Programmabarbeitung fortsetzt. Es stehen also zwei ALUs zur Verfügung, sodass in jedem Arbeitstakt zwei Bitketten gleichzeitig transformiert werden können.

Um das zu erreichen, bedarf es aber nicht unbedingt mehrerer Prozessoren. Es ist nämlich durchaus möglich, mehrere ALUs in eine einzige RALU zu integrieren oder mehrer RALUs durch einen einzigen Steueroperator zu steuern. Auf diese Weise ergibt sich neben dem Pipelining ein zweiter Weg der Prozessparallelisierung *innerhalb eines* Prozessors, m.a.W. Parallelisierung *unterhalb der Prozessorebene*. Wenn dagegen mehrere Prozessoren an der parallelen Programmausführung teilnehmen, liegt Prozessparallelisierung *oberhalb der Prozessorebene* vor. In diesem Sinne unterscheiden wir zwischen **Parallelität unterhalb** und **oberhalb der Prozessorebene**.

Wir wollen uns überlegen, welche architektonischen Varianten echter Parallelisierung (nicht Pipeline-Parallelisierung) unterhalb der Maschinenebene denkbar sind. Die Rede ist also von Architekturen, die *eine* CPU mit *mehreren* ALUs enthalten. Zunächst ist festzustellen, dass ein derartiger Computer ein und dasselbe Programm gleichzeitig auf mehrere Eingabeoperanden anwenden kann. Ein Computer, der dazu in der Lage ist, wird auch *SIMD-Computer* genannt. Die Bezeichnung stammt von M.J.Flynn, der alle Computer in vier Klassen eingeteilt hat und zwar in **SISD-, SIMD-, MISD-** und **MIMD-Computer**. Dabei bedeutet

SISD - Single Instruction Stream, Single Data Stream,

SIMD - Single Instruction Stream, Multiple Data Stream,
MISD - Multiple Instruction Stream, Single Data Stream,
MIMD - Multiple Intruction Stream, Multiple Data Stream.

Nach dieser Klassifikation gehört der konventionelle Einprozessorcomputer mit einer ALU (Bild 13.7) zur Klasse der SISD-Computer, denn er kann nicht mehrere Befehle gleichzeitig ausführen, also jeweils nur *eine* Operation an *einem* Operanden bzw. Operandentupel. Der MISD-Computer ergibt sich zwar theoretisch (sozusagen automatisch) aus der Klassifikation nach den angeführten beiden Merkmalen; er ist jedoch kaum von praktischer Bedeutung. Das MIMD-Regime verlangt den Einsatz mehrer Prozessoren und lässt sich folglich nicht durch Parallelisierung *unterhalb* der Prozessorebene realisieren, während das SIMD-Regime sowohl unterhalb als auch oberhalb der Prozessorebene möglich ist. Diesen Fall wollen wir näher betrachten.

19.2.3 ALU-Array. Simulation von Nahwirkungen

Der Einsatz eines SIMD-Computers ist dann sinnvoll, wenn mehrere Daten unabhängig voneinander derselben Operation unterzogen oder von demselben Programm bearbeitet werden, also dann, wenn auch Vektorcomputer zum Einsatz kommen können. Doch ist der SIMD-Computer schneller als der Vektorcomputer dank der echten Parallelverarbeitung. Diese wird *unterhalb* der Maschinenebene dadurch möglich, dass die ALU in Bild 13.7 durch eine ganze Batterie von ALUs, durch ein **ALU-Komposit** ersetzt wird. Im Weitern gehen wir davon aus, dass jede ALU über ihre eigenen Ein- und Ausgaberegister verfügt (DR und AC in Bild 13.7).

Das Wort *ALU-Komposit* ist unüblich. Wir verwenden es, um anzuzeigen, dass von einem Kompositoperator die Rede ist, dass wir also von der algorithmischen Sichtweise zur Operatorennetz-Sichtweise übergewechselt sind. Das ist bei der Untersuchung echter Parallelverarbeitung naheliegend und sinnvoll. Ein ALU-Komposit ist ein Kompositoperator, dessen Bausteinoperatoren ALUs sind, also variable Kombinationsschaltungen.

Die ALUs des Komposits können (aus der Sicht des Hauptspeichers, genauer des Busses) parallel angeordnet werden, sodass ihre Eingaberegister gemeinsam ein *Vektorregister* bilden wie im Falle des Vektorrechners. Die ALUs können untereinander Daten austauschen, ähnlich wie die Stellenaddierer eines Paralleladdierers [13.2], von denen jeder eine Stelle der Summe berechnet. Dort dienten die Querverbindungen zwischen den Stellenaddierern der Weitergabe des Übertrags.

Wir wollen uns eine häufig benutzte Konfiguration etwas genauer ansehen, das *ALU-Gitter*. Die Struktur eines ALU-Gitters ähnelt dem Leitergitter von Bild 12.5, wobei nun aber in jedem Schnittpunkt eine ALU mit ihren Ein/Ausgaberegistern angeordnet ist. Über die Verbindungslinien können zwischen *benachbarten* ALUs in *beiden* Richtungen Operanden übergeben werden. Jede ALU besitzt je einen externen Ein- und Ausgang. Ein solches ALU-Gitter wird auch **ALU-Array** genannt.

ALU-Arrays eignen sich für die Computersimulation von Nahwirkungsphänomenen und zur Lösung partieller Differenzialgleichungen. Da es sich dabei um eine

Computeranwendung von erheblicher praktischer Bedeutung handelt (z.B. für die Wettervorhersage), soll an einigen Beispielen veranschaulicht werden, was mit Nahwirkung gemeint ist und was sich hinter dem Begriff der partiellen Differenzialgleichung verbirgt.

Wir beginnen mit einem Beispiel, das weder mit Informatik noch mit Physik etwas zu tun hat, das aber den Kern des Problems, um das es geht, sehr anschaulich verdeutlicht. Man stelle sich eine Schafherde vor, die sich in Bewegung befindet. Die Bewegung jedes einzelnen Schafes ist in erster Linie durch die Bewegung seiner *unmittelbaren* Nachbarn bestimmt. Wenn man die *Fernwirkungen* vernachlässigt (Rufe des Hirten, Hundegebell, Wind, Erdanziehung u.ä.m.), lässt sich das Verhalten der Individuen durch reine Nahwirkungen (physischen Druck auf den Nachbarn) beschreiben. Die Nahwirkung möge für alle Schafe den gleichen physikalischen und physiologischen Gesetzen gehorchen, abgesehen von den Schafen am Rande der Herde, wo besondere *Randbedingungen* gelten. Die Nahwirkung prägt das *kollektive* Verhalten der Herde. 1

Ganz analog kann man das Verhalten der einzelnen Moleküle eines Gases, einer Flüssigkeit oder eines Festkörpers betrachten, nämlich als *Reaktion* auf das Verhalten der Nachbarmoleküle. Diese *Nahwirkungen* zwischen den Molekülen prägen (zusammen mit Fernwirkungen wie der Wirkung der Gravitation oder eines äußeren elektromagnetischen Feldes) das *kollektive* Verhalten der Moleküle, d.h. des Materials, das aus den Molekülen "komponiert" ist. Beispiele für solche kollektiven Phänomene sind Schwingungen einer Saite oder eines Kristalls, Wellen und Wirbel einer Flüssigkeit, das Wandern von Hoch- und Tiefdruckgebieten in der Atmosphäre, Wärmeleitung durch eine Wand, Diffusion von Molekülen, z.B. von Bor durch Silizium beim Dotieren, und viele andere Erscheinungen. 2

Die Eignung von ALU-Arrays für die Simulation derartiger Phänomene ist augenfällig. Denn die Nahwirkung, d.h. die Wechselwirkung mit der nächsten Umgebung (die "*Wechselwirkungsstruktur*") lässt sich in die *Kommunikationsstruktur* eines ALU-Arrays abbilden. Dazu überzieht man gedanklich das zu simulierende Gebiet mit einem *Koordinatengitter* und legt in jeden Gitterpunkt eine ALU. Jede ALU "reagiert" auf ihre Nachbarn, indem sie ihren Zustand, d.h. die Zustandsvariable z, aus den Zustandsvariablen der Nachbarn berechnet. Wenn alle Elemente des Systems das gleiche Verhalten zeigen, haben alle ALUs ein und dieselbe *Ergibtanweisung* für unterschiedliche Variablenwerte auszuführen. Es liegt also eine typische SIMD-Berechnung vor. Soweit ist die Anwendung eines ALU-Arrays auf ein Nahwirkungsproblem durchaus einleuchtend. Weniger offensichtlich ist der Weg, auf dem sich der Kollektivzustand eines Systems mit Nahwirkung berechnen lässt. Wegen der großen praktischen Bedeutung derartiger Rechnungen für die Steigerung künstlicher Intelligenz soll die Idee der Methode skizziert werden.

In einem kollektiven, statisch oder dynamisch stabilen Zustand müssen die "individuellen" Zustände (die z-Werte) der Komponenten des Systems miteinander konsistent sein, das heißt - mathematisch ausgedrückt -, zwischen ihnen müssen

bestimmete Relationen erfüllt sein. Wenn diese sich in Form einer relationalen Gleichung (vgl. Kap.8.3 [8.20]) für z ausdrücken lassen, sprechen wir von **Zustandsgleichung**. Die Zustandsgleichung ist in eine Ergibtgleichung umzuformen und als Ergibtanweisung zu formulieren, die jede ALU auszuführen hat. Wir nehmen im Weiteren an, dass eine Zustandsgleichung existiert und dass sie sich in eine Ergibtgleichung überführen lässt. Man beachte, dass die Ergibtgleichung zwar die "Lösung" der Zustandsgleichung, aber noch nicht die gesuchte Lösung des Problems darstellt, denn mit ihr ist noch nicht die z-Verteilung (die Gesamtheit der individuellen z-Werte) gefunden. Dafür bedarf es numerischer Rechungen [15.8]. Um sie durchführen zu können, muss eine z-Verteilung vorliegen. Aber es existiert keine. Darum muss ein "Trick" angewendet werden. Er besteht in Folgendem.

Man gibt eine "erfundene" Verteilung vor, die "vernünftig aussieht". Mit den Werten dieser Verteilung berechnet man für jeden Gitterpunkt einen "neuen" z-Wert. Wäre die erfundene Lösung zufällig die gesuchte, würden sich die neuen z-Werte nicht von den alten unterscheiden. Das aber ist kaum zu erwarten. Ersetzt man alle alten Werte durch die neuen, ergibt sich eine neue Verteilung, die, wenn man Glück hat, eine bessere Annäherung an die richtigen Werte, d.h. an die gesuchte *Lösung* darstellt. Ob das der Fall ist, erkennt man durch Iteration. Wenn bei wiederholter Neuberechnung die Korrekturen an den z-Werten ständig kleiner werden, konvergieren die schrittweise berechneten Verteilungen gegen die Lösung der Zustandsgleichung.

Es gibt eine große Klasse praktischer Nahwirkungsprobleme, die durch sogenannte lineare partielle Differenzialgleichungen zweiter Ordnung beschrieben werden. Solche Gleichungen lassen sich durch Diskretisierung der Koordinaten in *Differenzengleichungen zweiter Ordnung* überführen, die nur Differenzen enthalten und zwar Differenzen benachbarter z-Werte (Differenzen erster Ordnung) und Differenzen dieser Differenzen (Differenzen zweiter Ordnung). Durch Diskretisierung wird aus dem Koordinatengitter ein Punktgitter. In jeden Punkt wird eine ALU platziert. Die gemeinsame Ergibtanweisung für alle ALUs ergibt sich aus der Differenzengleichung.

Von der Ergibtgleichung gelangt man zurück zur Differenzialgleichung, indem man die Abstände zwischen den Gitterpunkten gedanklich beliebig klein werden lässt. Wenn die Abstände gegen Null gehen, müssen die Differenzen erster und zweiter Ordnung durch sogenannte *partielle Differenzialquotienten* erster bzw. zweiter Ordnung ersetzt werden.

Wir hatten bereits in Kap.4.2 mit Differenzialquotienten zu tun, dort aber nicht mit partiellen, sondern mit gewöhnlichen. Die gesuchten Funktionen hingen von einer einzigen Veränderlichen, der Zeit ab. Jetzt hängt die gesuchte Funktion nicht von der Zeit, sondern von zwei räumlichen Veränderlichen (Variablen), den Koordinaten x und y ab und an die Stelle gewöhnlicher treten partielle Differenzialquotienten. *Ein **partieller Differenzialquotient** einer Funktion von mehreren Veränderlichen ist die Ableitung der Funktion nach **einer** der Veränderlichen* (die anderen

werden als Konstante behandelt). Der partielle erste Differenzialquotient (die erste partielle Ableitung) einer Funktion $z(x,y)$ in Richtung der x-Kordinate wird als $\partial z/\partial x$ notiert. *Eine Gleichung, die partielle Differenzialquotienten enthält, heißt* ***partielle Differenzialgleichung***. Sie fasst alle Relationen, die in den Gitterpunkten erfüllt sein müssen, in infinitesimaler Form in einer einzigen Gleichung zusammen.

Partielle Differenzialgleichungen spielen in der theoretischen Physik eine herausragende Rolle. Dabei stellt z den lokalen Wert einer "Feldgröße" dar, d.h. einer Größe, deren stetiger räumlicher Werteverlauf durch eine Differenzialgleichung beschrieben wird, *Feldgleichung* genannt. Die Rolle einer Feldgröße kann beispielsweise die (lokale) Geschwindigkeit eines Strömungsfeldes, die Temperatur eines Temperaturfeldes, die Feldstärke eines elektrischen Feldes oder die Konzentration eines Konzentrationsfeldes ("Diffusionsfeldes") spielen.

Der historischen Korrektheit halber sei ergänzt, dass zunächst *analytische* und erst später *numerische*, iterative Lösungsmethoden entwickelt wurden. Partielle Differenzialgleichungen lassen sich, ebenso wie gewöhnliche Differenzialgleichungen, nicht immer analytisch lösen, eine numerische Lösung kann jedoch stets gefunden werden, vorausgesetzt, die Gleichung hat überhaupt eine Lösung. Allein aus dieser Tatsache ergibt sich die Bedeutung der Rechentechnik für die theoretische Physik. Ob dabei ein SIMD-Computer zum Einsatz kommen muss, ist eine Frage des Rechenaufwandes. Wenn ein Einprozessorrechner nicht ausreicht, werden heutzutage vorwiegend Vektorrechner verwendet, z.B. die *Cray*, das ist ein nach seinem Erbauer benannter, sehr leistungsfähiger Vektorrechner. Die Cray ist von ihrer Idee her *kein* SIMD-Computer, sondern ein Pipeline-Rechner, der die Komponenten eines oder mehrerer Vektoren, die in Vektorregistern gespeichert sind, zeitlich überlappend verarbeitet.

Zur Anregung der Phantasie stelle sich der Leser folgende, ganz andere Möglichkeit vor, die Arbeitsweise eines ALU-Array zu organisieren. Man betrachte eine Zeile eines ALU-Array, also die durch einen waagerechten Leiter miteinander verbundenen ALUs einer Arrayzeile, als Fließband (Pipeline), das von links eintretende Operanden nach rechts weitergibt, wobei die Operanden (Bitketten) durch die ALUs schrittweise (taktweise) transformiert werden. Das Array besteht also aus mehreren parallelen Fließbändern.

Nun lassen wir die Möglichkeit zu, dass die ALUs eines Fließbandes über die senkrechten Leitungen mit ihren oberen und unteren Nachbarn (den entsprechenden ALUs der benachbarten Fließbänder) Daten austauschen können, sodass sich ein recht komplizierter Datenfluss entwickeln kann. Damit der Gesamtprozess determiniert bleibt, muss er getaktet verlaufen, d.h. alle Datenübergaben müssen synchron stattfinden. Ein Array, das in einem solchen Regime arbeitet, wird **systolisches Array** genannt, in Analogie zum pulsierenden Blutkreislauf.

Weitere Kopplungsstrukturen sind denkbar. Man kann auch auf die Idee kommen, die ALUs durch Prozessoren zu ersetzen, sodass ein Prozessorarray entsteht. Doch

dann liegt eine Parallelisierung oberhalb des Maschinenniveaus vor. Ein Prozessorarray ist ein Spezialfall der Prozessor-Speichernetze, denen wir uns nun zuwenden.

19.3 Hardwarearchitektur oberhalb der Prozessorebene

Wir wollen nun die Komponierung informationeller Systeme *oberhalb* der Prozessorebene fortsetzen, aber nicht softwaremäßig, wie im Falle des Einprozessorrechners, sondern *hardwaremäßig*. Dabei bleiben wir beim *Datenflussparadigma*, sodass für die Komponierung die Regeln der USB-Methode gelten.

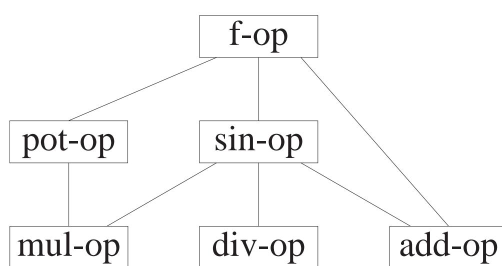
Es stellt sich die Frage nach den Bausteinen eines Operatorennetzes oberhalb der Pozessorebene. Die Antwort wird durch die Komponierung *unterhalb* der Prozessorebene nahegelegt. In Kap.13 hatten wir den Einprozessorrechner als KR-Netz entworfen, also als Netz von Kombinationsschaltungen und Registern. Der Entwurf als KR-Netz war durch die Forderung nach binär-statischer Codierung erzwungen. Damit stellt sich die Frage nach den Bausteinen oberhalb der Prozessorebene folgendermaßen: Welche Bausteine übernehmen das Transformieren von Bitketten (die Funktion der Kombinationsschaltungen) und welche das Aufbewahren von Bitketten (die Funktion der Register)? Die Antwort liegt auf der Hand: Das Transformieren ist von Prozessoren (P) und das Aufbewahren von Speichern (S) zu übernehmen. Damit besteht das weitere Komponieren im Bau von **Prozessor-Speicher-Netzen**, abgekürzt **PS-Netzen**.

Zwei Unterschiede zwischen KR-Netzen und PS-Netzen springen sofort ins Auge. Zum einen dürfen Prozessoren - im Gegensatz zu Kombinationsschaltungen - ohne Zwischenschaltung von Speichern beliebig vernetzt werden, da sie über interne Register verfügen. Zum anderen können Prozessoren per Programm zur Ausführung beliebiger Operationen befähigt werden, Steueroperationen eingeschlossen. Sie lassen sich folglich direkt (ohne hardwaremäßige Erweiterungen) nicht nur als *Arbeits*operatoren einsetzen wie die Kombinationsschaltungen in KR-Netzen, sondern auch als *Steuer*operatoren. Es besteht demnach die Möglichkeit, die Steueroperatoren einer Operatorenhierarchie als Prozessoren zu realisieren. Das ist sicher dann zweckmäßig oder sogar notwendig, wenn die Hierarchie flexibel sein soll oder wenn die Steueroperatoren umfangreiche organisatorische Aufgaben zu erledigen haben (siehe Kap.19.5.1).

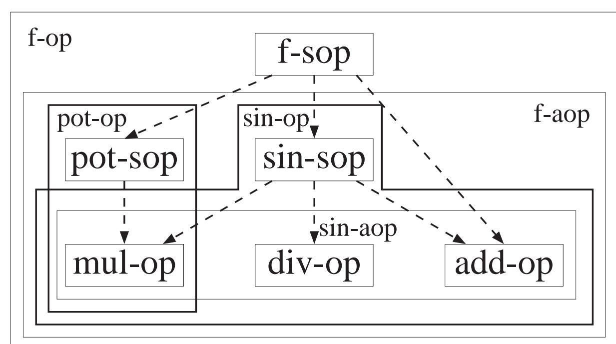
Wir wollen uns anhand eines überschaubaren Beispiels überlegen, welche hierarchischen Strukturen denkbar sind und welche Probleme auftreten können. Dazu greifen wir auf das uns gut bekannte Operatorennetz von Bild 8.1 zurück. Das Netz dient der Berechnung der Funktion

$$y = f(x) = f_1(x) = x^{\mathrm{n}} + x \quad \text{für } x \leq 0 \qquad (19.1)$$
$$y = f(x) = f_2(x) = x^{\mathrm{n}} + \sin(x) \quad \text{für } x > 0.$$

Zunächst überführen wir (19.1) in eine Operatorenhierarchie, wie es in Kap.8.1 [8.2] beschrieben wurde. Dazu wird von den Übergabewegen in Bild 8.1 abstrahiert und nur die hierarchische Struktur des f-Operators betrachtet und als Graph dargestellt. Im Gegensatz zu Bild 8.1 soll auch der Sinusoperator gemäß der Reihenzerlegung (15.5) [15.2] dekomponiert werden. Hinsichtlich der Erstellung eines realen f-Operators mögen ein Addierer (add-op), der auch subtrahieren kann, ein Multiplizierer (mul-op) und ein Dividierer (div-op) zur Verfügung stehen. Diese drei Operatoren dienen als elementare Operatoren, d.h. als Bausteinoperatoren der untersten Schicht. Damit ergibt sich der Hierarchiegraph von Bild 19.2a. Eine Kante des Graphen bedeutet, dass der jeweils tiefer liegende Operator Bausteinoperator des höher liegenden (übergeordneten) Operatores ist. Da nur ein einziger Multiplizierer zur Verfügung steht, müssen sich der pot-op und der sin-op in seine Dienste *teilen*; der mul-op ist ein sogenanntes **geteiltes Betriebsmittel**. Entsprechendes gilt für den add-op, in den sich der sin-op und der f-op teilen müssen. Dabei liegt der besondere Fall vor, dass ein Operator auf unterschiedlichen Ebenen als Bausteinoperator dient. Man beachte, dass der Graph von Bild 19.2a nur *schematisch* die hierarchische Struktur des f-Operators darstellt, die sich bei dessen Komponierung nach der USB-Methode ergibt. Darum sprechen wir von **schematischer Operatorenhierarchie**. Die Komponierung eines Kompositoperators nach der USB-Methode beinhaltet nämlich die Installierung eines Steueroperators, der die Operandenübergaben zwischen den Bausteinoperatoren steuert. Demzufolge enthält eine reale Operatorenhierarchie außer den elementaren Operatoren nur Steueroperatoren. Damit ergibt sich für den f-Operator der gestrichelte Graph in Bild 19.2b. Er entspricht dem schematischen Graph von Bild 19.2a, doch sind an die Stelle der Kompositoperatoren (der Operatoren oberhalb der elementaren Schicht) die entsprechenden Steueroperatoren getreten und die Kanten sind durch gestrichelte Pfeile ersetzt. Der so entstandene Graph



(a)



(b)

Bild 19.2 Hierarchische Struktur des Operators f-op von Bild 8.1 zur Berechnug der Funktion (19.1). (a) Schematische Darstellung; (b) Darstellung als Kombination von Rahmendiagramm und Steuerungsgraph.

gibt die *Steuerhierarchie* wieder. Beispielsweise bedeutet der Pfeil vom f-sop zum sin-sop, dass der f-sop den sin-sop steuert. Da das Steuern nur das Konditionieren, das Starten und evtl. das Stoppen betrifft, ist es zuweilen sinnfälliger zu sagen, dass der gesteuerte sop dem steuernden sop *untergeordnet* ist. Dementsprechend wird der gestrichelte Graph **Steuerungsgraph** oder **Unterordnungsgraph** genannt.

Ein Steueroperator wird mit dem von ihm gesteuerten Operatorennetz (ON) - ggf. gedanklich - zu einem Kompositoperator zusammengefasst, wie zu Beginn des Kapitels 8.1 beschrieben wurde. Nach dem Vorbild von Bild 8.1 ist in Bild 19.2b jede solche Zusammenfassung durch einen (nicht unbedingt rechteckigen) Rahmen dargestellt. Der Übersichtlichkeit halber sind der pot-op und der sin-op durch fette Rahmen hervorgehobenden. Die Gesamtheit der Rahmen heißt **Rahmendiagramm** oder **Schachteldiagramm**.

Hinter jedem gestrichelten Pfeil in Bild 19.2b verbirgt sich mindestens eine, in der Regel aber zwei hardwaremäßig realisierte Verbindungen, eine Verbindung "von oben nach unten" für die Übergabe von Steuersignalen und evtl. auch eine "von unten nach oben" für die Übergabe von Meldesignalen, z.B. von Operations-Endemeldungen. Die Pfeile stellen also **Signalwege** dar. Die graphische Darstellung der Signalwege in einer Operatorenhierarchie heißt **Signalwegegraph**.

Wir wollen nun unsere Operatorenhierarchie nach unten hin fortsetzen durch Dekomponierung des mul-op, des div-op und des add-op. Der mul-op könnte beispielsweise gemäß Bild 13.3 in einen Steueroperator und ein ON dekomponiert werden, das aus einem Addierer in einer Rückkopplungsschleife besteht. In Bild 19.3a ist eine andere Dekomponierung vorgenommen worden, und zwar ist die unterste Schicht von Bild 19.2b in ein PS-Netz dekomponiert worden. Jeder der drei Operatoren ist in einen Prozessor und einen Speicher, und die Prozessoren ihrerseits sind in je einen Steueroperator und eine RALU dekomponiert worden. Die höher liegenden Steueroperatoren sind nicht dekomponiert. Die Pfeile des Steuerungsgraphen sind in Bild 19.3b durch Signalwege (gestrichelt gezeichnet) ersetzt, sodass sich der Signalwegegraph der Hierarchie ergibt.

Die Bilder 19.2b und 19.3a zeigen eine Hierarchie mit **zentraler Steuerung** in allen Schichten. Das bedeutet, dass jeder Kompositoperator seinen eigenen Steueroperator besitzt, der die Signale generiert, welche die Ausführung der Kompositoperation (den betreffenden *Berechnungsprozess*) steuern. In diesem Fall kann der Steuerungsgraph in das Rahmendiagramm überführt werden und andersherum. Das ändert sich, wenn die Steuerung dezentralisiert wird (s.u.).

Wenn in einer Operatorenhierarchie ein Operator von mehreren Steueroperatoren gesteuert wird, d.h. wenn er mehreren Kompositoperatoren als Bausteinoperator dient (wenn er "mehreren Herren dient"), die sich in seine Dienste "teilen" müssen, kann es zu **Konflikten** kommen. Das trifft in Bild 19.2b für den Addierer zu, der vom f-sop und vom sin-sop gesteuert wird, sowie für den Multiplizierer, der vom pot-sop und vom sin-sop gesteuert wird; mit anderen Worten, ein und derselbe Operator kann von zwei *Prozessen* - eventuell gleichzeitig - als *Betriebsmittel*

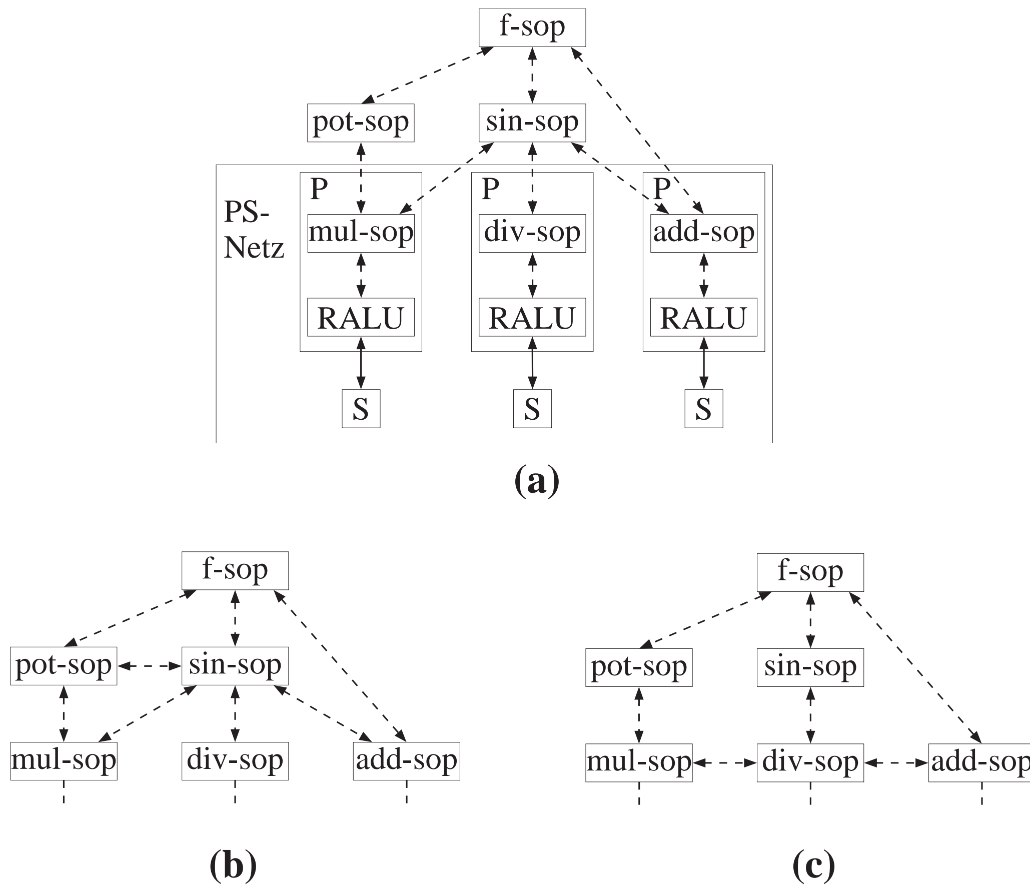


Bild 19.3 Signalwegegraph der Steuerungshierarchie von Bild 19.2; (a) - zentrale Steuerung; (b) und (c) - teilweise dezentralisierte Steuerung (RALUs und Speicher nicht eingezeichnet).

angefordert werden, der Multiplizierer beispielsweise vom Berechnungsprozess der Potenzfunktion und vom Berechnungsprozess der Sinusfunktion.

Derartigen Konflikten sind wir bereits in Kap.8.2 begegnet [8.6] [8.9]. Dort waren zwei Möglichkeiten genannt, Konflikte zu lösen, entweder durch eine übergeordnete Instanz (durch den übergeordneten Steueroperator) oder durch gegenseitige Absprache, also durch Signalaustausch zwischen den konkurrierenden Steueroperatoren. Die zweite Möglichkeit ist in Bild 19.3b für die beiden Steueroperatoren pot-sop und sin-sop angedeutet. Über den waagerechten Signalweg können die Steueroperatoren Signale austauschen und so den anderen über den eigenen *Zustand* (frei oder besetzt) informieren oder auch den anderen starten. Die Lösung von Konflikten zwischen dem pot-sop und dem sin-sop ist nun nicht mehr Aufgabe des f-sop.

In Bild 19.3c geht vom sin-sop nur ein einziger Signalweg aus und zwar zum div-sop. Das bedeutet, dass der sin-sop den div-sop wie zuvor unmittelbar, den mul-sop und den add-sop dagegen mittelbar über den div-sop steuert, sodass letzterer die Rolle eines sogenannten **Leit-Operators** spielt. Auf den sin-sop kann auch ganz verzichtet werden. Dann muss der div-sop die Aufgaben von zwei Steueroperatoren

unterschiedlicher Hierarchieebenen ausführen. Natürlich könnte auch der mul-sop als Leitoperator fungieren, oder beide könnten sich abwechseln.

Wie man sieht, gibt es viele Möglichkeiten der Dezentralisierung, und die betrachteten Beispiele sind nur erste Schritt zur *dezentralen Steuerung*. Die Einbeziehung der Bausteinoperatoren in die Steuerung kann bis zur *vollständigen Dezentralisierung* getrieben werden. *Die Steuerung in einem Kompositoperator, d.h. die Steuerung des ON, heißt* ***dezentral****, wenn sie von den Bausteinoperatoren selbst durchgeführt wird*, wenn diese also nicht nur die *Arbeitsoperationen*, sondern auch die *Steueroperationen* ausführen und sich selbst bzw. gegenseitig starten und die erforderlichen Steuersignale für die Tore bzw. Weichen generieren. Bei vollständiger Dezentralisierung stellen die einzelnen Operatoren selbständige Akteure dar, die sich auch selber starten. Ein ruhender Operator muss also erkennen, ob er eine Operationsausführung beginnen kann. Er muss lediglich darüber informiert werden, wohin er seine Ausgabeoperanden weiterzugeben hat, beispielsweise dadurch, dass ihm der Datenflussplan (bzw. ein Ausschnitt davon) verfügbar gemacht wird. Ähnlich wird vorgegangen, wenn in einem Fertigungsbetrieb der Transport der Werkstücke von Werkbank zu Werkbank *dezentral*, z.B. durch die Bediener der Werkbänke erfolgt. Damit ein Arbeitsoperator erkennt, wann er sich selbst starten kann, muss er ständig kontrollieren, ob ihm die Operanden für die nächste Operation übergeben sind, d.h. ob sie sich in den dafür vorgesehenen Operandenplätzen befinden.

Wir unterbrechen unseren Gedankengang, der zum Mehrprozessor führen soll, um uns den Unterschied zwischen Mehr- und Einprozessorrechner hinsichtlich des Steuermechanismus im Detail zu verdeutlichen. Es handelt sich um den Unterschied zwischen dem Datenflussparadigma und dem Aktionsfolgeparadigma oder auch zwischen Netzparadigma und Satzparadigma (imperativem Paradigma), der in den Kapiteln 13.7 und 18.3 ausführlich diskutiert wurde. Zu diesem Zweck ist in Bild 19.4 die Komponierung des f-Operators durch einen Einprozessorrechner dargestellt. Dementsprechend sind die drei RALUs von Bild 19.3a zu einer einzigen RALU zusammengefasst. Aus dem Steueroperator f-sop wird ein Maschinenprogramm zur Berechnung der Funktion *f*, also ein imperativer Algorithmus. Um das sichtbar auszuweisen, ist der Steueroperator als imp-f-sop bezeichnet.

In Bild 19.4 gibt es also nur eine einzige RALU und damit nur einen einzigen Prozessor, der die Funktionen der drei spezialisierten Prozessoren von Bild 19.3a übernimmt. Seine RALU muss die Operationen der drei RALUs von Bild 19.3a ausführen und sein Steueroperator muss sämtliche Steuersignale zur Steuerung der zwischen dem imp-f-sop und der RALU liegenden Schichten generieren. Dies lässt sich, wie wir wissen, mittels Firmware realisieren, z.B. in Form eines Matrixsteuerwerks (siehe Kap.13.5.5). Durch die Operationscodes der Maschinenbefehle werden die entsprechenden ROM-Zeilen des Matrixsteuerwerks gestartet. Bild 19.4 kann - ohne den imp-f-sop - als CPU eines Spezialcomputers aufgefasst werden.

Man beachte folgenden Umstand. Die Steuersignale, die bei der Abarbeitung des Maschinenprogramms generiert werden, sind völlig andere als diejenigen, die das

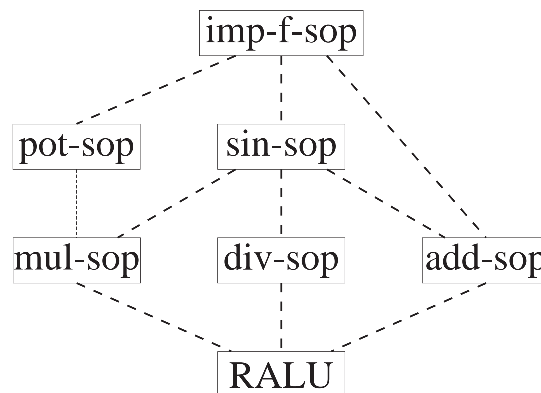


Bild 19.4 Steuerungsgraph, der aus dem Steuerungsgraphen von Bild 19.3a durch Zusammenfassung der drei RALUs zu einem RALU hervorgeht; imp-f-sop - sprachlicher, imperativer Steueroperator (Maschinenprogramm) zur Berechnung der Funktion (19.1).

PS-Netz von Bild 19.3a steuern, die also einen Operandenfluss durch ein Operatorennetz steuern. Das bedeutet, dass beim Übergang von Bild 19.3a zu Bild 19.4 das Konzept der durchgehenden hierarchischen Operatorenkomponierung zusammenbricht. Darauf wurde bereits in Kap.13.5.2 [13.12] hingewiesen. Die Suche nach einem Ausweg provoziert eine scheinbar abwegige Frage. Sollte es nicht möglich sein, den f-sop in Bild 19.3a ebenso wie den imp-f-sop als Maschinenprogramm einer geeignet konstruierten "Maschine" zu artikulieren, ohne den Sprung in das imperative Programmierparadigma zu vollziehen? Das hätte zur Voraussetzung, dass die "Maschine" eine "Datenflussmaschine" ist, die Operandenflussprogramme abarbeiten kann. Die Lösung liegt auf der Hand: die Maschine muss ein Netz oder eine Hierarchie von Prozessoren sein. Bild 19.3a liefert bereits ein Beispiel, wenn es als Hierarchie aus 6 Prozessoren interpretiert wird und zwar folgendermaßen.

Die Hierarchie enthält drei "elementare" Prozessoren, den mul-, den div- und den add-Prozessor. In der ersten Komponierungsschicht enthält sie den pot- und den sin-Prozessor und in der obersten Schicht den f-Prozessor. Jeder Prozessor verfügt über seine eigene RALU und ist in der Lage, beliebige Programme abzuarbeiten. Aber nur die drei elementaren Prozessoren führen diejenigen Rechnungen aus, die Werte der Funktion (19.1) liefern. Die übrigen drei Prozessoren sind reine Steueroperatoren. Jeder steuert den Datenfluss in dem ihm untergeordneten PS-Netz.

In dieser Weise interpretiert stellt Bild 19.3a einen *Mehrprozessorrechner mit hierarchischem Aufbau* dar. Wenn die Prozessoren ihre eigenen Speicher besitzen, wird aus dem Mehrprozessorrechner ein *Mehrcomputersystem*. Um diesen Aspekt deutlicher hervortreten zu lassen, ist die in Bild 19.3a dargestellte Architektur in Bild 19.5 um zusätzliche Prozessoren, Speicher und E/A-Geräte erweitert worden, die über einen Bus miteinander verbunden sind. Der Bus stellt eine *Schiene* dar, über

die alle “Teilnehmer” (Prozessoren, Speicher) untereinander Signale und Daten austauschen können (vgl. Kap.12.3.2). Bild 19.5 soll auf zweierlei Weise interpretiert werden:

Erste Interpretation: Die Prozessoren oberhalb der Schiene sind Steueroperatoren, die Prozessoren unterhalb der Schiene sind Arbeitsoperatoren.

Zweite Interpretation: Sämtliche Prozessoren sind Arbeitsoperatoren, die auch die Rolle von Leitprozessoren übernehmen können.

Zur ersten Interpretation. Sie ist auf das Komponieren höherer Kompositoperatoren orientiert. Jeder der oberen Prozessoren kann aus den unteren Prozessoren und den Speichern PS-Netze komponieren. Die Komponierungshierarchie kann nach obenhin fortgesetzt werden, wobei ein Bus der nächsthöheren Ebene einzurichten wäre. Es entsteht eine Prozessor- und Bushierarchie. Zentrale wie teilweise dezentrale Steuerung ist möglich.

Für einen Nutzer, der das PS-Netz Berechnungen ausführen lassen will, ist es wünschenswert, das Netz als einen einzigen Operator behandeln und wie einen

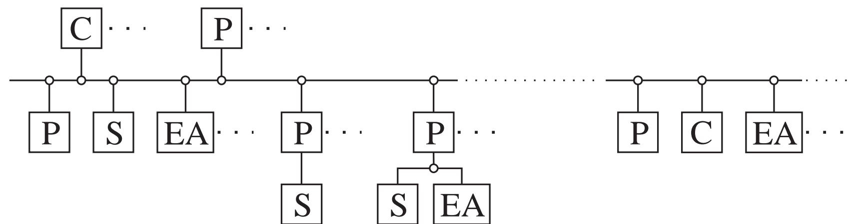


Bild 19.5 PS-Netz mit Busarchitektur. P - Prozessor, S - Speicher, C - Computer, E/A - Ein-Ausgabegerät.

Einprozessorrechner programmieren zu können. Das lässt sich mit Hilfe geeigneter Organisationsprogramme durchaus erreichen, sodass das PS-Netz zu einem *virtuellen Einprozessorrechner* wird. Diese Organisationsform eines PS-Netzes ist i.Allg. gemeint, wenn von *Mehrprozessorrechnern* die Rede ist.

Wir vereinbaren: *Ein PS-Netz, das durch ein einziges Programm gesteuert wird, bezeichnen wir als* ***Mehrprozessorrechner****, unabhängig davon, ob in dem PS-Netz Programmteile parallel abgearbeitet werden oder nicht.* Ein Mehrprozessorrechner kann als SIMD- oder als MIMD-Rechner arbeiten (siehe den letzten Absatz von Kap.19.2.2). Maschinenprogramme von Mehrprozessorrechnern können sowohl Operandenflussprogramme als auch Aktionsfolgeprogramme (imperative Programme) sein. Der Zweck, der mit Mehrprozessorrechnern verfolgt wird, ist in erster Linie die Erhöhung der Rechengeschwindigkeit durch *Parallelisierung*.

Zur zweiten Interpretation. Sie ist auf die Kommunikation zwischen *gleichberechtigten* Prozessoren orientiert. Bei vollständiger Gleichberechtigung muss die Kommunikation von den Prozessoren selbst, also *dezentral* organisiert werden, ebenso wie der Zugriff auf die Speicher. Wenn jeder Prozessor seinen eigenen

Speicher besitzt, ergibt sich ein **Rechnernetz**. Die Prozessoren können sich (evtl. unter Einbeziehung von Speichern) zu Netzen (Operatorennetzen, Kompositoperatoren) zusammenschließen. Da der Bus von mehreren, eventuell sogar von sehr vielen Teilnehmern als Betriebsmittel in Anspruch genommen werden kann, muss die Busvergabe an die Prozessoren geregelt sein. Sie kann *zentral* durch einen speziellen Steuerprozessor (auch **Busarbiter** genannt; in Bild 19.4 nicht vorhanden) oder *dezentral* durch die kommunizierenden Prozessoren selber erfolgen. Eine bewährte dezentrale Methode besteht darin, dass ein Prozessor, der den Bus in Anspruch nehmen möchte, zuvor in ihn "hineinhorcht", um festzustellen, ob er frei ist oder ob gerade "gesprochen wird".

Ergänzend sei an das Kapitel 12.3.2 erinnert. Dort hatten wir verschiedene *Kommunikationsstrukturen* betrachtet, z.B. die Ringverbindung von Bild 12.6, aber auch die Möglichkeit, den Bus durch einen *Mehrkanalkommutator* zu ersetzen, z.B. durch einen *Kreuzschienenverteiler* (vgl. Bild 12.5), sodass gleichzeitig mehrere Verbindungen unabhängig voneinander hergestellt werden können. Die Schiene in Bild 19.5 kann als symbolische Darstellung jeder beliebigen Art von Kommunikation zwischen den Elementen des Netzes aufgefasst werden.

Diese wenigen Bemerkungen über PS-Netze und Prozessorhierarchien lassen die strukturelle (architektonische) Vielfalt erkennen, die möglich ist. Sie lassen aber auch die Probleme erkennen, die sich aus der Vielfalt der Kommunikationsmöglichkeiten ergeben. Dabei haben wir uns nur für die *Struktur* interessiert. Wie aber kann gewährleistet werden, dass ein System, das aus sehr vielen kooperierenden Prozessoren und Speichern besteht, "richtig" funktioniert, dass jeder Operator in jedem Augenblick "das Richtige tut" und dass die unübersehbare Menge von Datenübertragungen nicht in einem Chaos versinkt? Wie kann gesichert werden, dass in einem so komplexen System ein reibungsloser "*Betrieb*" aufrechterhalten wird? Dieser Frage werden wir uns in Kapitel 19.5 zuwenden, das die Überschrift "Betriebssystem" trägt.

Dieses Kapitel soll mit einer vielleicht überraschenden Feststellung hinsichtlich der praktischen Realisierung und Nutzung von Mehrprozessor- und Mehrcomputersystemen beendet werden. Wie man den Medien entnehmen kann, die viel von Datenautobahn und Internet reden, befinden sich die Rechnernetze (zweite Interpretation von Bild 19.5) in stürmischer Entwicklung. Verglichen damit ist relativ wenig von neuen Entwicklungen auf dem Gebiet der Multiprozessorrechner, also der echten Parallelrechner (erste Interpretation von Bild 19.5) zu hören.

Der Motor der schnellen Entwicklung von Rechnernetzen ist die Wirtschaft. Der Grund für die relative Stagnation der Entwicklung von Parallelrechnern ist in erster Linie die Schwierigkeit, diese zu programmieren. Es stellt sich die Frage, ob vorrangig neue Architekturen oder neue Sprachen zu entwickeln sind. Um beides bemühen sich Computer- und Spracharchitekten seit vielen Jahren. Es gibt zwar Fortschritte, ein Durchbruch konnte jedoch bisher nicht erzielt werden. Das ist insofern erstaunlich, als jeder Mensch über ein informationelles System verfügt, das

in höchstem Grade parallel arbeitet, das Gehirn. Sollte es nicht möglich sein, diesen Schatz zu heben?

Zwar haben die Neurophysiologen viele Einsichten in die Tätigkeit des Gehirns zutage gefördert. Die Erkenntnisse lassen sich aber nicht so einfach für den Entwurf eines *universell programmierbaren* Parallelrechners verwerten. Der Grund liegt in dem Umstand, dass das Gehirn nicht *programmiert* wird, sondern *sich anpasst*, dass es *lernt*. Es ist also durchaus verständlich, dass die Übernahme biologischer Prinzipien der Informationsverarbeitung zu *lernfähigen* Architekturen geführt hat, den künstlichen *neuronalen Netzen* (vgl. Kap.9.4), die jedoch nicht *programmierbar* sind wie ein Prozessorcomputer. Andrerseits scheint der Schritt vom ALU-Array zum neuronalen Netz gar nicht so groß zu sein, zumindest nicht hinsichtlich der Struktur. Wir werden diesen Gedanken nicht weiter verfolgen, da er über das Thema des Buches hinausgeht.

19.4 Architektur kausaldiskreter Systeme

An dieser Stelle unterbrechen wir den Gang unserer Überlegungen, bevor wir uns dem Betriebssystem zuwenden. Wir wollen uns noch einmal die Grundzüge des Komponierens von Operatorenhierarchien vergegenwärtigen und uns die Anwendungsbreite der Methode der uniformen Systembeschreibung anhand zweier Beispiele verdeutlichen, eines Mathematiksystems und einer automatischen Fabrik. Zur Veranschaulichung der zu betrachtenden Hardware-Software-Hierarchie benutzen wir das in Bild 19.6 dargestellte Blockbild.

Bild 19.6 kann als verallgemeinertes Blockbild kausaldiskreter Systeme aufgefasst werden, die sowohl Information verarbeitende Operatoren (IV-Operatoren) als auch Stoff verarbeitende Operatoren (Fertigungsoperatoren) enthalten können. Wir schließen zunächst die Stoff verarbeitende Schicht (Block 1) aus der Betrachtung aus und konzentrieren uns auf die Information verarbeitenden Schichten (IV-Schichten) der Hierarchie oberhalb der strichpunktierten Linie.

Wenn das Blockbild oberhalb dieser Linie die gesamte IV-Hierarchie bis hinab zu den elementaren Operatoren darstellen soll, muss die strichpunktierte Linie die Grenze bilden zwischen kausalkontinuierlicher Beschreibung unterhalb der Linie und kausaldiskreter Beschreibung oberhalb der Linie. Die Operatoren der untersten Schicht (Schicht 0) müssen demzufolge Schwellenoperatoren sein [8.13] [9.5]. Für sie interessieren wir uns hier nicht, auch nicht für die Schalter (Transistoren) und die booleschen Operatoren. Wir überspringen alle vor dem Kapitel 13.5 behandelten Komponierungsschritte und beginnen mit der RALU-Schicht (Schicht 4, ALUs mit ihren Registern).

Um unsere Aufmerksamkeit nicht vom zentralen Thema, dem Komponieren von Operatoren, ablenken zu lassen, abstrahieren wir von allen weiteren Bestandteilen des Systems, von Operanden, Speichern und E/A-Geräten, von der gesamten *peri-*

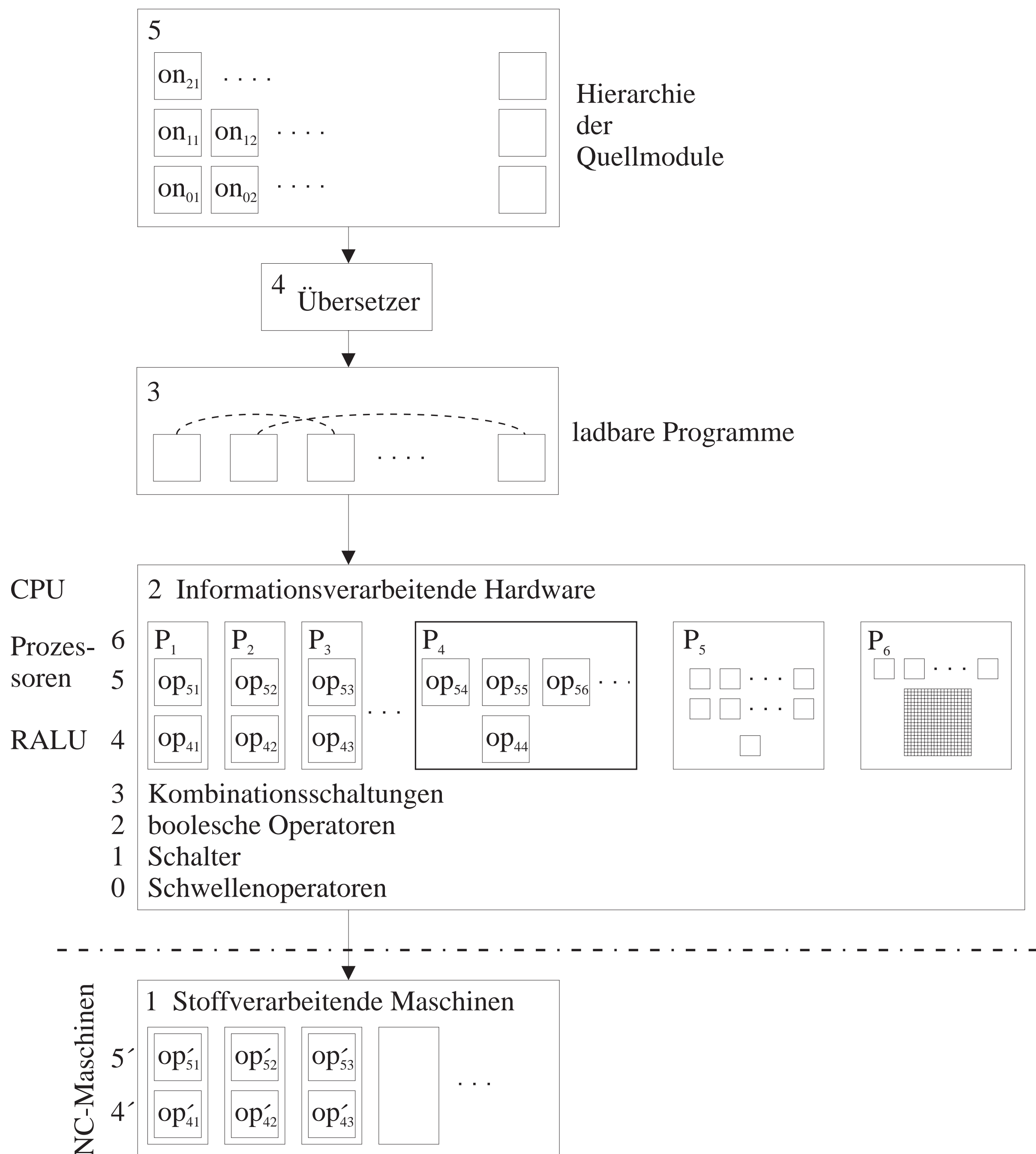


Bild 19.6 Blockbild kausaldiskreter Systeme

pheren Hardware und von allen eventuell erforderlichen organisatorischen Maßnahmen. Diese Bestandteile werden in Kap.19.5 eine umso größere Rolle spielen.

Die Behandlung der angekündigten Beispiele soll durch eine verallgemeinernde Wiederholung vorbereitet werden. Dazu wollen wir die hierarchische Komponierung des f-Operators von Bild 19.3a in die Hierarchie des Bildes 19.6 hineinprojizieren und zwar in den Block 2, in die "Informationsverarbeitende Hardware". Dadurch erhalten die abstrakten Operatoren von Bild 19.6 konkrete Inhalte. Die Operatoren

op_{41}, op_{42} und op_{43} der Schicht 4 werden zu den RALUs von Bild 19.3a und die Operatoren op_{51}, op_{52} und op_{53} der Schicht 5 zu den RALU-Steueroperatoren mul-sop, div-sop und add-sop. Block 2 enthält also drei "Spezialprozessoren" P_1, P_2 und P_3 für das Multiplizieren, Dividieren und Addieren. Allgemein kann Block 2 beliebig viele Prozessoren enthalten, wobei einzelne Operatoren auch in mehreren Exemplaren realisiert sein können. Wenn ein oder mehrer Prozessoren gemeinsam mit einem Arbeitsspeicher einen Computer bilden, stellt die Gesamtheit der Prozessoren die CPU des Computers dar.

Der Aufbau der CPU eines Computers (Block 2) in der beschriebenen Form als "*Spezialprozessor-Architektur*" durch Zusammenschalten von Spezialprozessoren (im Beispiel die Prozessoren P_1, P_2, P_3) ist unüblich. Der Prozessor P_4 (durch einen dicken Rahmen hervorgehobenen) spiegelt die typische Architektur der CPU eines PC wider. P_4 kann man sich entstanden denken durch Zusammenfassung der Prozessoren P_1, P_2 und P_3 im Sinne des Überganges von Bild 19.3a zu Bild 19.4. Die Operatoren op_{54}, op_{55} und op_{56} werden meistens als Firmware realisiert in Form eines Matrixsteuerwerks.

Mit dem Prozessor P_5 in Block 2 soll eine Prozessorarchitektur angedeutet werden, die zwei Firmwareschichten enthält. Auch mehrere Schichten sind möglich. P_6 stellt einen Prozessor mit ALU-Array dar. Block 2 kann auch die anderen in Kap.19.2 besprochenen Architekturen enthalten, wie Pipeline- und Vektorrechner-Architekturen. Wir werden im Weiteren aber davon ausgehen, dass Block 2 nur den typischen PC-Prozessor P_4 enthält mit einer mehr oder weniger großen Anzahl von Steueroperatoren op_{5i}.

Wenn die Steueroperatoren der Schicht 5 als Firmware realisiert sind, ist es naheliegend, sie zur Hardware zu rechnen, da sie in ihrer äußeren Gestalt als Hardware in Erscheinung treten.[6] Den Operatoren der Schicht 5 (gegebenenfalls auch höherer Firmwareschichten) entsprechen die Operationscodes der Maschinensprache. In einem Maschinenprogramm ähnlich dem von Bild 15.2a entsprächen den Operatoren op54, op55 und op56 die Operationscodes MUL, DIV und ADD. Wenn der Potenzierer und die Sinusfunktion nicht als Firmware realisiert sind und demzufolge nicht durch einen Operationscode in der Prozessorsprache vertreten sind, wird man versuchen, geeignete Assemblerprogramme in einer Programmbibliothek zu finden. Hat die Suche Erfolg, sind die Programme zu assemblieren und in Block 3 einzuordnen. Andernfalls müssen die Funktionen programmiert werden, evtl. in einer geeigneten höheren Programmiersprache. Dann gehören sie zu Block 5.

Block 5 beinhaltet die Hierarchie sprachlicher Operatoren. Sie sind im Unterschied zu den realen Operatoren von Block 2 mit on (Operation) bezeichnet. Die

6 Es ist allerdings eine strittige Frage, ob es gerechtfertigt ist, Firmware zur Hardware zu rechnen, denn es handelt sich um Programme. Die Frage wird im Zusammenhang mit der Patentierfähigkeit von Firmware zu einem juristischen Problem.

elementaren Bausteinoperationen der Hierarchie (die Elemente der untersten Schicht in Block 5) sind die Operationen der gewählten Programmiersprache. Bei der Projektion unseres f-Operators in das Blockbild 19.6 werden der pot-sop und der sin-sop, falls sie nicht zur Firmware gehören, zu den Operationen on_{11} bzw. on_{12} und der f-sop zur Operation on_{21}. Im Hinblick auf das zu besprechende Mathematiksystem ist das Wort *Operation* passender als das Wort *Operator*, weil es dem mathematischen Sprachgebrauch besser entspricht. Anstelle der Wörter *Operator* und *Operation* darf auch das Wort *Funktion* verwendet werden. Das bietet sich insbesondere dann an, wenn die verwendete höhere Programmiersprache eine funktionale Sprache ist.[7]

Die Operationen höherer Programmiersprachen werden, wie wir wissen, nicht nach der USB-Methode (also nicht unter Verwendung von Operandenflussknoten) aus den Operationen der Maschinensprache komponiert. Vielmehr wird zunächst mehr oder weniger intuitiv eine Sprache zur Artikulierung von Operationsvorschriften vereinbart, die den Bedürfnissen und Gewohnheiten der Programmierer entgegenkommt. Danach erst wird die interne Semantik der Sprache definiert, d.h. es wird festgelegt, welche Prozesse durch Ausdrücke der vereinbarten Sprache im Computer ausgelöst werden sollen. Diese Vorgehensweise ist uns aus den Kapiteln 13.5.2 und 13.5.3 bekannt, wo wir zuerst die Maschinensprache vereinbart und dann erst die interne Semantik der Operationscodes “rekonstruiert” haben.

Aufgrund der festgelegten Semantik wird ein Übersetzerprogramm geschrieben, das die “Ankopplung” der höheren Programmiersprache an die Maschinensprache bewerkstelligt. In Kap.16.4 haben wir uns die einzelnen Schritte überlegt, aus denen das Übersetzen besteht. Die dortigen Überlegungen können noch dahingehend erweitert werden, dass innerhalb einer Programmhierarchie verschiedene Programmiersprachen zugelassen sein können, sodass Block 5 in Teilhierarchien zerfällt, die durch Übersetzerprogramme miteinander kompatibel gemacht werden. Das kann z.B. zweckmäßig sein, wenn ein Operator hoher Komponierungsstufe (ein Softwaresystem) sowohl numerische als auch analytische Rechnungen durchführen soll. Dann kann es sinnvoll sein, numerische Operationen imperativ und analytische Operationen funktional zu programmieren. Wenn der Operator darüberhinaus logische Schlussfolgerungen ziehen soll, kann die zusätzliche Verwendung einer logischen Sprache angebracht sein.

Die Produkte der Übersetzung werden als *ladbare* Programme abgespeichert, meistens in einem Scheibenspeicher, z.B. auf der Festplatte eines PC. Dabei kommt das Prinzip der relativen Adressierung zur Anwendung. Nach Ersetzen der relativen Adressen eines ladbaren Programms durch absolute Adressen lädt der *Lader* das

---

7 Der Leser lasse sich nicht durch das Hin- und Herspringen zwischen dem Operator-, dem Operations- und dem Funktionsbegriff irritieren. Angesichts unserer Vereinbarungen hinsichtlich Eindeutigkeit und Realisierbarkeit [8.16], die nach wie vor gelten, sind wir berechtigt, die Wörter *sprachlicher Operator, Operation* und *Funktion* als Synonyme zu verwenden.

Programm in einen bestimmten Bereich der Hauptspeichers (des Arbeitsspeichers von P_4). Wenn das zu ladende Programm aus einzelnen Teilen besteht, z.B. aus einem Hauptprogramm (etwa dem f-sop), welches Unterprogramme aufruft (den pot-sop und den sin-sop), können diese vor dem Laden durch den *Binder* in das Hauptprogramm eingebunden werden [15.4]. Das Aufrufen zwischen den ladbaren Programmen ist in Block 3 durch gestrichelte Linien angedeutet. Entlang der senkrechten Pfeile werden sprachliche Operatoren (Befehle, Programme) übergeben.

Wir wollen nun die Einschränkung auf den Einprozessorrechner aufheben und annehmen, dass Block 2 nicht nur einen Prozessor vom Typ P_4 enthält, sondern aus einem PS-Netz mit mehreren Prozessoren besteht. Es können also evtl. mehrere Prozessorsprachen existieren. Wie ist dann die Maschinensprache zu definieren? Im Prinzip könnte sie eine Operandenflusssprache sein, denn Block 2 stellt nun ein *Prozessornetz* dar, und Aufgabe eines Maschinenprogramms wäre es, den Operandenfluss durch dieses Netz zu steuern. Tatsächlich sind gegenwärtig die Maschinensprachen von Mehrprozessorrechnern i.d.R. imperative Sprachen und ein Maschinenprogramm stellt eine Folge von Maschinenbefehlen (Prozessorbefehlen) dar. Das ist eine Konzession an den Stand der Technik. Für den breiten Einsatz von Operandenflusssprachen auf der Maschinenebene sind die technischen und wohl auch die begrifflichen Voraussetzungen noch nicht in ausreichendem Maße geschaffen.[8]

Nach dieser verallgemeinernden Wiederholung kommen wir zu dem ersten der beiden angekündigten Beispiele.

4 Beispiel 1: Mathematiksystem[9]

Wir versetzen uns nun in die Situation eines Softwareentwicklers (eines Programmierers, eines Programmierteams oder eines Softwarehauses), dem ein “Handbuch” der Mathematik in die Hand gedrückt wird mit der Bitte, ein System zu entwickeln, das in der Lage ist, sämtliche Operationen auszuführen, die in dem Handbuch mathematisch (d.h. mittels mathematisch formulierter Vorschriften) definiert sind. Ein Nutzer soll unter Verwendung der ihm gewohnten mathematischen Sprache das System mit der Ausführung irgendeiner der implementierten Operationen beauftragen können.

Zuerst wird sich der Entwickler für eine bestimmte Gerätekonfiguration und für eine Programmiersprache entscheiden (evtl. für mehrere). Damit legt er den Block 2, die unterste Schicht von Block 5 und den Block 4 fest. Die Programmiersprache sollte der mathematischen Sprache des Handbuches möglichst gut angepasst sein, denn die weitere Aufgabe des Entwicklers besteht darin, den Block 3 zu füllen, d.h.

---

8 Siehe die Schlussbemerkungen zu Kap.19.3.

9 Es werden umfangreiche Mathematiksysteme angeboten, z.B. die Systeme Axiom, Maple und Mathematica. Das Arbeiten mit diesen Systemen ist in [Bronstein 95] im Kapitel Computeralgebrasysteme beschrieben.

die Operationsvorschriften des Handbuches in ladbare Programme zu überführen. Unter der Annahme, dass Block 4 bereits existiert, besteht die Aufgabe des Programmierers im Aufbau der Hierarchie von Block 5. Er wird zunächst eine *schematische* Operationenhierarchie (Funktionenhierarchie) entwerfen (vgl. Bild 19.2a) und anschließend die einzelnen Operationen ausprogrammieren. Dabei kann er sowohl *aufwärts* (*bottom up*), also *komponierend*, als auch *abwärts* (*top down*), also *dekomponierend* vorgehen. Beim Abwärtsprogrammieren einer Operation müssen für deren Bausteinoperationen zunächst "leere" Bezeichner eingeführt werden, d.h. Bezeichner von noch nicht programmierten Programmen, es sei denn, die Maschinenebene ist bereits erreicht.

Jeder Operator (jede Operation) der höheren Schichten steht an der Spitze einer Teilhierarchie. Diese ist umso umfangreicher, je mehr Methoden der Operator zur Verfügung stellt (bei der Operation zur Anwendung kommen). Wenn das System beispielsweise das Integrieren mittels Partialbruchzerlegung "beherrschen" soll, muss der Operator "*Integrierer*" alle dafür erforderlichen Bausteine enthalten. Dazu gehören sowohl analytische als auch numerische Operatoren (Operationen). Die Grundideen des Komponierens mathematischer Operationen, sowohl numerischer als auch analytischer, kennen wir aus Kap.15.[10]

Block 5 kann relativ unabhängig von Block 2 mit Programmen angefüllt werden. Doch früher oder später muss sich der Entwickler überlegen, wie die Ausführung der Programme durch Block 2 im einzelnen zu organisieren ist. Die Probleme, die sich beim Systementwurf in dieser Hinsicht stellen, unterscheiden sich in charakteristischer Weise je nachdem, für welche Gerätekonfiguration der Entwickler sich entschieden hat, für einen Einprozessorrechner oder für ein PS-Netz der Art, wie es in Bild 19.5 dargestellt ist.

Im Falle des Einprozessorrechners müssen sich sämtliche Prozesse in die Dienste *eines* Prozessors, *eines* Arbeitsspeichers und eventuell *einer* Festplatte *teilen*, überhaupt in die Dienste jedes Betriebsmittels, über das der Computer nur in einem einzigen Exemplar verfügt. Das Hauptproblem ist folglich die **geteilte Nutzung von Betriebsmitteln**. Ihm sind die Kapitel 19.5.2 und 19.5.3 gewidmet.

Im Falle eines PS-Netzes muss gesichert sein, dass alle Prozessoren, überhaupt alle Betriebsmittel, die an der Programmausführung teilnehmen und die auf ein großes Areal, eventuell sogar auf mehrere Rechnernetze in verschiedenen Städten oder gar Ländern *verteilt* sein können, richtig miteinander *kooperieren*. Damit kommt zur *ge*teilten Nutzung von Betriebsmitteln ein zweites Problem hinzu, die **Nutzung *ver*teilter Betriebsmittel**. Ihm ist das Kapitel 19.5.4 gewidmet.

10 Es sei daran erinnert, dass wir jedes Rechnen mit Variablen als *analytisches* Rechnen bezeichnet hatten. Danach gehören die Operationen der Algebra, der Analysis und des Prädikatenkalküls zum analytischen Rechnen.

Außer dem Einprozessorrechner und dem PS-Netz ist eine weitere Hardwarekonfigurationen möglich, an die man zunächst vielleicht nicht denkt, weil sie ungewöhnlich ist. Man erkennt sie, wenn man sich vergegenwärtigt, dass sämtliche auszuführenden Operationen durch das Handbuch festgelegt sind. Das System braucht also
5 nicht in der Lage zu sein, *neue* Programme, die ein Nutzer programmiert hat, abzuarbeiten. Insofern braucht das System nicht flexibel zu sein, es kann "fest verdrahtet" werden. Das bedeutet, dass das System im Prinzip ohne Verwendung von Prozessoren realisiert werden kann.

Wir wollen für einen Augenblick diesem Gedanken nachgehen und vergessen, dass es Prozessoren gibt. Dann müssen die Operationsvorschriften des Handbuches hardwaremäßig realisiert werden, z.B. als ROM-Bausteine. Das Ergebnis ist eine *ROM-Hierarchie* [13.7] ohne jede Software, ein reiner *ROM-Computer*. Es handelt sich gewissermaßen um den *hardwaremaximalen Grenzfall* (Firmware zur Hardware gerechnet). Er ist in Bild 19.6 in Form des Prozessors P_5 enthalten, der aus vielen Firmwareschichten bestehen würde und evtl. über mehrere ALUs oder auch über ein ALU-Array verfügen könnte. Oberhalb von P_5 gäbe es keinerlei Softwareschichten. Eine Operation würde vom Nutzer per Kommando abgerufen. Im entgegengesetzten Fall, dem *softwaremaximalen Grenzfall*, werden sämtliche Programme von einem einzigen Prozessor ausgeführt. Dieser Fall liegt z.B. vor, wenn das Mathematiksystem auf einem PC läuft.

Ein ROM-Computer ist sehr schnell. Er ist aber auch sehr teuer. Darum wird der Entwickler eines Systems, das große Programme sehr schnell abarbeiten muss (man denke z.B. an die Wettervorhersage oder an die Raketensteuerung), nicht gleich zum ROM-Computer greifen, sondern zu einem Mittelding zwischen den beiden Grenzfällen, zu irgendeinem PS-Netz, das z.B. als Vektorrechner, als Mehrprozessorrechner oder als Mehrcomputersystem aufgebaut sein kann.

Beispiel 2: Automatische Fabrik.

Wir werden nun auch den Block 1 von Bild 19.6 in die Betrachtung einbeziehen und beginnen wieder mit einer verallgemeinernden Wiederholung und projizieren den Fertigungs-Kompositoperator von Bild 8.3 in das Blockbild 19.6. Um die Analogie zwischen dem IV-Kompositoperator von Bild 8.1 und dem Fertigungs-Kompositoperator von Bild 8.3 auch in Bild 19.6 zu verdeutlichen, bezeichnen wir in Block 1 diejenige Schicht, die der Schicht 4 in Block 2 entspricht, als Schicht 4′. Dann stellen die Operatoren op'_{41}, op'_{42} und op'_{43} den Bohrer, den Polierer und den Fügeoperator, z.B. einen Schweißautomaten, dar. Der Trennoperator muss als zusätzlicher Operator eingeführt werden. Er war im Falle des IV-Operators von Bild 8.1 überflüssig, da das "Trennen" einer Bitkette (des Operanden eines IV-Operators) durch eine Spaltegabel ausgeführt werden kann.

Die Operatoren der Schicht 4 bzw. 4′, also RALUs bzw. Werkzeugmaschinen, sind diejenigen Operatoren, welche letzten Endes "die Arbeit machen", d.h. die IV-Operationen bzw. Fertigungsoperationen ausführen. Darum nennen wir die

Schichten *Arbeitsschichten*. Ihre Operatoren, die *Arbeitsoperatoren*[11], können weiter dekomponiert werden, die RALUs in Kombinationsschaltungen und Register und weiter in boolesche Operatoren, die Maschinen in ihre Bauelemente (dem Leser ist es überlassen, die Bohr-, Polier- und Schweißmaschine in Bausteine zu dekomponieren.). Die Dekomponierung kann bis an die Grenze zwischen kausaldiskreter und kausalkontinuierlicher Beschreibung fortgesetzt werden [8.12]. Für das Weitere interessiert die Dekomponierung lediglich bis zur Arbeitsschicht.

Ein Fertigungsoperator besitzt in der Regel seinen eigenen, evtl. mehrschichtigen Steueroperator, z.B in Form eines Matrixsteuerwerks (vgl. die Waschmaschinensteuerung in Kap.12.3.4). Die Operatoren op'_{51}, op'_{52} und op'_{53} in Block 1 sind solche Steueroperatoren. Damit entspricht die Schicht 5′ von Block 1 der Schicht 5 von Block 2. Ein Arbeitsoperator kann mit seinem Steueroperator zu einem Kompositoperator zusammengefasst werden, zu einem Prozessor bzw. zu einer sogenannten **NC-Maschine** (NC von numeric control). Wir gehen von der Vorstellung aus, dass die stoffverarbeitenden Maschinen in Block 1 NC-Maschinen sind, die durch ein Transportnetz zu Kompositoperatoren (NC-Maschinennetzen) verbunden werden können.

