Teil 3

Von der Maschinensprache zur künstlichen Intelligenz

14 Zwischenbilanz

Zusammenfassung

Mit dem Einprozessorrechner ist die boolesche Algebra als derjenige Kalkül realisiert worden, in den jeder andere Kalkül, dessen sich die natürliche Intelligenz bedient, zu transformieren ist. Einige Transformationsschritte sind bereits ausgeführt worden und zwar die Überführung der Maschinenoperationen in boolesche Operationen. Um uns nicht in der unübersehbaren Vielfalt der Vorgehensweisen und Methoden der natürlichen Intelligenz zu verlieren, konzentrieren wir unsere Versuche, künstliche Intelligenz zu verwirklichen, auf drei markante Rollen, die der Computer als intelligentes Werkzeug des Menschen übernehmen soll:

- der Computer als Helfer beim Lösen mathematischer Probleme,
- der Computer als Helfer beim Lösen nichtmathematischer Probleme,
- der Computer als Unterhalter, als Gesprächs- und Spielpartner.

Die Unterscheidung zwischen mathematischen und nichtmathematischen Problemen ist aus der Sicht des Menschen mit seinen alltäglichen Aufgaben, nicht aus der Sicht des Computers zu verstehen. Für den Computer gibt es keine nichtmathematischen Aufgaben. Anhand dieser Rollen wird die KI-Problematik in Teil 3 abgehandelt. Auf diese Weise kommen die wesentlichen Probleme der künstlichen Intelligenz zur Sprache.

Es versteht sich von selbst, dass der Rechner mit Zahlen, oder, wie man auch sagt, dass er *numerisch* rechnen kann. Nicht ganz so selbstverständlich ist, dass der Computer auch mit Variablen oder, wie wir im Weiteren auch sagen werden, dass er *analytisch* rechnen kann[1]. Noch merkwürdiger mag es erscheinen, dass der Computer auch nichtmathematische Probleme lösen kann. Das scheinbare *Paradoxon der KI* besteht darin, dass der "Rechner", der zum *Rechnen* gemacht ist, "*nichtmathematisch denken*" kann. Doch hat es nur den Anschein. Tatsächlich kann der Computer nur "kalkülisiert denken", d.h. rechnen. Er "rechnet " auch dann, wenn er an Spielen oder Gesprächen teilnimmt.

Der Weg zur künstlichen Intelligenz hat kein Ende; zumindest kann er kein Ende haben, solange wir nicht wissen, was *natürliche* Intelligenz "wirklich" ist, was sie vermag. Voraussetzung dafür wäre die vollständige Kenntnis der Funktionsweise des Gehirns. Und selbst dann bliebe das prinzipielle Problem der Zirkularität der Fragestellung bestehen: Kann der Mensch erkennen, was er selber kann?

---

1 Zuweilen wird das Rechnen mit Variablen auch als *symbolisches* Rechnen bezeichnet.

14.1 Probleme des Softwareweges zur KI

Da es aus Aufwandsgründen unmöglich ist, rein hardwaremäßig einen universellen Rechner zu bauen, sehen wir uns gezwungen, das Komponieren von Operatoren, aufbauend auf einer geeigneten Hardware, softwaremäßig fortzusetzen. Als Hardware wählen wir den Einprozessorrechner. Wir müssen also oberhalb der Maschinenebene, d.h. unter Verwendung der Maschinensprache, eine Softwarehierarchie komponieren. Zunächst fixieren wir noch einmal das Ziel, das erreicht werden soll. Dazu erinnern wir uns an das Ziel 2, wie es zu Beginn von Kap.13 [13.1] formuliert worden ist: *Realisierung eines* ***Basiskalküls****, in den sich alle Kalküle transformieren lassen, derer sich die natürliche Intelligenz bedient, und Durchführung der Transformationen in den Basiskalkül*. Als Basiskalkül haben wir die boolesche Algebra gewählt und auf ihrer Grundlage den Maschinenkalkül entwickelt.

Die Aufgabe von Teil 3 muss offenbar darin bestehen, verschiedene Kalküle in den Maschinenkalkül zu transformieren. Es fragt sich, um welche Kalküle es sich handeln kann. In Kap.8.6 [8.41] hatten wir das "Fernziel der KI-Forschung" folgendermaßen formuliert: *Kalkülisierung und Implementierung des folgerichtigen Denkens des gesunden Menschenverstandes*. Dieses Ziel ist recht einfach zu erreichen, wenn eine Art des Denkens simuliert werden soll, die bereits kalkülisiert ist. Das gilt z.B. für das mathematische Denken. Dieser Aufgabe ist Kapitel 15 gewidmet. Schwieriger wird es, das menschliche Denken zu simulieren, wenn es nicht oder zumindest nicht deutlich sichtbar kalkülisiert ist. Diesem Problem wenden wir uns in Kap.16 zu. Wir werden uns anhand von zwei Beispielen überlegen, wie das Kalkülisieren erfolgen kann. Das Implementieren eines Kalküls, also das Übersetzen der Kalkülsprache in die Maschinensprache, wird in Kap.16.4 behandelt.

Von den speziellen Problemen der Programmierung und der Softwaretechnologie wird in Teil 3 kaum die Rede sein. Darüber findet der Leser einiges in Teil 4. Im Augenblick kommt es uns darauf an, eine Zwischenbilanz zu ziehen, die wichtigsten Feststellungen und Ergebnisse der vorangehenden Kapitel zusammenzufassen und zu verallgemeinern und auf dieser Grundlage unser Ziel noch schärfer und unseren weiteren Weg deutlicher zu bestimmen.

Der Computer, den wir in Kap.13 entworfen haben, kann unter Verwendung seiner Maschinensprache programmiert, d.h. zur Ausführung unterschiedlicher Operationen konditioniert werden. Wie wir wissen, kann für jede rekursive Funktion ein Maschinenprogramm zu ihrer Berechnung geschrieben werden. Unser Computer ist universell, das heißt, er kann jede rekursive Funktion berechnen, wenn er über ein entsprechendes Programm verfügt, vorausgesetzt, Rechenzeit und Speicherkapazität stehen in ausreichendem Maße zur Verfügung.

Man könnte hinzufügen: Sonst kann der Computer nichts. Dieser Satz ist jedoch irreführend. Denn es gibt "fast nichts", was er nicht kann. Doch dieses "fast nichts" ist etwas sehr Wichtiges. Der Computer scheint nicht imstande zu sein, die Zeichenketten, mit denen er hantiert, mit externer Semantik zu belegen, zumindest nicht in

dem Sinne, wie der Mensch dies tut, wenn er über die Welt nachdenkt oder wenn er sich mit anderen Menschen unterhält. Anders ausgedrückt, der Computer kann den Zeichenrealemen keine Ideme zuordnen, es sei denn, man würde ihm ein Bewusstsein zuschreiben. Diesen Gedanken wollen wir nicht weiterverfolgen. Er würde uns vom Wege abbringen. Denn unser Weg soll nicht zum künstlichen Bewusstsein, sondern "nur" zur künstlichen Intelligenz führen, d.h. zur Fähigkeit des Computers, zu wahren Aussagen über die Wirklichkeit zu gelangen, aber eben zu Aussagen, die erst im Bewusstsein des Menschen mit externer Semantik belegt werden müssen.

Der Weg, auf dem wir das erreichen wollen, ist durch die USB-Methode vorgezeichnet. Er besteht in der Fortsetzung des in der Hardware begonnenen Weges in der Software, m.a.W. in der Komponierung von Operationsvorschriften, d.h. sprachlicher Operatoren, nach den gleichen Regeln, nach denen wir bisher reale Operatoren komponiert haben. Das bedeutet, dass die Flussknoten der Hardware ihr Pendant in der Software haben müssen, dass also die verwendete Programmiersprache über Sprachelemente zur Beschreibung aller Flussknotentypen verfügen muss, wenn sie universell sein soll. Für die Maschinensprache trifft das zu, wovon wir uns in Kap.13.7 überzeugt haben. Das bedeutet, dass mit ihrer Hilfe jede KR-berechenbare Funktion in Form eines imperativen Programms beschrieben, d.h. softwaremäßig aus den Sprachelementen der Maschinensprache "komponiert" werden kann. Es bedeutet aber noch nicht, dass ihre Ausdrucksmittel ausreichen, um Netze und Hierarchien sprachlicher Operatoren zu komponieren.

Hier liegt ein Problem der Programmierungstechnik, das den Informatikern erheblich zu schaffen gemacht hat und das auch uns noch beschäftigen wird. Im Augenblick begnügen wir uns mit folgender Feststellung. Damit der Programmierer sprachliche Kompositoperatoren nach den Regeln der USB-Methode komponieren kann, müssen die Bausteinoperatoren ebenso selbständige, in sich abgeschlossene Einheiten darstellen, wie Hardwareoperatoren, sodass der Programmierer mit ihnen hantieren und sie zu Netzen verknüpfen kann. Dieser Forderung genügen Maschinensprachen, wie wir sie in Kap.13 entworfen haben, leider nicht, denn sie dienen ausschließlich der Artikulierung von Aktionsfolgeprogrammen, nicht aber von Datenflussprogrammen. Dass das Problem im Prinzip lösbar sein muss, folgt aus der Tatsache, dass sich jeder Aktionsfolgeplan in einen Datenflussplan überführen lässt. In Kap.15.4 werden wir die Idee zur Lösung des Problems kennen lernen und die Lösung selber als *prozedurale Abstraktion* bezeichnen.

Wenn der Schritt von der Aktionsfolge zum Datenfluss geschafft ist, sodass mit Datenflussplänen weitergearbeitet (weiterkomponiert) werden kann, ist der Weg, der zu dem gesuchten intelligenten informationellen System führen soll, durch die USB-Methode klar vorgezeichnet. Wenn wir als Ziel unserer Bemühungen die künstliche Intelligenz deklarieren, ergibt sich eine merkwürdige Situation: Der Weg ist klar, das Ziel dagegen nicht. Denn es ist durchaus nicht klar, worin die natürliche Intelligenz, die simuliert werden soll, genau besteht, und wieweit sie geht, m.a.W. wieweit die Fähigkeit des Menschen geht, die Welt sprachlich zu modellieren. Der

Begriff der Modellierbarkeit der Welt ist ein intuitiver Begriff, und als solcher nicht exakt definierbar. Wir erinnern uns, dass das Gleiche für den Begriff der Berechenbarkeit gilt. Die Ursache ist in beiden Fällen ein und dieselbe: die Zirkularität, die in der Frage nach den eigenen Fähigkeiten liegt (vgl. Kap.8.3 [8.23]).

14.2 Drei Rollen des Computers als eines intelligenten Partners des Menschen

Die Unmöglichkeit, das Ziel der KI-Forschung explizit, extensional zu bestimmen, lässt sich nicht dadurch aus dem Wege räumen, dass man das "Fernziel" der KI-Forschung als Simulation des gesunden Menschenverstandes definiert und die beiden Schritte dahin festgelegt, das Kalkülisieren und das Implementieren. Die extensionale Undefinierbarkeit der Intelligenz bedeutet aber nicht, dass es überhaupt keinen Weg zur KI gibt; sie bedeutet lediglich, dass der Weg kein Ende hat. Die Überschrift des Teils 3 bezeichnet nicht Anfang und Ende, sondern Anfang und Richtung des Weges. Aus diesem Grunde lassen wir es bei obiger Konkretisierung des Fernziels der KI bewenden, geben ihr aber eine Form, die der öffentlichen Diskussion zur künstlichen Intelligenz besser gerecht wird, die oft als Diskussion um die "Rolle des Computers als Partner der Menschen" geführt wird.

Wir konzentrieren uns auf drei Rollen, die der Computer als intelligentes Werkzeug des Menschen spielen soll:

1. der Computer als Helfer beim Lösen mathematischer Probleme,
2. der Computer als Helfer beim Lösen nichtmathematischer Probleme,
3. der Computer als Unterhalter, als Gesprächs- und Spielpartner.

Nur in der letzten Rolle tritt der Computer tatsächlich als Partner auf. In den ersten beiden Rollen tritt er eher als Werkzeug auf. Wenn sich jedoch Mensch und Computer beim Problemlösen gegenseitig helfen, kann der Computer als *Kooperationspartner* auch in den ersten beiden Rollen auftreten. Damit der Computer diese beiden Rollen spielen kann, muss er offenbar in der Lage sein, aus gegebenen Prämissen logisch richtige Schlüsse zu ziehen, und zwar je nach Aufgabe auf mathematischem oder auf (scheinbar) nichtmathematischem Wege. In diesem Sinne liegt es nahe, zwischen mathematischer und nichtmathematischer KI zu unterscheiden. Doch ist die Wortverbindung "nichtmathematische KI" missverständlich. Denn der Computer kann nur in seinem Maschinenkalkül "denken", d.h. rechnen, bzw. in irgendeinem anderen Kalkül, falls dieser durch ein Übersetzerprogramm in seinen eigenen Kalkül transformiert worden ist. Nach dem Kalkültransformationssatz [8.39] ist das stets möglich. Im Gegensatz zum Menschen ist der Computer also nicht in der Lage, nichtmathematisch zu denken. Darum gibt es aus der Sicht des Computers keine nichtmathematischen Probleme, sondern nur aus der Sicht des Menschen und es wäre richtiger, anstelle von nichtmathematischer KI von simulierter natürlicher nichtmathemati-

scher Intelligenz zu sprechen. Das ist zu bedenken, wenn im Weiteren der Kürze halber von Nichtmathematischer KI die Rede ist.

Auch wenn man weiß, dass der Computer das Vorgehen des Menschen beim Lösen nichtmathematischer Probleme nur simuliert, ist es - selbst für den Fachmann - immer wieder faszinierend zu erleben, was der Computer mit den Operationen der ALU anstellen kann. Es erscheint geradezu paradox, dass er "vernünftig denken" kann, obwohl er letztlich nur Bitketten vergleicht und transformiert. Diesen erstaunlichen Umstand bezeichnen wir als das "**Paradoxon der KI**". In Kap.16 werden wir uns überlegen, wann und wie es möglich ist, den Computer zu befähigen, nichtmathematisches Denken zu simulieren.

Die Reihenfolge, in der die drei Rollen des Computers aufgezählt sind, spiegelt etwa die Entwicklungsetappen der KI wider. Als erstes konnte der Rechner rechnen. Dagegen ist er bis heute kaum im Stande, ein "Allerweltsgespräch" zu führen und über alles Mögliche sinnvoll zu plaudern. Die Entwicklung der natürlichen Intelligenz des Menschen vom Kleinkind bis zum Erwachsenen verläuft entgegengesetzt. Diesen merkwürdigen Gegensatz hatten wir in Kap.7.1 [7.1] mit dem Wort "Gegenläufigkeitsphänomen" charakterisiert.

Die drei Rollen, die der Computer spielen soll, verschieben in der angegebenen Reihenfolge die Grenze der KI immer weiter in den Bereich hinein, den viele Menschen naturgemäß als ihre eigene "menschliche" Domäne empfinden und den sie gegen das Eindringen der KI aus den unterschiedlichsten, teilweise emotionalen oder ethischen Gründen zu verteidigen sich veranlasst fühlen. Die Folge ist, dass die Frage nach den Grenzen der KI breit und heiß diskutiert wird.

Angesichts der Unbegrenztheit des Weges zur KI ist die Frage nach den Grenzen der KI, also nach dem Ende des Weges zu ihr, inhaltlos. Doch spielt sie in der öffentlichen Diskussion eine derartige Rolle, dass wir sie nicht übergehen dürfen. Wir werden sie immer wieder stellen. Wir werden sie aber nie allgemein, sondern nur von unserem momentanen Standpunkt aus stellen und beantworten. Wir werden stets bemüht sein, die Grenze dessen, was wir auf unserem Weg (in den einzelnen Kapiteln) erreicht haben, klar zu benennen.

Unser Weg wird gemäß der Überschrift des dritten Teils die Richtung der technischen Entwicklung "vom Rechnen zum Erkennen" (vgl.[7.1]) einschlagen und mit der "mathematischen KI" beginnen, genauer mit dem Rechnen "unmittelbar oberhalb" der Hardware, also mit dem Rechnen, das über die Operationen der ALU und des Prozessors, m.a.W. über die Operationen der Maschinensprache hinausgeht, zumindest ein klein wenig, indem es diese Operationen als Bausteinoperationen verwendet. Dabei werden diese jedoch nicht hardwaremäßig miteinander verbunden, sondern die Verbindungen werden in einem Programm beschrieben, das vom Prozessor abzuarbeiten ist.

Zum Begriff der mathematischen KI, wie er hier verwendet wird, ist folgende wichtige Bemerkung zu machen. Wir werden uns bei ihrer Realisierung zunächst auf deduktive Intelligenz beschränken und intuitive Intelligenz ausschließen. Dies be-

deutet für das im Weiteren zu besprechende mathematische Modellieren, dass die Erstellung eines ersten mathematischen Modells Aufgabe des Menschen ist, denn sie erfordert Intuition (oder Kreativität, wie zuweilen gesagt wird). Der Rechner soll ausschließlich beim deduzierenden Arbeiten mit dem Modell, also beim Rechnen in dem Kalkül helfen, aus dem sich das Modell durch Interpretation ergibt, d.h. durch Ersetzen kalkülinterner Bezeichner durch externe Bezeichner.

Abschließend sei darauf hingewiesen, dass hinter der Zielstellung der KI, den natürlichen Menschenverstand zu simulieren, andere Ziele stehen können, beispielsweise die Herausbildung besserer gesellschaftlicher Organisationsformen, um so die anstehenden sozialen Probleme zu lösen und die Überlebenswahrscheinlichkeit der Menschheit zu erhöhen. Das Ergebnis wäre ein Evolutionsschritt, bewirkt durch den (bewussten) Willen zur Arterhaltung. Auf derartige Sekundärziele der KI werden wir nicht eingehen. Doch sei eine eher philosophische Bemerkung zu den “Zielen” der KI erlaubt. Das Ergebnis der “zielstrebigen” Bemühungen der Menschen ist der “Fortschritt” der kulturellen Evolution. Die aber hat keine Ziele, auch nicht, wenn der Mensch sich einbildet, sie stellen zu können - eine paradoxe Situation. Die Volksweisheit hat sie in die Worte gekleidet: “ Der Mensch denkt, Gott lenkt”.

15 Lösen mathematischer Probleme

Zusammenfassung

Aus den Maschinenoperationen, letztendlich also aus den ALU-Operationen, können unter Verwendung der Maschinensprache beliebige arithmetische Operationen komponiert (programmiert) werden, m.a.W. der arithmetische Kalkül kann in den booleschen transformiert werden. Das Programmieren im internen Binärcode des Computers (in der *intern codierten Maschinensprache*) ist unangenehm und aufwendig. Es wird durch die Definition und Implementierung von *Assemblersprachen* und von *höheren Programmiersprachen* erleichtert.

Eine Assemblersprache kann als "*gehobene Maschinensprache*" aufgefasst werden, denn sie stellt im Gegensatz zu höheren Programmiersprachen keine Erweiterung der intern codierten Maschinensprache dar, erlaubt aber die Verwendung von Bezeichnern für Operanden und Operationen, auch für Kompositoperationen, also für Programme. Das Übersetzerprogramm, das die Übersetzung von Programmen aus der Assemblersprache in den internen Code ausführt, heißt *Assembler*.

Höhere Programmiersprachen enthalten ausdrucksstärkere Sprachelemente für die Komponierung sowohl von Kompositoperanden, beispielsweise Vektoren oder Matrizen, als auch von Kompositoperationen. So sind für die Programmierung von Iterationen verschiedene Sprachelemente entwickelt worden, z.B. die WHILE-Anweisung. Ein Programm, das in einer höheren Programmiersprache geschrieben ist, muss in die Maschinensprache des Computers übersetzt werden, der das Programm abarbeiten soll.

Assemblersprachen und höhere Programmiersprachen dienen der Überwindung der "semantischen Lücke" zwischen der Sprache, in welcher der Programmierer normalerweise denkt und sich ausdrückt, und der Sprache, die der "sprachbegabte" (d.h. mit Übersetzerprogrammen ausgerüstete) Computer versteht. Damit dienen sie der Lösung des *technischen Semantikproblems*, d.h. der Anbindung der Nutzersemantik, also der externen oder formalen Semantik, in welcher der Nutzer denkt, an die interne Computersemantik, an die Prozesse im Computer. Die Anbindung lässt sich verhältnismäßig leicht bewerkstelligen, wenn der Nutzer (beispielsweise ein Mathematiker) in einem Kalkül denkt. Ihm fällt der Sprung über die semantische Lücke umso leichter, je verwandter seine Kalkülsprache der Sprache des Computers (der Programmiersprache) ist. Der Wunsch, die semantische Lücke zu verringern, ist der Motor der Entwicklung neuer Programmiersprachen.

Der Computer kann nicht nur numerische (zahlenmäßige), sondern auch analytische Rechnungen (Rechnungen mit Variablen) ausführen. Analytisches Rechnen besteht im Überführen analytischer Ausdrücke in andere Ausdrücke, z.B. $(a+b)(a-b)$ in (a^2-b^2) oder $\sin x/\cos x$ in $\tan x$. Der Computer bedient sich beim analytischen Rechnen - ebenso wie gegebenenfalls der Mensch - einer Formelsammlung. Sie kann

beispielsweise die “Formel” sin*x*/cos*x* = tan*x* enthalten. Das Gleichheitszeichen in einer Formel kann als zweiseitige Ergibtgleichung gelesen werden; die rechte Seite ergibt sich aus der linken und umgekehrt. Das Hantieren des Computers mit Formeln heißt *Formelmanipulation*.

Die wichtigsten Bausteinoperationen der Formelmanipulation sind das *Suchen* (es schließt *Vergleichen* ein) und das *Substituieren*. Um einen analytischen Ausdruck umzuformen, wird zunächst durch Vergleich nach einer Formel gesucht, die auf den umzuformenden Ausdruck anwendbar ist. Das ist dann der Fall, wenn der Ausdruck eine Teilzeichenkette enthält, die eine vom Kontext unabhängig ausführbare Anweisung darstellt und die mit der linken oder rechten Seite der Formel identisch ist oder durch Bezeichnerabgleich (geeignete Änderung der Bezeichner) identisch gemacht werden kann, sodass die Teilzeichenkette durch die andere Seite der Formel substituiert werden kann. Existiert eine solche Formel, wird die Substitution ausgeführt. Wenn mehrere Formeln gleichzeitig anwendbar sind, muss der Lösungsweg durch Versuchen gefunden werden, sodass das Rechnen zu einem nichtdeterministischen Prozess wird. Die Berechnungsvorschrift stellt einen *nichtdeterministischen Algorithmus* dar. Durch eine zusätzliche Vorschrift für die eindeutige Wahl der anzuwendenden Formel kann ein nichtdeterministischer Algorithmus in einen deterministischen überführt werden.

Programme der Formelmanipulation und des analytischen Rechnens werden vorteilhafterweise in funktionalen Programmiersprachen geschrieben. Das Übersetzen in eine Maschinensprache ist immer möglich. Auf diese Weise lässt sich jeder analytische Kalkül in den booleschen transformieren.

Der Mechanismus der Formelmanipulation ist auch dann anwendbar, wenn nicht analytische Ausdrücke gemäß mathematischer Formeln, sondern wenn beliebige Zeichenketten (Wörter) gemäß einer Liste von Substitutionsregeln umgeformt werden. Dann spricht man von *Wortmanipulation*. Durch einen deterministischen Manipulationsalgorithmus (samt Regelliste) wird eine eindeutige Abbildung aus einer gegebenen Eingabewortmenge in eine Ausgabewortmenge, also eine Funktion festgelegt. Die Klasse der durch deterministische Wortmanipulation berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Nicht nur das analytische, sondern auch das numerische Rechnen ist im Grunde ein Substituieren. Beispielsweise können die Zeilen des kleinen Einmaleins als Substitutionsregeln aufgefasst werden. Die Tätigkeit eines Computers ist *immer* ein Substituieren. Dabei arbeitet er niemals mit Bedeutungen, niemals mit externer Semantik, sondern ausschließlich mit Zeichenketten, er arbeitet nicht mit Idemen, sondern mit Realemen.

Mathematisches Operieren ist stets Deduzieren in einem Kalkül, d.h. numerisches oder analytisches Rechnen. Durch entsprechende Software kann der Computer befähigt werden, jede beliebige numerische oder analytische Rechnung und damit jede mathematische Operation auszuführen.

15.1 Funktionales Programmieren

Wir wenden uns der ersten der drei Rollen des Computers zu, die wir in Kap.14.2 ins Auge gefasst haben, der Rolle eines Helfers beim Lösen mathematischer Aufgaben. Beim Nachdenken darüber, wie der Computer befähigt werden kann, diese Rolle zu spielen, stößt man auf einen scheinbar fatalen Widerspruch zwischen den Fähigkeiten des Computers und der ihm zugedachten Rolle. Der Einprozessorrechner von Bild 13.7 kann nur seine Maschinenprogramme, d.h. speziell notierte imperative Algorithmen ausführen. Die übliche Mathematik, wie sie in Naturwissenschaft und Technik, nicht selten aber auch im Alltag zur Anwendung kommt, bevorzugt formalisierte Sprachen, die nicht die imperative, sondern die funktionale Notation verwenden. Man denke an die Gleichung der Geraden, an das Fallgesetz oder an die Funktion (8.1), die durch das Operatorennetz von Bild Abb.8.1 berechnet wird. Zur Erinnerung sei (8.1) noch einmal angegeben, diesmal jedoch unter der Annahme, dass der Exponent *n* kein konstanter Parameter, sondern eine Variable ist:

$$y = f_1(x,n) = x^n + x \quad \text{für } x \leq 0 \tag{15.1a}$$

$$y = f_2(x,n) = x^n + \sin x \quad \text{für } x > 0, \tag{15.1b}$$

wobei *n* eine ganze Zahl ≥0 ist.

