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.

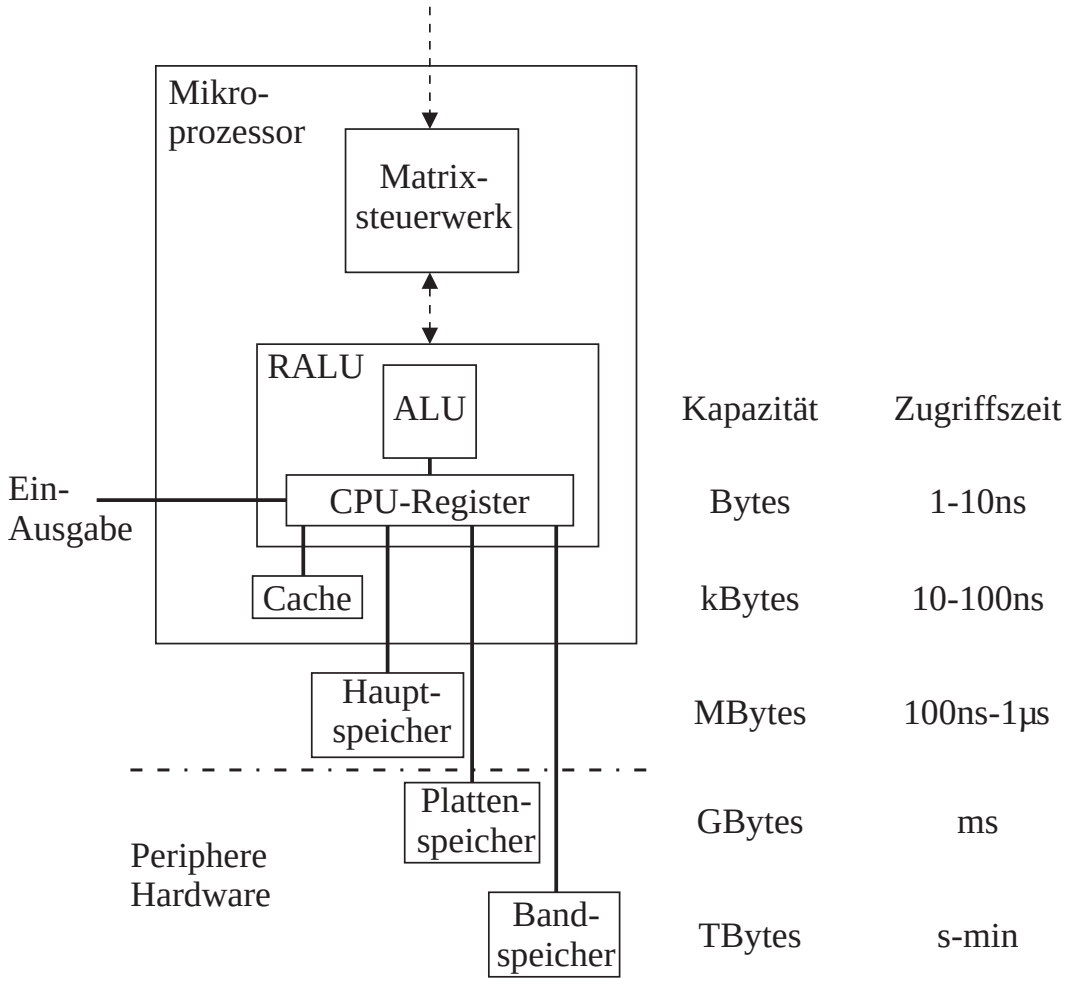


Bild 19.1 Mikroprozessor mit Speicherhierarchie; erweiterte Architektur von Bild 13.7. - - - - Steuerleitungen; ——— Datenleitungen. Auf der rechten Seite sind die Speicherkapazitäten und Zugriffszeiten angegeben. Die skalierenden Buchstaben vor den Maßeinheiten sind zu lesen als

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.