In Kap.8.2 hatten wir das Operatorennetz von Bild 8.3 zur Herstellung von Winkeln eingesetzt. Um die Winkelproduktion zu automatisieren, müssen die NC-Maschinen durch Transportoperatoren zu einem Maschinennetz (Operatorennetz) verbunden werden (in Analogie zur Verbindung der Prozessoren P_1, P_2 und P_3 zu einem Prozessornetz), und der Operandenfluss durch das Netz muss durch einen Steueroperator gesteuert werden. Als Steueroperator könnte beispielsweise ein Prozessor vom Typ P_4 oder P_5 fungieren. Seine Aufgabe ist die Generierung aller erforderlichen Steuersignale einschließlich der Operationsanweisungen (der “Operationscodes”) für die NC-Maschinen.

Man beachte, dass diese Aufgabe nicht identisch ist mit derjenigen, die ein Prozessor im Falle reiner Informationsverarbeitung spielt. Dort hat er drei Teilaufgaben auszuführen, den nächsten Befehl zu holen, die Steuersignale zu generieren und die befohlene Operation auszuführen. Wenn der Prozessor einen Fertigungsprozess steuert, entfällt die letzte Teilaufgabe, denn sie wird vom NC-Maschinen-Netz ausgeführt, dem der Prozessor seine Steuersignale über den Halbkommutator HK übergibt. *Dies ist der einzige Punkt, in dem sich der Steuermechanismus eines automatischen Fertigungssystems von dem eines IV-Systems unterscheidet*.

Die firmwaremäßige Komponierung von Fertigungsoperationen ist auf Schicht 5′ in Block 1 und Schicht 5 in Block 2 aufgeteilt. Von dieser Aufteilung hängt die *NC-Sprache* ab, d.h. die Sprache, die vom NC-Maschinen-Netz, also von den Bearbeitungs- und Transportoperatoren verstanden wird. Falls keine höhere Pro-

6

11 Als Arbeitsoperatoren werden in diesem Kapitel ausschließlich die Operatoren der Arbeitsschicht bezeichnet, also der Schicht 4 bzw. 4′.

grammiersprache zur Verfügung steht, muß der Anwender seine Programme (Fertigungsvorschriften, beispielsweise ein Programm für die Winkelproduktion) in der NC-Sprache schreiben. Wenn eine höhere Programmiersprache verwendet wird, könnte on'_{11} die Operation “Winkelproduktion” sein.

Wir erweitern nun gedanklich den Fertigungs-Kompositoperator von Bild 8.3 Schritt für Schritt (schichtweise) zu einer automatischen Fabrik für die Produktion von Autos. Block 1 wird zu einem großen Park von NC-Maschinen erweitert. Wir nehmen an, dass eine höhere Programmiersprache zur Formulierung von Operationsvorschriften zur Verfügung steht, sodass die Operationen von Schicht 0 in Block 5 den Operationen dieser Sprache entsprechen. In Schicht 1 von Block 5 liegen Steueroperatoren, die relativ kleine Bausteinprozesse steuern. Dazu kann auch die Herstellung der Winkel gemäß Bild 8.3 gehören. Von Schicht zu Schicht werden Werkstücke zunehmender Komplexität hergestellt. In einer entsprechend hohen Schicht werden die Werkstücke auf einem Fließband zu Bausteinen eines Autos montiert und in einer noch höheren Schicht zu einem Auto. Sämtliche Operatoren oberhalb der Schicht 4′ sind *Steueroperatoren* und damit IV-Operatoren, denn sie empfangen, verarbeiten und senden Signale (Meldungen und Steuersignale).

Bei unserem gedankliche Komponieren haben wir von vielem abstrahiert, was neben der Ausführung der eigentlichen Fertigungsoperationen erfolgen muss. Dazu gehört das Zwischenlagern von Operanden, d.h. von Materialien oder Zwischenprodukten (Halbzeugen) und ihr Transport von und zum Lager. Die Ablagen in der Nähe der Maschinen kann man mit den Registern, Caches und Speicherzellen des Arbeitsspeichers vergleichen, größere Lagerhallen mit den externen (peripheren) Speichern (Scheibe und Band).

Es scheint sich anzubieten, auch in Fertigungssystemen zwischen *zentralen* Operatoren (NC-Maschinen) und *peripheren* Operatoren (Transportoperatoren, Lager) zu unterscheiden. (Letztere können als Operatoren aufgefasst werden, deren Ausgabeoperanden mit den Eingabeoperanden identisch sind.) Diese Trennung, die für IV-Systeme üblich ist, kann für Fertigungssysteme oft nicht verwirklicht werden, wenn Bearbeitungs- und Transportoperationen miteinander kombiniert sind, wie z.B. bei Fließbändern oder Robotern.

Unsere Beschreibung einer automatischen Autofabrik ist natürlich enorm vereinfacht, demonstriert aber doch anschaulich die zu Grunde liegende architektonische Idee, die Hierarchie aus Fertigungs- und IV-Operatoren. Die steuernden IV-Operatoren sind in der Regel Prozessoren und die Steuerhierarchie ist eine Prozessorenhierarchie.

Wenn die Produktion und damit auch die Produkte normiert sind, wenn beispielsweise nur ein einziger Autotyp hergestellt wird, ist es nicht unbedingt notwendig, dass die Steueroperatoren programmierbar sind. Das bedeutet, dass auch die automatische Fabrik in Analogie zum Mathematiksystem als ROM-Hierarchie realisiert werden kann. Auf die Programmierbarkeit der Steuerung kann freilich nicht verzichtet werden, wenn der Produktionsprozess flexibel sein soll, wie beispielsweise die

Arbeit einer automatischen *Maß*schneiderei. Aber auch die Autofabrik wird auf den höheren Ebenen der Arbeitsorganisation programmierbar sein müssen, auch wenn sie Standardprodukte herstellt.

Abschließend soll auf einen charakteristischen Unterschied der Komponierung von IV-Operatoren einerseits und Fertigungsoperatoren andrerseits aufmerksam gemacht werden. Bei der Komponierung von IV-Operatoren reicht es im Prinzip aus, ein entsprechendes Programm zu schreiben (z.B. das Programm der Operation on_{21} in Block 5 zur Komponierung des f-op). Um dagegen einen Fertigungsoperator aus Bausteinoperatoren zu komponieren reicht es nicht aus, ein Steuerprogramm zu schreiben. Vielmehr können zusätzliche Maschinen, z.B. Montagemaschinen erforderlich sein. Ein Beispiel auf unterster Ebene ist der Fügeoperator in Bild 8.3, dem in Bild 8.1 eine Spaltegabel entspricht.

Die Aussage, dass für das Komponieren eines Operators nichts weiter erforderlich ist, als das Schreiben eines entsprechenden Programms, ist auch für IV-Operatoren nur "im Prinzip" richtig. In der Regel sind weitere Programme erforderlich, damit ein Computer die vorgeschriebene Kompositoperation tatsächlich ausführen kann. Im nächsten Kapitel werden wir uns überlegen, um welche Art von Programmen es sich dabei handelt und welche Aufgaben sie zu erfüllen haben.

19.5 Betriebssystem

19.5.1 Anwendungsprogramme und Organisationsprogramme

Die Feststellung, dass für die Ausführung einer Kompositoperation außer der Komponierungsvorschrift noch andere Programme erforderlich sind, überrascht insofern, als sie auf den Menschen offenbar nicht zutrifft. Ihm genügt es, eine Operationsvorschrift zu kennen, um die entsprechende Operation auszuführen. Wenn er weiß, wie multipliziert wird, sei es im Kopf, mit Papier und Stift oder mit Hilfe eines Taschenrechners, kann er das Produkt zweier Zahlen berechnen. Diese Aussage ist, genau genommen, falsch. Tatsächlich reicht das Wissen alleine nicht aus. Es müssen außerdem die notwendigen Hilfsmittel, die "*Betriebsmittel*" wie Papier, Stift oder Taschenrechner zur Verfügung stehen.

Beim Kopfrechnen benötigt der Mensch jedoch scheinbar keine Betriebsmittel. Das scheint ihm aber nur so, weil ihm die Verfügbarmachung der notwendigen "Betriebsmittel", die neuronalen Strukturen und Gedächtnisinhalte nicht zum Bewusstsein kommt. Sie stehen ihm "von selbst", gewissermaßen automatisch zur Verfügung. Von der internen Semantik einer Multiplikation hat er "keine Ahnung".

Im Falle des Computers erfolgt die Bereitstellung der Betriebsmittel "von selbst" in dem Sinne, dass der Computer selbst für sie verantwortlich ist. Dafür benötigt er Vorschriften, Programme. Diese Programme nennen wir **Organisationsprogramme**. Organisationsprogramme werden in der Literatur häufig Steuerprogramme genannt. Wir vermeiden diese Bezeichnung, da *jedes* Programm der Steuerung dient

und insofern ein "Steuerprogramm" ist. Der Eindeutigkeit halber nennen wir die Komponierungsvorschriften, von denen bisher ausschließlich die Rede war, **Anwendungsprogramme**, um anzudeuten, dass sie in der Regel von einem "Anwender" stammen, m.a.W. dass sie mit dem Ziel programmiert sind, den Computer auf dieses oder jenes Problem "anzuwenden", d.h. das Problem mit seiner Hilfe zu lösen.

In analoger Weise wird im Bereich der Produktion zwischen *Arbeits-* und *Organistationsmitteln* unterschieden. Arbeitsmittel führen die vom Anwender (Nutzer, Auftraggeber) geforderte Arbeit (Operation) aus; Organisationsmittel organisieren die Arbeit, indem sie dafür sorgen, dass für die Durchführung der Arbeiten die notwendigen Arbeitsmittel zur Verfügung stehen. Aus dieser Sicht könnte man Anwendungsprogramme auch als *Arbeitsprogramme* bezeichnen. Beide Bezeichnungen werden verwendet.

Wir fassen das Gesagte zusammen und präzisieren (zunächst ohne Bezugnahme auf den Begriff des Betriebssystems): *Ein* ***Anwendungsprogramm*** *ist eine Vorschrift für die Komponierung einer Anwendungsoperation, also einer von einem Anwender geforderten Operation. Ein* ***Organisationsprogramm*** *ist eine Vorschrift für die Zuweisung eines Betriebsmittels an eine* ***Anwendungsoperationsausführung***. Man beachte, dass ein Betriebsmittel nicht primär einem Operator bzw. einer Operation zugewiesen wird, sondern einer Operations*ausführung*, einem *Prozess*.

Eine Organisationsoperation kann recht umfangreich sein. Man erinnere sich an das Paging oder an das Lösen von Konflikten. Organisationsprogramme sind - ebenso wie die Anwendungsprogramme - Operationsvorschriften, die in einer höheren Programmiersprache formuliert und hierarchisch aufgebaut sein können. Alles, was über das Programmieren, Übersetzen, Speichern, Laden und Ausführen von Anwendungsprogrammen gesagt wurde, kann hier wiederholt werden.

Es wäre naheliegend, die Gesamtheit aller Organisationsprogramme, über die ein Computer verfügt, als dessen *Organisationssystem* zu bezeichnen. Statt dessen hat sich ein anderer Begriff durchgesetzt, der des *Betriebssystems.* Es besteht aber keine einheitliche Meinung darüber, wie das Betriebssystem zu definieren ist. Wenn man zum *Betriebs*system alle Programme rechnet, die den *Betrieb* des Computers ermöglichen, die also die Abarbeitung von Anwendungsprogrammen ermöglichen, so gehören dazu offensichtlich außer den Organisationsprogrammen auch die **peripheren Steuerprogramme**, die der Steuerung der peripheren Geräte dienen. Sie werden ebenso programmiert und verarbeitet, wie alle anderen Programme.

7 Wir vereinbaren: *Organisationsprogramme und periphere Steuerprogramme werden unter der Bezeichnung* ***Systemprogramme*** *zusammengefasst. Die Gesamtheit aller Systemprogramme bezeichnen wir als* ***Betriebssystem***. (Hier könnte ein vorgezogener Blick auf Bild 19.7 hilfreich sein. Der "Kern" wird später erklärt.)

Das Wort *Systemprogramm* "macht wenig Sinn" (ruft wenig Assoziationen hervor). Sinnvoller ist das Wort *Dienstprogramm*, denn sowohl die Organisationsprogramme als auch die peripheren Steuerprogramme stellen den Anwendungspro-

grammen ihre *Dienste* zur Verfügung. Präziser müsste man sagen: *Systemprozesse stellen Anwendungsprozessen ihre Dienste zur Verfügung*.

Terminologische Anmerkung. In der Literatur werden zuweilen unter der Bezeichnung *Dienstprogramm* eine Reihe von Programmen zusammengefasst, die sehr oft in Anwendungsprogrammen benötigt werden, so z.B. Suchprogramme und Sortierprogramme. Da derartige Programme häufig vom Hersteller mit den Organisationsprogrammen zu einem Programmpaket zusammengefasst und als solches verkauft werden, hat es sich eingebürgert, diese Art von Dienstprogrammen zum Betriebssystem zu zählen. Aus dem gleichen Grunde werden oft auch Übersetzerprogramme zum Betriebssystem gezählt. Wir schließen uns diesem Sprachgebrauch nicht an, sondern verwenden das Wort "Betriebssystem" in der oben definierten Bedeutung, wobei wir die Worte "Systemprogramm" und "Dienstprogramm" als Synonyme verwenden.

Nach dieser Anmerkung soll der Begriff des Dienstes im Zusammenhang mit dem Betriebssystem näher erläutert und zwischen zwei Klassen von Diensten unterschieden werden. Die Unterscheidung lässt sich "handgreiflich" am Beispiel von Fertigungsprozessen erfassen. Auch dort wird von *Diensten* gesprochen, wobei zwischen *Handlangerdiensten* und *Einrichtediensten* unterschieden wird. Die Handlangerdienste stellen Werkzeuge und Material bereit, die Einrichtedienste bereiten Werkzeuge und Material für einen bestimmten Fertigungsprozess vor. Beispielsweise wird eine Bohrmaschine auf die Bohrung bestimmter Löcher "eingerichtet". In Analogie dazu leisten im Falle des Betriebssystems die Organisationsprogramme die Handlangerdienste, sie geben dem Anwendungsprozess die erforderlichen Betriebsmittel "in die Hand". Die peripheren Steuerprogramme leisten die Einrichtedienste. Sie präparieren die *realen* ("nackten") Geräte zu "*virtuellen*" Geräten, die genau diejenigen Eigenschaften besitzen, die der Anwendungsprozess erwartet, z.B. die Eigenschaft, auf eine Adresse mit der Übergabe eines Datums zu reagieren, das auf einem der externen Speicher abgespeichert ist, oder einen Text in einer bestimmtem Schriftart auszudrucken.

Das Zusammenwirken von Anwendungs- und Systemprogrammen wird in Kap.19.5.5 anhand von Bild 19.7 erläutert. Der Leser werfe schon jetzt einen Blick auf Bild 19.7. Es stellt einerseits eine Verkürzung und andrerseits eine Erweiterung von Bild 19.6 dar. Die Blöcke 1, 4 und 5 sind nicht dargestellt, die Blöcke 2 und 3 sind nicht dekomponiert und die Blöcke 6 bis 10 sind hinzugefügt. Block 3 enthält wieder die ladbaren Anwendungsprogramme, Block 7 enthält die ladbaren peripheren Steuerprogramme und Block 8 die ladbaren Organisationsprogramme. Die zentrale Hardware (Block 2) ist um die periphere Hardware (Block 6) ergänzt.

19.5.2 Prozessbegriff und geteilte Nutzung von Hardwareoperatoren

In Kap.19.4 waren zwei charakteristische Probleme genannt worden, mit denen der Entwickler eines Betriebssystems konfrontiert ist, die ***geteilte*** *Nutzung von*

Betriebsmitteln und die *Nutzung* ***verteilter*** *Betriebsmittel*. Das eine wie das andere gehört offensichtlich zum Aufgabenbereich der Organisationsprogramme.

In diesem und dem folgenden Kapitel werden wir uns überlegen, welche Probleme bei der geteilten Nutzung von Betriebsmitteln auftreten können. Zuvor soll eine präzisierte Definition des Begriffs der *geteilten Nutzung* von Betriebsmitteln gegeben werden. *Ein Betriebsmittel heißt sequenziell geteilt genutzt oder kurz* ***sequenziell geteilt****, wenn zwei oder mehrere Prozesse im Wechsel seine Dienste in Anspruch nehmen*, m.a.W. wenn es jeweils einem der Prozesse für ein begrenztes Zeitintervall *zugeteilt* wird. Zu den Betriebsmitteln eines Prozesses gehören sämtliche Hardware- und Softwarebausteine, die erforderlich sind, damit der Prozess laufen kann, also die abzuarbeitenden Programme, die zu bearbeitenden Daten, die erforderlichen Speicherplätze und evtl. die erforderlichen E/A-Geräte.

Man beachte, dass Teilen im allgemeinen nicht unbedingt ein *sequenzielles* Teilen sein muss. Beispielsweise kann ein Speicher in verschiedene Bereiche *geteilt* werden, die verschiedenen Prozessen zugeteilt werden. Im Weiteren ist aber unter *geteilter Nutzung* stets *sequenziell geteilte Nutzung* zu verstehen. Dabei wird nicht das Betriebsmittel, sondern die Zeit seiner Nutzung geteilt. Darum sprechen die Informatiker von **Time sharing**.

Die Bereitstellung geteilter Betriebsmittel ist Aufgabe des Betriebssystems. Bei seinem Entwurf sind zwei Nebenbedingungen zu erfüllen; der Nutzer soll möglichst wenig mit den organisatorischen Maßnahmen belastet werden und die Programmabarbeitung soll möglichst *effizient* ablaufen, d.h. mit maximalem Nutzen für alle Beteiligten, was in erster Linie minimalen Aufwand an Zeit und Betriebsmitteln bedeutet. Angenommen, es ist eine Anzahl von Programmen abzuarbeiten, wozu ein einziger PC zur Verfügung steht. Dann ist das wichtigste Mittel der Effizienzerhöhung die Herabsetzung der mittleren Programmlaufzeit. *Die* ***Laufzeit*** *eines Programms ist das Zeitintervall vom Start bis zur Beendigung der Programmabarbeitung einschließlich der Resultatausgabe*.

Die Probleme, die sich hinter der geteilten Nutzung von Betriebsmitteln verbergen, sind teilweise so versteckt, dass sie erst im Laufe der Zeit erkannt worden sind. Manche Probleme mussten durch wiederholte, unerwartete Rechnerabstürze auf sich aufmerksam machen. Wir werden uns damit begnügen, die wichtigsten Probleme herauszuarbeiten, ohne auf die Lösungsmethoden genauer einzugehen. Dem interessierten Leser bleibt es überlassen, sich zu überlegen, wie man die einzelnen Probleme eventuell lösen könnte (es gibt eine Vielzahl von Möglichkeiten), oder sich in der Literatur kundig zu machen.[12]

12 Stellvertretend für die vielen einschlägigen Bücher seien die Monographie [Tanenbaum 94] und das Taschenbuch [Werner 95] genannt.

Wir wollen uns zunächst überlegen, welche Probleme bei der geteilten Nutzung von *Operatoren* auftreten können. Um sie leichter zu erkennen, legen wir uns folgende Fragen vor:

1. Welche Operatoren werden bei der Programmabarbeitung durch einen Einprozessorrechner eventuell geteilt genutzt?
2. Um welche Prozesse handelt es sich im Einzelnen, denen die CPU als Betriebsmittel zugewiesen werden muss?
3. Welche notwendigen und welche wünschenswerten Ziele sollen erreicht werden?

Es sei noch einmal betont, dass es nicht Operatoren sind und auch nicht Operationen, von denen Betriebsmittel angefordert werden, sondern Operations*ausführungen*, d.h. *Prozesse*. Die Entwickler von Betriebssystemen müssen also nicht nur operationsorientiert, sondern vor allem *prozessorientiert* denken. Das hat dazu geführt, dass der Prozessbegriff im Laufe der Jahre immer mehr zum zentralen Begriff des Entwurfs und der Programmierung von Betriebssystemen geworden ist und dass er sich dementsprechend immer genauer den Bedürfnissen der Systemprogrammierung angepasst hat. Er ist spezieller und gleichzeitig abstrakter geworden.

Wir haben den Prozessbegriff in Kap.8.1 sehr allgemein als Synonym zu *Operationsausführung* eingeführt: *Ein* ***Prozess*** *ist die Ausführung einer Operation durch einen Operator*. Wenn ein Computer die Rolle des Operators spielt, ist diese Definition gleichbedeutend mit der spezielleren Definition: *Ein* ***Prozess*** *ist die Ausführung eines Programms*. Im Laufe längerer Programmiertätigkeit nimmt der Prozessbegriff für den Programmierer eine allgemeinere, abstraktere Bedeutung an. Für ihn ist ein Prozess ein Abstraktum, das einen (computerverständlichen) Namen haben muss und dem Betriebsmittel zugewiesen werden müssen. Das führt zu einer Definition, die wir aus [Tanenbaum 94] zitieren: *"Ein* ***Prozess*** *ist die Abstraktion eines in Ausführung befindlichen Programms"*. Vom physischen Vorgang der Ausführung wird abstrahiert.

Man kann noch weiter gehen und von jeglicher externen Semantik abstrahieren, indem man einen Prozess dadurch "*definiert*", dass man die zeitliche Folge der computerinternen Zustände angibt, die der Prozess der Reihe nach auslöst; das sind zum einen die CPU-Zustände (ein CPU-Zustand ist der Inhalt aller Register der CPU) und zum anderen die Zustände (Inhalte) der Speicher*umgebung*, d.h. der Speicherplätze außerhalb der CPU. *Die Gesamtheit der Adressen aller Speicherplätze, die einem Prozess außerhalb der CPU zugewiesen sind, wird* **Adressraum** *des Prozesses genannt*. Man sagt: Ein Prozess "läuft in seinem Adressraum ab". Der Adressraum eines Prozesses ist ein Unterraum des gesamten Adressraumes, d.h. der Gesamtheit aller Adressen, die dem Computer, auf dem der Prozess läuft, zur Verfügung stehen.[13]

---

13 Oft wird zwischen *physischem* und *logischem* Adressraum unterschieden. Für unsere Überlegungen ist diese Unterscheidung nicht notwendig. Wir gehen von der Vorstellung aus, dass zwischen der Menge aller physischen Speicherplätze und der Menge aller Adressen eine eineindeutige Abbildung existiert.

Den Zustand (Inhalt) eines Adress-Unterraumes nennen wir ***partiellen Speicherzustand***.

Auf diesem Abstraktionsniveau kann folgende Begriffsbestimmung geben werden: *Ein* ***Prozess*** *ist eine Folge von CPU-Zuständen und partiellen Speicherzuständen, die sich während der Abarbeitung eines Programms einstellen*. Ein partieller Speicherzustand eines Prozesses ist die *kausale* Folge der ihm vorangehenden CPU-Zustände des Prozesses. Man beachte, dass die soeben gegebene Prozessdefinition genau genommen keine *Definition*, sondern eine *internsemantische Beschreibung* ist. Sie gibt die *computerinterne Bedeutung* des Prozessbegriffs wieder, befreit von jeder externen Semantik. Die Zustandsfolge, aus der ein Prozess besteht, muss nicht zeitlich zusammenhängend (keine lückenlose Folge von Takten) sein. Sie überspringt alle Zeitpunkte (Takte), in denen der Prozess unterbrochen ist (nicht läuft).

Wenn man in obigem, kursiv gedruckten Satz die Wortfolge "Ein Prozess ist" im Sinne von "Ein Prozess ist definiert als" versteht (wie es in der Mathematik üblich ist), wird die Beschreibung des Prozesses zur Definition des Prozessbegriffs. Ein noch allgemeinerer Prozessbegriff bezieht die Zustände sämtlicher Betriebsmittel ein, auch die der EA-Geräte: *Ein* ***Prozess*** *ist die zeitliche Folge von* ***Betriebsmittelzuständen****, die sich während der Abarbeitung eines Programms einstellen.*

Wenn ein hierarchisch strukturiertes Programm (z.B. ein Hauptprogramm mit geschachtelten Unterprogrammrufen) ausgeführt wird, überträgt sich die hierarchische Struktur auf den Prozess. Man spricht dann in Analogie zur *Operatorenhierarchie* von **Prozesshierarchie**.

Einem physikalisch denkenden Menschen mag es befremdlich erscheinen, dass in der Beschreibung eines Prozesses als Zustandsfolge die eigentlichen *physikalischen* Prozesse, aus denen letztlich jeder informationelle Prozess besteht, nämlich die *Übergangsprozesse* in den elektronischen Bauelementen, insbesondere in der ALU (bzw. in den ALUs), überhaupt nicht in Erscheinung treten. Sie werden negiert. Das aber ist der Kern der *kausaldiskreten* Prozessbeschreibung [8.4], die ausschließlich und auf allen Komponierungsebenen anzuwenden ist, nicht nur auf der Ebene der KR-Netze, sondern auch auf der relativ hohen Ebene, auf der das Betriebssystem agiert.

Nach dieser begriffliche Ergänzung wenden wir uns den drei oben gestellten Fragen zu und schaffen uns einen Überblick über die zu erwartenden Probleme.

Zu Frage 1: *Welche Operatoren werden bei der Programmabarbeitung durch einen Einprozessorrechner eventuell geteilt genutzt*? Die zu teilenden Operatoren können reale, aber auch sprachliche Operatoren sein. Letzteres ist der Fall, wenn ein Programm von verschiedenen Hauptprogrammen als Unterprogramm aufgerufen wird. Die Behandlung der geteilten Nutzung von Programmen verschieben wir auf Kap.19.5.3, wo sie sich besser in den allgemeinen Gedankengang einfügt. Im Augenblick interessieren wir uns nur für reale Operatoren.

Zu den *realen* Operatoren (Hardwareoperatoren), die von Prozessen in Einprozessorrechnern benötigt werden, gehören die CPU und die peripheren Geräte. Der Hauptspeicher, der auch als realer Operator aufgefasst werden kann, wird hier nicht betrachtet. In der Regel kann ein peripheres Gerät zu einem bestimmten Zeitpunkt nur einen einzigen Prozess bedienen. Wenn zwei Prozesse einen realen Operator gleichzeitig anfordern, kommt es zu einem Konflikt, den das Betriebssystem lösen muss. Man kann auch sagen, dass die CPU ihn lösen muss, indem sie das entsprechende Organisationsprogramm ausführt. In dieser Ausdrucksweise kann eine Zirkularität liegen, nämlich dann, wenn die CPU einen Konflikt lösen muss, in den sie selber geraten ist.

Soweit periphere Geräte über eigene reale Steueroperatoren verfügen, können sie parallel miteinander und parallel mit der CPU arbeiten. Ein wesentlicher Unterschied zwischen CPU und peripheren Geräten ist die Arbeitsgeschwindigkeit. Wegen der Langsamkeit der peripheren Geräte müssen sie für relativ lange Zeitintervalle einem Prozess zugewiesen werden. Das wirkt sich maßgeblich auf die Strategie der Betriebsmittelzuweisung aus.

Zu Frage 2: *Um welche Prozesse handelt es sich im Einzelnen, denen die CPU oder auch andere Betriebsmittel zugewiesen werden müssen*? Jeder Prozess, jeder Anwendungs- und jeder Organisationsprozess benötigt Betriebsmittel, zumindest benötigt er die CPU. Wenn mehrere Prozesse gleichzeitig die CPU benötigen, muss einem von ihnen der Vorrang gewährt werden. Angenommen, ein Anwendungsprozess benötigt Daten aus dem Plattenspeicher. Zu diesem Zweck muss die CPU ein oder mehrere Organisationsprogramme ausführen, z.B. ein Paging-Programm. Um das erledigen zu können, muss als erstes eine sog. **Unterbrechung** eingeleitet werden, denn der Anwendungsprozess, der die Daten benötigt, muss zugunsten eines Organisationsprozesses *unterbrochen* werden. Der Organisationsprozess hat vor dem Anwendungsprozess das Vorrecht auf die CPU, er ist der *privilegierte* Prozess. Es findet eine "*zeitlich verschachtelte*" Abarbeitung verschiedener Programme statt, die das Betriebssystem organisieren muss.

Damit die CPU nach Erhalt der benötigten Daten die Ausführung des unterbrochenen Anwendungsprogramms fortsetzen kann, muss im Zeitpunkt der Unterbrechung der *aktuelle CPU-Zustand* notiert werden; die Inhalte aller CPU-Register müssen "*gerettet*", d.h. an einem geeigneten Ort aufbewahrt werden. Dafür bietet sich ein *Stack* (Kellerspeicher) an. Es tritt nämlich häufig der Fall ein, dass ein Prozess, wir nennen ihn Prozess B, der einen Prozess A unterbrochen hat, selber von einem Prozess C unterbrochen wird. Die Bereitstellung der geretteten Daten nach Beendigung der jeweiligen unterbrechenden Prozesse muss in umgekehrter Reihenfolge wie ihre Abspeicherung erfolgen, d.h. zuerst die von B, dann die von A. Diese Situation ist uns vom verschachtelten Unterprogrammruf her bekannt [16.16]. Bei jedem Unterprogrammruf wird die Abarbeitung des rufenden Programms unterbrochen. Die Verwendung eines Stacks ist deswegen so vorteilhaft, weil der Zugriff auf seinen obersten Speicherplatz unmittelbar, ohne Adressierung des Platzes, erfolgt,

sodass die Wiederherstellung des alten CPU-Zustandes nur wenig Zeit in Anspruch nimmt. Für die Speicherung aller Daten, die für die Organisation der Ausführung zeitlich verschachtelter Prozesse erforderlich sind, wird üblicherweise ein spezieller Speicherbereich eingerichtet, der Prozesssteuerblock genannt wird, abgekürzt PCB (Process Control Block).

Vorrechte von Prozessen sind nicht nur hinsichtlich der CPU, sondern auch hinsichtlich anderer Betriebsmittel wie peripherer Speicher oder E/A-Geräte zu berücksichtigen. Beispielsweise ist es sicher zweckmäßig, einem Prozess A das Vorrecht auf einen peripheren Speicher vor einem Prozess B einzuräumen, wenn Prozess B zunächst auch ohne Zugriff auf den peripheren Speicher fortgesetzt werden kann, Prozess A hingegen nicht. Wenn beide Prozesse nicht fortgesetzt werden können, kann es zu einer Blockierung (Deadlock) kommen (vgl. Bild 8.4a; man erinnere sich an die Beispiele der beiden Schraubenschlüssel in Kap.8.2.2. [8.8]).

Zu Frage 3: *Welche notwendigen und welche wünschenswerten Ziele sollen erreicht werden*? Bisher war vorwiegend von solchen Prozessunterbrechungen die Rede, die stattfinden, um ein Betriebsmittel bereitzustellen, ohne welches der laufende Arbeitsprozess nicht fortgesetzt werden kann. Derartige Unterbrechungen von Arbeitsprozessen und das Einschieben ("Einschachteln") von Organisationsprozessen lassen sich nicht umgehen, es sei denn, sämtliche erforderlichen Betriebsmittel sind dem Prozess schon vor seinem Start zugewiesen worden.

Daneben gibt es Situationen, in denen Prozessunterbrechungen nicht unbedingt notwendig, aber zweckmäßig sind, beispielsweise um die Laufzeit eines Programms herabzusetzen und damit die Effizienz des Computers zu erhöhen. Eine oft sehr effektive Methode der Laufzeitverkürzung durch das Betriebssystem ist die Parallelisierung schneller und langsamer Prozesse.

Infolge der Langsamkeit der peripheren Geräte kann sich die Laufzeit eines Programms erheblich verlängern. Es liegt der Gedanke nahe, periphere Prozesse, z.B. Ein- und Ausgaben oder das Zugreifen auf periphere Speicher, in Zeitintervallen erledigen zu lassen, in denen die CPU mit anderen Aufgaben beschäftigt ist. Zur Verwirklichung der Idee müssen die peripheren Geräte, wie bereits erwähnt, mit eigenen realen Steueroperatoren (Spezialprozessoren) ausgestattet werden, welche die Ausführung eines Teils der peripheren Steuerprogramme übernehmen. Das Vorgehen ähnelt der in Kap.19.4 [3] besprochenen Firmwareaufteilung zwischen Schicht 5 (dem steuernden Prozessor) und Schicht 5′ (den NC-Maschinen). Unter diesem Aspekt entsprechen die peripheren Geräte den NC-Maschinen. Je stärker sich periphere und CPU-Prozesse gegenseitig überlappen, m.a.W. je vollständiger sie *parallelisiert* sind, umso größer ist der Zeitgewinn. Das bedeutet beispielsweise, dass mit dem Holen von Daten vom Plattenspeicher nicht unbedingt gewartet wird, bis sie gebraucht werden, oder dass mit der Ausgabe von Resultaten nicht unbedingt bis zum Ende der Programmabarbeitung durch die CPU gewartet wird, sondern dass Datentransporte ausgeführt werden, sobald die Möglichkeit dazu besteht. Wieweit

periphere Prozesse vorgezogen werden können, hängt vom Arbeitsprogramm ab. Wieweit sie tatsächlich vorgezogen werden, hängt vom Betriebssystem ab.

Eine andere Möglichkeit der Effizienzerhöhung liegt in der zeitlich verschachtelten Abarbeitung zweier oder mehrerer *Anwendungs*programme. Angenommen, es sollen zwei Programme ausgeführt werden. Eins der beiden Programme sei *CPU-intensiv* (es benötige viel CPU-Zeit), das andere sei *E/A-intensiv* (es benötige viel Zeit für Ein- und Ausgaben). Dann ist es zweckmäßig, CPU- und EA-Prozesse so zu verschachteln, dass maximale Parallelität der Arbeit der CPU und der E/A-Prozesse resultiert.

Eine etwas andere Situation der verschachtelten Ausführung von Programmen liegt vor, wenn zwei Programme mit unterschiedlichem Vorrang gleichzeitig abgearbeitet werden. Wir nennen das Programm mit höherer Proiritat *Vordergrundprogramm*, das andere *Hintergrundprogramm*. Letzteres wird immer dann aktiviert, wenn das Vordergrundprogramm auf Betriebsmittel wartet, z.B. auf Eingabedaten. Den Wechsel von dem einen zum anderen Programm muss das Betriebssystem organisieren. Dabei ist es eventuell nicht damit getan, den jeweiligen aktuellen CPU-Zustand herzustellen. Wenn die Programme zu lang sind, um im Arbeitsspeicher abgelegt werden zu können (bei großen Hintergrundprogrammen ist das oft der Fall), muss bei jedem Wechsel das zu unterbrechende Programm aus dem Arbeitspeicher ausgeladen und das zu aktivierende Programm (ein)geladen werden. Wenn das Umladen *vor* der Unterbrechung vorgenommen wird, was möglich sein kann, evtl. teilweise, wird Laufzeit gespart.

8

Ein Arbeitsregime, bei dem das Betriebssystem die Ausführung mehrerer Programme stückweise sequenziell (zeitlich geschachtelt) organisiert, heißt **Multitaskregime** oder *Multitasking* (task = Auftrag; in der Betriebssystem-Fachsprache wird das einen Prozess steuernde Programm auch *Task* genannt). Die Teile, in die der Ausführungsprozess eines Programms (einer Task) zerteilt wird, heißen **Thread**. Wenn das Hin- und Herspringen zwischen den Prozessen schnell erfolgt, hat es für den Nutzer den Anschein, als würden die betreffenden Programme parallel ausgeführt. Da es sich aber nicht um echte Parallelität handelt, hat sich das Wort “*Quasiparallelität*” eingebürgert und man spricht von quasiparalleler Programmausführung und von quasiparallelen Prozessen.

Beim Entwurf eines Betriebssystems, welches das Multitaskregime unterstützt, muss der Entwickler dafür Sorge tragen, dass quasiparallele Prozesse sich nicht gegenseitig stören. Mit gegenseitigen Störungen zweier Prozesse muss gerechnet werden, wenn sich ihre Adressräume überlappen. Dann können nämlich die gemeinsamen Speicherplätze von einem Prozess beschrieben und vom anderen gelesen werden. In diesem Fall sprechen wir von **direkter Prozesskommunikation**.

Direkte Kommunikation kann erwünscht oder unerwünscht, d.h. störend sein. Über gemeinsame Speicherplätze kann ein Prozess einen anderen eventuell unkontrolliert beeinflussen und unbeabsichtigte Wirkungen, sogenannte *Seiteneffekte* hervorrufen. Erwünschte Kommunikation zu ermöglichen und gleichzeitig unerwünsch-

te Kommunikation zu verhindern, ist ein Problem, dass die Informatiker jahrelang beschäftigt hat. Es musste ein Weg gefunden werden, Prozesse so voneinander *abzukapseln*, dass sie kommunizieren, sich aber nicht stören können. Im Ringen um eine brauchbare Lösung hat sich der Begriff "*Objekt*" herausgebildet. In Kap.18.2 [18.5] war er im Zusammenhang mit der Evolution der Programmiersprachen bereits genannt worden. Seine Rolle für die Softwaretechnologie, die dort nur angedeutet worden war, wird in Kap. 19.5.4 deutlich zutage treten, wenn wir nach einem Kompromiss zwischen erwünschter und unerwünschter Prozesskommunikation suchen werden. Partielle Lösungen dieses Problems sind schon früher vorgeschlagen worden, bevor das "*informatische Objekt*" erfunden worden war. Ihnen wenden wir uns nun zu.

19.5.3 Geteilte Nutzung von Speicherplätzen, Daten und Programmen

Ein Speicherplatz, der nur gelesen, aber nicht überschrieben werden kann, darf ohne besondere Vorsichtsmaßnahmen einem Prozess oder auch mehreren Prozessen gleichzeitig zugewiesen werden. Der Inhalt des Speicherplatzes spielt die Rolle einer *Konstanten*. Wenn der Speicherplatz dagegen eine *Variable* enthält, wenn also der Prozess, dem der Speicherplatz als Betriebsmittel zugewiesen ist, dessen Inhalt überschreiben kann, ist Vorsicht geboten. Es werden spezielle organisatorische Maßnahmen notwendig, falls der Speicherplatz verschiedenen Prozessen zugewiesen, d.h. *geteilt genutzt* wird. Es liegt dann ein spezieller Fall von geteilter Betriebsmittelnutzung vor.

Zum Betriebsmittel "Speicherplatz" gehören genau genommen auch die Register einer CPU. Mit der geteilten Nutzung der CPU werden auch sie geteilt genutzt. Dieser Sonderfall der geteilten Speicherplatznutzung ist bereits in Kap.19.5.2 behandelt worden. Im Weiteren betrachten wir den Fall, dass sich Prozesse in Speicherplätze teilen, die als *Variablen*plätze benutzt werden, die aber *nicht* zur CPU gehören. Offensichtlich liegt dieser Fall immer dann vor, wenn sich die Adressräume zweier Prozesse überlappen, d.h. wenn die beiden Adressräume gemeinsame Adressen (Variablenplätze) enthalten. Ist dies der Fall, können die Prozesse über die gemeinsamen Speicherplätze *direkt* miteinander kommunizieren (Daten austauschen); sie können sich aber auch stören. Bevor auf die Probleme eingegangen wird, die mit der Verwirklichung direkter, aber ungestörter Prozesskommunikation verknüpft sind, soll die in Kap.19.5.2 zurückgestellte Behandlung der geteilten Nutzung von *Programmen* nachgeholt werden.

Wenn zwei Programme keine gemeinsamen Variablen verwenden, wenn sich die Adressräume ihrer Ausführungsprozesse also nicht überlappen, können sie im Multitaskregime ausgeführt werden, ohne dass besondere Rettungsmaßnahmen getroffen werden müssen, abgesehen von der Rettung des CPU-Zustandes. Die Ausführungsprozesse können "nebeneinander herlaufen", ohne sich zu stören. Das ist in der Regel gemeint, wenn Programme als *nebenläufig* bezeichnet werden.

Ein Sonderfall der Adressraumüberschneidung liegt vor, wenn ein und dasselbe Programm mehrmals ausgeführt wird. Wenn die entsprechenden Prozesse im Multitaskregime ausgeführt werden sollen, sind spezielle Maßnahmen erforderlich. Entweder muss bei jeder Unterbrechung der Inhalt des gesamten Adressraumes gerettet werden oder es muss dafür gesorgt werden, dass die (physischen) Adressräume disjunkt sind (keine gemeinsamen Adressen besitzen), z.B. dadurch, dass bei jedem Start des Programms jede Variable ihren prozessspezifischen Namen erhält, sodass ihr ein eigener Speicherplatz zugewiesen wird. Ein Programm, das wiederholt gestartet und quasiparallel ausgeführt werden kann, ohne dass die einzelnen Abarbeitungsprozesse sich gegenseitig stören, wird **reentrant** oder *reenterabel* genannt. Ein reentrantes Programm kann ohne zusätzliche Rettungsmaßnahmen im Multitaskregime "quasiparallel zu sich selbst" ausgeführt werden.

Nach diesem Einschub kehren wir zur Prozesskommunikation zurück. Das Problem, das gelöst werden muss, soll an einem Beispiel erläutert werden. Angenommen, von einem Bankkonto werden per Computer zwei Abbuchungen durchgeführt, einmal 100 und einmal 200 EUR und zwar durch zwei verschiedene Programme. Wenn das eine Programm gestartet wird, beginnt der betreffende Abbuchungsprozess, den wir mit A bezeichnen. Prozess A liest zuerst den alten *Kontostand*, der 1000 EUR betrage. Davon subtrahiert er 100 EUR. Bevor er den neuen Kontostand von 900 EUR ausgibt, wird er unterbrochen, und Prozess B beginnt die Abbuchung von 200 EUR. Der alte Kontostand beträgt nach wie vor 1000 EUR, und Prozess B berechnet als neuen Kontostand 800 EUR. Je nachdem, welcher der beiden Prozesse seine Ausgabe als letzter tätigt, beträgt der neue *Kontostand* nach Beendigung beider Prozesse entweder 800 oder 900 EUR, während der richtige Kontostand 700 EUR ist.

Die Ursache des Fehlers liegt darin, dass ein und dieselbe Variable von zwei Prozessen zeitüberlappend bearbeitet wird. Es muss eingeräumt werden, dass das obige Beispiel nicht sehr realistisch ist, denn Buchungsvorgänge werden kaum von einem Einprozessorrechner, sondern eher vom Informationssystem einer Bank oder Sparkasse durchgeführt, das viele Computer und Dateien umfassen kann, also ein *verteiltes System* darstellt. Für verteilte Systeme ist das Problem noch charakteristischer. In jedem Falle muss garantiert sein, dass während des sog. *kritischen Zeitintervalls* vom Moment des Lesens eines Speicherplatzes bis zum Moment des Überschreibens kein anderer Prozess auf den Speicherplatz zugreifen kann. Diesem kritischen Zeitintervall entspricht ein **kritischer Programmabschnitt** desjenigen Programms, das sich während des Prozesses in Ausführung befindet.

9

Beim Eintritt eines Prozesses in einen kritischen Programmabschnitt mit der *öffentlichen* (d.h. auch von anderen Prozessen benutzbaren) Variablen a muss der Speicherplatz von a zeitweilig *reserviert* (*privatisiert*) und der Zugriff anderer Prozesse ausgeschlossen werden. Dafür sind eine ganze Reihe von Mechanismen entwickelt worden (siehe z.B. in [Tanenbaum 94]), deren Implementierung zum Teil Aufgabe des Anwendungsprogrammierers ist. Er muss die kritischen Programmab-

schnitte erkennen und markieren. In Kap. 19.5.4 werden wir eine Methode kennen lernen, die weniger zu Lasten des Anwenderprogrammieres und mehr zu Lasten des Systemprogrammierers (des Programmierers des Betriebssystems) geht.

Der Umgang mit kritischen Programmabschnitten birgt eine neue Gefahr in sich. Es kann nämlich der Fall eintreten, dass zwei Prozesse A und B sich gegenseitig ausschließen. Angenommen, beide Prozesse benutzen die gemeinsamen Variablen a und b und zu einem bestimmten Zeitpunkt befinden sich beide Programme in einem kritischen Abschnitt des jeweiligen Programms, wobei Prozess A die Variable a und Prozess B die Variable b für sich reserviert hat. Wenn nun Prozess A die Variable b benötigt, muss er warten, bis Prozess B seinen kritischen Bereich verlassen hat. Wenn dieser seinerseits die Variable a benötigt, um seinen kritischen Bereich verlassen zu können, blockieren sich die Prozesse gegenseitig. Es liegt ein *Deadlock* vor.

Mit derartigen Situationen hatten uns bereits in Kap.8.2.2 beschäftigt und Blokkierungssituationen ganz unterschiedlicher Natur zusammengestellt (Schraubenschlüssel, eingleisige Fahrbahn u.ä.m. [8.8]). Auch in informationellen Systemen können unterschiedliche Umstände Deadlocks verursachen, nicht nur der gegenseitige Entzug von Speicherplätzen, sondern auch der Entzug von anderen Betriebsmitteln, z.B. von E/A-Geräten oder von Leitungsstrecken (in Analogie zum Brükkenbeispiel [8.8]).

Die Bemühungen der Entwickler, den Nutzer von der Berücksichtigung kritischer Abschnitte zu befreien, haben zum Konzept der *Transaktion* geführt. *Ein Prozess heißt* ***Transaktion****, wenn unerwünschte Wechselwirkungen mit anderen Prozessen ausgeschlossen sind.* Da unerwünschte Wechselwirkungen nicht immer vorhersehbar sind, muss gewährleistet werden, das bei ihrem Eintreten die beteiligten Prozesse abgebrochen werden und der Zustand, der vor ihrem Start bestand, wiederhergestellt wird.

10

19.5.4 Verteilte Systeme

Das Gebiet der verteilten Systeme befindet sich gegenwärtig noch in seiner Entwicklungsphase. Ständig werden neue Methoden und Algorithmen erfunden, um den Umgang und die Nutzung verteilter Systeme einfacher, sicherer und billiger zu machen. Dabei unterliegt der Begriff des verteilten Systems selber einer Entwicklung. Die meisten Begriffsbestimmungen gehen davon aus, dass ein verteiltes System hardwaremäßig aus mehreren miteinander verbundenen Prozessoren und Speichern besteht. Wir nehmen diese Charakterisierung als Definition und vereinbaren: *Die Bezeichnungen* ***Verteiltes System*** *und* ***PS-Netz*** *verwenden wir als Synonyme.*

Nach dieser Vereinbarung kann die Behandlung verteilter Systeme unmittelbar an Kap.19.3 anknüpfen. Um welche konkreten Systeme es sich handeln kann, zeigt folgende Liste, in der einige Beispiele mehr oder weniger zufällig aufgeführt sind. Sie gliedern sich in zwei Gruppen.

1. Mehrprozessorrechner (virtueller Einprozessorrechner)
   - Schachcomputer

- Computerarray
- Mathematiksystem

2. Rechnernetze
- Rechnernetz einer Universität
- Platzbuchungssystem
- Informationssystem einer Bank
- Informationssystem einer automatischen Fabrik
- Informationssystem eines Betriebes mit Telearbeit
- Internet

Kommentar: Ein informationelles System, das die Verarbeitung der in einer Einrichtung anfallenden Informationen durchführt, wird oft als **Informationssystem** der Einrichtung bezeichnet. Es enthält meistens einen nicht automatisierten und einen automatisierten Teil. In obiger Liste ist mit *Informationssystem* jeweils der automatisierte Teil gemeint. Uns interessiert, welche Probleme beim Entwurf und bei der Nutzung eines verteilten Systems auftreten können.

Die Liste der Beispiele gibt eine Vorstellung hinsichtlich des Umfangs der Probleme und ihrer Abhängigkeit vom Gebiet und von der Art des Einsatzes, beispielsweise von der Anzahl der Nutzer und von der Länge der Übertragungswege. Wir werden die Probleme nur punktuell behandeln.

Mehrprozessorrechner. Hier wäre alles zu wiederholen, was in Kap.19.3 über PS-Netze und Prozessorhierarchien unter dem Aspekt der Leistungssteigerung durch Einsatz mehrerer Prozessoren gesagt worden ist. Zu diesem Zweck können aus Prozessoren und Speichern Netze und Hierarchien aufgebaut werden. Entsprechende PS-Architekturen sind in den Bildern 19.3 und 19.5 (erste Interpretation) dargestellt. Den Unterschied zwischen der Arbeitsweise eines Einprozessorrechners und der eines Mehrprozessorrechners hatten wir uns in Kap.19.3 anhand der Bilder 19.3a und 19.4 klargemacht und gesehen, wie im Mehrprozessorrechner das imperative Paradigma und das Netzparadigma zusammenwirken. Der Einsatz von Mehrprozessorrechnern zum Zwecke der Leistungssteigerung ist immer dann angebracht, wenn eine Operation in mehrere voneinander unabhängige (*nebenläufige*) Teiloperationen zerlegt werden kann. Drei Beispiele sollen das Vorgehen illustrieren.

1. Schachcomputer. Durch Verwendung mehrerer Prozessoren lässt sich die Rechengeschwindigkeit eines Schachcomputers dadurch erhöhen, dass mehrere Spielstellungen gleichzeitig analysiert, d.h. "in Gedanken" weitergespielt werden (siehe Kap.17.3). Die Menge der (zu einem bestimmten Zeitpunkt) zu analysierenden Stellungen könnte auf die Prozessoren aufgeteilt werden, sodass die erreichbare Tiefe der Analyse (die Anzahl der vorausspielbaren Züge) etwa proportional mit der Anzahl der Prozessoren zunimmt. Die tatsächliche Leistungserhöhung ist verständlicherweise niedriger, denn es ist einiger organisatorischer Aufwand erforderlich, um zu sichern, dass die Prozessoren richtig miteinander kooperieren.

2. Partielle Differenzialgleichungen. In Kap.19.2.3 war die Modellierung von Nahwirkungsphänomenen und die numerische Lösung partieller Differenzialglei-

chungen mittels eines ALU-Array behandelt worden. Ersetzt man die ALUs durch Prozessoren oder durch kleine Computer (Prozessoren mit eigenen kleinen Arbeitsspeichern) entsteht ein *Prozessor-* bzw. *Computerarray*, und es ergeben sich neue Möglichkeiten, derartige numerische Rechnungen durchzuführen. Im Grenzfall wäre jedem Gitterpunkt ein kleiner Computer zuzuordnen, sodass ein Computerarray entsteht. Ein spezieller Computer, der als "*Knotenoperator*" in gitterartigen Strukturen (in gitterartigen homogenen Operatorennetzen) entwickelt worden ist, wird unter der Bezeichnung *Transputer* vertrieben.

3. Mathematiksystem. In den bisherigen Beispielen führen die beteiligten Prozessoren bzw. Computer in der Regel ähnliche oder sogar identische Programme aus (SIMD-Prinzip). Das vereinfacht den Entwurf des betreffenden Mehrprozessorrechners. Mehr Aufwand kann notwendig sein, wenn die einzelnen Prozessoren unterschiedliche Aufgaben zu lösen haben (MIMD-Prinzip), beispielsweise wenn das oben erwähnte Mathematiksystem auf einem Mehrprozessorrechner läuft und die einzelnen Prozessoren jeweils auf spezielle mathematische Operationen spezialisiert sind. Bei der Ausführung größerer Rechnungen könnten voneinander unabhängige Berechnungsschritte parallel und zudem von "Spezialisten" ausgeführt werden.

Die angeführten Beispiele verdeutlichen den Begriff des Mehrprozessorrechners als Vernetzung mehrerer Prozessoren zum Zwecke der Ausführung einer Kompositoperation, wobei die Bausteinoperationen von verschiedenen Prozessoren ausgeführt werden. Voraussetzung für ein fehlerfreies Arbeiten eines Mehrprozessorrechners ist, dass die verschiedenen Prozesse, die in ihm ablaufen, in der gewünschten Weise miteinander kooperieren, ohne sich gegenseitig zu stören. Eine ausreichende gegenseitige *Abkapselung* der Prozesse ist notwendig.

Rechnernetze.. Sieht man sich die Beispiele der zweiten Gruppe in obiger Liste an, stellt man fest, dass der Aspekt der Komponierung von Operatoren mehr in den Hintergrund rückt und der Aspekt der Kooperation weiter an Bedeutung gewinnt. Ein Rechnernetz dient in erster Linie nicht der Komponierung von Operatoren, sondern der Kommunikation, beispielsweise im Zuge von Recherchen oder anderen Dienstleistung, wobei in dem Netz viele Kommunikationsprozesse gleichzeitig ablaufen können. Der Leser könnte die Frage aufwerfen, über welche ganz konkrete Leitung (über welchen "Draht") ein Computer an ein Netz angeschlossen wird. Die Antwort lautet: Über die Leitung für die Ein- und Ausgaben, also über den Halbkommutator HK in Bild 13.7 und weiter über eine Leitung "nach draußen", beispielsweise an den Telefonstecker, über den der Computer an das Telefonnetz angeschlossen wird. Weiter könnte gefragt werden, wie Verbindungen zwischen den "Abonnenten" eines Rechnernetzes hergestellt werden. Wir werden darauf später kurz eingehen.[14]und wenden uns jetzt demjenigen Problem zu, auf das wir in Kap.19.5.2 bei der Organisation des Multitaskregimes [8] und in Kap.19.5.3 bei der geteilten

14 Ausführliche Informationen findet der Leser z.B. in [Tanenbaum 98].

Nutzung von Speicherplätzen gestoßen sind, der *störenden* Kommunikation zwischen Prozessen.

Das Problem der Störung zwischen Prozessen spielt in verteilten Systemen, in denen viele Prozesse parallel oder quasiparallel ablaufen können, eine noch größere Rolle als im Multitaskregime des Einprozessorcomputers. Zwei Vorgehensweisen im Kampf mit störender Kommunikation waren besprochen worden, das Markieren *kritischer Programmabschnitte* [9] durch den Anwendungsprogrammierer und das *Transitionsprinzip* [10], das darin besteht, dass eine eingetretene Störung ungeschehen gemacht wird. Die erste Methode hat den Vorteil, dass Störungen von vornherein vermieden werden, die zweite hat den Vorteil, dass der Anwendungsprogrammierer nicht belastet wird.

Um beide Vorteile miteinander zu verbinden, könnte man auf die Idee kommen, Prozesse gegen Störungen dadurch zu schützen, dass man sie gegeneinander "*abkapselt*", indem man direkte Kommunikationen zwischen ihnen ausschließt. Man hätte dafür zu sorgen, dass ihre Adressräume disjunkt sind (keine gemeinsamen Adressen enthalten). Doch das scheint unmöglich zu sein. Denn ohne gemeinsame Speicherplätze ist nicht nur unerwünschte, sondern auch erwünschte direkte Kommunikation unmöglich. Die Abkapselung darf also keine absolute, sie muss eine "*relative*" sein, d.h. die "Kapsel" muss so durchlässig wie nötig und so undurchlässig wie möglich sein. Die im Weiteren zu besprechende Lösungsidee liegt in der **indirekten Prozesskommunikation**, d.h. in der Kommunikation über einen dritten Prozess, dessen Adressraum sich mit den Adressräumen der gegeneinander abzukapselnden Prozesse überlappt. Diese Idee wird weiter unten anhand zweier Beispiele erläutert. Zuvor soll ein Blick auf das Endergebnis geworfen werden, zu dem die Entwicklung geführt hat.

Die Realisierung der Idee der indirekten Kommunikation ist im Laufe der Jahre immer weiter perfektioniert worden. Auf der Suche nach einer optimalen Lösung hat sich ein neuer Begriff herausgebildet, der mit dem Wort *Objekt* bezeichnet wird. Ein *informatischer Objektbegriff* wurde bereits in Kap.18.2 [18.5] eingeführt, jedoch in anderem Zusammenhang und mit anderer Bedeutung. Dort war von Programmobjekten, jetzt ist von Prozessobjekten die Rede. In beiden Fällen ist der inhaltliche Kern die Kapselung. Er lässt sich sehr anschaulich am Beispiel der Zelle, des Bausteins aller Organismen verdeutlichen.

Eine Zelle ist mit einer Membran umgeben. Diese *kapselt* die Zelle von anderen Zellen *ab*. Die Abkapselung ist keine absolute. Vielmehr können die Zellen (bzw. die Prozesse in verschiedenen Zellen) in ganz bestimmter, wohlkontrollierter Weise miteinander kommunizieren. Träger der Kommunikation sind spezifische Moleküle, die unter bestimmten Bedingungen in der Lage sind, die Membran in der einen oder anderen Richtung zu passieren. Die gegenseitige Abkapselung der Zellen bewirkt eine gegenseitige Abkapselung der Prozesse. Tatsächlich ist die *Prozess*kapselung das eigentliche Ziel der Trägerkapselung (Zellkapselung).

In Analogie dazu kann auch hinsichtlich der Prozesse in Computern zwischen Trägerkapselung und Prozesskapselung unterschieden werden, wenn man das Programm, dessen Ausführung den Prozess bildet, als "softwaremäßigen *Prozessträger*" bezeichnet. In diesem Sinne wäre zwischen **Programmobjekt** (gekapseltem Programmbaustein) und **Prozessobjekt** (gekapseltem Prozess) zu unterscheiden. Die gegenseitige Abkapselung zweier Prozesse gegen direkte störende Einwirkungen (Seiteneffekte) kann dadurch erreicht werden, dass zwischen ihnen keine direkte, sondern ausschließlich indirekte Kommunikation zugelassen ist. Damit kommen wir zu den angekündigten Beispielen, anhand derer die Idee der Methode erläutert werden soll.

Im ersten Beispiel benötigt ein Prozess Daten, auf die er aus irgendeinem Grunde nicht direkt zugreifen kann oder darf. Im zweiten Beispiel wollen zwei Prozesse mit disjunkten Adressräumen miteinander kommunizieren. In beiden Fällen leistet ein dritter Prozess Hilfestellung. Wir bezeichnen ihn mit b, die beiden anderen mit a und c und ihre Adressräume entsprechend mit B, A und C. Damit lassen sich die Voraussetzungen, unter denen sich die Vorgänge in den beiden Fällen abspielen, kurz folgendermaßen beschreiben. A und C sind disjunkt, B überlappt sich mit A und C. Die Überlappungsbereiche bezeichnen wir mit $A \cap B$ und $C \cap B$. Außerdem gibt es eine Datenmenge DM, die "innerhalb" von B (präziser: unter den in B enthaltenen Adressen), aber "außerhalb" von A und C abgespeichert ist. Die Daten aus DM werden sowohl von a, als auch von c bearbeitet.

Beispiel 1. Prozess a benötigt Daten aus DM, auf die er nicht direkt zugreifen kann. Doch besteht die Möglichkeit des *indirekten* Zugriffs über den Prozess b, den er um seine "*Dienste*" bitten kann. Dazu muss er ihm die Bezeichner der gewünschten Daten übergeben. Prozess b kann dann die bestellten Daten holen und an Prozess a über den gemeinsamen Adressbereich abliefern. Dazu muss er folgende Maßnahmen treffen. Als erstes muss er die Daten suchen, d.h. er muss feststellen, auf welchem Speichermedium sie unter welchen Adressen abgespeichert sind. Handelt es sich um einen peripheren Speicher, muss er prüfen, ob der Zugang zu ihm frei oder von einem anderen Prozess belegt ist. Außerdem muss er prüfen, ob die bestellten Daten selber "frei" sind, d.h. ob sie nicht gerade von Prozess c oder von einem anderen Prozess bearbeitet werden. Wenn alle Voraussetzungen für einen Speicherzugriff erfüllt sind, holt Prozess b die Daten und liefert sie an Prozess a ab.

Prozess b führt für Prozess a eine "*Dienstleistung*" aus. Dabei ist a als *Dienstnutzerprozess* und b als *Dienstleistungsprozess* zu bezeichnen. Die ausführenden Operatoren (die auszuführenden Programme) kann man als *Dienstnutzer* und *Dienstanbieter* bezeichnen. In der Informatik spricht man stattdessen von **Client** und **Server**, und das Dienstleistungsprinzip heißt **Client-Server-Prinzip**. Die Realisierung des Prinzips ist Aufgabe des Systemprogrammierers. Der Anwendungsprogrammierer merkt davon wenig.

Wie das Beispiel zeigt, lässt sich das Problem der geteilten Nutzung von Speicherplätzen (in dem Beispiel durch die Prozesse a und c) mit Hilfe des Client-Ser-

ver-Prinzips in eleganter Weise lösen, vorausgesetzt, der Serverprozess (b) kontrolliert, ob die angeforderten Daten nicht zufällig gerade von c bearbeitet werden. Führt b die Kontrolle nicht durch, könnte man dennoch von einer Dienstleistung sprechen, sie würde aber a und c nicht gegeneinander abkapseln und nicht vor ungewünschter Wechselwirkung schützen. *Je nachdem, ob bei einer Dienstleistung die gegenseitige Abkapselung der beteiligten Prozesse garantiert ist oder nicht, liegt eine* ***schützende*** *bzw.* ***nicht schützende Dienstleistung*** *vor und wir sprechen von* ***schützendem*** *bzw.* ***nicht schützendem Client-Server-Prinzip*** . Zwei Bemerkungen zum Fachsprachgebrauch sind notwendig.

Bemerkung 1. Wir hatten Organisationsprogramme und periphere Steuerprogramme unter der Bezeichnung *Systemprogramme* oder *Dienstprogramme* zusammengefasst [7], weil sie für Anwendungsprogramme Dienste leisten. Dabei handelt es sich der ursprünglichen Idee nach nicht unbedingt um schützende Dienste. Die Tendenz geht aber dahin, möglichst viele Dienste als *schützende* Dienste zu programmieren.

Bemerkung 2. Der "Schutz von Prozessen" ist nicht mit dem "Schutz von Daten", dem sog. **Datenschutz** zu verwechseln. Bei letzterem geht es darum, Daten und Programme (allgemein Dateien) vor *unbefugtem* Zugriff zu schützen. Auf Fragen des Datenschutzes und der Datensicherheit gehen wir nicht ein. Über die Datenschutzproblematik kann sich der Leser z.B. in [Fleissner 97] informieren..

Beispiel 2. Die Prozesse a und c wollen miteinander kommunizieren. Das können sie nicht direkt, da sie keinen gemeinsamen Speicherplatz adressieren können, um Daten auszutauschen. Das Client-Server-Prinzip löst das Problem. Wenn beispielsweise Prozess a Daten an Prozess c übergeben will, trägt er sie in den Überlappungsbereich $A \cap B$ ein, b überträgt sie von $A \cap B$ nach $C \cap B$ und c liest sie aus $C \cap B$ aus. Es findet eine *indirekte* Kommunikation zwischen a und c statt. Dabei spielt b die Rolle des Serverprozesses. Falls er alle erforderlichen Kontrollen durchführt, erbringt er eine *schützende Dienstleistung*. Offensichtlich können an einer indirekten Kommunikation auch mehrere Server beteiligt sein, sodass die Daten über eine Kette von Speicherplätzen weitergegeben werden. Ein Beispiel dafür ist die Weiterleitung einer E-Mail von Server zu Server, worauf wir sogleich zurückkommen.

Das Dienstleistungsprinzip bringt für die Prozessorganisation mehrere Vorteile. Der Hauptvorteil aus der Sicht des Anwenderprogrammierers besteht darin, dass dieser sich "um nichts zu kümmern braucht". Es ist derselbe Grund, aus dem die meisten alltäglichen Dienstleistungen in Anspruch genommen werden, auch wenn sie etwas kosten. Auch das Client-Server-Prinzip kostet etwas, nicht nur Arbeitszeit des Systemprogrammierers, sondern auch Rechenzeit und Speicherplatz. Die Analogie zwischen alltäglichen Dienstleistungen und dem "*informatischen*" Client-Server-Prinzip soll an einigen Beispielen illustriert werden.

Eine Person, die einer anderen Person eine Nachricht zukommen lassen will, kann sich der Post *bedienen*. Dazu schreibt sie einen Brief und wirft ihn in einen Briefkasten. Den Rest erledigt der "Dienstleistungbetrieb Post", vorausgesetzt der Brief trägt die richtige Adresse. Eine modernere Möglichkeit wäre, dem Empfänger per Internet

eine E-Mail (“*elektronische Post*”) zu schicken. Dann übernimmt das verteilte System *Internet* die Weiterleitung und Zustellung der Nachricht. Ähnlich liegen die Verhältnisse, wenn ein Produzent den Vertrieb seiner Produkte einem Händler oder einer Handelskette überträgt. Andere Beispiele sind die Dienstleistungen eines Restaurants oder eines Reisebüros.

Die Beziehungen zwischen den Beteiligten sind in all diesen Beispielen die gleichen: Ein Dienstnutzer oder Besteller nimmt die Dienste eines Dienstanbieters in Anspruch. Es liegt eine Arbeitsteilung vor. Sie bewirkt, dass der Dienstnutzer sich ganz auf das konzentrieren kann, was ihn unmittelbar interessiert, auf seine Dienst*nutzer*prozesse, z.B. auf das Formulieren eines Briefes. Das gilt auch für den Dienstanbieter. Er kann sich ganz auf seine Dienst*leistungs*prozesse konzentrieren, z.B. auf das Transportieren und Zustellen von Briefen. Dienstnutzer- und Dienstleistungsprozess sind weitgehend voneinander abgekoppelt oder *abgekapselt*. Es liegt eine *gekapselte* Dienstleistung vor im Gegensatz zur *kapselnden* (*schützenden*) Dienstleistung, die zwei oder mehrere Klienten gegeneinander abkapselt.

Man beachte folgende Entsprechungen in der Analogie zwischen Alltags-Dienstleistungsprinzip und Client-Server-Prinzip. Dem Dienst*nutzer*prozess einer Alltagsdienstleistung entspricht im Falle einer informatischen Dienstleitung nach dem Client-Server-Prinzip die Ausführung eines *Anwendungsprogramms*, während dem alltäglichen Dienst*leistungs*prozess die Ausführung eines *Systemprogramms* entspricht und zwar eines *Organisationsprogramms*. Man erinnere sich, dass wir Systemprogramme auch als *Dienstprogramme* bezeichnet hatten [7]. Der Mechanismus des Client-Server-Prinzips, d.h. die Wechselwirkung zwischen Anwendungs- und Systemprogrammen nach diesem Prinzip soll am Beispiel der Kommunikation per E-Mail noch einmal demonstriert werden.

Ein Nutzer des Internet will einen Brief per E-Mail verschicken. Zunächst schreibt er an seinem Computer, der Zugang zu einem “*Server*” des Internet hat, den Text des Briefes und fügt die E-Mail-Adresse des Empfängers hinzu (Anwenderprozess). Beides übergibt er dem Server und bestellt damit eine Dienstleistung. Der Server ist ein Programm, das den Transport des Briefes zu organisieren hat, insofern ist es ein “Organisationsprogramm”. Seine Abarbeitung wird durch die Übergabe des adressierten Brieftextes gestartet. Der Serverprozess sucht den Adressaten und eine Verbindung zu ihm. In der Regel übergibt er die E-Mail einem anderen Server, der sie weiterleitet, bis sie einen Server erreicht, an den der Adressat “angeschlossen” ist.

Dieses Beispiel hinkt; es ist unvollständig insofern, als das Briefschreiben (der Clientprozess) nicht im Computer und nicht nach einem Programm abläuft (es sei denn man fasst den Vorsatz, einen Brief zu schreiben als “Programm” auf). Dagegen war das obige Beispiel der indirekten Kommunikation zwischen den Prozessen a und c vollständig insofern, als alle beteiligten Prozesse Ausführungen von Computerprogrammen waren.

Die *geschützte* (gekapselte) Dienstleistung bringt weitere Vorteile, die zunächst für Dienstleistungen des Alltagslebens erläutert werden sollen. Die Entkopplung von Dienstnutzer- und Dienstleistungsprozess hat zur Folge, dass Dienstnutzer und Dienstanbieter relativ unabhängig voneinander ihre Prozesse projektieren und verändern können. Derartige Änderungen dürfen sich jedoch nicht auf die Dienstleistung selber auswirken. Die Wünsche des Bestellers (aber nur diese) müssen genau erfüllt werden. Das wird dadurch gesichert, dass bei der Bestellung einer Dienstleistung ein Vertrag abgeschlossen wird, in welchem die Dienstleistung ausreichend genau charakterisiert ist. Der Vertrag muss eingehalten werden, auch bei Änderungen hinsichtlich der konkreten Ausführung der Dienstleistung.

Die Vorteile der beschriebenen Entkopplung sollen am Beispiel einer Reiseplanung illustriert werden. Herr X bereitet eine Reise vor. Dabei muss er viele Instanzen anschreiben, um Plätze in Verkehrsmitteln und Hotels sowie Karten für Veranstaltungen zu bestellen. Aus irgendeinem Grund muss er die Reise verschieben. Das bedeutet, dass er die ganze Vorbereitungsarbeit noch einmal leisten muss. Wenn Herr X die Reise aber nicht selber organisiert, sondern ein Reisebüro damit beauftragt, genügt es im Falle einer Verschiebung, den neuen Reisetermin anzugeben, eventuell ergänzt um einige Wunschänderungen. Für das Reisebüro bedeutet die Änderung die Wiederholung bestimmter Routineaktionen wie das erneute Ausfüllen bzw. Verändern einiger Formulare und deren Versendung, natürlich gegen Bezahlung.

In Analogie dazu muss ein Anwendungsprogrammierer, der Einzelheiten eines nichtgeschützten Dienstprogramms berücksichtigen muss (beispielsweise Details des Zugriffs auf Speicherinhalte oder des Hantierens mit kritischen Programmabschnitten), damit rechnen, dass infolge geringfügiger Änderungen in seinem Anwendungsprogramm Dienste eventuell falsch ausgeführt werden. Im Falle geschützter Dienstprogramme braucht er das nicht zu befürchten, vorausgesetzt, er hat die Vorschriften für die Schnittstelle zwischen Client und Server eingehalten.

Das Entsprechende gilt für einen Systemprogrammierer. Änderungen in einem Systemprogramm können zwar die konkrete Ausführung der angebotenen Dienste verändern, doch führen sie zu keiner Fehlbedienung, solange die Schnittstellenvorschriften eingehalten werden (in Analogie zur Einhaltung des Dienstleistungsvertrages). In der Rechnernetztechnik werden Schnittstellenvorschriften **Protokolle** genannt.

Wie sich das Client-Server-Prinzip programmtechnisch realisieren lässt, ist z.B. in [Tanenbaum 94] dargelegt. Ursprünglich ist das Prinzip für verteilte Systeme entwickelt worden. Angesichts seiner großen Vorteile stellt sich die Frage, ob es nicht auch in Systemen mit einfacherer Architektur zum Einsatz kommen kann. Es ist unschwer zu erkennen, dass dies der Fall ist. Ganz offensichtlich kann es z.B. bei der Organisation des Multitaskregimes in einem Einprozessorrechner angewendet werden. Tatsächlich kann es nicht nur bei der Speicherplatzzuweisung, sondern bei jeder Betriebsmittelzuweisung, bei jeder Prozesskommunikation und in jedem Rechner

eingesetzt werden. Wie das zu verwirklichen ist, werden wir uns in Kap.19.5.5 überlegen.