Es wäre eine verlockende Idee, die Sprache der Mathematik als Programmiersprache zu verwenden, sodass man nur die Formeln (15.1) über die Tastatur in seinen PC einzugeben brauchte, um ihn zu befähigen, die Funktion $f(x,n)$ zu berechnen. Dass dies im Prinzip möglich ist, folgt aus dem Kalkül-Tansformationssatz [8.39]. Erforderlich wäre ein Übersetzerprogramm, dass die funktionale Sprache der Mathematik in die imperative Maschinensprache des PC übersetzt. In Kap.15.9 werden wir uns überlegen, wie das im Prinzip möglich ist. Im Augenblick gehen wir davon aus, dass das Übersetzungsproblem gelöst ist. Die Lösung verlangt aber eine "*computerangepasste*" funktionale Notation der zu übersetzenden Sprache, d.h. eine Notation, die vollkommen eindeutig ist und die ein effizientes Übersetzen ermöglicht. Wenn wir darangehen, eine funktionale Programmiersprache zu entwickeln, die diese Forderungen erfüllt, bleibt die interne Semantik funktionaler Programme im Dunkeln, denn wir wissen nicht, wie sie übersetzt werden und welche Prozesse bei der Ausführung der übersetzten Programme im Computer auslöst werden. Darin liegt ein Verstoß gegen das Trägerprinzip, den wir in Kauf nehmen, um den begonnenen Gedankengang nicht unterbrechen zu müssen.

Ein kritischer Blick auf die Ausdrücke (15.1) zeigt, dass die Notation nicht sehr geeignet ist, um unmittelbar als Computerprogramm zu dienen. Man erkennt nämlich, dass für die Notation von Operationen drei unterschiedliche Syntaxregeln zu Anwendung kommen. Bei der Addition steht der Operationsbezeichner *zwischen* den beiden Operanden, bei der Sinusoperation steht er *vor* dem Operanden und bei der Potenzierung tritt gar kein Bezeichner auf, sondern die Operation ist durch Hochstel-

lung des Exponenten gekennzeichnet. Diese Uneinheitlichkeit der Syntax geht zu Lasten der Übersetzung. Hinzu kommt, dass die Hochstellung die Linearität (Eindimensionalität) der Sprache verletzt; die Eingaben sind keine reine Zeichen*ketten*. Wir vereinheitlichen die Notationsweise und gewährleisten die Linearität, indem wir festlegen, dass das Operationssymbol (der Bezeichner der Operation oder Funktion) stets explizit angegeben und den Operanden vorangestellt wird.

In Kap. 8.4.7 [8.34] waren zwei spezielle derartige funktionale Notationsweisen eingeführt worden, die *Präfixnotation* in (8.15) und die *Listennotation* in (8.16). Danach sind f_1 und f_2 folgendermaßen zu notieren, in Präfixnotation

$$f_1 = +(\text{pot}(x,n), x)$$
$$f_2 = +(\text{pot}(x,n), \sin(x)) \qquad (15.2)$$

und in Listennotation

$$f_1 = (+ (\text{pot } x\ n)\ x))$$
$$f_2 = (+ (\text{pot } x\ n)\ (\sin\ x)) \qquad (15.3)$$

Diese Notationen erinnern wenig an einen Algorithmus oder eine Aktionsfolge, eher an einen Datenflussplan. Beispielsweise kann man die zweite Formel in (15.3) folgendermaßen lesen. Der *x*-Wert wird sowohl dem sin-Operator als auch dem Potenzoperator als Eingabeoperand übergeben; sodann werden die Ausgabeoperanden beider Operatoren an den Additionsoperator als Eingabeoperanden weitergegeben, der als Ausgabeoperand schließlich den Funktionswert liefert. Die Eingabeoperanden des Addierers treten nicht explizit auf, sie werden vielmehr durch (pot *x n*) und (sin *x*) dargestellt. Eine solche Notations- und Interpretationsweise hatten wir *funktional* genannt.

Hierin liegt der Kern der funktionalen Notation und des funktionalen Programmierens. *In einem* ***funktional notierten Programm*** *steht der Bezeichner eines Operators (einer Operation, einer Funktion) zusammen mit dem Eingabeoperanden (Eingabeoperandentupel) für den Wert, den die Operationsausführung (die Funktionsberechnung) liefert.*

Danach stellt jeder Klammerausdruck in (15.3) einen Wert dar. Syntaktisch stellt er eine Liste von Elementen dar. Das erste Element der Liste bezeichnet die Operation (Funktion), die folgenden Elemente bezeichnen die Argumente. Sie können ihrerseits wiederum Listen darstellen, sodass Listen hierarchisch strukturiert oder geschachtelt sein können. Für (15.3) trifft dies zu. In der USB-Terminologie wäre eine geschachtelte Liste als *Kompositliste* zu bezeichnen, die aus *Bausteinlisten* komponiert ist. Komponieren ist in diesem Fall ein Aneinanderreihen, eine sog. *Konkatenation*. Die Möglichkeit der listenförmigen Notation war offenbar ein Anlass für McCarthy, die listenorientierte Programmiersprache Lisp zu entwickeln.

Aus der Sprache Lisp sind viele Abkömmlinge hervorgegangen, u.a. die Sprache CommonLisp. In dieser Sprache nimmt (15.3) folgenden Form an (vgl. Bild 20.3 Zeile 3):

```
(defun f2(x n)(+(pot x n)(sin x))).
```

Die Zeichenkette `defun f2(x n)` bedeutet, dass die Funktion `f2(x n)` durch den nachfolgenden Ausdruck definiert wird. Dem entspricht die Bedeutung des Gleichheitszeichen in (15.3), wo es ein Definitionszeichen darstellt. Weitere Kommentare erübrigen sich.

Der Umstand, dass in der funktionalen Notation Zwischenresultate nicht explizit benannt werden, hat zur Folge, dass Iterationen *rekursiv* formuliert werden müssen. Darauf war bereits in Kap.8.4.6 [8.32] hingewiesen worden und auch darauf, dass die Definition der rekursiven Iteration in Kap. 8.4.5 eine entsprechende Notation in Form von (8.9) anbietet. Speziell ist in (8.11) für das Potenzieren durch iteratives Multiplizieren eine funktionale Notation angegeben. Ersetzt man *f* durch pot, geht (8.11) in

$$\text{pot}(x,n) = x * \text{pot}(x,n-1) \quad \text{für } n>0$$

$$\text{pot}(x,0) = 1 \qquad (15.4a)$$

über. Die funktionale Notation wird dadurch möglich, dass die zu definierende Funktion (in (8.9) und (8.11) mit *f* und in (15.4a) mit pot bezeichnet) in dem definierenden Ausdruck auftritt, dass also die Funktion sich gewissermaßen durch sich selbst definiert. Eben in diesem Sinne wird das Wort Rekursion in der Programmierungstechnik benutzt [8.30]. In CommonLisp wäre folgendermaßen zu notieren (vgl. Bild 20.3 Zeile 4):

```
(defun pot(x n)(if(= n 0)1(* x(pot x(- n 1)))))
```
(15.4b)

Die if-Funktion war in Kap.8.4.7 [8.36] eingeführt und im Anschluss an (8.21) erklärt worden.

Die Darstellung der rekursiven Berechnung durch ein Operatorennetz in den Bildern 8.1 und 8.9 und in den Formeln (8.9) und (8.11) demonstriert die enge Verwandtschaft zwischen funktionaler Programmierung und Operandenfluss- bzw. Datenflussprogrammierung (in der Literatur oft als datenflussorientierte Programmierung bezeichnet). Beide Programmierungsarten beruhen auf der Vorstellung eines Operandenflusses durch ein Operatorennetz. Insofern ist funktionales Programmieren datenflussorientiert zu nennen. Auch bei der Operandenflussprogrammierung erübrigt sich die explizite Benennung von Zwischenergebnissen, da deren Weg durch den Operandenflussplan unmittelbar vorgegeben ist, im Unterschied zur Aktionsfolgeprogrammierung, wo alle Wege über den zentralen Arbeitsspeicher führen, sodass die jeweiligen Speicherlätze über die Namen der Zwischenergebnisse explizit angegeben werden müssen [13.8].

Die vorangehenden Überlegungen verstoßen, wie gesagt, gegen das Trägerprinzip, weil die interne Semantik funktionaler Programme völlig im Dunkeln geblieben

ist. Wir wollen etwas Licht in das Dunkel bringen, indem wir uns überlegen, wie das Maschinenprogramm aussehen könnte, in das ein funktionales Programm übersetzt wird. Das Übersetzen selber bleibt zunächst außer Betracht (siehe Kap.15.9).

15.2 Imperatives Programmieren

Unsere Maschinensprache von Kap.13.5.2 ist offensichtlich keine funktionale Sprache, denn jeder Operand wird explizit benannt; die Konstruktion des Prozessorrechners verlangt dies, denn das Resultat einer Rechnung steht dem Prozessor nicht mehr unmittelbar zur Verfügung, nachdem es im Hauptspeicher abgespeichert und der AC (Akkumulator) mit einem anderen Wert überschrieben worden ist. Wenn der Prozessor es benötigt, muss er es benennen und anfordern. Das bedeutet, dass Maschinenprogramme (falls sie terminieren) imperative Algorithmen [7.10] sind, wie wir bereits wissen. Es sei an die folgenden, in Kap.13.5.1 [13.9] eingeführten Begriffe erinnert: *Ein maschinenverständlich notierter imperativer Algorithmus heißt* ***imperatives Programm***. *Programmiersprachen, die das Schreiben imperativer Programme unterstützen, heißen* ***imperative Sprachen***.

Wir wollen nun ein Maschinenprogramm zur Berechnung der durch (15.1b) definierten Funktion $y = f_2(x,n)$ schreiben. Wir nehmen an, dass die Maschinensprache den Operationscode `SIN` für die Sinusfunktion und den Operationscode `POT` für das Potenzieren enthält. Ihre Komponierung aus den ALU-Operationen kann beispielsweise durch ein Matrixsteuerwerk realisiert werden (siehe Kap.13.5.6). Die vier Befehle im Programm von Bild 15.1 stellen den zentralen Abschnitt eines möglichen Maschinenprogramms dar. Anstelle der Variablenadressen sind (nach dem Vorbild von Bild 13.6) die in spitze Klammern gesetzte Variablenbezeichner eingetragen. Beispielsweise stellt <x> die Adresse dar, unter der die Werte der Variablen *x* abgespeichert werden. Die Befehle haben folgende Wirkungen im Computer (folgende interne Semantik):

```
1   TVS <x>
2   POT <n> <y>
3   SIN <x>
4   ADD <y> <y>
```

Bild 15.1 Fiktives Maschinenprogramm zur Berechnung der Funktion $y = x^n + \sin x$.

Befehl 1: Transport des Wertes der Variablen *x* vom Speicher in den Akkumulator.

Befehl 2: Holen des Exponenten (d.h. Transport des Wertes von *n* vom Speicher in das dafür vorgesehene Datenregister), Potenzieren und Speichern der Potenz unter der Adresse der Variablen *y* (das ist erlaubt, obwohl das Ergebnis noch nicht der Wert von f_2 ist).

Befehl 3: Holen des Wertes der Variablen *x* und Berechnen von sin(*x*). Das Ergebnis bleibt nur im AC stehen, es wird nicht im HS gespeichert. Wenn der Befehl gar keine Adresse enthielte, würde nach seiner Ausführung im AC der Wert von sin(x^n) stehen.

Befehl 4: Holen des unter <*y*> gespeicherten Wertes, Addieren dieses Wertes mit dem im AC stehenden Wert von sin(*x*) und Abspeichern der Summe (des Wertes von f_2) in der Speicherzelle der Variablen *y*.

Wenn die Maschinensprache die POT-Operation nicht zur Verfügung stellt, obliegt es dem Programmierer, das Potenzieren auf das Multiplizieren zurückzuführen (vorausgesetzt die Maschinensprache enthält die MUL-Operation). Im Programm von Bild 15.2a ist dies geschehen. Es wird davon ausgegangen, dass die Werte von *x* und *n* bereits im Hauptspeicher stehen. Der dafür zuständige *Deklarationsteil* des Programms, der die Variablen *deklariert* (dem Computer "erklärt") und die entsprechenden Wertzuweisungen sind unterschlagen. Ferner wird angenommen, dass die Konstanten 0 und 1 unter den Adressen der Bezeichner `NULL` und `EINS` fest abgespeichert sind.

```
BEGINN 4000
 1  TVS <EINS>
 2  TNS <y>
 3  TVS <NULL>
 4  TNS <z>
 5  TVS <x>
 6  MUL <y> <y>
 7  INK <z>
 8  SPK <n> 5


 9  SIN <x>
10  ADD<y> <y>
ENDE
```

(a)

```
PROGRAMM f2;

    y := 1;

    z := 0;

ZYK y := x*y;
    z := z+1;
    Wenn z kleiner als n
     dann springe nach ZYK
   sonst

      y := y+sin(x);
ENDE
```

(b)

Bild 15.2 Maschinenprogramm zur Berechnung der Funktion $y = x^n + \sin x$. (a) - Programm; (b) - Kommentar.

Der Leser wird das Programm ohne Schwierigkeiten verstehen, wenn er zum Verständnis der linken Zeilen von Bild 15.2 (der Zeilen von Bild 15.2a) die rechten Zeilen (die Zeilen von Bild 15.2b) als Kommentare zurate zieht. Die Bedeutungen (die interne Semantik) der Operationscodes `TVS` und `TNS` (Transport vom bzw. nach dem Speicher) sind aus Kap. 13.5.3 bekannt. Offenbar bewirken die linken Befehle

der Zeilen 1 und 2 gemeinsam die Abspeicherung einer 1 unter der Adresse <y>, also die auf der rechten Seite angegebene Wertzuweisung. Analoges gilt für die Befehle 3 und 4. Der MUL-Befehl entspricht intersemantisch dem ADD-Befehl, nur wird die ALU nicht auf Addition, sondern auf Multiplikation konditioniert. Zeile 7 befiehlt die Inkrementierung von z.

Befehl 8 ist auf der rechten Seite ziemlich ausführlich kommentiert. Der Operationscode SPK ist die Abkürzung von "SPringe, wenn Kleiner". Der Sprungbefehl ist folgendermaßen zu lesen: "Wenn der Inhalt des Akkumulators kleiner ist als z, springe nach Befehl 5". Auf der rechten Seite steht zwischen Wenn und dann die Bedingung, unter der gesprungen werden soll. Der Kommentar zu Befehl 8 artikuliert einen *bedingten Sprungbefehl* in einer unmittelbar verständlichen Form. Das Sprungziel (Befehl 5) ist im Kommentar *markiert*, d.h. die Zeile beginnt mit einer *Marke*, in diesem Fall mit der Marke ZYK. Die Markierung ist notwendig, da die Anweisungen (Kommentare) nicht nummeriert sind.

Das Programm bewerkstelligt das Potenzieren mit Hilfe eines *Zyklus*, d.h. durch wiederholtes Zurückspringen von Zeile 8 zu Zeile 5. Dem Zyklus entspricht die Rückkopplungschleife in Bild 8.1. Das Potenzieren erfolgt durch iteriertes Multiplizieren. Nach der Inkrementierung (Zeile 7) steht unter der Adresse <z> die aktuelle Iterationszahl, also die Anzahl der bis dahin ausgeführten Multiplikationen und unter der Adresse <y> die entsprechende Potenz von x. Wenn beispielsweise der Wert von f_2 für x=2 und n=3 berechnet werden soll, wird die ursprüngliche 1 unter der Adresse <y> der Reihe durch die Zahlen 2, 4, 8 überschrieben. Sobald unter der Adresse <z> die Zahl 3 steht, wird die Iteration abgebrochen. Die Zeilen 5 bis 8 bilden gemeinsam die imperative Notation einer funktionalen Iteration [8.32]. Die Befehle 9 und 10 sind auf der rechten Seite in einer einzigen Anweisung zusammengefasst.

Der bedingte Sprungbefehl 8 unterscheidet sich vom Sprungbefehl der Registermaschine in Kap. 8.4.3. Es sind viele bedingte Sprungbefehle denkbar, von denen in der Regel aber nur einige wenige in einer konkreten Maschinensprache realisiert sind. Das Grundprinzip der Iteration ist, unabhängig vom speziellen Sprungbefehl, stets das gleiche. Es ist in Kap.8.4.5 behandelt worden.

Damit die rechten Zeilen in Bild 15.2 die Rolle von Kommentaren spielen können, müssen sie die Semantik der linken Zeilen "mit einfacheren Worten", d.h. sinnfälliger artikulieren. Die für die weitere Entwicklung der Rechentechnik entscheidende Idee bestand darin, den Kommentar so exakt und vollständig auszuformulieren, dass seine Übersetzung in die Maschinensprache algorithmierbar und implementierbar wird. Diese *Exaktheitsforderung* verlangt, dass die Kommentarsprache zu einer Kalkülsprache formalisiert wird. Besonders wichtig für die Lesbarkeit der Kommentare ist die Ersetzung der Adressen durch Bezeichner, beispielsweise <y> durch y. In Kap.15.4 werden wir uns überlegen, wie der Computer befähigt werden kann, Bezeichner zu "verstehen".

Die Kommentare von Bild 15.2b erfüllen die Exaktheitsforderung an eine Programmiersprache bereits im Wesentlichen (abgesehen von dem fehlenden Deklara-

tionsteil). Das ist durch Befolgung bestimmter Syntaxregeln erreicht worden. Dazu gehört das Abschließen jeder Anweisung mit einem Semikolon und das "Einklammern" des Programms in die Wörter `PROGRAMM` und `ENDE`. Diese Regeln gewährleisten, dass das Übersetzerprogramm Anfang und Ende der einzelnen Anweisungen sowie des ganzen Programms richtig erkennt. Die Nummerierung der Anweisungen erübrigt sich.

Das `Programm f2` in Bild 15.2b ist offensichtlich ein *imperatives* Programm, denn es besteht, ebenso wie das Maschinenprogramm, aus Befehlen, die üblicherweise nicht als Befehle, sondern als *Anweisungen* bezeichnet werden. Die Artikulierung der Kommentare ist recht willkürlich. Ebenso willkürlich ist die Definition einer Programmiersprache, die aus derartigen Kommentaren hervorgegangen ist. Abgesehen von der Exaktheitsforderung sind der Phantasie der Spracherfinder kaum Grenzen gesetzt. Es ist also nicht verwunderlich, dass die unterschiedlichsten Sprachen entwickelt worden sind. Ob sich eine Sprache durchsetzt, hängt zum einen von ihrer Nutzerfreundlichkeit ab, vor allem von der Lesbarkeit, der Ausdrucksstärke und der Fehleranfälligkeit der artikulierbaren Programme, und zum anderen davon, wie effektiv sich die Sprache implementieren lässt, d.h. welcher Aufwand für die Übersetzung in eine Maschinensprache getrieben werden muss. In Kap.20 werden einige Sprachen anhand von Programmbeispielen vorgestellt.

Bei der Definition höherer Programmiersprachen ist es üblich, als Sprachelemente, die keine Bezeichner sind, englische Wörter oder mathematische Symbole zu verwenden. Anstelle des Kommentars zu Befehl 8 könnte beispielsweise `IF z<0 THEN GOTO ZYK ELSE` notiert werden, in Übereinstimmung mit der Syntax einiger gängiger Programmiersprachen.

Wir wollen nun das `PROGRAMM f2` von Bild 15.2b dahingehend erweitern, dass es von einem Computer ausgeführt werden kann, dessen Maschinensprache den Operationscode SIN nicht enthält. Die Sinusfunktion muss also ausprogrammiert werden. Für kleine Winkel x bietet sich die Reihenentwicklung (15.5) an[1].

$$\sin x = x - x^3/3! + x^5/5! - \ldots + (-1)^n * x^{(2n+1)}/(2n+1)! \ldots. \qquad (15.5)$$

2

Wie unschwer zu erkennen ist, ergibt sich das n-te Glied der Reihe aus dem vorangehenden durch dessen Multiplikation mit $-x^2/(2n(2n+1))$. Dieses Bildungsgesetz eignet sich für die Formulierung eines Programms zur iterativen Berechnung der Sinusfunktion. Bild 15.3 zeigt zwei Möglichkeiten. Beide Programme enhalten eine sog. **Laufanweisung** (die Zeilen zwischen `FOR` und `ENDFOR` bzw. zwischen `WHILE` und `ENDWHILE`), das linke Programm in Form einer **STEP-UNTIL-Anweisung**, das rechte in Form einer **WHILE-Anweisung**. Beide Programme sind unvollständig; u.a. sind die Deklarationen unterdrückt. Nur die Laufanweisungen mit ihren Anfangswertbelegungen sind vollständig wiedergegeben.

---

1 Für Winkel um 90° wäre eine Entwicklung für cos(90°–x) günstiger.

```
s := x;
y := x;

FOR n=2 STEP 1 UNTIL 20 DO
  s := -s*x^2/(4*n^2+2*n);
  y :=  y+s;

ENDFOR
```

(a)

```
s := x;
y := x;
n := 0;

WHILE ABS(s)>0,00001  DO
  n := n+1;
  s := -s*x^2/(4*n^2+2*n);
  y := y+s;
ENDWHILE
```

(b)

Bild 15.3 Programme zur Berechnung der Sinusfunktion gemäß (15.5), (a) - unter Verwendung einer STEP-UNTIL-Anweisung, (b) - unter Verwendung einer WHILE-Anweisung. Das hochgestellte Dach ist das Operationssymbol des Potenzierens.

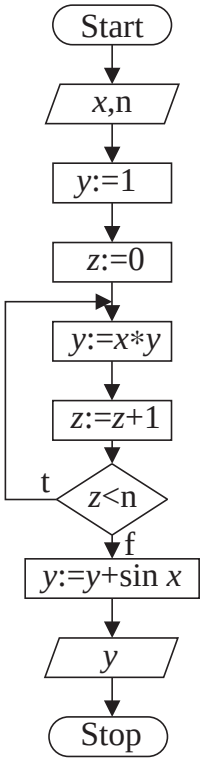
In der zweiten Zeile des linken Programms wird für die Variable `n`, *Laufvariable* genannt, ein Wertebereich von 1 bis 20 festgelegt, den die Laufvariable aufwärts in Einer-Schritten durchläuft. In jedem Schritt wird für den jeweiligen Wert von `n` der Wert des Summanden `s` (des laufenden Gliedes der Reihe) gemäß dem Bildungsgesetz der Reihe berechnet und zur bisherigen Teilsumme hinzuaddiert bzw. von ihr subtrahiert (in der Zeile vor `ENDFOR`). Die Iteration wird bei n=20 beendet. Die Summanden werden mit wachsendem n immer kleiner. Je mehr Glieder der Reihenentwicklung berücksichtigt werden, umso genauer fällt die Gesamtsumme mit dem exakten Wert der Sinusfunktion zusammen.

Oft ist es wünschenswert, das Abbruchkriterium der Iteration so festzulegen, dass der letzte Summand das Endergebnis nicht mehr als um eine vorgegebene Schranke verändert, sodass eine geforderte Ganauigkeit des berechneten Wertes gewährleistet ist. Das rechte Programm trägt diesem Wunsche Rechnung. Die Iteration wird fortgesetzt, solange ("while") das Prädikat `ABS(s) > 0.00001` erfüllt ist, solange also der Absolutwert des laufenden Summanden größer als 0,00001 ist. Sobald er kleiner oder gleich 0,00001 wird, bricht die Iteration ab.

Man beachte folgenden Unterschied zwischen den Abbruchprädikaten der beiden Programme von Bild 15.3. Im linken Programm ist die maximale Iterationszahl, bei der die Iteration abzubrechen ist, von vornherein festgelegt. Im rechten Programm ist das nicht der Fall. Vielmehr ergibt sich der Wert der Variablen im Abbruchprädikat, also der Wert von `ABS(s)`, aus dem Ergebnis der laufenden Iteration. Die Entscheidung, ob abgebrochen wird oder nicht, wird nach der in Kap.8.4.5 besprochenen Methode der *Minimalisierung* getroffen. Das Programm mit der WHILE-Anweisung definiert demzufolge eine μ-*rekursive* Funktion, während das Programm mit der STEP-UNTIL-Anweisung eine *primitiv rekursive* Funktion definiert.

Nach diesen Bemerkungen ist der Leser vielleicht bereits imstande, das Pascal-Programm von Bild 20.1 für die Berechnung der durch (15.1) definierten Funktion zu verstehen.