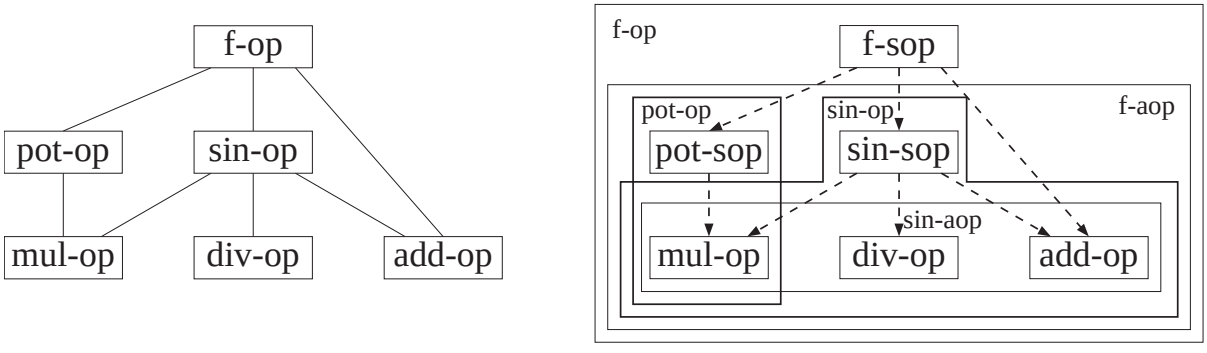
Es stellt sich die Frage nach den Bausteinen eines Operatorennetzes oberhalb der Pozessorebene. Die Antwort wird durch die Komponierung *unterhalb* der Prozessorebene nahegelegt. In Kap.13 hatten wir den Einprozessorrechner als KR-Netz entworfen, also als Netz von Kombinationsschaltungen und Registern. 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

$$y = f(x) = f_1(x) = x^n + x \quad \text{für } x \leq 0$$
$$y = f(x) = f_2(x) = x^n + \sin(x) \quad \text{für } x > 0. \tag{19.1}$$