In diesem Kapitel hat sich ein weiterer Zugang zum Begriff des Objektes ergeben, wie er in der Informatik verwendet wird. Der Zugang in Kap.18.3 ging von einer begrifflichen und vorstellungsmäßigen Entsprechung zwischen Original und Modell [8.11] und von dem Wunsche nach semantischer Verdichtung aus. Er führte zum Begriff des *Programm*objekts [18.5]. Der Zugang dieses Kapitels ging von dem Wunsche aus, Prozesse gegen störende Einflüsse zu schützen. Er führte zum Begriff des *Prozess*objekts. Aus dem Zusammenwachsen der verschiedenen, zunächst getrennten Objektbegriffe ist ein Begriff entstanden, der primär den Begriff des Programmobjekts beinhaltet und erst sekundär den des Prozessobjekts. Es lässt sich folgendermaßen bestimmen:

Ein Programmobjekt oder kurz ***Objekt*** *ist ein relativ abgeschlossener Teil eines Programms (ein Programm-Modul), der einem relativ abgeschlossenen Bereich des modellierten Originals (Diskursbereiches) und einem relativ abgeschlossenen Bewusstseinsausschnitt (Denkobjekt) des Nutzers entspricht und bei dessen Ausführung ein mehr oder weniger* ***geschützter Prozess*** *abläuft*. Der Anwendungsprogrammierer kann evtl. mitbestimmen, wieweit ein Prozess gegen Störungen geschützt (abgekapselt) wird. Hierfür stellen objektorientierte Programmiersprachen geeignete Sprachelemente zur Verfügung. Das sei kurz erläutert.

Ein wichtiges Mittel in der Hand des Programmierers, einen Prozess gegen störende Einwirkungen von außen zu schützen, besteht darin, im Programm *öffentliche* (externe) und *private* (interne) Variablen zu deklarieren. Dadurch werden die als “privat” deklarierten Variablen von außen (für andere Prozesse) “unsichtbar” und dadurch geschützt, während öffentliche Variablen für alle Prozesse “sichtbar”, d.h. ungeschützt sind. Wenn ein Prozess einen anderen Prozess ruft, übergibt er die erforderlichen Operanden als Werte öffentlicher (externer) Variablen. Der gerufene Prozess verarbeitet sie intern unter Verwendung seiner privaten Prozeduren und seines privaten Speichers, in welchem die privaten (internen) Daten abgespeichert sind. Anschließend übergibt er das Resultat dem rufenden Prozess als Werte öffentlicher (externer) Variablen. Der Leser kann die Einzelheiten des Vorgehens an Hand des Programms von Bild 20.8 verfolgen. Dort sind die externen Ein- bzw. Ausgabeoperanden eines Objekts mit `inod` bzw. `outod` bezeichnet, die privaten Operanden dagegen mit `x, y,` oder anderen Buchstaben.

Eine weitere Möglichkeit, Prozesse zu schützen, besteht in der konsequenten Anwendung des Client-Server-Prinzips durch das Betriebssystem, worauf im folgenden Kapitel eingegangen wird.

19.5.5 Systemrufe und geschützte Prozesskomponierung

Wir hatten erkannt, dass Anwendungsprozesse mit Systemprozessen kooperieren müssen, damit ihr reibungsloser Ablauf in einem Computer oder PS-Netz gewährleistet ist. Wir wenden uns nun den Details der Kooperation zu. Ziel ist die Kompo-

nierung von *Anwendungs-Kompositprozessen*. Sie laufen im sog. *Anwendungsregime* eines oder mehrerer verfügbarer Prozessoren ab. Der richtige und ungestörte Ablauf der Prozesse wird durch Systemprogramme gewährleistet. Die *Systemprozesse* (die Abarbeitung von Systemprogrammen) laufen im sog. *Systemregime* ab. Gefragt ist nach dem Wechselwirkungsmechanismus zwischen Anwendungs- und Systemregime.

Im Falle eines Einprozessorrechners muss die CPU sämtliche Programme ausführen. Sie muss also zwischen dem Anwendungsregime und Systemregime hin- und herwechseln. Die Vorstellung vom "Hin- und Her*springen*" weist auf ein Datail des gesuchten Mechanismus hin. Offensichtlich muss er sich des Sprungbefehls (genauer eines sog. Sprungbefehls mit Rückkehrabsicht) bedienen. Wenn im Verlauf der Kooperation ein Prozess einen anderen ruft, muss durch den Ruf die Eintragung der Startadresse des gerufenen Programms in das Befehlsregister veranlasst werden. Um zu sichern, dass sich quasiparallele bzw. parallele Prozesse nicht gegenseitig stören, dürfen sie sich nicht *direkt* (durch direkte Kommunikation) aufrufen. Der Schutz der Prozesse kann dadurch gewährleistet werden, dass jeder Ruf über eine Zentrale erfolgt, die zum Betriebssystem gehört. Diese Zentrale heißt **Betriebssystemkern**. Sie ist in Bild 19.7 als Kreis dargestellt und kurz als *Kern* bezeichnet (Block 10).

Bild 19.7 stellt eine Erweiterung von Bild 19.6 dar. Die Blocknummerierung ist fortgesetzt. Die Erweiterung betrifft das Betriebssystem (die Blöcke 7, 8 und 10). Die nicht eingezeichneten Blöcke 1, 4 und 5 sind in Gedanken zu ergänzen. Zusätzlich denke man sich oberhalb von Block 8 einen Übersetzer und darüber einen Block, der dem Block 5 entspricht und der die Hierarchie der Quellprogramme des Betriebssystems enthält. Doch spielen die beiden Blöcke bei den folgenden Überlegungen keine Rolle.

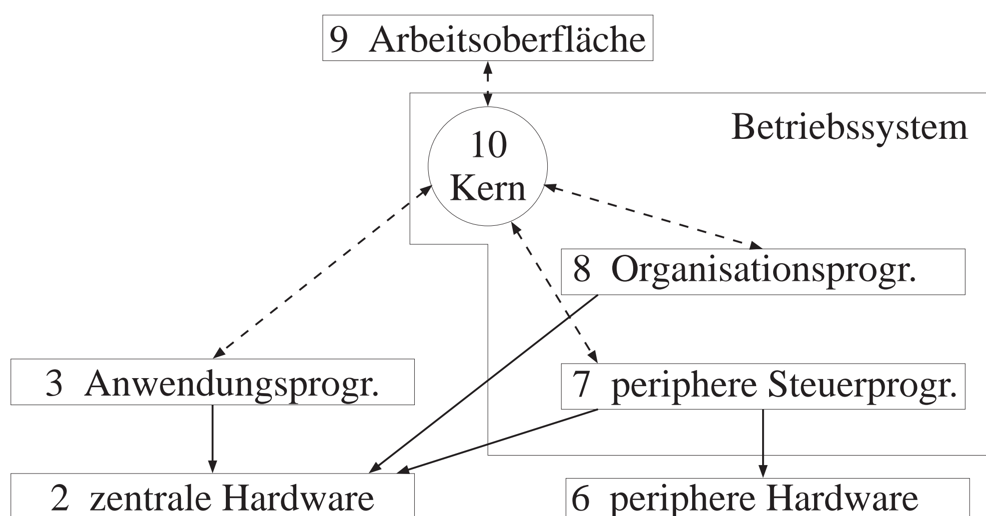


Bild 19.7 Geschützte Prozesskomponierung. Durchgezogene Pfeile - Befehlswege; gestrichelte Pfeile - Rufwege.

Entlang der durchgezogenen Pfeile in Bild 19.7 werden Steueranweisungen bzw. Steuersignale an die Hardware übergeben. Die gestrichelten Pfeile sind *Rufwege*. Jeder Rufweg zwischen zwei Programmen verläuft über den Kern. Der Kern stellt gewissermaßen die "Verbindung" vom Rufenden zum Gerufenen her. Insofern kann er als *Kommutator* aufgefasst werden. Direkte Rufwege zwischen ladbaren Anwenderprogrammen, die in Block 3 von Bild 19.6 durch gestrichelte Pfeile innerhalb von Block 3 dargestellt sind, existieren nicht. Der Kern vermittelt die "Bedienung" des rufenden Prozesses. De facto kommt das Client-Server-Prinzip zur Anwendung. Auf diese Weise wird eine klare Trennung der Prozesse erreicht, und es besteht die Möglichkeit, einen rufenden Prozess eventuell nicht zu bedienen, d.h. das gerufene Programm nicht zu rufen, nämlich dann, wenn die Bedingungen für eine ungestörte Prozesskommunikation nicht erfüllt sind. Eine solche Prüfung ist beim Entwurf des Betriebssystems einzubauen.

Wir wollen nun den speziellen Fall betrachten, dass der "rufende Prozess" die Tätigkeit des Computernutzers ist. Der Nutzer bzw. Bediener des Computers muss die Möglichkeit haben, Programme zu starten und Prozesse zu steuern. Jede Eingabe über die Tastatur ist ein *Systemruf* und bewirkt den Start des erforderlichen Systemprogramms. Wenn beispielsweise ein PC-Nutzer sein Textprogramm ruft, um einen Brief zu schreiben, bewirkt er damit, dass zunächst der Lader gestartet wird, der das Textprogramm in den Hauptspeicher lädt. Anschließend wird das Textprogramm gestartet. Das alles führt das Betriebssystem aus, nachdem der Name des Textprogramms über die Tastatur eingegeben oder das entsprechende Symbol auf dem Bildsschirm angeklickt worden ist.

Der Nutzer fungiert also als "Systemrufer". Er ist sogar der *allererste* Systemrufer, denn wenn er den Computer einschaltet, startet er den sog. *Urlader*. Dieser lädt die zu Beginn benötigten Systemprogramme von der Festplatte in den Hauptspeicher, wo sie nicht aufbewahrt werden können, weil sie beim Abschalten des Computers gelöscht würden. Der Anfangslader ist in der Regel auf einem ROM gespeichert mit direkter Verbindung zur CPU. Er gehört also nicht zu Block 8 in Bild 19.7. Das Urladen wird **Booten** genannt.

Ein bestimmter Dienst wird vom Nutzer durch Eingabe einer bestimmten Zeichenfolge über die Arbeitsoberfläche (Block 9) angefordert. Die Zeichenfolge wird **Kommando** genannt. Eingaben werden von Block 9 über den Kern zum **Kommandointerpreter** (in Bild 19.7 nicht eingezeichnet) weitergeleitet, der die Ausführung des Kommandos veranlasst. Die Kommandoeingabe erfolgt bei modernen Betriebssystemen wie z.B. Windows über eine komfortable Arbeitsoberfläche, wobei das *Fensterprinzip* ("Windows"-Prinzip) zur Anwendung kommt (siehe Kap.18.2).

Wir kehren noch einmal zu dem Fall zurück, dass das System nicht vom Nutzer, sondern von einem intern ablaufenden Prozess gerufen wird, beispielsweise von einem Hintergrundprozess, der Daten vom Plattenspeicher benötigt. Dazu muss der Hintergrundprozess unterbrochen und das entsprechende Systemprogramm gestartet werden. Wenn nun zur Laufzeit des Systemprogramms ein unterbrochener Vorder-

grundprozess die CPU verlangt, muss das laufende Systemprogramm unterbrochen und ein anderes gestartet werden, das die Fortsetzung des Vordergrundprogramms einleitet. Wie man sieht, werden Systemprogramme und Anwendungsprogramme hinsichtlich ihres Aufrufmechanismus völlig gleich behandelt. Infolgedessen ist es naheliegend, aus der Sicht des "Umschaltens" (des Unterbrechens und Startens von Programmen) keinen Unterschied zwischen Anwendungs- und Systemprogrammen zu machen und letztere *nicht* gedanklich zum Betriebssystem zusammenzufassen. Dann bleibt vom ursprünglichen Betriebssystem nur noch ein kleiner "Kern" übrig, eben der sog. Betriebssystemkern, der das Umschalten, die *Kommutation* zwischen den Prozessen besorgt. Das Betriebssystem wird zu einem reinen *Prozesskommutator*. Die Realisierung dieses Gedankens ist das Ergebnis moderner Betriebssystementwicklungen.

Bild 19.7 gibt eine etwas konservativere Sicht wieder, indem es die ladbaren Organisationsprogramme (Block 8), die ladbaren peripheren Steuerprogramme (Block 7) und den Betriebssystemkern (Block 10) zum Betriebssystem zusammenfasst. Über die (gestrichelten) Rufwege kann der Kern gerufen und angewiesen werden, die Ausführung dieses oder jenes Programms aus Block 3, 7 oder 8 zu starten, zu unterbrechen oder fortzusetzen, und über die gleichen Rufwege kann der Kern die Programme rufen und ihren Start bewirken.

Auf diese Weise steuert der Kern die Kommunikation zwischen Prozessen. Wenn die Kommunikation der Komponierung eines Kompositprozesses dient, steuert er eine "*Prozesskomponierung*" und wenn bei jedem Ruf die Bedingungen für eine störfreie Prozesskommunikation geprüft werden oder wenn die Bausteinprozesse *Prozessobjekte* sind, steuert er eine *geschützte Prozesskomponierung*. Die gestrichelten Kanten spielen dann die gleiche Rolle wie die gestrichelten Kanten in Bild 19.3 mit dem Unterschied, dass jetzt nicht Operationen, sondern deren Ausführungen, also Prozesse komponiert werden. Wenn man Bild 19.7 in diesem Sinne interpretiert, müssen die Programme durch Prozesse ersetzt werden, sodass hinter einem Programm in Block 3, 7 oder 8 mehrere Prozesse stehen können.

Mit einiger Phantasie könnte man den Kern mit dem menschlichen Bewusstsein vergleichen. Der Rufmechanismus über den Kern, also der indirekte Programmstart, entspräche der Ausführung einer bewussten Handlung. Der direkte Start unter Umgehung des Kerns entspräche der Ausführung einer reflektorischen Handlung unter Umgehung des Bewusstseins.

Es muss noch einmal betont werden, dass "vor" dem Kern alle ladbaren Programme "gleich sind". Der Kern macht keinen Unterschied zwischen Dienenden und Bedienten. Jeder Prozess kann als Dienstleistung oder als Dienstnutzung aufgefasst werden. Infolgedessen können alle Prozesse von den Vorteilen des Client-Server-Prinzips profitieren. Aus der Sicht des Programmierers besteht ein wesentlicher Vorteil in der Entkopplung der Prozesse bzw. der Programme. Wenn die Schnittstellenvorschriften gut formuliert und konsequent befolgt werden, können die Kompo-

nenten eines großen Systems relativ unabhängig voneinander entworfen und programmiert werden.

Wenn man den zurückgelegten Weg noch einmal bis zum Beginn von Kap.19.5.4 zurückverfolgt, kann es so scheinen, als hätten wir den ursprünglichen Anlass für die Entwicklung des Client-Server-Prinzips aus dem Auge verloren, nämlich den Entwurf von und den Umgang mit *verteilten* Systemen. Diesem Anschein liegt die zu enge Interpretation von Bild 19.5 als Ein- oder Mehrprozessorrechner zu Grunde. Die dargestellte Blockstruktur ist auch auf "weit verteilte" Systeme anwendbar. Die Elemente aller Blöcke können über ein großes Areal verteilt sein. Ein Prozess kann über den Kern bzw. über eine Kette von Kernen um Dienste bitten, die möglicherweise in einer anderen Stadt angeboten werden. Beispielsweise kann man darum bitten, in einer Datenbank, die in einem anderen Land existiert, nach bestimmten Daten zu recherchieren. Man kann auch um die Übersendung von Programmen oder von Datenbankausschnitten bitten, in denen man dann selber recherchieren möchte.

Die vorangehenden Darlegungen und die Bilder 19.6 und 19.7 geben nur eine sehr grobe Vorstellung von den vielseitigen, z.T. sehr diffizilen Problemen und Lösungen, die auf dem Gebiete des Systementwurfs, der Betriebssystemtechnik und der Computertechnik ganz allgemein bearbeitet bzw. erarbeitet worden sind. Doch wird ein Leser, der sich nach der Lektüre des Buches in die Struktur und Arbeitsweise eines Computers und speziell eines modernen Betriebssystems vertieft, in deren Entwurfsprinzipien die Ideen und Lösungsansätze wiederfinden, die hier angedeutet wurden. Beispielsweise kombiniert das Betriebssystem Windows NT (von New Technology) und ebenso sein Nachfolger Windows 2000 sehr geschickt das Prinzip der Schichtenstruktur mit dem der geschützten Prozesskomponierung. Gleichzeitig wird der Leser erkennen, wie wenig von den realisierten Einzellösungen zur Sprache gekommen ist. Das gilt nicht nur für die Betriebssystemtechnik, sondern für alle Bereiche, die in Kapitel 19 behandelt worden sind.

Noch viel weniger gibt das Buch eine Vorstellung davon, was "überhaupt möglich ist". Die Menge der realisierten Lösungen verschwindet im Meer der möglichen Lösungen. Dem phantasievollen Leser wird nicht entgangen sein, dass zu vielen Problemen, die besprochen wurden, zahllose andere Lösungen hätten gefunden werden können. Nachträglich sieht es so aus, als gäbe es diese anderen Lösungen gar nicht, denn Fachbücher behandeln nur das Existierende. Das weite Feld des Möglichen bleibt bis auf ganz Weniges unverwirklicht und außerhalb unserer Aufmerksamkeit. Das ist der Gang der kulturellen Evolution. Es ist ein mehr oder weniger zufälliger Zickzackweg über ein "weites Feld". Das meiste bleibt brach liegen. Denn die Menschen verlassen kaum den Bereich des Existierenden und die Wege des bereits Gedachten, sei es mangels Notwendigkeit, mangels Interesse, mangels Phantasie oder einfach mangels äußeren Anstoßes. Wer wann in welche Denkrichtung angestoßen wird, wem wann welche Idee kommt, das unterliegt keinen uns bekannten Gesetzmäßigkeiten, und es ist Zufall, dass die Rechentechnik so ist, wie sie ist.

20 Programmbeispiele

Zusammenfassung

In Teil 3 haben wir vier Programmierparadigmen kennen gelernt, das *imperative* oder anweisungsorientierte Paradigma, das *funktionale* oder applikative Paradigma, das *logische* oder relationale Paradigma und das *objektorientierte* oder direktive Paradigma. Die kursiv gedruckten Bezeichnungen sind die üblicheren. In diesem Kapitel sollen die vier Paradigmen anhand einiger Programmbeispiele illustriert werden. Alle Programme dienen der Berechnung der Funktion $y = f(x,n)$, die uns seit dem Kap.8 begleitet. Die Funktion ist durch die Formeln (vgl. (8.1))

$$f(x,n) = f_1(x,n) = x^n + x \quad \text{für } x \le 0 \tag{20.1}$$

$$f(x,n) = f_2(x,n) = x^n + \sin x \quad \text{für } x > 0$$

n > 0, ganzzahlig

festgelegt und wird von dem Operatorennetz von Bild 8.1 bei entsprechender Steuerung der Weichen berechnet. Unter den Programmen befinden sich imperative, funktionale, logische und objektorientierte Programme. In Kap.20.2 wird demonstriert, dass bei Zugrundelegung ein und derselben Sprache (in diesem Falle CommonLisp) Programme nach unterschiedlichen Paradigmen formuliert werden können. Doch führt das leicht zu einer Verfremdung (um nicht zu sagen "Vergewaltigung") der verwendeten Sprache, denn eine Programmiersprache ist i.Allg. auf ein bestimmtes Paradigma orientiert. Am Beispiel des logischen Programms von Bild 20.7 wird im Einzelnen verfolgt, wie ein logisches Programm interpretiert wird. Damit werden die Überlegungen von Kap.15.9 zur Interpretierung funktionaler Sprachen auf logische Sprachen erweitert.

Das letzte Beispiel (Kap.20.3) ist ein objektorientiertes Programm, geschrieben in BorlandPascal. Das Programm beinhaltet mehr als nur eine Vorschrift zur Berechnung der Funktion (20.1). Im Grunde stellt es ein, wenn auch sehr kleines, Software-Entwicklungssystem dar, ein Werkzeug zur Implementierung von Operatorennetzen. Zu einem praktikablen Werkzeug wird das Programm allerdings erst nach Ergänzung um weitere Objektklassen und um eine geeignete Arbeitsoberfläche (siehe Kap.20.4).

Ein Programm ist nicht nur durch das Programmierparadigma geprägt, das von der verwendeten Programmiersprache besonders unterstützt wird, sondern auch durch den *Programmierstil* des Programmierers. Das gilt insbesondere für professionelle Programmierer. Wer häufig eine bestimmte Programmiersprache verwendet, entwickelt sehr bald bestimmte Gewohnheiten, d.h. individuelle Regeln zusätzlich zu den Syntaxregeln, an die er sich beim Programmieren hält und die seinen persönlichen Programmierstil ausmachen.[1] Wie in der Programmierungstechnik üblich, werden anstelle der mathematischen Symbole $-, *, <, \le, >, \ge$ die Tastaturzeichen -, *, <, <=, >, >= verwendet.

20.1 Imperative Programme

Wir beginnen mit dem ältesten und nach wie vor am häufigsten verwendeten Paradigma, dem imperativen, und vergegenwärtigen uns noch einmal, wodurch sich imperative Programme auszeichnen. Ein *imperatives* Programm ist eine Folge von Anweisungen. Eine Anweisung schreibt eine Aktion vor, d.h. eine bestimmte Operation an bestimmten, explizit in der Anweisung angegebenen Operanden. Ein imperatives Programm ist ein maschinenverständlicher *imperativer Algorithmus* [7.10].

In Kap.15.2 hatten wir bereits einige Programmvarianten zur Berechnung der Funktion (20.1) unter der vereinfachenden Annahme formuliert, dass der untere Zweig der Alternativmasche in Bild 8.1 nicht existiert (siehe die Programme der Bilder 15.1 und 15.2). Dabei waren wir davon ausgegangen, dass die Maschinensprache den Operationscode `SIN` für die Berechnung der Sinusfunktion enthält (m.a.W. dass ein entsprechendes Firmwareprogramm existiert), dass sie aber keinen Operationscode für das Potenzieren enthält.

Bild 20.1a zeigt ein vollständiges Pascal-Programm, das auch die Alternativmasche enthält und in Abhängigkeit vom x-Wert entweder f_1 oder f_2 in (20.1) berechnet. Wer sich an Kap. 15.2 erinnert, wird das Programm wahrscheinlich ohne große Schwierigkeiten verstehen. Dennoch soll es kommentiert werden, wofür die Programmzeilen durchnummeriert sind. Die Zeilennummern in diesem und den folgenden Bildern gehören nicht zum Programmtext.

Bild 20.1b zeigt das dazu analoge Basic-Programm. Die sich entsprechenden Programmzeilen im Pascal- und Basic-Programm stehen in ein und derselben Druckzeile. Der Leser wird das Basic-Programm ohne Kommentar verstehen und die Syntaxregeln ablesen können, z.B. die Regel, dass der Typ Real bzw. Integer durch ein nachgestelltes Doppelkreuz bzw. Prozentzeichen angegeben wird.

Ein Vorteil des imperativen Programmierens liegt in der Möglichkeit der *prozeduralen Abstraktion*, d.h. in der Möglichkeit, gedanklich und im Text eines Programms (genauer eines Hauptprogramms) von den Einzelheiten einer Prozedur (eines Unterprogramms) zu abstrahieren, sodass nur ihr Name im Hauptprogramm gleichsam als Operationscode auftritt, als sei die Prozedur firmwaremäßig realisiert. Das Programm von Bild 20.2 demonstriert dieses Vorgehen. Die Zeilen 16 bis 28 bilden das *Hauptprogramm*. Es ist die Vorschrift für eine Kompositoperation, die aus drei Bausteinoperationen mit den Bezeichnern `pot, sin, +` komponiert ist. Die Sinus- und die Additionsoperation werden als hardwaremäßig realisiert angenommen, sodass deren Details softwaremäßig nicht in Erscheinung treten. Dagegen ist die Potenzoperation als gesonderte Prozedur, als *Unterprogramm* ausprogram-

---

1 Die Programme in den Kapiteln 20.1 und 20.3 sind von Bernd Dupal, die in Kap.20.2 von Wolfgang Oertel erstellt.

```
 1  Program Beispiel1;
 2  Var
 3       n, z    :Integer;
 4       x, y, a :Real;
 5  Begin
 6       Write('x='); Readln(x);
 7       Write('n='); Readln(n);
 8       y := 1;
 9       z := 0;
10       While z<n Do
11       Begin
12          y := x*y;
13          z := z+1;
14       End;
15       If x<=0
16          Then a := x
17          Else a := Sin(x);

18       y := a+y;
19       Writeln('Ergebnis=',y):
20  End.
```

a) Pascal-Programm

```
REM Programm Beispiel2




INPUT 'x=',x#
INPUT 'n=',n%
y#=1.0
z%=0
WHILE z%<n THEN

    y#=x#*y#
    z%=z%+1
WEND
IF x#<=0
    THEN a#=x#
    ELSE a#=SIN(x#)
ENDIF
y#=a#+x#
PRINT 'Ergebnis=';y#
```

b) Basic-Programm

Bild 20.1 Imperative Programme in Pascal und Basic.

miert und dem Hauptprogramm vorangestellt. Doch ist das Unterprogramm in Zeile 2 nicht als Prozedur, sondern als *Funktion* deklariert und bei seinem Aufruf in Zeile 22 wird es als Funktion eingebunden, d.h. der Wert der Funktion ist nicht explizit durch einen Bezeichner im Programm vertreten, sondern die rechte Seite der Ergibtanweisung wird durch den Wert der pot-Funktion substituiert. Viele imperativen Programmiersprachen stellen den Funktionsmechanismus zur Verfügung, obwohl es sich um ein artfremdes (paradigmenfremdes) Element handelt.

Bevor das Programm ausgeführt werden kann, muss das Unterprogramm zur Berechnung der pot-Funktion in das Hauptprogramm eingebunden werden. Das ist Aufgabe des Binders [15.4]. Außerdem müssen die Operanden der pot-Funktion, die in Zeile 2 deklariert sind, die sogenannten *formalen Parameter*, durch die *aktuellen Parameter* substituiert werden, d.h. durch die Operandenwerte, die der pot-Funktion übergeben werden. Für die *Parameterübergabe* sind verschiedene Mechanismen entwickelt worden, auf die hier nicht eingegangen werden soll.

Die Unterprogrammtechnik (das Prozedurkonzept) ist ein mächtiges Programmierhilfsmittel. Es erleichtert nicht nur das Schreiben und Lesen von Programmen, es ermöglicht auch das Komponieren von Operatorenhierarchien (Prozedurhierarchien) [15.11] sowie die *Nachnutzung* vorhandener Programme. Beispielsweise könnte

das Programm zur Berechnung der Potenz einer Programmbibliothek entnommen werden, sodass in Bild 20.2 die vorangestellte Prozedur entfallen kann. In unserem Beispiel bringt die Unterprogrammtechnik keine Vorteile, weil die pot-Operation nur ein einziges Mal ausgeführt wird.

```
 1   Program Beispiel3;
 2   Function pot(x :Real; n :Integer) :Real;
 3   Var
 4      z  :Integer;
 5      y  :Real;
 6   Begin
 7      y := 1;
 8      z := 0;
 9      While z<n Do
10      Begin
11         y := x*y;
12         z := z+1;
13      End;
14      pot := y;
15   End;
16   Var
17      n  :Integer;
18      x, y, a  :Real;
19   Begin
20      Write('x='); Readln(x);
21      Write('n='); Readln(n);
22      y := pot(x,n);
23      If x<=0
24         Then a := x
25         Else a := Sin(x);
26      y := a+y;
27      Writeln('Ergebnis=',y);
28   End.
```

Bild 20.2 Pascal-Programm mit der pot-Funktion als Unterprogramm.

20.2 CommonLisp-Programme

20.2.1 Funktionale Programme

Die Sprache *CommonLisp* ist ein Abkömmling der Sprache Lisp, die von J.McCarthy auf der Grundlage des Lambdakalüls entwickelt wurde. Die Sprache CommonLisp unterstützt - ebenso wie Lisp selber - das Programmieren nach dem funktionalen Paradigma. Sie kann auch das imperative und objektorientierte Programmieren unterstützen. Für das logische Programmieren sind einige Sprachergän-

zungen erforderlich. Dafür werden Beispiele gebracht. Wir beginnen mit dem funktionalen Programmieren.

Ein *funktionales* Programm notiert die Vorschrift für eine Kompositoperation in Form eines Klammerausdrucks, wie es in der Mathematik üblich ist. Im Falle einer Kompositoperation mit mehreren Komponierungsebenen ergibt sich ein mehrfach geschachtelter Klammerausdruck. Dabei steht jeder geklammerte Ausdruck für den Wert, den die Ausführung der in der Klammer stehenden Operation liefert. Demzufolge treten in einem rein funktional notierten Programm keine internen Operanden auf; für sie sind Werte- und Variablenbezeichner überflüssig.

Denjenigen Lesern, die an die Sprache der Mathematik gewöhnt sind, liegt das funktionale Paradigma näher als das imperative, denn in der Algebra wie in der Analysis ist es üblich, Funktionen "*funktional*" zu notieren. In Kap.8.1 waren zwei funktionale Notationsweisen eingeführt worden, die in der Mathematik weniger üblich sind, die *Präfixnotation* und die *Listennotation* (siehe die Formeln (8.15) und (8.16) [8.34]). Für die Funktion (20.1), die durch das Operatorennetz von Bild 8.1 berechnet wird, lautet die Präfixnotation

$$f(x,n) = f_1(x,n) = +(\mathrm{pot}(x,n),\mathrm{x}) \quad \text{für } x <= 0;$$

$$f(x,n) = f_2(x,n) = +((\mathrm{pot}(x,n),\mathrm{x}),\sin(x)) \quad \text{für } x > 0 \tag{20.2}$$

und die Listennotation, wie sie in CommonLisp verwendet wird,

$$f(x\ \ n) = f_1(x\ \ n) = (+\ \ (\mathrm{pot}\ \ x\ \ n)\ \ x\ \ )\ \ \text{für } x <= 0$$

$$f(x\ \ n) = f_2(x\ \ n) = (+\ \ (\mathrm{pot}\ \ x\ \ n)\ \ (\sin\ \ x))\ \ \text{für } x > 0 \tag{20.3}$$

In den ersten drei Zeilen des folgenden Programms wird der Leser die Formel (20.3) wiedererkennen. Das Programm ist in der Programmiersprache CommonLisp notiert. In jeder der ersten vier Zeilen wird eine Funktion definiert. Das Schlüsselwort `defun` ist als "definiere Funktion" zu lesen. Zeile 5 ist die Anfrage. Zeile 4 ist die rekursive Definition der Potenzfunktion pot(x,n); wenn n größer als 0 ist, wird die n-te auf die (n-1)-te Potenz zurückgeführt. Die Zeilen 2 und 3 definieren unter Verwendung der pot-Funktion die Funktionen `f1` und `f2`. Die erste Zeile identifiziert `f` mit `f1`, falls x kleiner oder gleich 0 ist, andernfalls mit `f2`.

Funktionale Programme werden interpretativ abgearbeitet (siehe Kap.15.9). Der Interpreter wertet jede eingelesene und auswertbare Zeichenkette sofort aus. Er substituiert sie entweder durch ihren Wert, z.B. `(<= -2 0)` durch `true`, oder durch eine andere Zeichenkette, z.B. `(f1 x n)` durch `(+(pot x n)x)` gemäß Zeile 3. Das *Auswerten* oder *Evaluieren* ist in Kap.8.4.7 vom theoretischen Standpunkt aus behandelt worden. Jetzt wollen wir es am praktischen Beispiel verfolgen, wenn der Wert f(-2, 3) berechnet wird (vgl. auch Kap.15.9).

Zunächst wertet der Interpreter das Prädikat `(<= -2 0)` aus und substituiert es durch `true`. Damit weiß er, dass `(f -2 3)` durch `(f1 -2 3)` zu substituieren ist. Diesen Ausdruck substituiert er gemäß Zeile 2 durch `(+(pot -2 3)-2)`. Im

```
1  (defun f(x n) (if(<= x 0)(f1 x n)(f2 x n)))
2  (defun f1(x n)(+(pot x n)x)
3  (defun f2(x n)(+(pot x n)(sin x)))
4  (defun pot(x n)(if(= n 0)1(* x(pot x(- n 1)))))
5  (f -2 3)
```

—> 10

Bild 20.3 CommonLisp-Programm zur Berechnung der Funktion (20.3).

nächsten Schritt wendet er Zeile 4 an. Da das Prädikat `(= -2 0)` falsch ist, substituiert er `(pot -2 3)` durch `(* -2(pot -2 2))`. Auf den zweiten Faktor dieses Produktes wendet er Zeile 4 ein zweites Mal und anschließend ein drittes mal an; er führt also eine rekursive Iteration aus. In jedem Iterationsschritt erniedrigt sich die Potenz um 1. Im dritten Iterationsschritt ergibt sich der Ausdruck `(pot -2 0)`, der nun durch `1` substituiert wird, weil das Prädikat `(= 0 0)` wahr ist. Der Interpreter geht nach der Methode des rekursiven Berechnens vor (vgl. Kap.8.4.6) und erhält am Ende bei der "Hochrechnung", d.h. bei der Auswertung des Ausdrucks `(+(* -2(* -2(* -2 1)))-2)`, den Funktionswert `-10`.

```
(defun f(x n)(+(pot x n)(if(<= x 0)x(sin x))))
(defun pot(x n)(if(= n 0)1(*x(pot x(- n 1)))))
```

Bild 20.4 Variante des Programms von Bild 20.3 mit zwei statt vier `defun`-Klammerausdrücken.

Bild 20.4 zeigt eine Variante des Programms von Bild 20.3, das aus zwei statt aus vier Programmzeilen besteht. Es werden nur noch zwei Funktionen definiert. Die Funktionen `f1` und `f2` sind nicht explizit definiert, sondern die sie definierenden Ausdrücke sind unmittelbar in die `if`-Funktion eingetragen. Es handelt sich um eine *Funktionskomponierung*. In der ersten Zeile wird eine Kompositfunktion höheren Komponierungsgrades definiert. Es sei erwähnt, dass sich das Programm sogar in einer einzigen Zeile (`defun`-Klammer) schreiben lässt. Dazu muss die zweite Zeile in Bild 20.4, die Definition der `pot`-Funktion, in die erste integriert werden. Das verlangt die Verwendung des CHURCHschen LAMBDA-Operators [8.35] in der Form, wie er in CommonLisp definiert ist.

Beim *dekomponierenden* Übergang von einer zu zwei bzw. von zwei zu vier Programmzeilen werden Programmteile aus dem ursprünglichen Programm ausgegliedert. Dem entspricht beim imperativen Programmieren das Ausgliedern einer Bausteinoperation als Prozedur (Unterprogramm).

20.2.2 Imperatives Programm

In Bild 20.5 wird CommonLisp “missbraucht”, um ein imperatives Programm zu schreiben. Dabei tritt der charakteristische Unterschied zum funktionalen Paradigma sehr deutlich hervor. Das Programm ist in einem imperativen Dialekt von CommonLisp geschrieben und verwendet demzufolge ebenfalls die Listennotation. Um imperativ programmieren zu können, muss die Sprache dahingehend erweitert werden, dass Zwischenergebnisse *explizit* darstellbar sind, d.h. es muss eine funktionale Entsprechung der Ergibtanweisung geben. Der Leser hat vielleicht in dem Funktionsbezeichner `setq` bereits das geforderte Sprachelement erkannt.

```
 1  (defun f()
 2  (defvar x)(defvar n)(defvar y)(defvar m)(defvar a)
 3    (setq x(read))(setq n(read))
 4    (setq y 1)(setq m 0)(setq a 0)
 5    (loop(if(= m n)(return))
 6      (setq y(* x y))
 7      (setq m(+ m 1)))
 8    (if(<= x 0)(setq a x)(setq a(sin x)))
 9    (setq y(+ y a))
10    (print y))
11  (f)
12  -2
13  3
    —> 10
```

Bild 20.5 Imperatives Programm, formuliert in CommonLisp.

In Zeile 2 werden die Variablen deklariert und in den Zeilen 3 und 4 gemeinsam mit den Zeilen 12 und 13 werden ihnen Werte zugewiesen, ihre Werte werden “gesetzt” (`set`). Da der CommonLisp-Interpreter die imperative Notation der Ergibtanweisung nicht versteht, müssen die Wertzuweisungen in Form von Funktionen notiert werden. Doch muss dem Interpreter gesagt werden, dass er Klammerausdrükke, z.B. die Zeile 4, nicht wie üblich auswerten soll, dass er also nicht nach möglichen Substituionen suchen und sie ausführen soll, sondern dass er lediglich `y` auf den Wert `1` setzten (unter `y` eine `1` abspeichern) soll. Genau dies ist die Semantik von `setq`. Das `q` in `setq` ist ein Relikt der Sprache Lisp, in der es den Bezeichner `QUOTE` gibt (deutsch: zitiere) mit der beschriebenen Semantik (Auswerteverbot).

Die mit `loop` beginnende *Schleifenfunktion* schreibt - ähnlich wie die STEP-UNTIL-Anweisung und die WHILE-Anweisung - eine Iteration vor. Das Schlüsselwort `return` bedeutet den Abbruch der Iteration, die Rückkehr in die normale Befehlsfolge, sobald `m=n` wird, und die Rückgabe des berechneten Wertes an den übergeordneten Prozess. Die Zeilen 8 und 9 entsprechen den Zeilen 3 bis 5 in Bild 13.10.

Ein Vergleich der Bilder 20.5 und 20.4 verdeutlicht noch einmal den wiederholt diskutierten Unterschied zwischen dem imperativen Programmieren "*in kleinen Stücken*", in einzelnen, von einander getrennten Anweisungen, und dem funktionalen Programmieren "*in großen Stücken*", evtl. sogar "*in einem Stück*", in einem einzigen verschachtelten Klammerausdruck.

20.2.3 Objektorientiertes Programm

Die treibende Kraft für die Herausbildung des objektorientierten Paradigmas hatten wir in den beiden Wünschen gesehen, die semantische Lücke zu verringern und Prozesse vor gegenseitigen Störungen zu schützen. Wie wir aus den Kapiteln 18.3 und 19.5.4 wissen, war das Resultat der Bemühungen die Zusammenfassung "privater" Methoden und Daten zu einer programmtechnischen Einheit, **Objekt** genannt. Wenn mehrere Objekte sich durch gleiche Merkmale auszeichnen, wenn sie z.B. über gleiche Methoden verfügen, können sie zu einer **Objektklasse** zusammengefasst werden. Dadurch entsteht ein neuer *Datentyp*. In diesem Falle wird die Klasse (der Datentyp) als Objekt und ein Element der Klasse bzw. ein Exemplar (ein konkreter "Fall") des Typs als **Instanz** (von englischen instance = Fall) bezeichnet. Das Einrichten einer Instanz heißt *Instanzieren*. Dabei handelt es sich um einen Spezialfall des Instanzierens in Bild 5.4. Auf einer höheren Abstraktionsebene werden die Begriffe "Objektklasse" und "Instanz" unter dem Oberbegriff "Objekt" zusammengefasst.

1

Bei der Abarbeitung eines objektorientierten Programms führen Objekte Kooperationsaufträge anderer (Dienstleistungen für andere) Objekte aus. Solche Aufträge werden auch als *Direktiven* bezeichnet. Direktiven sind - ebenso wie Befehle - Aktionsvorschriften, d.h. sie enthalten nicht nur die gewünschte Operation, sondern auch die Operanden.

Durch das objektorientierte Paradigma kann die semantische Lücke in zweierlei Hinsicht verringert werden. Zum einen kann ein objektorientiertes Programm so entworfen werfen, dass Objekten (im umgangssprachlichen Sinn) des zu modellierenden Diskursbereiches Objekte (im programmtechnischen Sinn) des Programms entsprechen. Zum anderen kann der Programmierer die Objekte seines Programms als relativ selbständige, miteinander kooperierende Akteure auffassen. Aus dem kausalen Netz des Originals wird ein Kooperationsnetz im Modell, sodass der Programmierer auch beim Programmieren netzorientiert denken kann, wie er es gewohnt ist.

Der Entwurf eines objektorientierten Programms beginnt mit der Festlegung, welchen Objekten des zu modellierenden Diskursbereiches Objekte des Programms entsprechen sollen. In dem Programm von Bild 20.6 entspricht dem gesamten Operatorennetz von Bild 8.1 ein einziges Objekt. Infolgedessen kann das Programm zwar das Kapseln der Methoden und Daten in einem Objekt, nicht aber das Kooperieren zwischen Objekten demonstrieren. Das bleibt dem Programm in Kap.20.3 vorbehalten.

Die verwendete Programmiersprache *Clos* (Abkürzung von CommonLisp Object System) ist ein Bestandteil von CommonLisp und verwendet die Listennotation von CommonLisp. Ein Hochkomma (Apostroph) hat die gleiche Bedeutung wie `QUOTE` in Lisp (s.o.), d.h. ein "quotierter" (mit Hochkomma versehener) Bezeichner wird nicht ausgewertet.

```
 1 (defclass netz () (x n y))
 2 (defmethod f ((ne netz))
 3  (setf(slot-value ne 'y)
 4   (if(<=(slot-value ne 'x)0)(f1 ne) (f2 ne))))
 5 (defmethod f1 ((ne netz))
 6  (+(pot ne)(slot-value ne 'x)))
 7 (defmethod f2((ne netz))
 8  (+(pot ne)(sin(slot-value ne 'x))))
 9 (defmethod pot((ne netz))
10  (if(=(slot-value ne 'n)0)1
11   (progn(setf(slot-value ne 'n)(-(slot-value ne 'n)1))
12    (*(slot-value ne 'x) (pot ne)))))
13 (setq netz1(make-instance 'netz))
14 (setf(slot-value netz1 'x)-2)
15 (setf(slot-value netz1 'n)3)
   —> -10
```

Bild 20.6 Objektorientiertes Programm, geschrieben in der Sprache Clos, einem Bestandteil von CommonLisp.

In Zeile 1 wird eine Klasse namens `netz` mit drei öffentlichen (externen) Variablen `x,n,y` definiert. Für die Variablen werden Plätze im privaten (internen) Speicher des Objekts reserviert. Private Speicher sind durch das Schlüsselwort `slot` gekennzeichnet. In den Zeilen 2 bis 4 wird die Methode `f` des Objektes `ne` der Objektklasse (des Objekttyps) `netz` definiert. Das Schlüsselwort `setf` stellt zusammen mit den beiden folgenden Klammerausdrücken eine Ergibtanweisung dar. Der gemäß der zweiten Klammer (Zeile 4) zu berechnende Wert ist der privaten Variablen `y` zuzuweisen. Die Syntax der Zeile 4 ist die in Formel (8.21) [8.36] vereinbarte. Die Methode `f` wird aus den Methoden `f1(netz)` und `f2(netz)` komponiert, die ihrerseits in den Zeilen 5 bis 8 definiert sind. Sie enthalten die Methode `pot((ne netz))`, die in den Zeilen 9 bis 12 definiert ist. Das Schlüsselwort `progn` zeigt an, dass die nachfolgenden Klammerausdrücke nacheinander abzuarbeiten sind und dass der Wert des letzten Klammerausdrucks als Ergebnis zurückzugeben ist.

Wie an dem Schlüsselwort `slot-value` zu erkennen ist, arbeiten die Methoden (Prozeduren) `f1,f2,pot` ausschließlich mit privaten Variablen. Die Abarbeitungsprozesse können also nicht von außen (von Prozessen in anderen Objekten) gestört werden, sie sind *gekapselt*. Mit den Zeilen 13 bis 15 wird eine Instanz `netz1` der

Klasse `netz` mit den Werten `-2` und `3` für `x` bzw. `n` eingerichtet (instanziert), und anschließend wird das Programm gestartet.

20.2.4 Logisches Programm

Ein *logisches* Programm ist ein Prädikat (im Sinne des Prädikatenkalküls). In der Regel ist es ein *Kompositprädikat* aus einer mehr oder weniger großen Anzahl von *Bausteinprädikaten*. Der Computer entscheidet, ob das Prädikat wahr ist, bzw. er berechnet die Werte der Prädikatvariablen, für die es wahr ist. Ein Prädikat stellt an sich keinen Algorithmus dar. Wenn aber ein Computer über einen entsprechenden Interpreter verfügt und wenn er eine eingegebene Zeichenfolge als Prädikat erkennt, startet er ein Programm, welches das Prädikat auswertet. Dieses Programm stellt den Kern des Interpreters dar.

Um den Sprung in das logische Paradigma zu erleichtern, rekapitulieren wir noch einmal, wie ein Auftrag an den Computer in den verschiedenen Programmierparadigmen zu artikulieren ist, wenn er die Summe a + b berechnen soll.
Die **imperative Notation** lautet s := a + b. Es sei daran erinnert, dass im imperativen Paradigma das Resultat einen Bezeichner erhalten muss; wir bezeichnen die Summe mit dem Buchstaben “s”. Die Ergibtanweisung weist den Computer an, den Wert der Summe unter s abzuspeichern (d.h. auf dem Speicherplatz, an den s gebunden ist). Das “+”-Zeichen bezeichnet eine *Operation*.
Die **funktionale Listennotation** lautet (+ a b). Sie weist den Interpreter an, den Klammerausdruck durch ihren Wert zu substituieren. Das “+”-Zeichen bezeichnet eine *Funktion*.
Die **logische Notation**, wie wir sie bisher für die Notation von Prädikaten verwendet haben, lautet (a + b = s), wobei das Gleichheitszeichen ein *Relations*zeichen ist.
Die **logische Listennotation** lautet (+ a b s). Diese Zeichenkette weist den Interpreter an, diejenigen Wertetripel zu finden, für die das Prädikat (a + b = s) wahr ist. Das “+”-Zeichen ist diesmal als Bezeichner eines *Prädikats* aufzufassen. Wenn in analoger Weise der Bezeichner “*f*” der Funktion (20.1) als Prädikatbezeichner verwendet wird, dann wird durch $(f\ x\ n\ y)$ ein Prädikat notiert, das für diejenigen y-Werte wahr ist, die sich nach (20.1) als Werte der Funktion $f(x,n)$ ergeben.

Bild 20.7 zeigt ein logisches Programm zur Berechnung der Funktion (20.1). Es ist in einer Sprache geschrieben, die von einem in CommonLisp implementierten Prologinterpreter verstanden wird. Das Programm ist ein Beispiel dafür, wie man *nicht* programmieren sollte. Denn eine logische Sprache ist für die Programmierung mathematischer Funktionen wie (20.1) ganz und gar ungeeignet. Logische Sprachen sind für die Artikulierung logischer Zusammenhänge prädestiniert, wie am Beispiel des Verwandtschaftsproblems [16.2] durch das Prolog-Programm von Bild 16.3 in Kap.16.2 demonstriert wurde. Dass trotzdem das folgende Programm vorgestellt wird, hat verschiedene Günde. Es soll gezeigt werden, dass zur Lösung ein und desselben Problems Programme in Sprachen aller vier Paradigmen geschrieben werden können, dass es aber zu erheblichen sprachlichen Entstellungen kommt, wenn

das Paradigma nicht an das Problem angepasst ist. Das folgende Programm widerspricht geradezu dem gesunden Menschenverstand, dem normalen, folgerichtigen Denken. Außerdem wollen wir am Beispiel der uns inzwischen geläufigen Funktion (20.1) im einzelnen verfolgen, wie der Interpreter eines logischen Programms vorgeht, mit anderen Worten, wir wollen die interne Semantik des Programms (seine Wirkung im Computer) verstehen. Der Leser wird in jeder der nummerierten Zeilen des Programms eine Folge von Prädikaten erkennen. Die maschineninterne Semantik der Zeilen ist jedoch kaum ohne zusätzliche Erklärungen zu verstehen.

```
(setq f
1 '(((f x n y)(%(<= x 0))(f1 x n y))
2   ((f x n y)(%(> x 0))(f2 x n y))
3   ((f1 x n y)(pot x n u)($(+ u x y)))
4   ((f2 x n y)(pot x n u)($(sin x a))($(+ u a y)))
5   ((pot x 0 1))
6   ((pot x n u)(%(> n 0))($(- n 1 m))(pot x m v)($(* x v u)))))
    (prolog f ((f -2 3 y)))
    —> 10
```

Bild 20.7 Logisches Programm zur Berechnung der Funktion (20.1).

Um dem Verständnis des Programms näher zu kommen, überlegen wir uns, wie sich der Auftrag für die Berechnung der Funktion (20.1) unter Verwendung von Prädikaten verbal formulieren lässt. Wem der Sprung in das Denkschema des logischen Programmierparadigmas gelungen ist, der könnte den Auftrag zunächst folgendermaßen artikulieren:

"Finde die (x,n,y)-Tripel, für welche entweder die beiden Prädikate

$(x \leq 0)$ **und** $(x^n + x = y)$

gleichzeitig wahr sind **oder** für welche die beiden Prädikate

$(x > 0)$ **und** $(x^n + \sin(x) = y)$

gleichzeitig wahr sind!"

Dieser Satz enthält zwar Prädikate, ist selber aber kein Prädikat (im mathematischen Sinne), sondern ein *Imperativsatz*. Er muss in ein Prädikat umgewandelt werden. Dazu machen wir uns klar, dass für diejenigen Tripel, die gefunden werden sollen, das Prädikat `(f x n y)` - in der oben genannten Bedeutung - wahr wird. Der Imperativsatz kann also als *Prämisse* der *Konklusion* "`(f x n y)` ist wahr" aufgefasst werden, sodass das gesuchte Prädikat zu einer Implikation wird. Folglich kann das Prädikat, in das der Imperativsatz zu überführen ist, als Implikation notiert werden. Ersetzt man die in dem Imperativsatz fett gedruckten Verbindungswörter durch boolesche Operatoren, ergibt sich der Ausdruck

$$(f\,x\,n\,y) \Leftarrow ((x \leq 0)\ \texttt{AND}\ (x^n + x = y))\ \texttt{OR}\ ((x > 0)\ \texttt{AND}\ (x^n + \sin(x) = y)). \quad (20.4)$$

Das Implikationszeichen ist nach links gerichtet, d.h. die Konklusion steht auf der linken, die Prämisse auf der rechten Seite. Die Prämisse ist ein boolescher Ausdruck in disjunktiver Normalform. Wenn die Prädikate $(x^n + x = y)$ und $(x^n + \sin(x) = y)$ mit f_1 bzw. f_2 bezeichnet werden, geht (20.4) über in

$$(f\,x\,n\,y) \;\Leftarrow\; ((x{\le}0)\ \texttt{AND}\ (f_1\,x\,n\,y))\ \texttt{OR}\ ((x{>}0)\ \texttt{AND}\ (f_2\,x\,n\,y)). \tag{20.5}$$

Durch den `OR`-Operator sind zwei disjunktive Prämissen definiert, sodass die Implikation in zwei gleichberechtigte Implikationen zerlegt werden kann. Wenn die beiden Implikationen untereinander geschrieben werden und sämtliche booleschen Operatoren gestrichen werden, ergeben sich die Zeilen 1 und 2 des Programms von Bild 20.7 (das Prozentzeichen ignorieren wir zunächst). Ihre Semantik ist mit derjenigen von (20.5) identisch. Die beiden Zeilen zusammen sind also folgendermaßen zu lesen: “Das Prädikat `(f x n y)` ist wahr, wenn die Prädikate `(<= x 0)` **und** `(f1 x n y)` wahr sind **oder** wenn die Prädikate `(> x 0)` **und** `(f2 x n y)` wahr sind”. Die verkürzte Notation der beiden Programmzeilen entspricht der Syntax der verwendeten Sprache. Sie stammt aus der mathematischen Logik.

Die Zeilen 1 und 2 stellen die Notation einer Alternative im logischen Denkschema dar. Im datenflussorientierten Denkschema entspricht ihnen die Alternativmasche in Bild 8.1. Wenn das Steuerprädikat `[x<=0]` wahr ist, wird die Zweigeweiche nach oben gestellt, sodass der Sinusoperator umgangen wird. Im Pascal-Programm von Bild 20.1, also im imperativen Denkschema, entspricht dem die Wahl zwischen den beiden Anweisungen in den Zeilen 16 und 17. Wenn das Prädikat `x<=0` wahr ist, wird die Ergibtanweisung `a:=x` ausgeführt, andernfalls die Ergibtanweisung `a:=Sin(x)`. Im CommonLisp-Programm von Bild 20.3, also im funktionalen Denkschema, wird die Alternative als if-Funktion notiert, im Funktionalen Programm von Bild 20.7 durch die Zeilen 1 und 2. Wenn das Prädikat `(x<=0)` erfüllt ist, dann ist `(f x n y)` wahr, falls `(f1 x n y)` wahr ist (Zeile 1). Wenn das Prädikat `(x<0)` erfüllt ist, dann ist `(f x n y)` wahr, falls `(f2 x n y)` wahr ist (Zeile 2)

Sieht man sich die weiteren Programmzeilen an, erkennt man, dass alle Zeilen einheitlich aufgebaut sind. Sie sind auch einheitlich zu interpretieren, d.h. jeweils das erste Prädikat als Konklusion und die restliche Prädikatenfolge als Prämisse in Form einer Konjunktion. Die einzelnen Prädikate, aus denen eine Prämisse “komponiert” ist, nennen wir Bausteinprämissen. Wenn alle Bausteinprämissen erfüllt sind, ist auch die Konklusion erfüllt. Zwischen dem ersten und zweiten Prädikat jeder Zeile kann man also gedanklich einen “verkehrten” (nach links gerichtete Implikationspfeil einfügen. Jede Programmzeile stellt also eine *Hornklausel* in verkürzter Notation dar [16.3]. Im Weiteren nennen wir sie kurz *Klausel*. Damit der Interpreter Anfang und Ende einer Klausel erkennt, muss sie in Klammern gesetzt werden. Die Folge aller Klauseln, die das Gesamtprädikat bilden, muss ebenfalls geklammert werden. Damit der Leser die Prädikate, aus denen eine Klausel “komponiert” ist, leichter erkennt, sind sie in Bild 20.7 fett geklammert.

Zwecks Vereinfachung der weiteren Erläuterungen vereinbaren wir folgende Nummerierung der Prädikate des Programms. Das erste Prädikat (die Konklusion) der *k*-ten Zeile wird mit P_{k0} bezeichnet, die nachfolgenden Prädikate (die Bausteinprämissen der Klausel) der Reihe nach mit P_{k1}, P_{k2},... usf. Beispielsweise bezeichnet P_{11} das Prädikat (<= x 0). Jede Zeile (jede Klausel) kann als Wenn-dann-Satz gelesen werden: "Wenn die Prädikate P_{k1},P_{k2},... wahr sind, m.a.W. wenn alle Bausteinprämissen in der *k*-ten Zeile erfüllt sind, ist P_{k0} wahr". Beispielsweise ist P_{30} wahr, wenn P_{31} und P_{32} wahr sind. In diesem Sinne sagen wir, dass die Klausel in Zeile 3 das Prädikat P_{30} in die beiden Prädikate P_{31} und P_{32} "*dekomponiert*". Durch Zeile 4 wird P_{40} in drei Prädikate dekomponiert. Die Zeilen 1 bis 6 stellen insgesamt die dekomponierte Bedingung dafür dar, dass (f x n y) wahr ist.

Nach diesen Kommentaren wird der Leser erkannt haben, dass den Zeilen Kompositoperatoren entsprechen, z.B. den Zeilen 5 und 6 der pot-Operator in Bild 8.1. Er wird auch erkannt haben, dass Zeile 2 bzw. 3 die Bedingung dafür enthält, dass die Funktion f1 bzw. f2 den Wert y annimmt, und Zeile 6 die Bedingung dafür, dass die pot-Funktion den Wert u annimmt.

Würde man einem Interpreter, der die Sprache versteht, die 6 Zeilen als Auftrag anbieten, ohne den Variablen *x* und *n* Werte zuzuweisen, könnte er im Prinzip seine Arbeit beginnen, doch würde er vor dem Umfang der Arbeit kapitulieren, denn die Anzahl der Tripel, die das Prädikat (f x n y) erfüllen, ist unendlich groß. Der Nutzer muss den Bereich, in dem gesucht werden soll, auf eine endliche Menge einschränken, im Grenzfall auf ein einziges Tripel, beispielsweise auf das Tripel (-2 3 y). Der Interpreter würde dann für *y* den Wert -10 liefern. Wir wollen sein Vorgehen verfolgen, das heißt, wir wollen die interne Semantik des Programms verstehen.

Der Interpreter interpretiert das Programm, indem er der Reihe nach versucht, die Prädikate der Prämissen zu entscheiden. Er beginnt mit P_{10}, also mit (f -2 3 y). Er kann es aber nicht entscheiden; es fehlt ihm an "Wissen". Doch ist P_{10} in P_{11} und P_{12} "dekomponiert". Der Interpreter geht zu P_{11} über, das er entscheiden kann; (<= -2 0) ist wahr. Das Prozentzeichen vor P_{11} weist den Interpreter an, das Prädikat durch seinen Wert zu substituieren. Dabei handelt es sich offensichtlich um ein funktionales, also artfremdes Element des logischen Programms. Da P_{11} den Wert true besitzt, hat es auf die weitere Auswertung kleinen Einfluss und kann gestrichen werden. Bei positivem n, könnte die ganze erste Zeile gestrichen werden, weil P_{10} niemals erfüllt werden könnte.

Nach der Auswertung von P_{11} geht der Interpreter zu P_{12} über. Da er (f1 -2 3 y) nicht entscheiden kann, sucht er nach einer Dekomposition und findet sie in Zeile 3. In Analogie zum Unterprogrammruf könnte man sagen, dass ein nicht entscheidbares Prädikat eine andere Klausel zu Hilfe *ruft*. Die Variablen der "gerufenen" Klausel können, wie die Variablen eines gerufenen Unterprogramms, als *formale Parameter* aufgefasst werden, die durch die *aktuellen Parameter* des "rufenden" Prädikats substituiert werden (ggf. Bezeichnerabgleich [15.16]). Nachdem der Interpreter in der gerufenen Zeile 3 den "formalen Parameter" x durch -2 und n durch 3

substituiert hat, stößt er auf das nichtentscheidbare Prädikat (pot -2 3 y) und findet dessen Dekomposition in Zeile 6. Es handelt sich um eine rekursive Klausel, denn das `pot`-Prädikat tritt sowohl in der Konklusion als auch in der Prämisse auf. Der Interpreter gerät in eine rekursive Iterationsschleife, ebenso wie der CommomLisp-Interpreter bei der Interpretation der Zeile 4 des Programms von Bild 20.3. Er geht nach der gleichen Methode vor, wie sie in Kap. 8.4.6 [8.31] für die Fakultät-Funktion beschrieben wurde, d.h. er geht die Prädikate P_{61} bis P_{64} mehrmals durch. Im ersten Schritt wird P_{61} zu `true` entschieden. Das Dollarzeichen vor P_{62} weist den Interpreter an, das Prädikat als Ergibtanweisung `m:=n-1` zu interpretieren, den Wert von `m` zu berechnen und in den weiteren Prädikaten der betreffenden Zeile `m` durch den berechneten Wert zu substituieren. Die Auswertung von P_{63} erfordert den Rücksprung zu P_{60}, nachdem P_{63} und das als Ergibtanweisung zu interpretierende Prädikat P_{64} mit den aktuellen Parametern "gekellert" worden sind. Im dritten Iterationsschritt ergibt sich das Prädikat `(pot -2 0 u)`, das aufgrund der Zeile 5 ausgewertet werden kann, denn P_{50} ist wahr, da die Klausel keine Prämisse enthält. Folglich wird `(pot -2 0 u)` für `u=1` wahr.

Damit ist der rekursive Teil der Iteration (der "*rekursive Abstieg*") abgeschlossen. Es schließt sich die Ausführung der gekellerten Ergibtanweisungen in umgekehrter Reihenfolge (das "Hochrechnen") an. Es besteht in der dreimaligen Ausführung der Ergibtanweisung `u:=(-2)*v` (in imperativer Notation), wobei für `v` jeweils der im vorangehenden Aufwärtsschritt berechnete `u`-Wert einzusetzen ist. (Man beachte, dass Variablenbezeichner jeweils nur in derjenigen Zeile gelten, in der sie auftreten.) Im letzten Schritt ergibt sich für `u` der Wert `-8`. Nun kann die Ergibtanweisung in P_{32} ausgeführt werden mit dem Ergebnis `y=-10`, und anschließend können P_{30} und P_{10} ausgewertet werden. Der Interpreter kommt zu dem Schluss, dass das Prädikat `(f -2 3 y)` für den Wert `y=-10` erfüllt ist. Durch die äußere Klammerung der Zeilen 1 bis 6 mit vorangestelltem `setq f` wird der Interpreter angewiesen, der Variablen `f` den erhaltenen `y`-Wert zuzuweisen.

Vergleicht man das Programm von Bild 20.7 mit den obigen Pascal- und CommonLisp-Programmen, hat man das Gefühl, als sei das *logische* Programm "*wider alle natürliche Logik*" geschrieben. Das ist nicht verwunderlich. Es ist die Folge davon, dass das logische Programmierparadigma *missbraucht* wurde. Seine Verwendung zur Berechnung der Funktion (20.1) ist *zweckentfremdet*.

Wie wir aus Kap.16.1 wissen, sind logische Sprachen *nicht* für das numerische oder analytische Rechnen entwickelt worden, sondern für das "logische" Schlussfolgern, das Inferenzieren. Hinsichtlich des Inferenzierens ist das logische Paradigma ganz und gar nicht widernatürlich, sondern sehr *natürlich*, und Inferenzprobleme lassen sich in logischen Sprachen sehr kompakt formulieren, was durch das Prolog-Programm von Bild 16.3 zur Lösung des Verwandtschaftsproblems eindrucksvoll demonstriert wird.

Abschließend ist eine Bemerkung zum Paradigmenbegriff am Platze. Der Begriff des Programmierparadigmas hat in erster Linie *akademische* Bedeutung. Er ist von

großem systematisierendem und didaktischem Wert. Die Praxis des Programmierens hält sich nicht an die scharfe Trennung zwischen den Paradigmen, sondern vermischt sie nach Gutdünken. Die Güte eines Programms wird nicht durch seine paradigmatische Reinheit, sondern durch seine Effizienz bestimmt, durch den Aufwand, der für seine Erstellung, Benutzung, Abarbeitung und Wartung erforderlich ist. Welches Paradigma bzw. in welcher Mischung verschiedene Paradigmen in einem Programm zur Anwendung kommen, hängt vom Problem, von der (den) verwendeten Programmiersprache(n) und von den Gewohnheiten des bzw. der Programmierer ab. Dass die Sprachen selber nicht paradigmenrein sind, zeigen die obigen Programmbeispiele. Zum einen zeigen sie, wie ein und dieselbe Sprache an verschiedene Paradigmen *angepasst* werden kann. Zum anderen zeigen sie, dass praktisch jede Sprache paradigmenfremde Sprachelemente enthält. Das am häufigsten anzutreffende Beispiel hierfür sind arithmetische (d.h. *funktionale*) Ausdrücke, die in fast allen Sprachen erlaubt sind.

Bevor wir unsere flüchtige "Besichtigung" einiger Programmiersprachen und Programmbeispiele beenden[2], soll nun noch ein umfangreicheres und leistungsfähigeres Programm vorgestellt und analysiert werden.

20.3 Objektorientiertes Programm in Borland-Pascal

Das folgende Programm mit dem Bezeichner `opnetz` ist bedeutend leistungsfähiger als die bisherigen Beispielprogramme. Es ist in BorlandPascal 7.0 geschrieben.[3]

2 Für ein tieferes Studium steht eine umfangreiche Literatur zur Verfügung, genannt seien folgende Bücher: [Louden 94],[Schefe 87],[Abelson 91],[Stoyan 88,91].

3 Das Programm ist eine überarbeitete Version eines weit eleganteren, dafür aber schwerer lesbaren und schwerer verstehbaren Programms von Herrn Bernd Dupal. In [Borland 93] findet der Leser die Definition der Sprache BorlandPascal 7.0.

```
100  Program opnetz;
101  Uses Crt;
102  Const
103    maxop    = 7;
104    maxod    = 8;
105    maxsig   = 8;

106  Type
107    Pop      = ^Top;
108    Top      = Object
109                 opnummer :Shortint;
110                 platzinod1,   platzinod2,
                    platzoutod1,  platzoutod2,
                    platzinsig1,  platzinsig2,
                    platzoutsig1, platzoutsig2  :Shortint;
111                 Procedure berechnen; Virtual;
112                 Constructor opkopp(iplatzinod1, iplatzinod2,
                      iplatzoutod1,  iplatzoutod2,
                      iplatzinsig1,  iplatzinsig2,
                      iplatzoutsig1, iplatzoutsig2 :Shortint);
113                 Destructor loeschen;
114               End;
115    Pmul     = ^Tmul;
116    Tmul     = Object(Top)
117                 Constructor opkopp(iplatzinod1, iplatzinod2,
                      iplatzoutod1, iplatzoutod2,
                      iplatzinsig1, iplatzinsig2,
                      iplatzoutsig1, iplatzoutsig2 :Shortint);
118                 Procedure berechnen; Virtual;
119               End;
120    Piweiche = ^Tiweiche;
121    Tiweiche = Object(Top)
122                 iterationszahl :Shortint;
123                 Constructor opkopp(iplatzinod1, iplatzinod2,
                      iplatzoutod1, iplatzoutod2,
                      iplatzinsig1, iplatzinsig2,
                      iplatzoutsig1, iplatzoutsig2 :Shortint);
124                   Procedure berechnen; Virtual;
125               End;
126    Paweiche = ^Taweiche;
127    Taweiche = Object(Top)
128                 Procedure berechnen; Virtual;
129               End;
130    Psin     = ^Tsin;
131    Tsin     = Object(Top)
132                 Procedure berechnen; Virtual;
133               End;
134    Psync    = ^Tsync;
135    Tsync    = Object(Top)
```

```
136                Procedure berechnen; Virtual;
137              End;
138    Padd      = ^Tadd;
139    Tadd      = Object(Top)
140                Procedure berechnen; Virtual;
141              End;
142    Tsop      = Object
143                Procedure opord(Piop :Pop;
                     opnummer :Shortint);
144                Procedure steuern;
145                Procedure beenden;
146              End;
147  Var
148    odspei    :Array[1..maxod] of Real;
149    sigspei   :Array[1..maxsig] of Shortint;
150    opliste   :Array[1..maxop] of Pop;
151    sop       :Tsop;

{Ende der Deklarationen. Es folgen die Prozeduren}

200  Procedure Tsop.opord;
201  Begin
202    opliste[opnummer] := Piop;
203    Piop^.opnummer    := opnummer;
204  End;
205  Procedure Tsop.steuern;
206  Var  t :Shortint;
207  Begin
208    Writeln('Berechnung läuft');
209    While sigspei[maxsig]=0 Do
210    Begin
211      For t := 1 To maxop Do
212      Begin
213        If sigspei[opliste[t]^.platzinsig1]=1 Then
214          Begin
215            opliste[t]^.berechnen;
216          End;
217       End;
218    End;
219  End;
220  Procedure Tsop.beenden;
221  Var t :Shortint;
222  Begin
223    For t := 0 To maxop Do
224    Begin
225      Dispose(opliste[t],loeschen);
226    End;
227  End;
```

```
228  Constructor Top.opkopp;
229  Begin
230    platzinod1   := iplatzinod1;
231    platzinod2   := iplatzinod2;
232    platzoutod1  := iplatzoutod1;
233    platzoutod2  := iplatzoutod2;
234    platzinsig1  := iplatzinsig1;
235    platzinsig2  := iplatzinsig2;
236    platzoutsig1 := iplatzoutsig1;
237    platzoutsig2 := iplatzoutsig2;
238  End;
239  Procedure Top.berechnen; Begin End;
240  Destructor Top.loeschen; Begin End;

300  Constructor Tmul.opkopp;
301  Begin
302    Inherited opkopp(iplatzinod1, iplatzinod2, iplatzoutod1,
         iplatzoutod2, iplatzinsig1, iplatzinsig2,
         iplatzoutsig1, iplatzoutsig2);
303    odspei[platzinod2] := 1.0;
304  End;
305  Procedure Tmul.berechnen;
306  Var  x1, x2, y :Real;
307  Begin
308    x1 := odspei[platzinod1];
309    x2 := odspei[platzinod2];
310    y := x1*x2;
311    odspei[platzoutod1]   := y;
312    sigspei[platzinsig1]  := 0;
313    sigspei[platzoutsig1] := 1;
314  End;

315  Procedure Tsin.berechnen;
316  Var  x1, y1 :Real;
317  Begin
318    x1 := odspei[platzinod1];
319    y1 := Sin(x1);
320    odspei[platzoutod1]   := y1;
321    sigspei[platzinsig1]  := 0;
322    sigspei[platzoutsig1] := 1;
323  End;

324  Procedure Tadd.berechnen;
325  Var  x1, x2, y1 :Real;
326  Begin
327    x1 := odspei[platzinod1];
328    x2 := odspei[platzinod2];
329    y1 := x1+x2;
330    odspei[platzoutod1]   := y1;
```

```
331    sigspei[platzinsig1]  := 0;
332    sigspei[platzoutsig1] := 1;
333  End;

400  Constructor Tiweiche.opkopp;
401  Begin
402    Inherited opkopp(iplatzinod1, iplatzinod2, iplatzoutod1,
         iplatzoutod2, iplatzinsig1, iplatzinsig2,
         iplatzoutsig1, platzoutsig2);
403    iterationszahl := 0;
404  End;
405  Procedure Tiweiche.berechnen;
406  Var maxiterationszahl :Shortint;
407  Begin
408    maxiterationszahl := Shortint(Trunc(odspei[platzinod2]));
409    Inc(iterationszahl);
410    If iterationszahl> maxiterationszahl Then
411    Begin
412      odspei[platzoutod2] := odspei[platzinod1];
413      sigspei[platzinsig1] := 0;
414      sigspei[platzoutsig2] := 1;
415    End;
416    Else
417    Begin
418      odspei[platzoutod1]   := odspei[platzinod1];
419      sigspei[platzinsig1]  := 0;
420      sigspei[platzoutsig1] := 1;
421   End
422  End;

423  Procedure Taweiche.berechnen;
424  Var  x1 :Real;
425  Begin
426    x1 := odspei[platzinod1];
427    If x1 <= 0 Then
428    Begin
429      odspei[platzoutod1] := x1;
430      sigspei[platzoutsig1] := 1;
431    End
432    Else
433    Begin
434      odspei[platzoutod2] := x1;
435      sigspei[platzoutsig2] := 1;
436    End;
437    sigspei[platzinsig1] := 0;
438  End;

439  Procedure Tsync.berechnen;
440  Var u1, u2 :Shortint;
```

```
441  Begin
442    u1 := sigspei[platzinsig1];
443    u2 := sigspei[platzinsig2];
444    If u1*u2 = 1 Then
445    Begin
446      sigspei[platzinsig1] := 0;
447      sigspei[platzinsig2] := 0;
448      sigspei[platzoutsig1] := 1;
449    End;
450  End;

{Hauptprogramm}
{Installation des Netzes}
500  Begin
501    sop.opord(New(Pmul,      opkopp(1,4,3,0, 1,0,2,0)), 1);
502    sop.opord(New(Piweiche, opkopp(3,2,4,5, 2,0,1,5)), 2);
503    sop.opord(New(Paweiche, opkopp(1,0,6,7, 3,0,6,5)), 3);
504    sop.opord(New(Psin,      opkopp(7,0,6,0, 4,0,6,0)), 4);
505    sop.opord(New(Psync,     opkopp(0,0,0,0, 5,6,7,0)), 5);
506    sop.opord(New(Psync,     opkopp(0,0,0,0, 6,5,7,0)), 6);
507    sop.opord(New(Padd,      opkopp(5,6,8,0, 7,0,8,0)), 7);
508    sigspei[1] := 1;
509    sigspei[3] := 1;

{Berechnung eines Funktionswertes}
600    Write('x='); Readln(odspei[1]);
601    Write('n='); Readln(odspei[2]);
602    sop.steuern;
603    Writeln('Berechnung beendet,
         Resultat=',odspei[maxod]:5:4);
604    sop.beenden;
605  End.
```

Bild 20.8 BorlandPascal-Programm

Das Programm soll ausführlich kommentiert werden. In dem Programm sind alle Zeichenketten, die mit einem großen Buchstaben beginnen, Sprachelemente von BorlandPascal. Eine Ausnahme bildet die Verwendung der Buchstaben T und P. Ein Bezeichner, der mit einem T bzw. P beginnt, ist der Name eines Typs bzw. eines Zeigers, auch Pointers genannt (eines "Zeiger-Typs" in der Nomenklatur von BorlandPascal). Diese Vereinbarung ist nicht Bestandteil der Sprachsyntax, sondern des Programmierstils. Die Syntax von BorlandPascal unterscheidet nicht zwischen großen und kleinen Buchstaben. Bei den folgenden Erläuterungen werden die Zeiger zunächst ignoriert.