Mit dem Programm in Bild 15.2b ist zwar eine erheblich bessere Lesbarkeit erreicht im Vergleich zum Programm in Bild 15.2a, doch gibt die lineare Notationsweise als Folge von Anweisungen die Verzweigungsstruktur (die Rückkopplungschleife) nicht sehr anschaulich wieder. Das kann sich bei stark verzweigten Programmen recht unangenehm bemerkbar machen, insbesondere bei verschachtelten Rückkopplungsschleifen und Alternativmaschen. Um die Struktur leichter erkennbar zu machen, ist eine graphische (zweidimensionale) Darstellung sinnvoll. Dafür bietet sich der Aktionsfolgeplan oder der Programmablaufplan (PAP) an, dessen Symbole in Kap.13.5.4 erklärt worden sind (vgl. Bild 13.9). Bild 15.4 zeigt den PAP für das Programm von Bild 15.2b. Jedem rechteckigen Kästchen entspricht eine Aktion, also eine Ergibtanweisung in Bild 15.2b. Das rhombische Kästchen beschreibt den Sprungbefehl. Solange die aktuelle Iterationszahl *z* kleiner ist als die maximale Iterationszahl *n*, d.h. solange das Steuerprädikat im Rhombus erfüllt ist, erfolgt die Übergabe der Steuerung längs des mit t (true) beschrifteten Zweiges zurück zur nächsten Multiplikation. Sobald das Steuerprädikat nicht mehr erfüllt ist, erfolgt die Übergabe längs des mit f (false) beschrifteten Zweiges und der Zyklus wird verlassen.



15.4 Programmablaufplan (PAP) zum Programm von Bild 15.2

15.3 Näherungsverfahren

Aus unseren abstrakten Überlegungen über die Berechenbarkeit von Funktionen wissen wir zwar, dass sich für jede rekursive Funktion ein Maschinenprogramm des Von-Neumann-Rechners erstellen lässt, kurz, dass sie *von-Neumann-berechenbar* ist. Dennoch erhebt sich die Frage, wie die Berechnung in jedem Einzelfall möglich ist. Wir stellen die Frage konkreter. Die Sinusfunktion kann durch verschiedene Prädikate definiert werden, z.B. durch die Differenzialgleichung (4.2) oder durch die Faustregel “Gegenkathete durch Hypothenuse”. Wie lässt sich dieses Verhältnis mit Hilfe der Grundrechenarten rein digital aus dem Winkel, d.h. ohne Längenmessung berechnen? Die Antwort gibt die Reihenentwicklung (15.5).

Da alle rekursiven Funktionen von-Neumann-berechenbar sind, liegt es nahe anzunehmen, dass sich für alle Funktionen wenn nicht exakte, so doch stets Nä-

herungsformeln angeben lassen, die nur die Grundrechenarten enthalten. Dass die Vermutung stimmt, zeigt ein Blick in einschlägige Mathematikbücher, in denen die Reihenentwicklung von Funktionen behandelt wird, d.h. die Zerlegung von Funktionen in Summanden, die nur die Grundrechenarten enthalten und deren Absolutwerte monoton abnehmen. Da die Reihenentwicklung irgendwann abgebrochen werden muss, handelt es sich um eine Näherung. Das braucht uns nicht zu beunruhigen, denn aus Kap.4 wissen wir, dass Rundung eine immanente Notwendigkeit des digitalen Rechnens ist.

15.4 Assembler, Binder, Lader

Ein Programm, das man mit Mühe geschrieben hat, möchte man jederzeit zur Verfügung haben. Man möchte es bequem aufrufen und auf beliebige Eingabewerte anwenden können. Das ist durchaus möglich. Doch muss man sich dabei an die Anfangsadresse des Programms und an die Adressen aller Ein- und Ausgabewerte erinnern. Viel bequemer wäre es, wenn man die Variablenbezeichnungen unmittelbar verwenden könnte und wenn man das Programm über einen Namen, der sich leicht merken lässt, aufrufen könnte. Ein Programm, das über seinen Namen gerufen werden kann, wird **Prozedur** genannt.

Beide Wünsche erfüllt ein Programm, das die vom Programmierer gewählten **Bezeichner** (die Namen von Programmen oder Variablen) durch geeignete Adressen ersetzt. Variablenbezeichner werden auch **Identifikatoren** genannt. Das Ersetzen von Identifikatoren durch Adressen heißt **Assemblieren** und ein assemblierendes Programm heißt **Assembler**. Ein Assembler kann auch dafür eingesetzt werden, geeignet gewählte Operationcodes, also Bezeichner für die Maschinenoperationen, in Interncodes zu übersetzen. Das Assemblieren stellt eine sehr einfache Art des Übersetzens dar und lässt sich ohne besondere Schwierigkeit automatisieren, d.h. programmieren. Ein Übersetzerprogramm überführt einen Programmtext in einen neuen, *äquivalenten* Text, d.h. in einen Text mit gleicher interner Semantik.

Unabhängig davon, aus welcher Sprache in welche Sprache ein Programm übersetzt wird, ist es üblich, das ursprüngliche Programm als **Quellprogramm** oder **Quellcode** und das übersetzte Programm als **Zielprogramm** oder **Zielcode** zu bezeichnen. Dementsprechend heißen die verwendeten Sprachen **Quellsprache** und **Zielsprache**. Im Falle des Assemblers wird das Quellprogramm **Assemblerprogramm** genannt. Es enthält Identifikatoren (Variablenbezeichnern) und Operationscodes (Operationsbezeichner). Da es nur Operationscodes für diejenigen Operationen enthalten darf, die von der “Maschine” (vom Prozessor) angeboten werden, bezeichnen wir Assemblerprogramme ebenfalls als Maschinenprogramme und unterscheiden zwischen *höheren* und *internen Maschinenprogrammen*. Ein internes Maschinenprogramm enthält nur Adressen und intern codierte Operationscodes. Es ist “lauffähig”, d.h. es kann ohne weitere Bearbeitung in den Hauptspeicher “gela-

den" und abgearbeitet werden. Darum wird es auch *ladbares Maschinenprogramm* oder kurz **ladbares Programm** genannt. Auch die Quell- und Zielsprache des Assemblers bezeichnen wir beide als Maschinensprachen und unterscheiden zwischen **höherer** und **interner Maschinensprache**.

Wir wollen uns überlegen, wie der Assembler seine Aufgabe lösen kann. Offenbar 3
muss er zwei Arbeitsgänge ausführen, *Phasen* genannt. In der ersten Phase muss er alle Bezeichner erkennen und "sammeln" ("assemblieren"). Dazu muss er aus dem Assemblerprogrammtext diejenigen Bitketten heraussuchen, die keine Lexeme der internen Maschinensprache sind. Als **Lexeme** werden die kleinsten Einheiten einer Programmiersprache bezeichnet, die interne Semantik tragen.[2] Die Suchprozedur wird **lexikale Analyse** genannt. Sie kann dadurch vereinfacht werden, dass im Assemblerprogramm die verwendeten Variablenbezeichner in einem sog. *Deklarationsteil* aufgelistet, man sagt *deklariert* werden. Die Syntaxregeln vieler Sprachen schreiben vor, dass ein Programm mit einem Deklarationsteil beginnt.

Wenn der Assembler einen neuen Variablenbezeichner erkannt hat, legt er für ihn in der sog. **Symboltabelle** eine Zeile an, in welche später die Attribute (Merkmale) der bezeichneten Variablen eingetragen werden wie Speicherplatzbedarf und Adresse. Der Bezeichner eines Programms ist zweckmäßigerweise durch ein geeignetes vorgestelltes Codewort, z.B. `PROGRAM` oder `PROG`, zu kennzeichnen und an die Spitze des Programms zu setzen. Die Syntax eines Programms könnte in der *Backus-Naur-Form* [5.3] folgendermaßen festgelegt werden, wobei die geschweiften Klammern anzeigen, dass ein Programm beliebig viele Befehle enthalten kann:

Programm → `PROG` *Programmbezeichner Deklarationsteil* {*Befehl*} `ENDE` (15.6)

Um die vollständige Sprachsyntax zu definieren, muss die Syntax jeder der drei kursiv gedruckten Programmglieder - wir hatten sie *metasprachliche Variable* genannt - festgelegt werden. Für "*Befehl*" könnte das im Falle einer Dreiadressmaschine durch die Syntaxregel (5.2) (Kap.5.3) geschehen. Es ergibt sich eine hierarchische Syntaxdefinition, die sich als Syntaxbaum darstellen lässt, ähnlich wie für natürliche Sprachen (vgl.Bild 5.2).

Der Leser führe sich noch einmal Folgendes vor Augen. Nach dem Start eines Programms "liest" der Prozessor "Wort für Wort" den Speicherinhalt, beginnend bei der Anfangsadresse. Sobald er `ENDE` erkennt, bricht er ab. "Lesen" bedeutet "Kopieren in das Befehlsregister (BR)", und "Wort für Wort" bedeutet Adresse für Adresse oder Befehl für Befehl. Das setzt voraus, dass ein Befehl sowohl in eine Speicherzelle als auch in das Befehlsregister hineinpasst. (Dies ist keine prinzipielle Forderung an die Schnittstelle zwischen Hard- und Software.)

---

2 In diesem Zusammenhang wird häufig auch die Bezeichnung *Morphem* verwendet.

Der Aufbau des Befehlsregisters bestimmt das Format des Befehls und legt damit die Lexemgrenzen fest. Demzufolge sind Leerzeichen oder andere Trennzeichen in einem ladbaren Programm überflüssig. In einem Assemblerprogramm sind sie notwendig, falls Bezeichner unterschiedlicher Länge zugelassen sind. Die notwendige Formatanpassung der Software an die Hardware (der Befehle an das Befehlsregister) erfolgt automatisch bei der Adresszuweisung. Das erleichtert die Arbeit des Assemblers.

Die Einfachheit des Assemblers hat noch einen weiteren Grund, der in der Syntaxdefinition liegt. Die Syntax ist so definiert, dass sich eine syntaktische Analyse [5.4] [16.13] erübrigt. Denn für jedes Lexem ist die Lexemklasse (die metasprachliche Variable) durch das Programm- und Befehlsformat eindeutig festgelegt. Es erübrigt sich also auch der Aufbau eines Syntaxbaumes, wie wir ihn in Kap.5.3 für Aussagesätze der deutschen Sprache durchgeführt hatten. Das ändert sich bei höheren Programmiersprachen, wo die Syntaxanalyse eventuell recht aufwendig werden kann.

In der zweiten Phase muss der Assembler das Quellprogramm noch einmal durchgehen und dabei die Bezeichner durch *relative Adressen* (relativ zur Anfangsadresse des Programms) ersetzen, die er den Bezeichnern zuweist. Damit ist die Assemblierung abgeschlossen. Das assemblierte Programm ist ein *verschiebbares* (als Ganzes im Speicher durch Änderung der Anfangsadresse verschiebbares), in einen beliebigen freien Speicherbereich ladbares Programm.

Ein Computer verfügt gewöhnlich über eine große Zahl ladbarer Programme, die in einem peripheren Speicher, z.B. auf einer Festplatte gespeichert sind. Bevor ein ladbares Programm geladen und gestartet werden kann, muss es “gebunden” werden. Das **Binden** besteht i.Allg. aus zwei Schritten. Falls das Programm ein anderes ladbares Programm als Unterprogramm ruft, müssen zunächst die relativen Adressen des Unterprogramms an die relativen Adressen des Hauptprogramm *gebunden* werden. Das erledigt ein Programm namens **Binder**[3]. In diesem ersten Schritt des Bindens wird u.a. die sog. *Parameterübergabe* (Operandenübergabe) vom Hauptprogramm an das Unterprogramm (von der rufenden an die gerufene Prozedur) realisiert. Im zweiten Schritt werden alle relativen Adressen in absolute Adressen umgerechnet, d.h. die Befehle und Variablen des Programms werden an Speicherplätze des Hauptspeichers gebunden. Das erledigt ein Programm namens **Lader**. Der Lader lädt außerdem das Programm in den Hauptspeicher.

Das Binden vor dem Laden muss nicht unbedingt sämtliche Befehle und Variablen eines ladbaren Programms betreffen. In den Kapitel 19 und 20 werden wir Situationen kennen lernen, die ein Binden während der Laufzeit zweckmäßig oder sogar notwen-

3 Anstelle der Wörter *binden* und *Binder* werden unter Informatikern häufiger die eingedeutschten Wörter *linken* und *Linker* verwendet.

dig machen. *Binden vor dem Programmstart wird **statisches Binden**, Binden während der Laufzeit wird **dynamisches Binden** genannt.* 5

15.5 Semantische Lücke

Die Einführung des Assemblers hat bedeutsame Konsequenzen. Sie stellt den ersten, wenn auch zaghaften Schritt zur Lösung eines Grundproblems der Programmierungstechnik und der künstlichen Intelligenz dar, das aus dem Alltag als Sprachbarriere bekannt ist.

Als *Sprachbarriere* wird die Schwierigkeit bezeichnet, mit der jeder konfrontiert ist, der sich mit einem Menschen unterhält, der "eine andere Sprache spricht". Das kann eine Fremdsprache oder auch eine Fachsprache sein, die man nicht kennt. Es kann auch die Sprache einer anderen Kultur, einer anderen Weltanschauung oder einfach die Ausdrucksweise einer völlig anderen Lebensart sein.

Die Aufzählung zeigt, dass die Wurzel der Sprachbarriere entweder in der unterschiedlichen *Artikulierung* von Idemen liegt, also in der Benutzung unterschiedlicher Ausdrucksweisen bzw. Sprachen, oder in einem - möglicherweise sehr tiefliegenden - Unterschied der Ideme der Gesprächspartner, also der Bewusstseinsinhalte, mit denen ihr Denken operiert. Es liegt eine Inkompatibilität der den benutzen Zeichenrealemen zugeordneten Bedeutung (Semantik) vor. Wir nennen sie **semantische Lücke**. Diesen Begriff hatten wir bereits in Kap.5.4 [5.8] im Zusammenhang mit den drei Semantiken eingeführt, ohne näher auf ihn einzugehen.

In Kap.5.4 [5.5] hatten wir vereinbart, die Bedeutung (das Idem) eines Kompositzeichens immer dann als dessen Semantik zu bezeichnen, wenn man annehmen kann, dass sie für alle Beteiligten die gleiche oder zumindest ausreichend ähnlich ist. Wenn das nicht der Fall ist, liegt eine semantische Lücke vor. Sie kann das gegenseitige Verständnis zweier Gesprächspartner unmöglich machen.

Ersetzen wir den einen der Gesprächspartner durch einen Computer, so scheint die Überwindung der semantischen Lücke ein hoffnungsloses Unterfangen zu sein, denn die Lücke ist eher ein Abgrund, der zwischen zwei radikal unterschiedlichen Semantiken klafft, der *externen* Semantik des Menschen und der *internen* Semantik des Computers (Bild 5.3). Dennoch muss eine Brücke über den Abgrund geschlagen werden, sonst könnten Mensch und Computer sich nicht "verstehen", und der Computer könnte dem Menschen nicht helfen.

In Kap.8.6 hatten wir vereinbart, die Semantik, in welcher der Nutzer eines Computers denkt, **Nutzersemantik** zu nennen. Demgegenüber werden wir die Semantik, "in welcher der Computer denkt", **Computersemantik** nennen oder auch **Maschinensemantik**, wenn unterstrichen werden soll, dass die Semantik der Maschinensprache gemeint ist. Man könnte dann auch von der *Semantik des Maschinenkalküls* sprechen. Damit lässt sich das Problem der Schließung der semantischen Lücke prägnanter formulieren: *Die **Nutzersemantik** ist partiell* (soweit es erforder- 6

lich ist) *an die **Maschinensemantik** zu binden*. Dies Problem hatten wir in Kap.5.4 das *technische Semantikproblem* genannt. Das "*semantische Anbinden*" besteht darin, dass an bestimmte Zustände oder Vorgänge im Computer Externsemantik "angebunden" wird, dass ihnen Ideme zugeordnet werden. Das Wort "partiell" bringt zum Ausdruck, dass die externe Semantik nur zum Teil, und zwar i.Allg. zu einem sehr geringen Teil an Hardwarezustände gebunden wird. Der größte Teil der "Kontextsemantik", mit der das Gehirn hantiert (des aktivierten Idemkomplexes) bleibt außerhalb der semantischen Bindung, besitzt also keine Entsprechung im Computer (siehe dazu Kap.17.1).

Damit externe Semantik an Maschinensemantik gebunden werden kann, muss sie durch *Kalkülisierung* in *formale* Semantik überführt werden. Davon war in Kap.8.6 die Rede. Der Nutzer muss also parallel in externer und in formaler Semantik denken. Der Begriff der Nutzersemantik schließt externe und formale Semantik ein. Das gleiche gilt für die Semantik, in der ein angewandter Mathematiker denkt, der z.B. ein Statikproblem in mathematische Formeln fasst.

Es ist sicher nicht übertrieben, das technische Semantikproblem als das Kernproblem des Programmierens, des Sprachentwurfs, der künstlichen Intelligenz und schließlich auch der weltweiten Diskussion um die künstliche Intelligenz und um die Informatik überhaupt zu bezeichnen. Ingenieure, Naturwissenschaftler, Linguisten und Philosophen haben sich mit diesem Problem beschäftigt und werden es auch in Zukunft tun.

Auch wir werden uns noch lange und intensiv mit dem Semantikproblem und seiner Überwindung beschäftigen müssen (siehe Kap.18.3). Doch schon jetzt lässt sich Folgendes sagen. Der Assembler erleichtert das Anbinden der Nutzersemantik an die Maschinensemantik dadurch, dass er dem Nutzer erlaubt, Variable zu verwenden und für sie mnemotechnisch geeignete Bezeichner zu benutzen. Im nächsten Kapitel werden wir das semantische Binden im Falle des numerischen Rechnens im Detail untersuchen.

Zuvor wollen wir versuchen noch deutlicher zu verstehen, warum das Semantikproblem so schwierig ist und wo seine Wurzeln liegen. Dazu knüpfen wir noch einmal an die Vorstellung zweier Gesprächspartner an, von denen einer durch einen Computer ersetzt ist. Diese Vorstellung darf auf keinen Fall zu der Annahme verleiten, der Computer "verstehe" seinen Partner in der umgangssprachlichen Bedeutung, d.h. in der auf die Umgangssprache bezogenen Bedeutung des Wortes "verstehen". Er kann ihn nicht verstehen, weil ihm nur ein ganz geringer Teil der Semantik zur Verfügung steht, in der sein Partner denkt. Der gesamte Kontext, innerhalb dessen der Mensch dem Computer etwas mitteilt, existiert für den Computer nicht und bleibt von der Kommunikation ausgeschlossen.

Der Kontext eines Wortes oder Satzes kann enorm groß sein, so z.B., wenn von den newtonschen Axiomen oder von Beethovens Neunter die Rede ist. Verstehen setzt dann ein entsprechend großes Wissen beim Hörer (Interpretierer) voraus. Der Computer hat ein solches Wissen nicht, denn er verfügt über keine externe Semantik,

er “weiß” nichts von der Welt, zumindest nicht auf dem gegenwärtigen Stand des Computerwissens.

Diese Behauptung kann bei demjenigen Widerspruch provozieren, dem bekannt ist, dass der Computer zur “Wissensverarbeitung” eingesetzt wird. Es fragt sich, wie er Wissen über die Welt verarbeiten kann, ohne über externe Semantik zu verfügen. Um die Frage zu beantworten, muss zunächst der Wissensbegriff definiert werden. Primär handelt es sich um einen personenbezogenen Begriff. *Das **Wissen** eines Menschen ist die Gesamtheit aller Aussagen, an deren Wahrheit er nicht zweifelt. Das Wissen einer Gruppe von Menschen ist das gemeinsame, objektivierte individuelle Wissen der Gruppenmitglieder*.

Wissen ist also eine Menge von Aussagen. Nach unseren grundsätzlichen Überlegungen über den Begriff der Information in den Kapiteln 1 und 2 liegt damit die Antwort auf die gestellte Frage auf der Hand. Mitgeteiltes Wissen ist Information, besteht also aus Realemen und Idemen. Physisch übertragen und im Computer verarbeitet werden nur die Realeme. Das Gehirn fügt ihnen Ideme (Bewusstseinsinhalte, externe Semantik) hinzu. Der Computer leistet *Realemverarbeitung*, genau genommen also keine Wissensverarbeitung oder Informationsverarbeitung, wenn man, wie in diesem Buch, unter Information - und entsprechend unter Wissen - die Gesamtheit von Realem und Idem versteht. Nichtsdestoweniger kann der Computer durchaus den Eindruck erwecken, als denke er in externer Semantik, in den Idemen seines Nutzers. Dieser Eindruck kann mit einem einfachen Trick hergerufen werden, was an einem kleinen Beispiel demonstriert werden soll.

Angenommen, ein Physiker lässt sich bei der Auswertung seiner Formeln (beim numerischen Modellieren) vom Computer helfen. Er hat ein Programm zur Berechnung der Fallgeschwindigkeit einer Kugel in Flüssigkeiten geschrieben. Weil er vergesslich ist, möchte er sich die Resultate so kommentiert ausgeben lassen, dass er ihre Bedeutung auch noch morgen und in einem Monat sofort versteht. Eine Mög- 7
lichkeit ist die Ausgabe in Aussagesätzen, beispielsweise “Die maximale Geschwindigkeit der fallenden Kugel beträgt in Wasser 87 cm pro Sekunde”. Ein solcher Satz kann tatsächlich den Anschein erwecken, als arbeite der Computer mit externer Semantik, was natürlich eine Illusion ist. Die Illusion lässt sich durch eine sehr einfache *Zeichenkettenoperation* hervorbringen. Die Operation besteht darin, dass der Computer in eine vorgegebene Zeichenkette an einer bestimmten Stelle eine aktuelle Zeichenkette (z.B. 87) einfügt. Dies ist ein sehr einfaches Beispiel von **Textverarbeitung**. Noch primitiver ist das Täuschungsmanöver, wenn der Computer nach dem Einschalten seinen Partner mit “Hallo” begrüßt.

15.6 Numerisches Rechnen

Für den programmierenden Computeranwender besteht der Zwang zum semantischen Binden konkret darin, dass er seine Gedanken, genauer seine i.d.R. umgangs-

sprachlich artikulierten Wünsche an den Computer in einer Programmiersprache artikulieren muss, kurz, er muss eine Sprache an eine andere semantisch binden. Das ist oft sehr mühevoll, wenn nicht gar unmöglich. Es gibt jedoch einen Anwendungsbereich, in dem Denken und Programmieren (Algorithmieren) so nahe beieinander liegen, dass sich die semantische Bindung relativ mühelos erreichen lässt, nämlich dann, wenn der Partner (Nutzer) des Computers mathematisch denkt, beispielsweise ein Naturwissenschaftler, der mathematische Modelle der Welt aufstellt und sich dabei vom Rechner helfen lassen will.

Mathematisches Modellieren erfolgt auf zwei Ebenen, auf einer analytischen und einer numerischen. Auf der analytischen Ebene werden Gleichungen aufgestellt und umgeformt, z.B. die Gleichung der Erdbewegung um die Sonne. Dabei wird mit Variablen und Formeln hantiert, der Informatiker spricht von *Formelmanipulation*. Auf der numerischen Ebene werden die Gleichungen numerisch ausgewertet. Dabei werden arithmetische Operationen ausgeführt, d.h es wird *mit Zahlen gerechnet*. Zuweilen wird das Manipulieren mit Variablen (Bezeichnern, Symbolen) *symbolische* Informationsverarbeitung und das Rechnen mit Zahlen *Kalkulation* genannt. Diese Sprechweise werden wir *nicht* übernehmen, sondern vereinbaren:

8 Das Rechnen mit Bezeichnern für Variablen oder Funktionen heißt **analytisches Rechnen**. Das Rechnen mit Werten (Konstanten oder Werten von Variablen) heißt **numerisches Rechnen**. Zahlen sind spezielle Bezeichner von Werten. Analytisches Rechen kann numerisches Rechnen enthalten.

Wir haben die Begriffe des analytischen und numerischen Rechnens bewusst so definiert, dass mit ihnen alle Arten des mathematischen Operierens erfasst sind. Wenn sich zeigen lässt, dass der Computer zur Ausführung jeder numerischen und jeder analytischen Rechnung befähigt werden kann, so folgt daraus, dass er zur Ausführung *jeder* mathematischen Operation befähigt werden kann.

Die Bedeutung der so eingeführten Begriffe fällt nicht in jedem Falle mit der Bedeutung zusammen, in der sie zuweilen in der Literatur verwendet werden. So wird unter numerischem Rechnen häufig ausschließlich das Rechnen mit Zahlen verstanden. Wir dagegen verwenden den Begriff beispielsweise auch dann, wenn in der Prädikatenlogik mit Individuenkonstanten gerechnet wird oder in der booleschen Algebra mit booleschen Konstanten. Die Berechnung des Wertes von $1 \wedge 0$ ist nach obiger Definition eine numerische Rechnung, eine Umformung nach der morganschen Regel hingegen eine analytische Rechnung. Auch das Differenzieren, das Integrieren und das Lösen von Differenzialgleichungen ist analytisches Rechnen. Man beachte, dass das analytische Rechnen mehr umfasst als das Rechnen im Rahmen der Analysis, wenn unter Analysis, wie üblich, dasjenige Gebiet der Mathematik verstanden wird, das sich mit Funktionen auf der Grundlage der Infinitesimalrechnung beschäftigt, also u.a. mit dem Differenzieren, dem Integrieren und dem Lösen von Differenzialgleichungen. Die Bezeichner, mit denen beim analytischen Rechnen gerechnet wird, können auch Wahrscheinlichkeiten, Wahrscheinlichkeitsverteilungen oder Funktionen von Wahrscheinlichkeitsverteilungen benennen. Vor-

aussetzung, dass mit einem Bezeichner gerechnet werden kann, ist, dass er Element einer Kalkülsprache ist.