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*

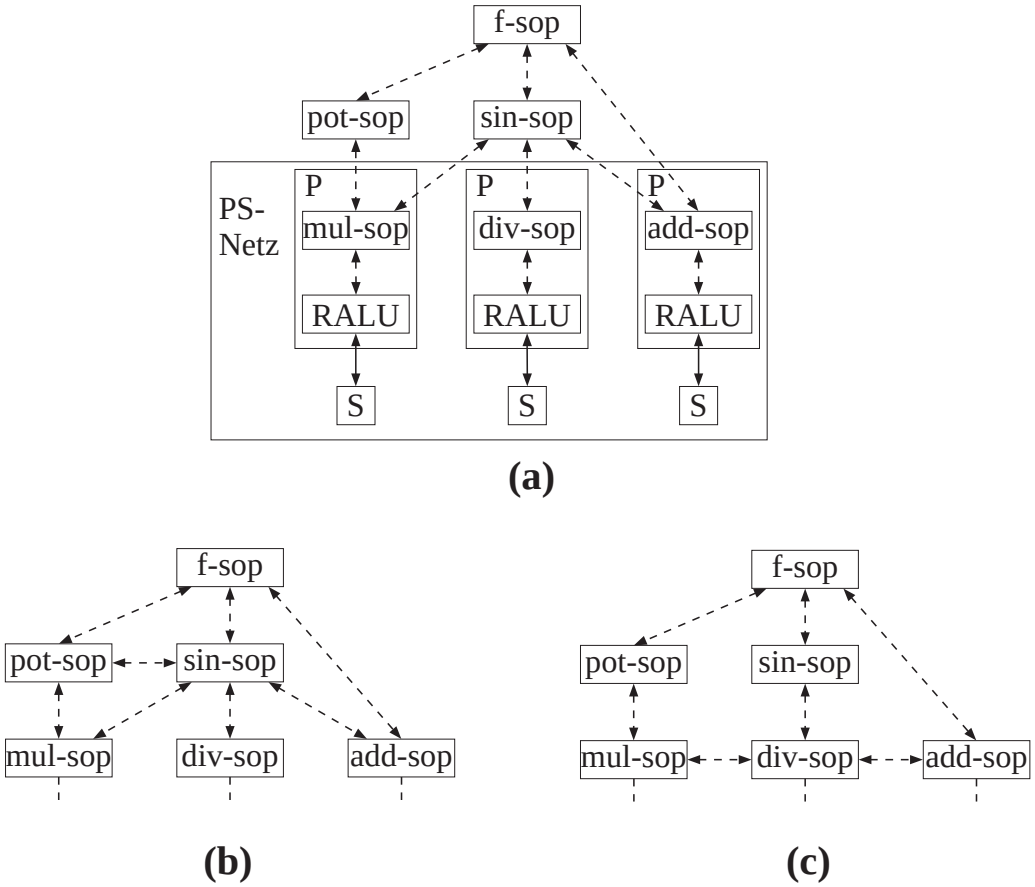


Bild 19.3 Signalwegegraph der Steuerungshierarchie von Bild 19.2; (a) - zentrale Steuerung; (b) und (c) - teilweise dezentralisierte Steuerung (RALUs und Speicher nicht eingezeichnet).

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
3 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

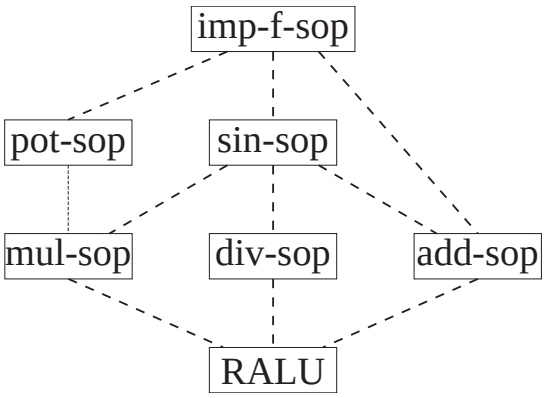


Bild 19.4 Steuerungsgraph, der aus dem Steurungsgraphen von Bild 19.3a durch Zusammenfassung der drei RALUs zu einem RALU hervorgeht; imp-f-sop - sprachlicher, imperativer Steueroperator (Maschinenprogramm) zur Berechnung der Funktion (19.1).

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

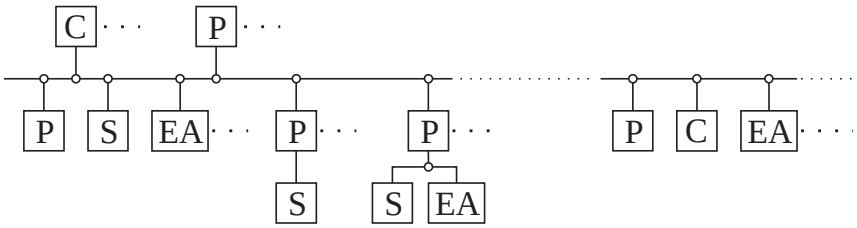


Bild 19.5 PS-Netz mit Busarchitektur. P - Prozessor, S - Speicher, C - Computer, E/A - Ein-Ausgabegerät.