Die Zeilennummerierung gehört nicht zum Programm; sie dient der Orientierung bei den folgenden Erläuterungen. Durch die erste Ziffer der Zeilennummer (1 bis 6)

wird das Programm in 6 Abschnitte unterteilt. Der letzte Abschnitt beinhaltet den Auftrag und die Ausführung einer Wertberechnung der Funktion (20.1) für die Werte `x=-2` und `n=3`, die über den Bildschirm eingegeben werden (Zeilen 600 und 601)[4]. Durch Zeile 602 wird die Berechnung gestartet. Das Ergebnis wird auf dem Bildschirm angezeigt (Zeile 603).

Die Programmerstellung beginnt mit dem Entwurf des erweiterten Operatorennetzes (des erweiterten Datenflussplans) von Bild 20.9 und des erweiterten Petrinetzes (Aktionsfolgeplans) von Bild 20.10. Die Erweiterungen bestehen in der Einführung zusätzlicher, im Folgenden zu erklärender Symbole. Angesichts der Zielstellung, ein Programm zu schreiben, das die “*Kooperation*” zwischen Objekten beschreibt, deren Tätigkeit *nicht* zentral gesteuert wird, hat der Leser die Bedeutung des Petrinetzes vielleicht schon erkannt. Wir werden auf sie weiter unten eingehen.

Zu Bild 20.9. Ein Vergleich von Bild 20.9 mit Bild 8.1 zeigt Folgendes. Die Zweigeweiche vor dem Sinusoperator ist zum Rhombus mit der Bezeichnung `awei` (Abkürzung für Alternativweiche) und die Zweigeweiche nach dem Multiplizierer zum Rhombus mit der Bezeichnung `iwei` (Abkürzung für Iterierweiche) geworden. Jeder Rhombus symbolisiert einen Steueroperator, der in Bild 8.1 nicht explizit dargestellt, sondern in den Steueroperator f-sop integriert ist. Letzterer ist also dekomponiert, zumindest teilweise, und zwei seiner Bausteine sind in das Operatorennetz eingefügt. Dadurch werden die entsprechenden Flussknoten, also die beiden Zweigeweichen zu Operatoren, die wir **Weichenoperatoren** nennen. Sie dienen nicht der Verarbeitung, sondern lediglich der steuerbaren Weitergabe der Operanden. (In Kap.8.1 war bereits angemerkt worden, dass Flussknoten als Operatoren aufgefasst werden können.) Die Operatoren `mul`, `sin` und `add` sind Arbeitsoperatoren. Sie werden durch große Quadrate symbolisiert. Die kleinen Quadrate mit der fetten Eingabeseite symbolisieren, wie üblich, Speicher für je einen Operanden. Sie werden im Weiteren Operandenplätze genannt. Die Nummerierung der Operatoren und Operandenplätze ist arbiträr (beliebig, aber ein für allemal verbindlich wählbar). Der kleine Kreis vor dem Operandenplatz 6 ist eine Sammelweiche, die als zweite Weiche einer Alternativmasche nicht gestellt zu werden braucht [8.1], da immer nur ein einziger Operand die Masche durchläuft, denn der f-op berechnet definitionsgemäß niemals mehrere Funktionswerte gleichzeitig (in dem Petrinetz nicht ausgewiesen).

Zu Bild 20.10. Das Bild zeigt ein Petrinetz, das eine Erweiterung der in Kap.8.2.2 eingeführten starren Petrinetze zu einem steuerbaren Petrinetz darstellt. Die Kreise symbolisieren wie üblich Markenplätze und die Quadrate Transitionen. Die Rhomben und Dreiecke symbolisieren Steuertransitionen. Durch sie wird das Petrinetz steuerbar. Steuertransitionen steuern den Markenstrom nach Maßgabe von Steuer-

4 Der besseren Lesbarkeit halber wurde in Zeile 600 der Bezeichner `x`, der eigentlich für private Variablen reserviert ist, inkonsequenterweise für einen Eingabeoperanden des Programms verwendet.

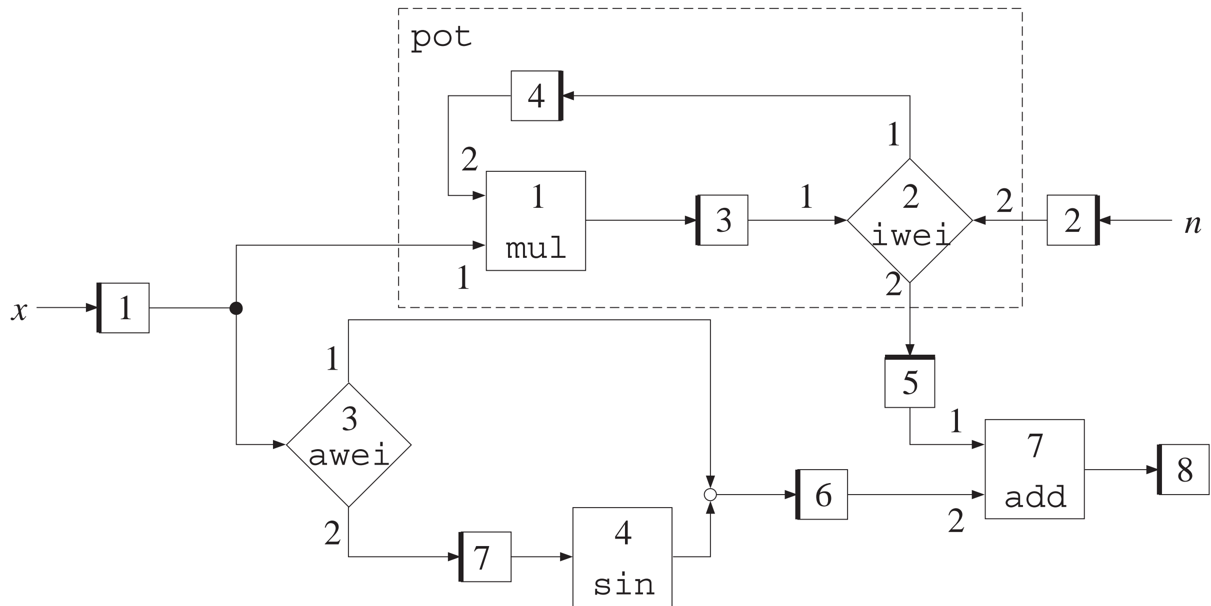


Bild 20.9 Erweitertes Operatorennetz. Große Quadrate und Rhomben - Operatoren; kleine Quadrate - Operandenplätze; Arbeitsoperatoren: 1 - Multiplizierer (`mul`), 4 - Sinusoperator (`sin`), 7 - Addierer (`add`); Steueroperatoren: 2 - Iterierweiche (`iwei`), 3 - Alternativweiche (`awei`), 5 und 6 - Synchronisierer (`sync1, sync2`).

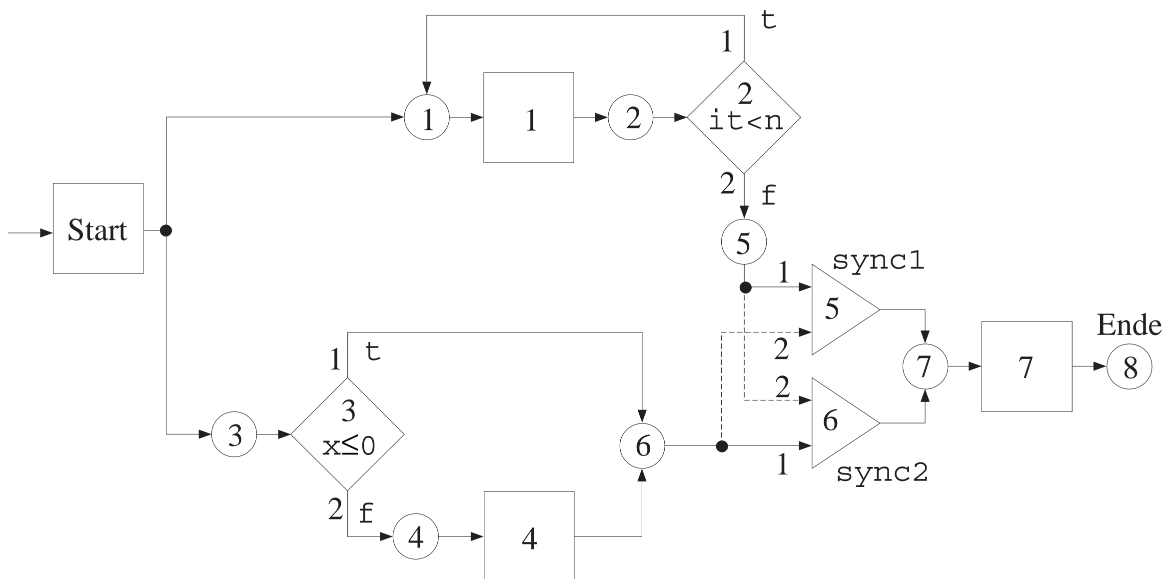


Bild 20.10 Steuerbares Petrinetz. Quadrate und Rhomben - Transitionen, die Nummerierung entspricht der von Bild 20.9; Kreise - Signalplätze (Markenplätze); in der Iterierweiche: it - Iterationszahl.

prädikaten. Die Rhomben entsprechen den Weichenoperatoren in Bild 20.9. In ihnen sind die Steuerprädikate der entsprechenden Zweigeweichen (`awei` bzw. `iwei`) angegeben. Die Steuertransitionen `sync1` und `sync2` bilden ein Paar mit gemeinsamem Ausgabeplatz und führen zusammen die synchronisierende Funktion der Vereinung vor dem Addierer in Bild 8.1 aus. Jeder der beiden Synchronisierer gibt seine Eingabemarke nur dann weiter, wenn auch der Eingabeplatz des Partners mit einer Marke belegt ist. Die beiden dafür notwendigen zusätzlichen Übergabepfeile sind gestrichelt gezeichnet. Man beachte, dass in der Vereinung vor dem Multiplizierer die beiden Eingabeoperanden nicht synchronisiert zu werden brauchen, da der externe Eingabeoperand x ständig auf dem Operandenplatz 1 zur Verfügung steht.

Die Strukturen beider Netze werden dem Computer in den Zeilen 501 bis 507 mitgeteilt. In jeder Zeile wird ein Bausteinoperator in das Netz eingekoppelt. Die Prozedur `New` instanziert je eine Instanz (ein Exemplar) eines Operatortyps. Die Prozedur `opord` ordnet die Operatorinstanzen gemäß ihrer Nummern (`opnummer`) in die Operatorenliste (`opliste`) ein. Die jeweilige Nummer ist die letzte Zahl in der betreffenden Zeile. Sie stimmt mit der Nummer des entsprechenden Operators in Bild 20.9 überein. Die Prozedur `opord` ist in Zeile 143 deklariert und in den Zeilen 200 bis 204 programmiert. Die Abkürzung `iop` kann als "eine Instanz eines Operators" (bzw. einer Transition) gelesen werden.

Die Prozedur `opkopp` koppelt eine Operatorinstanz in das Netz ein. Sie ist in Zeile 112 deklariert und in den Zeilen 228 bis 238 programmiert. (Anstelle von `Procedure` muss das Schlüsselwort `Constructor` verwendet werden, um dem Compiler anzuzeigen, dass es sich bei der Prozedur um die "*Konstruktionsvorschrift*" des Operators handelt. Jedem `Constructor` muss im Programm später ein zugeordneter `Destructor` folgen, der den "Abbau" (Löschung) der betreffenden Instanz auslöst.

In jeder der Zeilen 501 bis 507 werden jeweils für eine Operatorinstanz den 8 Variablen der Prozedur `opkopp` Werte zugewiesen (in der durch Zeile 112 festgelegten Reihenfolge), für jeden Operator also ein Tupel von 8 Werten. Gemeinsam bilden die Tupel eine Tabelle, die sog. *Kopplungstabelle*. Mit dem Ausfüllen der Kopplungstabelle (dem Notieren der Zeilen 501 bis 507) legt der Programmierer die Strukturen des Operatorennetzes und des Petrinetzes, m.a.W. den Datenflussgraph und den Aktionsfolgegraph fest.

Die ersten vier Spalten der Kopplungstabelle enthalten die Nummern der Ein- und Ausgabeoperandenplätze des betreffenden Operators (gemäß Bild 20.9) und die letzten vier Spalten die Nummern der Ein- und Ausgabemarkenplätze der betreffenden Transition (gemäß Bild 20.10). Die Bezeichner `platzinod` und `platzoutod` bedürfen keines Kommentars, doch die Bezeichner `platzinsig` und `platzoutsig` sowie der Index 1 bzw. 2 müssen erklärt werden.

Eine Marke auf einem Markenplatz kann als Signal an den übergeordneten Steueroperator sop (siehe unten) aufgefasst werden, und zwar in doppeltem Sinne; als Ausgabesignal hat sie die Bedeutung einer Fertigmeldung "Operation ist ausge-

führt" und als Eingabesignal hat sie die Bedeutung einer Bereitmeldung "Operation kann ausgeführt werden". Beispielsweise bedeutet eine Marke auf Platz 2 "`mul` ist ausgeführt" und "`iwei` kann ausgeführt werden". So interpretiert spielen die Marken die gleiche Rolle wie die Signale der USB-Methode. Dies ist der Grund dafür, dass im Weiteren anstelle von Marken von *Signalen* (Ein- bzw. Ausgabesignalen) und anstelle von Markenplätzen von *Signalplätzen* (Ein- bzw. Ausgabesignalplätzen) gesprochen wird. Ein Fertig- bzw. Bereitsignal ist ein Bit. Dem Signalwert 1 entspricht das Vorhandensein einer Marke auf dem betreffenden Platz des Petrinetzes.

Es wird davon ausgegangen, dass ein Operator höchstens zwei Ein- und zwei Ausgabeoperandenplätze und eine Transition höchstens zwei Ein- und zwei Ausgabesignalplätze besitzt. Die Eingabeoperanden eines Operators werden mit `inod1` und `inod2` und ihre Plätze mit `platzinod1` und `platzinod2` bezeichnet. Entsprechendes gilt für die Ausgabeoperanden und die Ein- und Ausgabesignale sowie für deren Plätze. Der Index 1 bzw. 2 ist in den Bildern 20.9 und 20.10 an den jeweiligen Ein- und Ausgängen der Operatoren bzw. Transitionen angegeben. Beispielsweise nimmt die Variable `platzinod2` des Constructors `opkopp` für den Multiplizierer den Wert 4 an (siehe erste Zeile, zweite Spalte der Kopplungstabelle) in Übereinstimmung mit der Nummer des betreffenden Oprandenplatzes in Bild 20.9. Mit `inod` und `outod` kann Input und Output eines schwarzen Kastens (eines abstrakten Automaten) assoziiert werden. Wir erinnern daran, dass nach dem objektorientierten Paradigma die Kooperationspartner (die Operatoren des Netzes) als "schwarze Kästen" betrachtet werden, deren Inneres von der Umwelt abgekapselt ist.

Wir wollen uns die Kopplungstabelle etwas näher ansehen. Eine Null in der Tabelle bedeutet, dass der betreffende Platz überflüssig ist, weil der betreffende Operand bzw. das betreffende Signal nicht existiert. Da Arbeitsoperatoren nur einen einzigen Ausgabeoperanden besitzen, steht in der 4. Spalte für die Instanzen 1, 4 und 7 eine Null. Im Falle der Zweigeweichen (Zeile 502 und 503) stehen in der 3. und 4. Spalte die beiden möglichen Ausgabeoperandenplätze (4 und 5 bzw. 6 und 7), zwischen denen die Zweigeweiche entscheidet.

In der 5. bis 8. Spalte stehen der Reihe nach die Werte der Variablen `platzinsig1, platzinsig2, platzoutsig1 platzoutsig2`, also die Nummern der Ein- und Ausgabeplätze der Transitionen. Beispielsweise weist Zeile 501 im Falle des Multiplizieres der Variablen `platzoutsig1` den Wert 2 zu (vgl. Bild 20.10; der Platz nach der Transition 1 hat die Nummer 2). Abgesehen von den Synchronisierern besitzt eine Transition gemäß Bild 20.10 einen einzigen Eingabeplatz und maximal zwei Ausgabeplätze. Ein Synchronisierer besitzt einen zweiten Eingabeplatz für das Signal, auf das gewartet werden muss. Dementsprechend steht in der 6. Spalte der Zeilen 505 und 506 keine Null.

Am Beispiel des Multiplizieres (Operatornummer 1) wollen wir kontrollieren, ob die Zahlenfolge `1,4,3,0, 1,0,2,0` in Zeile 501 den Bildern 20.9 und 20.10

entspricht. Die Zahl `1` am Ende der Zeile 501 zeigt an, dass durch die Zeile der `mul`-Operator eingekoppelt wird. Wie man aus Bild 20.9 ablesen kann, holt sich der Multiplizierer den Wert von `inod1` bzw. `inod2` von den Operandenplätzen 1 bzw. 4 und legt den Wert von `outod` auf Platz 3 ab. Außerdem verschiebt er gemäß Bild 20.10 die Eingabemarke von Platz 1 nach Platz 2. Da der Multiplizierer nur einen Ausgabeoperandenplatz und je einen Eingabe- und Ausgabesignalplatz besitzt, gilt `platzoutod2 = platzinsig2 = platzoutsig2 = 0`. Die Zahlenfolge `1,4,3,0, 1,0,2,0` entspricht also den Bildern 20.9 und 20.10.

Die Übereinstimmung der Zahl in Zeile1/Spalte3 (dort steht die Zahl `3`) mit der Zahl in Zeile2/Spalte1 zeigt an, dass der Ausgabeoperand des Multiplizierers an die Iterierweiche weitergeleitet wird. Multiplizierer und Iterierweiche bilden gemeinsam eine Iterationsschleife. Das Ergebnis eines Iterationsschrittes (einer Multiplikation) wird auf dem Ausgabeoperandenplatz `platzoutod1` des Multiplizierers (Zeilen 310 und 311), also auf Operandenplatz 3 abgespeichert und von der Iterierweiche auf den Operandenplatz 4 weitergegeben, solange die Iterationszahl `it` die maximale Iterationszahl `n` nicht erreicht ist, m.a.W. solange das Prädikat `it<n` erfüllt ist (vgl. Bild 20.10 und Zeile 410). Sobald es nicht mehr erfüllt ist, wird das Ergebnis der Iteration (die n-te Potenz von x) an den Operandenplatz 5 weitergegeben. Vor Beginn der Iteration (des Potenzierens) muss auf den Operandenplatz 4 der Anfangswert 1 eingespeichert werden (Zeile 303).

Nach diesen wenigen Erläuterungen wird der Leser vielleicht schon in der Lage sein, die Arbeitsweise der übrigen Operatoren im Prinzip zu verstehen. Um sie im Detail anhand der jeweiligen Operationsvorschriften (`Procedure berechnen`) nachvollziehen zu können, bedarf es weiterer Erläuterungen. Die Prozeduren der Arbeitsoperatoren sind im dritten (die Zeilennummern beginnen mit einer 3), die der Steueroperatoren im vierten Programmabschnitt zusammengefasst.

Eine wichtige Implementierungsidee besteht in der Zusammenfassung aller Operandenplätze zu einer (programmtechnischen) Speichereinheit. Es ist die gleiche Idee, die zum Konzept des zentralen Arbeitsspeichers und zum Begriff des abstrakten Automaten (Kap.8.2.3) geführt hat, die Idee der Zentralisierung der verteilten Operandenplätze eines Operatorennetzes. Die Speichereinheit für die Operanden hat den Bezeichner `odspei`. In Zeile 148 ist sie als Variable vom Typ `Array` deklariert. Der Datentyp `Array` ist ein strukturierter Datentyp oder - in unserer Sprechweise - ein Kompositdatentyp. Er stellt eine Folge indizierter Komponenten (Bausteine) dar. Jede Komponente entspricht einer indizierten Variablen. Die i-te Komponente kann als “Arrayplatz” der i-ten Variablen aufgefasst werden. Alle Variablen müssen vom gleichen Typ sein. Die Variablen des Arrays `odspei` sind vom Typ Real, denn die Operanden besitzen reelle Werte. Die Notation `Array[1..maxod]` bedeutet, dass der Index von 1 bis `maxod` läuft. Der Wert von `maxod` ist in Zeile 104 festgelegt.

Es wird nun der Operandenplatz mit der Nummer i mit dem Arrayplatz mit dem Index i identifiziert. Beispielsweise ist `platzoutod1` des Multiplizierers mit `odspei[3]` (das ist die Notation für den dritten Platz des Array odspei) zu

identifizieren. Beides sind Variablenbezeichner. Entsprechend bezeichnen `odspei[1]` und `odspei[2]` die beiden Eingabeoperanden des Multiplizierers. Die Multiplikationsvorschrift könnte also (im Sinne des imperativen Paradigmas) als Ergibtanweisung `odspei[3] := odspei[1]*odspei[2]` notiert werden. Das aber widerspräche der Idee der Kapselung und dem objektorientierten Programmierparadigma. Es entfiele nämlich die Unterscheidung zwischen Außenwelt und Innenwelt eines Objekts (Operators), zwischen öffentlichen und privaten Operanden (Variablen). Um diese Unterscheidung im Programm zu dokumentieren, verwenden wir für die Operanden aus externer Sicht die Bezeichner `inod` bzw. `outod` und aus interner Sicht die Bezeichner der USB-Methode `x` bzw. `y`.

Wenn ein Operator einen Auftrag ausführen soll, den er von einem Kooperationspartner erhält, sind die Variablen, die der Auftrag enthält, naturgemäß öffentlich. Die Variablenwerte des Auftrags müssen also entsprechenden privaten Variablen zugewiesen werden (z.B. in den Zeilen 308 und 309 der Berechnungsprozedur des Multiplizierers). Ein Operator kann über weitere private Variable verfügen, evtl. sogar über sehr viele. Die Iterierweiche verfügt über die private Variable `iterationszahl` (Zeile 122). Öffentliche Variablen, die ein Operator lediglich weiterleitet, ohne sie zu verändern, brauchen nicht in private Variablen überführt zu werden. Das trifft für die Operanden von Steueroperatoren zu. Das Zuweisen öffentlicher Variablenwerte an private Variablen entspricht in der Unterprogrammtechnik dem Zuweisen aktueller Parameterwerte an formale Parameter.

In Analogie zum Operandenspeicher wird eine zweite Speichereinheit für die Signale angelegt. In Zeile 149 ist sie als Variable mit dem Bezeichner `sigspei` vom Typ `Array` deklariert. Die Komponenten des Array sind vom Typ `Shortint` (kurze Integerzahl). Signale zeigen dem Steueroperator sop den Bearbeitungszustand (bereit bzw. fertig) an. Zusammen mit einer Auftragsausführung (Operationsausführung) wird das Bereitsignal (das Bit 1) vom Eingabeplatz auf den Ausgabeplatz verschoben; es erübrigt sich die Einführung entsprechender privater Variablen. Eine Ausnahme bilden die Synchronisierer, die ihre Eingabesignale nicht nur weiterleiten, sondern miteinander verrechnen (Zeile 444). Aus externer Sicht werden für die Ein- bzw. Ausgabesignale die Bezeichner `insig` bzw. `outsig` verwendet und aus interner Sicht die Bezeichner `u` bzw. `v`. (Eine Variable `v` tritt in dem Programm nicht auf. Auch auf die Variablen `u1` und `u2` hätte verzichtet werden können.)

Der Leser wird nun in der Lage sein, den Operanden- bzw. Signalfluss (Markenfluss) durch das Operatoren- bzw. Petrinetz aus dem Programm abzulesen. Ein Blick auf die Berechnungsprozeduren im 3. und 4. Programmabschnitt (z.B. auf die Prozedur `Tmul.berechnen` der Zeilen 305 bis 314) zeigt, dass die Ausgabeoperandenwerte eines Operators und die Ausgabesignalwerte der entsprechenden Transition durch eine einzige Prozedur berechnet werden. Ein Operator wird also mit der Transition zu *einer* programmierungstechnischen Verarbeitungseinheit zusammengefasst. Durch diese Zusammenfassung entsteht ein Objekt.

Die Objekte, mit denen unser Programm arbeitet, sind in den Zeilen 108 bis 146 deklariert. Beispielsweise wird der Multiplizierer in Zeile 116 als abgeleitetes Objekt des Objekts `op` (Operator) deklariert. Dieser Satz ist nicht ganz exakt. Richtig muss er lauten: “Zeile 116 deklariert den Typ Multiplizierer `Tmul` als abgeleiteten Typ des Objekttyps `Top`”. Statt Objekttyp wird häufig auch Objektklasse gesagt [1].[5] Dann lautet der Satz: “Zeile 116 deklariert die Klasse der Multiplizierer als Unterklasse der Klasse der Operatoren”.

Ein Multiplizierer ist eine Präzisierung (im Sinne von Bild 5.4) eines Operators. Er *erbt* vom Operator dessen Merkmale (Fähigkeiten) und fügt neue hinzu [18.4]. Beispielsweise erbt er vom Operator die Variablen des Constructors `opkopp` (Zeile 302; `Inherited` = geerbt) und fügt zum Constructor des Operators die Setzung des Anfangswertes von `odspei[platzinod2]` hinzu (Zeile 303). Durch die Zeilen 501 bis 507 wird je eine Instanz einer Klasse instanziert. Durch die Zeilen 505 und 506 werden zwei Instanzen der Klasse `Tsync` instanziert.

Die Erläuterungen zum Multiplizierer lassen sich auf die übrigen Operatoren übertragen. Nur zum Synchronisierer ist eine zusätzliche Bemerkung erforderlich. Seine Berechnungsprozedur (Zeilen 439 bis 450) enthält zwei Eingabesignalplätze. Dabei bezeichnet `platzinsig1` den eigenen Eingabeplatz, wie er in Bild 20.10 eingezeichnet ist, während `platzinsig2` den Eingabeplatz des Partners bezeichnet, mit dem gemeinsam die Syhchronisierung der Operanden bewerkstelligt wird.

Geht man nach diesen Erläuterungen das gesamte Programm durch, wird dessen Aufbau verständlich. Man erkennt unter anderem, dass in den Zeilen 106 bis 146 die Operatortypen als Objektklassen deklariert werden (dazu die jeweiligen Zeiger; s.u.) und dass in den Zeilen 142 bis 146 die Objektklasse `Tsop` deklariert wird, von der aber keine Unterklassen abgeleitet werden und nur das einzige Exemplar `sop` instanziert wird (Zeile 151). Dabei handelt es sich um den zentralisierten Rest des teilweise dezentralisierten Steueroperators f-sop von Bild 8.1. Die Aufgabe des hier deklarierten Steueroperators sop besteht nur noch darin, die bereiten Operatoren (d.h. die Operatoren mit dem Eingabesignal 1) der Reihe nach zu starten (Zeilen 205 bis 219) Die Reihenfolge ist arbiträr, d.h. beliebig aber verbindlich festlegbar. In den Zeilen 305 bis 450 sind die Methoden (Prozeduren) der Objektklassen programmiert und in den Zeilen 500 bis 509 werden die erforderlichen Instanzen der Objektklassen instanziert einschließlich der Anfangswertzuweisungen. Danach erst beginnt das eigentliche Hauptprogramm. Es erhebt sich die Frage, warum ein solcher “Vorbereitungsaufwand” getrieben wird, und insbesondere die Frage:

---

5 Das ist statthaft, obwohl ein Typ eine Eigenschaft festlegt, während eine Klasse eine Menge ist. Denn ein Typ definiert eine ihm entsprechende Klasse und umgekehrt. Die einem Typ entsprechende Klasse ist die Menge aller Exemplare des betreffenden Typs; vgl. auch Bild 5.4).

Aus welchem Grunde sind zwei Netze erforderlich, das Operatorennetz und das Petrinetz, und warum muss sowohl der Operandenfluss als auch der Markenfluss simuliert werden?

Die Frage ist insofern berechtigt, als jedes der beiden Netze alle erforderlichen Informationen enthält, um ein Programm zur Berechnung der Funktion $f(x,n)$ zu schreiben (die Bausteinoperationen als bekannt vorausgesetzt). Das Operatorennetz stellt den Datenflussplan dar und kann in ein funktionales Programm überführt werden, z.B. in ein Lisp-Programm (siehe die Bilder 20.3 und 20.4). Das Petrinetz beschreibt den Aktionsfolgeplan. Es kann als Programmablaufplan aufgefasst und in ein imperatives Programm überführt werden, z.B. in ein Pascal-Programm (siehe die Bilder 20.1 und 20.2).

Um die Frage zu beantworten, erinnern wir uns an Kap.19.3, wo die Möglichkeiten der Dezentralisierung der Steuerung diskutiert wurden. Die dortigen Schlussfolgerungen werden jetzt relevant, denn unser Programm soll das *selbständige*, d.h. *nicht zentral gesteuerte* Kooperieren von Objekten beschreiben. In Kap.19.3 hatten wir festgestellt: *Bei vollständiger Dezentralisierung stellen die einzelnen Operatoren selbständige Akteure dar, die sich auch selber starten. Ein ruhender Operator muss also erkennen, ob er eine Operationsausführung beginnen kann. Er muss lediglich darüber informiert werden, wohin er seine Ausgabeoperanden weiterzugeben hat, beispielsweise dadurch, dass ihm der Datenflussplan (bzw. ein Ausschnitt davon) verfügbar gemacht wird. [....] Damit ein Arbeitsoperator erkennt, wann er sich selbst starten kann, muss er ständig kontrollieren, ob ihm die Operanden für die nächste Operation übergeben sind, d.h. ob sie sich in den dafür vorgesehenen Operandenplätzen befinden* (Zitat von [19.3]).

Auf den ersten Blick mag es so scheinen, als sei das Petrinetz überflüssig. Tatsächlich ist das Petrinetz notwendig, weil ein Objekt (Operator) die geforderte Kontrolle der Eingabeoperandenplätze nicht ausführen kann. Es kann nicht erkennen, ob ein Eingabeoperand eingetroffen ist, denn dazu müssten mit der Entnahme von Operanden die entsprechenden Operandenplätze *geleert* werden, so wie die Eingabeoperandenplätze von Fertigungsoperatoren bei Operandenentnahme *geleert* werden. Ein Operandenplatz (Speicherplatz) eines IV-Systems mit statischer Codierung ist niemals "*leer*", in ihm ist *immer* eine Bitkette gespeichert. Ein Speicherplatz kann nicht *geleert*, sondern nur *gelöscht* werden, d.h. es kann die Bitkette 000... eingespeichert werden. Zwar kann ein Objekt erkennen, ob der Inhalt eines Eingabeplatzes mit dem vorangehenden, bereits bearbeiteten Inhalt übereinstimmt. Doch auch wenn er übereinstimmt, kann das Objekt nicht entscheiden, ob es sich um den alten oder einen neuen Inhalt handelt. Es ist also notwendig, dass ein Objekt bei Übergabe eines Operanden den Adressaten informiert. Dafür reicht ein Bit (eine Marke) aus, das in einen dafür vorgesehenen Ein-Bit-Speicher (Markenplatz) eingetragen wird. Hierin liegt die Aufgabe und die Notwendigkeit des Petrinetzes. Das Anlegen der Ein-Bit-Speicher ist nichts anderes als das Einrichten eines Platzes im Petrinetz.

Es ist nun noch zu klären, warum das Programm überhaupt einen Steueroperator `sop` enthält. Er ist erforderlich, wenn das Programm auf einem Einprozessorrechner laufen soll. Dann können die bereiten Instanzen sich nicht ohne Weiteres starten, sondern müssen evtl. auf den Prozessor warten. Die Zuweisung des Prozessors an die bereiten Instanzen ist die einzige Aufgabe des Steueroperators. Das gesamte Programm stellt für einen Einprozessorrechner einen imperativen Algorithmus dar, für einen Mehrprozessorrechner stellt es ein Datenflussprogramm dar, und bei seinem Start wird nicht nur der Multiplizierer (`mul`), sondern auch die Alternativweiche (`awei`) gestartet. Die Folge ist, dass die beiden Äste der starren Masche, in die sich der Operandenfluss gleich zu Beginn gabelt, parallel ausgeführt werden.

An dieser Stelle ist eine ergänzende Zwischenbemerkung angebracht. In unserem Beispiel verfügen die Operatorklassen nur über eine einzige Methode mit dem Bezeichner `berechnen` (abgesehen vom `Constructor` und `Destructor`). Allgemein kann eine Objektklasse über mehrere Methoden verfügen. In diesem Falle muss einer Instanz zusammen mit dem Operanden auch die anzuwendende Methode mitgeteilt werden, m.a.W. die Mitteilungen zwischen Instanzen sind Aktionsaufträge. Wir hatten sie Direktiven genannt. In der Literatur ist es üblich, Mitteilungen zwischen Instanzen als *Botschaften* oder *Messages* zu bezeichnen.

Damit schließen wir die Begründung der Notwendigkeit der Einbeziehung des Operatorennetzes (Bild 20.9) und des Petrinetzes (Bild 20.10) und deren Realisierung durch das Programm von Bild 20.8 ab. Aus dem Gesagten wird der große Vorbereitungsaufwand (bis Zeile 450) nur zum Teil erklärt. Der schwerwiegendere Grund liegt darin, dass das Programm als *Programmierwerkzeug* entworfen worden ist, als Hilfsmittel zur Realisierung anderer Operatorennetze, d.h. zur Programmierung neuer "Kompositobjekte" (neuer Operatorennetze).

Wenn beispielsweise ein Netz aus den Operatoren von Bild 20.9 komponiert werden soll, jedoch mit einer anderen Struktur, genügt es, das Operatorennetz zu zeichnen, die Operatoren und Plätze zu nummerieren und in der Kopplungstabelle (Zeilen 501 bis 507) die alten Platznummern durch neue zu ersetzen. Wenn das Netz mehrere Exemplare der bereits vorhandenen Operatortypen (Objektklassen) enthält, sind die entsprechenden Typen mehrmals zu instanzieren, d.h. die Kopplungstabelle ist entsprechend zu erweitern. Wenn das Netz neue Operatortypen enthält, sind diese als Objekte zu deklarieren und ihre Methoden zu programmieren und in den Prozedurteil aufzunehmen. De facto stellt das Programm ein Software-Entwicklungssystem dar, wenn auch ein sehr anspruchsloses mit wenig Komfort. Das Programm erhält diese Qualität durch die Verwendung des **Zeigerkonzepts**, dessen Besprechung nun nachgeholt werden soll.

Ein Systemprogrammierer, der ein Softwaresystem für die Entwicklung von Software, beispielsweise für das Implementieren von Operatorennetzen, entwerfen und realisieren will, steht vor folgendem Problem. Er muss beim Programmentwurf gedanklich mit Objekten hantieren, die noch nicht existieren, die erst dann existent werden, wenn das System zum Einsatz kommt, wenn beispielsweise ein Nutzer das

System beauftragt, ein Operatorennetz zu implementieren und ihm die Bausteinoperatoren und die Struktur vorgibt, z.B. in Form einer Kopplungstabelle. Die Lösung des Problems liegt in der Verwendung von *Zeigern*.

Ein Zeiger ist ein *Bezeichner* für ein variables Zielobjekt, auf das der Zeiger zeigt. Er stellt also eine Variable dar, dessen Wert ein Zielobjekt ist. Damit ist ein neuer Datentyp definiert, der *Datentyp* ***Zeiger*** *oder* ***Pointer***. Das Zielobjekt eines Zeigers ist ebenfalls eine Variable. *Eine Variable, auf die ein Zeiger zeigt, heißt* ***dynamische Variable***. Es wurde bereits gesagt, dass in dem Programm `opnetz` Bezeichner von Zeigern mit einem großen P beginnen. Der Zeigermechanismus soll anhand der Zeilen

```
134     Psync = ^Tsync;
505     sop.opord(New(Psync, opkopp(0,0,0,0, 5,6,7,0)), 5);
202     opliste[opnummer] := Piop;
203     Piop^.opnummer := opnummer;
215     opliste[t]^.berechnen;
```

erläutert werden.

In Zeile 134 wird eine Zeigervariable[6] mit dem Bezeichner `Psync` deklariert. Dass es sich um eine Zeigervariable handelt, erkennt der Compiler an dem Dach, das dem Zielobjekt `Tsync` vorangestellt ist (das große P in `Psync` ist Programmierstil). Der Zeiger `Psync` zeigt also auf die Klasse der Instanzen `sync`, m.a.W. auf den Objekttyp `Tsync`.

Die Prozedur `New` in Zeile 505 instanziert eine Instanz der Klasse `Tsync`, auf die `Psync` zeigt. Die Instanz wird dynamisch gebunden, d.h. ihr wird zur Laufzeit des Hauptprogramms ein Speicherbereich und der Zeigervariablen `Psync` wird als Wert die Anfangsadresse dieses Speicherbereiches zugewiesen. Der Speicherbereich ist der *private Speicher* des Operators `sync1` in Bild 20.10, der die Nummer 5 trägt. Der Bereich enthält u.a. einen Platz für den Wert der Variablen `opnummer`, die in Zeile 109 deklariert ist und eine Variable der Prozedur `opord` darstellt (vgl. Zeile 143). Da `opnummer` eine dynamische Variable ist, muss bei Ausführung der Zeile 505 sowohl die (dynamische) Bindung als auch die Wertzuweisung erfolgen. Wir wollen uns überlegen, wie der Computer diese Aufgabe erledigt, wir wollen also die interne Semantik der Prozedur `sop.opord` verstehen, die aus den Zeilen 202 und 203 besteht.

Zeile 202 bewirkt (wenn sie im Rahmen der Zeile 505 abgearbeitet und `iop` durch `sync1` substituiert wird), dass in den Platz mit dem Index 5 des Array `opliste` der Wert der Zeigers `Psync`, das ist die Anfangsadresse des zugewiesenen Speicherbereiches, eingetragen wird, m.a.W. dass die Operatorinstanz in die Operatorenliste "eingeordnet" wird. Zeile 203 bewirkt, dass die Operatornummer 5 in den dafür vorgesehenen Speicherplatz im Speicherbereich des Operators `sync1` (siehe Bild 20.10) eingetragen wird. Die Zahl 5 erscheint in Zeile 505 an letzter Stelle. Sie

6 In der Nomenklatur von BorlandPascal müsste es "Zeiger-Typ" heißen.

stellt den Wert der zweiten Variablen der Prozedur `sop.opord` dar. Es muss also auf den privaten Speicher zugegriffen werden, auf den der Zeiger `Psync` zeigt. Das wird dem Computer durch das dem Zeiger nachgestellte Dach in Zeile 203 mitgeteilt.

Die Zeichenkombination `^.` ist charakteristisch für ein BorlandPascal-Programm. Sie steht zwischen einem Zeiger und einem anderen Variablenbezeichner und weist den Computer an, auf den Speicherbereich, auf den der Zeiger zeigt, zuzugreifen und dort den Speicherplatz (bzw. den Unterbereich) zu suchen, dessen Bezeichner nach dem Punkt angegeben ist.

Auch in Zeile 215 tritt die Kombination `^.` auf. Diesmal handelt es sich nicht um einen Schreibzugriff wie in Zeile 203, sondern um einen Lesezugriff. Man beachte, dass in die Zeile 215 bei ihrer Ausführung für `t = 5` der Variablenbezeichner `opliste[t]` durch den Wert dieser Variablen, also durch `Psync`, und schließlich durch `Tsync` (gemäß Zeile 134) ersetzt wird. Die Zeile 215 weist den Computer an, auf den Speicherbereich der Prozedur `Tsync.berechnen` zuzugreifen und die Prozedur abzuarbeiten.

Die Syntax von BorlandPascal schreibt vor, dass im Deklarationsteil nach dem Namen einer Prozedur die Anweisung `Virtual` folgen muss (siehe z.B. die Zeilen 111 und 136), falls die Prozedur *dynamisch*, also nicht während der Compilierung in das Hauptprogramm eingebunden werden soll, sondern erst dann, wenn sie vom Hauptprogramm zu dessen Laufzeit aufgerufen wird .

Abschließend sei folgender Sachverhalt noch einmal besonders hervorgehoben. *Die Anwendung des* ***Zeigerkonzepts*** *im Rahmen des objektorientierten Programmierens ermöglicht das Hantieren mit* ***privaten, dynamischen Speichern*** *. Ein solcher Speicher ist "Privateigentum" einer Instanz und dient ihr zur Abspeicherung ihrer privaten Softwarebetriebsmittel*. Der private Charakter der Speicher sichert die Abkapselung der Instanzen voneinander; der dynamische Charakter ermöglicht das Anlegen von Instanzen zur Laufzeit eines objektorientierten Programms und damit die Verwendung des Programms als Entwurfswerkzeug.

20.4 Netzprogrammierung und Software-Lebenszyklus

Die Programmbeispiele des Kapitels 20 und die Überlegungen zur Evolution der Programmiersprachen in Kapitel 18 sollen mit einem Gedanken abgeschlossen werden, der in ähnlicher Form wiederholt geäußert und in verschiedenen Programmsystemen seinen Niederschlag gefunden hat, wenn auch nicht auf der Grundlage der Methode der uniformen Systembeschreibung (USB). Die in Kap.20.3 angewendete Programmiermethode, die sich an der USB-Methode orientiert, kann verallgemeinert werden. Wie wir wissen, lässt sich jede Funktion nach der USB-Methode beschreiben [8.26] und jeder kausaldiskrete Prozess (jedes kausaldiskrete System) nach der USB-Methode modellieren [8.7]. Das legt den Gedanken nahe, eine dieser Methode

angepasste spezifische Programmiermethode zu entwickeln. Wir nennen sie *operatorennetz-orientierte Methode* oder kurz **Netzmethode**.

Um die Netzmethode nutzerfreundlich zu gestalten, ist es angebracht, ein Programmiersystem mit graphischer Oberfläche zu entwickeln und dem Nutzer eine zweidimensionale Sprache zur Darstellung von erweiterten Operandenflussplänen und gesteuerten Petrinetzen zur Verfügung zu stellen. Graphische Entwicklungsumgebungen existieren bereits. Beispielsweise stellen die Versionen von Borland-Delphi (Weiterentwicklungen von BorlandPascal) eine solche Entwicklungsumgebung zur Verfügung. Doch unterstützt sie nicht die Implementierung USB-orientierter Netze.

Es bestehen viele Parallelen zwischen der Netzprogrammierung und anderen in der Literatur[7] beschriebenen objektorientierten Software-Entwicklungssystemen. Die angedeutete Netzmethode stellt also kein grundsätzlich neues Programmierparadigma dar, sondern ist eine spezielle Ausgestaltung des objektorientierten Paradigmas, in welchem wir eine Kombination des imperativen und des funktionalen Paradigmas erkannt hatten. Wenn die Methode auf größere Systeme angewendet werden soll, kann ein schrittweises oder schichtenweises Dekomponieren zweckmäßig sein.

Die Besonderheit der Netzprogrammierung liegt, wie der Name andeutet, in dem netzorientierten, *parallelen* Ansatz. Ein Netzprogramm ist ein *nebenläufiges* Programm, vorausgesetzt, das Netz enthält starre Maschen. Wenn ein solches Programm auf einem Mehrprozessorrechner mit einer ausreichenden Anzahl von Prozessoren und einem ausreichend mächtigen Bussystem läuft, kann der theoretisch mögliche Parallelitätsgrad (Anzahl parallel laufender Bausteinprozesse) erreicht werden. Bei hoher Parallelisierung, wenn also viele Operanden gleichzeitig das Netz durchlaufen, kann in Vereinungen die richtige Zusammenfassung der eingehenden Operanden zu Paaren (Tupeln) problematisch werden; wir sprechen vom *Vereinungsproblem*. Wer sich in das Problem vertieft, wird erkennen, dass im Falle komplizierter Netze auch das Vereinungsproblem kompliziert werden kann, dass es aber durch Erweiterungen des Netzes um zusätzliche Operandenplätze und Weichen lösbar ist. Wir belassen es bei diesen wenigen Bemerkungen.

Es sei noch einmal betont, dass das Programm von Kap.20.3 zwar als ON-Entwicklungssystem dienen kann, dass aber auf dem Markt angebotene Software-Entwicklungssysteme erheblich leistungsfähiger sind. Sie unterstützen viele oder alle Schritte, die während der Entwicklung und Nutzung eines Programms ausgeführt werden müssen, m.a.W. sie unterstützen alle *Phasen* des sogenannten **Lebenszyklus** eines Softwareproduktes[8]. Üblicherweise wird der Lebenszyklus untergliedert in

- Spezifikation: genaue Angabe, was das Programm leisten soll.

7 Siehe z.B. [Horn 93]

8 Die folgenden kurzen Darlegungen lehnen sich an [Horn 93] an.

- Grobentwurf: Entwurf der Softwarearchitektur (der Operatorenhierarchie) aus der Sicht des Anwenders.
- Feinentwurf: Festlegung des inneren Aufbaus der Architekturkomponenten (der Bausteinoperatoren).
- Codierung: Erzeugung lauffähiger Programme.
- Integration, Installation und Testung: Zusammenfügen der Programme zu einem System, Installieren auf einem Rechner und Austesten.
- Wartung: Fehlerbeseitigung, Anpassung an die konkrete Umgebung, in der das System genutzt wird; eventuell Erweiterungen.

Ein Software-Entwicklungssystem kann als Simulationssystem aufgefasst werden, das die Tätigkeiten von Ingenieuren simuliert, die Softwareprodukte entwerfen, programmieren und warten. Da nicht alle Tätigkeiten algorithmierbar sind, müssen bei der Herstellung und Wartung Mensch und Maschine (Ingenieur und Computer) miteinander kooperieren. Voraussetzung der Simulation ist eine genaue Beschreibung der Arbeitsschritte der einzelnen Phasen. Diese Beschreibung wird **Vorgehensmodell** genannt.

Ein Software-Entwicklungssystem stellt seinerseits ein Softwareprodukt dar. Für seinen Entwurf können - genauso wie für die Systeme, die mit seiner Hilfe produziert werden, - die verschiedenen Paradigmen zur Anwendug kommen. Moderne Entwicklungssysteme sind vorzugsweise nach dem objektorientierten Paradigma entworfen. Einen informativen Überblick über objektorientierte Vorgehensmodelle gibt der Artikel [Noack 99]. In ihm werden 7 verschiedene in praktischer Nutzung befindliche Vorgehensmodelle miteinander verglichen.

21 Komplexität

Zusammenfassung der Kapitel 21 und 22

Ein *Komplex* ist ein viele Bestandteile umfassendes Objekt der Realität oder des Denkens mit *globalen* Eigenschaften, die sich nicht unmittelbar oder auch gar nicht aus den *lokalen* Eigenschaften der Komponenten und aus der Struktur des Objekts ableiten lassen. Die Eigenschaft der Vielgliedrigkeit (evtl. auch Vielschichtigkeit) von Komplexen heißt *strukturelle Komplexität*.

Angesichts der Unschärfe des Komplexbegriffs liegt der Wunsch nahe, ihn zu objektivieren, nach Möglichkeit sogar zu quantifizieren und zu mathematisieren, ihn in einen Kalkül einzubinden. Viele Bemühungen gehen in diese Richtung. Dabei haben sich zwei Theorien herausgebildet, zum einen die *Theorie nichtlinearer dynamischer Systeme*, sie betrifft die *physische* Komplexität realer Objekte, und zum anderen die *Komplexitätstheorie*, sie betrifft die *logische* Komplexität von Problemen und Problemklassen bzw. von Algorithmen, die Problemklassen lösen.

Die Komplexitätstheorie definiert eine Hierarchie von Komplexitätsklassen und untersucht die Eigenschaften der Klassen und die Beziehungen zwischen ihnen. Zu einer *Komplexitätsklasse* gehören alle Probleme bzw. Algorithmen, deren Lösungs- bzw. Berechnungsaufwand nach ein und demselben Gesetz mit der Problemgröße zunimmt, genauer mit einem die Problemgröße charakterisierenden Parameter. Beispielsweise nimmt der Rechenaufwand für das Addieren zweier Dezimalzahlen linear mit der Stellenzahl zu. Darum sagt man, dass die Addition ein Problem linearer Komplexität ist.

Die Komplexität von Problemen bzw. Algorithmen ist ein klassifikatorisches Aufwandsmerkmal mit den Werten logarithmisch, linear, polynomial (sprich "potenziell", d.h. der Aufwand wächst mit einer Potenz der Problemgröße), exponentiell und weiteren. Es gilt die Faustregel, dass Probleme polynomialer Komplexität i.Allg. noch praktisch lösbar sind, Probleme exponentieller Komplexität hingegen nur bei sehr geringer Problemgröße. Beispielsweise ist die Suche nach dem besten Zug in einer Schachpartie durch Vorausspielen ein Problem exponentieller Komplexität, zumindest im Eröffnungs- und Mittelspiel. Aus diesem Grunde kann der Mensch nur wenige Züge in allen Variationen vorausspielen. Er stützt sich auf höhere Intelligenzleistungen wie Assoziation, Intuition, Lernen durch Üben, durch Übernehmen fremder und durch Sammeln eigener Erfahrungen. Da sich derartige Leistungen nur sehr eingeschränkt simulieren lassen, ist der Schachcomputer im Wesentlichen auf das Vorausspielen angewiesen und muss gegen den Menschen seine hohe Rechengeschwindigkeit ins Feld führen.

Auf dem Gebiet der physischen Komplexität realer Systeme sind die Arbeiten in vollem Gange. Zur strukturellen Komplexität von Systemen hoher Komponierungsstufe kommt die *nichtlineare Komplexität* des Verhaltens, die von Irregularitäten

herrühren. Eine solche nichtlinerare Irregularität ist z.B. das “Umkippen” in einen anderen Zustand oder in eine andere Verhaltensweise oder das Springen von Merkmalswerten. Diese Art von Komplexität ist Gegenstand der “Theorie nichtlinearer Systeme”. Sie gewinnt für das Verständnis der Entstehung und Verarbeitung von Information auf *subsymbolischem* Niveau zunehmend an Bedeutung.

Das Resümee der bisherigen Bemühungen, den Computer mit natürlicher Intelligenz auszustatten, lautet: Die Leistungen menschlicher Intelligenz wie Deduktion, Assoziation, Intuition (einschließlich Phantasie und Kreativität), Wiedererkennen, Erkenntnisgewinnung und Lernen lassen sich in konkreten Fällen kalkülisieren, algorithmieren und simulieren, in der Regel allerdings nur in engen Grenzen. Denn die Komplexität des menschlichen Denkens ist selten durchschaubar und noch seltener simulierbar. Der Mensch weiß mehr als der Computer und er kann sein Wissen effektiver nutzen als der Computer.

21.1 Zum Begriff der Komplexität

Das letzte Kapitel des Buches vor dem Resümee ist einem Begriff gewidmet, der in den letzten Jahren stark in Umlauf gekommen ist, dem Begriff der *Komplexität*. Fast kann man sagen, dass das Wort “Komplexität” zu einem Modewort geworden ist. Ein Grund dafür liegt in der Entwicklung der Informatik, in der zunehmenden Komplexität von Computern und Computerprogrammen sowie von Objekten und Problemen, die mit Hilfe von Computern simuliert bzw. gelöst werden können. Umgangssprachlich wird der Komplexitätsbegriff in einem nicht scharf definierten, insbesondere nicht in einem quantitativ definierten Sinne verwendet. Dennoch besteht ein weitgehender Konsens hinsichtlich der Anwendbarkeit des Begriffs und auch hinsichtlich des Grades der Komplexität beim Vergleich verschiedener komplexer Objekte. Er kann sich sowohl auf die Struktur als auch auf das Verhalten eines Objekts beziehen.

Jeder wird zustimmen, dass eine Ameise etwas Komplexeres ist als ein Sandkorn, sowohl hinsichtlich der Struktur als auch hinsichtlich des Verhaltens. Die meisten Menschen werden den Begriff der Energie intuitiv für komplexer halten als den Begriff der Geschwindigkeit und den Begriff des Bewusstseins für komplexer als den des Schmerzes. Auch wird jeder zustimmen, dass das Gehirn sowohl strukturell als auch verhaltensmäßig etwas Komplexeres ist als ein Computer und dass Kopfrechnen ein komplexerer Vorgang ist als maschinelles Rechnen.

Das Adjektiv *komplex* und das Substantiv *Komplex* leiten sich vom Lateinischen ab. Das Wort *complexus* bedeutet “verflochten” und das Wort *complexio* unter anderem “zusammenfassende Darstellung”. Das Substantiv Komplex enthält beide Bedeutungsanteile, den der Verflochtenheit (Vernetztheit) vieler Teile, und den der Einheit, des zu *einem* Objekt “Zusammengefassten”. Hinzu kommt die Vorstellung,

dass ein komplexes Objekt infolge seiner Vielgliedrigkeit schwer oder gar nicht zu durchschauen ist, dass es “kompliziert” ist. In diesem Sinne definieren wir:

*Ein **Komplex** oder **komplexes System** ist ein viele Bestandteile umfassendes Objekt der Realität oder des Denkens mit **globalen** Eigenschaften, auch **Makroeigenschaften** genannt, die sich nicht unmittelbar oder auch gar nicht aus den **lokalen** Eigenschaften der Komponenten, auch **Mikroeigenschaften** genannt, und der Struktur des Objekts ableiten lassen. Die Eigenschaft der Vielgliedrigkeit, eventuell auch Vielschichtigkeit eines Komplexes heißt **strukturelle Komplexität**.* Weiter unten werden zwei andere Arten von Komplexität eingeführt, die *nichtlineare Komplexität* und die *Berechnungskomplexität*.

Der Leser hätte die vorangehenden Darlegungen und Definitionen sicher auch dann akzeptiert, wenn anstelle der Wörter “*komplex*” und “*Komplexität*” die Wörter “*kompliziert*” und “*Kompliziertheit*” gestanden hätten. Der Unterschied zwischen komplex und kompliziert liegt im Kontext, in unterschiedlichen Sichten. Ein und dasselbe Objekt kann sich dem “Blick von außen”, der das Ganze sieht, als “komplex” darstellen, dem “Blick von innen”, der nicht das Ganze sieht, dagegen als “kompliziert”. Beispielsweise stellt sich die Denksportaufgabe 1 aus Kap.16.1 [16.2] (Verwandtschaftsverhältnis) demjenigen als kompliziert dar, der in der Menge der Beziehungen nach einem Lösungsweg sucht, aber nicht (oder noch nicht) das Ganze im Auge hat. Das “Ganze” ist in den Bildern 16.2 und 16.3 graphisch dargestellt. Die Vielgliedrigkeit der Graphen zeigt die *Komplexität* des Problems; sie veranschaulicht den *Grad der logischen strukturellen Komplexität*.

In ähnlicher Weise veranschaulicht die graphische Darstellung einer Operatorenhierarchie die *physische strukturelle* Komplexität eines nach der USB-Methode komponierten Systems, z.B. eines informationellen Systems. Das “Ganze”, von außen als Einheit betrachtet, bezeichnen wir als *komplexes* Objekt oder als *Komplex*. Ein Computer als Ganzes betrachtet ist ein komplexes Objekt. Seine Verdrahtung ist, im einzelnen (dekomponiert) betrachtet, kompliziert.

In dem so definierten Sinne wird der Komplexbegriff in den verschiedensten Wissensgebieten verwendet, besonders sinnfällig als “verflochtenes Objekt” in der Chemie. Dort wird er für eine besondere Art von Verbindungen benutzt, die sogenannten *Komplexverbindungen*. Das sind Molekülverbindungen höherer Ordnung, die durch Anlagerung vieler einfacher Moleküle, der *Liganden* (z.B. Wassermoleküle) um einen Zentralkörper herum entstehen. Der Komplex hat Eigenschaften, die sich nicht durch “Aufsummieren” der Eigenschaften der Liganden und des Zentralkörpers ergeben. Andrerseits können die Eigenschaften der Komponenten verloren gehen. Beispielsweise zeigt ein Ni-Komplex nach außen hin keine Eigenschaften des Nickels, solange der Komplex nicht zerstört wird.

In der Psychologie wird eine Gesamtheit vieler Empfindungen und Verhaltensweisen eines Menschen, die - eben in ihrer Gesamtheit - ein typisches Charakteristikum der betreffende Person bilden, als *psychischer Komplex* bezeichnet. Die detaillierte Erklärung der äußeren Erscheinungsform, der Symptome psychischer

Komplexe, ist ein weites Feld psychologischer Forschung. Analoges gilt für die Symptome vieler Krankheiten.

Ähnlich anschaulich wie im Strukturgraphen einer Komplexverbindung tritt die *Verflochtenheit* jedes Objektes zutage, wenn seine Struktur graphisch dargestellt wird. Der Komplexität des Objektes entspricht die Komplexität des Graphen. Wenn die Kanten eines komplexen Graphen physische Verbindungen darstellen, wie z.B. im Falle einer elektronischen Schaltung oder irgendeines anderen realen Kompositoperators, sprechen wir von **physischer Komplexität**. Wenn sie begriffliche Beziehungen darstellen, z.B. logische Beziehungen, Ähnlichkeitsbeziehungen, Verwandtschaftsbeziehungen, sprechen wir von **logischer Komplexität**. Die Hardware eines Computers ist ein physisch, die Software ein logisch komplexes System. Die Denksportaufgabe in Kap.16.1, in der eine *kompliziert* beschriebene Verwandtschaftsbeziehung zwischen zwei Menschen einfacher ausgedrückt werden sollte, ist ein Beispiel für ein *logisch* komplexes Problem. Es sei an Kap.18.1 [18.2] erinnert, wo zweidimensionale Sprachen zur Beschreibung komplexer Objekte verwendet wurden. Dabei war zwischen räumlicher, logischer und kausaler Komplexität unterschieden worden. Der soeben eingeführte Begriff der *physischen* Komplexität umfasst *räumliche* und *kausale* Komplexität.

Wenn ein Objekt so komplex ist, dass es keinen (oder scheinbar keinen) Weg zur "Erkenntnis" gibt, d.h. zum Erkennen der inneren Zusammenhänge, durch welche die vielen Bestandteile zu einer Einheit, zu einem Komplex zusammengeschlossen werden, dann ist es unmöglich, die Eigenschaften des Komplexes zu verstehen, geschweige denn sie abzuleiten. Die Komplexität versperrt den erforderlichen Durchblick. Sie *verbirgt* die Details des Komplexes. In diesem Falle sagen wir, dass die Komplexität **undurchschaubar** ist.

Wenn die inneren Zusammenhänge eines komplexen Objekts erkannt sind oder "im Prinzip", d.h. ohne Berücksichtigung des für eine detaillierte Beschreibung erforderlichen Aufwandes, erkannt werden können, sprechen wir von **durchschaubarer Komplexität**. Durchschaubare Komplexität heißt **beherrschbar**, wenn die inneren Zusammenhänge mit realisierbarem Aufwand erkannt und beschrieben und die globalen Eigenschaften abgeleitet oder simuliert werden können. Ein Modell eines Komplexes, das die globalen Eigenschaften beschreibt, nennen wir global oder *Makromodell*. Ein Modell, das die inneren Zusammenhänge beschreibt, nennen wir *Mikromodell*.

Das Charakteristikum von Komplexen, nämlich das Auftreten globaler Eigenschaften eines Objekts, das aus vielen Komponenten besteht, war uns in Kap.19.2.3 [19.1] im Zusammenhang mit der mathematischen Behandlung von Vielkörperproblemen begegnet. Als anschauliches Beispiel diente eine Schafherde. Sie stellt ein **homogenes komplexes System** dar, d.h. ein System, das aus vielen einheitlichen Bausteinen besteht. Die globalen Eigenschaften derartiger Komplexe werden *kollektive Eigenschaften* genannt. Bei den Vielkörperproblemen der Physik handelt es sich i.d.R. um weitgehend homogene komplexe Systeme.

Für Erscheinungen, die schwer oder gar nicht zu erklären sind, wird in den Naturwissenschaften, aber auch umgangssprachlich häufig das Wort *Phänomen* benutzt. Das Wort schließt i.Allg. die Vorstellung des *Komplexen* ein. Es korrespondiert mit dem Wort *Emergenz*. Beide Wörter artikulieren das "Hervortreten" einer beobachtbaren *Erscheinung* aus einem zugrundeliegenden komplizierten, nicht unmittelbar beobachtbaren Sachverhalt. Die kollektiven Bewegungen einer Schafherde oder eines Mückenschwarms sind derartige Phänomene. Sie "emergieren" aus den komplizierten Wechselwirkungen der Elemente des Komplexes (der Herde bzw. des Schwarms).

Bei den genannten Beispielen handelt es sich um natürliche Komplexe, beim Computer oder beim Internet um künstliche Komplexe. **Natürliche Komplexe** und ihre Komplexität sind das Produkt der (natürlichen) Evolution, **künstliche Komplexe** und ihre Komplexität sind das Produkt menschlicher Tätigkeit ("künstlicher Evolution"). Abgesehen vom Universum, die Menschheit eingeschlossen, ist das menschliche Gehirn das komplexeste uns bekannte Produkt der natürlichen Evolution.

Natürliche Komplexität ist in vielen Fällen dank der Wissenschaft durchschaubar geworden, beispielsweise im Bereich der Struktur der unbelebten Materie. Es erscheint aber zweifelhaft, ob der Mensch imstande ist, die Komplexität von Objekten zu durchschauen, die lebendig sind oder gar Bewusstsein besitzen. Die mögliche Undurchschaubarkeit komplexer Objekte bedingt eine gewisse Verschwommenheit, eine Unschärfe des Komplexitätsbegriffs in seiner umgangssprachlichen Bedeutung. Das trifft nicht nur für natürliche, sondern auch für künstliche komplexe Objekte zu. Der Mensch kann Dinge erfinden und erschaffen, deren Komplexität er nicht unbedingt durchschaut. Zur Illustration dieses Umstandes wird gerne die Dampfmaschine herangezogen, deren physikalisches Funktionsprinzip zur Zeit ihrer Erfindung nur sehr oberflächlich durchschaut worden war.

Angesichts der Unschärfe und Vagheit des Komplexitätsbegriffs liegt der Wunsch nahe, ihn zu *objektivieren*, nach Möglichkeit sogar zu *quantifizieren* und zu *mathematisieren*, ihn in einen Kalkül einzubinden. Hier sind in erster Linie Mathematiker und theoretische Physiker gefordert. Ihre Bemühungen haben zur Herausbildung zweier spezieller Theorien und zweier spezieller Komplexitätsbegriffe geführt. Auf dem Gebiet der Algorithmen, wo es um *logische* Komplexität geht, ist die sog. **Komplexitätstheorie** und der Begriff der *Berechnungskomplexität* entstanden. Auf physikalischem Gebiet ist eine **Theorie nichtlinearer dynamischer Systeme** (auch *Theorie nichtlinearer Dynamik* genannt) und ein Komplexitätsbegriff entstanden, den wir als *nichtlineare Komplexität* bezeichnen werden. Diese beiden speziellen Komplexitätsbegriffe haben nur noch sehr bedingt mit "Verflochtenheit" zu tun, sodass der Rückgriff auf das lateinische `complexus` eigentlich seinen Sinn verliert. In Kap.21.2 werden die Umrisse der Komplexitätstheorie dargestellt und in Kap.21.3 wird kurz auf nichtlineare Dynamik eingegangen.

21.2* Berechnungskomplexität

In Kap.8.3 [8.15] war der Begriff der Berechenbarkeit eingeführt worden, und in Kap.8.4 hatten wir nach Kriterien für die Berechenbarkeit von Funktionen gesucht. Am Ende von Kap.8.3 war jedoch bereits angemerkt worden, dass der dort definierte Begriff der Berechenbarkeit nutzlos ist, wenn die praktische Berechenbarkeit interessiert, wenn also nach dem *Aufwand* für eine Berechnung gefragt wird, an dem die Durchführung einer Berechnung möglicherweise scheitern kann. Das Aufwandsproblem ist Gegenstand der sogenannten **Komplexitätstheorie**, einem relativ jungen Zweig der Algorithmentheorie. In modernen Darstellungen werden die Probleme der Berechenbarkeit und der Berechnungskomplexität zuweilen in einem einheitlichen Theoriengebäude dargestellt.[1]

Wenn man das Wort "Komplexitätstheorie" zum ersten Mal hört, könnte man annehmen, es handele sich um eine "Theorie der Komplexität". Das ist jedoch nicht der Fall. Vielmehr handelt es sich um eine "Theorie des Berechnungsaufwandes". Da der Berechnungsaufwand (Lösungsaufwand) von der logischen Komplexität des jeweiligen Problems abhängt, spricht man von **Berechnungskomplexität**. Das Wort "Komplexitätstheorie" ist in ähnlicher Weise irreführend wie das Wort "Informationstheorie", das *nicht* als "Theorie der Information" zu verstehen ist [5.21]. Ursprung der Irritation ist dort der Informationsbegriff, der in der Informationstheorie nicht im umgangssprachlichen Sinne verwendet wird. Das Gleiche gilt für den Komplexitätsbegriff der Komplexitätstheorie. Er ist nicht identisch mit dem umgangssprachlichen Komplexitätsbegriff, dessen Präzisierung in Kap.21.1 gefordert wurde. Mit der Behandlung der Komplexitätstheorie kehren wir auf den Boden exakter Begriffe zurück, den wir nach Kapitel 16 verlassen hatten.

Wir werden zunächst die Worte *Berechnungsaufwand* und *Berechnungskomplexität* so verwenden, als seien sie Synonyme. Später wird deutlich werden, worin sich die beiden Begriffe unterscheiden. Im Augenblick begnügen wir uns mit der Feststellung, dass der Berechnungsaufwand gewissermaßen die sichtbare oder spürbare Wirkung der Komplexität ist, die sich in dem zu lösenden Problem verbirgt.

Der Begriff der Berechnungskomplexität ist uns aus Kap.17.3 bereits bekannt. Dort waren wir zu der Einsicht gelangt, dass es unmöglich ist, den besten Schachzug in einer bestimmten Spielsituation (abgesehen von sehr einfachen Endspielsituationen) durch vollständiges Durchmustern, d.h. durch vollständiges Vorausspielen der Restpartie zu finden, weil "das Problem zu komplex" ist, wobei wir uns auf ein intuitives Verständnis des Wortes "komplex" beim Leser verlassen hatten. Die Komplexität wird durch die Spielregeln "produziert"; sie ist eine "globale Eigenschaft" des Schachspiels. Sie lässt sich durch den Suchgraphen veranschaulichen.

1 Beispielsweise in [Börger 92].

Von jeder Stellung (jedem Knoten des Graphen) gehen i.Allg. viele Wege aus. Verschiedene Wege können sich wieder vereinigen (Bildung von Maschen).

Während des Spielens tritt die Komplexität im Denkaufwand (Suchaufwand) zutage, eben in der “Berechnungskomplexität”. In Kap.17.3 [17.4] hatten wir die Anzahl der Fortsetzungsmöglichkeiten in den nächsten n Zügen näherungsweise zu 20^n angesetzt. Das bedeutet, dass der Suchaufwand (und damit der Berechnungsaufwand) mit der Suchtiefe n wie 20^n, also exponentiell mit n wächst. In diesem Sinne spricht man von *exponentieller Komplexität*. Die Größe n charakterisiert den *Problemumfang*. Einen derartigen Parameter, dessen Wert den Umfang und damit den Lösungsaufwand eines Problems charakterisiert, nennen wir **Problemgröße**.

Ein Problem, dessen Lösungsaufwand exponentiell mit der Problemgröße, zunimmt, wird als ***Problem mit exponentieller Komplexität*** *bezeichnet.* Derartige Probleme werden schon für relativ kleine Parameterwerte unlösbar, d.h. praktisch nicht mehr berechenbar. Im Schachbeispiel ist der die Problemgröße charakterisierende Parameter die Tiefe n des gedanklichen Vorausspielens. Die Komplexität des Problems ist zwar *durchschaubar*, aber nicht *beherrschbar*. 1

Ein anderes Beispiel durchschaubarer aber nichtbeherrschbarer Komplexität ist die Schaltung eines nichtsteuerbaren Rechners, der für jede Funktion eine Kombinationsschaltung enthält. Auf dieses Problem waren wir in Kap.9.2.1 [9.6] gestoßen. Aus Formel (9.3) folgt, dass die Anzahl der elementaren booleschen Operatoren, die für die Realisierung einer zweistelligen Funktion (z.B. der Addition) erforderlich sind, mit der Länge der zu verarbeitenden Bitketten (z.B. der Summanden) *exponentiell* wächst. Aus mathematischer Sicht handelt es sich um *logische* Komplexität. Bei der hardwaremäßigen Realisierung wird die logische zu *physischer* Komplexität, zu einem “Gewirr von Drähten”, sie wird zu *Schaltungskomplexität*. Die Unbeherrschbarkeit der physischen Komplexität hatte uns nach einem Ausweg suchen lassen. Wir fanden ihn in der dritten Grundidee des elektronischen Rechnens, in der *Steuerbarkeit*. Auf diesem Wege gelangten wir zum Prozessor, zum Prozessorrechner und zur Berechnung gemäß Vorschrift (Algorithmus). Die physische Komplexität der Schaltung wird zur logischen Komplexität des Algorithmus.