Je nachdem, ob mathematisches Modellieren auf analytischem oder auf numerischem Rechnen beruht, m.a.W. ob es auf analytischer oder numerischer Ebene erfolgt, sprechen wir von **analytischer** bzw. **numerischer Modellierung**. Numerische Modellierung, die ein Rechner ausführt, wird häufig Computersimulation genannt. Das ist besonders dann üblich, wenn mit dem numerischen Modell "experimentiert" wird. Wir werden im Weiteren unter **Computersimulation** ganz allgemein das *"Nachmachen" irgendwelcher Prozesse oder Handlungen durch den Computer* verstehen, das "Nachmachen" von Denkprozessen eingeschlossen.

Charakteristisch für die Simulation ist, dass der Computernutzer (der Experimentator) mit Bezeichnern für variierbare Parameter arbeitet und durch Zuweisung verschiedener Werte an die Parameter mit dem Simulationsmodell experimentiert. Wenn beispielsweise das Fallen einer Kugel in einer Flüssigkeit simuliert wird, können Durchmesser und spezifisches Gewicht der Kugel und die Viskosität der Flüssigkeit variierbare Parameter sein.

Wir wollen uns überlegen, wie ein Physiker vorzugehen hat, der sich beim numerischen Modellieren helfen lassen will, der z.B. das Simulationsexperiment mit der Kugel ausführen will. Dabei wollen wir davon ausgehen, dass unser Experimentator über ein analytisches Modell der Kugelbewegung verfügt. Zuerst muss er diesem Modell die Form berechenbarer Funktionen geben, genauer die Form arithmetischer Ergibtanweisungen. Dabei wird er sich in der Regel geeigneter Näherungsformeln bedienen müssen. Sodann muss er allen Variablen (Parametern und Resultaten) geeignete Bezeichner zuordnen und schließlich ein Programm schreiben, d.h. die Bezeichner deklarieren und die Funktionen in Befehlsfolgen überführen.

Der Tätigkeit des Programmierens entspricht beim physikalischen Experiment das Aufbauen einer Versuchsapparatur. Diese besteht im Fallbeispiel aus einem Gefäß mit Flüssigkeit und Geräten zum Messen von Längen und Zeitintervallen. Das physikalische Experiment beginnt mit dem Fallenlassen der Kugel, das Simulationsexperiment mit dem Programmstart. Die Resultate werden von der physikalischen Apparatur zunächst in analoger Form geliefert und dann - durch den Experimentator oder durch die Apparatur - in die digitale Form konvertiert. Der Computer liefert sie unmittelbar in digitaler Form.

Die Resultate des Rechners muss der Experimentator *interpretieren*, d.h. er muss ihnen seine eigene Semantik, die *Nutzersemantik* zuordnen. Das kann er, auch wenn das Resultat nicht in Form eines Satzes ausgedruckt wird, denn er weiß, welche Bezeichner was bedeuten. Der Rechner weiß das nicht. Er kennt die Nutzersemantik nicht, sondern nur seine interne Semantik, die *Computersemantik*, d.h. er "weiß" in jedem Augenblick nur, was mit der gerade eingelesenen Zeichenkette zu geschehen hat. Der Computer kennt *nicht* die Bedeutung, die der Nutzer mit den Zeichenketten verbindet, er *operiert* ***nicht*** *mit* ***Bedeutungen*** (vgl. das *Bedeutungsprinzip* [Einleitung.2]).

Die physikalische Apparatur arbeitet nicht mit Zeichen, also auch nicht mit Semantik. Die Apparatur und ihr Verhalten treten unmittelbar als Ideme von Urrealemen in das Bewusstsein des Beobachters. Semantische Bindung erübrigt sich. Dennoch kann auch hier eine Art Interpretieren notwendig sein, nämlich dann, wenn der Experimentator sich im Grunde gar nicht für das Verhalten der Apparatur, sondern für ein ganz anderes Phänomen interessiert, das er mit Hilfe der Apparatur modelliert und zwar analog modelliert. Dabei kann das Original völlig anderer Natur sein. Das "Interpretieren" besteht dann darin, dass der Experimentator physikalische Größen des Modells als physikalische Größen des Originals deutet.

Eine besondere Art von Versuchsapparatur hatten wir in Kap.4.2 kennen gelernt, den Analogrechner. Bei einem Analogrechnerexperiment besteht das "Interpretieren" darin, dass elektrische Größen des Analogrechners als Größen des Originals gedeutet werden, z.B. die Ausgangsspannung als Fallgeschwindigkeit der Kugel, wenn das Fallexperiment mit einem Analogrechner modelliert wird. Diese Art des Interpretierens beinhaltet nicht das "Anbinden" von Bedeutungsinhalten an Zeichen, von Idemen an Realeme, ist also etwas anderes als das, was in Kap.2 als Interpretieren definiert und in Bild 2.1 als Pfeil 5 dargestellt ist. Es beinhaltet dementsprechend auch nicht das Binden interner an externe Semantik, sondern externer an externe Semantik.

Voraussetzung des *analogen* (physikalischen) Modellierens ist, dass das Verhalten von Original und Modell durch dieselben Gleichungen beschrieben wird. Voraussetzung des *digitalen* Modellierens (des Simulierens) ist dagegen, dass die Gleichungen, die das Original beschreiben, implementiert sind. Das Arbeiten mit Gleichungen setzt in jedem Falle die Bindung externer an formale Semantik voraus, also das Interpretieren eines Kalküls (vgl.Kap.5.4 [5.11]).

Nach diesem kleinen Ausflug in das analoge, d.h. nichtsprachliche Modellieren wenden wir uns der eingangs aufgestellten Behauptung zu, wonach semantisches Binden im Falle mathematischer Modellierung relativ einfach ist. Zunächst ist festzustellen, dass ein Maschinenprogramm Operatoren (Operationscodes), Operanden (Variablenbezeichner) und eventuell Weichen (bedingte Sprungbefehle) enthält und dass diese drei Sprachelemente ihre expliziten oder impliziten Entsprechungen sowohl im mathematischen Modell, das als Maschinenprogramm implementiert ist, als auch im Original besitzen. Wir werden das semantische Binden für die drei genannten Sprachelemente getrennt behandeln und beginnen mit der Frage: Wie erfolgt das *semantische Binden der Operatoren* beim numerischen Modellieren?

Die Antwort liegt auf der Hand. Das semantische Binden von Operatoren erfolgt dadurch, dass die arithmetischen Operatoren, mit denen der Programmierer gedanklich arbeitet, im Prozessor implementiert und über Operationscodes aufrufbar sind, die mehr oder weniger genau dem mathematischen Sprachgebrauch entsprechen. Das hat zur Folge, dass das Programmieren zu einem rein syntaktischen Transformieren wird, das lediglich die Substitution von Zeichenketten beinhaltet, beispielsweise der Zeichenkette *a*+*b* durch `ADD a b` oder `(+ a b)`.

Daraus darf allerdings nicht der allgemeine Schluss gezogen werden, die Semantik einer Sprache sei durch ihre Syntax gegeben. Das wäre sicher zu kurz gegriffen. Denn beim Simulieren und bei jeder Verhaltensmodellierung mittels Computer müssen Prozesse, die in dem zu modellierenden Original ablaufen, in Prozesse, die im Computer ablaufen, überführt werden. Wenn das auf rein syntaktische Transformationen zurückgeführt werden kann, ist das sicher die Ausnahme.

Kommen wir nun zur *semantischen Bindung der Operanden,* d.h. der Variablen in den arithmetischen Ausdrücken, aus denen das mathematische Modell besteht. Auch sie macht keine Schwierigkeiten, und wir kennen das Vorgehen bereits. Sie besteht aus zwei Schritten, dem Binden externer Größen (z.B. der Geschwindigkeit der fallenden Kugel aus obigem Beispiel) an Variablen arithmetischer Ausdrücke (das ist die Umkehrung der Interpretation eines Kalküls) und dem Binden der arithmetischen Variablen an Speicherplätze durch den Assembler. Es ist zu beachten, dass Variablenbezeichner vor der Programmausführung durch Zahlen ersetzt werden. Es wird also mit Zahlen gerechnet und nicht mit Variablen, m.a.W. analytisches Modellieren ist auf diese Weise nicht möglich. Wie es möglich wird, werden wir uns in Kap.15.8 überlegen. 9

Dem dritten Sprachelement, dem *bedingten Sprungbefehl*, entspricht nutzersemantisch das Denken in Alternativen und in Wenn-dann-Konstruktionen. Am Beispiel der Zyklusorganisation in Bild 15.2 haben wir gesehen, dass es ziemlich schwierig sein kann, die "Bedeutung" eines Sprungbefehls zu erkennen, ihn semantisch an das Denken zu binden. Wir haben das Problem dadurch gelöst, dass wir Sprachelemente eingeführt haben, die dem menschlichen Denken und Sprachgebrauch näher liegen. Dies sind die bedingte Sprunganweisung und im Falle von Iterationen die Laufanweisung (Bild 15.3). Damit haben wir aber bereits die Grenze zwischen Maschinensprachen und höheren Programmiersprachen überschritten. Man beachte, dass der Programmierer, der seine Wenn-dann-Artikulierungen in Sprungbefehle überführt, im Endeffekt wiederum nur Zeichenketten substituiert, also syntaktische Transformationen durchführt. Es müsste also möglich sein, diese Operation dem Computer zu übertragen, d.h. ein geeignetes Übersetzerprogramm zu schreiben. Auf diesen Gedanken werden wir in Kap.16.4 zurückkommen. Bis dahin gehen wir von der Annahme aus, dass für jedes Nutzerprogramm, das in einer beliebigen formalen Sprache geschrieben sein darf, ein Übersetzerprogramm erstellt werden kann, welches das Nutzerprogramm in ein äquivalentes Maschinenprogramm übersetzt.

15.7 Programmierung von Operatorenhierarchien

Mit der Möglichkeit, Variablenbezeichner zu verwenden, sind die Vorteile, die der Assembler dem Nutzer bietet, nicht erschöpft. Neben den Bezeichnern von Variablen "versteht" der Assembler auch Bezeichner (Namen) von Programmen, d.h.

er erkennt einen Programmnamen und ersetzt ihn durch die Anfangsadresse des betreffenden Programms. Wenn er außerdem einen Sprungbefehl zu dieser Adresse einfügt, sodass sie in den Befehlszähler geladen wird, reagiert der Prozessor auf den Programmnamen genauso wie auf einen Operationscode eines Programms, das im ROM des Matrixsteuerwerks als Firmware abgespeichert ist (vgl. Kap.13.5.5). Diese geringfügige Erweiterung des Assemblers erlaubt dem Programmierer, ein Programm, das er selber erstellt, benannt und im Hauptspeicher abgelegt hat, ebenso zu benutzen wie die Programme der Firmware, d.h. er darf den Programmbezeichner beim Schreiben weiterer Programme praktisch in der gleichen Weise verwenden, wie die Operationscodes der Maschinensprache.

Damit ist die Möglichkeit gegeben, Operatorennetze und Operatorenhierarchien softwaremäßig zu komponieren. Dem schrittweisen Komponieren von Operatoren entspricht ein verschachteltes Aufrufen von Prozeduren (Unterprogrammen). Auf diese Weise lassen sich selbst sehr große Programme effektiv - evtl. in Arbeitsteilung durch ein Programmierteam - und übersichtlich programmieren, sodass sie machbar und lesbar werden. Ein Programm ist gut lesbar, wenn seine Funktionsweise, d.h. wenn seine *interne* Semantik leicht zu erkennen ist. Für jeden, der sich in einem Programm zurechtfinden muss, also auch für denjenigen, der es im praktischen Einsatz wartet, ist seine Lesbarkeit von großer Bedeutung.

Für den Anwender eines Programms hingegen ist nicht die *interne*, sondern die dem Programm zuzuordnende *externe* Semantik für das Arbeiten mit dem Programm ausschlaggebend. Dem Anwender kommt es weniger auf die *Lesbarkeit*, als vielmehr auf die *Verstehbarkeit* des Programms an. Ein Beispiel soll den Unterschied zwischen Lesbarkeit und Verstehbarkeit illustrieren.

Angenommen, ein Unternehmer hat ein Programm in Auftrag gegeben, das seinen Betrieb modelliert. Es soll ihm helfen, die Produktion besser zu organisieren. Um das Programm sinnvoll einsetzen zu können, muss er die dem Programm zugeordnete *externe* Semantik *verstehen*. Er muss die Entsprechungen zwischen seinem gedanklichen, eventuell textlich und graphisch niedergelegten Modell des Betriebes und den Berechnungen, die das Programm ausführt, und den Bezeichnungen, die in den Ein- und Ausgaben des Computers auftreten, kennen. Das muss ihm vom Programmierer erklärt werden. Er muss aber nicht das Programm *lesen* (interpretieren) können, die *interne* Semantik braucht ihm nicht erklärt zu werden. Etwas verkürzt können wir zusammenfassend sagen: *Die **Lesbarkeit** eines Programms betrifft die **interne**, die **Verstehbarkeit** die **externe** Semantik.*

Gute Lesbarkeit ist weitgehend eine Frage des *Programmierstils*. Der Stil eines Programmierers ist - ähnlich wie der Stil eines Schriftstellers - die typische Art und Weise, sich in einer bestimmten Sprache auszudrücken. Er ist das Ergebnis von Gewohnheit, gegebenenfalls auch von Übereinkünften innerhalb eines Teams, aber auch von Geschmack und Mode.

Der besseren Lesbarkeit halber sollten kurze, in sich abgeschlossene Programmbausteine angestrebt und komplizierte Abhängigkeiten zwischen den Bausteinen,

d.h. komplizierte Operandenflüsse vermieden werden. Besonders beeinträchtigt wird die Lesbarkeit durch Überlappungen von Maschen und/oder Schleifen. In Kap.8.4.5 war anhand des Bildes 8.10 gezeigt worden, dass Überlappungen immer eliminiert werden können. Dort hatten wir Kompositoperatoren ohne Überlappungen wohlstrukturiert genannt. Die Erstellung wohlstrukturierter Programme ist eine Grundforderung der sogenannten *strukturierten Programmierung*.

Durch eine gute Programmstruktur kann nicht nur die Lesbarkeit, sondern auch die Verstehbarkeit eines Programms erhöht werden. Dazu muss sie die Struktur des Originals widerspiegeln. Das soll am obigen Beispiel der Betriebsmodellierung demonstriert werden. Wir gehen davon aus, dass der Unternehmer seinen Betrieb hierarchisch organisiert hat und dass die Funktionsweise der Hierarchie und ihrer Bausteine algorithmisch beschreibbar ist. Wenn nun das modellierende (simulierende) Programm eine Operatorenhierarchie darstellt, die der Hierarchie des Betriebes entspricht, wird der Unternehmer die für den Nutzer sichtbare Funktionsweise des Programms schnell erfassen. Wenn zudem die Programmbezeichner mit den Bezeichnungen der Betriebsteile, der Bereiche und Abteilungen der Produktion und der Verwaltung übereinstimmen bzw. an sie erinnern, werden Unternehmer und Angestellte die Ein- und Ausgaben des Programms sehr bald verstehen und sich mit der Anwendung des Programms anfreunden und den Computer als *bequemen* Helfer akzeptieren. Die Bezeichnungen müssen sich gewissermaßen selber “dokumentieren”. 11

Das reicht allerdings nicht aus, um ein Programm leicht verstehbar zu machen; dazu muss es ausführlich dokumentiert werden, am besten dadurch, dass in den Programmtext Kommentare für den Nutzer, aber auch für den Programmierer (nicht für den Computer) eingetragen werden, die wichtigen Programmzeilen ihre *externe* Semantik zuordnen. Es kommt darauf an, dass die “konkrete” (externe) Bedeutung von Programmzeilen, die Bezeichner eingeschlossen, sofort erkennbar ist, auch noch nach Jahren.

Aus abstrakter Sicht kommt es darauf an, dass Computer und Anwender ähnliche Begriffe verwenden, genauer gesagt, dass sie bei der *komponierenden Begriffsbildung* einheitlich vorgehen. In Kap.5.5 war die komponierende Begriffsbildung als spezielle Form der Begriffsbildung eingeführt und am Beispiel des Anzuges (bestehend aus Rock, Hose, Weste) illustriert worden. Dort [5.16] war auch bereits darauf hingewiesen worden, dass komponierende Begriffsbildung vorliegt, wenn im Rahmen einer Komponierungshierarchie ein neues Komposit gebildet wird, das im Weiteren die Rolle eines selbständigen Denkobjektes spielt.

Wir kommen zu folgender Aussage: *Bei der Computermodellierung hierarchisch organisierter Originale kann die semantische Lücke zwischen dem Denken des Anwenders und dem des Programmierers durch* ***einheitliche komponierende Begriffsbildung*** *teilweise überwunden werden.* Der Effekt wird noch verstärkt, wenn der Anwender für die Bausteine des Originals (z.B. des Betriebes) und der Programmierer für die entsprechenden Programmbausteine dieselben Namen (Bezeichner,

Fachausdrücke) verwenden. Die semantische Lücke zwischen dem Programmierer und seinem Auftraggeber verschwindet weitgehend. Beide verstehen sich auf einer abstrakten Ebene, auf der ein Name nur noch ein bestimmtes Verhalten, eine Funktionsweise bezeichnet, abstrahiert von allen technischen, auch programmtechnischen Datails.

12 In der Programmierungstechnik wird dieses Vorgehen **prozedurale Abstraktion** genannt, was zum Ausdruck bringen soll, dass ein Operator bzw. eine Operationsvorschrift (eine Prozedur) als "schwarzer Kasten" behandelt wird und von den Einzelheiten der Prozesse, die im Operator bzw. die bei Ausführung der Vorschrift (der Prozedur) ablaufen, abstrahiert wird. Diese Abstraktion gewinnt praktische Bedeutung, wenn die Vorgänge, die bei einer Operationsausführung ablaufen, für die Umgebung, d.h. für andere Operatoren *unsichtbar* sind, m.a.W. wenn die Operatoren einer Hierarchie (bzw. die in ihnen ablaufenden Prozesse) gegeneinander *abgekapselt* sind.

Das verlangt eine programmierungstechnische Realisierung, die garantiert, dass jeder Baustein der Operatorenhierarchie eine relativ abgeschlossene Einheit darstellt, dessen Verhaltensweise die anderen Bausteine zwar kennen, dessen Innenleben sie jedoch nicht kennen und auch nicht beeinflussen können. Dadurch können unerwünschte Wechselwirkungen zwischen den Prozessen, sog. **Seiteneffekte**, weitgehend ausgeschlossen werden. (In der betriebliche Hierarchie entsprechen Seiteneffekte z.B. der nichtkompetenten Einflussnahme auf fremde Produktionsbereiche.)

Zur Verwirklichung dieser Forderung haben sich die Sprachentwickler viele Varianten ausgedacht. Für einen Programmbaustein, der die Forderung erfüllt, hat sich die Bezeichnung Programmmodul eingebürgert. In diesem Sinne wird auch kurz von **Modul** und **Modulhierarchie** gesprochen. Das am weitesten gehende Konzept der Kapselung von Operatoren und Prozessen verwendet den Begriff des *Objekts* (präziser: des *informatischen* Objekts), auf den in den Kapiteln 18 und 19 eingegangen wird. Bild 20.8 zeigt ein Programm für den Softwareentwurf von Netzen aus gekapselten Operatoren und seine Anwendung auf das Operatorennetz von Bild 8.1. Es ist *objektorientiert* programmiert, wodurch trotz des Umfangs des Programms gute Verstehbarkeit und gute Lesbarkeit erreicht werden.

15.8 Analytisches Rechnen

Wie bereits erwähnt, vollzieht sich mathematisches Modellieren auf zwei Ebenen, auf der numerischen, von der in Kap.15.6 die Rede war, und auf einer analytischen, der wir uns nun zuwenden.

Wer erinnert sich nicht an die unbeliebten eingekleideten Mathematikaufgaben? Nach der verbalen Beschreibung irgendeiner Problemsituation wurde nach dem Wert einer oder mehrere Größen gefragt, z.B. nach dem Treffpunkt zweier sich entgegenfahrender Autos oder nach dem Winkel, unter dem zwei Kirchtürme von einem

bestimmten Punkt aus gesehen werden. Zur Lösung musste man in einem ersten 13
Schritt den richtigen Ansatz finden in Form einer oder mehrerer Gleichungen. Diese mussten in einem zweiten Schritt umgeformt werden, um für jede der gefragten Größen einen Ausdruck aus bekannten Größen zu erhalten. In einem dritten Schritt mussten die gefragten Werte berechnet werden, wofür die Schüler heutzutage eventuell ihre Taschenrechner benutzen dürfen.

Wie ein Taschenrechner oder ein programmierbarer Rechner numerische Aufgaben löst, haben wir ausführlich besprochen. Es stellt sich die Frage, ob man den Rechner auch die beiden ersten Lösungsschritte ausführen lassen kann. Auf die Delegierung des ersten Schrittes (Ansatzfindung) an den Rechner muss offenbar verzichtet werden, wenn der Ansatz sich nicht formal aus der verbalen Aufgabenstellung ableiten ("berechnen") lässt, wenn er vielmehr auf Intuition beruht, also intuitive (kreative, erfindende) Intelligenz erfordert [7.3]. Um die Frage hinsichtlich des zweiten Schrittes, des Umformens, zu beantworten, knüpfen wir an Kap.15.6 an, wo wir zwischen numerischem und analytischem Modellieren unterschieden hatten, und präzisieren die dortigen Vereinbarungen.

*Ein **analytisches Modell** besteht aus einem oder mehreren Sätzen (wahren Aus-* 14
*sagen oder Prädikaten) eines Kalküls über eine oder mehrere Variablen (evtl. auch über Funktionen), die das Original beschreiben (durch das Original interpretierbar sind). Das Umformen der Sätze nach den Regeln des Kalküls ist **analytisches Rechnen*** [8]. Die Sätze können die Form von Ergibtgleichungen (Ergibtanweisungen) oder Relationen (relationale Gleichungen oder Ungleichungen) besitzen [8.20].

Aus der Sicht der praktischen Anwendungen werden von einem mathematischen Modell in der Regel numerische Aussagen verlangt. Diese können unmittelbar nur aus Ergibtgleichungen berechnet werden. In den meisten Fällen liegt das Modell zunächst jedoch in Form von Relationsgleichungen vor, meistens von algebraischen Gleichungen oder von Differenzialgleichungen. Das Überführen in Ergibtgleichungen nennt man **Lösen** der Gleichung. Wenn überhaupt Lösungen existieren, können diese eventuell durch analytisches Rechnen gefunden werden. Ist das nicht möglich, lassen sich die gesuchten Werte unter Umgehung der analytischen Lösung mittels eines geeigneten Näherungsverfahrens numerisch berechnen. Hiermit beschäftigt sich die sog. *numerische Mathematik.*

Am Rande sei erwähnt, dass Differenzialgleichungen als primäre Beschreibung dann zu erwarten sind, wenn physikalische Eigenschaften modelliert werden sollen, denn die Gleichungen der Physik sind in aller Regel Differenzialgleichungen.

In Kap.4.2 war dargestellt, wie *Analog*rechner zur "Lösung" von Gleichungen eingesetzt werden können. In diesem Kapitel wollen wir uns überlegen, wie der *Digital*rechner befähigt werden kann, Gleichungen analytisch zu lösen und ganz allgemein mit analytischen Ausdrücken zu hantieren. Ziel muss dabei nicht unbedingt die Überführung einer Relationsgleichung in eine Ergibtgleichung sein. Ein anderes denkbares Ziel des analytischen Rechnens kann der Beweis sein, dass zwei gegebene Ausdrücke einander gleich sind oder dass ein bestimmter Satz einer formalisierten

Sprache wahr ist. Ein Programm, das derartige Aufgaben löst, wird **Theorembeweiser** genannt.

Der Entwurf eines Programms, das den Computer befähigen soll analytisch zu rechnen, geht, wie im Grunde jede "Beschriftung der tabula rasa" (Kap.7.1), von der Introspektion aus, unabhängig vom Aufgabentyp. So werden auch wir vorgehen. Zuerst überlegen wir uns, wie wir selber verfahren, und dann, ob bzw. wie sich das Verfahren implementieren lässt.

Erinnern wir uns genauer an den Mathematikunterricht. Die meisten Leser werden das Lösen linearer und quadratischer Gleichungen oder das Rechnen mit trigonometrischen Funktionen in mehr oder weniger freundlicher oder auch unfreundlicher Erinnerung haben. Manch einer wird auch das Differenzieren und Integrieren gelernt haben, vielleicht auch, wie einfache Differenzialgleichungen analytisch gelöst werden. All das ist analytisches Rechnen. Mit der Erinnerung daran taucht vielleicht eine Formelsammlung vor dem geistigen Auge auf, in der man nach irgendeiner Formel sucht, entweder nach einer bestimmten, die man vergessen hat, oder nach einer geeigneten, die einem bei der Lösung einer Aufgabe weiterhelfen könnte.