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-*

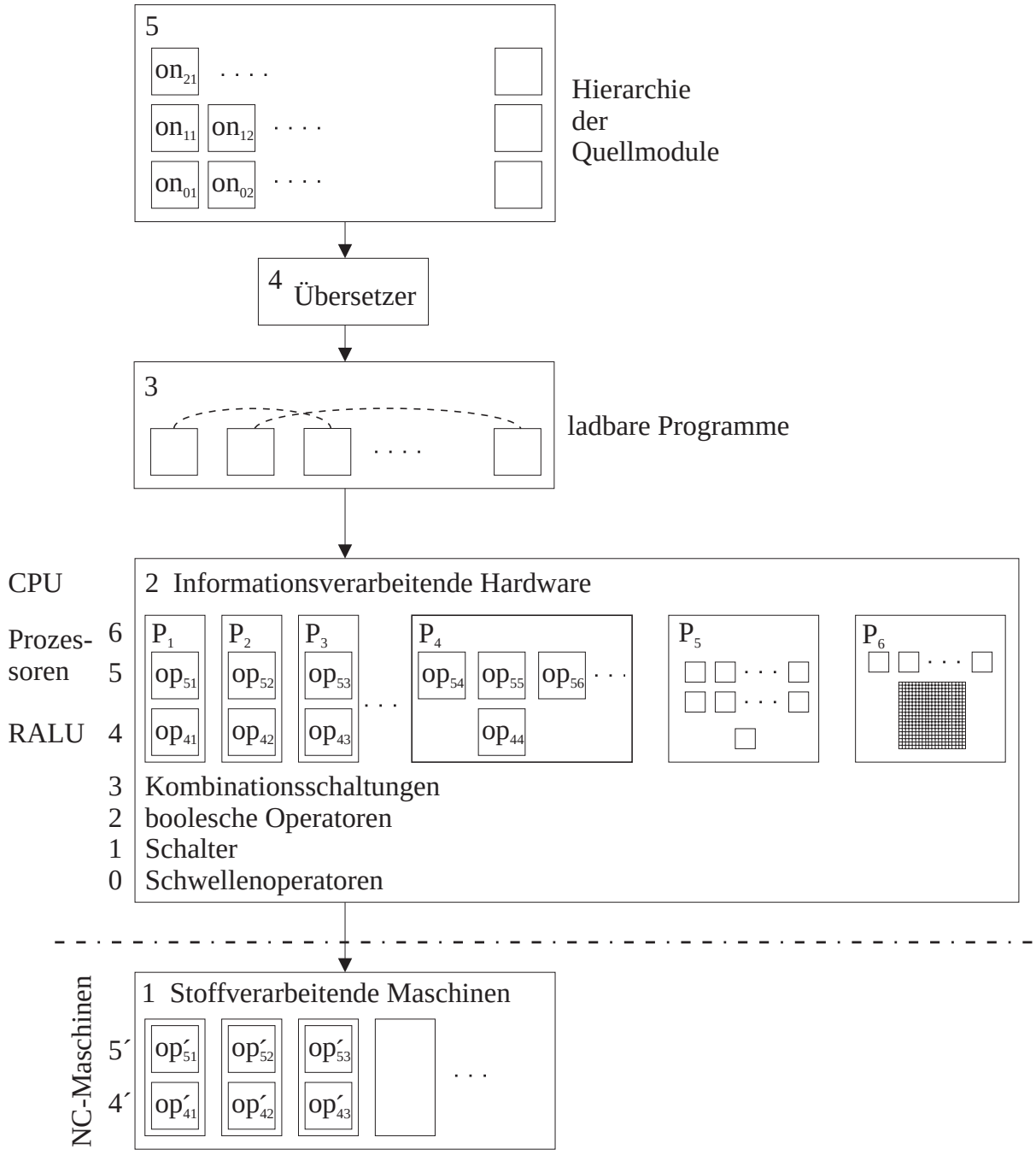


Bild 19.6 Blockbild kausaldiskreter Systeme

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 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 *Betrieb*ssystem 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.

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. 8

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.

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.

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.

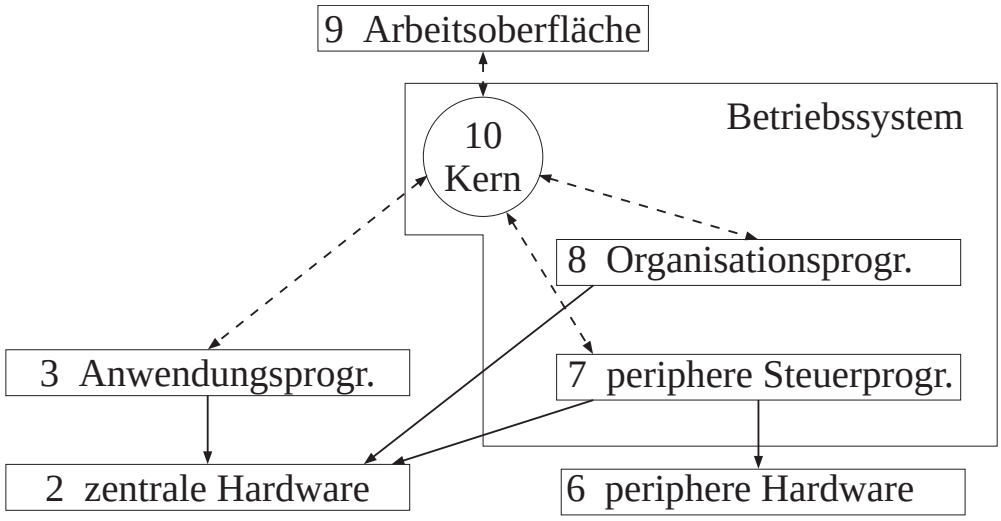


Bild 19.7 Geschützte Prozesskomponierung. Durchgezogene Pfeile - Befehlswege; gestrichelte Pfeile - Rufwege.