Der Aufwand für die Ausführung einer Addition nach einem Algorithmus wächst nicht exponentiell, sondern nur *linear* mit der Problemgröße, d.h. mit der Stellenzahl der Summanden. Die Komplexität ist beherrschbar geworden. Allgemein spricht man von **linearer Komplexität**, wenn der Aufwand linear mit der Problemgröße wächst. Damit ist der Unterschied der Begriffe “Berechnungsaufwand” und “Berechnungskomplexität” klar benannt. Aus dem Berechnungsaufwand ist ein *klassifikatorisches Merkmal* hervorgegangen mit den Werten *linear* und *exponentiell*. Probleme linearer bzw. exponentieller Komplexität werden kurz als *lineare* bzw. *exponentielle Probleme* bezeichnet.

Die Unterscheidung zwischen linearer und exponentieller Komplexität legt den Gedanken nahe, weitere *Komplexitätsklassen* einzuführen, indem man die Abhängigkeit des Aufwandes von der (sinnvoll festzulegenden) Problemgröße als Charak-

teristikum der Komplexität auffasst. In diesem Sinne liegt z.B. **logarithmische**, **quadratische** oder **kubische** Komplexität vor, wenn der Aufwand mit dem Logarithmus, mit der zweiten bzw. mit der dritten Potenz der Problemgröße wächst. Beispielsweise ist die Matrizenmultiplikation ein Problem kubischer Komplexität, zumindest wenn der Lösungsalgorithmus so arbeitet, wie man normalerweise selber "per Hand" verfährt. Angenommen es sollen zwei quadratische Matrizen mit je n Zeilen und Spalten miteinander multipliziert werden. Zur Berechnung eines Elements der Ergebnismatrix müssen n Multiplikationen und n-1 Additionen ausgeführt werden, also 2n-1 arithmetische Operationen. Insgesamt müssen n^2 Elemente berechnet, also $n^2(2n-1)$ Operationen ausgeführt werden. Der Berechnungsaufwand wächst also grob gesagt mit n^3. Diese Angabe ist umso genauer, je größer n ist.

Man könnte einwenden, dass der Aufwandsunterschied zwischen Multiplikation und Addition nicht berücksichtigt worden ist. Das ist richtig, ändert aber nichts an der kubischen Abhängigkeit von n. Wollte man den Aufwand quantitativ berechnen (den Zeitaufwand z.B. in Prozessortakten oder den Speicheraufwand in Byte), müsste man außer diesem Unterschied noch viele andere Einflussfaktoren berücksichtigen wie die Architektur des Computers, die Ausdrucksmöglichkeiten der verwendeten Programmiersprache und das Programmierparadigma, das sie unterstützt, weiterhin die Eigenschaften des Compilers und den Programmierstil des Programmierers. Wenn der Berechnungsaufwand in Sekunden angegeben wird, hängt er außerdem von der Taktfrequenz des Prozessors ab.

Viele dieser Einflüsse verlieren mit zunehmender Problemgröße an Bedeutung. Hinsichtlich des klassifikatorischen Komplexitätsmerkmals werden sie überhaupt bedeutungslos. Allerdings ist dieses Merkmal *kein* eindeutiges Charakteristikum für die Komplexität von *Problemen*, denn nicht das Problem selber, sondern sein Lösungsalgorithmus wird durch den Wert des Merkmals charakterisiert. Beispielsweise ist nicht die Matrizenmultiplikation als solche von kubischer Komplexität, sondern der Algorithmus, den wir obiger Abschätzung zugrunde gelegt haben. Es gibt "schnellere" Algorithmen. So ist ein Algorithmus vorgeschlagen worden, dessen Aufwand mit $n^{2,81}$ wächst, und es kann durchaus noch schnellere Algorithmen geben. Die Berechnungskomplexität des schnellsten existierende Algorithmus kann nur als obere Grenze der im Prinzip erreichbaren Berechnungskomplexität des Problems gewertet werden, da in Zukunft möglicherweise noch schnellere entwickelt werden. Die Sachlage ist ähnlich derjenigen von Wahrsageprädikaten [8.21]. Man weiß nie, was die Zukunft noch bringt.

Die Theorie liefert in der Regel Aussagen der Art "Das Problem ist höchstens von kubischer Komplexität" oder "Das Problem ist mindestens von quadratischer Komplexität", d.h. es wird eine obere bzw. untere Grenze der Komplexität angegeben. Formal werden diese Angaben als $O(n^3)$ bzw. $U(n^2)$ notiert. Die Notationen sind folgendermaßen zu lesen: Der Berechnungsaufwand steigt asymptotisch (wenn n gegen Unendlich geht) nicht schneller als kubisch bzw. nicht langsamer als quadratisch mit n. Die Problemgröße wird als unbeschränkt angenommen. Aufgrund dieser

Angaben lässt sich abschätzen, ob ein gegebenes Problem praktisch lösbar ist oder nicht.

Die Komplexitätstheorie untersucht die Eigenschaften der verschiedenen Komplexitätsklassen und die Beziehungen zwischen ihnen. Es wird eine Hierarchie der Komplexitätsklassen aufgebaut. Eine wichtige Rolle spielen dabei die sog. *P-Probleme*, *NP-Probleme* und *NP-vollständigen Probleme*. Ein Problem heißt **polynomial** oder kurz P-Problem, wenn sein Berechnungsaufwand mit der Problemgröße n nicht schneller wächst als der Wert eines Polynoms in n vom r-ten Grade, für sehr große n also nicht schneller als n^r. Die linearen Probleme bilden demnach eine Unterklasse der polynomialen Probleme. **NP-Probleme** sind solche, die durch einen nichtdeterministischen Algorithmus [15.15] mit polynomialem Aufwand gelöst werden können. Als **NP-vollständig** werden alle NP-Probleme bezeichnet, die mit polynomialem Aufwand ineinander überführt werden können. Die Frage, ob die Klasse der NP-vollständigen mit der Klasse der polynomialen Probleme identisch ist, konnte bisher nicht beantwortet werden. Wir müssen es bei diesen wenigen Bemerkungen bewenden lassen und den interessierten Leser auf die Literatur verweisen[2].

Wir beenden das Kapitel mit einer Ergänzung zu den möglichen Bedeutungen des Wortes "Berechnungskomplexität". Wie bereits bemerkt, hat das Wort auch in seiner "natürlichen" Bedeutung, nämlich als "Komplexität der Berechnung" Sinn. Beispielsweise kann ein Programm umso komplexer (in struktureller Hinsicht) genannt werden, je mehr ineinandergeschachtelte Maschen und/oder Schleifen in ihm enthalten sind, je größer die Schachtelungstiefe (z.B. der Unterprogrammrufe) ist und je mehr Maschen und/oder Schleifen sich überschneiden (sog. Spaghettiprogramme). Dabei handelt es sich um logische strukturelle Komplexität, die sich auf den Berechnungsaufwand, auf den Programmieraufwand und auch auf die Häufigkeit von Programmierfehlern auswirken kann. In diesem Sinne verwendet fällt der Begriff der Berechnungskomplexität in die Vagheit zurück, von der in Kap.21.1 die Rede war und von der wir uns befreien wollten.

Dennoch kann diese "natürliche" Auslegung des Begriffs der Berechnungskomplexität fruchtbar sein. Voraussetzung ist, dass alle Begriffe exakt definiert sind. Das ist im Rahmen der rekursiven Funktionsdefinition durchaus möglich, indem Maschen und Schleifen auf iterative Rekursion reduziert, Schachtelungen zu Iterationszahlen konkretisiert werden. Auf diesem Wege kann eine Brücke zwischen der Theorie der Berechenbarkeit und der Theorie der Berechnungskomplexität (im Sinne der Komplexitätstheorie) geschlagen werden (siehe z.B. [Börger 92]).

Blättert man ein Buch über Komplexitätstheorie durch, erkennt man, dass die Theorie weitgehend auf der Algorithmentheorie, speziell auf der Theorie der Berechenbarkeit und noch spezieller auf der Theorie der Turingmaschine aufbaut. Das ist

---

2 Siehe u.a. [Reischuk 90], [Börger 92], [Duden 89].

ein weiteres Beispiel dafür, dass die Theoretiker bei der Suche nach einem neuen theoretischen Ansatz bestrebt sind, auf Begriffe und Methoden zurückzugreifen, die ihnen geläufig sind. Das ist verständlich; es ist der Gang der Evolution.

21.3 Modellierung komplexer Systeme und Prozesse

21.3.1 Strukturelle und nichtlineare Komplexität

Bei dem Versuch, das menschliche Denken zu simulieren, sind wir auf Grenzen gestoßen, die durch die Komplexität des Denkprozesses gezogen sind. Diese Erfahrung provoziert folgende allgemeine Frage:

Wieweit ist es im Prinzip möglich, komplexe Systeme und Prozesse zu modellieren und zu simulieren?

2

Ziemlich nichtssagend kann die Frage folgendermaßen beantwortet werden werden. *Bedingung für die Simulierbarkeit komplexer Systeme und Prozesse ist ihre sinnvolle Beschreibbarkeit als Problem mit höchstens polynomialer Komplexität.* Damit wollen wir uns nicht zufrieden geben, sondern nach aussagekräftigeren Antworten suchen, auch wenn sie nur partiell gültig und für konkrete Probleme sinnvoll sind. Dazu holen wir etwas weiter aus und knüpfen an Kap.15 an.

In Kap.15.8 [15.13] hatten wir an den Mathematikunterricht in der Schule erinnert und drei Schritte genannt, in denen eingekleidete Aufgaben zu lösen waren: Ansatzfindung, analytisches Rechnen und numerisches Rechnen. Diese drei Schritte müssen bei der Erstellung jedes kalkülisierten Modells ausgeführt werden. Von ihnen wollen wir bei der Beantwortung der gestellten Frage ausgehen. Der Leser beachte, dass die folgenden Ausführungen keine Fortsetzung unseres Weges zur künstlichen Intelligenz darstellen. In dieser Hinsicht bringen sie nichts Neues. Doch werden sie unser Bild von der Leistungsfähigkeit und den Einsatzmöglichkeiten des Computers vervollständigen.

Auf dem gegenwärtigen Stand von Wissenschaft und Technik bleibt der erste Schritt, die Ansatzfindung, d.h. die Findung eines kalkülisierten Modells, dem Menschen vorbehalten, unabhängig davon, wie komplex das zu modellierende Original ist. Wenn dieses ein *physisches* Objekt ist, sollte man denken, dass die *Physik* für seine Kalkülisierung zuständig ist. Doch scheint das nicht zu stimmen, wenn das Original ein lebender Organismus ist. Von alters her ist der Bereich physikalischer Forschung im Wesentlichen auf Objekte "sehr niedriger Komplexität" eingeschränkt, d.h. auf Objekte, die auf einer Komponierungsebene betrachtet werden, auf der sie sich als "einfach", als nicht komplex darbieten.

Bei ausreichender Dekomponierung wird jedes Objekt komplex. Man denke an die komplizierte Bewegung der Moleküle einer Flüssigeit oder eines Gases. Dank der sehr großen Anzahl der Moleküle emergieren globale (makroskopische) Merkmale, beispielsweise die Temperatur eines Gegenstandes oder der Druck eines Gases,

und es emergieren globale Gesetzmäßigkeiten, beispielsweise die Proportionalität zwischen der Temperatur und dem Produkt von Druck und Volumen eines (idealen) Gases oder die zeitliche Zunahme der Entropie. Letztere wurde zunächst im Rahmen der Thermodynamik ohne Dekomponierung, auf der "Komponierungsebene" der kollektiven Phänomene *abgeleitet*. Ludwig Boltzmann gelang die *Reduktion*, die Zurückführung des Entropiesatzes auf die molekulare (mikroskopische) Ebene, indem er die Komplexität (die Kompliziertheit der Molekülbewegung) mit Hilfe des Wahrscheinlichkeitsbegriffs "verdrängte" [5.22]. Damit legte er die Grundlage für die Erweiterung des Forschungsgegenstandes der Physik auf Objekte, deren Verhalten mit statistischen Methoden global (makroskopisch) beschreibbar ist, auch wenn sie mikroskopisch betrachtet komplexe Objekte darstellen.

Gegenwärtig vollzieht sich eine nicht weniger bedeutende Erweiterung physikalischer Forschung. Die Physiker sind dabei, sämtliche Phänomene, die in der Natur zu beobachten sind, in ihre Forschung einzubeziehen, das Phänomen des Lebens eingeschlossen. Im Zentrum der Bemühungen steht die Analyse sog. *nichtlinearer dynamischer Systeme*. Das Wort "nichtlinear" kennzeichnet zwei Charakteristiken derartiger Systeme, die *Nichtlinearität* der beschreibenden Gleichungen und die *Irregularität* oder Unstetigkeit der resultierenden Verhaltensweise. Das heißt zum einen, dass in den modellierenden Relationsgleichungen die Variablen nicht rein linear miteinander verknüpft sind, und zum anderen, dass kleine Ursachen (kleine Änderungen von Variablen- oder Parameterwerten) zu großen Wirkungen, zu Sprüngen im Verhalten führen können.

Ein Beispiel aus der Physik ist der "Sprung" einer unterkühlten Flüssigkeit aus dem flüssigen in den festen Aggregatzustand. Ein anderes Beispiel ist das Auftreten von Turbulenzen bei einer bestimmten Strömungsgeschwindigkeit. Allgemein wird der Wert der Variablen, bei dem der Umschlag eintritt, als *Schwellenwert* bezeichnet. In Kap.4.1 wurde ein spezieller Operator mit dieser Eigenschaft eingeführt, der Schwellenoperatoren (Bild 4.1), und in Kap.9.5 wurde eine Schaltung mit dieser Eigenschaft entwickelt, der Flipflop (siehe die Bilder 9.6c und 9.7). Wenn sich ein Schwellenoperator in unmittelbarer Nähe der Schwelle befindet, kann er bei minimaler Änderung der Eingabe- oder Schwellenspannung in den anderen, ("völlig anderen") Zustand überspringen. In der Hierarchie informationeller Systeme ermöglichen Schwellenoperatoren den Übergang aus dem kontinuierlichen Bereich der zurunde liegenden physikalischen Prozesse in den kausaldiskreten Bereich der informationellen Prozesse. Ohne sie ist auf dem Boden der klassischen Physik keine Informationsverarbeitung möglich. Zu diesem Schluss waren wir in Kap.8.2.4 [8.13] gelangt. Das Sprungverhalten eines Schwellenoperators wurde als gegeben angenommen und nicht weiter untersucht.

Das ändert sich jetzt. Die Untersuchung von Verhaltenssprüngen eines kontinuierlich beschriebenen Prozesses mit Hilfe der mathematischen Analysis ist der zentrale Gegenstand der *Theorie nichtlinearer dynamischer Systeme*. Das Verhalten solcher Systeme, ihre "*nichtlinerare Dynmik*" ist kompliziert und schwer durch-

schaubar. In diesem Sinne werden sie komplex genannt. Zur Unterscheidung von der strukturellen Komplexität sprechen wir von **nichtlinearer Komplexität**. Die Theorie nichtlinearer dynamischer Systeme befindet sich im Zustand des Suchens und Sammelns. Ein Blick in die Literatur[3] zeigt die thematische Breite der Forschung. Beobachtungen und Phänomene aus Physik, Biologie und Sozialwissenschaften werden hinsichtlich zugrundeliegender nichtlinearer Dynamik untersucht. Von besonderer Bedeutung ist die Selbstorganisation, der Basisprozess jeder Art von Evolution[4].

Es werden sicher noch viele Ideen und Abstraktionen erforderlich sein, bevor die divergierenden Arbeits- und Denkrichtungen zu einer einheitlichen Theorie konvergieren. Dass dies früher oder später gelingt, ist kaum zu bezweifeln. Die modellierende und kalkülisierende Kraft des menschlichen Geistes kennt kaum Grenzen. Das ist die Antwort auf die eingangs gestellte Frage nach der Modellierbarkeit komplexer Systeme und Prozesse, soweit die Frage den ersten Schritt des Modellierens, die Ansatzfindung, d.h. die Kalkülisierung betrifft.

Da der Computer vom ersten Schritt mathematischer Problemlösungen, also von der Ansatzfindung, ausgeschlossen ist, kann er beim Kalkülisieren des Komplexen nicht helfen. Doch ist er bei der Untersuchung komplexer Phänomene auf der Grundlage bereits ausformulierter kalkülisierter Modelle unentbehrlich. Um seine Unentbehrlichkeit in vollem Umfange zu erkennen, fragen wir zunächst ganz allgemein, wann die Hilfe des Computers angezeigt oder sogar notwendig sein kann. Die offensichtliche Antwort lautet: wenn sehr viele numerische Rechnungen erforderlich sind, denn ihre Ausführung ist die eigentliche Domäne des Computers und im numerischen Rechnen ist er dem Menschen weit überlegen. Es gibt zwei wesentliche Gründe dafür, dass Modellieren umfangreiche numerische Rechnungen erfordert; entweder enthält das Modell Gleichungen (genauer Relationsgleichungen), die analytisch nicht lösbar sind, oder das Modell besitzt einen niedrigen Kalkülisierungsgrad. Der erste Fall soll durch das folgende Beispiel, der zweite Fall durch zwei weitere Beispiele illustriert werden.

Beispiel 1. Eine "ruhig" (laminar) strömende Flüssigkeit bietet ein "einfaches" Bild. Der Strömungsprozess ist aus makroskopischer Sicht nicht komplex. Er wird es jedoch bei höheren Strömungsgeschwindigkeiten, wenn Turbulenzen und Wirbel auftreten. Es existiert eine geschlossene Strömungstheorie. Die Strömung von Flüssigkeiten wird durch die sog. *Navier-Stokes-Gleichungen* vollständig beschrieben, ein System nichtlinearer Differenzialgleichungen, das analytisch nicht allgemein lösbar ist. Die Folge war z.B. dass es in der Vergangenheit trotz intensiver Bemü-

---

3 Die Artikelsammlungen [Parisi 96] und [Parisi 98] demonstrieren die Vielfalt der behandelten Probleme. Eine Einführung in die Gesamtproblematik findet der Leser in [Prigogine 79], [Nicolis 87], [Gell-Mann 94].

4 Die Beziehung zwischen nichtlinearer Dynamik und Selbstorganisation ist kurz und sehr anschaulich in [Ebeling 91] und ausführlicher in [Ebeling 94] und [Ebeling 98] dargestellt.

hungen nicht möglich war, die Bildung stabiler Wirbel aus den Gleichungen herzuleiten. Das veranlasste manche Forscher zu der Annahme, Wirbelbildung beruhe auf einem quantenmechanischen Effekt, der durch das mathematische Modell nicht beschrieben wird. Der Computer brachte die Lösung. In Simulationsexperimenten auf der Grundlage der Navier-Stokes-Gleichungen gelang es zu demonstrieren, wie bei geeigneten Nebenbedingungen (Geometrie des Kanals, Dichte und innere Reibung der Flüssigkeit) Turbulenzen entstehen und wie sich stabile Wirbel ausbilden. Damit hat der Computer gewissemaßen *seinen eigenen Beitrag zur Erkenntnisgewinnung* geleistet, denn er hat ein Problem gelöst, das zwar vollständig kalkülisiert, aber für viele interessante Fälle nicht hatte gelöst werden können. Inzwischen hat sich die Theorie nichtlinearer Dynamik des Problems angenommen, und eine ganze Reihe spezieller Lösungs- und Simulationsmethoden sind entwickelt worden[5].

3

Beispiel 2. Im Gegensatz zur Flüssigkeitsströmung lässt sich für komplexe Produktionsbetriebe kein geschlossenes analytisches Modell angeben. In Kap.18.3 [18.7] hatten wir festgestellt, dass solche Prozesse nicht global, sondern nur punktuell kalkülisiert werden können, sodass sich ein Modell mit niedrigem Kalkülisierungsgrad ergibt. Wenn auf der Grundlage eines solchen Modells die Produktion gesteuert werden soll, wird der numerische Rechenaufwand in der Regel so hoch, dass nur der Computer ihn bewältigen kann. (Es sei an die formale Definition des Kalkülisierungsgrades als Verhältnis von analytischem zu numerischem Rechenaufwand bei der Ableitung von Modellaussagen erinnert [18.9].) Man beachte, dass sich an diesem Sachverhalt auch dann nichts ändert, wenn der Produktionsbetrieb als Operatorenhierarchie nach der USB-Methode oder nach irgendeiner anderen Softwareentwurfsmethode beschrieben ist, denn die *USB-Methode wie auch jede andere Methode leistet **nicht** die globale Kalkülisierung der Hierarchie und der in ihr ablaufenden Prozesse, die mit Hilfe der Methode komponiert werden.*

Beispiel 3. Ähnlich ist die Situation hinsichtlich neuronaler Netze, von denen in Kap.9.2.2 und 9.4 die Rede war. Auch sie lassen nur einen niedrigen Kalkülisierungsgrad zu. Aber im Gegensatz zu Produktionsbetrieben handelt es sich um relativ homogene Systeme, wodurch die Erarbeitung einer analytischen Theorie erleichtert wird. Gegenwärtig ist man auf die Simulation des Netzverhaltens mittels Computer angewiesen oder auf die nichtsprachliche (analoge) Modellierung, d.h. auf die Herstellung einer großen Anzahl von künstlichen Neuronen z.B. in Form mikroelektronischer Schaltungen und deren Vernetzung. Nur in speziellen Fällen und hinsichtlich spezieller Eigenschaften ist eine analytische Beschreibung gelungen (siehe z.B. [Nauck 96], [Kistler 98]). Das Besondere der Arbeit [Kistler 98] besteht darin, dass Netze betrachtet werden, die nicht aus "*statischen*" Neuronen aufgebaut sind, die in Kap.9.2.2 beschrieben wurden, sondern aus sog. *spiking neurons* oder *Impulsneuronen*. Impulsneuronen besitzen nur nichtstabile Anregungszustände, aus denen sie

5 Siehe z.B. [Oertel 95], [Parisi 96], [Parisi 98].

sehr schnell in den Ruhezustand zurückkehren. Doch können sich im Netz verschiedene dynamisch stabile Zustände ausbilden. Auf symbolischer Ebene muss also mit *dynamischer Codierung* gearbeitet werden (siehe Kap.9.1).

An das dritte Beispiel sollen einige Überlegungen angeschlossen werden, obwohl sie über den Rahmen des Buches hinausführen. Sie betreffen ein Problem, das sich beim Übergang von der traditionellen zur alternativen Informationsverarbeitung ergibt und mit der Komplexität neuronaler Netze zusammenhängt. Zunächst ist festzustellen, dass ein neuronales Netz, sei es ein natürliches oder ein künstliches, nicht nur durch strukturelle, sondern auch durch nichtlineare Komplexität gekennzeichnet ist, denn das Verhalten der Bausteine ist nichtlinear. Das hat beispielsweise zur Folge, dass sich die globale Verhaltensweise eines künstlichen neuronalen Netzes bei minimaler Änderung der Schwellenspannung eines Neurons sprunghaft ändern kann [9.9]. Für ein Netz mit binärer Eingabe kann sich die boolesche Funktion bzw. die Binärwortfunktion sprunghaft ändern, die vom Netz berechnet wird. Aus dieser Sicht wird verständlich, dass die Untersuchung nichtlinearer dynamischer Systeme im Zusammenhang mit der Frage nach der Entstehung und Verarbeitung von Information auf subsymbolischem Niveau zunehmend an Bedeutung gewinnt[6].

Das Verhalten natürlicher und großer künstlicher neuronaler Netze wird infolge ihrer Komplexität undurchschaubar. Das scheint eine fatale Konsequenz zu haben. Es bedeutet nämlich, dass die Prozesse, die durch eine Eingabe im Netz auslöst werden, unbekannt sind, nicht vorhergesagt und auch nicht nachvollzogen werden können. Wie aber kann ein neuronales Netz oder allgemeiner ein Neurocomputer als Information verarbeitendes System verwendet werden, wenn die interne Semantik einer Eingabe - so hatten wir die durch eine Eingabe ausgelösten Prozesse genannt [5.7] - nicht bekannt ist. Wie kann dann die Forderung erfüllt werden, dass die Semantik der Ausgabe in eindeutiger Weise an die der Eingabe über die interne Semantik angebunden sein muss, um sie interpretieren zu können. Der Prozessorrechner erfüllt diese Forderung. Da der Neurocomputer sie nicht erfüllt, scheint er für die Informationsverarbeitung unbrauchbar zu sein.

Dass hier ein Trugschluss vorliegen muss, zeigt die Tatsache, dass Menschen sich verstehen können, obwohl ihnen die Vorgänge im Gehirn unbekannt sind. Um die Quelle des Trugschlusses zu finden, verallgemeinern wir die Problemstellung und fragen: Wie kann man psychologische Verhaltensweisen ohne Neurophysiologie verstehen? Und noch allgemeiner fragen wir: Wie kann man globale (makroskopische) Verhaltensweisen (Zusammenhänge, Gesetzmäßigkeiten) verstehen, ohne die zugrunde liegenden lokalen (mikroskopischen) Verhaltensweisen im Detail zu kennen?

Diese Fragestellung erinnert an das Problem der Beschreibung des *makro*skopischen Verhaltens eines Gases, dessen *mikro*skopisches Verhalten infolge seiner

---

6 Siehe z.B. [Ebeling 91], [Ebeling 98].

Komplexität unbekannt ist. Wir stehen vor einem ähnlichen Problem wie Ludwig Boltzmann, als er versuchte, das makroskopische (globale) Verhalten eines Gasvolumens aus dem im Detail nicht bekannten mikroskopischen Verhalten der Moleküle abzuleiten [5.22]. In dieser Analogie entspricht das Befinden eines bestimmten Moleküls in einer bestimmeten Zelle des Gasvolumens dem Befinden eines bestimmten Neurons in einem bestimmten Zustand, und ein Mikrozustand des Gases [5.22] entspricht dem Anregungszustand (Anregungsmuster) des Netzes. Daraus wird verständlich, dass man bei dem Bemühen, das Verhalten neuronaler Netze mathematisch zu modellieren, zur statistischen Beschreibung überging und nach Boltzmanns Vorbild die mikroskopische Komplexität des Originals mit Hilfe des Wahrscheinlichkeitsbegriffs in den "Griff" zu bekommen suchte. Dadurch wurde eine mathematische Beschreibung vieler Eigenschaften neuronaler Netze möglich (siehe z.B. [Nauck 96], [Churchland 97]). Auf diese Weise lässt sich sowohl strukturelle Komplexität "verdrängen", z.B. durch zufällig generierte Verbindungsstrukturen, als auch Verhaltenskomplexität, z.B. durch Fluktuationen (stochastische Schwankungen) der Schwellenspannungen der Neuronen. Die Komplexität neuronaler Netze lässt sich also durchschaubar und beherrschbar machen, und damit lassen sich auch die Ausgaben neuronaler Netze interpretierbar machen.

Es gibt eine viel einfachere Antwort auf die Frage nach der Interpretierbarkeit der Ausgaben neuronaler Netze. Doch hat sie wenig mit der Verarbeitung von "Informationen", genauer von Zeichenrealemen durch das Netz zu tun. In Kap.9.4 [9.22] hatten wir festgestellt, dass man die Ein- und Ausgaben neuronaler Netze mit vielen Ein- und Ausgabeneuronen als gerastertes Schwarz-weiß-Muster auffassen kann. Wenn man in einem Ausgabemuster ein bekanntes Objekt *erkennt*, z.B. den Buchstaben A, so hat man die Ausgabe *interpretiert*. Die Interpretation beruht auf geometrischen Merkmalen des Musters selber, nicht auf irgendeiner apriori zugewiesenen Semantik, die durch das Muster codiert wird. Mit anderen Worten, das Muster wird nicht als *Zeichenrealem*, sondern als *Urrealem* verstanden.

Anders verhält es sich beim Interpretieren der Ausgaben eines Prozessorrechners. Diese tragen eine *Apriori-Semantik*, eine von vornherein zugewiesene Semantik; sie haben für den Nutzer eine *Bedeutung*, von welcher der Computer nichts "weiß", weil er kein Bewusstsein besitzt. Bisher waren wir davon ausgegangen, dass die Ausgaben eines Prozessorrechners auf deterministische und durchschaubare Weise an die Eingaben angebunden sind. Diese Vorstellung muss korrigiert werden, wenn der Prozessorrechner ein neuronales Netz simuliert. Dann ist der Überführungsprozess von Eingaben in Ausgaben infolge seiner Komplexität i.d.R. nicht mehr durchschaubar. Wenn der Prozessorrechner ein stochastisches neuronales Netz simuliert (mit Hilfe eines Zufallszahlengenerators), ist der Überführungsprozess auch nicht mehr deterministisch. Nichtsdestoweniger kann die Anbindung von externer Semantik möglich sein, beispielsweise dadurch, dass der Nutzer im Ausgabemuster ein ihm bekanntes Objekt erkennt. Die Anbindung externer Semantik ist auch dadurch möglich, dass den Ausgaben des Netzes sinnvolle Zeichenketten (Zeichenrealeme)

durch Vereinbarung zugeordnet werden. Ein entsprechender Zuordner kann implementiert werden, sodass ein Bild, das dem Netz “gezeigt” wird, mit einer Zeichenkette reagiert, z.B. mit der Ausgabe “Das gezeigte Objekt ist ein A”.

Durch die Möglichkeit der Simulation neuronaler Netze auf Prozessorrechnern ist die naheliegende Unterscheidung zwischen traditioneller und alternativer KI nach dem Träger der Intelligenz (Prozessorcomputer bzw. Neurocomputer) nicht korrekt, vielmehr muss nach der Betrachtungsebene (symbolische bzw. subsymbolische) [4] unterschieden werden. Die korrekte Definition lautet: *Auf symbolischer bzw. subsymbolischer Ebene simulierte natürliche Intelligenz heißt* ***traditionelle*** *bzw.* ***alternative KI***.

21.3.2* Fuzzy-Kalkül

Im vorangehenden Kapitel haben wir gesehen, wie es möglich ist, Komplexität mit Hilfe des Wahrscheinlichkeitsbegriffs mathematisch beherrschbar zu machen. Es gibt eine weitere Möglichkeit, die besprochen werden soll. Sie bedient sich des *Fuzzy-Kalküls*. Zum Verständnis der Idee, die dem Fuzzy-Kalkül zugrunde liegt, ist die Beobachtung hilfreich, dass im Alltag drei Methoden der Beschreibung von Objekten zur Anwendung kommen. Für jede Methode geben wie ein Beispiel. Der Lehrer einer ersten Klasse schreibt ein A an die Tafel und sagt: “Das ist ein A”. Ein Biologielehrer sagt: “Der Fliegenpilz ist ein relativ großer und ziemlich giftiger Pilz mit rotem bis gelblichem Hut”. Ein Geographielehrer sagt: “Der Rhein ist 1320 km lang”.

Es handelt sich um drei unterschiedliche Methoden, ein Objekt zu beschreiben. Die erste Aussage “beschreibt” das Objekt “A” durch Präsentation und Benennung. Die zweite Aussage beschreibt das Objekt “Fliegenpilz” durch *linguistische* Angabe der Werte (Ausprägungen) von drei Merkmalen, Größe, Giftigkeit und Hutfarbe. Das Wort “linguistisch” bedeutet hier soviel wie “umgangssprachlich-unscharf”. Eine Aussage, die einem Objekt einen oder mehrere unscharfen Merkmalswerte zuordnet, heißt *unscharfe Aussage* oder **unscharfes Prädikat**. Die dritte Aussage beschreibt das Objekt “Rhein” durch quantitative (scharfe, exakte) Angabe des Wertes des Merkmals “Länge” (durch ein “scharfes” Prädikat). Die Beispiele illustrieren drei Beschreibungsmethoden von Objekten, die wir die **präsentative** Methode, die *unscharfe* oder **linguistische** Methode und die *scharfe* oder **exakte** Methode nennen. Die exakte Methode heißt **quantitativ**, wenn die Merkmalswerte numerisch angegeben werden, wie im Falle der Länge des Rheins.

In den drei Beispielen werden Menschen belehrt und erwerben dadurch Wissen. Durch den Wissenserwerb erhöht der Belehrte seine Fähigkeit zum sprachlichen Modellieren, das, wie wir wissen, auf dem Operieren mit Merkmalen beruht [5.13]. An erster Stelle steht dabei das Klassifizieren, auf dem das Beschreiben, das Wiedererkennen und das Entscheiden beruht. Es stellt sich die Frage, mit Hilfe welcher der drei Methoden Computer belehrt werden können. Dass die exakte Methode anwendbar ist, haben wir in Kap.16 erkannt. Das Belehren erfolgte dort u.a. in Form

von Wissensakquisition bei der Erstellung und Erweiterung der Wissensbasen von Datenbanken und Expertensystemen. Dass die präsentative Methode anwendbar ist, ergibt sich aus unseren Überlegungen in Kap.9.4, die zeigten, dass neuronale Netze das Klassifizieren von Objekten erlernen können [9.23]. Das Belehren kann folgendermaßen bewerkstelligt werden. Gegeben sei ein Netz mit einer mehr oder weniger zufälligen Struktur (mit zufälligen Synapsenwerten). Dem Netz werden der Reihe nach die bereits klassifizierten Objekte einer sog. *Stichprobe* präsentiert (“gezeigt”) und dazu die jeweilige Klasse angegeben.[7] Das Netz klassifiziert jedes Objekt der Stichprobe und vergleicht die zugeordnete Klasse mit der “richtigen” Klasse, die durch die Stichprobe vorgegeben ist. Wenn die eigene Klassifizierung falsch ist, werden die Gewichte der Synapsen nach einem geeignet formulierten Lernalgorithmus verändert. Bei ausreichendem Stichprobenumfang und ausreichender Anzahl von Objektpräsentationen kann das Netz nach dem Anlernprozess selbständig klassifizieren. Das “interiorisierte” Wissen ist auf die Synapsengewichte verteilt, es ist “strukturell” gespeichert. Die Fähigkeit zu lernen und zu klassifizieren kann durch Simulation auf Prozessorcomputer übertragen werden.

Wie man sieht, liegt die linguistische Methode hinsichtlich der Genauigkeit der Objektbeschreibung zwischen der präsentativen und der exakten Methode. Insofern sollte man erwarten, dass auch sie zur Belehrung eines Computers anwendbar ist. Doch liefern unsere bisherigen Überlegungen wenig Anhaltspunkte für die Realisierung der Methode. Darum soll auf sie etwas ausführlicher eingegangen werden.

Zunächst versetzen wir und in die Lage eines Meisters, der seinen Lehrling instruiert. Wir betrachten zwei Fälle. Im ersten instruiert ein Gärtner seinen Lehrling, wie Äpfel in verschiedene Körbe (Klassen) nach dem Reifegrad einzusortieren sind. Im zweiten instruiert ein Anlagenfahrer seinen Lehrling, wie die Anlage, beispielsweise eine Heizungsanlage, zu bedienen (steuern) ist. Im Sortierbeispiel wird der Meister (Gärtner) wahrscheinlich die präsentative Methode anwenden und zeigen, welche Äpfel in welche Körbe gehören. Das Lernen kann ohne Meister erfolgen, wenn jeder Korb bereits eine ausreichende Anzahl richtig einsortierter Äpfel enthält (Lernen ohne Lehrer). Der Meister kann auch die linguistische Methode anwenden und beispielsweise die Instruktion (*linguistische Regel*) geben

Lege die reifen Äpfel in Korb 1, die halbreifen in Korb 3! (21.1)

oder

Lege die halbreifen, mittelgroßen Äpfel in Korb 1! (21.2)

Im Steuerbeispiel wird der Meister sehr wahrscheinlich die linguistische Methode wählen. Seine Instruktionen (Bedienungsregeln) könnten z.B. lauten:

7 In Kap.9.4 [9.22] wurde erklärt, wie das “Zeigen” erfolgen kann.

Wenn dieses Rohr handwarm ist, dann mit mittlerer Leistung heizen. (21.3)

oder

Wenn das Rohr sehr heiß ist und die Druckanzeige die rote Marke erheblich überschreitet, dann das rote Ventil weit öffnen. (21.4)

Die beiden Steuerregeln sind Wenn-dann-Sätze und stellen linguistische (unscharfe) Implikationen dar. Eine vollständige Bedienungsanleitung, die mehrere Regeln enthält, stellt eine Entscheidungstabelle [12.5] dar.

Wir fragen nun, ob an die Stelle des Lehrlings ein Computer treten kann. Wir beginnen die Beantwortung mit einer vorbereitenden Überlegung. Angenommen, der Anlagenfahrer gibt die exakte Instruktion:

Wenn die Temperatur auf 90° steigt, dann die Leistung auf 24 kW herunterfahren.

Es handelt sich um eine *exakte* Regel. Sie ist nur dann ausführbar, wenn die Temperatur messbar und die Leistung exakt einstellbar ist. Eine vollständige und exakte Bedienungsanleitung setzt voraus, dass alle zu beobachtenden Merkmale (die **Messgrößen**) messbar und alle zu steuernden Merkmale (die **Stellgrößen**) exakt einstellbar sind. Wenn ein analytisches Modell des zu steuernden Prozesses existiert, können die Stellwerte aus den Messwerten nach Formeln berechnet werden, die aus dem Modell ableitbar sind. In diesem Falle kann die Steuerung durch einen Analogrechner erfolgen (siehe Kap.4.2). Für seinen Einsatz ist ein analytisches Modell unabdingbar. Dagegen kann ein Digitalrechner auch dann eingesetzt werden, wenn kein analytisches Modell existiert. Hierfür ist die in Kap. 12.3.4 [12.5] behandelte Steuerung eines Waschautomaten ein Beispiel. Das Belehren des Computers besteht im Implementieren von Steuervorschriften. Im Falle des Waschautomaten bestand es konkret im Einspeichern (Einprägen) der Entscheidungstabelle in die Matrix des Steuerwerks.

Nun wenden wir uns der zentralen Frage dieses Kapitels zu:
Wie kann ein Computer durch Eingabe ***linguistischer*** *Regeln zum Klassifizieren und zum Steuern befähigt werden?* Es wird vorausgesetzt, dass alle relevanten Größen exakt gemessen bzw. eingestellt werden können. Die Lösung des Problems ist von kaum zu überschätzender Bedeutung, denn sie ermöglicht u.a. die automatische Steuerung von Prozessen, die infolge ihrer Komplexität nur *linguistisch* beschrieben werden können und folglich nach *linguistisch* formulierten Regeln zu steuern sind.

Da der Computer nur im Rahmen eines Kalkül "denken" und belehrt werden kann, muss ein Kalkül gefunden bzw. erfunden werden, der erlaubt, die linguistischen Regeln in die Sprache des Kalküls zu übersetzen. Die Erfindung gelang Lofti A. Zadeh [Zadeh 65]. Der Kalkül baut auf dem Begriff der **unscharfen Klasse** oder *unscharfen Menge*, englisch *Fuzzy Set*, auf. Wir werden die Bezeichnung "unscharfe Klasse" bevorzugen.

Wie wir wissen, werden Klassen durch Merkmalswerte festgelegt. Wenn die Merkmalswerte unscharf definiert sind, wie beispielsweise “ziemlich giftig”, ist auch die durch sie festgelegte Klasse unscharf definiert, m.a.W. *eine* ***unscharfe Klasse*** *wird durch ein* ***unscharfes Prädikat*** *festgelegt*. Das Regelwerk für das Operieren mit unscharfen Klassen wird üblicherweise - auch in der deutschsprachigen Literatur - als **Fuzzy-Logik** bezeichnet. Sehr sinnfällig wäre die Bezeichnung “*unscharfe Klassenlogik*”. Wir werden, getreu den Sprachgewohnheiten des Buches, die Bezeichnung **Fuzzy-Kalkül** verwenden. Die potenzielle Bedeutung des Fuzzy-Kalküls für die Informatik wird deutlich, wenn man sich klar macht, dass umgangsprachliches Modellieren auf dem Operieren mit linguistischen Merkmalswerten, vor allem auf dem Klassifizieren nach linguistischen Merkmalswerten beruht.

Die Grundideen des Fuzzy-Kalküls sollen anhand eines Roboters erläutert werden, der Äpfel sortiert. Der Roboter soll in der Lage sei, die Anweisungen (21.1) und (21.2) richtig zu befolgen. Wir gehen davon aus, dass der Roboter über alle erforderlichen mechanischen und messtechnischen Einrichtungen verfügt. Die Größe werde durch Wiegen und der Reifegrad durch Messen der mittleren Wellenlänge λ des vom Apfel ausgesendeten Lichts (der Farbe des Apfels) bestimmt. Uns interessiert lediglich, wie der Roboter verfährt, wenn er aufgrund der beiden Messergebnisse entscheidet, ob ein Apfel einer auszusortierende Klasse angehört oder nicht.

Um zu erkennen, wie diese Aufgabe in unser Gebäude der Informatik einzuordnen ist, betrachten wir Bild 18.1. Ein kurzer Blick wird genügen, um zu erkennen, dass unser Problem darin besteht, den Übergang von der oberen zur mittleren Ebene zu ermöglichen, m.a.W. das Problem liegt in der Anbindung externer an formale Semantik. Um sie zu ermöglichen, muss eine Beziehung zwischen den unscharfen Wörtern der Auftragssprache “mittelgroß” und “halbreif” und den Messwerten, mit denen der Computer hantiert, definiert werden. Die Messwerte sind Elemente der Kalkülsprache und tragen als solche zunächst noch keine externe Semantik. Die Grundidee des Fuzzy-Kalküls besteht darin, die gesuchte Beziehung in Form des sog. *Zugehörigkeitsgrades* einzuführen. *Der* ***Zugehörigkeitsgrad*** *gibt an, in welchem Grade ein Objekt mit bestimmten Werten eines oder mehrerer Merkmale (Messwerten) zu einer entsprechenden* ***linguistschen*** *(d.h. linguistisch charakterisierten) Klasse gehört*. Der Zugehörigkeitsgrad wird mit μ bezeichnet. Er kann die Werte von 0 bis 1 annehmen. Der Verlauf des Zugehörigkeitsgrades in Abhängigkeit vom Messwert heißt **Zugehörigkeitsfunktion**.

Bild 21.1 zeigt zwei Zugehörigkeitsfunktionen und zwar den Verlauf des Zugehörigkeitsgrades in Abhängigkeit von λ für zwei Klassen, für die Klasse der reifen Äpfel (rechtes durchgezogenes “Dreieck”, der Index R kann als “reif” oder auch als “rot” gelesen werden) und für die Klasse der halbreifen Äpfel (linkes “Dreieck”, der Index H ist als “halbreif” zu lesen). Die Festlegung einer Zugehörigkeitsfunktion erfolgt relativ willkürlich, hat aber zwei Forderungen zu erfüllen. Zum einen muss sie die Erfahrung der Fachleute (des Meisters, der seinen Lehrling instruiert) wiedergeben und dem Urteil von Experten entsprechen, zum anderen sollte sie schnell

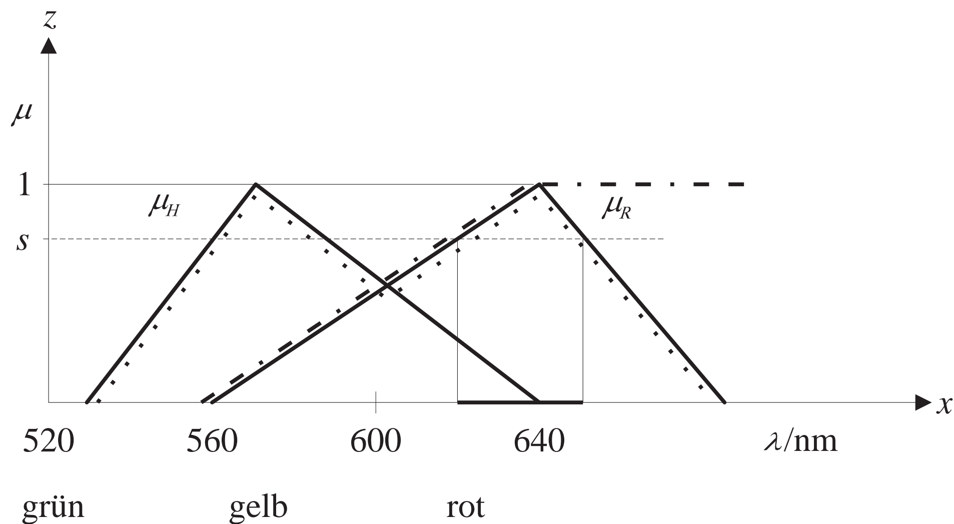


Bild 21.1. Zugehörigkeitsfunktionen von Äpfeln, die Licht der mittleren Wellenlänge λ aussenden, zur Klasse *H* der halbreifen bzw. zur Klasse *R* der reifen Äpfel.

berechenbar sein. Die Dreiecksform entspricht der zweiten Forderung. Man beachte, dass die Zuordnung zwischen Reifegrad und Wellenlänge in zwei Schritten erfolgt. Im ersten Schritt wird dem Reifegrad eine Farbe und im zweiten der Farbe eine Wellenlänge zugeordnet. Es handelt sich um ein *indirekte* Zuordnung (indirekte "Fuzzifizierung" s.u.) was jedoch das Ergebnis nicht beeinflusst, da ein Experte beide Schritte in einem Schritt ausführt.

Wenn der Sortierroboter die reifen Äpfel aussortieren soll, muss er die Funktion $\mu_{\mathrm{R}}(\lambda)$ "kennen" und es muss ihm "gesagt" werden, welche Äpfel, d.h. Äpfel mit welchem μ-Wert auszusortieren sind. Das kann durch Vorgabe einer *Schwellenfunktion* $s(\lambda)$ erfolgen. Alle Äpfel mit $\mu \geq s$ sind auszusortieren. Wenn die Schwellenfunktion durch die gestrichelte Gerade in der konstanten Höhe s über der λ-Achse vorgegeben ist, sortiert der Roboter diejenigen Äpfel aus, deren Wellenlänge (Farbe) in den dick gezeichneten Abschnitt der λ-Achse fällt.

Falls die Arbeit des Roboters den "Besteller" nicht befriedigt, kann sie an dessen Wünsche angepasst werden. Wenn zu unreife Äpfel aussortiert werden, kann beispielsweise eine Zugehörigkeitsfunktion verschoben oder ihre Form verändert oder die Schwellengerade kann gedreht werden. Wenn auch überreife Äpfel aussortiert werden sollen, kann man probeweise die Dreiecksfunktion durch eine Rampenfunktion ersetzen (strichpunktiert angedeutet). Der Eingriff wird dann erfolgreich sein, wenn bei Überreifung die mittlere Wellenlänge sich in Richtung des infraroten Bereiches verschiebt.

Die Festlegung einer Zugehörigkeitsfunktion heißt ***Fuzzifizierung*** *des betreffenden linguistischen Merkmalswertes. Die Festlegung erfolgt* ***intuitiv*** *nach Zweckmäßigkeitserwägungen*. Eine erste Festlegung erfolgt "auf gut Glück" und kann während der praktischen Anwendung korrigiert werden.

Betrachten wir nun die Sortieranweisung

Wenn ein Apfel halbreif ODER reif ist, dann sortiere ihn aus! (21.5)

In diesem Fall muss eine Verknüpfung von Merkmalen fuzzifiziert werden. Der Wenn-Satz (die Prämisse der Implikation) stellt eine *disjunktive* Verknüpfung der unscharfen Prädikate "ist halbreif" und "ist reif" dar. Die so festgelegte unscharfe Klasse ist die Vereinigung der unscharfen Klasse H der halbreifen Äpfel mit der unscharfen Klasse R der reifen Äpfel, formal notiert.

$$H \cup R = \{a\text{: } (P_1 \text{ OR } P_2)\}. \quad (21.6)$$

Mit P_1 und P_2 sind die Prädikate und mit a die Elemente (Apfelexemplare) der definierten Vereinigungsklasse bezeichnet. Wer sich an Kap.11 erinnert, wird erkennen, dass wir es mit einer unscharfen Variante der Beziehung zwischen Aussagenalgebra und Mengenalgebra (Klassenlogik) zu tun haben, die in Kap11.1 dargelegt wurde. Die Festlegung der Vereinigungsklasse durch die disjunktive Regel (21.5) entspricht der Formel (11.1b). Weiter unten werden wir auf die unscharfe Entsprechung der Formel (11.1a) stoßen.

Wir wollen uns überlegen, wie die Zugehörigkeitsfunktion $\mu_{H \cup R}(\lambda)$ der durch (21.5) definierten Vereinigungsklasse $H \cup R$ festgelegt werden kann. Eine häufig verwendete Möglichkeiten ist in Bild 21.1 durch die dick punktierte Linie veranschaulicht. Der Leser wird erkennen, dass die so definierte Zugehörigkeitsfunktion in jedem Punkt dem größeren der beiden Werte μ_H und μ_R entspricht, formal als Maximum-Funktion notiert:

$$\mu_{H \cup R}(\lambda) = \max(\mu_H(\lambda), \mu_R(\lambda)). \quad (21.7)$$

Die gleiche Überlegung stellen wir nun bezüglich der Anweisung (21.2) an. Zunächst formen wir sie um in die Anweisung

Wenn ein Apfel mittelgroß UND halbreif ist, dann sortiere ihn aus! (21.8)

Die Prämisse dieser Implikation stellt eine *konjunktive* Verknüpfung zweier unscharfer Prädikate dar.

Die durch ihn festgelegte unscharfe Klasse ist die Durchschnittsklasse $H \cap M$ der Klasse H der halbreifen Äpfel und der Klasse M der mittelgroßen Äpfel, formal notiert:

$$H \cap M = \{a\text{: } (P_1 \text{ AND } P_2)\} \quad (21.9)$$

Mit P_1 und P_2 sind die Prädikate "ist halbreif" bzw. "ist mittelgroß" und mit a die Exemplare (Äpfel) der Schnittklasse bezeichnet. Der Leser wird die Entsprechung zu Formel (11.1a) erkennen. Es stellt sich die analoge Frage wie oben: Wie kann die zweistellige Zugehörigkeitsfunktion $\mu_{H \cap M}(\lambda, G)$ für die Durchschnittsklasse $H \cap M$

festgelegt werden? Als Größenmerkmal G kann sowohl die Größe (das Volumen) als auch das Gewicht der Äpfel dienen.

Bild 21.2 zeigt eine sehr einfache Möglichkeit, die Zugehörigkeitsfunktion festzulegen. Es ist ein dreidimensionales Koordinatensystem perspektivisch dargestellt. Entlang der x-Achse ist die Wellenlänge, entlang der y-Achse das Gewicht und entlang der z-Achse die Zugehörigkeitsfunktion abgetragen. Die x-y-Ebene heißt *Merkmalsebene*. Die dargestellte Zugehörigkeitsfunktion hat die Form einer Pyramidenoberfläche. Sie ergibt sich aus der vernünftigen Festlegung, dass der Zugehörigkeitsgrad eines Apfels zu Klasse $H \cap M$ mit dem kleineren der beiden Werte μ_H und μ_M zusammenfallen soll. Folglich kann die Zugehörigkeitsfunktion $\mu_{H \cap M}$ formal als Minimum-Funktion notiert werden:

$$\mu_{H \cap M}(\lambda, G) = \min(\mu_H(\lambda), \mu_R(G)). \tag{21.10}$$

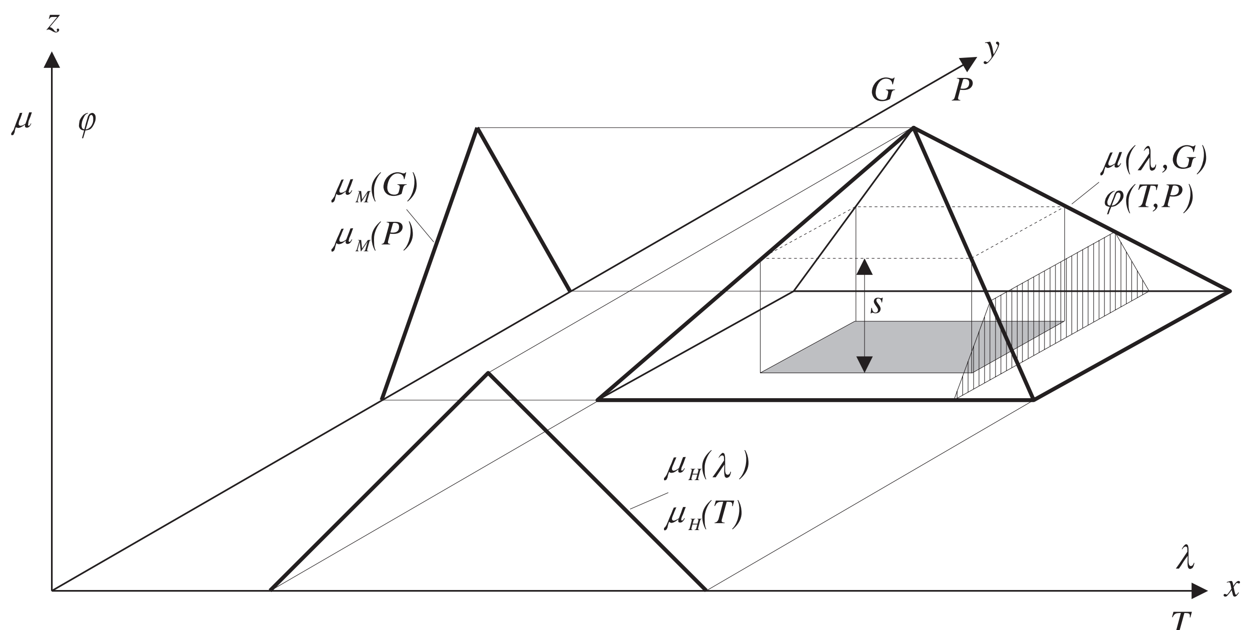


Bild 21.2. Zugehörigkeitsfunktion $\mu(\lambda, G)$ (Fuzzifizierung der Sortierregel (21.8) bzw. Erfüllungsfunktion $\varphi(T,P)$ (Fuzzifizierung der Steuerregel (21.3) λ - Wellenlänge, T - Temperatur, G - Gewicht, P - Leistung. Indizes: H - halbreif bzw. handwarm, M - mittelgroß bzw. mittlere Leistung.

Damit der Sortierroboter die Anweisung (21.8) ausführen kann, muss ihm neben der Zugehörigkeitsfunktion eine Schwellenfunktion bekannt gegeben werden. Sie kann z.B. als Ebene in Der Höhe s über der Merkmalsebene festgelegt sein. Ihre Schnittlinie mit der Pyramidenoberfläche ist in Bild 21.2 gestrichelt eingezeichnet. Mit diesem Wissen ausgerüstet sortiert der Roboter alle Äpfel aus, die in den in Bild 21.2 grau unterlegten Merkmalswertebereich fallen.

Wir wollen nun die Sortieraufgabe verfeinern. Der Roboter soll die Äpfel nach 7 Ausprägungsgraden (z.B. sehr schwach, schwach, halbschwach, mittelmäßig, halb-

stark, stark, sehr stark) beider Merkmale in 49 Körbe einsortieren. Die Bedienungsanweisung (Regeltabelle) besteht nun aus 49 Regeln. Die Zugehörigkeitsfunktionen der Merkmalsausprägungen mögen die Form von Dreiecken besitzen, 7 über der λ-Achse und 7 über der G-Achse. Die Dreiecke mögen sich zum Teil überlappen. Nun ist für jede Regel, d.h. für jede Klasse (jeden Korb) die zweidimensionale Zugehörigkeitsfunktion gemäß (21.10) zu berechnen. Es ergeben sich 49 Pyramiden, die sich z.T. gegenseitig durchdringen. Für einen Punkt der Merkmalsebene, in dem sich mehrere Pyramiden durchdringen (in denen mehrere Regeln anwendbar sind), ergeben sich mehrere Zugehörigkeitsgrade, aus denen vernünftigerweise der größte Wert als Zugehörigkeitsgrad in diesem Punkt zu wählen ist. Die zweidimensionale Zugehörigkeitsfunktion lässt sich also als Maximum-Funktion notieren:

$$\mu = \max(\mu^1, \mu^2, \ldots \mu^r, \ldots) \tag{21.11}$$

Dabei sind die Variablen λ und G der Zugehörigkeitsfunktionen unterdrückt. Der obere Index r ist die laufende Nummer der Regeln. Die μ^r-Werte berechnen sich als Minimum-Funktion gemäß (21.10). Setzt man die entsprechenden Ausdrücke in (21.11) ein, ergibt sich eine sog. Minimax-Funktion. Geometrisch veranschaulicht stellt sie eine "stilisierte Hügellandschaft" dar.

Wir haben bisher die Größe μ als *Zugehörigkeitsgrad* bezeichnet. Die Bezeichnung ist sehr sinnfällig, da der μ-Wert eines Objektes mit unscharfen Merkmalswerten die "Zugehörigkeit" des Objektes zu einer bestimmten unscharfen Menge charakterisiert. Der Wert von μ in einem bestimmten Punkt der Merkmalsebene kann aber auch anders interpretiert werden, nämlich als **Erfüllungsgrad der Regel**, aus welcher sich μ durch Fuzzifizierung ergibt. Wenn ein Apfel die Prämisse der Sortierregel (21.8) im Grade μ *erfüllt*, dann *gehört* er der Klasse der halbreifen, mittelgroßen Äpfel im Grade μ *an*. Im Weiteren werden wir die Wörter *Zugehörigkeitsgrad* und *Erfüllungsgrad* in Abhängigkeit von Kontext verwenden.

Damit beenden wir das Sortierproblem (Klassifikationsproblem) und wenden uns dem Steuerproblem zu. Die Aufgabe besteht jetzt in der Fuzzifizierung linguistischer *Steuer*regeln, beispielsweise der Steuerregel (21.3). Es wird sich herausstellen, dass die Fuzzifizierungsprozedur der Sortierregel (21.2) anwendbar ist. Um das zu erkennen, tragen wir entlang der x-Achse in Bild 21.2 die Messgröße T (Temperatur) und entlang der y-Achse die Stellgröße P (Leistung) auf. Den linguistischen Merkmalswerten "halbreif" und "mittelgroß" ensprechen jetzt die Merkmalswerte "handwarm" und "mittlere Leistung". Es stellt sich die Frage, wie aus der gemessenen Temperatur auf die einzustellende Leistung zu schließen ist. Offenbar ist derjenige P-Wert zu wählen, für welchen die Steuerregel am besten erfüllt ist. Diese intuitive Antwort setzt voraus, dass eine "Erfüllungsfunktion" existiert, deren Maximalwert gefunden werden muss. Wir vereinbaren: *Der Verlauf des Erfüllungsgrades einer Steuerregel in Abhängigkeit vom Messwert (Messwerttupel) und vom Stellwert wird* ***Erfüllungsfunktion*** genannt und mit φ bezeichnet. Sie könnte auch *Zugehörigkeitsfunktion* genannt und mit μ bezeichnet werden, denn es ist im Grunde gleichgültig, ob man

sagt, dass ein Punkt (T,P) mit dem Grade φ die Regel erfüllt, oder ob man sagt, dass der Punkt mit dem Grade μ zu der durch die Regel definierten unscharfen Klasse von "Steuersituationen" gehört, wobei eine Steuersituation einem Apfel im Sortierbeispiel entspricht und durch einen Messwert und einen Stellwert charakterisiert ist. Wir werden bezüglich Steuerregeln die Bezeichnungen *Erfüllungsgrad* und *Erfüllungsfunktion* vorziehen.

Das Fuzzifizierungproblem lautet damit: Wie ist für die Steuerregel (21.3) die Erfüllungsfunktion $\varphi(T,P)$ festzulegen? Offensichtlich ist es sinnvoll, den Erfüllungsgrad für ein Wertepaar (T,P) mit dem niedrigeren der beiden Zugehörigkeitsgrade μ_H und μ_M festzulegen, formal notiert

$$\varphi(T,P) = \min(\mu_H(T), \mu_M(P). \tag{21.12}$$

Damit ist die Fuzzifizierung der Steuerregel (21.3) vollzogen. Wir vereinbaren: *Wenn sämtliche Merkmalswerte und Verknüpfungen von Merkmalen einer linguistischen Beschreibung (Regel, Anweisung) fuzzifiziert sind, nennen wir die* ***Beschreibung*** *(Regel, Anweisung)* ***fuzzifiziert****. Wenn sämtliche linguistischen Regeln eines Klassifizierungs- oder Steuerungsproblems fuzzifiziert sind, nennen wir das* ***Problem fuzzifiziert****.*

Die Formeln (21.10) und (21.12) sind, abgesehen von den Merkmalsbezeichnern, identisch. Die Erfüllungsfunktion hat die Form der Pyramidenoberfläche von Bild 21.2. Die Steuerregel (21.3) wird also auf die gleiche Weise fuzzifiziert wie die Sortierregel (21.2). Der charakteristische Unterschied zwischen dem Sortieren und dem Steuern tritt erst bei der Anwendung der fuzzifizierten Regel zutage. Beim Sortieren wird eine Klasse, d.h. ein Merkmalswertebereich bestimmt, beim Steuern muss ein exakter Stellwert bestimmt werden. Es erhebt sich die Frage, wie aus der gemessenen Temperatur T auf die einzustellende Leistung P zu schließen ist. "Offenbar ist derjenige P-Wert zu wählen, für welchen die Steuerregel am besten erfüllt ist." So hatten wir diese Frage weiter oben beantwortet. Jetzt können wir antworten: Es ist derjenige P-Wert zu wählen, für welchen $\varphi(T,P)$ maximal wird.

Es ist unschwer zu erkennen, das es ein ganzes Intervall von P-Werten gibt, in welchem φ einen größten Wert annimmt. Dieses Intervall lässt sich als obere Seite der trapezförmigen Schnittlinie der (y,z)-Ebene bei $x{=}P$ mit der Pyramidenoberflächen konstruktiv bestimmen. Das Trapez ist in Bild 21.2 senkrecht schraffiert. Als einzustellender P-Wert kann beispielsweise der Mittelpunkt der oberen Seite des Trapezes oder der P-Wert des Schwerpunkts des Trapezes d.h. der Mittelwert der mit φ gewichteten P-Werte der Basis des Trapezes gewählt werden. Das Auswählen eines exakten Wertes eines linguistischen Merkmals wird **Defuzzifizieren** genannt.

Abschließend verfeinern wir die Steuerregel (21.3) und ersetzen sie in der gleichen Weise wie wir oben die Sortierregel (21.2) durch eine Entscheidungstabelle mit 49 Regeln. Die Erfüllungfunktion stellt ebenso wie die Zugehörigkeitsfunktion beim Sortieren eine stilisierte Hügellandschaft dar, die durch die einhüllende Oberfläche von 49 sich zum Teil durchdringenden Pyramiden gebildet wird. Wenn der Stellwert

zu einem Messwert gesucht ist, in dem sich zwei Pyramiden durchdringen, auf den also zwei Steuerregeln anwendbar sind, ergeben sich zwei Trapeze. Für das Defuzzifizieren bietet sich die gewichtete Mittelung (Schwerpunktberechnung) an.

Wir haben der Anschaulichkeit halber unsere Überlegungen auf ein- und zweidimensionale Zugehörigkeits- und Erfüllungsfunktionen eingeschränkt. Sie können auf drei und mehr Dimensionen erweitert werden. Die geometrische Darstellbarkeit geht dann zwar verloren, doch an den Fuzzifizierungsvorschriften ändert sich nichts Wesentliches. Damit schließen wir die kurze Einführung in den Fuzzy-Kalkül ab. Näheres findet der Leser in der Literatur.[8] In [Keller 00] sind eine Reihe informativer Anwendungsbeispiele enthalten. Nicht selten liest man in Werbetexten den Hinweis "mit Fuzzy Logic", wenn Geräte mit Steuerautomatik angeboten werden wie z.B. Fotoapparate oder Waschmaschinen.

Die Bedeutung und die Breite der Einsatzmöglichkeiten des Fuzzy-Kalküls wird der Leser erahnen, wenn er sich an folgenden Satz aus Kap.5.5 [5.18] erinnert: *Sprachliches Modellieren ist das Hantieren mit Denkobjekten bzw. mit Datenobjekten und damit ein Operieren mit Merkmalswerten*. Wir fügen hinzu: Wenn das Modellieren auf Computer-IV (Informationsverarbeitung durch den Computer) beruht, müssen die Merkmalswerte scharf sein, wenn es auf Human-IV beruht, sind sie i.Allg. unscharf. Wir hatten gesehen, wie das Operieren mit unscharfen Merkmalswerten durch Fuzzifizierung dem Computer zugänglich gemacht werden kann. Es fragt sich, wieweit die Fuzzifizierung möglich ist. Folgende Überlegung soll eine plausible Antwort geben.

Wie wir gesehen haben, lassen sich disjunktive Verknüpfungen unscharfer Prädikate durch die Maximum-Funktion und konjunktive Verknüpfungen durch die Minimum-Funktion (angewandt jeweils auf die betreffenden Zugehörigkeitsfunktionen) fuzzifizieren. Um einen vollständigen Satz "unscharfer boolescher Funktionen" zu erhalten muss noch die Fuzzifizierung der Negation festgelegt werden. Dafür ist offensichtlich die Funktion $1\text{-}\mu$ geeignet. Sie wird Komplement-Funktion genannt. Mit dem so definierten vollständigen Satz unscharfer Operationen lässt sich das unscharfe Äquivalent des booleschen Kalküls und - nach den Kalküläquivalenzsatz - jedes Kalküls aufbauen. Mit anderen Worten: *Jedes durch Regeln festgelegte Operieren mit* ***unscharfen Werten*** *lässt sich "****schärfen****" (fuzzifizieren) und dem Computer zugänglich machen*. Ein Blick in die Literatur lässt erkennen, wie breit das Angebot an praktischen Fuzzifizierungsmethoden bereits ist, und es erweitert sich laufend.

Damit ist noch nicht die vollständige Kalkülisierbarkeit des sprachlichen Modellierens gewährleistet. Denn dieses beginnt mit dem Erkennen (Begreifen, Herausbilden entsprechender Ideme) von Merkmalsausprägungen (unscharfen Merkmalswerten) wie z.B. rot, warm, weich, mit deren Hilfe Objekte erkannt (begriffen, aus dem

8 Siehe u.a. in [Grauel 95], [Nauck 96], [Keller 00], [Stöcker 95].

Strom der Sinneseindrücke herausgehoben) werden (siehe [5.17]). Dieser erste Schritt, das (gehirninterne) Herausbilden unscharfer Merkmalswerte geht jeder Informationsverarbeitung voraus, auch der Computer-IV. Beispielsweise muss die Wissensbasis eines Expertensystems von einschlägigen Experten erstellt werden. Das ist ein langer Prozess, der mit der Herausbildung linguistischer Merkmalswerte beginnt, die sich für das sprachliche Modellieren des Diskursbereiches eignen.

Der zweite Schritt besteht im Kalkülisieren (im Übergang von der oberen zur mittleren Ebene in Bild 18.1). Er kann dadurch erfolgen, dass das zunächst unscharfe Modell in eine analytische Form überführt wird und ein Kalkül gefunden wird, der durch das Modell interpretiert wird. Er kann auch durch Fuzzifizierung des unscharfen Modells erfolgen. Insbesondere dann, wenn der Diskursbereich zu komplex ist, um exakte Merkmalswerte definieren zu können, bietet sich der Fuzzy-Kalkül an. Wenn es gelingt, die unscharfen Werte und Regeln zu fuzzifizieren, kann eine "unscharfe Wissensbasis" implementiert werden, und der Computer kann mit ihr arbeiten, mit anderen Worten, das weitere sprachliche Modellieren des Diskursbereiches ist "computerisiert".

Die Frage ist erlaubt, ob vielleicht auch der erste Schritt computerisiert werden kann. Aufgrund unseres Wissens über neuronale Netze kann die Frage "im Prinzip" (d.h. ohne Berücksichtigung technischer und zeitlicher Begrenzungen) bejaht werden. Wir wissen nämlich, dass neuronale Netze lernen können zu klassifizieren, ohne dass ihnen geeignete Merkmalswerte mitgeteilt werden. Das Lernen schließt die Herausbildung von Merkmalswerten auf subsymbolischer Ebene ein. Danach müssten neuronale Netze in der Lage sein, den ersten Schritt des sprachlichen Modellierens auszuführen, bzw. den Menschen bei diesem Schritt zu unterstützen.

Diese Möglichkeit legt die Idee nahe, einem Fuzzy-System mit einem neuronalen Netz zu einem hybriden **Neuro-Fuzzy-System** zusammenzuschalten. Das ist ohne große Schwierigkeiten möglich, da beide Ähnliches leisten, wenn auch auf verschiedenem Wege, nämlich das Klassifizieren nach unscharfen Merkmalswerten, die entweder vorgegeben oder zu finden sind. Eine charakteristische Gemeinsamkeit beider Methoden ist die Existenz von Schwellenwerten, was jedoch nicht überraschen sollte. Denn beide Methoden rechnen mit kontinuierlichen Werten. Klassifizieren auf der Grundlage kontinuierlicher Werte setzt aber - wie jede Informationsverarbeitung - Schwellenwerte voraussetzt (vergleiche [8.13]). Der entscheidende Unterschied zwischen beiden Methoden besteht darin, dass neuronale Netze auf der subsymbolischen Ebene arbeitet, während der Fuzzy-Kalkül auf symbolischer Ebene arbeitet.

In einem hybriden neuro-Fuzzy-System könnte das neuronale Netz die linguistischen Merkmale liefern und den bzw. die Experten beim Fuzzifizieren, d.h. beim Herausfinden der optimalen Zugehörigkeits- bzw. Erfüllungsfunktionen, unterstützen oder sogar ersetzen. Viele derartige Systeme sind vorgeschlagen und z.T. auch realisiert worden.[9] Die Entwicklungen stehen erst am Anfang. Wir können auf sie nicht eingehen, um nicht noch länger auf der subsymbolischen Ebene, also jenseits

der thematischen Grenze des Buches zu verweilen. Wir kehren zurück zur traditionellen KI, um das Facit unserer Bemühungen zu ziehen, der Bemühungen, das menschliche Denken auf der symbolischen Ebene zu modellieren.