Die Erwähnung der Formelsammlung hat beim Leser, der sich noch an den *Markovalgorithmus* aus Kap.8.4.4 erinnert, möglicherweise ein Aha-Erlebnis ausgelöst, verbunden mit einer Idee, wie sich analytisches Rechnen programmieren lässt. In Kap.8.4.4 war das Vorgehen des Markovalgorithmus mit der Benutzung einer Formelsammlung verglichen worden. Ein Markovalgorithmus verfügt nämlich - in Analogie zur Formelsammlung - über eine *Regelliste*, die angibt, welche Zeichenketten durch welche ersetzt (substituiert) werden dürfen. Der Algorithmus legt die Reihenfolge der Regelanwendungen eindeutig fest, man sagt: das Verfahren ist (der Algorithmus arbeitet) **deterministisch**.

Beim Lösen einer Gleichung, d.h beim Überführen einer relationalen Gleichung in eine Ergibtgleichung geht man analog vor; man substituiert Zeichenketten durch andere. Angenommen, in einer Gleichung tritt der Ausdruck $(1\text{-}\cos^2 x)^{1/2}$ auf. Dann weiß man - z.B. aus einer Formelsammlung - , dass er durch eine ganze Reihe anderer Ausdrücke substituiert werden darf, z.B. durch $\sin x$ oder $\tan x * \cos x$. Im Gegensatz zum Markovalgorithmus hat man die Wahl, mit welcher Substitution man die Rechnung fortsetzt.

15 *Ein Algorithmus (eine Rechenvorschrift, ein Verfahren), der Wahlmöglichkeiten offen lässt, heißt* ***nichtdeterministisch*** oder indeterministisch. Analytisches Rechnen ist in zweifacher Hinsicht nichtdeterministisch. Zum einen kann man vor der Wahl stehen, welchen Teil der gesamten Zeichenkette man substitutieren will, zum anderen, welche konkrete Substitution (welche Formel) man anwenden will. Der Markovalgorithmus beseitigt den zuerst genannten Indeterminismus durch die Vorschrift, dass stets der am weitesten linksstehende substituierbare Teilausdruck substituiert wird. Ein Algorithmus, der im Zweifelsfalle vorschreibt, welcher Teilausdruck zu substituieren ist, heißt **kanonisch**. Es lässt sich zeigen, dass Kanonisierung möglich ist, ohne die Funktion, die der Algorithmus berechnet, zu verändern.

Beim Suchen nach einer passenden Formel kommt es auf die Bezeichner der Variablen nicht an (Arbitrarität der Codierung). Beispielsweise kann in den obigen trigonometrischen Ausdrücken der Winkel auch mit *y* oder mit irgendeinem griechischen Buchstaben bezeichnet sein. Nur muss bei der Verwendung einer Formel die Zuordnung der Bezeichner eindeutig sein. Diese Zuordnung, die man beim Benutzen 16
einer Formelsammlung ganz automatisch macht, nennen wir **Bezeichnerabgleich**.

Wenn man sich für eine aus mehreren möglichen Formeln entschieden hat, muss man damit rechnen, dass der eingeschlagene Weg nicht zum Ziel führt und man einen anderen probieren muss. Dazu wird man im Rechengang bis an den Punkt zurückgehen, wo man sich für den falschen Weg entschieden hat, gerade so wie ein Wanderer, der auf einen Abweg geraten ist. In unwegsamem Gelände verfolgt er die eigene Spur zurück bis zur Wegegabel, wo er falsch gegangen ist. Dieses Bild liegt der Bezeichnung **Backtracking** zugrunde, die sich in der Informatik für das beschrie- 17
bene Zurückgehen beim Suchen eingebürgert hat.

Allgemein kann festgestellt werden: *Das Lösen eines Problems führt bei Anwendung eines nichtdeterministischen Verfahrens auf ein **Suchproblem***. Als Suchmethode bietet sich das Backtracking an, bei dem das Suchen aus wiederholten Versuchen mit eventuellen Rücksprüngen besteht. Wenn die Lösungssuche nicht systematisch erfolgt, sondern in mehr oder weniger zufälligem Probieren besteht, spricht man von **Trial-and-Error-Methode**.

Wie problematisch Suchen sein kann, hat jeder erfahren, der einmal in einer Mathematikarbeit unter Zeitdruck nach dem richtigen Lösungsweg gesucht hat. Natürlich bedarf es nicht derartiger Erinnerungen, um sich die Bedeutung und Problematik des Suchens vor Augen zu führen. Analysiert man das eigene Alltagsverhalten, erkennt man, dass das Suchen eine ziemlich häufige Beschäftigung ist und dass man dabei meistens die Trial-and-Error-, seltener die Backtracking-Methode anwendet. Darum nimmt es nicht wunder, dass Suchen eines der zentralen Probleme nicht nur des analytischen Rechnens, sondern der künstlichen Intelligenz überhaupt ist. Das zeigt ein Blick in die Inhaltsverzeichnisse einschlägiger Lehrbücher[4].

Ein tieferer Blick in ein KI-Lehrbuch lässt den Weg erkennen, den die Informatiker gehen, um den Computer zum Suchen zu befähigen. Es ist der *Standardweg der Introspektion*: Wie ich es mache, so soll es der Computer machen. Dieser Weg hatte schon beim Addieren erfolgreich Pate gestanden. Er hat auch zum Backtracking geführt. Er wird aber problematisch, wenn beim menschlichen Verhalten *Intuition* ins Spiel kommt, wie es beim Suchen häufig der Fall ist. Das Suchproblem berührt also denjenigen Bereich der Intelligenz, den wir zunächst ausschließen wollten. Wir kommen darauf in den Kapiteln 16.3 und 21.3 zurück.

Damit der Computer dem Menschen beim analytischen Rechnen helfen kann, muss sein Speicher mit geeigneter Software gefüllt werden. Dabei wird es sich um

4 Z.B. [Russell 95],[Schefe 87].

ein mehr oder weniger umfangreiches Programmpaket handeln, je nachdem wie "gescheit" der Computer sein soll, wobei intuitive Intelligenz ausgeklammert bleiben soll. Wir nennen das Programmpaket **Analytiksystem**. *Sein zentraler Teil, der sog.* ***Formelmanipulator****, muss drei Bestandteile enthalten, eine* ***Regelliste*** *(Formelsammlung), ein* ***Suchprogramm*** *und einen* ***Substituierer****.*

Die Regelliste muss an die zu lösende Aufgabenklasse angepasst sein. Sie kann, wenn das Analytiksystem universell sein soll, den Umfang eines dicken mathematischen Nachschlagewerkes annehmen. Ob dieses sich implementieren lässt, ist in erster Linie eine Frage des Speicherplatzes und der Speicherorganisation. Von dem an sich unproblematischen Substituieren war in Kap.8.4 wiederholt die Rede in Verbindung mit dem Markovalgorithmus, der funktionalen Substitution und dem Lambda-Kalkül. Am problematischsten ist das Suchen. Wenn man es nicht dem Zufall überlassen will, auf welchem Wege der Computer ein analytisches Problem zu lösen versucht, müssen die vorwiegend indeterministischen Verfahren, die in Mathematikbüchern angeboten werden, in deterministische Verfahren überführt werden.

18 Die bekannte Suche im Labyrinth illustriert sehr anschaulich, wie sich Suchen deterministisch durchführen lässt. Gegen das Verlaufen im Labyrinth wird der Ariadnefaden empfohlen. An ihm kann man sich stets "zurückhangeln" (Backtracking). Außerdem ist es zweckmäßig, beim Suchen systematisch vorzugehen, sodass man nicht zweimal in dieselbe Sackgasse gerät, aber auch keine Möglichkeit unversucht lässt. Folgender Algorithmus, der aus zwei Wenn-Dann-Regeln besteht, leistet dies.

Regel 1. Wenn du an eine Gabel kommst (Einfach- oder Mehrfachgabel), dann wähle den am weitesten links liegenden noch offenen (d.h. nicht als Sackgasse markierten) Weg.

Regel 2. Wenn du in eine Sackgasse geraten bist, dann gehe den Weg, den du gekommen bist, bis zur nächsten Gabel mit einem noch offenen Weg zurück, markiere den Weg, auf dem du steckengeblieben bist, als Sackgasse und gehe gemäß Regel 1 weiter.

Das ist ein deterministischer Algorithmus (in jedem Moment ist der nächste Schritt eindeutig festgelegt), der auf Backtracking basiert (Regel 2). Es liegt nahe, ihn auch beim analytischen Rechnen anzuwenden. Dazu muss die Reihenfolge, in welcher die Formeln der Formelsammlung in jedem Rechenschritt durchzuprobieren sind, festgelegt werden, z.B. in der Reihenfolge, wie sie in der Formelsammlung bzw. im Speicher aufgelistet sind. Das würde heißen, dass nach jeder Formelanwendung die Liste von Anfang an durchmustert werden muss. In dieser Weise verfährt der Markovalgorithmus, wie wir aus Kap.8.4.4 wissen. Er stellt einen deterministischen Sonderfall des allgemeinen Formelmanipulators dar. Natürlich sind intelligentere Verfahren denkbar als das "stupide" Durchmustern. Auf eins von ihnen werden wir gleich zu sprechen kommen.

Es ist zu beachten, das sich das analytische Rechnen insofern deutlich vom Suchen im Labyrinth unterscheidet, als sich praktisch immer eine anwendbare Formel findet. Trotzdem kann Backtracking sinnvoll sein, nicht, weil man in eine Sackgasse geraten ist, sondern weil man sich deutlich vom Ziel entfernt hat. Die Methode, nach der man merkt, dass man sich vom Ziel entfernt hat, lässt sich oft durch Selbstbeobachtung erkennen und demzufolge auch implementieren.

Beim Theorembeweisen kann man beispielsweise versuchen, ein Maß für den Unterschied zwischen dem aktuellen Ausdruck und dem Zielausdruck zu finden und als **Zielabstand** zu definieren. Dieser sollte durch eine Substitution nicht oder nur 19
unerheblich vergrößert, nach Möglichkeit jedoch verkleinert werden. Die Einführung eines Zielabstandes ist ein häufig begangener Weg, den Computer intelligenter suchen zu lassen, d.h. ihn zu befähigen, effektiver zu suchen als mittels phantasielosen Durchmusterns. Der Erfindungsreichtum der Informatiker hat dazu geführt, dass der Computer beim Problemlösen, das auf Suchen beruht, dem Menschen schon heute ebenbürtig, nicht selten sogar überlegen ist. Das betrifft - innerhalb bestimmter Grenzen - z.B. das analytische Rechnen und das Schachspielen. Viele Strategien sind erdacht worden, die den Lösungsweg minimieren sollen. Die Informatiker nennen eine solche Strategie *Auswertungsstrategie*.

Wenn es dem Unerfahrenen so scheint, als wähle der Mathematiker “intuitiv” die 20
richtige Formel oder der Schachprofi den richtigen Zug, so kann der entsprechend programmierte Computer den gleichen Eindruck des Intuitionsbegabten auf den Ahnungslosen machen; doch muss ihm sehr viel Expertenwissen einprogrammiert sein, von dem der Ahnungslose keine Ahnung hat. Damit soll nicht behauptet werden, dass mathematische oder schachspielerische Fähigkeiten lediglich eine Frage des verfügbaren Expertenwissens sind.

Es ist an der Zeit, auf eine scheinbar widersprüchliche Schlussfolgerung einzugehen. Wer sich nach allem Bisherigen davon hat überzeugen lassen, dass der Rechner einerseits nichts kann, als rekursive Funktionen berechnen, dass er andrerseits aber analytisch rechnen kann, der muss schlussfolgern, dass analytisches Rechnen nichts anderes ist, als das Berechnen rekursiver Funktionen. Die Schlussfolgerung ist scheinbar widersprüchlich. Denn was haben diese beiden Tätigkeiten miteinander gemein?

Die Frage provoziert den Verdacht, dass die beiden Tätigkeiten vielleicht ebensoviel miteinander gemein haben, wie der Markovalgorithmus mit den rekursiven Funktionen. Denn in Kap.8.4.4 war (ohne Beweis) gesagt worden, dass der Markovalgorithmus rekursive Funktionen berechnet, und in diesem Kapitel haben wir festgestellt, dass er ein Sonderfall der Formelmanipulation ist.

Der kritische Leser wird sich damit nicht zufrieden geben, denn die Behauptung in Kap.8.4.4, der Markovalgorithmus berechne rekursive Funktionen, hat sich auf Autoritäten berufen und war nicht das Resultat eigenen Nachdenkens. Wir werden die Richtigkeit der Behauptung auch jetzt nicht mathematisch beweisen. Doch wollen wir versuchen, sie plausibel zu machen.

Zunächst überzeugen wir uns, dass sich eine konkrete analytische Rechnung durch einen *Aktionsfolgeplan* beschreiben lässt. Zentraler Akteur, der die Ausführung des Plans steuert, ist der Prozessor eines Computers. Der gesamte Inhalt des Arbeitsspeichers übernimmt jetzt die Rolle der Zeichenkette, die schrittweise verändert wird. Einer *Substitution* entspricht eine *Aktion* des Aktionsfolgeplans und damit eine Operation des Prozessors, bei der ein Speicherbereich überschrieben, d.h. sein Inhalt *substituiert* wird. Wenn bei der Formelmanipulation mehrere Formeln anwendbar sind, entspricht das einer Weiche im Aktionsfolgeplan. Die Weichenstellung (Entscheidung der Alternative bzw. Fallauswahl) nimmt der Prozessor gemäß Suchvorschrift vor.

Auf diese Weise lässt sich jede konkrete analytische Rechnung als Aktionsfolgeplan darstellen. Da nun aber jeder solche Plan (jedes Aktionsfolgeprogramm) eine rekursive Funktion berechnet [13.17], folgt, dass beim deterministischen analytischen Rechnen rekursive Funktionen berechnet werden.

Die Argumentation behält ihre Gültigkeit, wenn die mathematische Formelliste durch eine Liste beliebiger Substitutionsregeln ersetzt wird. Dadurch wird die Formelmanipulation zur *Zeichenketten*- oder *Wortmanipulation*. Die Darstellbarkeit als Aktionsfolgeplan bleibt dadurch unberührt und auch ihre Konsequenzen:

Konsequenz 1: Jede von einem deterministischen Wortmanipulator berechnete Funktion ist eine rekursive Funktion.

Diese Konsequenz ist offensichtlich auch in umgekehrter Lesart richtig:

Konsequenz 2: Jede rekursive Funktion kann von einem deterministischen Wortmanipulator berechnet werden,

denn erstens lässt sich zu jeder rekursiven Funktion ein Aktionsfolgeplan für ihre Berechnung angeben, und zweitens lässt sich die Ausführung jedes Aktionsfolgeplans als deterministische Wortmanipulation darstellen. Daraus folgt die Konsequenz 2. Sie wird fast zu einer Trivialität, wenn man bedenkt, dass jede Berechnung durch einen Computer letzten Endes von der ALU ausgeführt wird und dass die ALU Bitketten substituiert, also Wortmanipulationen vornimmt. Aus den Konsequenzen 1 und 2 folgt der

Satz: Die Klasse der durch deterministische Wortmanipulation berechenbaren Funktionen ist mit der Klasse der rekursiven Funktionen identisch.

Dieser Satz gilt auch für den speziellen Fall der Markovfunktionen (der nach dem Markovalgorithmus berechenbaren Funktionen), denn der Markovalgorithmus ist ein spezieller deterministischer Wortmanipulator. Damit ist die Richtigkeit der Behauptung aus Kap.8.4.4 nachgewiesen, dass die Klasse der Markovfunktionen mit der Klasse der rekursiven Funktionen identisch ist.

Unsere Schlussfolgerungen gelten auch für numerische Rechnungen, denn arithmetische Funktionen sind rekursive Funktionen. Folglich müssen auch sie sich als Wortmanipulation auffassen lassen, d.h. auch sie müssen Substitutionsregeln befolgen. Eine von ihnen lautet beispielsweise: “2+2 ist durch 4 zu ersetzen”. Jede Zeile

des kleinen Einmaleins ist eine Regel. Die vollständige Regelliste des numerischen Rechnens ist offenbar ziemlich lang, tatsächlich ist sie unbeschränkt.

Die Überlegungen haben noch einmal verdeutlicht, was der Computer letzten Endes tatsächlich macht: er substituiert. Zudem haben sie uns einen Einblick in Zusammenhänge zwischen unterschiedlichen Methoden der Algorithmenbeschreibung gewährt, die in Kap.8.4 ziemlich zusammenhanglos dargelegt worden sind und auf den ersten Blick wenig miteinander zu tun zu haben schienen. Folgende Bemerkungen mögen diese Einsicht noch ein wenig vertiefen.

Wir haben gesehen, dass sich jedes Rechnen, numerisches wie analytisches, auf ein und dieselbe Grundoperation, auf das Substituieren zurückführen lässt. Das gilt nicht nur für das Kopfrechnen und das Rechnen mit Papier und Schreibstift, wie wir es in der Schule gelernt haben, sondern auch für den Computer. Wenn der Prozessor eine TNS-Operation ausführt (Transport Nach dem Speicher), substituiert er den alten Speicherplatzinhalt durch einen neuen. Dabei wird der neue Inhalt entweder einem anderen Speicherplatz entnommen oder er wird von der ALU berechnet.

Es war wohl die zunächst intuitive Überzeugung, dass die Zeichenkettensubstitution als einzige Grundoperation ausreicht, um jedes beliebige algorithmische Verfahren zu beschreiben, die MARKOV zur Definition seines Normalalgorithmus animiert hat und CHURCH zur Definition des Lambda-Kalküls [8.33]. Vielleicht haben ähnliche Vorstellungen TURING zur Konstruktion seiner Maschine angeregt, wobei er nur ganz elementare Operationen zuließ. Dementsprechend begrenzte er das Substituieren auf das Ersetzen eines einzigen Zeichens. Die Regelliste und die Vorschrift für die Weichensteuerung (Bewegung des Schreib-Lesekopfes) schloss er in die Automatentabelle ein.

Damit beenden wir unsere Untersuchung zur Frage, wie der Computer zum "mathematischen Assistenten" qualifiziert werden kann, d.h. wie man ihn befähigen kann, dem Menschen beim Lösen mathematischer Aufgaben zu unterstützen. Die Untersuchung betraf lediglich die prinzipielle Seite der Fragestellung. Was konkret getan werden muss, um den Computer zu qualifizieren, m.a.W. um die tabula rasa mit der notwendigen Software zu beschriften, davon war bisher nicht Rede. Auch die Frage der Nutzersprache ist kaum diskutiert worden. Wir verschieben sie auf die Kapitel 16.4, 18 und 20.

Wir hatten in Kap15.6 [15.8] die Begriffe des analytischen und numerischen Rechnens so definiert, dass mit ihnen das Rechnen in jedem beliebigen Kalkül, m.a.W. alle Arten des mathematischen Operierens erfasst sind. Wir haben gesehen, dass jedes numerische und jedes analytische Rechnen ein Manipulieren mit Formeln ist und dass sich Formelmanipulation algorithmieren und damit implementieren lässt. Damit lautet das Resümee dieses Kapitels: *Der Computer (Prozessorrechner) kann zur Ausführung **jeder** mathematischen Operation befähigt werden.*

15.9 Bemerkung zur Programmübersetzung

Wenn ein Programm ausgeführt werden soll, das nicht in der Maschinensprache des ausführenden Computers programmiert worden ist, muss es zunächst übersetzt werden. Auf das Übersetzungsproblem wird in Kap. 16.4 eingegangen. Wir wollen uns aber schon jetzt fragen, wie im Prinzip vorzugehen ist, um das Programm von Bild 15.3b in das von 15.3a zu übersetzen. Die Antwort wird durch die einheitliche Grobstruktur (Folge von Aktionsvorschriften) beider Programme nahegelegt. Sie lautet: Die Aktionsvorschriften (Anweisungen) auf der rechten Seite von Bild 15.3 sind durch die entsprechenden Aktionsvorschriften (Befehle oder Befehlsfolgen) auf der linken Seite zu ersetzen. Dies ist offensichtlich das Grundprinzip, nach dem eine imperative Sprache in eine Maschinensprache zu übersetzen ist. Die beiden Sprachen unterscheiden sich im Wesentlichen nur in der Syntax der Aktionsvorschriften.

Fragt man aber, wie eine funktionale Sprache in eine Maschinensprache übersetzt werden kann, gibt es keine so einfache Antwort. Wir wollen versuchen, die Frage für den funktionalen Ausdruck (15.4b) zu beantworten. Er lautete

```
(defun pot(x n)(if(= n 0)1(* x(pot x(- n 1)))))
```
(15.7)

Wir erinnern daran, dass durch einen Ausdruck der Form `(defun f(..)(...))` die Funktion `f(..)` durch den nachfolgenden Klammerausdruck definiert wird, im Falle von (15.7) durch die If-Funktion

```
(if(= n 0)1(* x(pot x(- n 1))))).
```

Das Prinzip der aktionsweisen Übersetzung ist auf funktionale Programme nicht anwendbar. Ein anderes Prinzip ist gesucht. Um es zu finden, erinnern wir uns an die Wurzeln des funktionalen Programmierens, an den Lambda-Kalkül. In Kap.8.4.7. waren Funktionen mit Hilfe des Lambda-Operators durch funktional notierte Ausdrücke (Lambda-Ausdrücke) definiert worden. Wir hatten uns überlegt, wie eine so definierte Funktion "ausgewertet" werden kann, m.a.W. wie die Funktion berechnet werden kann. Die Idee liegt nahe, die dortige Methode auf unser Übersetzungsproblem anzuwenden. Das bedeutet, dass zunächst alle Substitutionen, die möglich sind, ausgeführt werden, denn die "Auswertung" eines Lambda-Ausdrucks begann mit der *Lambda-Eliminierung* [8.35] mittels Substitution. Damit das Prinzip der Methode deutlich sichtbar wird, substituieren wir zunächst nur eine der beiden Variablen durch einen Wert und zwar `n` durch denWert `3`. Dann geht (15.7) in (15.8) über:

```
(defun pot(x 3)(if(= 3 0)1(* x(pot x 2))).
```
(15.8)

Das Prädikat `(= n 0)` ist nicht erfüllt. Folglich ist die If-Funktion (laut Sprachdefinition von CommonLisp) durch den zweiten Ausdruck nach dem Prädikat, also

durch `(* x(pot x 2))` zu substituieren (der erste Ausdruck nach dem Prädikat ist der Wert `1`). Das ergibt

```
(defun pot(x 3)(* x(pot x 2))).
```

Nach dem gleichen Vorgehen ist der Reihe nach `(pot x 2)`, `(pot x 1)` und `(pot x 0)` zu substituieren. Die letzte Substitution liefert für die If-Funktion den Wert `1`. Das Resultat aller Substitutionen lautet

```
(defun pot(x 3)(* x(*x(*x 1)))).                    (15.9)
```

In der bisherigen Prozedur wird der Leser den ersten Teil einer rekursiven Berechnung erkannt haben. In Kap.8.4.6 hatten wir sie am Beispiel der Fakultät-Funktion durchexerziert. Es folgt nun der zweite Teil, die *Wertberechnung*, wie wir ihn in Kap.8.4.7 [8.35] genannt hatten. Er bereitet keine Schwierigkeiten, denn jeder Ausdruck der Form `(* x u)` steht für das Resultat `r` einer Ergibtanweisung `r := x*u)`. Demzufolge lässt sich (15.9) in eine Folge von Multiplikationsaktionen überführen:

```
a := x*1
b := x*a
y := x*b.
```

Darin sind `a` und `b` Hilfsvariablen und `y` bezeichnet den Wert `pot(x 3)`. Damit ist der funktionale Ausdruck (15.8) imperativ notiert. Der nächste Schritt ist die Übersetzung in die Maschinensprache, die Zuweisung eines Wertes an `x` und die Abarbeitung.

Nach diesem Prinzip werden funktionale Programme übersetzt und ausgeführt, man sagt *interpretiert*. Das interpretierende Programm heißt **Interpreter**. Auch logische Programme werden interpretiert, worauf in Kap.20.2.4 eingegangen wird. Das Interpretieren eines funktionalen oder logischen Programms beginnt stets mit dem Substituieren. Wenn in dem Programm keine rekursiven Funktionsdefinitionen auftreten (wie beispielsweise die Definition der Potenzfunktion), enthält die Substitutionsprozedur keine rekursiven Iterationen, sondern besteht aus einer Folge einzelner Substitutionen.

16 Lösen nichtmathematischer Probleme

Zusammenfassung

Mit der Wortmanipulation ist ein erster Schritt in den “nichtmathematischen” Bereich getan, denn die umzuformenden Zeichenketten müssen aus der Sicht des Menschen, der sich vom Computer helfen lässt, nicht Ausdrücke eines mathematischen Kalküls sein. Im nächsten Schritt werden Substitutionsregeln mit externer Semantik zugelassen, die dem alltäglichen logischen Schlussfolgern zugrunde liegen, beispielsweise der Schlussfolgerung, dass der Sohn der Schwiegermutter meiner Mutter entweder mein Vater oder mein Onkel ist. Menschliches *Schlussfolgern* im gängigen Sinne des Wortes ist ein Ableiten von Schlüssen aus gegebenen Fakten nach den Regeln des logischen Denkens, wobei aus introspektiver Sicht des denkenden Menschen das Ableiten nicht explizit formalisiert ist.