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

$$f(x,n) = f_1(x,n) = +(\text{pot}(x,n),\text{x}) \quad \text{für } x <= 0;$$
$$f(x,n) = f_2(x,n) = +((\text{pot}(x,n),\text{x}),\sin(x)) \quad \text{für } x > 0 \qquad (20.2)$$

und die Listennotation, wie sie in CommonLisp verwendet wird,

$$f(x\ \ n) = f_1(x\ \ n) = (+\ \ (\text{pot}\ \ x\ \ n)\ \ x\ \ )\ \ \text{für } x <= 0$$
$$f(x\ \ n) = f_2(x\ \ n) = (+\ \ (\text{pot}\ \ x\ \ n)\ \ (\sin\ \ x))\ \ \text{für } x> 0 \qquad (20.3)$$

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
1 "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.

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

($x \le 0$) **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

$$(f\ x\ n\ y) \Leftarrow ((x \le 0)\ \texttt{AND}\ (x^n + x = y))\ \texttt{OR}\ ((x > 0)\ \texttt{AND}\ (x^n + \sin(x) = y)). \quad (20.4)$$

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

$$(f\ x\ n\ y) \Leftarrow ((x{\leq}0)\ \texttt{AND}\ (f_1\ x\ n\ y))\ \texttt{OR}\ ((x{>}0)\ \texttt{AND}\ (f_2\ x\ n\ y)). \qquad (20.5)$$

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.

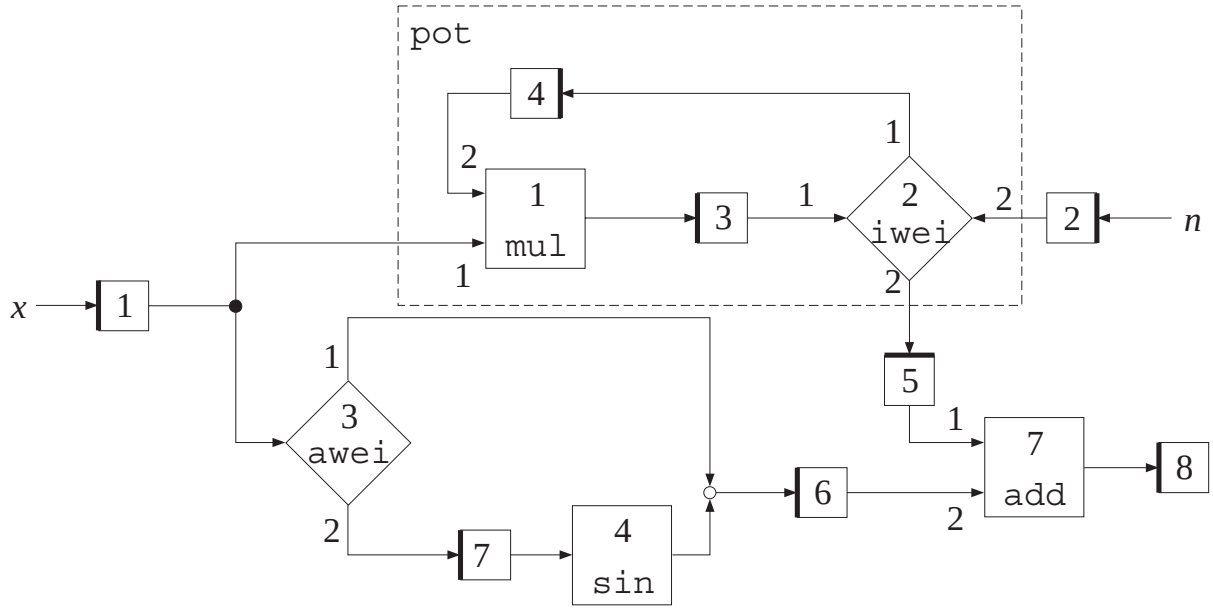


Bild 20.9 Erweitertes Operatorennetz. Große Quadrate und Rhomben - Operatoren; kleine Quadrate - Operandenplätze; Arbeitsoperatoren: 1 - Multiplizierer (`mul`), 4 - Sinusoperator (`sin`), 7 - Addierer (`add`); Steueroperatoren: 2 - Iterierweiche (`iwei`), 3 - Alternativweiche (`awei`), 5 und 6 - Synchronisierer (`sync1, sync2`).