21.4 Offene Fragen und die Komplexität des Denkens

Denken ist ein überaus komplexer Prozess. Das gilt sowohl für die subsymbolische (neurophysiologische) als auch für die symbolische (psychologische) Ebene. Denn auch aus psychologischer Sicht ist Denken ein vielgliedriger und vielschichtiger Prozess und insofern strukturell komplex. Außerdem kann das Denken Sprünge machen, sodass man in übertragenem Sinne auch von nichtlinearer Komplexität sprechen kann. Die "psychologische Komplexität" ist ebenso wie die neurophysiologische nicht Gegenstand des Buches. Dagegen liegt die Berechnungskomplexität der Simulation des Denken auf der symbolischen Ebene, also die Berechnungskomplexität der traditionellen KI, durchaus im Rahmen des Buches. Mit ihrer Untersuchung wollen wir unsere Betrachtungen zur künstlichen Intelligenz abschließen, bevor sie im folgenden Kapitel resümiert werden.

Um Angaben über die Berechnungskomplexität der KI machen zu können, müssen wir Algorithmen für die verschiedenen Ausprägungen menschlicher Intelligenz entwickeln. Diese Aufgabe werden wir dadurch lösen, dass wir die Schachalgorithmen aus Kap.17.3 verallgemeinern. Wir verbinden unser Vorhaben mit der Beantwortung einiger Fragen, die in Teil 3 offen geblieben waren. Damit ist der gedankliche Anschluss an frühere Überlegungen hergestellt und gleichzeitig eine repräsentative Auswahl von Problemen getroffen, die wir uns nun noch einmal hinsichtlich ihrer Komplexität genauer ansehen wollen. Dabei werden wir sowohl die unscharfen Begriffe "Komplex" und "strukturelle Komplexität" als auch den scharfen Begriff der Berechnungskomplexität verwenden. In der folgenden Liste sind die besonders schwierigen Fragen zusammengestellt, auf die wir beim Versuch, menschliches Denken zu simulieren, gestoßen sind.

1. Worin unterscheidet sich das Denken des Menschen vom "Denken" des Computers?
2. Lässt sich Wiedererkennen simulieren?
3. Lässt sich Assoziation simulieren?
4. Lässt sich Intuition simulieren?
5. Gibt es nichtreduzible Intuition?
6. Lässt sich Rätselraten simulieren?
7. Lässt sich das Erfinden von Regeln simulieren?
8. Lässt sich das Gewinnen von Erkenntnis simulieren?

---

9 Siehe z.B. [Grauel 95], [Nauck 96], [Keller 00].

Alle aufgeführten Fragen betreffen die Simulierbarkeit des menschlichen Denkens, allgemeiner die Simulierbarkeit der Fähigkeit des Menschen zum aktiven sprachlichen Modellieren, mit anderen Worten, sie betreffen die Möglichkeiten der künstlichen Intelligenz. In der Redeweise des Kapitels 21.1 kann diese Grundfrage auch folgendermaßen formuliert werden: *Wieweit ist der* ***Komplex Denken*** *(die Komplexität des Denkens)* ***durchschaubar*** *und wieweit ist er* ***beherrschbar****?* Die Fragen 2 bis 8 stellen spezielle Aspekte der ersten Frage dar. Wir werden die Fragen der Reihe nach behandeln und beginnen mit der ersten:

Worin unterscheidet sich das Denken des Menschen vom “Denken” des Computers?

Einen wichtigen, vielleicht sogar den entscheidenden Unterschied zwischen dem Denken des Menschen und dem des Computers haben wir in Kap.17.3 erkannt, allerdings bezogen auf einen ganz speziellen Problemlösungsprozess, auf das Schachspielen. Der Unterschied besteht darin, dass der Mensch im Gegensatz zum Computer eine Spielsituation sozusagen auf einen Blick erfasst. Diesen Unterschied hatten wir verallgemeinert und gesagt: Der Mensch kann anschaulich und in globalen Zusammenhängen, er kann *gestalthaft* denken. Der Computer (genauer der Prozessorcomputer) dagegen kann nur in Folgen von Computerworten denken [17.12].

In Kap.18.1 hat uns ein Vergleich von Maschinensprachen und natürlichen Sprachen zu einer ganz ähnlichen Aussage geführt: Der Mensch ist dem Computer (genauer dem Prozessorrechner) darin überlegen, dass er nicht nur satzorientiert (algorithmisch), sondern auch netzorientiert (dazu gehört bildhaft) denken kann [18.1].

In dem Wort “*gestalthaft*” kommen die Bedeutungen der vier Wörter (Wortverbindungen) “anschaulich”, “in globalen Zusammenhängen”, “bildhaft” und “netzorientiert” sehr sinnfällig zum Ausdruck. Die beiden dem *Sehen* entlehnten Wörter (anschaulich und bildhaft) haben zwar eine spezifischere Bedeutung, doch liegt die Annahme nahe, dass alle vier Wörter Charakteristiken des Denkens artikulieren, denen ein einheitliches Strukturprinzip des Trägers zugrunde liegt, nämlich die Netzstruktur des Gehirns.

Es wäre verlockend diesen Gedanken weiterzuspinnen. Das würde zu einer neuen Sicht auf unser Problem führen. Wir hätten nicht nach der Komplexität *sprachlicher Modelle* zu fragen, sondern nach der Komplexität ihres *materiellen Trägers*, des Gehirns, also nicht nach logischer, sondern nach physischer Komplexität. Das wäre ganz im Sinne unserer Informatikdefinition als Lehre vom *aktiven* sprachlichen Modellieren, das den Träger (die Hardware) einschließt. Doch würde es den Rahmen des Buches Sprengen. Denken, Assoziation, Intuition und Erkenntnisgewinnung sind *Phänomene*, die aus den komplizierten Prozessen *emergieren*, die in dem natürlichen neuronalen Netz unseres Gehirns ablaufen. Sie zeichnen sich sowohl durch strukturelle als auch durch nichtlineare Komplexität aus.

So allgemein, wie die erste Frage gestellt ist, so allgemein haben wir sie beantwortet. Die Antworten auf alle weiteren Fragen bilden in ihrer Gesamtheit eine detaillierte Antwort auf die erste Frage. Bei der Behandlung der Fragen werden wir wiederholt den scheinbaren Umweg über das Schachspielen gehen. Doch ist dies kein Umweg, sondern ein sehr direkter und gleichzeitig anschaulicher Weg. Es sei daran erinnert, dass wir das "Schachverhalten" (das Verhalten eines Schachspielers) als stilisiertes Alltagsverhalten" aufgefasst haben. Die schachspezifischen Beispiele lassen sich durchweg auf den Alltag übertragen, wobei die Schachintelligenz zur Alltagsintelligenz, zum gesunden Menschenverstand wird. Die Vorstellung, Schach sei stilisiertes Leben und Schachintelligenz kalkülisierte Alltagsintelligenz, wird sich als tragfähig und hilfreich erweisen.

Wenn wir nun versuchen, unsere schachintelligenten Algorithmen aus Kap.17.3 zu verallgemeinern, betreten wir wieder das Feld spekulativer Überlegungen. Doch werden wir uns stets Rechenschaft darüber ablegen, was realisierbar ist und was nicht. Auf diese Weise werden wir unsere Frageliste abarbeiten. Um die einzelnen Fragen zu beantworten, müssen wir jedes mal prüfen, ob bzw. wo der Anwendung der Algorithmen durch die Komplexität der Probleme praktische Grenzen gesetzt werden und ob die Verallgemeinerung auf "Alltagsintelligenz" Einfluss auf die Komplexität und damit eventuell auf die Simulierbarkeit hat. Dabei werden wir uns i.Allg. mit Plausiblitätsbetrachtungen begnügen. Wir beginnen mit dem Erfassen einer Schachstellung "auf einen Blick", d.h. ohne längere Analyse. Die Fähigkeit ist ein Spezialfall des *Wiedererkennens.* Damit sind wir bei der zweiten Frage (als Bestandteil der ersten):

Lässt sich Wiedererkennen simulieren?

Die Frage hat offensichtlich etwas mit *bildhaftem* Denken zu tun. Doch interessiert uns im Augenblick die spezielle Antwort aus Kap.17.3 [17.7]. Dort war die Frage im Zusammenhang mit der Verwendung von Fallwissen aufgetaucht. Schachstellungen mussten wiedererkannt werden. Das war unschwer zu realisieren, sodass in diesem speziellen Fall die Frage bejaht werden konnte, allerdings nur im Prinzip, d.h. unter der Voraussetzung ausreichender Rechenzeit. Die Lösung bestand darin, dass die aktuelle Stellung mit abgespeicherten Stellungen Feld für Feld verglichen wurde.

Die Methode lässt sich auf beliebige Bilder erweitern. Um ein aktuelles (zu erkennendes) Bild mit abgespeicherten Bildern vergleichen zu können, überzieht man zweckmäßigerweise sämtliche Bilder mit einem Gitternetz und vergleicht sie Feld für Feld miteinander. Während beim Schach die Besetzung der Felder durch Figuren verglichen wurde, müssen im Falle von Bildern Schwärzungsgrade oder Farben verglichen werden. Die Größe der Felder (der Abstand der Gitterlinien) sind an die Feinheit der Darstellung anzupassen. Die minimale Größe ist ein einziges Pixel (digitaler Bildpunkt).

Es ist offensichtlich, dass ein derartiges Vorgehen praktisch unbrauchbar ist. Um beispielsweise irgendeine gedruckte oder gar handgeschriebene Acht zu erkennen,

müssten sämtliche Linienzüge (im Grenzfall sämtliche Pixelkonfigurationen), die eine Acht darstellen könnten, abgespeichert werden, was wegen der kombinatorischen Explosion unmöglich ist, selbst dann, wenn nur zwischen Acht und "Nicht-Acht" zu unterscheiden ist, also eine Dichotomie durchzuführen ist. Das Problem ist zu komplex. Bezüglich der Anzahl der Pixel als Maß für die Problemgröße es ist von kombinatorischer, d.h supraexponentieller Komplexität.

Hinsichtlich des Wiedererkennens von Schachstellungen hatten wir uns einen Ausweg einfallen lassen. Er bestand darin, dass der Computer die aktuelle Stellung nach bestimmten *Konfigurationen* (Ausschnitten der Gesamtstellung [17.8]) absucht. Die Konfigurationen verallgemeinern wir nun zu *Bildmerkmalen*. In Analogie zu Konfigurationen sind Bildmerkmale charakteristische Elemente des Gesamtbildes, beispielsweise die Kreuzung zweier Linien (wie z.B. bei der Acht) oder ein in sich geschlossener Linienzug (wie bei der Null).

Man kann sich nun einen Algorithmus zur Erkennung von Zeichen, z.B. der zehn Ziffern, ausdenken, der ähnlich arbeitet, wie der in Kap.17.3 [17.9] angedeutete Bewertungsalgorithmus für Stellungen. Zu diesem Zwecke müssen Bildmerkmale festgelegt werden, mit deren Hilfe sich die Ziffern erkennen (voneinander unterscheiden) lassen. Für die Acht könnten das die Merkmale "genau zwei in sich geschlossene Linien" und "genau ein Kreuzungspunkt" sein. Diese Merkmale reichen i.Allg. aus, um gedruckte Achten von allen anderen gedruckten Ziffern zu unterscheiden. Sie werden kaum ausreichen, um beliebige handgeschriebene Achten zu erkennen.

Die Schwierigkeiten des Vorgehens liegen auf der Hand. Sie betreffen nicht nur das Festlegen einer geeigneten Merkmalsmenge, sondern auch den Entwurf eines Leseprogramms, das den Computer befähigt, die Merkmale in den dargebotenen Zeichen zu finden. Im Falle gedruckter Ziffern und Buchstaben sind die Schwierigkeiten relativ leicht überwindbar. Es existieren viele Erkennungssysteme, die nach dem skizzierten Prinzip arbeiten, z.B. für die Erkennung von Zahlen (etwa Postleitzahlen).

Die Schwierigkeiten können unüberwindbar werden, wenn die Objekte sehr komplex sind, wenn man etwa den Computer befähigen will, zu erkennen, ob auf einem Bild ein Laubbaum oder ein Nadelbaum dargestellt ist oder ob ein Passbild eine weibliche oder eine männliche Person zeigt. Zwar lassen sich auch in diesen Fällen charakteristische Merkmale angeben; eine zuverlässige Identifikation des Geschlechts einer beliebigen Person nach ihrem Passbild ist jedoch kaum zu erreichen. Das Erkennungsproblem ist von einer Komplexität, mit welcher auch der Mensch nicht immer fertig wird, i.Allg. aber doch besser als der Computer.

Die dargestellte Methode des Merkmalvergleichs zum Zwecke der Objekterkennung lässt sich noch weiter verallgemeinern. Tatsächlich ist sie auf beliebige Objekte, mit denen der Mensch oder der Computer hantiert, anwendbar. Mehr noch, es gibt keine andere Methode, um ein Objekt zu erkennen, zumindest nicht im Rahmen der traditionellen, symbolischen Informationsverarbeitung. Das folgt unmittelbar aus der Definition des Denkobjekts.

In Kap.5.5 [5.17] hatten wir uns klargemacht, dass Ausschnitte der beobachteten Welt dadurch im Denken Selbständigkeit erhalten und zu Denk*objekten* werden [5.18], das heißt, dass ihnen Merkmale (bzw. Merkmalswerte) zugeordnet werden, durch welche sie sich aus der Umgebung herausheben und herausgetrennt bzw. identifiziert (wiedererkannt) werden können. Das gilt nicht nur für Ideme (Denkobjekte), denen Realeme entsprechen, sondern für beliebige Ideme, also auch für abstrakte Begriffe.

Da das Operieren mit Objekten durch einen Computer das Operieren durch Menschen voraussetzt, folgt die Richtigkeit obiger Behauptung, die wir verallgemeinern und präzisieren: *Das Operieren mit Objekten ist ein Operieren mit* ***Merkmalen*** *bzw.* ***Merkmalswerten*** *der Objekte*. Dieser Satz gilt unabhängig davon, ob Menschen oder Computer mit Objekten operieren.

Die Objekterkennung mittels Merkmalsvergleich ist also eine im Prinzip universelle Methode. Die Grenzen ihrer technischen Anwendbarkeit sind uns inzwischen wohl bekannt; sie sind durch die Komplexität des Problems gegeben. Als Beispiel war die Komplexität des menschlichen Antlitzes erwähnt worden. Sie ist vom Computer (zur Zeit) nicht beherrschbar.

Als weiteres Beispiel kann der Begriff der Komplexität selber dienen. Man versuche, ihn mittels Merkmalen zu definieren, sodass auch der Computer mit ihm *inhaltlich* arbeiten kann. “Inhaltlich” bedeutet nicht, dass der Computer dem Wort eine externe Semantik zuordnet (dazu ist er prinzipiell nicht in der Lage, weil er kein Bewusstsein besitzt), sondern dass der Mensch einen Satz, den der Computer artikuliert (nicht nur redigiert) hat und der das Wort “Komplexität” enthält, sinnvoll interpretieren kann.

Das *Wiedererkennen* ist ein spezielles Operieren mit Merkmalen. In Kap.16.2 [16.4] [16.5] hatten wir festgestellt, dass das *Speichern und Wiederfinden* von Daten in einer Datenbank auf einem Operieren mit Merkmalen beruht. Auch das Wiederfinden im menschlichen Speicher, im Gedächtnis, also das *Erinnern*, ist ein Operieren mit Merkmalen und zwar, soweit es im Unterbewusstsein abläuft, ein Assoziieren [5.19] [7.2] [7.3]. Damit wird aus der Frage, ob Wiedererkennen simulierbar ist, die Frage:

Lässt sich Assoziation simulieren?

Ein Schachspieler, der beim Anblick einer aktuellen Stellung an eine frühere Stellung erinnert wird, die sich seinem Gedächtnis eingeprägt hat, ist sich keines Suchprozesses bewusst. Sein Gehirn “assoziiert” mit der aktuellen eine frühere Stellung. Demgegenüber muss der Computer in einer “Stellungsdatei” nach der aktuellen Stellung suchen, um sie zu “erkennen”. Wenn es auf diese Weise gelingt, das menschliche Verhalten zu simulieren, ist damit die Frage positiv beantwortet, obwohl unbekannt ist, was beim Assoziieren im Gehirn vor sich geht. Die Bejahung der Frage ist allerdings nur soweit gerechtfertigt, wie Assoziieren mit Erinnern oder Wiedererkennen gleichzusetzen ist. Wir hatten die Gleichsetzung in Kap.7.1 [7.3]

per Definition festgelegt. Dort hatten wir auch die Unschärfe der Begriffe *Assoziation* und *Intuition*, wie sie gewöhnlich verwendet werden, durch folgende Vereinbarung beseitigt: ***Assoziation** ist die Reproduktion (das **Auffinden** im Gedächtnis) bekannter Zuordnungen zwischen Objekten und Merkmalen ohne bewusstes Suchen.* ***Intuition** ist die Produktion (das **Erfinden**) neuer Zuordnungen zwischen Objekten und Merkmalen ohne bewusstes Deduzieren.* Artikulationen von Merkmalszuordnungen hatten wir *Prädikate* und das Artikulieren *sprachliches Modellieren* genannt. Die Artikulation neuer Modellaussagen aufgrund von Intuition schließt i.Allg. Assoziation ein.

Damit lautet unsere Antwort auf die Frage: ***Assoziation** ist simulierbar, wenn die unbewusste Suche in der Erfahrung durchschaubar und beherrschbart ist, d.h. wenn ihre Simulation mit polynomialer Komplexität möglich oder die Problemgröße sehr niedrig ist* [2]. Wenn diese Bedingungen erfüllt sind, ist die Komplexität des Assoziierens *beherrschbar*. Die Antwort wurde durch die vereinfachende, aber sinnvolle Definition des Assoziationsbegriffs ermöglicht. Hinsichtlich der Intuition trifft das leider nicht zu. Trotz unserer vereinfachenden Definition des Intuitionsbegriffs ist die Beantwortung der nächsten Frage schwieriger:

Wie weit ist Intuition simulierbar?

Bei der Beantwortung können wir uns auf zwei spezielle Fälle von Intuition stützen, die wir bereits genauer untersucht haben und bei denen es sich nicht um bloßes Erinnern oder Wiedererkennen, also nicht um reine Assoziation handelt. In Kap.16.3 [16.10] hatten wir die Intuition Kekulés dadurch kalkülisiert, dass wir sie auf eine *Erweiterung des Suchraumes*, in welchem Kekulé nach der Strukturformel des Benzolmoleküls suchte, zurückgeführt haben. In Kap. 17.3 [17.6] haben wir die Intuition eines Schachspielers dadurch kalkülisiert, dass wir sie auf interiorisierte Erfahrung, auf nichtbewusstes Wissen zurückgeführt haben.

Beide Fälle lassen sich verallgemeinern. Die auf Erfahrung beruhende Intuition des Schachspielers kann auf andere Entscheidungen und Reaktionen, denen unbewusste Erfahrung zugrunde liegt, verallgemeinert werden. Die Verallgemeinerbarkeit der Suchraumerweiterung im Benzolbeispiel wurde in Kap.16.3 [16.11] diskutiert und in diesem Zusammenhang vereinbart, immer dann von *reduzibler* (auf Deduktion reduzierbare) Intuition zu sprechen, wenn eine richtige Aussage *erfunden* wurde (vom Träger der intuitiven Intelligenz), obwohl sie hätte *abgeleitet* werden können. Diese Bezeichnung trifft offensichtlich auch auf die auf unbewusster Erfahrung beruhenden Intuition zu, denn ihr liegt nichtbewusstes Ableiten aus nichtbewusstem Wissen zugrunde.

In beiden Spezialfällen (Benzolring und Schach) handelt es sich also um *reduzible Intuition*. Um sie zu simulieren, müssen Algorithmen entwickelt werden, nach denen sich die Produkte reduzibler Intuition *berechnen* lassen. Das setzt voraus, dass es gelingt, das *Wissen*, *aus* welchem abgeleitet wird, und die *Regeln*, *nach* welchen abgeleitet wird, zu *erkennen*, d.h. aus dem Dunkel des Nichtbewussten ans Licht zu

ziehen. Voraussetzung der Simulierbarkeit ist also die *Durchschaubarkeit* der Komplexität des unbewussten Ableitungsprozesses (des unbewussten Dunkels der Gehirntätigkeit). Dabei muss die logische (“psychologische”) Komplexität durchschaubar, d.h. nachvollziehbar sein. Die physische (neurophysiologische) Komplexität kann im undurchschaubaren Dunkel verbleiben. Ferner müssen sich die Regelanwendungen in ein Programm überführen lassen, das in akzeptabler Zeit ausgeführt werden kann, m.a.W. die Komplexität muss *beherrschbar* sein.

Wir fassen die Ergebnisse der Analyse der beiden Spezialfälle zusammen: ***Reduzible Intuition** ist simulierbar, wenn die logische Komplexität der Prozesse, die der Intuition zugrunde liegen, durchschaubar und wenn die Simulation mit höchstens polynomialer Komplexität möglich oder die Problemgröße sehr niedrig ist*. Die Aussagekraft dieses Satzes ist gering. Der Satz enthält nichts Überraschendes und eigentlich nichts Neues. Allerdings haben unsere Überlegungen gezeigt, dass der Bereich der reduziblen Intuition mehr umfasst, als es zunächst scheinen könnte. Diese Einsicht provoziert eine speziellere Formulierung der ursprünglichen Frage:

Gibt es nichtreduzible Intuition?

Die Frage ist zu verneinen, wenn man die Feststellung aus Kap.7.1 [7.6] ernst nimmt, wonach *jede **Intuition** auf Ableiten beruht*, und zwar auf *nichtbewusstem **Ableiten** aus nichtbewusstem **Wissen***. Die Feststellung ergab sich zwingend, wenn von dem Wirken einer höheren Instanz abgesehen und Intelligenz als Produkt der Evolution angesehen wird, genauer gesagt, wenn davon ausgegangen wird, dass Intelligenz eine Eigenschaft des Nervensystems und speziell des Gehirns ist, dessen Struktur sich im Laufe der genetischen und der individuellen, intellektuellen Evolution entwickelt hat. Anders ausgedrückt, die Gehirnstruktur trägt die persönliche “Erfahrung” des Individuums, sie ist *materialisierte Erfahrung*. 6

Es ist zu bemerken, dass die Nutzung von Erfahrung stets das *Erkennen von Ähnlichkeiten* zwischen vergangenen (erfahrenen) und aktuellen Gegebenheiten impliziert. Auf diesen Aspekt wird in der dritten Bemerkung am Ende des Kapitels näher eingegangen. Ferner ist zu bemerken, dass der Mensch nicht nur aus persönlicher Erfahrung handelt und denkt, sondern auch aus Erfahrung der Gattung, die im Laufe der genetischen Evolution “gesammelt” worden ist und - ebenso wie die persönliche Erfahrung - in der Struktur des Gehirns ihren Niederschlag gefunden hat, wobei “genetische Erfahrung” nicht erworben, sondern ererbt wird. Genetische Erfahrung hatten wir in Kap.7.1 [7.5] mit *Instinkt* gleichgesetzt.

Wenn man davon ausgeht, dass es keine weiteren Quellen der Gehirnstruktur gibt als die Erfahrung der Gattung und des Individuums und wenn man anerkennt, dass das Gehirn der Träger der Intelligenz ist, kommt man zu dem Schluss: ***Intelligenz** und damit auch **Intuition** beruhen einzig und allein auf **Erfahrung***. Das scheint plausibel zu sein, wenn man “Erfahrung” in dem genannten verallgemeinerten Sinne versteht. Dennoch bleiben Zweifel zurück. Man braucht sich nur an unsere Überlegungen bezüglich des Lösens nichtmathematischer Probleme zu Beginn des Kapitels

16.1 [16.1] zu erinnern. Dort hatten wir eine Frage aufgeworfen, die immer noch unbeantwortet ist:

Lässt sich Rätselraten simulieren?

Um Rätselraten zu simulieren, müssen Regeln gefunden werden, nach denen man beim Raten vorgeht, d.h. es müssen die Strategien und Taktiken erkannt werden, nach denen der Mensch beim Raten bewusst oder unbewusst verfährt. Was sind das für Regeln? Denkt man an das Rätsel der Sphinx, sind relevante Regeln auf den ersten Blick nicht erkennbar. Zur Erweiterung unseres Denkhorizontes ziehen wir noch zwei weitere Rätsel hinzu: "*Gott sieht es nie, der Kaiser selten, doch alle Tage der Bauer Velten*" (Lösung: Seinesgleichen), und "*Hängt an der Wand, gibt jedem die Hand*" (Lösung: das Handtuch).

In jedem Falle verlangt Raten ein Operieren mit Merkmalen. Im ersten Rätsel ist die Lösung - ebenso wie im Rätsel der Sphinx - durch drei Merkmale, im letzten durch zwei Merkmale charakterisiert. (Das Wort "Merkmal" ist hier im weiten Sinne verwendet und schließt Merkmale, Merkmalswerte, Relationen und Relationswerte ein.) Allen drei Rätseln ist gemeinsam, dass sich Lösungsregeln nicht erkennen lassen, dass aber dennoch die Lösung eine Art Aha-Erlebnis auslöst, ein Gefühl, wie es einen beim Verstehen einer Pointe erfüllt, selbst dann, wenn kein nachvollziehbarer Weg, kein vernünftiger, "*regel*mäßiger" Zusammenhang zu existieren scheint. Das Handtuchrätsel ist nicht nur sinnlos (ohne sinnvolle, vernünftige Grundlage), sondern geradezu widersinnig und irreführend. Lösungsregeln sind nicht erkennbar.

Andrerseits deutet das Aha-Erlebnis darauf hin, dass die Lösung dennoch ihre innere Logik besitzt, dass sie irgendwie *folgerichtig* ist, dass ein Ableiten der Lösung "im Prinzip" möglich sein muss, dass es also Lösungsregeln geben *muss*. Zu dem gleichen Schluss führt die Tatsache, dass Rätsel nicht durch Würfeln, nicht mit Hilfe eines Zufallsgenerators erfunden werden, der zufällige Wörter oder Sätze generiert, sondern dass der Erfinder eines Rätsels sich von einer gewissen Logik leiten lässt.

Dieser Widerspruch zwischen Nichterkennbarkeit und Notwendigkeit einer impliziten Logik ist ein Indiz dafür, dass ein Mensch, der eine Pointe erfasst oder dem die Lösung eines Rätsels "einfällt", unbewusst die innere Logik erkannt hat und dass er, solange er die Lösung nicht gefunden hat, nach dieser Logik sucht, dass er - bewusst oder unbewusst - nach Regeln sucht, nach denen sich die Lösung *ableiten* lässt. Falls ihm seine Erfahrung (sein Wissen) geeignete Regeln zur Verfügung stellt, besteht das Raten aus dem *Auffinden* dieser Regeln und dem Ableiten der Lösung. Andernfalls muss er Regel *erfinden*. Damit sind wir bei der nächsten Frage:

Lässt sich das Erfinden von Regeln simulieren?

Auch bei der Beantwortung dieser Frage stehen wir nicht ganz hilflos da, sondern wir können auf dieselben Spezialfälle zurückgreifen, die zum Begriff der reduziblen Intuition geführt hatten, auf das Benzol- und das Schachbeispiel. In beiden Fällen wurden neue Regeln eingeführt. Im Benzolbeispiel handelte es sich um die Hinzu-

nahme einer neuen Regel, der Regel 3 [16.9], die ursprünglich nicht explizit vorlag, die sich aber aus dem Grundwissen ableiten ließ. Beim Schach handelte es sich um Regeln, die sich aus dem Fallwissen (aus punktuellen Erfahrungen) ableiten ließen [17.10].

Wenn das *Erfinden* neuer Regeln simuliert werden soll, muss es in ein *Ableiten* überführt werden, d.h. es müssen Regeln zum Ableiten von Regeln gefunden werden. Solche Regeln nennen wir **Metaregeln**. Worin bestehen die Metaregeln in den beiden Beispielen? Wir begnügen uns mit einer verbalen Charakterisierung.

Im Benzolbeispiel ist ein Algorithmus erforderlich, nach dem aus den 6 Axiomen [16.8] die Regel 3 [16.9] abgeleitet werden kann. Es macht keine prinzipiellen Schwierigkeiten, sich einen solchen Algorithmus auszudenken. In Kap. 16.3 war auf die Möglichkeit hingewiesen worden, den Computer Regeln mit Hilfe eines Zufallszahlengenerators "*erwürfeln*" zu lassen, deren Gültigkeit oder Ungültigkeit er anschließend beweisen müsste, z.B. nach der Methode der Rückwärtsverkettung.

Im Gegensatz zum Benzolbeispiel haben wir uns bei der Behandlung des Schachbeispiels bereits in Kap.17.3 [17.10] genauer überlegt, welche Metaregeln zur Erstellung von Regeln erforderlich sind. Die Regeln dienten der Extraktion von Regelwissen aus Fallwissen. Wir hatten sie *Erfahrungsregel* genannt zur Unterscheidung von den *Spielregeln*. Die Metaregeln hatten wir verbal als Vorschriften beschrieben, nach denen Konfigurationsklassen gebildet werden können. Da Klassen mittels Merkmalen definiert werden, schreiben die Metaregeln entsprechende Operationen mit Merkmalen vor. Durch Zuweisung von Namen zu den Klassen können *Begriffe* gebildet werden. So entstand z.B der Begriff der "gefahrenträchtigen Konfiguration". Das bedeutet, dass das Extrahieren von Regelwissen aus Fallwissen in die Kategorie der *begriffsbildenden Operationen* fällt (siehe Bild 5.4). Die zur Anwendung kommenden Merkmalsoperationen sind die gleichen wie diejenigen des Erkennens, genauer des Wiederkennens. Das legt den Gedanken nahe, dass vielleicht auch die andere Kategorie des Erkennens, die *Erkenntnisgewinnung*, auf ähnliche Weise dem Computer zugänglich gemacht werden kann. Wir wollen dem Gedanken nachgehen, stellen das Rätselraten zurück und fragen:

Lässt sich das Gewinnen von Erkenntnis simulieren?

"Wohl kaum!" mag die spontane Antwort lauten. Konkretisiert man die Frage auf die Gewinnung *physikalischer* Erkenntnis, erkennt man sogleich, dass es sich dabei im Grunde genommen um das Extrahieren von Regeln aus Fallwissen handelt. Derartige Regeln (die physikalischen Gesetze) sind oft von erstaunlicher Einfachheit und Eleganz. Es bleibt dahingestellt, ob die Einfachheit eine Eigenschaft der Natur oder ein Produkt der Intelligenz des Menschen ist, ein Produkt seiner hochentwikkelten Fähigkeit zum sprachlichen Modellieren. Letzteres wäre insofern plausibel, als Modelle einfach sein *müssen*, um mit ihnen leicht hantieren zu können. Um die Komplexität der Natur gedanklich in den Griff zu bekommen, ist der Mensch *gezwungen*, durch Abstraktion solche Begriffe zu erfinden, die geeignet sind, die

Komplexität überschaubar und durchschaubar zu machen, das heißt Begriffe, die eine *einfache* Beschreibung, ein *einfaches* sprachliches Modell ermöglichen.

Wir wollen das Finden von Regeln (das Gewinnen von Erkenntnis) an einem berühmten Beispiel verfolgen, an der Herleitung der keplerschen Gesetze aus den Messdaten Tycho de Brahes. Die Messdaten (die Himmelskoordinaten der Planeten in verschiedenen Zeitpunkten) bilden das Fallwissen, die Gesetze sind die extrahierten Regeln. Sie haben die Form analytischer *Rechen*regeln. Jeder wird zustimmen, dass die Aufstellung der Gesetze eine *geniale* Leistung war. Dennoch kann sie simuliert werden, allerdings nur soweit, wie es eine *mathematische* Leistung ist.

Die Aufgabe, das erste keplersche Gesetz (Planetenbahnen sind Ellipsen, in deren einem Brennpunkt sich die Sonne befindet) herzuleiten, erinnert an die Denksportaufgabe 2 aus Kap.16.3. Dort sollten 9 Punkten in der Ebene durch einen Streckenzug miteinander verbunden werden (siehe Bild 16.4). Jetzt sollen sehr viele Punkte im Raum durch eine Linie (die Bahn eines Planeten) miteinander verbunden werden. Die Voraussetzung für eine aussichtsreiche Suche nach einer Lösung ist offenbar eine radikale Beschränkung der Menge zugelassener Lösungen, d.h. die Einengung des Suchraumes. Ein Schritt in dieser Richtung ist die Annahme, dass die Planetenbahnen in sich geschlossen sind, sodass immer die gleiche Bahn durchlaufen wird. Das reicht aber noch nicht aus. Verlangt man außerdem, dass die gesuchten Linien ebene Kurven zweiter Ordnung sein sollen, so ist der Computer der Aufgabe durchaus gewachsen. Kreise und Ellipsen fallen z.B. in diese Klasse von Kurven.

Beim gegenwärtigen Stand der Mathematik und der Programmierungstechnik kann ein Programm, das dem Computer die erforderliche Intelligenz verleiht, aus mathematischen Standardberechnungen komponiert werden. Als Kepler lebte, bedurfte es dessen Intuition und Genialität, um einen adäquaten mathematischen Apparat zu entwickeln. Die entscheidende Intuition betraf jedoch nicht die Mathematik, sondern die Konkretisierung der Fragestellung, m.a.W. die Beschränkung des Suchraumes. (Bemerkung am Rande: Man könnte den Computer beauftragen, die beste Näherungskurve beliebiger Ordnung durch alle Messpunkte zu legen. Der Suchraum würde bei diesem Vorgehen *nicht* eingeschränkt, und die Lösung enthielte *keinen* Erkenntnisgewinn.)

Keplers Bemühungen waren zunächst von symmetrischen Vorstellungen geprägt. Er versuchte, die Planetenbahnen durch eine symmetrische Schalenstruktur des Universums in Form konzentrischer Polyeder zu erklären. Der Versuch blieb erfolglos. Kepler musste sich von dem gewohnten Denken in zentralsymmetrischen Strukturen befreien. Das gelang ihm. Freilich hätte er die “Vision” der elliptischen Planetenbahn kaum haben können, wenn nicht seit der Antike die Kegelschnitte ein beliebtes Objekt mathematischer Untersuchungen gewesen wären.

Keplers Erfahrung und Wissen waren zwar die Voraussetzung seiner Intuition, doch sind keine Regeln erkennbar, nach denen er vorgegangen sein könnte. Die Produkte seiner Intuition sind nicht ableitbar. Der Weg zu den keplerschen Gesetzen ist, wie der Weg zu jeder neuen, großen Erkenntnis, verschlungen und voll “unlogi-

scher" Sprünge, er ist zu komplex, um ihn auf lückenloses Schlussfolgern zurückführen, ("*reduzieren*") zu können.

Diese Aussage gilt nicht nur für große Entdeckungen und Erfindungen, sondern in gleichem Maße für Rätsel, womit wir auf die ursprüngliche Fragestellung **Lässt sich Rätselraten simulieren?** zurückkommen. Welcher Weg führt zur Lösung des Handtuchrätsels oder des Rätsels der Sphinx? Wir hatten uns darüber schon einige Gedanken gemacht. Es bedarf großer Phantasie und geistiger Beweglichkeit, um mit den drei, zeitlich aufeinanderfolgenden Merkmalen "4 Beine", "2 Beine", "3 Beine" dasjenige Objekt, welches diese Merkmale besitzt, den Menschen, zu assoziieren. Nichtsdestoweniger ist die natürliche Intelligenz durchaus in der Lage, diese Leistung zu vollbringen, kaum jedoch die künstliche Intelligenz.

Man überlege sich, über welches Wissen der Computer verfügen müsste und nach welchen Regeln er in diesem Wissen nach dem im Rätsel beschriebenen Objekt suchen müsste. Bei einiger Vertiefung in das Problem erkennt man, dass der erforderliche Aufwand sehr bald die Grenzen des praktisch Machbaren erreicht. Die Komplexität des Lösungsprozesses ist eventuell *durchschaubar*, jedoch nicht *beherrschbar*.

Noch hoffnungsloser steht es um das Handtuchrätsel. Ihm ist selbst die Findigkeit *menschlicher* Intelligenz nur in Ausnahmefällen gewachsen. Die Lösung kann vielleicht durch Assoziationen über das Wort "Hand" gefunden werden. Der Wortlaut des Rätsels könnte beispielsweise im Geiste das Bild eines Tuches mit einer Hand auftauchen lassen. Doch wird dieses Bild wohl erst durch die Lösung hervorgerufen. Dennoch ist sein Auftauchen auch ohne Lösung angesichts des bildhaften, phantasievollen Denkens des Menschen nicht völlig ausgeschlossen. Für das algorithmische Denken des Computers ist dieser Weg zur Lösung wohl kaum gangbar. Der Denkprozess beim Rätselraten ist undurchschaubar komplex. Er lässt sich nicht simulieren, ganz zu schweigen von der Berechnungskomplexität eines hypothetischen Algorithmus.

Wir beenden das Kapitel mit drei ergänzenden Bemerkungen.

7

Erste Bemerkung. Entgegen früherer Vermutungen ist künstliche Metaintelligenz möglich, zumindest in Ansätzen. Denn durch das Erfinden von Regeln und durch das Bilden von Begriffen steigert der Computer seine Fähigkeit zum sprachlichen Modellieren, also seine Intelligenz.

8

Zweite Bemerkung. Vielleicht hat sich der eine oder andere Leser darüber gewundert, dass oft von *deduktiver*, doch nie von *induktiver* Intelligenz die Rede war. Im philosophischen Sprachgebrauch wird ein Schluss vom Allgemeinen auf das Einzelne *deduktiv* und ein Schluss vom Einzelnen auf das Allgemeine *induktiv* genannt. Ganz in diesem Sinne haben wir das Ableiten spezieller Aussagen (z.B. der Strukturformel des Benzols) aus allgemeingültigen Regeln *Deduzieren* und die Fähigkeit dazu *deduktive Intelligenz* genannt. Analog hätten wir das *Ableiten* allgemeingültiger Regeln aus Fallwissen als *Induzieren* und die Fähigkeit dazu als *induktive Intelligenz* bezeichnen können. Wir haben davon abgesehen; vielmehr

haben wir auch das “Induzieren” als Ableiten (nach Metaregeln), also als Deduzieren aufgefasst.

Dritte Bemerkung. Das Rätsel mit der Lösung “Seinesgleichen” ist einem Vortrag von Walter Ehrenstein [Ehrenstein 56] entnommen. Die Gedankengänge des Vortrages ähneln in vielem denjenigen dieses Buches. Der Begriff der Intelligenz wird dort folgendermaßen definiert: “*Intelligenz ist dasjenige Zusammenspiel erblicher psychischer Faktoren, das den Gewinn neuer Ähnlichkeitserkenntnis ermöglicht*”. Demgegenüber hatten wir die Intelligenz als *Fähigkeit zum sprachliche Modellieren* definiert. Sprachliches Modellieren ist das Erfinden, Wiederfinden oder Ableiten von Aussagen über das zu modellierende Original. Ein Vergleich beider Definitionen ist interessant und wertvoll für das Verständnis des Intelligenzbegriffs.

Zunächst ist zu beachten, dass Ehrenstein den Intelligenzbegriff, ebenso wie wir, *nicht* unmittelbar mit der Fähigkeit zum Problemlösen, sondern mit einer anderen Fähigkeit in Verbindung bringt, welche die Voraussetzung der Fähigkeit zum Problemlösen ist. Syntaktisch-lexikalisch betrachtet geht die Definition Ehrensteins in unsere über, wenn die Wortverbindung “Zusammenspiel erblicher psychischer Faktoren” durch “Fähigkeit” und “Gewinn neuer Ähnlichkeitserkenntnis” durch “sprachliches Modellieren” substituiert wird.

Die zweite Substitution zeigt, dass Ehrenstein den Intelligenzbegriff auf die Erkenntnisgewinnung, genauer auf das Erkennen von Ähnlichkeiten einschränkt, und die erste Substitution zeigt, dass er Intelligenz nicht als Fähigkeit zum Erkenntnisgewinn, sondern als erbliche psychische Voraussetzung dieser Fähigkeit versteht. Die Herausstellung der Erblichkeit entspricht dem üblichen Gebrauch des Wortes “Intelligenz”. Allerdings schließt die psychische Natur und die Erblichkeit die Anwendung seiner Definition auf künstliche Intelligenz aus. Die genannten Unterschiede spielen eine untergeordnete Rolle, wenn es um ein tieferes inhaltliches Verständnis des Intelligenzbegriffs geht. In dieser Hinsicht ist die zweite Substitution die wesentlichere. Es lohnt sich, sie zu analysieren.

Das “Gewinnen neuer Ählichkeitserkenntnisse” würden wir als Erkennen neuer Merkmalsübereinstimmungen" bezeichnen. Voraussetzung dafür ist das Erkennen und Vergleichen von Merkmalen, also diejenigen Operationen, die jedem sprachlichen Modellieren zugrunde liegen. Alle simulierbaren intelligenten Leistungen beinhalten das *Erkennen* und *Vergleichen* von Merkmalen. Das geht aus den in Teil 3 und in diesem Kapitel angestellten Überlegungen und Beispielen hervor. Führt das Vergleichen zum Erkennen nicht vorhergesehener Übereinstimmungen, so liegt darin eine *Ähnlichkeitserkenntnis*. Das *Erfinden neuer* Merkmale ist Bestandteil der Intuition. Konkret ist es im Erfinden von Regeln, in der Erkenntnisgewinnung und oft auch im Rätselratens enthalten. *Ganz allgemein ist das **Erfinden von Merkmalen und Merkmalswerten** die **Grundoperation** des “Begreifens” der Welt.*

Der Vergleich hat gezeigt, dass die beiden Definitionen im Anliegen und in ihrem Kern übereinstimmen, nämlich in der Zurückführung intelligenten Verhaltens auf das Erfinden von und Operieren mit Merkmalen und Merkmalswerten.

22 Resümee und Perspektiven

Die Arbeit der Wissenschaft stellt sich uns also dar als ein unablässiges Ringen nach einem Ziel, das grundsätzlich niemals erreicht werden kann. Denn das Ziel ist metaphysischer Art, es liegt hinter jeglicher Erfahrung.

MAX PLANCK[1]

Welche Erkenntnis steht am Ende unseres Weges zur künstlichen Intelligenz? Was ist die “philosophische Quintessenz” unserer Überlegungen? Eine Durchsicht der Gedankengänge und der “klugen” Algorithmen des Teils 3 und des Kapitels 21.4 führt zu einem zwiespältigen Ergebnis. Ein optimistischer, vielleicht sogar euphorischer Vorkämpfer oder Anhänger der Rechentechnik kann zu der Überzeugung gelangen, dass bei gehörigem Nachdenken das intelligente Verhalten des Menschen vollständig durchschaubar und dass seine Simultion lediglich eine Frage der technischen Möglichkeiten und damit eine Frage der Zeit ist. Ein bekannter Vertreter dieses Standpunktes ist der amerikanische Wissenschaftler MARVIN MINSKY.[2]

Eine objektive und nüchterne Analyse dessen, was gegenwärtig tatsächlich möglich ist, führt jedoch zu einer erheblich bescheideneren, um nicht zu sagen pessimistischen Schlussfolgerung. Das Resümee unserer Bemühungen, die menschliche Intelligenz zu simulieren lautet:

Die Komplexität des menschlichen Denkens ist sehr selten durchschaubar und fast nie beherrschbar.

Dies ist die Antwort auf die letzte Frage unserer Fragenliste:

Ist der gesunde Menschenverstand simulierbar?

Es sei daran erinnert, dass wir *Denken* als nicht extern codiertes, also nur gedachtes, bewusstes oder unbewusstes sprachliches Modellieren und *Intelligenz* als Fähigkeit zum sprachlichen Modellieren definiert haben, gedankliches Modellieren eingeschlossen.

Das Resümee ist ernüchternd und stellt eine harte Herausforderung an die KI-Forschung dar. Es klingt vielleicht allzu pessimistisch und kann auf Widerspruch stoßen. Denn es ist nicht zu leugnen, dass sehr viel erreicht worden ist. Wir haben viele Ideen und Methoden kennen gelernt, die weite Bereiche des sprachlichen Modellierens der Welt durch den Menschen simulierbar machen.

---

1 Zitat aus dem Vortrag “Positivismus und reale Außenwelt”, abgedruckt in [Planck 91].

2 Siehe z.B. [Minsky 86].

Dass das Resümee dennoch gerechtfertigt ist, wird sofort deutlich, wenn man sich überlegt, dass das gegenwärtig *simulierbare* Denken verglichen mit dem nichtsimulierbaren Denken praktisch zu vernachlässigen ist. Die Zeit, die ein Mensch mit simulierbarem Denken verbringt, ist ein winziger Bruchteil der Zeit, die er überhaupt mit Denken verbringt, also praktisch mit seiner gesamten Lebenszeit, denn das Gehirn ist nicht nur am Tage, sondern auch in der Nacht, wenn es träumt, mit dem Modellieren der Welt beschäftigt. Doch erreicht Denken (Modellieren) nur selten mathematische Schärfe. Man könnte der Meinung sein, dass der Anteil des simulierbaren Denkens zunehmen wird, da die Technik uns alle zu immer exakterem Denken zwingt. Jedoch zeigt die folgende Überlegung, dass eher das Gegenteil zu erwarten ist.

Der Mensch erfand mechanische Maschinen und Motoren, um sich die physische Arbeit zu erleichtern. Er erfand Computer, um sich das Rechnen zu erleichtern. Er erfand die künstliche Intelligenz, um sich das Denken zu erleichtern. Voraussichtlich wird die simulierbare Denkarbeit gemittelt über die Zeit und über alle Menschen ebenso abnehmen, wie seine physische Arbeit bereits abgenommen hat. Danach ist zu erwarten, dass der nichtsimulierbare Anteil des Denkens eher zunimmt als abnimmt. Das würde bedeuten, dass der Mensch dank der Errungenschaften von Wissenschaft und Technik den Prozess des eigenen Denkens immer seltener durchschaut. Zu dieser paradoxen Schlussfolgerung wird im Schlusswort noch einiges zu sagen sein. Im Augenblick interessiert uns die Frage, wodurch der nichtsimulierbare "Rest" des Denkens charakterisiert ist und *warum* er (noch) nicht simulierbar ist?

Zu dieser Frage haben wir uns bereits in Kap.17.1 Gedanken gemacht im Zusammenhang mit Versuchen, den Computer an Alltagsgesprächen teilzunehmen zu lassen (man erinnere sich an den Turingtest und an das System "Eliza"). Wir hatten zwei Gründe für die unüberwindlichen Schwierigkeiten derartiger Versuche erkannt:

- Der Mensch weiß hinsichtlich konkreter Lebenssituationen mehr als irgendein Computer.
- Der Mensch kann sein Wissen effektiver nutzen als der Computer.

Dem könnte entgegengehalten werden, dass einem Computer mit CD-ROM-Laufwerk ein riesiges Wissen verfügbar gemacht werden kann und dass ein Computer mit Internetanschluss über "alles Wissen der Welt" verfügt. Der Einwand ist jedoch nicht stichhaltig, denn viel wichtiger als der Umfang ist der Charakter des verfügbaren Wissens. Der gesunde Menschenverstand benötigt im Alltag gar nicht so sehr *enzyklopädisches* Wissen als vielmehr *Alltags*wissen, *Allerwelts*wissen und *Situations*wissen. Zum Wissen eines Menschen gehört die gesamte Erfahrung seines Lebens, alles, was er erlebt und gelernt hat. Dazu gehören gegebenenfalls auch die Einzelheiten des Gesprächs, das er gerade führt. Dieses Wissen steht dem Menschen bei seinem Handeln und Sprechen ständig zur Verfügung, und er kann es sehr effektiv nutzen.

Der letzte Satz ist hinsichtlich seines Subjekts ("er") nicht scharf und infolgedessen irreführend. Wenn *ich* handle oder spreche, ist es in der Regel *nicht* das *bewusste*

Ich, das sein Wissen und seine Erinnerungen nutzt, denn Wissensnutzung ist, wie wiederholt festgestellt wurde, ein weitgehend unbewusster Prozess. Nicht *ich*, sondern *mein Gehirn* hantiert mit *meinem* Wissen und *meinen* Erinnerungen. Dabei scheint das Gehirn mit vielen Wissens- und Erinnerungselementen gleichzeitig (simultan) hantieren zu können, denn wie sollte sonst das anschauliche, gestalthafte, netzorientierte und assoziative Denken zustande kommen? Wie anders wäre man imstande, an einem vielseitigen und anspruchsvollen Gespräch, d.h. an einem Gespräch ohne semantische Verarmung [17.2] teilzunehmen. Wie anders wäre die emotionale Einstimmung (emotionale Konsensfindung [17.3]) auf den Gesprächspartner in einem "Augenblick" oder durch einen Blick in die Augen des anderen möglich? Zusammenfassend stellen wir fest:

Der ***Wissenserwerb*** *eines Menschen ist ein lebenslanger Lernprozess. Intelligentes menschliches Verhalten beruht auf der Fähigkeit zu* ***lernen*** *und Wissen* ***simultan*** *zu nutzen*.

Die Frage, ob der gesunde Menschenverstand und ob seine Alltagsintelligenz simuliert werden kann, führt damit zu der Frage:

Lassen sich Lernen und simultane Wissensnutzung simulieren?

Die Antwort lautet: Nur sehr begrenzt. Es sind zwar viele Verfahren des maschinellen Lernens und der maschinellen Wissensverarbeitung entwickelt worden, doch von der Effizienz, mit welcher der Mensch lernt und sein Wissen nutzt, ist die künstliche Intelligenz weit entfernt, zumindest die traditionelle KI. Hier liegt letzten Endes die Ursache für die niederschmetternde Bilanz, die obiges Resümee zieht.

Die Härte und die Resignation, die aus dem Resümee herausgelesen werden kann, wird dadurch gemildert, dass die Aussage sich auf den aktuellen Stand von Wissenschaft und Technik bezieht, dass es sich also nicht um eine *prinzipielle* Aussage handelt, die für alle Zukunft gültig bleiben muss (vgl. [17.13]). Dennoch ist sie schwerwiegend, und niemand kann vorhersagen, ob das *Fernziel* der Informatik je erreicht wird, ob es jemals gelingen wird, den gesunden Menschenverstand in allen seinen Erscheinungsformen vollständig zu simulieren, selbst bei Einbeziehung alternativer KI auf der Grundlage neuronaler Netze. Das ist keine pessimistische, sondern eine nüchterne Feststellung. Gegenteiligen Behauptungen fehlt die gesicherte Grundlage. Aber auch die Behauptung, dass der gesunde Menschenverstand *prinzipiell nicht* simulierbar ist, lässt sich nicht beweisen.

Da der weitere Weg von Wissenschaft und Technik durch keine Verbote eingeengt oder versperrt ist, abgesehen von den Gesetzen der Physik, stellt das Resümee eine Herausforderung an die KI-Forschung dar. Entsprechend intensiv wird nach Wegen gesucht, die Fähigkeiten des Computers zum Lernen und zur Wissensnutzung den Fähigkeiten des Menschen anzunähern. Verständlicherweise wird versucht, den eingeschlagenen Weg der *traditionellen KI*, d.h. der Softwareentwicklung weiterzugehen. Es werden neue Algorithmen entwickelt, die den Computer befähigen, Wissen zu erwerben, zu lernen und Wissen effektiver anzuwenden. Gleichzeitig kommen

Mehrprozessorcomputer zum Einsatz, um die Rechengeschwindigkeit zu erhöhen. All das bedeutet eine Steigerung simulierbarer Komplexität und eine Zunahme der Komplexität der simulierenden Software.

Einen anderen Weg geht die *alternative KI*. Sie versucht, die Fähigkeiten der natürlichen Intelligenz mit Hilfe neuronaler Netze zu modellieren bzw. zu simulieren. In Kap.9.4 hatten wir erkannt, dass neuronale Netze die Fähigkeit zum Lernen, zum Klassifizieren und zum Wiedererkennen besitzen. Obige Frage betrifft eben diese Fähigkeiten. Außerdem arbeiten neuronale Netze, ebenso wie Kombinationsschaltungen, parallel. Sie besitzen also diejenigen Eigenschaften, in denen die menschliche Intelligenz die *traditionelle KI*, also die Intelligenz des Prozessorrechners einschließlich seiner Software übertrifft. Das erzeugt neue Hoffnung, das Fernziel der Informatik zu erreichen. Vielleicht lässt sich künstliche Alltagsintelligenz mit Hilfe neuronaler Netze produzieren.

Wir werden den Entwicklungen in dieser Richtung nicht weiter verfolgen, da wir damit das eigentliche Thema des Buches verlassen würden. Vielmehr wollen wir auf Perspektiven hinweisen, die durch die mathematische Beherrschung nichtlinearer Komplexität eröffnet werden. Um sie zu erkennen vergegenwärtigen wir uns folgenden Umstand.

Fast alle Vorgänge, die wir um uns herum beobachten, sind durch Nichtlinearitäten und Irregularitäten gekennzeichnet. Das gilt für die unbelebte und insbesondere für die belebte Welt und für jede Art evolutionärer Prozesse. Durch die Entwicklung mathematischer Methoden für die Behandlung nichtlinearer Dynamik erweitert sich das Feld physikalischer Forschung auf viele Phänomene, die durch Nichtlinearitäten, durch sprunghafte Änderungen von Verhaltensweisen oder von Merkmalswerten, allgemein durch Irregularitäten gekennzeichnet sind. Dazu gehört auch der Sprung der Ausgangsspannung eines Flipflops oder Schwellenoperators oder der Übergang des Anregungszustandes eines künstlichen oder natürlichen neuronalen Netzes in einen anderen.

Derartige irregulären Prozesse hatten wir *nichtlinear komplex* genannt. Durch die Möglichkeit ihrer mathematischen Beschreibung öffnen sie sich den Untersuchungsmethoden der theoretischen Physik. Das bedeutet, dass viele Phänomene des Lebens der Physik zugänglich werden, insbesondere das sprachliche Modellieren der Welt durch den Menschen oder den Computer. Denn seine Voraussetzung, die "kausale Diskretisierung" durch Schwellenoperatoren, wird Gegenstand der Physik, sogar der klassischen Physik, da zur Erklärung von Sprüngen nicht auf Quantensprünge zurückgegriffen werden muss. Die mathematische Theorie (die vollständig kalkülisierte Modellierung) nichtlinearer Dynamik eröffnet damit einen Weg, der zur Reduktion der Informatik auf Physik führen könnte, ähnlich wie die mathematische Theorie atomarer Dynamik, die Quantenmechanik, zur Reduktion der Chemie auf Physik geführt hat. Wie bereits in Kap.21.1 bemerkt wurde, ist es also nicht verwunderlich, dass die Untersuchung nichtlinearer dynamischer Systeme im Zusammen-

hang mit der Frage nach der Entstehung und Verarbeitung von Information auf subsymbolischem Niveau zunehmend an Bedeutung gewinnt[3].

Diese Entwicklung illustriert noch einmal die fundamentale Bedeutung der Mathematik für den Fortschritt der exakten Naturwissenschaften und speziell der Informatik. Ihre Bedeutung für die KI-Forschung als Teilgebiet der Informatik (vgl. Bild 3.2) trat im gesamten Teil 3 zutage. Es sei noch einmal wiederholt: *Künstliche Intelligenz beruht auf Mathematik und deren Überführung in Software*. Nicht der Computer als reine Hardwarehierarchie, wie wir sie in Teil 2 konstruiert haben, sondern der Computer als Einheit von Hardware und Software, als Hardware-Software-Hierarchie (siehe Bild 19.6) ist zu intelligenten Leistungen fähig. Wir beenden das Resümee mit folgender

Schlussbemerkung.

Das begriffliche Fundament, das in Teil 1 und in Kapitel 9 gelegt wurde, ist breiter, als es für die Zielstellung des Buches erforderlich gewesen wäre, denn es erlaubt die Beschreibung nicht nur symbolischer, sondern auch subsymbolischer Verarbeitungsprozesse. Das war notwendig, um zeigen zu können, dass die USB-Methode der Vollständigkeitsforderung gerecht wird, also der Forderung, dass mit der USB-Methode sämtliche im Prinzip möglichen informationellen Operatoren mit statischer Codierung komponierbar sein müssen. Auf dieser Grundlage konnte der *Berechenbarkeits-Äquivalenzsatz*, d.h. die Äquivalenz von rekursiver und statischer Berechenbarkeit bewiesen werden.

Es liegt nahe, den in diesem Buch eingeführten Begriffs- und Methodenapparat zu benutzen, um die Informationsverarbeitung in künstlichen neuronalen Netzen zu behandeln und zu versuchen, auch die Fragen der biologischen Informationsverarbeitung anzugehen. Der symbolischen bzw. der subsymbolischen Methode entspricht hinsichtlich der Informationsverarbeitung durch den Menschen die *psychologisch-linguistische* bzw. die *neurophysiologische* Methode. Auf die linguistische Methode wurde in Teil 1 wiederholt Bezug genommen, allerdings weniger aus inhaltlichen Gründen, sondern mehr der besseren Verständlichkeit halber. Ziel der Bezugnahme war es, bei der Definition so grundlegender Begriffe der Informatik wie "Syntax" und "Semantik" von einer begrifflichen Grundlage auszugehen, die jedem verständlich ist. Überhaupt war der Wunsch nach Allgemeinverständlichkeit mitbestimmend bei der Ausformulierung jeder Idee, jeder Begriffserklärung und jedes Gedankenganges. Dafür musste eine gewisse Umständlichkeit mancher Passagen in Kauf genommen werden.

Die Einbeziehung psychologisch-linguistischer und neurophysiologischer Erkenntnisse ist eine Notwendigkeit, wenn die Informatik das von Max Planck gestellte

---

3 Siehe z.B. [Ebeling 91] und [Ebeling 98]. Eine relativ leicht verständliche Einführung in das Gebiet der nichtlinearen dynamischen Systeme findet der Leser in [Nicolis 87].

letzte Ziel einer jeden Wissenschaft erreichen soll, die *vollständige Durchführung der kausalen Betrachtungsweise* (siehe das Planck-Zitat [Einleitung.1]. Denn diese Forderung, angewandt auf die Wissenschaft vom aktiven sprachlichen Modellieren, bedeutet, dass sprachliches Modellieren auf die neurophysiologischen und letztlich auf die physikalischen Prozesse im Gehirn zurückgeführt wird. Von diesem Ziel sind wir weit entfernt. Insofern charakterisiert der Titel "Kausale Informatik" nicht den Inhalt des Buches, sondern das Ziel der Wissenschaft Informatik. Aber auch wenn die Reduktion des Denkens auf Physik gelingen sollte, ist die vollständige kausale Durchdringungnicht nicht erreicht. Sie ist nie zu erreichen. "Denn das Ziel ist metaphysischer Art, es liegt hinter jeglicher Erfahrung" (siehe das Planck-Zitat, das diesem Kapitel vorangestellt ist).

Schlusswort

Zur gesellschaftlichen Bedeutung der Informatik

Bei uns selbst dürfen wir an unbegrenzte Möglichkeiten, an die stärksten und seltsamsten schlummernden Kräfte, an jedes Wunder glauben, ohne je fürchten zu müssen, dass wir einmal mit dem Kausalgesetz in Konflikt geraten könnten.

MAX PLANCK[1]

Wie im Vorwort angekündigt wurde, verfolgt das Buch ein zweifaches Ziel. Zum einen will es durch Einführung eines geeigneten begrifflichen Apparates den Weg zu einem geschlossenen Gebäude der Informatik als Wissenschaft ebnen. Zum anderen will es seinen Lesern dasjenige Wissen in allgemein verständlicher Weise vermitteln, das nötig ist, um sich selber eine begründete Meinung über die Bedeutung und die Auswirkungen der Rechentechnik und der auf ihr basierenden modernen Kommunikationstechnik und ganz allgemein über die gesellschaftliche Relevanz der Informatik zu bilden. Ich meine aber, dass der Leser, nachdem er mir auf dem mühsamen Weg vom Bit zur künstlichen Intelligenz gefolgt ist, das Recht hat, die Meinung des Autors zu einigen Fragen bezüglich der gesellschaftlichen Bedeutung der Informatik, die in der Öffentlichkeit diskutiert werden, zu erfahren. Es handelt sich um ganz persönliche Ansichten, die sicher nicht jeder teilt.

Neue wissenschaftliche Erkenntnisse und neue technische Erfindungen sind insofern ambivalent, als die durch sie eröffneten neuen Möglichkeiten sowohl zu Hoffnungen, als auch zu Befürchtungen Anlass geben. Nicht selten stoßen sie gleichzeitig auf begeisterte Aufnahme und auf heftige Ablehnung. Das gilt auch für die Informatik. Die diesbezüglichen Diskussionen kreisen im Wesentlichen um folgende Fragen, auf die ich der Reihe nach eingehen werde.

1. Was bringt der Menschheit die Informationsgesellschaft?
2. Welchen Einfluss wird die Kommunikationstechnik auf die Entwicklung der menschlichen Gesellschaft haben?
3. Lässt sich die menschliche Intelligenz durch maschinelle Intelligenz substituieren?
4. Welche Auswirkungen einer solchen Substitution sind denkbar?
5. Lässt sich Denken auf Physik reduzieren?
6. Welche Auswirkungen hätte die Reduktion?

---

1 Zitat aus dem Vortrag “Kausalgesetz und Willensfreiheit”, abgedruckt in [Planck 90].

Die Fragen 5 und 6 und teilweise die Frage 1 liegen außerhalb der Thematik des Buches. Doch sind alle sechs Fragen miteinander verflochten. Demzufolge werden sie häufig im Zusammenhang diskutiert, und deshalb sollen sie alle in die folgenden Betrachtungen einbezogen werden.

Frage 1: Was bringt der Menschheit die Informationsgesellschaft?

Sieht man sich die Fragen 2 bis 6 genauer an, stellt man fest, dass sie alle beantwortet werden müssen, bevor Frage 1 beantwortet werden kann. Die Beziehungen der Fragen 2 bis 6 zu Frage 1 bleiben allerdings verschwommen, solange man nicht geklärt ist, was genau unter dem Wort *Informationsgesellschaft* zu verstehen ist. Im Allgemeinen wird darunter recht undifferenziert die "*kommende Gesellschaft*" verstanden, die unsere gegenwärtige vertraute Gesellschaft ablöst. Sie wird sich - so die Annahme - Schritt für Schritt herausbilden, unter anderem infolge des Eindringens der Rechentechnik in alle Bereiche des Lebens und infolge der auf Rechentechnik beruhenden modernen Kommunikationstechnik. Eben darum spricht man von *Informations*gesellschaft. Dabei ist allerdings völlig klar, dass andere Faktoren, wie z.B. die Dichte und Bewegung der Erdbevölkerung oder der Verzicht auf traditionelle Kriege zwischen Staaten die kommende Gesellschaft in den nächsten Jahrzehnten viel stärker prägen wird als primär die Rechen- und Kommunikationstechnik. Doch sind diese Faktoren nicht Gegenstand des Buches. Eine lebendige Beschreibung dessen, was uns in der Informationsgesellschaft infolge moderner Rechen- und Kommunikationstechnik erwartet, findet der Leser in dem Buch "Der Weg nach vorn. Die Zukunft der Informationsgesellschaft" [Gates 97]. Die Beschreibung ist naturgemäß optimistisch, denn der Autor, BILL GATES, ist als Software*erfinder* einer der technischen Pfad*finder* in die Zukunft.

Es liegt nahe, das Wort "Informationsgesellschaft" in dem genannten *technischen* Sinne zu verstehen, wenn man unter Informatik die *Lehre von der traditionellen Rechen**technik*** versteht, deren Kern der Prozessorcomputer bildet. Dann stellen die Fragen 2, 3 und 4 Konkretisierungen der Frage 1 unter dem Aspekt der *technischen* Entwicklung dar. Wenn man jedoch, wie in Kap.3 begründet wurde, unter Informatik die *Lehre vom aktiven sprachlichen Modellieren* versteht, dann geht Frage 1 bedeutend weiter und tiefer. Denn Gegenstand der Informatik ist nach dieser Definition sowohl das *technische* (künstliche) als auch das *biologische* (natürliche) und speziell das menschliche sprachliche Modellieren.

Die Informatik, wie sie in diesem Buche verstanden wird, ist also viel mehr als nur *Technik*. Sie ist gleichzeitig - und sogar primär - die Lehre davon, auf welche Weise die *Menschen* und überhaupt *Lebewesen* die Welt sprachlich (d.h. in codierter Form, bewusst oder unbewusst) modellieren. In diesem Sinne haben wir in Kap.3.1 zwischen *technischer* und *biologischer* Informatik unterschieden. Damit rückt die Informatik in die unmittelbare Nähe der Erkenntnistheorie, insbesondere der evolutionären Erkenntnistheorie, worauf wiederholt hingewiesen wurde (siehe die Diskussion vor [7.8] und die Bemerkung am Ende des Kapitels 8.2.5).

Für die technische und für die biologische Informatik lassen sich *Fernziele* formulieren, die zwar angestrebt, eventuell aber nicht erreicht werden können. Für die technische Informatik könnte als Fernziel die *Substitution* der natürlichen Intelligenz durch künstliche, die Substitution des Gehirns durch Technik angestrebt werden, ein Ziel, das sicherlich nicht jeder stellen würde. Für die biologische Informatik dagegen ist das Fernziel die naturwissenschaftliche *Erklärung* der Gehirnprozesse und damit des Denkens und des Bewusstseins, d.h. die Zurückführung oder "*Reduktion*" von Denken und Bewusstsein auf Physiologie und letztendlich auf Physik.

Ob die Fernziele erreichbar sind, wissen wir nicht. Wir können es nicht wissen. Aber ihre Erreichbarkeit ist eine vernünftige *Arbeitshypothese*. Natürlich kann ein engagierter Wissenschaftler von der Erreichbarkeit überzeugt sein. Doch handelt es sich um einen "*Glauben*", um einen *metaphysischen* Standpunkt, genauso wie die Überzeugung eines Physikers, die Welt sei durch eine formalisierte Theorie beschreibbar, *meta*physischer Natur ist. Die genannten Endziele liegen, wie das Ende des Weges der Erkenntnis überhaupt, in undurchdringlichem Nebel. Endgültige Antworten auf die gestellten Fragen sind also nicht zu erwarten.

Ich möchte zunächst zu den Fragen 2 und 4 Stellung nehmen. Auf Frage 3 (Lässt sich die menschliche Intelligenz durch maschinelle Intelligenz substituieren?) werde ich nicht eingehen, da ihr der gesamte Teil 3 gewidmet ist. Dort haben wir die Frage untersucht, ob menschliche Intelligenz *simuliert* werden kann. In denselben Grenzen, in denen diese Frage zu bejahen ist, ist auch Frage 3 zu bejahen. Denn simulierte Intelligenz kann ebenso angewendet werden wie ihr Original, die natürliche Intelligenz, d.h. man kann sie beim sprachlichen Modellieren und speziell beim Problemlösen anstelle der eigenen Intelligenz einsetzen, indem man den Computer als intelligentes Werkzeug benutzt, m.a.W. *natürliche Intelligenz kann durch das Produkt ihrer Simulation substituiert werden*. In welchem Umfang das möglich ist, wo die Grenzen der Simulierbarkeit liegen, ist allerdings eine offene Frage. Wir wissen es nicht.

Frage 2: Welchen Einfluss wird die Kommunikationstechnik auf die Entwicklung der menschlichen Gesellschaft haben?

Spezialisten der verschiedensten Fachgebiete sind bemüht, die Auswirkungen der modernen, rechnergestützten Kommunikationstechnik auf die menschliche Gesellschaft zu prognostizieren. Dabei hängen die Prognosen nicht nur vom Fachgebiet, sondern auch vom Weltbild des Prognostizierenden ab. Der Grundtenor der Prognosen reicht von Weltverbesserungsutopien bis zu Horrorvisionen. Ich habe nicht vor, den bereits gemachten Voraussagen meine eigenen hinzuzufügen, sondern verweise den Leser auf die umfangreiche Literatur [2] und beschränke mich auf zwei allgemeine Bemerkungen.

Frage 2 kann nicht zuverlässig beantwortet werden, zumindest nicht im Detail und nicht auf lange Sicht. Denn es gibt keine mathematische Theorie gesellschaftlicher

Prozesse, aus der sich die "Zukunft der Kommunikationsgesellschaft" berechnen ließe. Dennoch sind Voraussagen möglich, doch können sie, ähnlich wie Wetterprognosen, falsch sein. Selbst wenn eine mathematische Theorie bestünde, wären zuverlässige Voraussagen nicht möglich. Dafür ist das Problem zu komplex.

Die menschliche Gesellschaft ist ein dynamisches System von undurchschaubarer Komplexität. Das Verhalten bereits "ganz einfacher" nichtlinearer dynamischer Systeme ist nicht vorhersehbar, weil in ihnen kleine Ursachen große Wirkungen nach sich ziehen können und weil *chaotische* Sprünge in neue Zustände stattfinden können. Wie sollte da der Sprung der menschlichen Gesellschaft in einen neuen stabilen Zustand, "Informationsgesellschaft" genannt, vorausgesagt werden können, der durch die Kommunikationstechnik initiiert wird? (Man lege sich die Frage vor, wie sich die Schafherde aus Kap.19.2.3 verhalten würde, wenn der Hirte - in Analogie zu kommunikativen Verbindungen über große Entfernungen - weit voneinander entfernte Schafe mit Stricken verbinden würde.) Sicher ist nur, dass mit der Veränderung der möglichen Wechselwirkungen zwischen den Menschen (zwischen den Elementen des "Vielkörpersystems Menschheit") sich auch die Eigenschaften des Gesamtsystems verändern werden.

Die Menschheit wird nach einem schwierigen und gefährlichen Übergangsprozess ("Einschwingvorgang") - ich gehe davon aus, dass sie ihn überstehen wird - in einen neuen relativ stabilen Zustand übergehen, der durch neue kollektive Eigenschaften charakterisiert sein wird. Die neuen Eigenschaften können sich erheblich von denjenigen der gegenwärtigen Gesellschaft unterscheiden. Der Prozess der Herausbildung neuer, i.Allg. nicht vorhersehbarer kollektiver Eigenschaften hatten wir *Emergenz* genannt. Da er in der Regel unerwartet und schlagartig einsetzt (wie ein "Blitz einschlägt"), hat KONRAD LORENZ ihn **Fulguration** genannt.

Wie das Neue aussieht, das infolge weltumspannender Kommunikation entstehehen wird, ist unbekannt. Es könnte sehr positive Seiten haben. Wenn alle Menschen ihre sprachlichen Modelle der Welt (ihr Weltbild) *direkt* einander mitteilen könnten, wäre zu hoffen, dass ein weltweites gegenseitiges Verständnis und auf dieser Grundlage eine neue, stabile Gemeinschaft aller Menschen heranwächst. Doch ist der Wert derartiger zwar sehr schöner, aber auch sehr ferner Hoffnungen aus oben genannten Gründen recht fragwürdig. Ich will auf alle diesbezüglichen Überlegungen verzichten und in einer zweiten Bemerkung etwas über das Nächstliegende, über den Übergangsprozess sagen, der bereits begonnen hat.