Um Schlussfolgern zu implementieren, muss es zunächst kalkülisiert werden. Dazu sind die Fakten und Regeln in der formalen Sprache eines Kalküls, beispielsweise in der Sprache des Prädikatenkalküls (der Prädikatenlogik) zu formulieren. Das Schlussfolgern kann dann auf formalem Wege nach den Regeln des Prädikatenkalküls erfolgen. Dabei wird von jeder externen Semantik abstrahiert, nachdem sie an die formale Semantik des Prädikatenkalküls angebunden worden ist. Kalkülisierung betrifft immer einen speziellen Bereich des Denkens (des Nachdenkens), den sog. *Diskursbereich*, auf den sich das externsemantische *Fakten*- und *Regelwissen* bezieht. Insofern stellt das Ergebnis der Kalkülisierung einen *speziellen Denkkalkül* dar.

Auf diese Weise wird das Schlussfolgern zum formalen Ableiten. Es besteht aus Syntaxvergleichen, Substitutionen und Transformationen gemäß Regeln. Diese Art des Schlussfolgerns wird *Inferenzieren* genannt. Eine formale Schlussfolgerung, eine *Inferenz*, kann von einem Computer vollzogen werden, der über ein einschlägiges Inferenzprogramm und über das erforderliche *Fakten*- und *Regelwissen* verfügt.

Ein Inferenzprogramm zusammen mit dem Fakten- und Regelwissen eines bestimmten Anwendungsgebietes (z.B. der Buchführung oder der Konstruktion von Motoren) heißt *Expertensystem* für das betreffende Gebiet. Ein Expertensystem muss über ein ausreichend universelles und effizientes Inferenzprogramm, über eine geeignete Dialogsprache und über ein ausreichendes und leicht abrufbares Wissen über das Fachgebiet (über den Diskursbereich) verfügen.

Ein Expertensystem kann Leistungen vollbringen, die als intuitive Leistung oder als Erfindung bezeichnet werden können. Beispielsweise könnte ein ausreichend “qualifiziertes” Expertensystem für Chemie in der Lage sein, aus dem Wissen, das KEKULÉ besaß, die Formel des Benzolringes abzuleiten, d.h. die Formel zu “erfinden”, wie KEKULÉ sie erfunden (intuitiv gefunden) hat. Wenn eine Aussage auf intuitivem

Wege gefunden wird, obwohl sie durch Ableitung hätte gefunden werden können, sprechen wir von *reduzibler Intuition*.

Voraussetzung für die Implementierung irgendeines speziellen Denkkalküls ist die automatische Übersetzung der betreffenden Kalkülsprache in die Maschinensprache des verwendeten Rechners, m.a.W. die Verwirklichung der vierten Grundidee des elektronischen Rechnens. Aus der Sicht des übersetzenden Menschen, beispielsweise eines Dolmetschers, ist Übersetzen eine nichtmathematische Leistung menschlicher Intelligenz. Doch ist sie kalkülisierbar und kann vom Computer erbracht werden.

Das Übersetzen kann durch einen Interpreter oder einen Compiler ausgeführt werden. Ein *Interpreter interpretiert* die Sätze (die interpretierbaren Programmstücke) der Sprache des speziellen Denkkalküls, d.h. er übersetzt die einzelnen Programmstücke und führt sie aus. Ein *Compiler* übersetzt ein Programm im Ganzen aus der Eingabesprache, auch *Quellsprache* genannt, in die Ausgabesprache, auch *Zielsprache* genannt. Zielsprache ist die Maschinensprache, der sog. Objektcode für den Binder.

Ein Compiler (Übersetzerprogramm) besteht aus einem Scanner, einem Parser und einem Codegenerator. Der *Scanner* führt die *lexikale Analyse* durch, d.h. er klassifiziert die einzelnen Wörter (die Lexeme) des Quellprogramms. Beispielsweise können alle Variablenbezeichner zu einer Klasse, alle Operationssymbole zu einer anderen Klasse zusammengefasst werden. Der *Parser* führt die *syntaktische Analyse* durch, d.h. er erkennt und klassifiziert syntaktische Konstrukte und benennt sie mit metasprachlichen Klassennamen. Beispielsweise erkennt er eine Folge Bezeichner-Ergibtzeichen-Ausdruck als Ergibtanweisung. Aus den metasprachlichen Klassen baut der Parser eine hierachische Struktur auf, welche die syntaktische Struktur des Quellprogramms vollständig und ohne Rest wiedergeben muss. Der *Codegenerator* überführt diese Struktur in die Zielsprache, indem er die Konstrukte der Quellsprache durch Konstrukte der Zielsprache gemäß einer Regelliste substituiert.

Das Vorgehen ist dem nichtmaschinellen Übersetzen zwischen natürlichen Sprachen abgeschaut und kann auf diese angewendet werden. Die syntaktische Strukturierung, mit anderen Worten die Verwendung grammatikalischer Regeln, ist das entscheidende Mittel natürlicher wie künstlicher Sprachen zur Erhöhung der Aussagekraft einer Sprache, zur Steigerung der semantischen Dichte. Bei Verwendung *generativer Grammatiken* kann die aufwendige syntaktische Analyse sehr effektiv gestaltet werden, und es lassen sich Regeln angeben, nach denen Programmiersprachen zu entwerfen sind, um sie übersetzerfreundlich zu machen.

16.1 Schlussfolgern

Wir waren zu der Einsicht gelangt, dass ein Problem, das der Computer lösen soll, in kalkülisierter Form vorliegen muss, damit eine Lösungsvorschrift programmiert

werden kann. Eingedenk dieser Einsicht wird der Leser die Überschrift “Lösen nichtmathematischer Probleme” richtig verstehen. Um dennoch mögliche Missverständnisse zu vermeiden, soll die Überschrift ausführlicher artikuliert werden: “Lösen von Aufgaben durch den Computer, die vom Menschen auf nichtmathematischem Wge gelöst werden”. Nur aus der Sicht des Menschen hat es Sinn, ein Problem als “nichtmathematisch” zu bezeichnen. Für den Computer gibt es keine nichtmathematischen Probleme (siehe Kap.14.2).

In Kap.7 hatten wir verschiedene Wege herausgearbeitet, auf denen der Mensch zu neuen Aussagen über die Welt und zu neuen sprachlichen Modellen der Welt gelangen kann, den deduktiven, den assoziativen und den intuitiven Weg. Dementsprechend hatten wir zwischen *deduktiver, assoziativer* und *intuitiver Intelligenz* (Fähigkeit zum sprachlichen modellieren) unterschieden. Deduzieren (ableiten) kann entweder *Rechnen* oder *Schlussfolgern* sein. In Kap.15 haben wir uns überlegt, wie der Computer rechnen kann, und zwar rechnen in einem *beliebigen* Kalkül. Jetzt werden wir uns überlegen, ob der Computer auch zum Schlussfolgern und zum Assoziieren befähigt werden kann. Wir werden sogar die Frage aufwerfen, ob er mit intuitiver Intelligenz ausgestattet werden kann. Der Leser erkennt, das wir im Sinne des Gegenläufigkeitsprinzips vorgehen.

Auf dem Wege zur künstlichen Intelligenz hat uns der Prozessorrechner mit seinem - wenn auch recht primitiven - Sprachverständnis bisher nicht im Stich gelassen. Er scheint allen gängigen mathematischen Problemen gewachsen zu sein. Nur muss er, genauso wie der Mensch, “gelernt haben, mit den Problemen umzugehen”. Die Art und Weise des Lernens ist allerdings kaum mit der des Menschen zu vergleichen. Dass die Wiederholung die Mutter der Wissenschaft (des Wissens und Könnens) ist, trifft auf den Computer nicht zu. Es genügt, wenn ihm der “Lehrstoff” einmal vermittelt (“eingetrichtert”) wird, allerdings genau und bis in alle Einzelheiten. Wiederholen und Üben erübrigt sich. Daran wird sich nichts ändern, wenn wir nun die Gefilde der Mathematik verlassen, d.h. den Bereich, in dem der Mensch durch Rechnen zu neuen Aussagen gelangt, und die “Reise zur KI” fortsetzen, auf der Suche nach der Grenze der künstlichen Intelligenz. Wir bleiben, getreu unserer Zielstellung, auf der symbolischen Betrachtungsebene [1.4], und unser Werkzeug ist der Prozessorcomputer, nicht der Neurocomputer.

Es mag zweifelhaft erscheinen, dass der Computer, der “eigentlich doch nur” rekursive Funktionen berechnen kann, imstande ist, logische Schlussfolgerungen zu ziehen, dass er sinnvolle Assoziationen haben oder irgendetwas Sinnvolles intuitiv finden oder erfinden kann. Es wird sich zeigen, dass der Zweifel zum Teil unseren “Denkgewohnheiten über das Denken” entspringt und dass auch intuitives und erfinderisches Denken in gewissem Grade durchaus simulierbar ist. Außerdem werden wir erkennen, dass das “Feld der Mathematik” ein viel weiteres ist, als dasjenige, welches einem in der Schule gezeigt wurde. Bevor wir dieses Feld betreten, vergegenwärtigen wir uns noch einmal unser Anliegen.

Wir wollen verstehen, was es mit der künstlichen Intelligenz auf sich hat, was man von ihr erwarten kann und was sie offenbar nicht zu leisten vermag. Wir begnügen uns damit, zu erkennen (nachzuerfinden), wie es *im Prinzip* möglich ist, dass der Computer auf der Grundlage ihm bekannten Wissens zu neuen Einsichten gelangt, die sich scheinbar nicht durch numerisches oder analytisches Rechnen ableiten lassen. Der Zusatz "im Prinzip" bedeutet, dass wir unsere Überlegungen abbrechen, sobald wir den "prinzipiellen" Weg erkannt haben. Bekanntlich liegt der Teufel im Detail, d.h. in den Schwierigkeiten, die im konkreten Fall zu überwinden sind. Sie erfordern eventuell umfangreiche theoretische Untersuchungen, die über das Anliegen und den Rahmen des Buches hinausgehen. Wir brechen also genau da ab, wo es für die Theoretiker anfängt interessant zu werden[1].

In der Hoffnung, "auf den ersten Blick" Grenzen der künstlichen Intelligenz zu erkennen, betrachten wir eine Eigenschaft, die als *Findigkeit* (geistige Wendigkeit, Pfiffigkeit, Einfallsreichtum u.ä.m.) bezeichnet wird. Das ist eine Eigenschaft, die Menschen in mehr oder weniger hohem Grade besitzen, über die der Computer aber offenbar nicht verfügt. Rätsel sind ein beliebtes Mittel, die Findigkeit eines Menschen auf die Probe zu stellen, seine "Intelligenz" zu testen. Was liegt näher, als diese Methode auf den Computer anzuwenden, wenn es darum geht, die Grenzen seiner
1 Intelligenz zu erkennen? Nehmen wir das Rätsel aus dem Ödipusmythos: "Am Morgen geht es auf vier Beinen, am Mittag auf zwei, am Abend auf drei. Was ist das?" Wie kann der Computer befähigt werden, die Antwort (der Mensch) zu "erraten"? Eine triviale Methode bestünde darin, die Lösung "vorzusagen". Würde man sämtliche Rätsel der Welt samt ihrer Lösungen abspeichern, könnte der Computer alle Rätsel "lösen".

Diese Methode, die im Grunde ein Nachplappern oder Nachschlagen ist, hat mit Intelligenz scheinbar nicht viel zu tun. Tatsächlich ist sie genauso "intelligent", wie die maschinelle Berechnung einer booleschen Funktion, deren Wertetafel im Computer abgespeichert ist. Man ist geneigt, derartiges einfaches "Nachschlagen" nicht als intelligente Methode anzuerkennen. Unwillkürlich meint man, eine intelligente Problemlösungsmethode müsste mit Nachdenken, mit logischem Schließen oder mit Ableiten, mit Deduzieren zu tun haben. Für das Lösen mathematischer Aufgaben durch numerisches oder analytisches Rechnen trifft das offensichtlich zu. Darum ist es verständlich, dass der Computer sie lösen kann. Doch wie kann er ohne "Vorsagen" *nicht*mathematische Aufgaben lösen, für die es keine Rechenregeln zu geben scheint? Wie kann der "Rechner nichtmathematisch denken"? Dass er es kann, hatten wir das "Paradoxon der KI" genannt.

Wir stehen vor der Frage, wie sich die Grenze der simulierbaren Intelligenz über das in Kap.15 besprochene Rechnen hinaus verschieben lässt. In den vorangehenden

---

1 Dem interessierten Leser steht eine umfangreiche Literatur zur Verfügung, u.a. [Schefe 87], [Schöning 89], [Russel 95].

Kapiteln haben wir angedeutet, wie sich die Kalküle der Arithmetik und der Analysis[2] und überhaupt jeder analytische Kalkül in den Maschinenkalkül, also letzten Endes in den booleschen Kalkül abbilden lässt. Jetzt erhebt sich die Frage, wieweit sich menschliches Denken, das scheinbar nicht, zumindest nicht bewusst, nach den Regeln irgendeines Kalküls abläuft , kalkülisieren lässt, sodass es simulierbar wird.

Kehren wir zu unserem Rätsel zurück und suchen nach irgendwelchen Anhaltspunkten, wie der Mensch die Lösung möglicherweise finden könnte. Dabei stellt sich heraus, dass eine Art Findigkeit erforderlich ist, über die auch die natürliche Intelligenz nur selten verfügt. Wenn jemand das Rätsel ohne Hilfe "errät", hat er es wahrscheinlich gekannt. Möglich wäre allerdings, dass einem hellen Kopf durch *Assoziation* "einfällt", dass der Mensch *vier* Extremitäten besitzt, die er zunächst alle vier, später aber nur zwei von ihnen zur Fortbewegung benutzt. Das Finden der Antwort scheint aber nicht kalkülisierbar zu sein, zumindest nicht so ohne Weiteres. Auf jeden Fall muss das Problem in einer geeigneten verallgemeinerten, abstrakteren Form beschrieben werden.

Im Augenblick wollen wir den "assoziativen Weg" nicht weiter verfolgen, sondern nach Rätseln suchen, deren Lösungen zwar nicht "mathematisch" (im üblichen Sinne des Wortes), aber doch "irgendwie" ableitbar zu sein scheinen, die sich in gewissem Sinne kalkülisieren lassen. Das ist immer dann möglich, wenn dem Ableiten ein Schlussfolgern nach bestimmten Regeln zugrunde liegt. Das ist für das Lösen von Denksportaufgaben charakteristisch. Wir dürfen erwarten, dass sich in diesem Fall das menschliche Vorgehen simulieren lässt, freilich nur soweit, wie es auf Ableiten aus vorhandenem Wissen beruht. Versuchen wir es mit einer bekannten Aufgabe.

Denksportaufgabe 1: Verwandtschaftsproblem 2

Eine gedachte Person stellt eine andere mit folgenden Worten vor: "*Ist doch dieses Mannes Mutter meiner Mutter Schwiegermutter*." Frage: In welchem Verwandtschaftsverhältnis steht der Vorgestellte zum Vorstellenden?

Man muss schon sehr fix im Denken sein, um die Antwort (Vater oder Onkel) sofort parat zu haben. Sogar einige Minuten des Nachdenkens reichen eventuell nicht aus. Das Nachdenken artet leicht in ein Probieren aus, eventuell in ein ständig wiederholtes Neuanfangen, wobei die Systematik des Suchens nach wenigen Schlussschritten verloren geht. Offenbar reicht das Kurzzeitgedächtnis nicht aus, um alle versuchten Wege ständig klar gegenwärtig zu haben. Der Ariadnefaden fehlt.

Beim Schachspiel ist es ähnlich. Die Anzahl der Züge, die ein Spieler vorausdenken kann, ist durch sein Kurzzeitgedächtnis begrenzt[3]. Aus dem gleichen Grunde ist die Länge von Kopfrechnungen begrenzt. Ist das Kurzzeitgedächtnis überfordert,

2 Analysis ist nicht mit analytischem Rechnen zu verwechseln.

3 Ob der Begriff des Kurzzeitgedächtnisses den Sachverhalt genau trifft, bleibt dahingestellt.

greift man zu Papier und Stift und ergänzt das Gedächtnis durch einen "externen Speicher".

Wenn man nicht zu den Schnelldenkern gehört und sich an die Lösung obiger Denksportaufgabe macht, beginnt man vernünftigerweise damit, sich das Wissen über Verwandtschaftsbeziehungen zu vergegenwärtigen. Dabei handelt es sich um Sprach"*regeln*" (Bezeichnungsregeln), die z.B. festlegen, unter welchen Bedingungen welche Verwandtschaftsbeziehung bestehen oder welche Beziehungen welche anderen nach sich ziehen. Zweckmäßigerweise notiert man sich alles, was für die Lösung der Aufgabe brauchbar zu sein scheint. Das notierte Wissen könnte folgende "*Regeln*" enthalten:

1. Wenn *x* Schwiegermutter von *y* ist, dann existiert eine Person *z*, die Kind von *x* ist und die mit *y* verheiratet ist (Schwiegermutterregel).
2. Wenn *x* mit *y* verheiratet und die Mutter von *z* ist, dann ist *y* der Vater von *z* (Vaterregel).

Beim anschließenden eigentlichen Lösen der Aufgabe kann man dann - in Analogie zum Nachschlagen in einer Formelsammlung - die Liste auf anwendbare Regeln durchsuchen. Ebenso wie in einer Formelsammlung haben gleiche Variablenbezeichner, die in verschiedenen Formeln/Regeln auftreten, an sich nichts miteinander zu tun; erst der Kontext, in dem sie auftreten, legt Beziehungen zwischen den Variablen fest.

Die beiden genannten Regeln haben die Form von Wenn-dann-Sätzen; sie stellen *Implikationen* dar (man erinnere sich an die Regeln in Entscheidungstabellen [12.5]). Der Wenn-Teil enthält die **Prämisse**(n), der Dann-Teil die **Konklusion**(en). Die Schwiegermutterregel gilt stets, die Vaterregel nur "in der Regel". Wenn wir im Weiteren eine Regel anwenden, nehmen wir zunächst an, dass sie im betrachteten konkreten Fall tatsächlich gilt (z.B. dass *y* der Vater und nicht der Stiefvater von *z* ist). Die erste Regel ist für die Lösungsfindung sicher erforderlich. Ob das auch für die zweite Regel gilt und ob noch andere Regeln benötigt werden, ist nicht ohne Weiteres zu erkennen.

Um mit unserem Wissen besser hantieren zu können, überführen wir es in eine übersichtliche, standardisierte Kurznotation. Dazu wählen wir die Sprache des **Prädikatenkalküls**[4]. Wir vereinbaren vier zweistellige Prädikate und interpretieren sie, d.h. wir ordnen ihnen ihre externe Semantik zu:

`ehe(x,y)` - *x* ist mit *y* verheiratet,
`va(x,y)` - *x* ist Vater von *y*,
`mu(x,y)` - *x* ist Mutter von *y*,
`smu(x,y)` - *x* ist Schwiegermutter von *y*.

4 Anstelle von Prädikatenkalkül wird auch Prädikatenlogik gesagt. Wenn im Weiteren von Prädikatenkalkül die Rede sein wird, ist darunter stets der sogenannte Prä*dikatenkalkül erster Ordnung* zu verstehen. In der englischsprachigen Literatur wird er häufig als *First-Order Logic* bezeichnet.

Die Zeichenketten `ehe, va, mu, smu` sind Namen (Bezeichner) von Prädikaten. Die Variablen eines Prädikats hatten wir Individuenvariablen genannt [8.19]. Bild 16.1 stellt die Verwandtschaftsverhältnisse in Form eines Graphen dar.

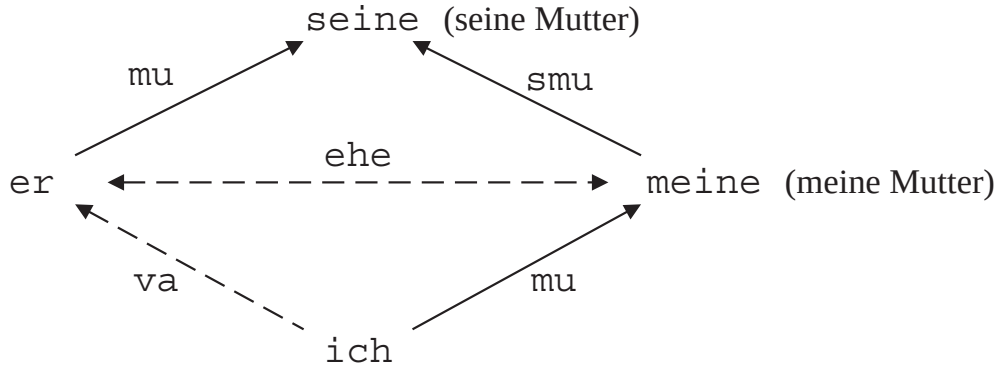


Bild 16.1 Verwandtschaftsgraph zu Denksportaufgabe 1 (Verwandtschaftsproblem). Bezeichnungen siehe Text. Die Pfeile zeigen jeweils zum ersten Prädikatargument, z.B. der Pfeil des Prädikats `va` auf den Vater. Die Beziehungen der durchgezogenen Pfeile sind durch die Aufgabenstellung gegeben.

Mit den vereinbarten Bezeichnungen ist folgende Notation der ersten der oben genannten Regeln naheliegend:

`smu(x,y)` ⇒ `mu(x,z) AND ehe(y,z).`

Intuitiv ist klar, dass die Konklusion für alle *x* und *y* gilt, für welche die Prämisse `smu(x,y)` gilt. Sie gilt aber nicht für jedes *z*, doch existiert ein solches *z* mit Sicherheit. Um das zum Ausdruck zu bringen, verwendet der Prädikatenkalkül den sogenannten **Existenzquantor**. Man schreibt ∃*x*:..., liest dies als "Es existiert ein *x*, für das gilt:..." und sagt, dass *x* durch den Existenzquantor ∃ **gebunden** ist. Wenn dagegen ein Prädikat für alle Werte einer Individuenvariablen *x* wahr ist, schreibt man ∀*x*:..., liest dies als "Für alle *x* gilt" und sagt, dass *x* durch den **Allquantor** ∀ gebunden ist. Damit können wir unser **Regelwissen** (die beiden Regeln, die wir uns gemerkt haben) folgendermaßen notieren:

Regel A: ∀`x` ∀`y: smu(x,y)` ⇒ ∃`z: (mu(x,z) AND ehe(y,z)),`

Regel B: ∀`x` ∀`y` ∀`z: mu(x,z) AND ehe(x,y)` ⇒ `va(y,z).`

Um auch unser **Faktenwissen** über den Diskursbereich in Kurzform notieren zu können, vereinbaren wir für die beteiligten Personen folgende Namen (Bezeichner von Individuen*konstanten*):

`ich` - der Vorstellende,
`er` - der Vorgestellte,
`meine` - Mutter des Vorstellenden,
`seine` - Mutter des Vorgestellten.

Damit lautet das Faktenwissen:

Fakt 1: `smu(seine,meine),`
Fakt 2: `mu(meine,ich) ,`
Fakt 3: `mu(seine,er) .`

Die Frage könnte folgendermaßen notiert werden:

Frage: `?(er,ich).`

Das Fragezeichen bezeichnet das gesuchte Prädikat (die gesuchte Beziehung). Fakt 1 ist die Kurznotation desjenigen Satzes, mit dem in der Aufgabenstellung "er" vorgestellt wird. Die Fakten 2 und 3 ergeben sich als selbstverständlicher Kontext aus dem Satz des Vorstellenden, sodass sie kaum der Erwähnung wert zu sein scheinen. Dennoch dürfen sie nicht fehlen, wenn der Computer in der Lage sein soll, die Lösung aus dem ihm mitgeteilten Wissen abzuleiten. Er kann nämlich auf keinerlei externe Semantik zurückgreifen. Er kann nur "blindlings", ohne jedes Verständnis, mechanisch, automatisch - oder wie immer man das "Denken" des Computers bezeichnen will - vorgehen. Unter externer Semantik (siehe Bild 5.3) ist das gesamte Kontextwissen zu verstehen, das sich auf die konkrete Situation bezieht bzw. mit ihr in Verbindung gebracht werden kann und das jedem Menschen, der sich in der betreffenden Situation befindet, "automatisch" zur Verfügung steht, d.h. das er mit der Situation *assoziiert*. Das Problem der Anbindung der *externen* Semantik (Nutzersemantik, Humansemantik) über die *formale* Semantik (Kalkülsemantik) an die *interne* Semantik des Computers (Maschinensemantik, die Prozesse in der Hardware) hatten wir das technische Semantikproblem genannt.

Für das *formale* Schlussfolgern, d.h. für das Schlussfolgern nach den Regeln des Prädikatenkalküls unter Abstraktion von jeglicher externen Semantik, wird das Wort *Inferenzieren* verwendet. Wir vereinbaren: *Das formale Schlussfolgern nach den Regeln des Prädikatenkalküls heißt* ***Inferenzieren****. Eine durch Inferenzieren hergeleitete Schlussfolgerung heißt* ***Inferenz****. Die dem Inferenzieren zugrunde liegenden Regeln bzw. Fakten bilden das sog.* ***Regel-*** bzw. ***Faktenwissen***.

Das gesteckte Ziel besteht also in der Lösung der Denksportaufgabe durch Inferenzieren. Wenn man dabei die obigen Regeln A und B verwendet, stellt man fest, dass der Ableitungsprozess etwas umständlich wird, dass er sich aber durch Umformulierung dieser Regeln zu

Regel 1: `smu(x,y) AND mu(x,z)` $\Rightarrow$ `ehe(y,z),`
Regel 2: `mu(u,v) AND ehe(u,w)` $\Rightarrow$ `va(w,v)`

vereinfachen lässt.