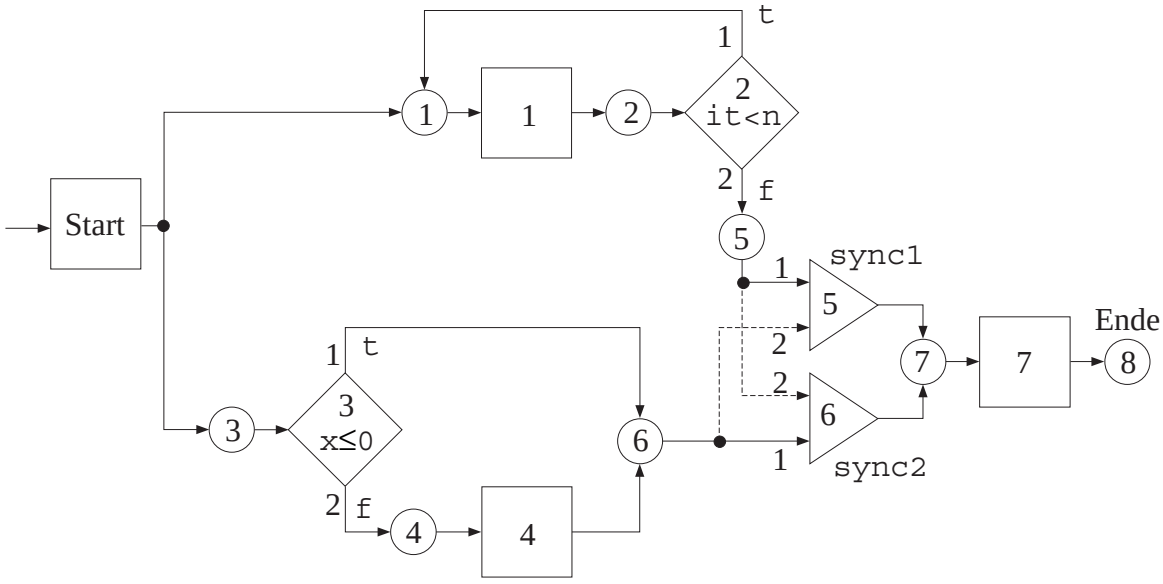


Bild 20.10 Steuerbares Petrinetz. Quadrate und Rhomben - Transitionen, die Nummerierung entspricht der von Bild 20.9; Kreise - Signalplätze (Markenplätze); in der Iterierweiche: it - Iterationszahl.

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, zu- 1
nimmt, 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*.

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 $n^2(2n\text{-}1)$ 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) 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

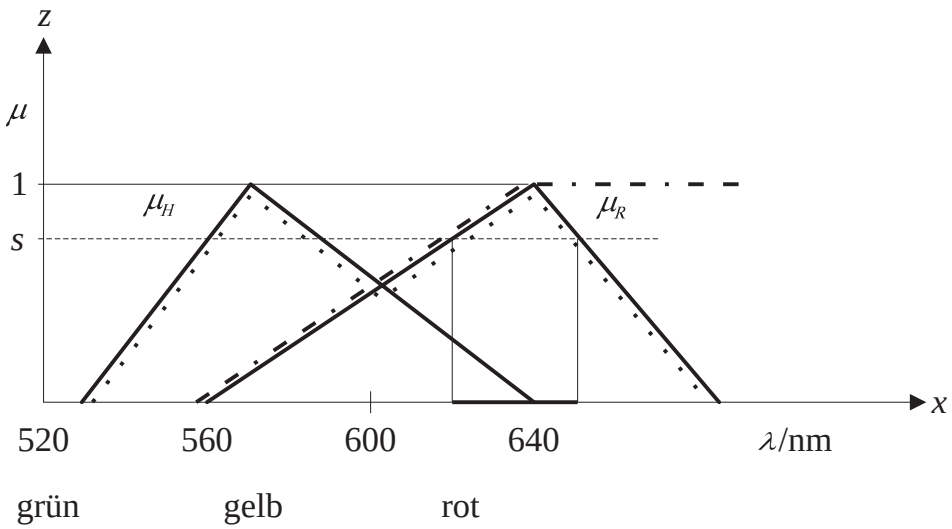


Bild 21.1. Zugehörigkeitsfunktionen von Äpfeln, die Licht der mittleren Wellenlänge λ aussenden, zur Klasse *H* der halbreifen bzw. zur Klasse *R* der reifen Äpfel.