Der Übergang eines nichtlinearen Systems in einen neuen Zustand kann bekanntlich "gesetzloses", anarchisches, sogenanntes *chaotisches* Verhalten zeigen. Das gilt insbesondere für ein so komplexes System wie die Menschheit. Es hat also kaum Sinn, nach den Charakteristiken der zukünftigen Gesellschaft zu fragen. Sinn hat es

---

2 Stellvertretend sei ein älteres und ein neueres Buch zu dieser Thematik genannt, [Masuda 81] und [Tapscott 96].

jedoch, sich zu überlegen, wie man den Übergangsprozess in den neuen Zustand so beeinflussen kann, dass er für den einzelnen erträglich und für die Menschheit nicht zur Katastrophe wird.

Die erste Etappe des Weges in die Informationsgesellschaft wird naturgemäß dadurch geprägt sein, dass die technische Entwicklung viel schneller voranschreitet als die Entwicklung entsprechender neuer *Wertvorstellungen*. Dass neue technische Möglichkeiten neue Werte hervorbringen, die zu erstreben und für die zu leben sich lohnt, ist offensichtlich; und ebenso offensichtlich ist, dass sich die Wertvorstellungen gegenüber den technischen Möglichkeiten zeitverzögert entwickeln und konsolidieren.

In einem relativ stabilen gesellschaftlichen Zustand müssen die Werte, denen die Einzelnen nachstreben und nach denen sie ihr Tun und Lassen ausrichten, so ausbalanciert sein, dass die Gesellschaft lebensfähig bleibt. Es muss ein tragfähiger *Interessenabgleich* zwischen den Akteuren stattfinden, wobei als Akteure nicht nur Einzelpersonen, sondern auch Personengruppen, Firmen, Institutionen und Organisationen auftreten, wodurch die Rolle der Politik ins Blickfeld kommt. Das individuelle Steuerorgan, das den Interessenabgleich zwischen dem Individuum und der Gesellschaft, allgemeiner zwischen dem Ich und dem Wir bewerkstelligt, nenne ich **Gewissen** und meine, dass dieses Wort auch umgangsprachlich in etwa diesem Sinne verwendet wird. Doch will ich (in etwas großzügiger Interpretation des Wortes) nicht unterscheiden, ob Gewissen angeboren, anerzogen oder lediglich die Folge von Angst vor Bloßstellung oder vor Strafe ist.

Das Zurückbleiben des Gewissens und der Wertevorstellungen hinter den technischen Möglichkeiten, das bei jeder Erweiterung menschlichen Handlungsspielraums zu beobachten ist, kann zur Destabilisierung der Gesellschaft führen. Intensität und Grundsätzlichkeit, mit der *Wertediskussionen* geführt werden, sind also durchaus gerechtfertigt. Man denke an die Diskussionen, die im Zusammenhang mit neuen Möglichkeiten geführt werden, die durch die Kernenergie oder die Gentechnik eröffnet werden.

In der öffentlichen Wertediskussion nimmt die Informatik mit ihren neuen Möglichkeiten an Gewicht zu. Das ist nicht verwunderlich. Denn trotz aller positiven Potenzen sind erhebliche Gefahren nicht zu übersehen. Eine von ihnen, die zu erwarten war, kündigt sich bereits an. Zu erwarten ist nämlich, dass eine nicht zu vernachlässigende Anzahl von Menschen sich die Segnungen der Informationstechnik auf Kosten anderer persönlich zunutze machen wird, solange dem nicht durch Regeln, Gesetze, Kontrollen und Strafen Einhalt geboten wird. Die Versuchung ist zu groß.

Man sagt, dass Papier geduldig ist. Aber die Tastatur und der Bildschirm eines Computers sind noch geduldiger. Jeder kann praktisch alles ins Netz geben (senden), was ihm beliebt, wahre Informationen, Lügen, Versprechungen, Drohungen, Beleidigungen oder irgendwelchen Unsinn. Der Versuchung, derartige Mitteilungen mit rein egoistischen Zielen zu verbreiten, sind “unsichere Kandidaten” angesichts der

neuen Kommunikationsmittel bedeutend stärker ausgesetzt als früher, wobei ein solcher Kanditat jeder der oben genannten Akteure sein kann. "Unsichere Akteure" sind der Versuchung umso mehr ausgesetzt, je vollständiger sie ihre Kontakte mit der Umwelt über das weltweite Datennetz abwickeln. Außerdem kann ein Empfänger den Wahrheitswert einer Nachricht aus dem Netz schwerer beurteilen als den eines persönlichen Gesprächs, Telefongespräche bedingt eingeschlossen. Die Gesetzgeber stehen vor keiner leichten Aufgabe.

Ich verzichte darauf, die Machenschaften der Unehrlichkeit auszumalen, die bis zum Wirksamwerden geeigneter Gesetze denkbar sind. Auch auf die Auswirkungen, die durch technische Mängel, durch menschliches Versagen oder mutwillig, z.B. durch Einschleusen von Computerviren, verursacht werden können, will ich nicht eingehen. Vielmehr werde ich im Weiteren davon ausgehen, dass die Technik und das von Gesetzen unterstützte Gewissen ausreichend funktionstüchtig sind. Das kann eine Utopie sein, aber eine notwendige, denn ohne sie ist keine kommende Gesellschaft zu bauen. Ohne sie kann sich kein relativ stabiler und gleichzeitig menschenwürdiger Zustand der Gesellschaft ausbilden.

Neben den Folgen der weltweiten Kommunikation und der damit zusammenhängenden sog. Globalisierung gibt es andere, nicht weniger schwerwiegende Gründe zu Befürchtungen. Sie betreffen die Auswirkungen, die, unabhängig von der Kommunikationstechnologie, eintreten können, wenn natürliche Intelligenz durch künstliche Intelligenz substituiert wird. Damit komme ich zu

Frage 4: Welche Auswirkungen einer solchen Substitution sind denkbar?

Die *positiven* Folgen der KI (oder, wie man es auch ausdrücken könnte, der Intelligenzverstärkung bzw. Intelligenzsubstitution durch den Computer) sind so offensichtlich, dass ich auf sie nicht einzugehen brauche, vielmehr frage ich:

Welche *negativen* Folgen kann es haben, wenn die Menschen sich der künstlichen Intelligenz wie eines Werkzeuges bedienen können?

Auf diese Frage werden die unterschiedlichsten Antworten gegeben. Es werden Erscheinungen prognostiziert, die ihrerseits sekundäre Folgen verschiedener primärer Folgen der KI sind. Die wohl am häufigsten genannten primären Folgen lassen sich in 4 Punkten zusammenfassen.

Punkt 1: Die Menschen benutzen ihre Köpfe immer seltener zum Rechnen, zum logischen Schlussfolgern und zum Abspeichern von Wissen.

Punkt 2: Die Menschen passen ihr Denken zunehmend an die Sprachen und Fähigkeiten der Computer an.

Punkt 3: Die Menschen gewöhnen sich daran, den Computer für eigenes Versagen verantwortlich zu machen.

Punkt 4: Die Einzigartigkeit des menschlichen Geistes gerät ins Zwielicht.

Über die sekundären Folgen wird viel geredet und gerätselt. Zuweilen wird als Folge jedes einzelnen der angeführten Punkte der Ruin der Menschheit vorausgesagt, der intellektuelle Ruin infolge intellektueller Unterforderung (Punkt 1), der psychi-

sche Ruin infolge psychischer Überforderung durch “unmenschliche” Anforderungen an das Gehirn, auf die es durch die Evolution nicht vorbereitet ist (Punkt 2), der Ruin der öffentliche Ordnung infolge des Verlustes an Verantwortungsbewusstsein (Punkt 3) und der sittliche Ruin infolge des Verlustes an Selbstwertgefühl (Punkt 4).

Ich teile derartige Meinungen und Ängste *nicht*. Die genannten negativen Vorhersagen zu den Punkten 1 bis 3 halte ich für voreilig. Sie unterschätzen m.E. die Fähigkeiten der Menschen (präziser einer ausreichenden Anzahl von Menschen), sich an die Technik und die Technik an sich anzupassen, zwei ganz unterschiedliche, aber gleich schwerwiegende Prozesse. Die Anpassung der Informationstechnik an den Menschen ist die vornehmste, weil menschlichste Aufgabe der Informatiker. Auf Punkt 4 will ich ausführlicher eingehen.

Angenommen, man ist imstande, die Funktionsweise des Gehirns als Träger des Denkens *vollständig* zu simulieren. Dann lässt sich alles, was gedacht wird, simulieren, auch die Entstehung der Begriffe, in denen gedacht wird, und auch die Herausbildung von Wertvorstellungen und schließlich auch die Vorstellung von der Freiheit des eigenen Willens und sogar die Idee Gottes (des Idems, das durch das Wort “Gott” codiert wird). Das würde *scheinbar* bedeuten, dass niemand für sein Handeln verantwortlich gemacht werden kann und dass man vor niemandem verantwortlich ist. Das aber wäre der Tod jeder Ethik und würde möglicherweise das Ende der Menschheit bedeuten. Zumindest würde es das Ende der mehr oder weniger kontinuierlichen kulturellen Evolution der letzten Jahrtausende bedeuten.

Daraus ziehen viele - bewusst oder unbewusst - den Schluss, “dass nicht sein kann, was nicht sein darf”, das heißt, dass das Denken nicht simulierbar sein kann, weil es nicht simulierbar sein darf. Hier scheinen mir die Wurzeln vieler Angriffe und gerade der vehementesten Angriffe gegen die KI zu liegen.

Ich halte die Sorge, die KI würde Moral und Ethik untergraben, zumindest für übertrieben. Bevor ich meine Ansicht begründe, möchte ich auf ein merkwürdiges Missverständnis hinweisen. Man kann zuweilen beobachten, dass sehr prinzipielle Angriffe gegen die KI unter dem Banner eines kämpferischen *physikalischen Antireduktionismus* vorgetragen werden. Derartige Angriffe beruhen m.E. auf einem Missverständnis, zu dessen Aufklärung ich folgende Terminologie einführe.

Ich nenne im Weiteren die Einstellung eines Menschen **reduktionistisch** und spreche von **Reduktionismus**, wenn der Betreffende die Erklärung der Phänomene des Denkens und des Bewusstsein durch die exakten Wissenschaften (die “Reduktion” auf die exakten Wissenschaften; dazu gehören u.a. Physik und Informatik) *im Prinzip* für möglich hält. Wenn die Reduktion auf *Physik* für möglich gehalten wird, spreche ich von **physikalischem Reduktionismus**. Die entgegengesetzte Einstellung nenne ich **antireduktionistisch** und spreche von **Antireduktionismus**. In dieser Redeweise ist derjenige, der Frage 5 bejaht, ein “physikalischer Reduktionist”, der sie verneint, ein “physikalischer Antireduktionist”.

Das merkwürdige Missverständnis, von dem die Rede ist, liegt in der Annahme, dass mit der Simulation des Denkens dieses auch *verstanden* ist im Sinne der Physik.

Simulation des Denkens ist aber höchstens ein *Nach*ahmen und kein *Vorher*sagen des Denkens eines Menschen (man erinnere sich an die Diskussion zur KI in Kap.7.1 [7.6] und zum Kalkülisierungsgrad im Zusammenhang mit Bild 18.1). Die Prognose, die KI werde Moral und Ethik zersetzen, ist nur dann begründet, wenn das Denken und Handeln exakt vorausgesagt, d.h. vorausberechenbar ist. Sie ist unbegründet, wenn es *nur* simulierbar ist. Denn niemand wird seiner Verantwortung für sein Denken und Handeln dadurch enthoben, dass dieses *nach*geahmt wird.

Das Banner des physikalischen Antireduktionismus ist also fehl am Platze, wenn gegen die *KI* zu Felde gezogen wird. Denn der Kampf gegen die KI richtet sich nicht gegen den *physikalischen*, sondern gegen einen "**algorithmischen**" **Reduktionismus**, d.h. gegen die Annahme, Denken sei auf Algorithmen *reduzierbar* und gewissermaßen "*algorithmisch erklärbar*". Das merkwürdige Missverständnis beruht also auf einer Verwechselung *algorithmischer* oder - noch schärfer - *softwaremäßiger* mit *physikalischer* Reduzierbarkeit.

Die Argumente des *physikalischen* Antireduktionismus haben hinsichtlich der Neurophysiologie Sinn, wenn diese sich das Ziel stellt, das menschliche Denken auf Neurophysiologie und damit auf Chemie und Physik zurückzuführen. Dieser Kampf ist viel älter als der gegen die KI. Er wird auf einem Felde ausgetragen, das außerhalb des Rahmens dieses Buches liegt. Dennoch will ich auf das Problem eingehen und damit gleichzeitig eine Klärung des Unterschiedes zwischen algorithmischer und physikalischer Reduzierbarkeit verbinden.

Frage 5: Lässt sich Denken auf Physik reduzieren?

Die Physiker sagen von einem Phänomen (einer Beobachtung oder einer Gesamtheit von Beobachtungen), dass es *erklärt* ist, wenn die Beobachtungsdaten (Messwerte) aus der Theorie ableitbar sind. Ich erinnere daran, dass eine Theorie ein interpretierter Kalkül ist. Die beobachteten Größen sind Interpretationen formaler Größen des Kalküls.

Hieraus folgt der Unterschied zwischen *algorithmischer Reduzierbarkeit* (Simulierbarkeit) und *physikalischer Reduzierbarkeit* (Erklärbarkeit) des Denkens. Zwar setzt beides die *Kalkülisierbarkeit* des Denkens voraus, doch werden unterschiedliche Forderungen an die Interpretationen der Kalküle gestellt. Die KI verlangt, dass die Interpretation derjenigen *externen* bzw. *formalen* Semantik entspricht, in welcher der Nutzer denkt. Die Physik verlangt, dass die Interpretation der *internen* Semantik entspricht, also den Prozessen, die im Träger des Denkens, im Gehirn, ablaufen. Die Größen des Kalküls einer physikalischen Theorie des Denkens müssen also physiologischen Messwerten der Gehirntätigkeit entsprechen (als solche interpretierbar sein). Außerdem wird verlangt, dass die Interpretation keinen anderen existierenden und als richtig anerkannten physikalischen Theorien widerspricht. In dem Maße, in dem dies gelingt, ist die Reduktion des Denken auf Physik gelungen.

Diejenigen Leser, die gefühlsmäßig die Reduzierbarkeit des Denkens auf Physik für unmöglich oder gar für verboten halten und darum ablehnen, seien daran erinnert,

dass die Reduktion der Chemie auf Physik als vollzogen angesehen werden kann und dass die Reduktion der untersten Stufen des Lebens auf Chemie und damit auf Physik im Prinzip gelungen ist[3], obwohl dies vor nicht allzu langer Zeit von vielen als unmöglich, vielleicht sogar als verboten angesehen wurde.

Ich will versuchen, meine eigene Meinung zur Hypothese des physikalischen Reduktionismus durch Gegenüberstellung mit der Position des Philosophen KARL POPPER verständlich zu machen. Ich werde meine Ansichten, ebenso wie Popper, an Hand des Bildes von den drei Welten darlegen und komme damit auf den Anfang des Buches, auf Bild 1.1 zurück. Popper hat seine diesbezüglichen Ansichten wiederholt geäußert, z.B. in dem Vortrag "Wissenschaftliche Reduktion und die essenzielle Unvollständigkeit der Wissenschaft" (1972/74) sowie in dem Vortrag "Bemerkungen eines Realisten über das Leib-Seele-Problem" (1972)[4].

Popper erkennt den Wert der reduktionistischen Hypothese als *Arbeitshypothese* durchaus an und räumt ein, dass sie sehr fruchtbar war und dass die Reduktion sogar teilweise gelungen ist. Seine grundsätzliche Ansicht formuliert er jedoch in dem Satz "*Als Philosophie ist der Reduktionismus gescheitert*". Demgegenüber halte ich die Hypothese, dass Denken und Bewusstsein auf Physik zurückführbar sind, nicht nur für eine fruchtbare, sondern auch für eine "richtige" Hypothese, d.h. ich erwarte, dass sie sich früher oder später bestätigen wird. Meine Erwartung gründet sich auf eine andere Hypothese, an deren Richtigkeit ich nicht zweifle.

Ich gehe von der Richtigkeit der Hypothese aus, dass Idemen, also *mentalen* Zuständen, *neuronale* Zustände entsprechen. Diese Hypothese nenne ich **Brükkenhypothese**, weil sie die Brücke zwischen Körper und Geist, zwischen Leib und Seele, zwischen Physiologie und Psychologie und zwischen Poppers Welt 1 und Welt 2 schlägt. An der experimentellen Bestätigung der Brückenhypothese wird nicht ohne Erfolg gearbeitet. Wenn es gelingt zu zeigen, dass jeder Gedanke und jede Vorstellung (jedes *Idem*) seine neurophysiologische Basis hat und dass Denken auf neurophysiologischen, letzten Endes also auf chemischen und physikalischen Prozessen beruht, würde das für mich bedeuten, dass Welt 2 Teil von Welt 1 ist.

Aus meiner Sicht wäre damit auch die Reduzierbarkeitshypothese bestätigt. Denn ich zweifele nicht daran, dass es den Menschen stets durch ausreichend langes Experimentieren und Nachdenken gelingt, eine objektive Beobachtung zu *erklären*, d.h. in die existierenden physikalischen Theorien einzubauen bzw. diese entsprechend zu erweitern. Manche Wissenschaftler erwarten, dass der Einbau einer Theorie des Denkens und des Bewusstseins eine ganz neue Pysik verlangt. In seinem Buch "Schatten des Geistes" [Penrose 95] begründet ROGER PENROSE diese Erwartung ausgehend vom GÖDEL'schen Unvollständigkeitssatz (vgl. Kap.6). Er macht auch konkrete Vorschläge, in welcher Richtung die Physik zu erweitern wäre.

---

3 Siehe z.B. [Eigen 93].

4 Beide Vorträge sind in [Popper 96] abgedruckt.

Popper begründet seinen Satz, dass der Reduktionismus philosophisch gescheitert ist, auf der Grundlage seiner Drei-Welten-Theorie. Kernpunkt seiner Begründung ist folgende Aussage: Die stoffliche Welt, die uns umgibt und zu der wir gehören (Welt 1 in Bild 1.1) ist *offen*. Der Begriff der **Offenheit** ist aus der Systemtheorie übernommen. Ein System, das Einwirkungen von außen unterliegt oder auf die Außenwelt einwirkt, wird **offen** genannt; andernfalls wird es **abgeschlossen** genannt.

Popper begründet die Offenheit von Welt 1 durch folgende Überlegung. Zunächst zeigt er, dass Welt 1 von Welt 3 beeinflusst wird und zwar über Welt 2. Welt 2 umfasst die Gesamtheit aller individuellen Vorstellungen, Gedanken und Überlegungen der Menschen; Welt 3 umfasst alle von der kulturellen Evolution hervorgebrachten geistigen Güter, die im Prinzip jedem zur Verfügung stehen und die sich jeder aneignen kann (siehe Bild 1.1). Die Artikulierungen dieser Güter, z.B. in Form von Büchern oder Bildern, gehören zu Welt 1.

Der Feststellung, dass Welt 3 über Welt 2 auf Welt 1 einwirkt, wird niemand widersprechen. Denn die Menschen verändern durch ihr Denken (Welt 2) und Handeln die Welt 1 unter Verwendung des angehäuften Wissens (Welt 3). Popper begründet die Offenheit von Welt 1 mit der Feststellung, dass Welt 3 *außerhalb* der Welten 1 und 2 existiert. Das ist in meinen Augen eine Hypothese, die nicht durch Beobachtungen *erzwungen* wird. Sie ist "fingiert", was dem newtonschen Prinzip "Hypotheses non fingo" widerspricht.

Ich schreibe der Welt 3 keine selbständige Existenz zu. Das "Weltbild" meines Verstandes ist "*monistisch*" in dem Sinne, dass es in ihm nur eine einzige Welt gibt, die Welt 1.[5] Dass Welt 2 in meinem Weltbild Teil von Welt 1 ist, ergibt sich aus meiner Überzeugung, dass die Brückenhypothese zutrifft. Es bleibt die Frage zu klären, warum das Gleiche auch für Welt 3 gilt. Das *aufgezeichnete* objektive Wissen (der unter 1. genannte Teil von Welt 3 in Bild 1.1, ihr "Realem-Teil") besteht aus physischen Objekten und gehört aus meiner Sicht zu Welt 1. Das *nichtaufgezeichnete* objektive Wissen (der unter 2. genannte Teil von Welt 3, ihr "Idem-Teil") umfasst Wissen in Form *objektivierter* Ideme [5.1]. Dieser Teil gehört also zu Welt 2, die alle Ideme umfasst, und damit gehört er zu Welt 1.

Abschließend zu Frage 5 möchte ich den Verdacht äußern, dass die Argumente, die ich zugunsten der Reduzierbarkeitshypothese vorgebracht habe, von ihren Geg-

---

5 Um falschen Schlussfolgerungen hinsichtlich meines Weltbildes zuvorzukommen, füge ich Folgendes hinzu. Das Weltbild meines Verstandes ist ein anderes als das meiner Seele. Das Weltbild meines Verstandes, der die drei Welten rational zu erfassen sucht, ist ein materialistisches. Das Weltbild meiner Seele, die versucht, die drei Welten zu erfühlen, um in ihnen und mit ihnen leben zu können, ist ein religiöses. Ich wage diese Formulierung, obwohl ich nicht erwarten darf, dass die Zeichenrealeme "Seele", "materialistisch" und "religiös" im Bewusstsein des Lesers genau diejenigen Ideme erscheinen lassen, die ich mit ihnen artikuliere. Beide Weltbilder hinterfragen sich ständig gegenseitig, aber sie leben miteinander. Ich empfinde sie nicht als Spaltung meines Bewusstseins in These und Antithese, sondern ihre Gemeinsamkeit als Synthese.

nern als unwesentlich abgetan werden, weil sie deren Argumente gar nicht tangieren. Die Argumente gerade der vehementesten Gegner betreffen nämlich oft nicht die Hypothese selber, sondern die Gefahren, die ihre Bestätigung nach sich ziehen würde. Damit komme ich zu der Frage, welche Auswirkungen zu erwarten sind, falls die Reduktion gelingt.

Frage 6: Welche Auswirkungen hätte die Reduktion?

Falls sich die Reduzierbarkeitshypothese bewahrheiten sollte, scheint für den freien Willen kein Platz zu sein. Damit wäre der Verantwortung für das eigene Handeln und folglich jeder Moral und Ethik der Boden entzogen. Die Menschen würden dasjenige Organ verlieren, das ein Zusammenleben der Menschen ermöglicht, das *Gewissen*. Damit wäre der Untergang der gegenwärtigen menschlichen Gesellschaft besiegelt.

Fast unbemerkt bin ich zu der pessimistischen Vorhersage zurückgekehrt, die oben im Zusammenhang mit der KI formuliert wurde, die aber als falsch verworfen wurde, weil sie auf der Verwechselung der *algorithmischen* mit der *physikalischen* Reduktion des Denkens beruhte. Doch jetzt ist von physikalischer Reduktion die Rede, sodass die pessimistische Vorhersage nicht ohne weiteres verworfen werden kann. Dennoch blicke ich optimistisch in die Zukunft. Im Vorwort habe ich meinem Glauben an die Fähigkeit der Menschen Ausdruck gegeben, derartiger Gefahren Herr zu werden. Ich werde meinen Optimismus begründen.

Der Mensch ist von Natur aus davon überzeugt, dass er seine Handlungen "aus freiem Willen" (introspektiv betrachtet) steuern kann, unabhängig davon ob er wacht oder schläft (träumt), und unabhängig davon, ob er an die Reduzierbarkeit des Denkens glaubt oder nicht, und schließlich auch unabhängig davon, ob die Steuerung seines Willens von außen objektiv verfolgt wird oder ob der Prozess der Willensbildung neurophysiologisch beobachtet und evtl. vorausgesagt wird.

Der freie Wille "funktioniert" unabhängig davon, ob man seinen Mechanismus kennt oder nicht. Mir scheint diese Unabhängigkeit nicht nur für die Überzeugung zu gelten, der Wille sei frei, sondern für jede geistige Tätigkeit, für alles Denken und Fühlen. Meine These lautet: *Das* ***Denken*** *und* ***Fühlen*** *eines Menschen hängt nicht davon ab, was er über das Denken und Fühlen weiß und was er darüber denkt*. Meine ganze Erfahrung beweist mir die Richtigkeit dieser These. Darum nenne ich sie das **Autonomieprinzip der geistigen Tätigkeit**. Die spezielle Anwendung des Prinzips auf die Vorstellung, man habe einen freien Willen, nenne ich das **Autonomieprinzip der Willensfreiheit**.

Eine sehr überzeugende Begründung der Unantastbarkeit der Willensfreiheit findet der Leser bei MAX PLANCK in dessen Vorträgen "Kausalgesetz und Willensfreiheit" (1923) und "Vom Wesen der Willensfreiheit" (1936).[6] Kern seiner Begrün-

6 Beide Vortäge sind in [Planck 1991] abgedruckt.

dung ist die Unmöglichkeit einer *objektiven* Analyse der Motive der *eigenen* Willensentscheidungen, ihrer kausalen Bedingtheit, im Moment der Entscheidung. Diese Selbstanalyse ist ein Beispiel für eine unlösbare Operand-Operator-Zirkularität (vgl. Kap.6.3 [6.5]).

Abschließend möchte ich einem Missverständnis vorbeugen, das durch die Brükkenhypothese entstehen könnte. Es wäre ein Irrtum anzunehmen, dass mit der Bestätigung der Brückenhypothese das Leib-Seele-Problem aus der Welt geschafft wäre. Das Problem lässt sich infolge des Autonomieprinzips der geistigen Tätigkeit nicht verdrängen. So unantastbar, so autonom wie die Willensfreiheit, d.h. das subjektive Wissen, dass man einen freien Willen besitzt, ebenso autonom ist auch die Seele, d.h. das subjektive Wissen, dass in einem etwas wohnt, das nicht der eigene Leib ist und das Seele genannt wird. Carl Gustav Jung schreibt in “Antwort auf Hiob”: “*Die Seele ist ein autonomer Faktor*” [Jung 53].

Also auch dann, wenn zweifelsfrei bestätigt sein wird, dass mentalen Zuständen neuronale Zustände entsprechen, selbst dann wird die Vorstellung lebendig und wirksam bleiben, dass Leib und Seele oder dass Körper und Geist verschiedenen Welten angehören. Und die Frage, wie beide aufeinander einwirken, wird nicht aufhören, die Menschen zu beschäftigen. Was für die Überzeugung, man besitze einen freien Willen oder man besitze eine Seele, gesagt wurde, gilt für jede Überzeugung, auch für den Glauben an Gott oder an Götter.

Es gibt außer dem Autonomieprinzip noch einen zweiten, rein wissenschaftlichen Grund dafür, dass selbst bei Bestätigung der Brückenhypothese Raum für Glauben und Religion bleibt. Die Hypothese nimmt nämlich lediglich eine *Entsprechung* zwischen mentalen Zuständen (Idemen) und neuronalen Zuständen an im Sinne einer Abbildung aus der Menge der neuronalen Zustände in die Menge der Ideme. Die Frage, worin die Entsprechung zwischen Idem und neuronalem Zustand konkret besteht, bleibt offen. Was verbirgt sich “*tatsächlich*” hinter dieser Entsprechung? Was ist ein Idem, was ist Denken, was ist Bewusstsein “wirklich”? Wir wissen es nicht, genauso wie wir nicht wissen, was das Elektron “wirklich” ist oder was das elektrische Feld “wirklich” ist.

Auch die Brückenhypothese gibt keine Antwort auf die Frage, was das Bewusstsein wirklich ist. Es gibt keine Auskunft über die “wirkliche” (wirkende) Beziehung zwischen Welt 1 und Welt 2. Diese Unkenntnis eröffnet den Freiraum für jede Art von Glauben an “jenseitige” Kräfte, für Mystik, Magie, Esoterik, für den Glauben an Gott und für alle Religionen. Daran wird eine Bestätigung der Brückenhypothese nichts ändern. Die Frage, ob die Wissenschaft irgendwann einmal verstehen wird, was Denken “wirklich” ist und was Bewusstsein “wirklich” ist, und ob die Wissenschaft beweisen wird, dass der Mensch *keine* Willensfreiheit besitzt, kann niemand beantworten. Darum hat es auch wenig Sinn, sie zu stellen. Freilich kann jeder die Frage bejahen, sozusagen als persönliches Postulat. Doch ist das ziemlich unwichtig für das wirkliche Leben, das jeder zu leben hat. Denn das Autonomieprinzip wird wirksam bleiben, solange der Mensch sich diejenigen Eigenschaften bewahrt, die ihn

befähigt haben, die kulturelle Evolution der vergangenen Jahrtausende zu tragen. Zu dieser Leistung war er imstande, *weil* er *wusste*, dass er einen freien Willen und eine Seele besitzt. Und er wird sein Werk, das wir “unsere Kultur” nennen, fortsetzen, solange ihm sein freier Wille, seine Seele und sein Gewissen nicht abhanden gekommen sind. Durch die künstliche Intelligenz wird er sie *nicht* verlieren.

Glossar

Im Glossar sind Begriffe aufgeführt, die in verschiedenen Kapiteln verwendet aber nur einmal definiert werden oder die in einer vom üblichen Sprachgebrauch abweichenden Bedeutung verwendet werden. Letzteres kann zwei Gründe haben. Entweder ist die Verwendung der betreffenden Begriffe in der Literatur nicht einheitlich, oder die angestrebte Vereinheitlichung des Begriffsapparats der Informatik war anders schwierig oder unmöglich zu erreichen. Das vorgestellte Dach, z.B. ^Semantik, ist zu lesen als "Siehe in diesem Glossar unter dem Begriff Semantik".

Abbildung 1. im weiten Sinne (**Abbildung i.w.S.**) Menge beliebiger Zuordnungen je eines Elements einer Menge, Originalmenge genannt, zu einem Element einer anderen oder auch derselben Menge, Bildmenge genannt; 2. im engen Sinne (**Abbildung i.e.S.**) eindeutige Abbildung, d.h. einem Element der Originalmenge darf durch die Abbildung nur ein einziges Element der Bildmenge zugeordnet sein; Synonym zu ^Funktion.

Ableiten Herleiten einer neuen Aussage aus bekannten Aussagen.

abzählbar unendliche Menge bzw. **Folge** unendliche ^Menge/^Folge, die dadurch entsteht, dass eine endliche Menge/Folge unendlich oft durch eine endliche Anzahl von Elementen erweitert wird, m.a.W. unendliche Menge/Folge, die sich in die Menge der natürlichen Zahlen abbilden lässt.

adressierte Speicherung Abspeicherung von Daten in einem adressierbaren Speicher, sodass auf die einzelnen Daten über deren Adressen zugegriffen werden kann.

Aktion Operation an einem oder mehreren explizit angegebenen Operanden. Eine maschinenverständliche Aktionsvorschrift heißt **Befehl** bzw. - im Falle einer höheren Programmiersprache - **Anweisung**.

Aktionsfolge zeitliche, nicht unbedingt vollständig geordnete Folge von Aktionen; in der Informatikliteratur meistens als **Steuerfluss** bezeichnet.

Aktionsfolgegraph graphische Darstellung der Struktur einer ^Aktionsfolge oder eines imperativen Programms mittels der Symbole der USB-Methode ohne Angaben hinsichtlich der Steuerung steuerbarer ^Flussknoten.

Aktionsfolgeplan ^Aktionsfolgegraph, der alle Angaben zur Steuerung der steuerbaren Flussknoten enthält, sodass er eine vollständige Operationsvorschrift darstellt.

Aktionsfolgeprogramm maschinenverständlich notierter Aktionsfolgeplan; Synonym zu **imperatives Programm**.

aktives sprachliches Modell siehe unter ^Modell.

Algorithmus Handlungsvorschrift, die angibt, in welcher Reihenfolge mehrere, als bekannt vorausgesetzte Einzelhandlungen oder Operationen auszuführen sind, um ein bestimmtes Ziel zu erreichen. Die Handlungsfolge muss eindeutig sein und nach endlich vielen Schritten enden. Ein Algorithmus, der eine Folge von ^Aktionen vorschreibt, heißt **imperativer Algorithmus**.

alternative KI auf ^subsymbolischer Ebene simulierte natürliche ^Intelligenz; Synonym: **nichttraditionelle KI**.

Alternativmasche siehe ^Masche.

ALU arithmetisch-logische Einheit; steuerbare Kombinationsschaltung, welche die elementaren Bitkettentranformationen durchführt, aus denen alle Prozesse komponiert sind, die in einem Prozessorcomputer ablaufen.

analoges Modell Gegenstand (z.B. Gerät oder Bild), dessen Merkmalswerte mit den Merkmalswerten des Originals in dem für die Modellierung ausreichenden Maße übereinstimmen; Synonym zu **nichtsprachliches Modell**.

analytisches Rechnen Rechnen mit Bezeichnern für Variablen oder Funktionen. In analytischen Rechnungen können auch Werte (Konstante) auftreten. Analytisches Rechnen wird zuweilen auch als *symbolisches* Rechnen bezeichnet.

Anweisung siehe unter ^Aktion.

Anwendungsprogramm Ausführungsvorschrift einer von einem Anwender geforderten Computeroperation.

Anwendungsprozess Ausführung eines ^Anwendungsprogramms.

Arbeitsoperator siehe ^Kompositoperator.

arbiträr beliebig, aber verbindlich festlegbar.

Arbitrarität des Artikulierens Fehlen eines zwangsläufigen, durch Naturgesetze eindeutig diktierten Zusammenhanges zwischen einem Idem und dem ihm zugeordneten Zeichenrealem.

Artikulieren Zuordnen eines ^Zeichenrealems zu einem ^Idem.

artikulierte Information ^Information, deren Realem ein ^Zeichenkörper ist.

Assemblerprogramm ^Maschinenprogramm, in dem Operanden und Operationen durch Bezeichner benannt sind.

Assoziation 1. (als Fähigkeit natürlicher Intelligenz) Wiederauffinden bekannter Aussagen (d.h. bekannter Zuordnungen zwischen Objekten und Merkmalen) ohne bewusstes Suchen; Leistung unbewusster, reproduktiver Intelligenz. 2. (als Me-

thode des Zugreifens auf Gedächtnis- bzw. Speicherinhalte) Aufrufen von Denkobjekten bzw. von im Computer gespeicherten Objekten über ihre Merkmalswerte oder Aufrufen von Merkmalswerten über Objekte, die sich durch diese Merkmalswerte auszeichnen, oder Kombination beider Aufrufmethoden.

Aussage 1. Sprachlicher Ausdruck (Code), der einem oder mehreren Objekten Merkmalswerte zuordnet oder Merkmale zueinander in Beziehung setzt. Im verallgemeinerten Sinne sind auch solche Ausdrücke Aussagen, die derartige Zuordnungen verlangen (imperative Aussagen, Befehle) oder nach ihnen fragen (interrogative Aussagen, Fragen): 2. siehe unter ^Prädikat.

Aussageform siehe unter ^Prädikat.

Axiomensystem Menge voneinanderder unabhängiger und untereinander widerspruchsfreier Aussagen, aus denen sich sämtliche wahren Aussagen eines Kalküls ableiten lassen.

axiomatisierter Kalkül Kalkül mit ^Axiomensystem.

Bausteinoperator siehe ^Kompositoperator.

Begriff benanntes ^Denkobjekt.

berechenbare Funktion ^Funktion für deren Berechnung eine Berechnungsvorschrift angegeben und in endlicher Zeit ausgeführt werden kann.

Berechnungskomplexität siehe ^Komplexität.

Betriebsmittel jede Komponente der Hardware und Software eines informationellen Systems, die der Ausführung von Operationen durch das System dient.

Betriebssystem Gesamtheit aller ^Systemprogramme, über die ein Computer verfügt.

Binäralphabet Alphabet, das nur zwei Zeichen enthält

binär-statische Codierung Codierung mittels ^Binäralphabet durch ^statisch stabile codierende Zustände.

Binärwortfunktion Funktion, deren Argument- und Funktionswerte Binärworte (Bitketten) sind.

Bioinformatik auf die Anwendung in der Biologie spezialisierte Informatik.

biologische Informatik Teil der ^Informatik, deren Gegenstand das sprachliche (codierte) Modellieren durch Lebewesen ist.

Bit Synonym zu Binärzeichen; Zeichen des Binäralphabets (eines Alphabets mit zwei Zeichen).

boolescher Operator Operator, der einzelnen Bits oder Bitketten einzelne Bits zuordnet.

boolescher Speicher Speicher, dessen ^codierende Zustände Eigenzustände zirkulärer ^boolescher Netze sind.

boolesches Netz Operatorennetz aus booleschen Operatoren.

Byte 8-stellige Bitkette.

Client-Server-Prinzip Methode zur Realisierung ^indirekter Kommunikation.

CD-ROM Compakt-Disk-^ROM.

Code realer oder gedachter Zeichenkörper (oft kurz Zeichen genannt), der einen bestimmten Bedeutungsinhalt codiert ("verschlüsselt", artikuliert).

codierende Evolution Evolution, die sich der Codierung bedient; zusammenfassende Bezeichnung von genetischer, ^intellektueller und kultureller Evolution.

codierende Modellierung siehe ^sprachliche Modellierung.

codierender Zustand stabiler Zustand eines Trägermediums, der einen Code darstellt.

codiertes Modell siehe ^sprachliches Modell.

Codierung 1. primäre Codierung: Zuordnung von Zeichenkörpern zu Bedeutungsinhalten; 2. sekundäre Codierung: ^Umcodierung eines Zeichenrealems.

Compiler Programm, das ein anderes Programm im Ganzen aus einer Sprache, Quellsprache genannt, in eine andere Sprache, Zielsprache genannt, übersetzt.

Computer-IV Informationsverarbeitung durch den Computer**.**

Computerscience siehe ^technische Informatik

Computersemantik siehe ^interne Semantik.

Daten (Einzahl: **Datum**) 1. Operanden technischer sprachlicher Operatoren; 2. Bitketten, die in Rechnern oder Rechnernetzen transportiert und in Speichern abgespeichert werden.

Datenflussgraph ^Operandenflussgraph eines sprachlichen Operators.

Datenflussplan ^Operandenflussplan eines sprachlichen Operators.

Deduktion Ableitung durch ^Rechnen oder durch ^Schlussfolgern bzw. Inferenzieren; produktive Intelligenzleistung; beim Menschen eine Leistung bewusster Intelligenz.

Denken sprachliches Modellieren ohne externes Codieren (Artikulieren).

Denkkalkül nicht unbedingt ausformulierter Kalkül, den ein Mensch beim Ableiten von Aussagen bewusst oder unbewusst anwendet.

Denkobjekt Idem, das ein durch Merkmale charakterisiertes gedachtes Objekt darstellt.

direkte Kommunikation Datentransfer zwischen Prozessen über einen gemeinsamen Adressraumbereich.

Dualzahl Zahl, die im Stellenwertsystem mit der Basis 2 notiert ist. (Eine Dezimalzahl ist eine Zahl, die im Stellenwertsystem mit der Basis 10 notiert ist).

DRAM ^dynamischer RAM .

dynamisches Binden Adresszuweisung an Bezeichner während der Laufzeit eines Programms.

dynamische Codierung Codierung durch ^dynamisch stabile Zustände.

dynamischer RAM RAM, dessen codierende Zustände Ladungszustände von Kondensatoren sind, die automatisch regeneriert werden.

dynamisch stabiler Zustand Zustand eines stofflichen Mediums, der sich periodisch oder repetierend ändert, m.a.W. eine bestimmte zeitliche Folge von Parameterwerten, die sich ständig wiederholt.

EPROM löschbarer (erasable) ^PROM.

Erfinden Finden neuer richtiger Aussagen durch unbewusste Wissensverarbeitung.

Erkennen 1. im Sinne von Wiedererkennen: Erkennen eines bekannten Objekts bzw. seiner Klassenzugehörigkeit; 2. im Sinne von Erkenntnisgewinnung: Finden einer neuen, sinnvollen Aussage über die Welt.

externe Interpretation Zuordnung externer ^Semantik zu Zeichen einer ^Kalkülsprache.

externe Semantik siehe ^Semantik.

Fernziel der KI perfekte Simulation (Kalkülisierung und Implementierung) des folgerichtigen Denkens des gesunden Menschenverstandes.

Firmware In ^ROMs eingeprägte Programme, zu denen der Nutzer i.Allg. keinen Zugang hat.

Flaschenhals 1.von-neumannscher Flaschenhals: Kommunikationsweg zwischen Prozessor und Hauptspeicher, über den alle Befehle und Daten in beiden Richtungen übergeben werden; 2. Flaschenhals des sprachlichen Modellierens: linearsprachliche, satzorientierte Schicht zwischen der externen Netzstruktur der Außenwelt und der internen Netzstruktur des Computers bzw. des Gehirns.

Flussknoten Punkte in einem Operatorennetz, an denen sich Operandenübergabewege treffen oder verzweigen (siehe Bild 8.2).

Folge Menge, deren Elemente (falls nichts Gegenteiliges gesagt ist) vollständig geordnet sind, d.h. jedes Element, mit Ausnahme des ersten und letzten, hat genau einen Vorgänger und einen Nachfolger.

formale Interpretation Zuordnung formaler (mathematischer) Bedeutungen (formaler ^Semantik) zu den Zeichen einer formalen Sprache.

formale Semantik siehe ^Semantik.

formale Sprache 1. (intensionale Definition) Menge elementarer Zeichen (Alphabet) zusammen mit einer Menge eindeutiger Syntaxregeln zur Komponierung von Kompositzeichen; 2. (extensionale Definition) Menge aller syntaktisch richtigen Kompositzeichen.

formale Theorie Kalkül, dessen Zeichen extern (durch einen Ausschnitt der Realität) interpretiert, d.h. mit externer ^Semantik belegt sind; sprachliches Modell eines Diskursbereiches, das eine Interpretation eines Kalküls darstellt.

Formalisierungsgrad siehe ^Kalkülisierungsgrad.

Funktion Synonym zu **Abbildung i.e.S.**; Menge eindeutiger Zuordnungen von Elementen (Funktionswerten) einer gegebenen Menge, der sog. Wertemenge, zu Elementen (sog. Argumentwerten) einer andere gegebenen Menge, der sog. Argumentmenge, wobei einem Argumentwert nur höchstens ein Funktionswert zugeordnet wird; die Argumentmenge kann mit der Wertemenge identisch sein.

Gabel starrer (nichtsteuerbarer) Flussknoten, in dem sich ein Operandenübergabeweg in zwei oder mehrere Wege gabelt, die ständig geöffnet sind.

Hardware gerätetechnische Ausrüstung von Computern.

Hauptprogramm ^Unterprogramm.

hierarchisches Komponieren Aufbau (Komponierung) neuer Objekte, sog. **Komposite**, aus Bausteinen. Komposite können ihrerseits als Bausteine einer höheren Komponierungsebene verwendet werden, sodass eine Hierarchie von Kompositen entsteht.

Humaninformatik Teil der ^Informatik, deren Gegenstand das sprachliche Modellieren durch den Menschen ist.

Human-IV Informationsverabeitung durch den Menschen; überdeckt sich weitgehend mit dem Gegenstand der ^Kognitionswissenschaft.

Humansprache Sprache, in der sich Menschen sowohl mündlich als auch schriftlich zum Zwecke des Informationsaustausches artikulieren können, die also als Laut-

sprache (auditive Sprache) und als Schriftsprache (visuelle Sprache) verwendet wird.

Idem relativ abgeschlossener Bewusstseinsinhalt oder "Bewusstseinsausschnitt", mit dem das Denken als einer selbständigen Einheit operiert. Siehe auch ^objektiviertes Idem.

Idemobjektivierung Aufhebung bzw. Verringerung der Abhängigkeit eines benannten Idems vom interpretierenden (artikulierenden) Subjekt.

imperatives Paradigma Grundprinzip des maschinellen sprachlichen Modellierens, nach dem die Beschreibung des Originals (des Diskursbereichs) durch Aktionsvorschriften (Befehle, Imperative) erfolgt; Sonderform des ^Satzparadigmas.

imperatives Programm siehe ^Aktionsfolgeprogramm**.**

implementierte Sprache Eine Sprache ist auf einem Computer implementiert, wenn dieser über ein Übersetzerprogramm aus der betreffenden Sprache in die ^Maschinensprache des Computers verfügt.

indirekte Kommunikation Datentransfer zwischen Prozessen ohne Benutzung eines gemeinsamen Adressbereichs.

Inferenzieren Schlussfolgern durch den Computer; Computersimulation des ^Schlussfolgerns durch den Menschen.

Informatik Lehre (Wissenschaft und Technik) vom aktiven ^sprachlichen Modellieren.

Information 1. (allgemeiner Begriff) Zusammenfassung eines ^Realems mit dem ihm zugeordneten ^Idem; eine Information heißt artikulierte Information oder **Zeicheninformation**, wenn das Realem ein ^Zeichenrealem ist; sie heißt nichtartikulierte oder **uneigentliche Information**, wenn ihr Realem ein ^Urrealem ist. 2. (spezieller Begriff, wie er in diesem Buch verwendet wird) Synonym zu Zeicheninformation, also ^Zeichenrealem mit einem zugeordneten ^Idem; oder (unter Vermeidung der Wörter Idem und Realem) ^Zeichenkörper zusammen mit der Bedeutung, die der Artikulierer (Sender) oder ein Interpretierer (Empfänger) dem Zeichenkörper zugeordnet hat.

informationeller Operator ^Operator für die Verarbeitung von Information.

informationelles System ^System für die Verarbeitung und/oder Speicherung von Information; Synonym zu **Information verarbeitendes System** (**IV-System**).

Instanzieren Übergang von einem Sammelbegriff oder von einer Klasse zu einem Exemplar der Klasse.

intellektuelle Evolution Entwicklung der individuellen Denkfähigkeit, speziell des begrifflichen Gebäudes, in welchem ein Mensch denkt.

Intelligenz Fähigkeit zum internen ^sprachlichen (intern codierenden) Modellieren.

interncodiertes Programm Programm, das im maschineninternen Binärcode geschrieben ist.

Interncodierung Codierung, die ausschließlich computerinterne binäre Codierung verwendet.

interne Semantik Prozess oder Zustand, der in einem informationellen System durch ein Realem ausgelöst wird. Im Falle eines Computers heißt interne Semantik auch Computersemantik oder Maschinensemantik.

Interpretation 1. (Allgemein) Zuordnung von Idemen zu Realemen. 2. (im Falle ^objektivierter Ideme) Zuordnung von Semantik zu Zeichenkörpern. 3. (Interpretation eines Kalküls) Zuordnung formaler oder externer Semantik zu den Bezeichnern eines Kalküls.

Interpreter Programm, das ein anderes Programm interpretiert, d.h. die einzelnen Sätze (die interpretierbaren Teile des Programms) der Reihe nach übersetzt und sofort ausführt.

Interpretieren 1. Zuordnen eines ^Idems zu einem ^Zeichenrealem, 2. Tätigkeit eines ^Interpreters.

Interpretierer realer Operator, der einen sprachlichen Operator als Operationsvorschrift versteht und ausführt.

Intuition Erfinden neuer Aussagen, d.h. neuer Zuordnungen zwischen Objekten und Merkmalen; produktive Intelligenzleistung, beim Menschen eine Leistung unbewusster Intelligenz.

iterative Berechnung wiederholte *sequenzielle* Ausführung ein und derselben Operation; die nächstfolgende Operationsausführung beginnt, *nachdem* die vorangehende beendet ist.

Kalkül ^Kalkülsprache zusammen mit einer Menge von Rechenregeln zur Transformation von Ausdrücken der Kalkülsprache.

Kalkülisierung Formulierung eines Problems als Interpretation eines ^Kaklüls zum Zwecke seiner systematischen (mathematischen) Lösung.

Kalkülisierungsgrad Grad der Vollständigkeit und Geschlossenheit der Kalkülisierung eines sprachlichen Modells; Synonym zu **Formalisierungsgrad**; Maß des Kalkülisierungsgrades: Verhältnis von analytischem zu numerischem Rechenumfang beim Ableiten von Modellaussagen ohne Berücksichtigung des numerischen Lösens von Gleichungen.

Kalkülsprache formal interpretierte ^formale Sprache; durch formale Interpretation wird den Zeichen der formalen Sprache formale, kalkülspezifische Semantik zugeordnet (^formale Interpretation).

kausaldiskrete Prozessbeschreibung Beschreibung von Prozessen als kausale Zustandsfolge in diskreten Zeitpunkten.

kausalkontinuierliche Prozessbeschreibung Beschreibung von Prozessen als zeitlich kontinuierliche kausale Zustandsfolge.

Klasse ^Menge, deren Elemente in einem oder mehrenen Merkmalen übereinstimmen; m.a.W.: Menge, die durch ein oder mehrere ^Prädikate festgelegt ist. Wenn ein festlegendes Prädikat unscharf ist, heißt die Klasse (Menge) **unscharf**.

Klassieren Einordnen eines Objekts in eine Klasse; häufig als *Klassifizieren* bezeichnet.

Klassifizieren Definieren einer ^Klasse.

Kognitionswissenschaft Lehre von den menschlichen Methoden der Organisation und Verarbeitung von Wissen; zuweilen wird der Begriff in einem erweiterten Sinne verwendet, der auch entsprechende maschinelle Methoden (KI-Methoden) einbezieht.

Kombinationsschaltung zirkelfreies boolesches Netz.

Komplex ein viele Bestandteile umfassendes Objekt der Realität oder des Denkens mit globalen Eigenschaften, auch Makroeigenschaften genannt, die sich nicht unmittelbar oder auch gar nicht aus den lokalen Eigenschaften der Komponenten, auch Mikroeigenschaften genannt, und der Struktur des Objektes ableiten lassen.

Komplexität 1. **strukturelle Komplexität** Eigenschaft der Vielgliedrigkeit, eventuell auch Vielschichtigkeit von ^Komplexen (unscharfer Begriff); 2. **nichtlineare Komplexität** irreguläres (sprunghaftes) Verhalten nichtlinearer dynamischer Systeme; 3. **Berechnungskomplexität** klassifikatorisches Merkmal für den Berechnungsaufwand eines Algorithmus; in einer Komplexitätsklasse werden alle Algorithmen zusammengefasst, deren Berechnugsaufwand mit der Problemgröße in dem gleichen Maße wächst.

Kompositoperator ^Operatorennetz mit je einem einzigen Ein- und Ausgang zusammen mit einem ^Steueroperator, der die steuerbaren Flussknoten steuert. Sowohl das Operatorennetz als auch die Operatoren, aus denen das Operatorennetz komponiert ist, heißen **Arbeitsoperatoren**. Steueroperator und Arbeitsoperatoren heißen **Bausteinoperatoren** des Kompositoperators.

Kopiergabel Gabel, die den Eingabeoperanden über die Ausgabewege weiterleitet.

KR-Netz Operatorennetz, dessen Operatoren Kombinationsschaltungen und dessen Operandenplätze Register sind.

kulturelle Evolution Entwicklung der sprachlichen Modellierung der Welt durch eine *Kulturgemeinschaft* und die darauf aufbauende technische, wissenschaftliche, künstlerische und weltanschauliche Entwicklung.

künstliche Intelligenz (KI) Fähigkeit eines technischen Systems zum ^sprachlichen Modellieren. **Traditionelle KI** auf ^symbolischer Ebene simulierte natürliche Intelligenz. **Alternative KI** auf ^subsymbolischer Ebene simulierte natürliche Intelligenz.

Lexem Zeichenkette eines Programms, die gemäß Sprachsyntax einer metasprachlichen Klasse zuordenbar ist.

lexikale Analyse Klassifizierung (Klassierung) der Wörter eines Programms nach Lexemklassen.

Masche Teilstruktur eines Operatorennetzes, die aus zwei gleichgerichteten Operandenwegen zwischen zwei Operatoren besteht, wobei in jedem Weg beliebig viele weitere Operatoren liegen können; wenn beide Äste gleichzeitig von Operanden durchlaufen werden, heißt die Masche **Parallelmasche** oder **starre Masche**; wenn die Äste nur alternativ durchlaufen werden können, heißt sie **Alternativmasche** oder **steuerbare Masche**.

Maschinenbefehl maschinenlesbare ^Aktionsvorschrift.

Maschinenebene Komponierungsebene, auf der ^Maschinenprogramme komponiert (interpretiert) werden.

Maschinenkalkül Maschinensprache eines Computers zusammen mit ihrer internen Semantik (den semantischen Regeln, nach denen der Computer die Befehle der Maschinensprache ausführt).

Maschinenprogramm Berechnungsvorschrift in Form einer Folge von Maschinenbefehlen; Oberbegriff der Begriffe ^interncodiertes Programm und ^Assemblerprogramm; im Falle eines Einprozessorrechners identisch mit ^Prozessorprogramm.

Maschinensemantik ^interne Semantik.

Maschinensprache Sprache zur Programmierung von ^Maschinenprogrammen.

Menge mathematischer Begriff, unter dem unterschiedliche *Elemente* beliebiger Natur (Objekte der Realität oder des Denkens) zusammengefasst werden.

Metaintelligenz Fähigkeit, das eigene sprachliche Modellieren zu modellieren und zielgerichtet weiterzuentwickeln.

Metasprache Sprache, in der Aussagen über eine andere Sprache, **Objektsprache** genannt, gemacht werden.

metasprachliche Variable Bezeichner einer Klasse sprachlicher Konstrukte, z.B. der Klasse der Aussagesätze oder der Klasse der bedingten Anweisungen.

Modell Oberbegriff der Begriffe ^analoges Modell und ^sprachliches Modell.

nebenläufige Prozesse Prozesse, die kausal voneinander unabhängig sind, d.h. keiner ist die Ursache (Voraussetzung) des/der anderen. Nebenläufige Prozesse können parallel (gleichzeitig) ausgeführt werden.

Netzparadigma Art und Weise des Denkens und des Beschreibens komplexer Objekte oder Prozesse, das von der Vorstellung eines Netzes miteinander in Beziehung stehender Objekte oder Akteure ausgeht.

Neurocomputer informationelles System, das ein oder mehrere neuronale Netze enthält.

neuronaler Speicher Speicher, dessen statisch codierende Zustände Eigenzustände zirkulärer neuronaler Netze sind.

nichtberechenbare Funktion ^Funktion, für deren Berechnung keine Berechnungsvorschrift oder keine in endlicher Zeit ausführbare Vorschrift angegeben werden kann.

nichtmathematisches Problem Problem, das vom Menschen normalerweise nicht als mathematisches Problem aufgefasst (erkannt) wird.

nichtlineare Komplexität siehe ^Komplexität.

nichttraditionelle KI siehe alternative KI.

numerisches Rechnen Rechnen mit Werten; in numerischen Rechnungen treten keine Variablen, sondern nur Konstanten oder Werte von Variablen auf, z.B. Zahlen oder Wahrheitswerte.

Nutzersemantik Semantik, in welcher der Nutzer eines Computers denkt, m.a.W. die Ideme, die ein Computernutzer den Ein- und Ausgaben des Computers zuordnet.

Objekt 1. siehe ^Denkobjekt; 2. siehe ^Programmobjekt.

objektiviertes Idem Idem, das einem ^Zeichenrealem zugeordnete ist und für alle Beteiligten in einem für die Verständigung ausreichenden Maße übereinstimmt; Synonym zu ^Semantik.

Objektsprache siehe ^Metasprache.

Operandenflussgraph graphische Darstellung eines Kompositoperators als Operatorennetz nach der USB-Methode ohne Angaben hinsichtlich der Steuerung der steuerbaren Flussknoten.

Operandenflussplan ^Operandenflussgraph, der alle Angaben zur Steuerung der steuerbaren Flussknoten enthält, sodass er eine vollständige Operationsvorschrift darstellt.

Operation Abbildung, die durch einen *gedachten* Operator realisiert wird.

Operationsausführung Synonym zu ^Prozess; Ausführung einer Operation durch einen realen Operator.

Operator ein Mensch, eine Vorrichtung (ein Gerät) oder eine Vorschrift, der/die einem Eingabeoperanden einen Ausgabeoperanden zuordnet. Wird die Zuordnung durch einen Menschen oder ein Gerät vorgenommen, heißt der Operator **real**; wird sie durch eine Vorschrift festgelegt, heißt er **sprachlich**.

Operatorabstraktion Abstraktion von der inneren Struktur eines ^Kompositoperators.

Operatorennetz (ON) Menge von Operatoren, die durch Operandenübergabewege miteinander verbunden sind. Der Operandenfluss durch ein ON wird durch starre und steuerbare ^Flussknoten festgelegt.

Organisationsprogramm Vorschrift für die Zuweisung eines Betriebsmittels an einen ^Anwendungsprozess.

Paradoxon der KI Möglichkeit der Komponierung nichtmathematischer Denkoperationen aus den Operationen der ^ALU.

Parellelmasche siehe ^Masche.

peripheres Steuerprogramm Programm zur Einrichtung (Konditionierung) eines peripheren Gerätes auf eine bestimmte Operation.

Prädikat (im Sinne der Prädikatenlogik) Sprachlicher Ausdruck, der ein oder mehrere Merkmalswerte eines oder mehrerer Objekte festlegt oder der eine Relation zwischen Merkmalen verschiedener Objekte festlegt. Objekte können durch Variablen vertreten sein. Dann wird das Prädikat auch **Aussageform** genannt. Ein Prädikat, das nur Konstanten enthält, heißt **Aussage** (im Sinne der Prädikatenlogik). Ein Prädikat heißt **unscharf**, wenn es einen Merkmalswert nicht exakt festlegt.

Programmablaufplan (PAP) genormte Darstellung eines ^Aktionsfolgeplans.

Programmobjekt Objekt im Sinne der objektorientierten Programmierung; Teil eines Programms (ein Programm-Modul), der einem relativ abgeschlossenen Bereich des modellierten Originals (Diskursbereiches) und einem relativ abge-

schlossenen Bewusstseinsausschnitt (^Denkobjekt) des Nutzers entspricht; i.Allg. kann der Anwendungsprogrammierer bestimmen, wieweit ein Prozess, der bei Ausfrührung eines Objekts abläuft, gegen andere Prozesse geschützt (gekapselt) werden soll.

Prozedur siehe ^Unterprogramm.

prozedurale Abstraktion Abstraktion von der inneren Struktur einer ^Prozedur, d.h. einer Kompositaktion.

Prozess 1. (umgangssprachlich) Ausführung einer Operation; 2. (aus der Sicht des Computernutzers) Abstraktion eines in Ausführung befindlichen Programms; 3. (aus der Sicht des Systemprogrammierers) zeitliche Folge von Betriebsmittelzuständen, die sich während der Abarbeitung eines im Hauptspeicher befindlichen Programms einstellen.

Prozessor zentrale Verarbeitungseinheit eines Computers; er führt Maschinenprogramme Befehl für Befehl aus.

Prozessorcomputer oder **traditioneller Computer** Computer, der Programme mit Hilfe eines oder mehrerer ^Prozessoren abarbeitet.

Prozessorebene Komponierungsebene, auf der ^Prozessorprogramme komponiert (interpretiert) werden.

Prozessorprogramm Folge von Befehlen aus dem Operationsrepertoire und mit dem Format des Befehlsregisters eines Prozessors.

Prozessor-Speicher-Netz (PS-Netz) Operatorennetz, dessen Operatoren Prozessoren sind.

Prozessorsprache Programmiersprache, die eine direkte Schnittstelle mit einem Prozessor besitzt und der Artikulierung von ^Prozessorprogrammen dient. Ein Mehrprozessorrechner kann mehrere Prozessorsprachen, aber nur eine ^Maschinensprache besitzen.

PS-Netz siehe ^Prozessor-Speicher-Netz.

RALU ^ALU zusammen mit Registern, die für die Ausführung der zentralen Steuerschleife erforderlich sind.

RAM Schreib-Lesespeicher (Random Access Memory); Speicher, auf dessen Speicherzellen über Adressen schreibend und lesend zugegriffen werden kann.

Realem Ausschnitt der realen Welt, der sich im Bewusstsein eines Menschen widerspiegelt, dem also ein ^Idem entspricht.

Realisierbarkeitsprinzip Forderung, dass alle IV-Systeme und alle informationellen Prozesse, von denen die Rede ist, realisierbar sind, d.h. dass sie mit endlichem

materiellen Aufwand und in endlicher Zeit gebaut bzw. ausgeführt werden können.

Rechnen formales ^Ableiten im Rahmen eines Kalküls.

Referenzieren 1. (allgemein) Verweisen auf ein Objekt der Realität oder des Denkens (auf ein Realem oder Idem), durch Nennung eines Stellvertreters, eines Namens oder Pronomens, oder durch Angabe seines Ortes (z.B. postalische Adresse, bibliographische Angaben, Speicheradresse); 2. (in der Computer-IV) Verweisen auf ein Zeichenrealem mittels eines anderen Zeichenrealems.

rekursive Berechnung wiederholte, verschachtelte Ausführung ein und derselben Operation, d.h. die nächstfolgende Operationsausführung beginnt, *bevor* die vorangehende beendet ist.

rekursive Funktion ^Funktion, die mittels Substitution, Selektion, Iteration und Minimalisierung festgelegt und berechnet werden kann.

Ressource siehe ^Betriebsmittel.

ROM Festwertspeicher, Nur-Lese-Speicher (Read Only Memory).

Rückkopplungsschleife siehe ^Schleife.

Sammelweiche steuerbarer, sequenzialisierender Flussknoten, in dem sich zwei oder mehrere Operandenübergabewege treffen, von denen jeweils nur einer geöffnet ist.

Satz i.w.S. Oberbegriff der Begriffe humansprachlicher Satz, ^Maschinenbefehl und ^Anweisung.

Satzparadigma am häufigsten angewandtes Grundprinzip des sprachlichen Modellierens, nach dem die Beschreibung des Originals (des Diskursbereichs) durch vollständige ^Sätze i.w.S. erfolgt.

Scheibenspeicher Sammelbezeichnung für alle Speicher, die mit rotierenden Scheiben als Speichermedium ausgerüstet sind, unabhängig davon, welche physikalischen Speicherverfahren (magnetische, optische) zum Einsatz kommen.

Schleife Synonym zu **Rückkopplungsschleife** Teilstruktur eines Operatorennetzes, die einen in sich zurücklaufenden (zirkulären) Operandenweg darstellt.

Schlussfolgern (durch den Menschen) ^Ableiten von Schlüssen aus gegebenen Fakten, das aus introspektiver Sicht des denkenden Menschen i.d.R. nicht explizit formalisiert ist.

Schwellenoperator Operator mit reellwertigen Eingabeoperanden und binärwertigen Ausgabeoperanden; das Eingabe-Ausgabeverhalten ist als Stufenfunktion

darstellbar; bei einem bestimmten Eingabewert, Schwellwert genannt, springt der Ausgabewert von dem einen auf den anderen Binärwert.

Semantik 1. Teilgebiet der Sprachwissenschaft, das sich mit den Beziehungen zwischen Zeichenkörpern und ihren Bedeutungen beschäftigt. 2. Synonym zu ^objektiviertes Idem; nur in diesem Sinne wird das Wort "Semantik" in diesem Buche verwendet. Ein objektiviertes Idem heißt **externe Semantik** des zugeordneten Zeichenrealems, wenn dieses ein Objekt der Umwelt bezeichnet; wenn es Element einer Kalkülsprache ist, heißt das objektivierte Idem **formale Semantik**. Siehe auch ^interne Semantik.

semantische Dichte 1. (s.D. eines natürlichsprachigen Textes) Umfang des im Mittel pro Wort assoziierbaren Gedächtnisinhaltes (Menge der assoziierbaren Denkobjekte oder Ideme); 2. (s.D. von Programmiersprachen bzw. Programmen) mittlere Anzahl elementarer Operationen (ALU-Operationen) pro Zeichen.

semantische Lücke 1. Inkompatibilität (das Nicht-zueinander-passen) von Idemartikulierungen durch verschiedene Menschen oder Inkompatibilität von Zeichenrealemen verschiedener Sprachen. 2. Inkompatibilität zwischen ^interner und ^externer ^Semantik.

Software programmtechnische Ausrüstung von Computern.

Spaltegabel ^Gabel, die ein Eingabetupel in Teiltupel aufspaltet und getrennt weiterleitet.

spezielles Denkkalkül Ergebnis der (evtl. unbewussten) Kalkülisierung beim schlussfolgernden Denken.

Sprache 1. (allgemein) Mittel der codierenden (sprachlichen) Modellierung; 2. (im Sinne der Linguistik) vereinbartes System von Zeichen und syntaktischen Regeln zum Artikulieren von sprachlichen Ausdrücken (Kompositzeichen), die als ^Aussagen interpretiert werden können. Kompositzeichen können ein-, zwei- oder dreidimensional sein; 3. (extensionale Definition) Menge aller definitionsgemäß (z.B. von einer Grammatik) zugelassenen Zeichenketten.

sprachliches Modell durch idealisierende Abstraktion vereinfachte Beschreibung eines Originals (Diskursbereiches) in Form von gedachten (intern codierten) oder artikulierten (extern codierten) ^Aussagen über das Original. Ein sprachliches Modell, das den Träger der modellierenden Prozesse einschließt, heißt **aktives** sprachliches Modell.

sprachliches Modellieren Synonym zu **codierendes Modellieren**; Finden richtiger ^Aussagen über das Original. Richtige Aussagen können durch ^Deduktion, ^Assoziation oder ^Intuition gefunden werden.

Sprachparadigma konzeptionelles Grundschema einer Programmiersprache, das einem bestimmten *Denkschema* des Programmierers angepasst ist, z.B. dem Denken in Algorithmen, in Funktionen oder in Operatorennetzen.

statisch berechenbare Funktion Funktion, für deren Berechnung ein informationelles System existiert oder angegeben werden kann, das mit statischer Codierung arbeitet.

statische Codierung Codierung durch ^statisch stabile Zustände.

statisches Binden Adresszuweisung an Bezeichner vor dem Start eines Programms.

Steueroperator Operator, der die steuerbaren Flussknoten eines Operatorennetzes steuert.

statisch stabiler Zustand Zustand eines stofflichen Mediums, der sich im Laufe der Zeit nicht ändert (dessen Zustandsparameter zeitlich konstante Werte besitzen).

strukturelle Komplexität siehe ^Komplexität.

strukturelle Speicherung Abspeicherung einer Funktionstafel durch Realisierung einer Kombinationsschaltung mit der abzuspeichernden Funktionstafel, sodass bei Eingabe eines Argumentwertes der entsprechende Funktionswert ausgegeben wird.

Subroutine siehe ^Unterprogramm.

subsymbolische Ebene Betrachtungsebene informationeller Prozesse, auf der keine ^Informationen (^Symbole) betrachtet werden, sondern nur der stoffliche Träger von Zeichenkörpern und informationellen Prozessen; die Festlegung codierender Zustände und ihre semantische Belegung erfolgt nicht oder erst nach der Verarbeitung, d.h. nach der Ausbildung stabiler Zustände im Träger.

Symbol vereinbarte ^Zeicheninformation, deren Zeichenkörper kompakt, evtl. ein elementares Zeichen ist, dem durch Gewohnheit oder Vereinbarung (Konvention) eine feste Bedeutung (objektiviertes Idem, ^Semantik) zugeordnet ist.

symbolische Ebene Betrachtungsebene informationeller Prozesse, auf welcher Zeichenkörper stets in Verbindung mit ihrer Bedeutung, also nur ^Informationen (^Symbole) betrachtet werden; die semantische Belegung der Zeichenkörper erfolgt vor der Verarbeitung.

Symboltabelle Tabelle, in der für jeden Variablenbezeichner eines Programms die Adresse und weitere Attribute der Variablen eingetragen werden. Die Eintragungen führt das Übersetzerprogramm bzw. der Assembler aus.

Syntax 1. (in diesem Buch nicht verwendeter grammatikalischer Begriff) Satzlehre; 2. (Syntax einer Sprache) Gesamtheit aller Regeln (sog. **Syntaxregeln**) für das

Komponieren von Kompositzeichen der Sprache (z.B. Wörter, Sätze, Kommandos, Programme) ohne Berücksichtigung der Bedeutung.

System Ein allgemeiner Systembegriff wird in dem Buch nicht definiert; verwendet wird er i.Allg. als Synonym zu "komplexer Kompositoperator".

Systemprogramm Oberbegriff der Begriffe ^Organisationsprogramm und ^peripheres Steuerprogramm.

technische Informatik Teil der ^Informatik, deren Gegenstand das sprachliche Modellieren durch technische Geräte (Computer) ist; Synonym zum amerikaischen **Computerscience**.

technisches Semantikproblem Problem der partiellen Anbindung der ^Nutzersemantik an die ^Maschinensemantik.

Theorie siehe ^formale Theorie.

traditionelle KI auf ^symbolischer Ebene mittels ^Prozessorcomputer simulierte natürliche ^Intelligenz.

traditioneller Computer siehe ^Prozessorcomputer.

Träger 1. Träger eines Codes: stoffliches Medium (reales Objekt), in das der Code als stabiler Zustand des Mediums eingeprägt ist; 2. Träger eines Prozesses: stoffliches Medium (reales Objekt), in dem der Prozess abläuft.

Trägerprinzip Forderung, bei allen Überlegungen und Begriffsdefinitionen bezüglich der Informationsverarbeitung vom ^Träger auszugehen.

Tupel Menge von Elementen, z.B. Zahlen oder Bezeichnern, die in einer bestimmeten Reihenfolge angeordnet sind. Ein Zahlentupel heißt **Vektor**.

Umcodierung Ersetzen eines Zeichenrealems durch ein anderes, ohne das artikulierte Idem (die Bedeutung) zu verändern; wird häufig auch als ^Codieren bezeichnet.

unbeschränkte Menge Menge, die abzählbar unendlich sein kann, aber nicht sein muss.

uniforme Systembeschreibung (USB) Beschreibung komplexer Operatoren als Operatorennetze bzw. Operatorenhierarchien unter Verwendung der Begriffe ^Operatoren, Operandenplätze und ^Flussknoten und deren Symbole.

unscharfe Menge siehe ^Klasse

Unterprogramm Synonym zu **Prozedur** und **Subroutine**; Teil eines imperativen Programms, des sog. **Hauptprogramms**, der über einen Bezeichner aufgerufen und als selbständiges Programm ausgeführt werden kann.

Uridem Idem eines ^Urrealems.

Urrealem Realem, das kein Zeichenrealem ist; nichtsymbolisches Objekt der Außenwelt.

USB-Funktion ^Funktion, die von einem nach der USB-Methode komponierten Operator berechnet werden kann.

Vektor siehe ^Tupel.

Vereinung starrer, synchronisierender Flussknoten, in dem zwei oder mehrere Operandenübergabewege zusammentreffen, sodass die Eingabeoperanden sich zu einem Tupel vereinigen; alle Eingabewege sind ständig geöffnet; der Ausgabeweg wird nur geöffnet, wenn das Tupel vollständig ist.