Die Richtigkeit der beiden Regeln ist unschwer zu verifizieren. Regel 2 unterscheidet sich von Regel B durch andere Variablenbezeichner und durch das Fehlen der Allquantoren. Die Wahl neuer Bezeichner ist keine Notwendigkeit; sie soll die Gefahr ausschließen, dass der Leser die Regeln falsch interpretiert, indem er annimmt, dass gleiche Bezeichner in den beiden Regeln dieselben Variablen bezeichnen. Es wäre genauso richtig gewesen, auch in Regel 2 die Bezeichnungen *x*, *y* und *z* zu verwenden. Wenn in Regel 1 dieselben Bezeichner auftreten, führt das zu keiner

Fehlinterpretation durch den Interpreter (durch das Übersetzerprogramm), denn, wie bereits erwähnt, haben Variablenbezeichner in verschiedenen Regeln an sich (ohne Berücksichtigung des Kontextes, in dem sie auftreten) nichts miteinander zu tun; beispielsweise hat das *x* in Regel A nichts mit dem *x* in Regel B zu tun. Dieser Sachverhalt sei wegen seiner Wichtigkeit noch einmal herausgestellt. *Der **Gültigkeitsbereich** eines Variablenbezeichners, d.h. die Bindung eines Bezeichners an eine Variable und letzten Endes an einen Speicherplatz beschränkt sich auf die jeweilige Formel*. Das Fehlen von Allquantoren in den Regeln 1 und 2 beruht auf der stillschweigenden Vereinbarung, dass Allquantoren nicht geschrieben zu werden brauchen, wenn sie für alle Variablen gelten.

Die Regeln 1 und 2 sind sogenannte Hornklauseln. *Eine Implikation, deren Prämisse eine Konjunktion aus beliebig vielen elementaren Prädikaten ("Bausteinprämissen") und deren Konklusion ein elementares Prädikat ist, wird als **Hornklausel** bezeichnet*. Die Überführung prädikatenlogischer Ausdrücke in Hornklauseln ist ein wichtiger "Normalisierungsschritt" des maschinellen Inferezierens. 3

Nach diesen längeren Vorbereitungen können wir mit dem Inferenzieren beginnen. Es besteht in einem "systematischen Versuchen", die Regeln durch die Konstanten (durch die beteiligten Personen) unter Berücksichtigung der Fakten zu befriedigen. Wir beginnen mit dem ersten Prädikat in der Prämisse von Regel 1 und suchen im Faktenwissen einen "passenden" Fakt, d.h. einen solchen, dessen syntaktische Struktur mit der des Prädikats übereinstimmt. Das trifft für Fakt 1 zu, der sich von `smu(x,y)` nur darin unterscheidet, dass er Konstante anstelle der Variablen enthält. Demzufolge substituieren wir versuchsweise *x* durch `seine` und *y* durch `meine`. Für die Beschreibung einer Substitution werden verschiedene Notationsweisen verwendet. Eine gängige Notation ist z.B. [`x/seine`] und entsprechend [`y/meine`]. Da im vorliegenden Fall Variablen durch Konstante substituiert werden, handelt es sich um "Wertzuweisungen" in einem verallgemeinerten Sinne, sodass das Ergibtzeichen (Wertzuweisungszeichen) := gerechtfertigt ist (siehe Bild 16.2).

Man beachte die Ähnlichkeit zum Vorgehen nach dem Markovalgorithmus oder zum Vorgehen beim analytischen Rechnen. Ganz im Sinne des Markovalgorithmus nehmen wir nun das zweite Prädikat in der Prämisse von Regel 1 und suchen nach einem *passenden* Fakt. Zuerst stoßen wir auf Fakt 2, stellen aber fest, dass er mit der ersten Substitution nicht "zusammenpasst", denn der Variablen *y* ist bereits die Person (die Konstante) `meine` zugewiesen. Die weitere Suche führt auf Fakt 3, und man stellt fest, dass er mit der vorherigen Zuweisung `x:=seine` zusammenpasst.[5] Wir substituieren also `z:=er`.

---

5 Im Fachjargon hat sich eine sprachästhetisch unschöne "neudeutsche" Sprechweise eingebürgert: Für "passen" wird das eingedeutschte Wort "*matchen*" (vom englischen Verb to match = passen, zusammenpassen) verwendet. Statt "Fakt 3 passt" würde "Fakt 3 matcht" gesagt werden.

Mit diesen Substitutionen ergibt sich der
Schluss 1: `ehe(meine,er)`

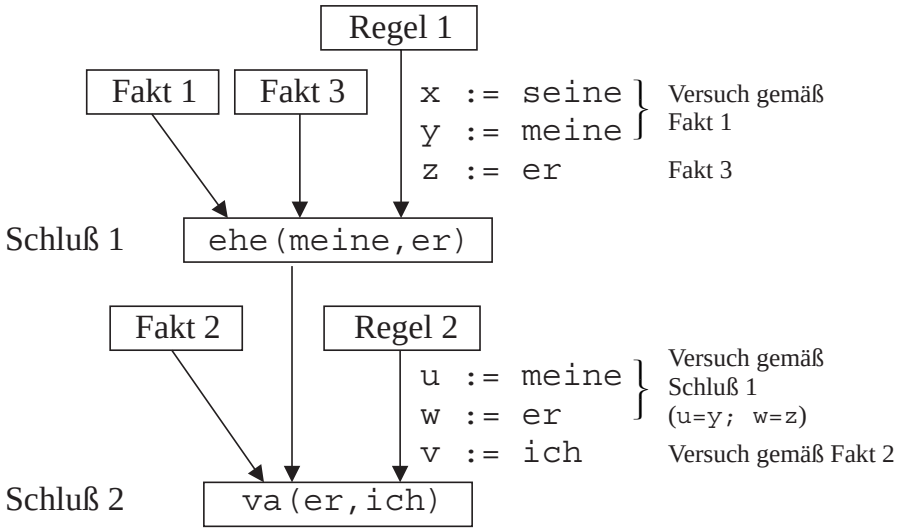


Bild 16.2 Inferenzbaum zum Verwandtschaftsproblem

Der Weg zu Schluss 1 ist in Bild 16.2 graphisch dargestellt. Die Pfeile zeigen von der Regel 1 und den verwendeten Fakten zur Konklusion, zum Schluss 1. Rechts neben dem Pfeil, der von Regel 1 ausgeht, sind die dabei durchgeführten Substitutionen angegeben, sowie die Begründungen der Substitutionen. Nach dem gleichen Muster ist der Weg von Schluss 1 zu Schluss 2 dargestellt. In Klammern ist die Entsprechung zwischen Variablenbezeichnern hinzugefügt. Beispielsweise bedeutet *w=z*, dass den Variablen *w* und *z* derselbe Wert (dieselbe Person), nämlich `er` zugewiesen wird.

Die Richtigkeit des zweiten Schlussfolgerungsschrittes ist so offensichtlich, dass der gedanklich schlussfolgernde Mensch ihn "automatisch" (unbewusst, reflektorisch) vollzieht. Wenn man weiß, dass `er` der Mann meiner Mutter ist, weiß man auch (ohne nachdenken zu müssen), dass `er` mein Vater ist.

Wir haben eine Lösung der Denksportaufgabe durch formales Schlussfolgern, durch Inferenzieren gefunden, also mit Hilfe einer Methode, die sich implementieren lässt, sodass auch der Computer befähigt werden kann, die Lösung zu finden. Die Aufgabe ist aber noch nicht vollständig gelöst. Wie man leicht verifiziert, kann der Vorgestellte auch der Onkel des Vorstellenden sein. Die Lösung `va(er,ich)` ist nur dann mit Sicherheit richtig, wenn `er` keinen Bruder hat, d.h. unter der Voraussetzung, dass das benutzte Faktenwissen alle relevanten Personen umfasst und insofern *vollständig* ist. Falls `er` einen Bruder hat, muss dieser als fünfte Person

(abgekürzt `p5`) im Faktenwissen vertreten sein. Ausserdem wäre eine zusätzliche Regel notwendig. Die Wissensbasis müsste durch

Fakt 4: `bru(p5,er)` und

Regel 3: `mu(x,y) AND mu(x,z)` $\Rightarrow$ `bru(y,z).`

ergänzt werden.

Die Notwendigkeit, den Diskursbereich ausreichend vollständig zu erfassen, ist ein charakteristisches Problem des formalen Schlussfolgerns. Neben der Vollständigkeit muss die *Widerspruchsfreiheit* der Wissensbasis gewährleistet sein. Weder die Fakten noch die Regeln dürfen sich widersprechen. Die Wissensbasis muss in sich **konsistent** sein. Konsistenz muss auch für alle ableitbaren Aussagen (Fakten) gefordert werden. Ihre Gewährleistung kann problematisch sein, insbesondere dann, wenn im Diskursbereich Veränderungen eintreten. Neue Fakten müssen nicht unbedingt mit den alten konsistent sein oder sie können alte Inferenzen in Frage stellen. Wenn beispielsweise `er` ein zweites mal heiratet, kann aus Regel 2 nicht mehr mit Sicherheit der Schluss 2 gezogen werden.

Damit ist die Aufzählung der Probleme, die beim Inferenzieren auftreten können, nicht beendet. Im folgenden Kapitel werden einige weitere aufgezeigt. Manche der Probleme vereinfachen sich oder entfallen vollständig, wenn eine andere Strategie des Inferenzierens angewendet wird, die jeder aus dem Alltagsleben kennt, das Probieren, genauer das Verifizieren von Annahmen.

Beim Nacherfinden des maschinellen Inferenzierens hatten wir uns vom Vorgehen eines systematisch veranlagten Menschen inspirieren lassen, der zunächst relevantes Wissen, Regeln und Fakten sammelt und dann versucht, mit Hilfe des gesammelten Wissens die Frage zu beantworten. Ein Mensch, der mehr zum Probieren neigt, wird wahrscheinlich anders vorgehen. Der erste Schluss, den wohl die meisten Menschen zunächst einmal ziehen werden, die Systematiker wie die Probierer, ist der, dass die Schwiegermutter eine Generation älter ist als `er` und dass sie zwei Generationen älter ist als `ich`, dass folglich `er` eine Generation älter ist als `ich`. Das trifft z.B. für den Vater von `ich` zu. Der Probierer wird nun von der *Hypothese* ausgehen, dass `va(er,ich)` die Lösung ist und prüfen, ob in diesem Fall das Prädikat `smu(seine,meine)` erfüllt ist. Die Prüfung fällt positiv aus, womit der Probierer sich eventuell zufrieden gibt.

Das Inferenzieren aufgrund einer Hypothese und deren Prüfung wird Inferenzieren durch **Rückwärtsverkettung** oder kurz **Rückwärtsinferenz** genannt. Es wird von der Lösung *rückwärts*, d.h. in Richtung vorgegebener Fakten geschlossen. Das Vorgehen ohne Hypothese, bei dem, ausgehend von den gegebenen Fakten *vorwärts* (in Richtung Lösung) geschlossen wird, heißt Inferenzieren durch **Vorwärtsverkettung** oder kurz **Vorwärtsinferenz**.

Im Falle vieler Fakten und weniger Hypothesen ist Rückwärtsinferenz offensichtlich einfacher zu realisieren und zu implementieren als Vorwärtsinferenz, denn der Suchraum wird durch Hypothesen erheblich eingeschränkt. Darum ist es verständ-

lich, dass viele Inferenziersysteme mit Rückwärtsverkettung arbeiten. Das gilt beispielsweise für Prolog-Systeme (siehe Bild 16.3).

Wir kehren noch einmal zu unserem Probierer zurück. Wenn er gründlich ist, wird er weitersuchen und andere Hypothesen prüfen. Die naheliegendste Hypothese ist `onk(er,ich)`. Die Prüfung zeigt, dass auch sie mit Fakt 1 vereinbar ist. Damit lautet die Lösung `va(er,ich) ODER onk(er,ich)`. Ein Prologsystem würde die Hypothesen (Anfrage) `?va(er,ich)` und `?onk(er,ich)` bestätigen. Die Hypothese `?bru(er,ich)` würde es ablehnen.

Damit ist die Frage, ob es noch andere Personen geben könnte, für die Fakt 1 zutrifft, immer noch nicht aus der Welt geschafft. Zum Nachweis der **Vollständigkeit der Antwort** kann wiederum die Rückwärtsverkettung angewendet werden. Dabei ist zu beweisen, dass die Annahme irgendeiner anderen verwandtschaftlichen Beziehung, also irgendeines anderen Prädikats `P(er,ich)` zu einem Widerspruch mit Fakt 1 führt. Dazu reicht das Suchen und die Wertzuweisung als "*Bausteinoperationen* des Inferenzierens" nicht aus. Um den Widerspruch herzuleiten, bedarf es zum einen eines umfangreicheren Regelwissens, und zum anderen müssen eventuell Prädikate transformiert werden, m.a.W. es müssen *Formelmanipulationen*, konkret *Prädikattranformationen* gemäß den verfügbaren Regeln durchgeführt werden, bis man zu einem Widerspruch gelangt, also zu einem Prädikat, das stets falsch ist.

Beim Inferenzieren wechseln sich *Suchen* von Wissen mit *Anwenden* von Wissen ab. Gesucht wird nach *anwendbarem* Wissen (Regeln, Fakten). Wissen anwenden ist stets ein *Substituieren*. Im einfachsten Fall werden Variable durch Konstante substituiert (Wertzuweisung). Es können auch Variable durch Variable oder Prädikate durch Prädikate substituiert, d.h. Prädikate oder Prädikatverbindungen (Klauseln) *transformiert* werden. Die Transformation (Formelmanipulation) stellt also - ebenso wie die Wertzuweisung - eine spezielle Art der Substitution dar (vgl. die Überlegungen am Ende des Kapitels 15.8.). In jedem Fall setzt Substitution die *syntaktische Übereinsimmung* der involvierten Prädikate voraus. Der Syntaxvergleich ist Bestandteil des Suchprozesses.

Das Substituieren beim Inferenzieren erfolgt ganz analog zum Substituieren beim analytischen Rechnen, wie es in Kap.15.8 dargelegt wurde. Dort bestand die Anwendung einer Formel (z.B. aus einer Formelsammlung) auf einen gegebenen Ausdruck aus zwei Schritten, dem *Bezeichnerabgleich* und der *Substitution*, in völliger Analogie zum Inferenzieren. Nur wird der Bezeichnerabgleich in Falle des Inferenzierens **Unifikation** genannt. Damit setzt sich ein Inferenzprozess aus drei *Bausteinoperationen* zusammen:

- Suche (einschließlich Syntaxvergleich),
- Unifikation,
- Substitution.

In der Sprechweise der USB-Methode können wir also abschließend feststellen: ***Inferenzieren*** *ist eine Kompositoperation aus Such-, Unifikations- und Substitutionsoperationen.*

16.2 Wissensverarbeitung

Im vorangehenden Kapitel haben wir das formale Schlussfolgern anhand einer Denksportaufgabe nacherfunden. Das ist ein *theoretischer* Erfolg. Die *praktische* Bedeutung der "Erfindung" hängt davon ab, wieweit sich das Vorgehen verallgemeinern und in den verschiedensten Situation anwenden lässt, in denen mit Wissen "hantiert", Wissen "verarbeitet" wird. Aus dieser Sicht drängt sich eine etwas andere, allgemeinere Frage auf: *Lässt sich Wissensverarbeitung automatisieren*? Zur Wissensverarbeitung gehören

- **Wissenserwerb**, d.h. das Sammeln und Abspeichern von Wissen, auch *Wissensakquisition* genannt,
- **Wissenszugriff**, d.h. das *Wiederauffinden* von abgespeichertem Wissen,
- **Inferenzieren**, d.h. das *Produzieren* von abgeleitetem aus dem gespeicherten Wissen.

Ein **Wissensverarbeitungssystem** muss über Programme für alle drei Tätigkeiten verfügen, vom Sammeln abgesehen, obwohl auch das in gewissen Grenzen automatisiert werden kann. Wenn die Fähigkeit zum Inferenzieren fehlt, spricht man i.Allg. von **Datenbanksystem**. Ebenso wie ein Mensch, kann auch ein Wissensverarbeitungssystem nicht *alles* wissen. Wenn seine Wissensbasis auf ein bestimmtes Gebiet spezialisiert ist, wird es **Expertensystem** für dieses Gebiet genannt. Ein Expertensystem kann nur dann die Rolle eines echten Experten übernehmen, wenn es über

- ein ausreichend umfangreiches und schnell und zuverlässig abrufbares Wissen über den Diskursbereich,
- ein ausreichend vollständiges und effizientes Inferenzierprogramm und
- eine geeignete Dialogsprache

verfügt. Wir wollen uns überlegen, ob bzw. wie diese Bedingungen erfüllt werden können.

Vollständigkeit des Inferenzierers

Wir beginnen mit der zweiten Bedingung und fragen, ob bzw. wieweit sich das in Kap.16.1 beschriebene Schlussverfahren verallgemeinern lässt, ob sich vielleicht sogar ein universeller Inferenzoperator, ein "*universeller Inferenzierer*" (in Analogie zum universellen Rechner) realisieren lässt, mit anderen Worten, ob ein Programm geschrieben werden kann, das in der Lage ist, aus jedem beliebigen vorgegebenen Regel- und Faktenwissen "alle nur möglichen" Schlüsse zu ziehen.

Man könnte den Eindruck haben, als würden wir in eine ähnlich "bodenlose" Frage verstrickt, wie es die Frage nach der Berechenbarkeit von Funktionen war.

Denn Inferenzieren ist de facto nichts anderes als das Berechnen von Funktionswerten, wobei die Funktionen durch *Prädikate* [8.17] festgelegt sind und die Funktionswerte *Merkmalswerte von Individuenvariablen* darstellen.

Dennoch ist die Frage nach dem "universellen Inferenzierer" nicht "bodenlos", denn wir befinden uns bereits auf "formalem Boden", auf dem Boden des Prädikatenkalküls, und die Frage kann formal beantwortet werden, wenn man davon ausgeht, dass Inferenzieren eine Kompositoperation aus den Bausteinoperationen *Suchen, Unifizieren* und *Substituieren* ist. Man kann auch auf den Algorithmenbegriff (genauer auf den Begriff des *imperativen* Algorithmus) zurückgreifen und definieren:

*Ein **Inferenzieralgorithmus** ist eine Vorschrift zur Ableitung von Konklusionen aus Prämissen in endlich vielen Such-, Unifizierungs- und Substitutionsschritten.* Durch Artikulierung des Algorithmus in einer Programmiersprache und Implementierung wird der Computer zu einem (realen) Inferenzierer.

Damit ist der Inferezierer formal definiert, und seine "Universalität" kann formal (mit den Mitteln des Prädikatenkalküls) untersucht werden, wobei an die Stelle des unscharfen Begriffs der Universalität der scharfe Begriff der **Vollständigkeit** tritt. *Ein Inferenzierer ist **vollständig** hinsichtlich einer gegebenen Wissensbasis, wenn er aus den Fakten der Wissensbasis (den Prämissen) **alle** wahren Konklusionen (neue Fakten) ableiten kann.*

Der Nachweis, ob ein Inferenzierer vollständig ist oder nicht, stellt ein theoretisch anspruchsvolles Problem dar, auf das wir nicht näher eingehen werden. Wir begnügen uns mit dem Verweis auf die angegebene Literatur (insbesondere auf [Russell 95]) und mit der Erwähnung einiger charakteristischer Schwierigkeiten theoretischer Natur.

Auf zwei Probleme wurde bereits aufmerksam gemacht, das Problem der Vollständigkeit eines gegebenen Faktenwissens hinsichtlich konkreter Anfragen und das Problem der Widerspruchsfreiheit oder Konsistenz der Wissensbasis. Darüber hinaus können alle diejenigen Probleme auftreten, mit denen wir beim analytischen Rechnen konfrontiert wurden. Denn Inferenzieren ist - ebenso wie das analytische Rechnen - ein Rechnen mit Variablen, jedoch nicht nach den Regeln der Algebra oder Analysis, sondern nach den Regeln des Prädikatenkalküls. Hinsichtlich der grundsätzlichen Vorgehensweise sowie der auftretenden Schwierigkeiten wäre hier alles zu wiederholen, was in Kap.15.8 gesagt worden ist, nur treten an die Stelle der Formeln der Algebra und Analysis die Formeln des Prädikatenkalküls. Zu den Einzelheiten dieser Art des Rechnens muss auf die Literatur verwiesen werden[6]. Wir überlassen es dem Leser, sich noch einmal die Gedankengänge und Aussagen von Kap.15.8 zu vergegenwärtigen und auf das Inferenzieren zu übertragen.

Es besteht stets die Möglichkeit, dass ein Inferenzprozess nicht terminiert. Dann gibt es keinen "Schluss" in zweifachem Sinne, es kann kein "Schluss" *gezogen* (keine

6 Z.B. [Russell 95],[Schöning 89].

Antwort gegeben) werden, weil das Programm zu keinem “Schluss” (Ende) *kommt*. Das kann verschiedene Ursachen haben. Beispielsweise kann der Inferenzierer in einen unendlichen Suchzyklus geraten oder er kann auf ein Prädikat stoßen, das nicht entscheidbar ist, z.B. weil es eine *widersprüchliche Zirkularität* enthält oder weil es ein *Wahrsageprädikat* ist.

Das Versagen eines Inferenzierers muss also durchaus nicht die Folge von Mängeln in der Hard- oder Software sein oder eine Folge prinzipieller Grenzen der Intelligenz des Computers, sondern es kann ein Mangel des sprachlichen Modellierens an sich vorliegen. Zwar dient Sprache der Modellierung der Realität, also dessen, was *der Fall*, was *wahr* ist. Doch lässt Sprache beliebig freien Raum für das Artikulieren unwahrer, widersprüchlicher, sinnloser oder nichtentscheidbarer Aussagen. Mit den Ursachen und Folgen hatten wir uns in Kap.6 auseinandergesetzt. Beim sprachlichen Modellieren durch den Rechner kann die Folge darin bestehen, dass ein Programm nicht terminiert.

Effizienz des Inferenzierens

Es gibt heute leistungsfähige Inferenzprogramme. Praktisch für jeden Beruf, in welchem regelbasiertes Schlussfolgern eine Rolle spielt, kann ein Expertensystem erstellt werden, das einen in dem Beruf Tätigen unterstützen kann, vorausgesetzt, es verfügt über die erforderliche Wissensbasis (sprich: Fachwissen). Auf dem Wege zu diesem Erfolg waren viele Schwierigkeiten zu überwinden, von denen einige bereits genannt wurden. Die Aufgabe, mit der die Informatiker konfrontiert waren, als sie versuchten, das menschliche Schlussfolgern zu simulieren, ist hinsichtlich der Höhe des Anspruchs vergleichbar mit der Aufgabe, numerisches Rechnen zu simulieren, d.h. *Rechen*maschinen im ursprünglichen Sinne des Wortes zu bauen.

Das numerische Rechnen hat uns den gesamten zweiten Teil bis einschließlich Kapitel 15.6 beschäftigt. Der Weg zu einer technisch brauchbaren Lösung führte über die Normalisierung von Ausdrücken und die Standardisierung der Bausteinoperationen. Ausgangspunkt waren die elementaren booleschen Operatoren, aus denen Kompositoperatoren komponiert wurden. Eine zentrale Rolle spielte die KNDF, die kanonische disjunktive *Normal*form und ihre mikroelektronische Realisierung, die zum *Standard*baustein der verschiedensten Operatoren höherer Komponierungsstufe wurde, insbesondere verschiedener Matrizenschaltkreise.

In der technischen Handhabung des Prädikatenkalküls und in der “Technologie” des Inferenzierens spielt die Hornklausel eine ähnliche “normalisierende” Rolle wie die KNDF in der Technologie der booleschen Algebra und des numerischen Rechnens. Auch die Hornklausel ist aus booleschen Operatoren komponiert, doch deren Operanden sind keine Aussagen über Konstanten, keine Fakten, sondern Aussagen über Variablen.

Durch die Normalisierung der Ausdrücke zu *Implikationen* in der Form von Hornklauseln wird die *Standardisierung* des Inferenzierens zum sogenannten **Resolutionsverfahren** möglich. Es besteht aus einzelnen Schritten. Bild 16.2 stellt einen

konkreten Inferenzprozess nach dem Resolutionsverfahren graphisch dar. Die Ableitung eines Knotenprädikats heißt **Resolution**, das Ergebnis des letzten Inferenzschrittes (der letzten Resolution) heißt **Resolvente**. Es lässt sich Folgendes zeigen: *Das Resolutionsverfahren ist hinsichtlich falscher Prädikate stets vollständig*. Damit kann die Wahrheit jedes Prädikats dadurch nachgewiesen werden, dass aus seiner Negation ein Widerspruch, also die Resolvente `wahr`⇒`falsch` folgt.

Das Ziel ist erreicht. Wir haben erkannt, wie der Computer schlussfolgern kann. Doch aus dem Schlussfolgern ist durch Kalkülisierung des Problems Rechnen geworden. Der beschriebene Lösungsprozess des Verwandtschaftsproblems ist Deduzieren im Rahmen eines (nichtaxiomatisierten) Kalküls, der durch Sprachvereinbarungen (die sich an das Prädikatenkalkül anlehnen) und zwei Regeln definiert ist. Das bedeutet nicht, dass auch der Rätselrater, der vom Prädikatenkalkül nichts weiß, die Lösung des Verwandtschaftsproblems durch Rechnen findet. Für ihn ist das Problem nichtmathematischer Natur.