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 $\mu_R(\lambda)$ "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.

$$H \cup R = \{a: (P_1 \text{ OR } P_2)\}. \tag{21.6}$$

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:

$$\mu_{H\cup R}(\lambda) = \max(\mu_H(\lambda), \mu_R(\lambda)). \tag{21.7}$$

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:

$$H \cap M = \{a: (P_1 \text{ AND } P_2)\} \tag{21.9}$$

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:

$$\mu_{H\cap M}(\lambda,G) = \min(\mu_H(\lambda), \mu_R(G)). \tag{21.10}$$

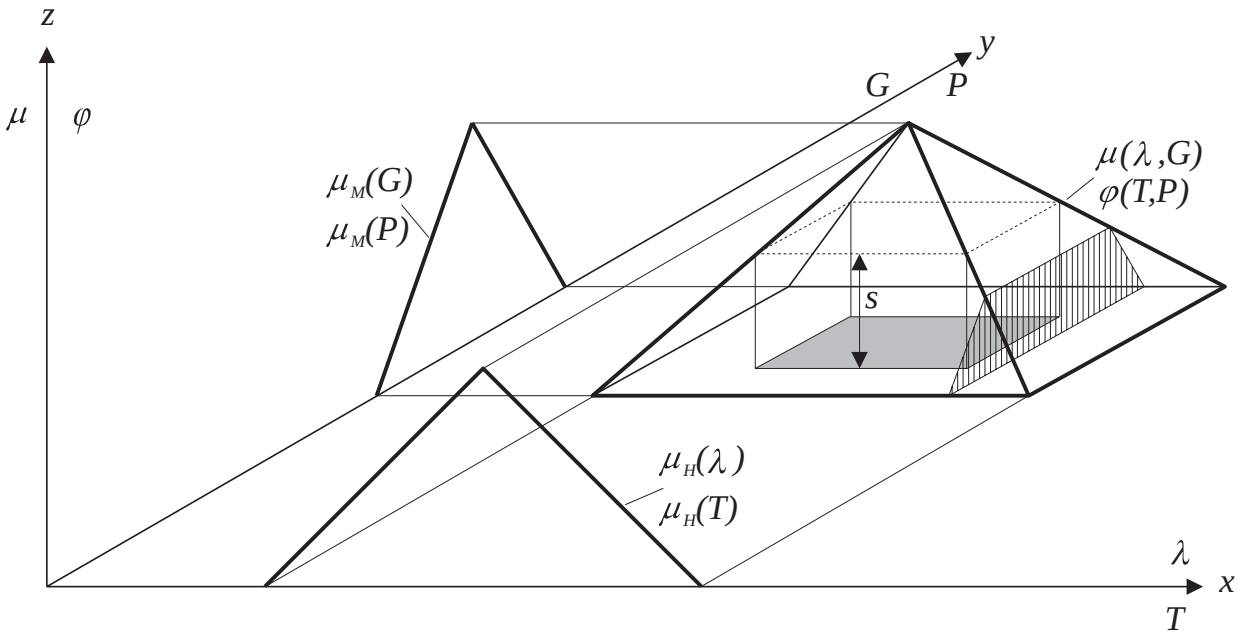


Bild 21.2. Zugehörigkeitsfunktion $\mu(\lambda, G)$ (Fuzzifizierung der Sortierregel (21.8) bzw. Erfüllungsfunktion $\varphi(T,P)$ (Fuzzifizierung der Steuerregel (21.3) λ - Wellenlänge, *T* - Temperatur, *G* - Gewicht, *P* - Leistung. Indizes: *H* - halbreif bzw. handwarm, *M* - mittelgroß bzw. mittlere 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:

$$\mu = \max(\mu^1, \mu^2, ...\mu^r, .\,..) \tag{21.11}$$

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-μ 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 6
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*.

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 hochentwickelten 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.

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. 7

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 8

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 Ähnlichkeitserkenntnisse” 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