Von-Neumann-Rechner Computer, der im Wesentlichen aus einem Prozessor als zentraler Verarbeitungseinheit und einem Hauptspeicher besteht, in dem sowohl die Daten als auch die Programme abgespeichert werden und dem Prozessor zur Verarbeitung zur Verfügung stehen.

Weiche steuerbarer ^Flussknoten.

Wertetafel (einer Operation bzw. Funktion) Folge von Paaren, deren erstes Element ein Eingabeoperand bzw. Argumentwert (Argumentwertetupel) und deren zweites Element der zugeordnete Ausgabeoperand bzw. Funktionswert (Funktionswertetupel) ist. Die Folge wird i.d.R. als zweispaltige Tabelle (Tafel) dargestellt. Eine Wertetafel heißt endlich bzw. abzählbar unendlich, wenn die Länge der Tafel und/oder die Längen der Zeilen endliche bzw. ^abzählbar unendliche Folgen sind.

Wissen Menge von Aussagen, an deren Wahrheit derjenige, der das Wissen besitzt, nicht zweifelt.

Wohlstruktur Struktur eines Operatorennetzes, dessen Graph keine Überlappungen von Maschen oder Schleifen enthält.

Zeichenidem gedachter ^Zeichenkörper, dem vom Denkenden ein ^Idem zuordnet ist.

Zeicheninformation ^Information, deren ^Realem ein ^Zeichenkörper ist.

Zeichenkörper elementares Zeichen (z.B. ein Buchstabe) oder ein Kompositzeichen (z.B. eine Kette oder ein Muster elementarer Zeichen).

Zeichenrealem realer ^Zeichenkörper, dem vom Artikulierer ein ^Idem zugeordnet ist oder dem von einem Interpretierer ein Idem zugeordnet werden kann.

zirkelfreies Netz Operatorennetz, das keine zirkulären Verbindungswege (Schleifen, Rückkopplungen) enthält.

zirkuläres Netz Operatorennetz, das zirkuläre Verbindungswege (Schleifen, Rückkopplungen) enthält.

Zirkularität alle Arten von Rückbezüglichkeit und Rückwirkung auf sich selbst oder Rückfluss vom Ausgang zum Eingang (Reflexivität, Rekursivität, Selbstaufruf, Rückkopplung).

Zweigeweiche steuerbarer Flussknoten, in dem sich ein Operandenübergabeweg verzweigt, wobei jeweils nur ein Ausgabeweg geöffnet ist.

Literatur

Abelson, Harolf; Sussman, Gerald Jay; Sussman, Julie: Struktur und Interpretation von Computerprogrammen. Berlin Heidelberg New York: Springer-Verlag 1991

Adler, H.: Elektronische Analogrechner. Berlin: VEB Deutscher Verlag der Wissenschaften, 1966

Aho, Alfred V.; Ravi Sethi: Jeffrey V. Ullman: Compilerbau; Bd. 1 und 2. Bonn u.a. 1992, Addison-Wesley Verlag (Deutschland) GmbH

Appelrath, Hans-Jürgen; Boles, Dietrich; Claus, Volker; Wegener, Ingo: Starthilfe Informatik. Stuttgart, Leipzig: Teubner, 1998

Balzert, Helmut: Lehrbuch Grundlagen der Informatik : Konzepte und Notationen in UML, Java und C++, Algorithmen und Software-Technik, Anwendungen. Heidelberg; Berlin: Spektrum, Akad. Verl., 1999

Bauer, Friedrich L.: Wer erfand den von-Neumann-Rechner?. Informatik Spektrum 21, 84-89 (1998)

Berka, Karel; Kreiser, Lothar: Logik-Texte. Kommentierte Auswahl zur Geschichte der modernen Logik. Berlin: Akademie-Verlag, 1983

Biaesch-Wiebke, Claus: CD-Player und R-DAT-Recorder. Würzburg: Vogel, 1992

Böhm, C.; Jacopini, G.: Flow Diagrams, Turing Machines and Languages with only two Formation Rules. Commun. ACM **9** (1966) H.5, S. 366-371

Börger, Egon: Berechenbarkeit, Komplexität, Logik. Braunschweig, Wiesbaden: Vieweg, 1992

Borland GmbH (Hrsg.): Borland Pascal mit Objekten 7.0. Programmierhandbuch. Langen: Verlag Borland, 1993

Briegel, Hans-Jürgen; Ignacio Cirac; Peter Zoller: Quantencomputer. Physikalische Blätter **55** (1999) Nr. 9, S. 37-43

Brockman, John: Die dritte Kultur. Das Weltbild der modernen Naturwissenschaften. München: Goldmann Verlag, 1996

Bronstein, Ilja N.; Konstantin A. Semendjajew; Gerhard Musiol; Heiner Mühlig: Taschenbuch der Mathematik. Thun, Frankfurt am Main: Deutsch, 1995

Broy, Manfred: Informatik. Eine grundlegende Einführung. Berlin, Heidelberg u.a.: Springer, Bd.1 1992, Bd.2 2993, Bd.3 1994, Bd.4 1995

Broy, Manfred: Mathematik des Software-Engineering. In: Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer-Verlag, 1996

Broy, Manfred; Spaniol, Otto (Hrsg.): VDI-Lexikon Informatik und Kommunikationstechnik. Berlin; Heidelberg u.a.: Springer, 1999

Capurro, Rafael: Information. Ein Beitrag zur etymologischen und ideengeschichtlichen Begründung des Informationsbegriffs. München: K.G.Saur, 1978

Chomsky, Noam: Sprache und Geist. Frankfurt am Main: Suhrkamp Verlag, 1970

Churchland, Patricia S.; Terrence J. Sejnowski: Grundlagen zur Neuroinformatik und Neurobiologie. Braunschweig, Wiesbaden: Vieweg, 1997

Conrad, Rudi (Hrsg.): Kleines Wörterbuch sprachwissenschaftlicher Termini. Leipzig: VEB Bibliographisches Institut, 1975

Coy, Wolfgang et al. (Hrsg.): Sichtweisen der Informatik. Braunschweig, Wiesbaden: Friedrich Vieweg und Sohn, 92

Cutland, Nigel : Computability. Cambridge: Cambridge University Press 1980, S. 366-371

Cyranek, Günther; Coy, Wolfgang (Hrsg.): Die maschinelle Kunst des Denkens. Braunschweig, Wiesbaden: Vieweg, 1994

Dorffner, Georg: Konnektionismus. Stuttgart: Teubner, 1991

Duden Informatik. Mannheim, Wien, Zürich: Dudenverlag, 1989

Ebeling, Werner: Strukturbildung bei irreversiblen Prozessen. Leipzig: Teubner, 1976

Ebeling, Werner; Feistel, Rainer: Physik der Selbstorganisation und Evolution. Berlin: Akademieverlag, 1982

Ebeling, Werner: Chaos, Ordnung, Information: Selbstorganisation in Natur und Technik. Frankfurt am Main, Thun: Deutsch, 1991

Ebeling, Werner; Feistel, R : Chaos und Kosmos. Prinzipien der Evolution. Heidelberg: Spektrum Akademischer Verlag, 1994

Ebeling, Werner; Freund, Jan; Schweizer, Frank: Komplexe Strukturen, Entropie und Information. Stuttgart, Leipzig: Teubner, 1998

Eberle, Hans: Architektur moderner RISC-Mikroprozessoren. Informatik Spektrum, H. 5/97, S. 259-267

Eccles, John C.: Gehirn und Seele. München, Zürich: Piper, 1988

Eigen, Manfred; Winkler, R.: Das Spiel. München: Piper, 1975

Eigen, Manfred: Wie entsteht Information? Prinzipien der Selbstorganisation in der Biologie. Berichte d. Bunsenges. **80** (1976) 1059

Eigen, Manfred: Stufen des Lebens. Die frühe Evolution im Visier der Molekularbiologie. München: Piper, 1993

Ehrenstein, Walter: Intelligentes Denken. Veröffentlicht in "Die Ganzheit in Wissenschaft und Schule". Dortmund: W.Crüwell Verlagsbuchhandlung, 1956

Ernst, Hartmut: Grundlagen und Konzepte der Informatik. Eine Einführung in die Informatik ausgehend von den fundamentalen Grundlagen. Braunschweig, Wiesbaden: Vieweg, 2000

Ernst, Bruno: Der Zauberspiegel des M.C.Escher. Deutscher Taschenbuchverlag, 1985

Feldmann, Rainer; Monien, Burkhard; Mysliwietz, Peter: Ein effizienter verteilter Algorithmus zur Spielbaumsuche. In: Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer-Verlag, 1996

Fenzl, Norbert; Hofkirchner, Wolfgang; Stockinger, Gottfried (Hrsg.): Information und Selbstorganisation. Annäherung an eine vereinheitlichte Theorie der Information. Innbruck: Studienverlag, 1998

Fleissner, Peter; Fleissner, Gregor: Jenseits des chinesischen Zimmers: Der blinde Springer. Selbstorganisierte Semantik und Pragmatik am Computer. In: Fenzl, Norbert; Hofkirchner, Wolfgang; Stockinger, Gottfried (Hrsg.): Information und Selbstorganisation. Annäherung an eine vereinheitlichte Theorie der Information. Innsbruck: Studienverlag, 1998, S.325

Flik, Thomas; Liebig, Hans: Mikroprozessortechnik. CISC, RISC, Systemaufbau, Programmierung. Berlin, Heidelberg: Springer-Verlag, 1994

Gates, Bill: Der Weg nach vorn. Die Zukunft der Informationsgesellschaft. München: Wilhelm Heyne Verlag, 1997

Gell-Mann, Murray: Das Quark und der Jaguar. München, Zürich: Piper, 1996

Gödel, Kurt: Über formal entscheidbare Sätze der Principia Mathematika und verwandter Systeme. Monatshefte für Mathematik und Physik, **38** (1931), S. 173-198

Goos, Gerhard:Vorlesungen über Informatik. Vier Bände. Berlin, Heidelberg u.a.:Springer 1997

Grauel, Adolf: Neuronale Netze. Grundlagen und mathematische Modellierung. Mannheim u.a.: BI Wissenschaftsverlag, 1992

Grauel, Adolf: Fuzzy-Logik. Mannheim u.a.: BI Wissenschaftsverlag, 1995

Gumm, Heinz-Peter; Sommer, Manfred. Unter Mitw. von Bernhard Seeger und Wolfgang Hesse: Einführung in die Informatik. München; Wien: Oldenbourg, 2000

Haken, H.: Erfolgsgeheimnisse der Natur. Stuttgart: Deutsche Verlagsanstalt, 1981

Hennessy, John L.; David A. Patterson: Rechnerarchitektur. Analyse, Entwurf, Implementierung, Bewertung. Braunschweig, Wiesbaden: Vieweg, 1994

Hermes, Hans: Aufzählbarkeit Entscheidbarkeit Berechenbarkeit. Berlin, Heidelberg, New York: Springer-Verlag, 1971

Hertz, Heinrich: Die Prinzipien der Mechanik. Leipzig: Johann Ambrosius Barth, 1894

Hofstadter, Douglas R.: Gödel, Escher, Bach - ein endloses geflochtenes Band. Stuttgart: Klett-Cotta, 1985

Hofstadter, Douglas R.: Metamagikum. Stuttgart: Klett-Cotta, 1991

Horn, Erika; Wolfgang Schubert: Objektorientierte Software-Konstruktion. München, Wien: Hanser, 1993

Hotz, Günter; Reichert, Armin: Hierarchischer Entwurf komplexer Systeme. In: Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer-Verlag, 1996

Jungclaussen, Hardwin: Grundlagen der Kybernetik III. Asynchrone Operatorennetze. Institutspublikationen des Zentralinstituts für Kernforschung Rossendorf bei Dresden. Teil 1: ZfK 420 (1980), Teil 2: ZfK 475 (1982), Teil 3: ZfK 501 (1983), Teil 4: ZfK 536 (1984), Teil 5 : ZfK 598 (1986), Teil 6: ZfK 702 (1990)

Jungclaussen, Hardwin: Zur Einheit von Hardware und Software. Wiss Z. Techn. Univers. Dresden **34** (1985) H.4, S.95-102

Jungclaussen (88), Hardwin: Platz der Informatik im System der Wissenschaften. 4.Kongreß der Informatiker der DDR INFO 88, 22.-26.2.1988 Dresden, Kongressmaterialien, S.386-388

Jungclaussen (88a), Hardwin: Mathematik und Informatik - Versuch einer Abgrenzung. Dresdner Reihe zur Forschung 6/1988, S.53-57; Päd. Hochschule Dresden, 1988

Jungclaussen (90a), Hardwin: Uniforme Systembeschreibung (USB). Wiss. Beiträge zur Informatik 1/1990, S.43-52, TU Dresden, 1990

Kemper, Alfons; André Eickler: Datenbanksysteme. Eine Einführung. München, Wien: R.Oldenbourg Verlag, 1997

Kistler, Werner M.; J. Leo van Hemmen: An Analytically Solvable Model of Collective Excitation Patterns in Cortical Tissue. In: Parisi, Jürgen; Stefan C.Müller; Walter Zimmermann(Eds.): A Perspective Look at Nonlinear Media. From Physics to Biology an Social Sciences. Berlin u.a.: Springer, 1998

Klix, Friedhart: Erwachendes Denken. Eine Entwicklungsgeschichte der menschlichen Intelligenz. Berlin: VEB Deutscher Verlag der Wissenschaften, 1980

Kreß, Dieter: Informations- und Kodierungstheorie. In: Philippow, Eugen (Hrsg.): Taschenbuch Elektrotechnik, Bd.2 Grundlagen der Informationstechnik. Berlin: VEB Verlag Technik, 1977

Küppers, Bernd-Olaf (Hrsg.): Ordnung aus dem Chaos. München, Zürich: Piper, 1987

Lehmann, Nikolaus N.: Wiss. Z. Tech. Univ. Dresden **27** (1978) H.1, S.104

Lockemann, Peter C.; Gehard Krüger; Heiko Krumm: Telekommunikation und Datenhaltung. München, Wien: Carl Hanser Verlag, 1993

Louden, Kenneth C.: Programmiersprachen. Grundlagen, Konzepte, Entwurf. Bonn u.a.: International Thomson Publishing GmbH, 1994

Malcew, A.I.:Algorithmen und rekursive Funktionen. Braunschweig, Vieweg, 1974

Märtin, Christian: Rechnerarchitektur. Struktur, Organisation, Implementierungstechnik. München, Wien: Carl Hanser Verlag, 1994

Masuda, Yonej: The Information Society as Post-Industrial Society. Washington C.D.: World Future Society, 1981

Matschke, Joachim: Von der einfachen Logikschaltung zum Mikrorechner. Berlin: Verlag Technik, 1986

Merzenich, Wolfgang; Zeidler, Hans Christoph: Informatik für Ingenieure : eine Einführung. Stuttgart: Teubner, 1997

Messmer, Hans-Peter: PC-Hardwarebuch: Aufbau, Funktionsweise, Programmierung; ein Handbuch nicht nur für Profis. Bonn u.a.: Addison-Wesley, 1995

Minsky, Marvin; The Society of Mind. New York et al.: Simon & Schuster, 1986

Nauck, Detlef; Frank Klawonn; Rudolf Kruse: Neuronale Netze und Fuzzy-Systeme. Braunschweig, Wiesbaden: Vieweg, 1996

Neumann, John von: Theory of Self-Reproducing Automata (A.Brucks, ed.) Univ. Ill. Press, 1966

Nicolis, Grégoire; Ilya Prigogine: Die Erforschung des Komplexen. Auf dem Wege zu einem neuen Verständnis der Naturwissenschaften. München, Zürich: Piper, 1987

Nievergelt, Jürg: Gewußt oder gesucht: Spieltheorie für Menschen und für Maschinen. In: Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer-Verlag, 1996, S. 25-41

Noack, Jörg; Schienmann, Bruno: Objektorientierte Vorgehensmodelle im Vergleich. Informatik Spektrum Band 22, Heft 3, 1999

Norretranders, Tor: Spüre die Welt. Die Wissenschaft des Bewußtseins. Hamburg: Rowohlt, 1997

Oertel, Hebert jr.; E.Laurien: Numerische Strömungsmechanik. Berlin u.a.: Springer 1995.

Ortoli, Sven; Witkowski, Nicolas: Kekulés Schlange. In: Bohnet-von der Thüsen, Heidi (Hrsg.): Denkanstöße '99. München, Zürich: Piper, 1998

Parisi, Jürgen; Stefan C.Müller; Walter Zimmermann (Eds.): Nonlinear Physics of Complex Systems. Current Status and Future Trends. Berlin u.a.: Springer, 1996

Parisi, Jürgen; Stefan C.Müller; Walter Zimmermann(Eds.): A Perspective Look at Nonlinear Media. From Physics to Biology an Social Sciences. Berlin u.a.: Springer, 1998

Penrose, Roger: Computerdenken. Die Debatte um künstliche Intelligenz. Bewußtsein und die Gesetze der Physik. Heidelberg: Spektrum der Wissenschaft Verlagsgesellschaft, 1991

Penrose, Roger: Schatten des Geistes. Wege zu einer neuen Physik des Bewußtseins. Heidelberg, Berlin, Oxford: Spektrum Akad. Verl., 1995

Pflüger, Jörg: Über die Verschiedenheit des maschinellen Sprachbaues. In: Norbert Bolz; Friedrich A. Kittler; Christoph Tholen (Hrsg.): Computer als Medium. Wilhelm Fink Verlag München, 1994

Pflüger (97), Jörg: Distributed Intelligence Agencies. In: Martin Warnke; Wolfgang Coy; Georg Christoph Tholen: HyperKult. Stroemfeld/Nexus, 1997

Planck, Max: Vom Wesen der Willensfreiheit und andere Vorträge. Frankfurt am Main: Fischer Taschenbuchverlag 1991

Popper, Karl R.; Eccles, John C.: Das Ich und sein Gehirn. München, Zürich: Piper, 1982

Popper, Karl R.: Alles Leben ist Problemlösen. München: Piper, 1996

Postman, Neil: Das Technopol. Die Macht der Technologien und die Entmündigung der Gesellschaft. Frankfurt am Main: Fischer Verlag, 1992

Prigogine, Ilya: Vom Sein zum Werden. Zeit und Komplexität in den Naturwissenschaften. München: Piper 1979

Prigogine, Ilya; Stenger, Isabelle: Dialog mit der Natur. München: Piper, 1981

Rechenberg, Peter: Was ist Informatik? Eine allgemeinverständliche Einführung. München, Wien: Carl Hanser Verlag, 1991

Rechenberg, Peter; Pomberger, Gustav (Hrsg.): Informatik-Handbuch. Müchen; Wien: Hanser, 1999

Reischuk, Karl Rüdiger: Einführung in die Komplexitätstheorie. Stuttgart: Teubener, 1990

Reischuk, Rüdiger: Zeit und Raum in Rechnernetzen. In: Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer-Verlag, 1996

Reisig, Wolfgang: Petrinetze. Berlin u.a.: Springer, 1990;

Rembold, Ulrich; Levi, Paul: Einführung in die Informatik für Naturwissenschaftler und Ingenieure. München, Wien: Carl Hanser Verlag, 1999

Riedl, Rupert: Biologie der Erkenntnis. Hamburg: Parey, 1980

Russell, Stuart; Norvig, Peter: Artificial Intelligence. A Modern Approach. Prentice-Hall, Inc. Englewood Cliffs, New Jersey, 1995

Sander, Peter; Wollfried Stucky; Rudolf Herschel: Automaten Sprachen Berechenbarkeit. Stuttgart:Teubner, 1992

Schefe (87), Peter: Informatik - Eine konstruktive Einführung. Mannheim, Wien, Zürich: BI-Wiss.-Verl.,1987

Schefe (87a), Peter: Künstliche Intelligenz - Überblick und Grundlagen. Mannheim, Wien , Zürich: Wissenschaftsverlag, 1987

Schefe, Peter; Boden, Margaret A. (Hrsg.): Informatik und Philosophie. Mannheim u.a.: B.I. Wissenschaftsverlag, 1993

Schefe, Peter: Softwaretechnik und Erkenntnistheorie. Informatik Spektrum, **22**, H.2 1999, S.122-135

Schiffmann, Wolfram; Schmitz, Robert: Technische Informatik. Band 1: Grundlagen der digitalen Elektronik; Band 2: Grundlagen der Computertechnik. Berlin, Heidelberg: Springer-Verlag, 1992; Neuauflage 1999

Schmitt, Franz Josef: Praxis des Compilerbaus. München, Wien: Carl Hanser Verlag, 1992

Schneider, Hans-Jochen (Hrsg.): Lexikon Informatik und Datenverarbeitung. München u.a.: Oldenbourg, 1998

Schnorr, Claus Peter: Rekursive Funktionen und ihre Komplexität. Stuttgart: B.G.Teubner, 1974

Schöning, Uwe: Logik für Informatiker; 172 S. Mannheim, Wien, Zürich: Wissenschaftsverlag, 1989; Neuauflage 2000

Schöning, Uwe: Theoretische Informatik - kurzgefaßt; 188 S. Heidelberg u.a.: Spektrum Akademischer Verlag, 1995; Neuauflage 1999

Schreiber, Peter: Grundlagen der Mathematik. Berlin: VEB Deutscher Verlag der Wissenschaften, 1984

Schwarz, Wolfgang: Analogprogrammierung. Theorie und Praxis des Programmierens für Analogrechner. Leipzig: VEB Fachbuchverlag, 1971

Stetter, Franz: Grundbegriffe der Informatik. Berlin, Heidelberg: Springer, 1988

Stöcker, Horst (Hrsg.): Taschenbuch mathematischer Formeln und moderner Verfahren. Thun, Frankfurt am Main: Deutsch, 1995

Störig, Hans Joachim: Kleine Weltgeschichte der Philosophie. Frankfurt am Main: Fischer, 1989

Stoyan, Herbert: Programmiermethoden der Künstlichen Intelligenz. Berlin u.a.: Springer, Band 1 1988, Band 2 1991

Tanenbaum, Andrew S.: Moderne Betriebssysteme. München, Wien: Hanser; London: Prentice-Hall Internat., 1994

Tanenbaum, Andrew S.: Computer-Netzwerke. München u.a.: Prentice Hall, 1998

Tapscott, Don: Die digitale Revolution: Verheißungen einer vernetzten Welt - Folgen für Wirtschaft, Management und Gesellschaft. Wiesbaden: Gabler, 1996

Turing, Alan M.: Computing Machinery. Mind **59** (1950), S.433-460

Vollmer, Gerhard: Was können wir wissen? Stuttgart: Hirzel Verlag, 1988

Völz, Horst: Information. Berlin: Akademie-Verlag, 1982

Wedekind, Hartmut; Theo Härder: Datenbanksysteme, Band 2. Mannheim, Wien, Zürich: B.I. Wissenschaftsverlag, 1989

Wedekind, Hartmut: Datenbanksysteme, Band 1. Mannheim: B.I. Wissenschaftsverlag, 1991

Wegener, Ingo (Hrsg.): Highlights aus der Informatik. Berlin, Heidelberg: Springer, 1996

Weizenbaum, Joseph: Eliza. Communication of the ACM 9, 1966, S. 36-45

Weizenbaum, Joseph; Haeffner, Klaus: Sind Computer die besseren Menschen? Ein Streitgespräch. München, Zürich: Piper, 1992

Weinzenbaum, Joseph: Wer erfindet die Computermythen? Freiburg u.a.: Herder, 1993

Wendt, Siegfried: Nichtphysikalische Grundlagen der Informationstechnik. Interpretierte Formalismen. Berlin u.a.: Springer, 1989

Werner, Dieter (Hrsg.): Taschenbuch der Informatik. Leipzig: Fachbuchverlag, 1995; Neuauflage 2000

Wiener, Norbert: Kybernetik, Regelung und Nachrichtenübertragung im Lebewesen und in der Maschine. Düsseldorf: Econ, 1963

Winograd, Terry; Flores, Fernando: Erkenntnis Maschinen Verstehen. Berlin: Rotbuchverlag, 1989

Zadeh, Lofti A.: Fuzzy Sets. Information and Control, Vol 8, 1965, S. 338 - 353

Zemanek, Heinz: Das geistige Umfeld der Informatik. Berlin u.a.: Springer, 1992

Namen- und Sachverzeichnis

A

Abbildung, 114, 577
isomorphe, 210
Abbildung i.e.S., 577
Abbildung i.w.S., 577
Abbildungstafel, 114
Abbruchkriterium, 122
Abbruchprädikat, 120, 122
Abstraktion, 55
generalisierende, 56, 57, 410
idealisierende, 54, 55
klassifizierende, 56, 410
komponierende, 56, 57, 409, 410
prozedurale, 285, 312, 400, 410, 412, 486, 589
Addierer
sequenzieller, 184, 244
Adressbus, 232
Adresse, 255
relative, 302, 438
adressierbarer Speicher, 245
Adressiermatrix, 223, 224
Adressraum, 465
Adressrechnung, 438
Agent, 424
Ähnlichkeit, 394
Akkumulator, 237, 259, 260, 261, 262
Aktion, 75, 84, 259, 577
Aktionsfolge, 268, 577
Aktionsfolgegraph, 280, 577
Aktionsfolgeparadigma, 448
Aktionsfolgeplan, 238, 254, 267, 278, 280, 505, 577
Aktionsfolgeprogramm, 254, 275, 280, 577
Aktionsfolgeprogrammierung, 255, 260, 274
Aktionsschritt, 255, 264
Algebra, 155
boolesche, 172, 210, 212, 283
Algorithmentheorie, 125, 126, 127, 129
Algorithmus, 75, 83, 84, 125, 274, 578
imperativer, 75, 84, 134, 240, 256, 259, 406, 486, 578
kanonischer, 314
nichtdeterministischer, 290, 314, 360
Allmenge, 212
Allquantor, 329
Alphabetzeichen, 47
Alternativmasche, 91, 505
ALU, 76, 85, 237, 253, 256, 262, 270, 578
ALU-Array, 429, 440
ALU-Komposit, 440
analog, 33
Analog-digital-Konverter, 33, 167, 173, 197
Analogrechner, 28, 38, 39
Analyse
lexikale, 301, 324, 355, 586
semantische, 363
syntaktische, 324, 355, 360
Analysis, 78
Analytik, 209
Anfragesprache, 339
Antinomie, 71
Antireduktionismus, 569
physikalischer, 570
Antivalenz, 183
Anweisung, 75, 297, 577
Anwendungsprogramm, 462, 481, 578
Anwendungsprozess, 578
Anwendungsregime, 481
APL, 412
applikativ, 485
Äquivalenz, 183
Arbeitsoperation, 448

Arbeitsoperator, 90, 444, 505, 585
Arbeitsspeicher, 253
arbiträr, 69, 578
Arbitrarität, 69, 578
ARISTOTELES, 201, 207, 208, 353
arithmetisch-logische Einheit, 253, 262
Array, 509
  systolisches, 429, 443
Artikulieren, 11, 21
Assembler, 289, 300
Assemblerprogramm, 300, 578
Assemblersprache, 49
Assoziation, 60, 75, 77, 327, 342, 549, 550, 578
Assoziativspeicher, 246, 343
Auftragssprache, 354, 411
Ausdruck, 359, 366
Aussage, 43, 75, 119, 579
  nichtentscheidbare, 72
Aussageform, 579
Aussagenalgebra, 181, 210
Aussagenkalkül, 181, 210
Aussagenlogik, 181, 210
Aussagesatz, 19
Auswerten, 489
Auswertungsstrategie, 317
Automat, 235, 367, 370
  abstrakter, 106, 107
  akzeptierender, 368, 370
  autonomer, 107
  endlicher, 107, 235, 254, 367
  nichtdeterministischer, 368
Automatentafel, 107
Automatentheorie, 108, 367
Autonomieprinzip, 573
Axiomatisierung, 54, 161
Axiomensystem, 54, 161, 579

B

BABBAGE, C., 242
Backtracking, 315
Backus-Naur-Form, 48, 301
Bandbreite, 216
BARDEEN, J., 215
Basic, 487
Basiskalkül, 239, 284
Bausteinoperand, 90
Bausteinoperator, 89, 585
Bedeutung, 4
Bedeutungsfreiheit, 4
Befehl, 49, 75, 577
Befehlsformat, 257, 260
Befehlsregister, 237, 257
Befehlsrepertoire, 277
Befehlssatz, 277, 433
Befehlsweg, 481
Befehlszähler, 260, 262
Begriff, 43, 45, 55
  metasprachlicher, 49
Begriffsbildung, 45, 55, 408
Begriffsschrift, 213
Benzolring, 345
Berechenbarkeit, 524
  effektive, 126
Berechenbarkeits-Äquivalenzsatz, 248, 279
Berechnung, 34
  rekursive, 245, 590
Berechnungsaufwand, 524
Berechnungskomplexität, 524, 585
Beschreibbarkeit
  rekursive, 132
Beschreibung
  kausaldiskrete, 171
Beschreibungsraum, 351
Betrachtungsebene
  subsymbolische, 11
  symbolische, 11
Betriebsmittel, 103, 429, 438, 446, 461, 579
  geteiltes, 445
  verteiltes, 457
Betriebsmittelnutzung
  geteilte, 457, 470
Betriebsmittelzuweisung, 467
Betriebssystem, 429, 462, 481, 579

Betriebssystemkern, 429, 481
Bewusstsein, 4
Bezeichner, 300
Bezeichnerabgleich, 315
Bilderschrift, 377
Bildmerkmal, 548
Binärkanal, 61
Binärwortfunktion, 579
Binärwortoperator, 170
Binden, 302
  dynamisches, 303, 423, 581
  statisches, 303, 592
Binder, 302, 456
Bioinformatik, 579
Bit, 579
Bitkettenoperator, 170, 176
BOLTZMANN, L., 65, 529, 533
BOOLE, G., 172, 210, 213
boolesches Netz, 399, 406
Booten, 482
BorlandPascal, 499
Botschaft, 513
Bottom-up-Analyse, 361
BRATTAIN, W.H., 215
Brückenhypothese, 571
Buchstabenschrift, 377
Bus, 231
Busarbiter, 451
Bushierarchie, 450
Byte, 153, 580

C

Cache, 435
CANTOR, G., 210
CD-ROM, 200, 225, 580
CD-RW, 200
CHAGALL, M., 377
chaotisch, 566
Chip, 175, 215
CHOMSKY, N., 47, 366, 379
Chomsky-Hierarchie, 370
CHURCH, A., 126, 127, 135, 146, 151, 249, 319, 412
Church-Funktion, 127, 146
CHURCHsche These, 151, 156
Client, 476
Client-Server-Prinzip, 476, 477, 580
  nicht schützendes, 477
  schützendes, 477
Clos, 493
Code, 11, 13, 580
Codegenerator, 324, 357, 362
Codeumsetzer, 224, 234, 259
Codierbarkeit
  binär-statische, 162
Codieren, 11
  externes, 11, 21
  internes, 21
  primäres, 62
  sekundäres, 62
Codierer, 219
Codiermatrix, 202
Codierung, 13, 580
  binär-statische, 167, 170, 579
  dynamische, 169, 402, 532, 581
  externe, 403
  interne, 403
  statische, 169, 238, 402, 592
Codierungssatz, 63
Codierungstheorie , 44
CommonLisp, 293, 488
Compiler, 324, 363, 580
Compilerbau, 357
Computer
  personal, 203
  traditioneller, 173, 589
Computer-IV, 5, 580
Computerarray, 474
Computerschach, 373
Computersemantik, 44, 159, 303, 307, 580
Computersimulation, 307
Computervirus, 568
CPU, 432, 454, 467
CPU-Register, 435
CPU-Zustand, 465

Cray, 443
CUTLAND, N., 129
Cyberspace, 383

D

data hiding, 415
Datei, 342, 413
Daten, 580
Datenabstraktion, 400, 412
Datenbank, 342, 344
Datenbankbetriebssystem, 342, 344
Datenbanksystem, 335
Datenbasis, 342, 344
Datenflussgraph, 280, 580
Datenflussmaschine, 421
Datenflussparadigma, 448
Datenflussplan, 278, 580
  erweiterter, 505
Datenflussprogramm, 275, 280
Datenflussprogrammierung, 274
Datenkapsel, 415
Datensatz, 342, 413
Datenschutz, 477
Datentyp, 413
  abstrakter, 415
Datenübergabe, 277
Datenverarbeitung
  elektronische, 199
Deadlock, 472
Decodierer, 219
Decodiermatrix, 202
Deduktion, 75, 580
Defuzzifizieren, 542
Demultiplexer, 228
Denken, 11, 20, 403, 545, 557, 580
  deduktives, 207
  gestalthaftes, 404
  netzorientiertes, 404
  regelbasiertes, 158
  satzorientiertes, 404
  simulierbares, 558
Denkkalkül, 158, 323, 324, 353, 373, 581, 591
Denkobjekt, 43, 57, 60, 75, 342, 423, 549, 581
Denotator, 23
Dezentralisierung, 448
Diagnosesystem, 376
Dichotomie, 195
Dichteverteilung, 65
Dienst, 478
Dienstanbieter, 476, 478
Dienstleistung, 416, 476
  nicht schützende, 477
  schützende, 477
Dienstleistungsprozess, 476, 478
Dienstnutzer, 476, 478
Dienstnutzerprozess, 476, 478
Differenzengleichung, 442
Differenzenquotient, 36
Differenzial, 78
Differenzialgleichung, 37, 78
  partielle, 441, 442, 443, 473
Differenzialquotient, 31, 36, 78
  gewöhnlicher, 442
  partieller, 442
Differenzialrechnung, 78
Differenzieren, 37
digital, 33
Digitalisierung, 33
Digitalrechner, 28
Diodenmatrix, 202, 221
Direktive, 400, 416, 492
Direktzugriff, 245
Direktzugriffspeicher, 245
Disjunktion, 167, 181
Diskettenspeicher, 199
Diskretisieren, 33
Diskursbereich, 43, 323, 333
Dotieren, 215
DRAM, 246, 581
Drei-Adress-Maschine, 49, 238
Dualzahl, 62, 581
Durchschaubarkeit, 551
Durchschleppen, 273
Durchschnittsmenge, 211

DVD, 200
dynamisch stabil, 169
dynamischer RAM, 246

E

E-Mail, 478
E/A-Gerät, 433
ECKERT, J.P., 214, 256
EHRENSTEIN, W., 556
EIGEN, M., 16
Eigenzustand, 167, 191, 197
Ein-Bit-Speicher, 197
Einbitoperator, 172
Einprozessorrechner, 262, 263, 273, 283, 284
EINSTEIN, A., 352
elektronische Post, 478
Eliza, 375
Emergenz, 266, 523, 566
Empfängersemantik, 354
Endzustand, 367
ENIAC, 214
Entdeckung, 345
Entropie, 63, 529
  informatische, 65
  thermodynamische, 65
Entscheidbarkeit
  formale, 157
Entscheiden
  fallbasiertes, 391
  regelbasiertes, 391
Entscheidungsgehalt, 63, 64
Entscheidungstabelle, 235, 252, 394
Entschlüsseler, 219, 232
EPROM, 226, 227, 581
erben, 511
Erfahrung, 551
Erfahrungsregel, 390, 553
Erfinden, 345, 550
Erfüllungsfunktion, 541
Erfüllungsgrad , 541
Ergebnisfunktion, 107
Ergibtanweisung, 359
Ergibtgleichung, 119
Erhaltungssatz, 23
Erkennen, 75, 76, 581
Erkenntnis, 545
  physikalische, 553
Erkenntnisgewinnung, 75, 76, 531, 553
Erkennungssystem, 548
ESCHER, M.C., 73
Esperanto, 379
EUKLID, 54, 161
Evaluieren, 489
Evaluierung, 148
Evolution, 13, 47, 169, 407, 530
  codierende, 13, 580
  genetische, 13
  intellektuelle, 11, 13, 48, 580
  kosmische, 13, 580
  kulturelle, 1, 11, 13, 48, 207, 288, 580
  künstliche, 523
  programmiersprachliche, 399
Existenzquantor, 329
Expertensystem, 323, 335
Extensionalitätsprinzip, 4
externsemantische Dichte, 409

F

Faktenwissen, 323, 329, 330
Faktor, 366
fallbasiert, 390
Feldeffekttransistor, 216
FELDENKRAIS, M., 19
Fensterprinzip, 410, 482
Fernziel der KI, 158
Ferritkernspeicher, 199
Fertigungsoperator, 99
Festkommadarstellung, 152
Festplatte, 437
FIFO-Prinzip, 243
Findigkeit, 349
Firmware, 236, 271, 581
First-Order Logic, 328
Fixpunkt, 191
Flächentransistor, 215

Flaschenhals
  von-neumannscher, 237, 274, 434
Flipflop, 167, 190, 197
Floating-Gate-MOS-FET, 226
Flüssigkeitsströmung, 530
Flussknoten, 91, 505, 582
FLYNN, M.J., 439
Folge, 582
  abzählbar unendliche, 577
Folgefunktion, 106, 107, 145
Folgeschaltung, 247
Formalisierungsgrad, 584
Formelmanipulation, 290, 306, 334
Formelmanipulator, 131, 316
FREGE, F.L.G., 213
FREUD, S., 70
Fulguration, 566
Fulleren, 350
Funktion, 114, 577
  berechenbare, 579
  boolesche, 167, 172, 179
  charakteristische, 121, 139
  effektiv berechenbare, 126, 156, 249
  elementare boolesche, 172
  geschachtelte, 133
  imperativ berechenbare, 279
  KR-berechenbare, 247, 248
  markovberechenbare, 131
  nichtberechenbare, 587
  partiell-rekursive, 142
  partielle, 128
  primitiv-rekursive, 139
  rekursive, 6, 126, 131, 132, 140, 151, 156, 167, 179, 188, 194, 238, 248, 318, 590
  statisch berechenbare, 171, 238, 247, 248, 592
  totale, 128
  turingberechenbare, 129
  URM-berechenbare, 130, 279
Funktionsgenerator, 234
Funktionsgenerator-Prinzip, 269
Funktionstafel, 114
Fuzzifizierung, 538

G

Gabel, 91, 278, 582
GATES, B., 564
Gatter, 202
Gedächtnis, 97, 171
Gefühl, 379
Gegenläufigkeitsphänomen, 75, 76, 79, 287
Gehirnhälfte, 404
Generalisieren, 423
Generalisierung, 56
Generalisierungsgrad, 414
Gerät
  peripheres, 467
Gestalt, 398
gestalthaft, 546
Gestaltpsychologie, 403
Gewissen, 567, 573
Gitterspannung, 214
Gleichung
  relationale, 119, 442
Gleitkommadarstellung, 152
Gleitkommaoperation, 432
GÖDEL, K., 69, 72, 132, 157, 571
Gödelisierung, 153
Gödelnummer, 153
GOETHE, J.W. von, 380
Grammatik, 45, 55, 357, 365, 370, 408
  generative, 324, 358
  kontextabhängige, 366
  kontextfreie, 366, 370
  kontextsensitive, 366, 370, 371
  reguläre, 366, 370
Grenzwert, 31
Grundwissen, 351

H

Halbaddierer, 185
Halbkommutator, 230, 245, 263

Halbordnung, 101
Haltefunktion, 123
Halteproblem, 123, 241
Hardware, 5, 227, 582
  periphere, 433, 481
  zentrale, 433, 481
Hardware-Software-Schnittstelle, 276
Hauptspeicher, 237, 253, 262
Havard-Computer, 273
Havardprinzip, 435
Heap-Speicherung, 245
HERBRAND, J., 132
HERTZ, H., 112
HILBERT, D., 25
Hintergrundprogramm, 469
Hollerithmaschine, 174
Hornklausel, 331, 339, 496
Human-IV, 5, 582
Humaninformatik, 30, 582
Humansemantik, 330, 364
Humansprache, 399, 582

I

IBN MUSA AL-CHWARISMI, 83
Idealisieren, 54
Idem, 3, 4, 11, 20, 43, 46, 583
  objektiviertes, 55, 587
  universelles, 53
Idemobjektivierung, 45, 46, 51, 52, 383, 408, 583
Idemschärfung, 53, 55
Identifikator, 300, 359
Identifizieren, 57
Identitäsoperator, 204
if-Funktion, 149
imperatives Paradigma, 274
Implikation, 328, 495
Impulsgenerator, 250
Impulsneuron, 531
Individuenvariable, 118
Inferenz, 323, 330
Inferenzbaum, 332
Inferenzieralgorithmus, 336
Inferenzieren, 323, 330, 335, 583
Inferenzierer
  vollständiger, 336
Infinitesimalrechnung, 78, 99
Informatik, 6, 12, 27, 583
  biologische, 30, 564, 579
  technische, 30, 564, 593
  traditionelle, 173
Information, 11, 14, 22, 583
  artikulierte, 578
  genetische, 170
  kulturelle, 169
  nichtartikulierte, 23
  syntaktische, 60, 64
  uneigentliche, 23, 583
Informationsentropie, 65
Informationsgesellschaft, 373, 384, 564
Informationssystem, 473
Informationstheorie, 44, 62, 524
Informationsübertragung, 23
Informationsverarbeitung, 347
Inkrementieren
  iteratives, 84
Instanz, 511
Instanzieren, 22, 58, 208
  materielles, 22
Instinkt, 551
Integral, 39
Integrieren, 39
Intelligenz, 6, 11, 29, 551, 584
  assoziative, 77
  bewusste, 77
  deduktive, 77, 207, 287
  induktive, 555
  intuitive, 77, 287, 345
  künstliche, 6, 75, 76, 82, 207, 283
  natürliche, 75, 76, 283, 285
  produktive, 29
  reproduktive, 29
  unbewusste, 77
Intelligenztransfer, 384
Interface, 257
Interncodierung, 584

Internet, 478
Internrealem, 113
internsemantische Dichte, 409
Interpretation, 43, 52, 584
  externe, 43, 53, 581
  formale, 53, 156, 582
Interpretator, 94
Interpreter, 324, 363, 584
Interpretieren, 21, 363
Interpretierer, 584
Intuition, 75, 77, 79, 80, 315, 389, 391, 395, 550, 551, 584
  nichtreduzible, 551
  reduzible, 324, 351, 550, 551
  scheinbare, 389
Irregularität, 529, 560
Isomorphie, 210
Iteration, 74, 238, 251
  nichtterminierende, 238
  rekursive, 136, 293, 490
Iterationszahl, 251
iterative
  Berechnung, 584
Iterator, 136
IV-System, 583

J

JUNG C.G., 574

K

Kalkül, 43, 52, 76, 155, 156, 157, 284, 288, 323, 584
  axiomatisierter, 54, 579
  boolescher, 159, 327
  universeller, 156, 157
Kalkülisieren, 43, 158
Kalkülisierung, 304, 584
Kalkülisierungsgrad, 419, 420, 530, 531, 584
Kalkülsemantik, 330, 341
Kalkülsprache, 53, 156, 284, 585
Kalkültransformationssatz, 157
Kalkültransformation, 154, 156
Kanalkapazität, 61
Kapselung, 400, 414
Kassettenspeicher, 199
kausaldiskret, 36, 99
kausalkontinuierlich, 36
KEKULÉ, A., 345, 352
Kellerautomat, 145, 368, 369, 370
Kellerspeicher, 144, 245, 368
KEPLER, J., 36, 352, 554
KI
  alternative, 534
  traditionelle, 534
Klasse, 56, 57, 208, 413, 585
  unscharfe, 536
Klassieren, 57, 394, 585
Klassifizieren, 33, 56, 394, 585
Klassifizierung, 377
Klausel, 496
KLEENE, S.C., 132
Kognitionswissenschaft, 6
Kombinationsschaltung, 167, 172, 188, 202, 219, 222, 225, 585
  disjunktive, 187
  steuerbare, 239, 253
Kommando, 482
Kommandointerpreter, 482
Kommunikation
  direkte, 581
  indirekte, 583
Kommunikationsmittel
  bidirektionales, 401
  unidirektionales, 401
Kommunikationsstruktur, 229, 263, 451
Kommutator, 231, 263
Komplementmenge, 211
Komplex, 398, 519, 520, 521, 585
  künstlicher, 523
  natürlicher, 523
  psychischer, 521
Komplexität, 266, 520, 585
  beherrschbare, 522
  durchschaubare, 522

exponentielle, 388, 519, 525
kausale, 405, 522
lineare, 525
logische, 404, 519, 522, 523, 525
nichtlineare, 519, 530, 532, 585
physische, 519, 522, 525
polynomiale, 519, 550, 551
räumliche, 404, 522
strukturelle, 519, 521, 532, 585
undurchschaubare, 522
Komplexitätsklasse, 519, 525
Komplexitätstheorie, 125, 519, 524
Komplexverbindung, 521
Komponieren
hierarchisches, 47, 582
Komponierungsmittel
rekursives , 132
Komponierungsregel, 132
Komposit, 47, 582
Kompositflussknoten, 229
Kompositoperand, 90
Kompositoperation, 89
Kompositoperator, 89, 585
Kompositprozess, 481
Kompositschalter, 205
Kompositzeichen, 47
Konditionierung, 253
Konflikt, 100
Konjunktion, 167, 181
Konkatenation, 292
Konklusion, 328, 495
Konsensfindung
emotionale, 373, 379
rationale, 373, 379
Konsistenz, 333, 336
Kontext, 361, 366
Konvergenz
begriffliche, 155, 212, 348, 423
Konvergenzprinzip, 155, 212, 371
Konversation, 375
Konverter, 33
Konvertierung, 383
analog-digitale, 33
digital-analoge, 34
KOPERNIKUS, N., 352
Kopiergabel, 90, 585
Kopplungstabelle, 507
Koprozessor, 439
KR-Funktion, 247
KR-Netz, 247, 256, 429, 444, 586
KR-Operator, 247, 250
gesteuerter, 253
universeller, 253
Kreativität, 75, 349
Kreuzschienenverteiler, 232
kritischer Programmabschnitt, 471
kritisches Zeitintervall, 471
Kurzzeitgedächtnis, 327, 406
Kybernetik, 41

L

Lader, 302
Lambda-Eliminierung, 148
Lambda-Kalkül, 146, 256
Laufanweisung, 297
Laufvariable, 298
Laufzeit, 464
Lautsprache, 401, 583
Lebenszyklus, 516
LEIBNIZ, G.W., 209
Leit-Operator, 447
Leitermatrix, 185
Leitung-Wort-Zuordner, 202, 220, 221
Lernen, 178
Lesbarkeit, 310
Lexem, 301, 324, 358, 586
Lichtleiter, 61, 234
LIFO-Prinzip, 244
Ligand, 521
Linearisierung, 268
linguistische Regel, 535
Linie, 32
Lisp, 292, 412, 488
Listennotation, 147, 292, 489
funktionale, 494
logische, 494

Logik, 207, 208, 209
LORENZ, K., 566
Lügnerantinomie, 72

M

Magnettrommel, 199
Makromodell, 522
Marke, 102
Markenplatz, 505, 506
MARKOV, A.A., 130, 319
Markovalgorithmus, 130, 314, 331, 357, 367, 393, 394, 395
Markovfunktion, 131
Masche, 586
    starre, 91, 437, 516
    steuerbare, 91
Maschinenbefehl, 49, 280, 586
Maschinenebene, 284, 432, 586
Maschinenkalkül, 159, 327, 353
Maschinenprogramm, 49, 159, 237, 238, 257, 268, 280, 586
Maschinensemantik, 159, 303, 330, 341, 364, 586
Maschinensprache, 159, 257, 266, 269, 272, 284, 586
Maskentechnik, 215
Massenspeicher, 200, 217
Mathematiksystem, 474
Matrixsteuerwerk, 235, 237, 269, 433
MAUCHLY, J.W., 214, 256
Mausklick, 410
MAXWELL, J., 212
McCARTHY, J., 292, 488
Megaflops, 432
Mehrcomputersystem, 449
Mehrfachweiche, 228
Mehrkanalkommutator, 232
Mehrprozessorrechner, 429, 439, 449, 450, 473
Menge, 586
    abzählbar unendliche, 35, 115, 577
    überabzählbar unendliche, 35
    unbeschränkte, 116, 593
    unscharfe, 536
Mengenalgebra, 210
Mengenlehre, 210
Menü, 410
Menüsprache, 411
Merkmal, 56, 549, 552
Merkmalsbildung, 57
Merkmalswert, 56, 549
Message, 513
Messen, 34
Messgröße, 536
Metaaussage, 49
Metaintelligenz, 75, 82, 586
Metamodellierung, 29
Metaregel, 366, 553
Metasprache, 49, 356, 365, 587
metasprachliche Variable, 409
Methode, 416
Mikroelektronik, 205
Mikromodell, 522
Mikroprozessor, 271
Mikrozustand, 65
MIMD-Computer, 440
Miniaturisierung, 430
Minimalisieren, 140
Minimalisierung, 298
MINSKY, M., 557
MISD-Computer, 440
Modell, 6, 587
    agierendes, 26
    aktives, 26
    analoges, 26, 31, 578
    analytisches, 313
    digitales, 31
    exaktes, 26
    formalisiertes, 27
    metrisches, 26
    nichtagierendes, 26
    nichtexaktes, 26
    nichtformalisiertes, 27
    nichtmetrisches, 26
    nichtsprachliches, 26, 31, 578
    passives, 26

sprachliches, 19, 31, 43, 400, 418, 591
Modellieren
codierendes, 591
sprachliches, 400, 591
Modellierparadigma, 276
Modellierung
analytische, 307
kausale, 403
mathematische, 338
numerische, 307
Modellierungsparadigma, 274
Modul, 415
morgansche Regel, 182, 223
Morphem, 301
Morphologie, 45
MOS-FET, 215
Multimedia, 385
Multiplexer, 228
Multitasking, 469
Multitaskregime, 469
Mutation, 47

N

Nachrichtenkanal, 61
Nahwirkung, 441
NAND-Flipflop, 198
Naturwissenschaft, 2
Navier-Stokes-Gleichungen, 530
nebenläufig, 437, 587
Negation, 167, 181
Negator, 204
Netz
boolesches, 172, 189, 399, 406, 580
boolesches zirkelfreies, 189
boolesches zirkuläres, 184, 189, 190, 191
neuronales, 176, 177, 178, 189, 397, 399, 532
neuronales zirkelfreies, 189, 193
neuronales zirkuläres, 189, 195
zirkelfreies, 172, 176, 594
zirkurläres, 595
Netzklassen, 189
Netzmethode, 516
Netzparadigma, 274, 276, 448, 587
NEUMANN, J. VON, 74, 214, 256, 273
Neuro-Fuzzy-System, 544
Neurocomputer, 168, 173, 196, 197, 238, 532, 587
Neuroinformatik, 3, 16, 173
Neuron
künstliches, 167, 176, 177
neuronales Netz, 399, 531
Neuronennetz, 406
NEWTON, I., 35, 53, 78, 161, 178
Nichtlinearität, 560
nichtlinerare Dynmik, 529
Nichtlinerarität, 529
Nichtterminalsymbol, 365, 366
NIEVERGELT, J., 385, 397
Normalform
kanonische disjunktive, 183
kanonische kunjunktive, 187
Notation
funktionale, 135
imperative, 135
NP-Problem, 527
NP-vollständig, 527
Nur-Lese-Speicher, 224
Nutzersemantik, 44, 160, 303, 307, 330, 340, 341, 354, 420, 587

O

Objekt, 312, 400, 415, 422, 480, 510, 587
informatisches, 416, 423
umgangssprachlich es, 416, 423
Objekterkennung, 549
Objektivierung
semantische, 51, 383
Objektklasse, 511
Objektsprache, 49, 365, 587
Objekttyp, 511
OC-Entschlüsseler, 270
ohmsches Gesetz, 203
Operand

öffentlicher, 510
privater, 510
Operandenfluss, 90
Operandenflussgraph, 94, 280, 588
Operandenflussplan, 94, 248, 254, 280, 588
Operandenflussprogramm, 266, 280
Operation, 114, 588
begriffsbildende, 57, 397, 553
boolesche, 167
interiorisierte, 109
Operationsausführung, 114, 588
Operationscode, 49, 255
Operationsprinzip
von-neumannsches, 255
Operationsrepertoire, 257, 270
Operationstafel, 114
Operator, 88, 588
boolescher, 172, 580
deterministischer, 97
elementarer, 88, 109, 167, 171
elementarer boolescher, 167, 172
informationeller, 583
interpretierender, 94
nichtdeterministischer, 97
realer, 168, 588
sprachlicher, 114, 588
steuerbarer, 96
stochastischer, 97
Operatorabstraktion, 90, 588
Operatorenhierarchie, 88, 93, 109, 170
Operatorennetz, 88, 274, 588
wohlstrukturiertes, 141, 248
Ordnung
vollständige, 101
Organisationsprogramm, 429, 461, 481, 588

P

Pädikatenkalkül erster Ordnung, 328
Paging, 436
PAP, 267, 299
Paradigma
imperatives, 274, 276, 448, 583
objektorientiertes, 209
raum-zeitliches, 274
rein zeitliches, 274
Paradoxon der KI, 287
Paralleladdierer, 244
Parallelisierung, 429, 450
Parallelität, 439
Parameter
aktueller, 487, 497, 510
formaler, 487, 497, 510
Parameterübergabe, 302, 487
Parser, 324, 357, 358, 361, 369
Pascal, 413, 487
PAWLOV, I.P., 21, 390
PC, 203
PENROSE, R., 571
peripheres Steuerprogramm, 481
Peripherie, 433
Petrinetz, 101, 505
gesteuertes, 506
steuerbares, 505
Phantasie, 75, 349, 395
Pipeline, 437
Pipeline-Rechner, 443
Pipelinetiefe, 438
Pipelining, 429, 437
PLA, 227
PLANCK, M., 1, 7, 352, 557, 563, 573
Plansprache, 379
Platz, 102
Plotter, 234
Pointer, 504, 514
POPPER, K., 16, 45, 51, 571
Prädikat, 117, 118, 208, 588
entscheidbares, 119
unscharfes, 534
Prädikatenkalkül, 160, 328
Prädikatenlogik , 328
Prädikatoperator, 122, 250
Präfixnotation, 147, 292, 489
Prämisse, 328, 495
Präzisieren, 208, 423

Problemgröße, 525
Produkt
  kartesisches, 97
Produktionsbetrieb, 531
Produktionsregel, 365, 376
Programm, 240
  funktionales, 292, 369, 489
  imperatives, 254, 256, 280, 294, 297, 486, 577
  interncodiertes, 584
  ladbares, 301, 302, 362, 455
  logisches, 494
  nebenläufiges, 470
  objektorientiertes, 492
  reentrantes, 471
Programmablaufplan, 267, 280, 299, 588
Programmformat, 257
Programmierbarkeit, 159, 240
Programmieren, 240
  direktives, 416
  funktionales, 369
  logisches, 339
Programmierparadigma, 276, 277, 412
  datenflussorientiertes, 421
  funktionales, 399, 421
  imperatives, 421
  logisches, 399, 421
  objektorientiertes, 400, 421, 422
Programmiersprache, 159, 399, 429
  höhere, 455
Programmierstil, 310, 485
Programmierung
  logische, 160
  strukturierte, 311
Programmierwerkzeug, 513
Programmobjekt, 588
Prolog, 334, 339, 341
PROM, 226, 227
Protokoll, 479
Prozedur, 300, 593
Prozess, 114, 429, 446, 462, 465, 466, 589
  kausaldiskreter, 167
  kausalkontinuierlicher, 167
  nebenläufiger, 100
  paralleler, 100
  privilegierter, 467
  sequenzieller, 196
Prozessbeschreibung
  ereignisorientierte, 98
  kausaldiskrete, 98, 585
  kausalkontinuierliche, 585
  raum-zeitliche, 108
  rein zeitliche, 108
Prozesshierarchie, 466
Prozesskommunikation
  direkte, 469
  indirekte, 475
Prozesskommutator, 483
Prozesskomponierung, 483
Prozessor, 94, 237, 253, 256, 370, 589
Prozessor-Speicher-Netz, 444, 589
Prozessorcomputer, 173, 197, 238, 249, 589
Prozessorebene, 432, 589
Prozessorinformatik, 3
Prozessorprinzip, 5
Prozessorprogramm, 589
Prozessorrechner, 6
Prozessorsprache, 257, 589
Prozesszirkularität, 73, 74
PS-Netz, 429, 444, 472, 589
PUSCHKIN, A., 380

Q

Quantencomputer, 111
Quanteninformatik, 3, 111
Quasiparallelität, 469
Quellprogramm, 300
Quellsprache, 300, 324, 358
Querverweis, 8

R

Rahmendiagramm, 446
RALU, 237, 256, 262, 263, 589
RAM, 202, 245, 589
Rasterung, 382
Rätselraten, 552
Read Only Memory, 224
Realem, 3, 11, 20, 46, 589
Realisierbarkeitsprinzip, 6, 589
Realität
  virtuelle, 383
Rechnen, 75, 76, 88, 590
  analoges, 38
  analytisches, 306, 313, 331, 357, 394, 395, 578
  numerisches, 306, 587
Rechner
  elektronischer, 203
  programmierbarer, 175
  universeller, 88, 178
Rechnerarchitektur, 429, 431
Rechnergeneration
  dritte, 202
  erste, 201
  fünfte, 215
  zweite, 201
Rechnernetz, 229, 429, 451, 474
Rechnerwort, 257
Rechnerwortlänge, 243
Reduktion, 560
  algorithmische, 573
  physikalische, 573
Reduktionismus, 27, 569, 571
  algorithmischer, 570
  physikalischer, 569
Redundanz, 62, 64
Reduzierbarkeit
  algorithmische, 570
  physikalische, 570
Referenzieren, 71, 590
Reflex
  bedingter, 390
Regel, 235
  morgansche, 223
Regelungstechnik, 41
Regelwissen, 323, 329, 330
Register, 243
Registermaschine, 267, 269, 277
Registertransfer, 266
Rekursion, 132
rekursive Funktion, 127
rekursiver Abstieg, 498
Relais-Mechanismus, 201
Resolution, 338
Resolutionsverfahren, 337
Resolvente, 338
Ressource, 103, 590
Ringbus, 233
RISC-Prozessor, 433
Röhren-Mechanismus, 201
ROM, 203, 227, 590
  elektronischer, 224
ROM-Computer, 458
ROM-Hierarchie, 253, 458
Rückkopplung, 92, 260
Rückkopplungsschleife, 590
Rückwärtsinferenz, 333
Rückwärtsverkettung, 333
Rufweg, 481, 482

S

Sammelweiche, 91, 228, 505, 590
Satz i.w.S., 590
Satzglied, 48
Satzlehre, 357
Satzparadigma, 448, 590
Scanner, 324, 357, 361
Schach, 524, 546, 547
Schachcomputer, 473, 519
Schachintelligenz, 386, 547
Schachteldiagramm, 446
Schaltalgebra, 212
Schalter, 159
Schaltermechnanismus, 201
Schalternetz, 205, 221, 233

zirkelfreies, 202
Schaltkreis, 213
Schaltung
logische, 201, 207
rein elektronische, 199
Schaltungskomplexität, 525
SCHEFE, P., 418
Scheibenspeicher, 437
Schichtenarchitektur, 88
Schieberegister, 244
Schleife, 92, 590
Schlüsselwort, 343, 359
Schlussfolgern, 75, 76, 77, 208, 209, 323, 357, 590
Schlussregel, 77
Schnittpunktoperator, 185, 186, 221, 224
Schnittstelle, 227, 257
SCHOTTKY, W., 214
Schriftsprache, 401, 583
Schwellenelement, 197
Schwellenfunktion, 173, 538
Schwellenoperator, 33, 110, 167, 529, 590
mehrstelliger, 173
Schwellenwert, 110, 173, 529
Seite, 436
Seiteneffekt, 278, 312, 469
Selbstorganisation, 530
Selektor, 135
starrer, 135
steuerbarer, 135
Semantik, 43, 51, 52, 591
formale, 55
emotionale, 379
externe, 44, 52, 55, 284, 303, 323, 330, 341, 416, 417, 591
formale, 43, 53, 156, 304, 323, 330, 417, 591
interne, 44, 52, 55, 303, 330, 417, 532, 584
maschineninterne, 159
Semantikproblem, 52, 160, 304, 330, 373, 340, 364, 416, 593
semantische Anbindung, 416
semantische Dichte, 55, 61, 399, 408, 591
semantische Lücke, 52, 303, 399, 420, 591
semantische Objektivierung, 51, 55, 378
semantische Spezialisierung, 373, 377, 378
semantische Verarmung, 373, 376, 378
semantische Verdichtung, 400
semantischer Konsens, 378
Semiotik, 29
Sequenz, 133
Sequenzierer, 100
Server, 476
Serverprozess, 477
SHAKESPEARE, W., 380
SHANNON, C., 62, 212, 387
SHOCKLEY, W.B., 215
Signal, 508
Signalplatz, 508
Signalweg, 446
Signalwegegraph, 446
Silo-Prinzip, 243
Silospeicher, 245
SIMD-Computer, 440, 443
SISD-Computer, 440
Software, 5, 227, 591
Software-Entwicklungssystem, 485
Softwarehierarchie, 284
Sonderzeichen, 153
Spaltegabel, 91, 591
Speicher, 241
boolescher, 192, 580
dynamischer, 515
elektronischer, 199
magnetischer, 199
neuronaler, 196, 587
optischer, 199, 200
privater, 514, 515

virtueller, 437
Speicherhierarchie, 433, 434
Speichermatrix, 223, 224
Speicherperipherie, 437
Speicherplatz, 254
Speicherumgebung, 465
Speicherung
adressierte, 219, 577
strukturelle, 202, 219, 592
verteilte, 254
zentrale, 254
Speicherzustand
partieller, 466
Spiking neuron, 531
Spitzensymbol, 358
Spitzentransistor, 215
Sprache, 44, 357, 358, 370, 400, 591
algorithmische, 125, 154, 155, 156
auditive, 401, 402, 583
deklarative, 339
ebene, 410
eindimensionale, 402
formale, 52, 125, 156, 367, 582
funktionale, 134
imperative, 256, 294
implementierte, 583
kontextabhängige, 366
kontextfreie, 366
kontextfreie, 369, 370
kontextsensitive, 366, 370
lineare, 399, 402
logische, 412
natürliche, 19
nichtimperative, 134
objektorientierte, 416
prozedurale, 339
reguläre, 370
visuelle, 401, 583
zweidimensionale, 131, 410
Sprachevolution, 412
Sprachparadigma, 421
Sprachwissenschaft, 29
Sprachzentrum, 403, 404
Sprechen, 11, 20, 403
Sprung
bedingter, 267
Sprungadresse, 268
Sprungbefehl, 238, 267, 481
bedingter, 268, 296
Stack, 144, 245
Standardisierung, 377, 383
Stapel-Prinzip, 244
Stapelspeicher, 144, 245
Startadresse, 260
Startsymbol, 365
statisch stabil, 169
Steckkarte, 271
Stellgröße, 536
STEP-UNTIL-Anweisung, 297
Steuerautomat, 236
Steuerbus, 232
Steuerfluss, 268, 577
Steuerflussplan, 280
Steuerhierarchie, 252, 270, 446
Steuerimpuls, 235
Steuermatrix, 235, 269
Steueroperation, 448
Steueroperator, 90, 234, 248, 444, 585
interpretierender, 250
Steuerprädikat, 122
Steuerprogramm
peripheres, 462, 588
Steuerschleife
zentrale, 255, 363
Steuersignal, 90, 198, 234
Steuertransition, 505
Steuerung
dezentrale, 448
rückgekoppelte, 269
rückkopplungsfreie, 269
zentrale, 446
Steuerungsgraph, 446
Steuerwort, 235
Stichprobe, 535
Stirlingsche Formel, 66
Stoiker, 209

Stromschalter, 201
Struktur
  algebraische, 155
  lexikale, 358
  syntaktische, 358
Strukturgesetz, 196
Subroutine, 593
Substitution, 323, 331, 334
  einstellige, 133
  funktionale, 133
  mehrstellige, 133
Substitutionsfunktion, 148
Substitutionsregel, 130, 323, 358, 365
subsymbolische Ebene, 15, 545, 592
Suchargument, 246, 343
Suchbaum, 370
Suche, 345
Suchen
  regelbasiertes, 351
Suchgraph, 524
Suchproblem, 387
Suchraum, 348, 350, 351, 550
Suchwissen, 351
Syllogismus
  kategorialer, 208
Syllogistik, 201, 207, 208
Symbol, 11, 15, 53, 592
symbolische Ebene, 15, 545, 592
Symboltabelle, 301, 361, 592
Synchronisierer, 91, 100, 508
Syntax, 45, 48, 357, 592
Syntaxanalyse, 50
Syntaxbaum, 50, 301, 356, 360, 364
Syntaxregel, 43, 48, 357, 592
Syntaxvergleich, 323
System, 593
  abgeschlossenes, 65
  algorithmisches, 125, 127, 154, 155
  autonomes, 36
  homogenes, 522
  informationelles, 583
  interaktives, 364
  nichtautonomes, 36
  nichtlineares dynamisches, 519, 529
  verteiltes, 430, 472, 484
Systembeschreibung
  uniforme, 94, 593
Systemprogramm, 462, 593
Systemprozess, 481
Systemregime, 481
Systemruf, 482

T

Taktfrequenz, 272, 430, 432
Taktverzögerer, 106, 107
Taschenrechner, 219, 259, 270
Task, 469
Telekommunikationsnetz, 229
Term, 148, 366
Terminal, 229
Terminalsymbol, 365, 366
Terminierung, 122
Theorembeweiser, 314, 348
Theorie
  formale, 3, 52, 53, 340, 582
  physikalische, 338, 418
Thread, 469
Time sharing, 464
Token, 361
Tokenfolge, 362
Tokenstrom, 361
Top-down-Analyse, 360
Tor, 91, 241
Träger, 593
Trägerfrequenz, 61
Trägerprinzip, 5, 593
Transfer, 264
Transformationsgrammatik, 366
Transistor, 215
Transistor-Mechanismus, 202
Transistorenmatrix, 202, 221
Transition, 102, 505
Transputer, 474
Trial and Error, 315
Trigger, 198
Triode, 214

Tupel, 593
Turbulenz, 530
TURING, A., 128, 162, 319, 375, 387
Turing-Funktion, 127
Turingautomat, 128, 368
    linear beschränkter, 370
Turingmaschine, 128, 367
Turingtest, 373, 375
TYCHO DE BRAHE, 36, 352, 554
Typ, 56, 413
    generischer, 414
    polymorpher, 414
Typanpassung, 363
Typisieren, 57

U

Übergangsprozess, 171, 196, 466, 566
Übersetzen, 154, 159, 348
Übersetzerprogrammtechnik, 357
Umcodierung, 13, 62, 593
Umcodierungseffizienz, 62
Unendlichkeit, 32
Unentscheidbarkeit, 72
Unifikation, 334
uniforme Systembeschreibung, 3
UNIVAC, 214
Universalität, 241
Unterbrechung, 362, 467
Unterfunktion, 409
Unterklasse, 208
Unterordnungsgraph, 446
Unterprogramm, 409, 593
Unterprogrammruf, 369
Unvollständigkeitssatz, 72, 157
Uridem, 21
Urlader, 482
URM-Funktion, 127, 130
Urrealem, 11, 20
Ursache-Wirkung-Zirkularität, 41
Ursache-Wirkung-Zirkel, 74
USB-Funktion, 127, 139, 279
USB-Methode, 6, 203, 238, 266, 515, 531
USB-Sprache, 134

V

Variable
    dynamische, 514
    freie, 147
    gebundene, 147
    metasprachliche, 49, 301, 356, 358, 359, 364, 409, 587
    öffentliche, 510
    private, 510
Vektor, 593
Vektor-Pipelining, 439
Vektorrechner, 439
Vektorregister, 439, 440
Vektorverarbeitung, 429
Verarbeitung
    verteilte, 253
    zentrale, 254
Vereinigungsmenge, 211
Vereinung, 90, 278, 594
Vereinungsproblem, 516
Vererbung, 208, 400, 415, 423
Vergleichsoperator, 250
Verschlüsseler, 219
Verstehbarkeit, 310
verteiltes System, 472, 484
Vielkörperproblem, 522
Vier-Adress-Maschine, 258
Vier-Adress-Maschinensprache , 260
virtuelle Realität, 373, 383
Volladdierer, 184, 244
Vollkommutator, 230
Vollständigkeit, 336
Vollständigkeitsforderung, 168, 170, 197, 561
Von-Neumann-Rechner, 237, 253, 273, 277, 594
Vordergrundprogramm, 469
Vorgehensmodell, 517
Vorkopplung, 91

Vorwärtsinferenz, 333
Vorwärtsverkettung, 333

W

WAGNER, R., 377
Wahrheitskalkül, 213
Wahrsageprädikat, 122, 337, 526
Warteschlange, 245
Waschmaschine, 234
Weiche, 91, 594
Weichenoperator, 505
WEIZENBAUM, J., 375
Wellengleichung, 37, 40
Wenn-dann-Satz, 328
Wertediskussion, 567
Wertetafel, 114, 594
WHILE-Anweisung, 297, 359
Widerspruchsfreiheit, 336
Wiedererkennen, 75, 547, 549
WIENER, N., 41
WILKES, M.W., 235
Willensfreiheit, 395
Windows, 410
Wirbel, 530
WIRTH, N., 413
Wissen, 305, 379
Wissensakquisition, 335
Wissenserwerb, 335, 559
Wissensverarbeitung, 335, 340
Wissensverarbeitungssystem, 335
Wissenszugriff, 335
WITTGENSTNEIN, L., IX
Wohlstruktur, 594
wohlstrukturierbar, 279
Wohlstrukturierung, 141
Wort-Leitung-Zuordner, 186, 202, 220
Wortmanipulation, 290, 318, 323

Z

ZADEH, L. A., 536
Zahl
  irrationale, 34
  rationale, 35
  reelle, 31, 35
Zählen, 84
Zahlengerade, 32
ZAMENHOF, L., 379
Zeichen
  alphanumerisches, 153
Zeichenidem, 21, 594
Zeicheninformation, 11, 23, 583, 594
  vereinbarte, 23
Zeichenkettenverarbeitung, 347
Zeichenkörper, 13, 594
Zeichenmustertransformation, 347
Zeichenrealem, 11, 20, 594
Zeiger, 504, 514
Zeigerkonzept, 513, 515
Zeigervariable, 514
zentrale Steuerschleife, 265
Zielabstand, 317, 388
Zielprogramm, 300
Zielsprache, 300, 324, 358
Zirkularität, 71, 73, 286, 595
  operationale, 69, 73, 408
  referenzielle, 69, 71, 190
  widersprüchliche, 337
Zugehörigkeitsfunktion, 537
Zugehörigkeitsgrad, 537
ZUSE, K., 213
Zustand
  codierender, 159, 169, 196, 580
  dynamisch stabiler, 167, 581
  innerer, 106, 235
  statisch stabiler, 167, 592
Zustandsgleichung, 442
Zustandsparameter
  codierender, 169, 176, 221
Zwei-Adress-Maschine, 238
Zwei-Adress-Maschinensprache, 260
Zweigeweiche, 91, 228, 278, 505, 595
Zwischencode, 363
Zyklus, 74