Es entspricht dem gängigen Sprachgebrauch, bei der kalkülisierten Beschreibung eines Sachverhaltes von *mathematischer* Modellierung zu sprechen, unabhängig davon, ob der Kalkül axiomatisiert ist oder nicht. Der Begriff des mathematischen Modells ist an die Existenz eines Kalküls gebunden, doch muss der Kalkül nicht unbedingt axiomatisiert sein. Die Verwendung eines nichtaxiomatisierten Kalküls birgt die Gefahr in sich, dass eine versuchte Ableitung ein falsches oder widersprüchliches Resultat oder auch gar kein Resultat liefert. Ein Blick in die Geschichte der Naturwissenschaft zeigt, dass zur Beschreibung neuer Phänomene i.Allg. zunächst nichtaxiomatische Theorien entwickelt und erfolgreich angewendet werden, deren Axiomatisierung oft erst viel später gelingt. (Man erinnere sich: Eine physikalische Theorie ist die Interpretation eines Kalküls durch Objekte der Wirklichkeit). Die Entwicklung der klassischen Mechanik oder der Quantenmechanik sind Beispiele hierfür.

Wenn das Wort “mathematisch” nicht unbedingt “axiomatisch”, unbedingt aber “kalkülisiert” beinhaltet, ist es gerechtfertigt, das Schlussfolgern insoweit dem Bereich der Mathematik zuzuordnen, wie es kalkülisiert ist, d.h. wie es Inferenzieren ist. Unter dieser Bedingung wird Schlussfolgern zu einer mathematischen Operation. Ungerechtfertigt dagegen wäre es, das alltägliche Schlussfolgern mathematisch zu nennen, es sei denn, es verwendet *bewusst* einen Kalkül. Der Schluss vom Blitz auf den bevorstehenden Donner verlangt keine Mathematik, weder jetzt, noch vor hunderttausend Jahren. In diesem Zeitmaßstab gemessen ist bewusstes Kalkülisieren und Inferenzieren sicher ein ziemlich junges Produkt der kulturellen Evolution.

Anfragesprache

Expertensysteme wären nutzlos, wenn ein Anwender, der den Computer eine Inferenz ausführen lassen will, dafür selber ein Inferenzprogramm schreiben müsste. Diese Arbeit nimmt ihm das System ab, richtiger der Systemprogrammierer. Der Wert von Expertensystemen und der Wert der KI-Technologie ganz allgemein

besteht gerade darin, dass der Nutzer lediglich das für sein Problem spezifische Wissen und seine Fragen oder Hypothesen in einer geeigneten Programmiersprache zu artikulieren braucht. Diese Sprache nennen wir **Anfragesprache**.

Anfragesprachen sind i.d.R. nicht dafür geeignet, eine Berechnungsvorschrift zu artikulieren. Vielmehr dienen sie der *Beschreibung der Bedingungen*, unter denen Schlussfolgerungen gezogen (Vorschläge gemacht, Expertisen erstellt) werden sollen. In diesem Sinne spricht man von **deskriptiven** oder **deklarativen** Sprachen, im Gegensatz zu den **prozeduralen** Sprachen, die der Artikulierung von Operationsvorschriften und damit unmittelbar der Steuerung von Prozessen dienen. Die Bezeichnungen *prozedural* und *deklarativ* werden auch auf Programme angewendet. Ein deklaratives Programm artikuliert Prädikate (Eigenschaften und Relationen) und *deklariert* sie als *wahr*. Danach bietet sich als Anfragesprache die Sprache des Prädikatenkalküls an, denn sie ist eine sehr allgemeine formale Sprache, die mit Prädikaten hantiert. Bei der Lösung der Denksportaufgabe in Kap.16.1 haben wir sie ohne weitere Begründung verwendet.

Auch die Sprache Prolog ist eine deklarative Sprache, die sich an die Sprache der Prädikatenlogik anlehnt. Prolog ist als “PROgrammieren in LOgik” zu lesen, oder genauer “Programmieren in Prädikatenlogik”. Sie unterstützt das sog. *logische Programmieren*. In den Kapiteln 18 und 20 werden nähere Ausführungen über logische Sprachen gemacht. Prolog ist als Anfragesprache für Inferenziersysteme geeignet und als solche konzipiert. Bild 16.3 zeigt ein Prolog-Programm zum Lösen des Verwandtschaftsproblems.

```
((ehe y z)(smu x y)(mu x z))
((va w v)(mu mu u v)(ehe u w))
((smu seine meine))
((mu meine ich))
((mu seine er))
? (va er ich)
```

Bild 16.3 Prolog-Programm zum Lösen des Verwandtschaftsproblems.

In den letzten drei Programmzeilen, vor der eigentlichen Anfrage, die mit einem Fragezeichen beginnt, wird der Leser das Faktenwissen erkennen (oben als Fakt 1, Fakt 2 und Fakt 3 bezeichnet), wobei anstelle der Präfixnotation die Listennotation verwendet ist. Die ersten beiden Zeilen stellen das Regelwissen dar. Sie entsprechen den Regeln 1 und 2, die wir bei der Lösung des Verwandschaftsproblems verwendet haben; es sind *Hornklauseln*, allerdings in einer speziellen Syntax notiert. Der jeweils erste Klammerausdruck, `(ehe y z)` bzw. `(va w v)`, ist die Konklusion, die nachfolgenden Klammerausdrücke sind die Prämissen (Bausteinprämissen). Der Implikationspfeil ist unterdrückt. Wäre er notiert, müsste er nach links gerichtet sein.

Das Programm demonstriert die Nutzerfreundlichkeit der Sprache und die Leistungsfähigkeit ihres Interpreters. Die letzte Zeile ist eine Hypothese, die der Programmierer vorschlägt. Sie kann bestätigt oder abgelehnt werden. Das Inferenzieren erfolgt also nach der Methode der Rückwärtsverkettung.

Die Frage nach der Sprache, in welcher der Computer seine Ausgaben "*artikuliert*", stand und steht nicht zur Debatte, weil sie selten definiert ist, jedenfalls nicht intensional durch Vorgabe von Syntaxregeln, sondern höchstens extensional durch Aufzählung konkreter Ausgabezeichenketten, die der Systementwickler vorprogrammiert und der Computer gegebenenfalls ergänzt. Man erinnere sich an die Ausgabe des Simulationssystems an den Experimentator hinsichtlich der Fallgeschwindigkeit der Kugel in Kap.15.5 [15.7]. Normalerweise gibt es also gar keine syntaktisch definierte "Dialog"-Sprache. Die einzige Forderung an die Ausgabe ist die leichte Interpretierbarkeit durch den Nutzer, das leichte Anbinden der *Nutzersemantik* an die Ausgabezeichen.

Die Bedeutung des Prädikatenkalküls ist nicht auf die Anfragesprache beschränkt, sondern jede Art von Wissensverarbeitung verwendet mehr oder weniger explizit den Prädikatenkalkül. Das ist verständlich, denn die Aufgabe des Computers besteht darin, *richtige Aussagen über die Welt zu produzieren*, also erfüllte Prädikate. Der Computer "weiß" davon nichts, aber Systementwickler und Nutzer wissen es und der Nutzer eines Computers interpretiert dessen Ausgaben als erfüllte Prädikate, als wahre Aussagen über die Welt. Eben das entspricht dem Anliegen des *technischen aktiven sprachlichen Modellierens*.

Diesem Ziel dienten alle unsere Überlegungen. In Kap.15 haben wir im Einzelnen verfolgt, wie der Computer das sprachliche Modellieren im Bereich der exakten Naturwissenschaften, insbesondere der Physik, also das *mathematische* Modellieren unterstützen kann. Dort werden die Aussagen über die Welt formalisiert, indem sie in der Sprache eines mathematischen Kalküls formuliert werden, sodass sie als Interpretationen des Kalküls aufgefasst werden können. Durch das Implementieren physikalischer oder anderer formaler Theorien wird es möglich, neue richtige Aussagen nach den Regeln eines Kalküls abzuleiten, und zwar durch numerisches oder analytisches Rechnen. Die *Wissensverarbeitung*, d.h. das "nichtmathematische" Modellieren durch den Computer, verfährt nach dem gleichen Rezept, nur verwendet sie den Prädikatenkalkül (die Prädikaten*logik*), sie "rechnet logisch". Im Sinne unserer Definition des analytischen Rechnens [15.8] [15.14] führt die Wissensverarbeitung *analytische Rechnungen* durch.

An dieser Stelle ist ein kurzer Rückblick angebracht. Wenn man sich noch einmal die Bilder 5.3 und 8.6 und die Überlegungen zum technischen Semantikproblem [15.9] und zur semantischen Bindung von Operanden vergegenwärtigt, erkennt man, dass die Einführung der obigen Bezeichnungen für Verwandte und für Verwandtschaftsrelationen dem ersten Schritt der semantischen Bindung, nämlich der Anbindung der *externen* Semantik (der "Verwandtschaftssemantik") an die *formale* Semantik des Kalküls dient. Für die Richtigkeit dieses *ersten* Schrittes, also dafür, dass

die externe Semantik logisch richtig und in ausreichendem Umfang an die Kalkülsemantik angebunden wird, ist derjenige verantwortlich, der das Regel- und Faktenwissen sowie die Anfrage artikuliert, wofür zweckmäßigerweise eine implementierte Programmiersprache verwendet wird, die eine effiziente Artikulation verwandtschaftlicher Beziehungen gestattet, beispielsweise Prolog. Für den *zweiten* Schritt, für die Anbindung der Kalkülsemantik an die Maschinensemantik sind der Entwickler der verwendeten Sprache und ihr Implementierer, der das Übersetzerprogramm schreibt, verantwortlich.

In beiden Schritten können Fehler unterlaufen. Sie sind i.d.R. schwer zu erkennen. Oft machen sie sich erst in offensichtlich falschen Schlussfolgerungen bemerkbar. Die Richtigkeit der Schlussfolgerungen, die der Computer bei der Lösung des Verwandtschaftsproblems durch *abstraktes* Schlussfolgern (Inferenzieren) zieht (siehe Bild 16.2), werden viele Leser durch "*konkretes*" Schlussfolgern anhand der *externen* Semantik verifiziert haben. Um dem Leser das zu ermöglichen, wurde ein Beispiel "aus dem Leben" mit durchschaubarer externer Semantik gewählt. Die Schlussfolgerungen können aber, wie gezeigt, auch rein formal, also ausschließlich aufgrund der formalen Semantik, d.h. nach den Regeln des Prädikatenkalküls gezogen werden. Das erfordert jedoch ein anders geartetes, abstrakteres Denken, das dem Nichtmathematiker in der Regel ungewohnt ist.

Wissenszugriff

Das Zugreifen auf gespeichertes Wissen (das Wiederauffinden von Wissen) hätte aus historischen Gründen *vor* dem Inferenzieren behandelt werden müssen, aber auch aus systematischen Gründen, denn es ist Bestandteil des Inferenzierens; nach einem "passenden" Fakt oder nach einer passenden Regel muss eventuell in einer umfangreichen Wissensbasis gesucht werden. Dabei hilft die **Datenbanktechnologie** , auf die ein kurzer Blick geworfen werden soll.

Im Falle der *adressierten* Speicherung wird das Abspeichern und Wiederfinden 4
mit Hilfe von Speicherplatzadressen realisiert. Die üblichen Datenbanken arbeiten nach diesem Prinzip. Wie der adressierte Zugriff hardwaremäßig realisiert wird, ist uns bekannt. Die technischen Einzelheiten sind in Kap.13.2 besprochen worden. Das Problem, das jetzt vor uns steht, liegt in der Anbindung der (externen) Nutzersemantik an die (interne) Maschinensemantik. Welche Prozesse müssen im Computer ablaufen, wenn beispielsweise ein Nutzer nach der Telefonnummer eines Bekannten fragt oder nach den Zugverbindungen von Berlin nach Wien?

Nach altbewährter Methode könnte man versuchen, das Problem durch Introspektion zu lösen. Bei der Simulation des Kopfrechnens sind wir so vorgegangen. Das war insofern möglich, als man selber weiß, wie man beim Rechnen verfährt. Aber was genau "*tut* man" (was passiert im Kopf), wenn man sich etwas merkt oder sich an etwas erinnert? Die Methode scheint zu versagen. In Kap.7.2 [7.9] hatten wir gesagt "*Gedächtnisleistungen sind Leistungen der reproduktiven, intuitiven Intelli-*

genz" und in Kap.7.1 [7.7] "*Intuition beruht auf nichtbewusster Verarbeitung von nichtbewusstem Wissen*".

Danach scheint uns die Introspektion nicht weiterhelfen zu können. Dennoch kann sie es. Denn wenn man sich an etwas nicht erinnern kann, es aber dennoch versucht, beginnt ein Suchprozess, dessen man sich mehr oder weniger deutlich bewusst ist. Offenbar handelt es sich um ein Weitertasten ("Navigieren") in einem "Meer" von Erinnerungen oder - unter Verwendung früher eingeführter Begriffe - um das Verketten von *Denkobjekten* mittels *Assoziation*.

In Kap.5.5 [5.19] hatten wir Assoziieren (als Methode des Zugreifens auf Gedächtnisinhalte) folgendermaßen definiert (der besseren Lesbarkeit halber ist überall "Merkmalswert" durch "Merkmal" ersetzt):

Assoziieren ist entweder

- das Zugreifen auf Objekte mittels eines oder mehrerer ihrer Merkmale oder
- das Zugreifen auf ein oder mehrere Merkmale mittels eines Objekts, das diese Merkmale besitzt, oder
- eine Kombination dieser beiden Zugriffsmethoden.

Eine Kombination liegt z.B. vor, wenn mit einem Merkmal eines Objekts ein anderes Merkmal desselben Objekts assoziiert wird oder wenn mit einem Objekt ein anderes Objekt über ein gemeinsames Merkmal assoziiert wird. Die zuletzt genannte Art des Assoziierens liegt der Verwendung *mnemotechnischer Hilfsmittel* zugrunde, wobei das gemeinsame Merkmal (oft eine gemeinsame Silbe in den Bezeichnern) die "Eselsbrücke" zum gesuchten Objekt bildet (vgl. auch die Beispiele in Kap.5.5 [5.19]).

In den zitierten Sätzen steht "Objekt" stets für "*Denkobjekt*". Erinnert man sich nun noch an die Begriffsbestimmung des Denkobjekts als Tupel von Merkmalswerten und bedenkt, dass das Faktenwissen hinsichtlich eines Objekts ein Merkmalswertetupel ist, erkennt man einen Weg zum maschinellen Erinnern. Es ist ein Mechanismus zu implementieren, der es erlaubt, in einer großen Menge von Merkmalswerten bestimmte (gesuchte) Merkmalswerte mit Hilfe anderer (vorgegebener) Merkmalswerte aufzufinden.

Zu diesem Zweck wird für das Merkmalswertetupel eines jeden Objekts des Diskursbereichs ein Speicherbereich vorgesehen, der in Felder unterteilt ist, für jedes Merkmal ein Feld. *Ein gespeichertes Tupel wird* ***Datensatz*** *genannt. Ein Datensatz enthält für jedes Merkmal eines Merkmalstupels ein* ***Feld****. Die Datensätze der verschiedenen Objekte können in einer* ***Datei*** *verbunden werden. Die Zusammenfassung einer oder mehrerer, eventuell auch sehr vieler Dateien und eines sog.* ***Datenbankbetriebssystem*** *wird* ***Datenbank*** *genannt. Die Dateien einer Datenbank bilden die sog.* ***Datenbasis*** *der Datenbank.* Die Datenbasis ist also die Gesamtheit aller Daten, die in einer Datenbank abgespeischert sind. Die Daten einer Datenbank lassen sich auch nach anderen Gesichtspunkten zu Dateien verbinden. Beispielsweise können die Inhalte (Werte) eines bestimmten Feldes (Merkmals) aller Datensätze (Objekte) in einer Datei zusammengefasst werden, und es kann für jedes Merkmal

eine spezielle Datei eingerichtet werden. Die Struktur der Datenbasis einerseits und die Strategien des Speicherns und Suchens andrerseits müssen so aneinander angepasst sein, dass sich minimale Speicher- und Suchzeiten ergeben. Das Speichern und Suchen ist eine der Aufgaben des Datenbankbetriebssystems.

Das Suchen in einer Datei soll am Beispiel eines "maschinellen Telefonbuchs", einer "*Abonnentendatei*" demonstriert werden. Die Datei bestehe aus Datensätzen, die den Eintragungen eines *gedruckten* Telefonbuchs zu einem bestimmten Abonnenten entsprechen. Um eine Rufnummer in einem gedruckten Telefonbuch zu finden, sucht man nach dem Familiennamen, meistens zusätzlich nach dem Vornamen und, wenn notwendig, auch noch nach der Straße oder nach dem Beruf. Diese Suchmethode kann auf die Abonnentendatei übertragen werden. Ein Merkmalswert bzw. ein Teiltupel, das einem Datensatz eindeutig zugeordnet ist (das nur ein einziges mal vorkommt), heißt **Schlüssel**. Um auf einen Datensatz (und damit auf die einzelnen Felder) über seinen Schlüssel zugreifen zu können, muss ein "Adressbuch" implementiert sein, das jedem Schlüssel die Adresse des betreffenden Datensatzes zuordnet. Will man die Telefonnummer eines Teilnehmers erfahren, muss man dessen Schlüssel eingeben. Daraufhin sucht das Datenbankbetriebssystem im Adressbuch nach dem Schlüssel. Wenn es ihn findet, kennt es die Adresse des betreffenden Datensatzes und kann die gewünschte Telefonnummer auslesen. Als Außenstehender kann man den Eindruck gewinnen, als "*assoziiere*" der Computer wie der Mensch mit dem Schlüssel (z.B. dem Namen) die Telefonnummer.

Durch die Wörter *Assoziieren* und *Schlüssel* wird der Leser vielleicht an Kap.13.2.2 [13.3] erinnert. Dort war vom *Assoziativspeicher* die Rede und vom *Schlüsselwort*, das einen Teil eines Speicherplatzinhalts bildet und gleichzeitig als "Adresse" (*Suchargument*) des Speicherplatzes dient. Dort war auch schon auf die Analogie des "*assoziativen Zugriffs*" in einem Assoziativspeicher einerseits und in einer Datenbank andrerseits hingewiesen worden. Die Schlüsselwörter in einer Datei (Datenbank) spielen dieselbe Rolle wie die Suchargumente in einem Assoziativspeicher oder wie der gemeinsam mit einer Nachricht verschickte Schlüssel bei Anwendung des Schlüssel-Schloss-Prinzips [12.4]. 6

Wie nahe das skizzierte Zugriffsprinzip über Merkmalswerte dem menschlichen Erinnerungsvermögen kommt, wird durch folgendes Beispiel verdeutlicht. Auf einem Spaziergang im August sieht man in der Ferne ein Kornfeld mit blauen Farbflecken. Mit den Merkmalswerten Farbe, Standort und Jahreszeit wird ein Naturliebhaber wahrscheinlich sofort (ohne nachzudenken) Kornblumen assoziieren. Hat er noch nie eine Kornblume gesehen, kann er sie mit Hilfe eines Blumenerkennungsbuches bestimmen, indem er nach den genannten Merkmalen sucht. Das Vorgehen lässt sich simulieren. Für jede Blume wird ein Datensatz abgespeichert. Das Anfragetupel (blau, Acker, August) wird der Computer wahrscheinlich mit mehreren Namen beantworten, z.B. "Ackerskabiose, Wegwarte, Kornblume". Für eine eindeutige Antwort bedarf es weiterer Merkmale, was evtl. schwieriger zu verwirklichen ist, am anschaulichsten mit Hilfe einer Zeichnung.

Das Blumenbeispiel hat vielleicht den einen oder anderen Leser auf eine weitere Idee gebracht, das Vorgehen des Menschen zu simulieren und zwar das "Heranpirschen" an das gesuchte Objekt durch immer genauere Beschreibung, durch *Präzisierung*, d.h. durch Absteigen in einer Klassenhierarchie (siehe Bild 5.4). Zunächst wird die Klasse "Blau" aufgeschlagen. In ihr wird nach der Unterklasse "Acker" und schließlich nach "August" gesucht. In jedem Präzisierungsschritt wird ein weiteres Merkmal in die Suche einbezogen, sodass der *Suchraum* immer enger begrenzt wird.

7 Diese Suchmethode setzt eine *begriffliche Strukturierung* der Wissensbasis voraus, genauer eine *klassifikatorische* Strukturierung. Grundlage sind die begriffsbildenden Operationen *Klassifizieren* und *Generalisieren*. Hier zeigt sich, dass Begriffsbildung nicht nur der Erhöhung der Ausdruckskraft (der *semantischen Verdichtung*) der Sprache dient, sondern auch dem Sicherinnern, dem Navigieren im eigenen Wissen. Es stellt sich die Frage, ob sich dafür noch andere begriffsbildende Operationen eignen.

Ein Blick auf Bild 5.4 zeigt, das dies für das *Komponieren* (*Aggregieren*) offensichtlich der Fall ist. Beispielsweise kann bei der Nutzung der Wissensbasis eines Fertigungsbetriebes das *dekomponierende* Suchen angezeigt sein. Bei der Suche nach dem Geburtstag eines Mitarbeiters könnte in der Verwaltungshierarchie des Betriebes "abgestiegen" werden, bei der Suche nach den Maßen einer Welle eines Motors in der Hierarchie der Bauelemente der Fertigungsprodukte.

Diese wenigen Hinweise sind im Grunde ausreichend, um vieles von dem nachzuerfinden, was auf dem Gebiete der Datenbanktechnologie entwickelt worden ist[7]. Allerdings darf nicht der Eindruck entstehen, dass die Datenbanktechnologie auf das Wiederauffinden gespeicherter Daten beschränkt ist. Sie umfasst weit mehr, denn eine Datenbank besteht aus der **Datenbasis** und dem **Datenbankbetriebssystem**. Letzteres ist die zum *Betreiben*, d.h. zur Nutzung und Wartung der Datenbasis erforderliche Software. Die *Datenbasis* ist die Gesamtheit aller Daten, also sämtlicher gespeicherten Merkmalswerte. Zu ihrer Wartung gehört u.a. das Erweitern und Aktualisieren der Datenbasis sowie deren Konsistenzerhaltung.

So eindruckvoll der Stand der Technik auch ist, mit der Leistungsfähigkeit des menschlichen Gehirns kann sich die Technik nicht messen. Man überlege sich, wie viele Felder der Datensatz zum Denkobjekt "Erde" oder "Meine Mutter" enthalten müsste und welches Assoziationsfeld durch diese Worte berührt wird. Der volle Umfang externer Semantik passt in keinen Computer. Auf diese Grenze der KI kommen wir in Kap.17 zurück.

---

7 Aus der umfangreichen Datenbankliteratur seien [Wedekind 91], [Wedekind 89][Lockeman 93] und [Kemper 97] genannt.

16.3 Erfinden

Wir haben uns überzeugen können, dass nicht nur die natürliche, sondern auch die künstliche Intelligenz die Fähigkeit zum Schlussfolgern besitzt. Damit ist aber die Frage nach ihren Grenzen nach wie vor nicht beantwortet. Die Grenze der natürlichen Intelligenz ist jedenfalls *nicht* erreicht. Der Mensch kann nämlich nicht nur rechnen und schlussfolgern, er kann auch erfinden, m.a.W. er besitzt die Fähigkeit, neue richtige Aussagen nicht nur auf deduktivem Wege (durch Schließen oder Rechnen), sondern auch auf *intuitivem* Wege, d.h. durch *unbewusste Wissensverarbeitung* zu finden. Diese Art des Findens hatten wir **Erfinden** genannt.

Der so definierte Begriff des Erfindens schließt das Erfinden technischer Funktionsprinzipien (z.B. der Dampfmaschine) insofern ein, als diese Art des Erfindens die gedankliche Vorwegnahme der Erfindung voraussetzt, m.a.W., dass er das "Erfinden der Aussagen" voraussetzt, die der technischen Erfindung zugrunde liegen (vgl. Kap.7.1 [7.4]).

Ursprünglich schien uns die Simulation intuitiver Intelligenz ein hoffnungsloses Unterfangen zu sein. Dennoch wollen wir versuchen, das Erfinden zu simulieren. Bestärkt werden wir von unserem Erfolg, Assoziation und Erinnerungsvermögen in gewissem Umfange zu simulieren, obwohl beide vorwiegend auf *unbewusster* Wissensverarbeitung beruhen.

Wir gehen von einer konkreten Erfindung aus und überlegen uns, ob auch der Computer sie hätte machen können. Wir wählen dafür die Erfindung der Struktur des Benzolringes durch AUGUST KEKULÉ. Offensichtlich handelt es sich dabei um eine *Erfindung* im Sinne unserer Definition. Sicher hat Kekulé keine Entdeckung gemacht. Zwar war das Benzol etwas bereits Vorhandenes. Gesucht ist aber dessen molekulare Struktur, und die war nicht "zu sehen", sie konnte nicht experimentell "*entdeckt*" werden. Auch scheint Kekulé die gesuchte Struktur nicht abgeleitet zu haben, denn die Lösungsidee ist ihm in einem Tagtraum gekommen, den er selber folgendermaßen beschreibt[8].

"Ich drehte meinen Stuhl zum Feuer und fiel in einen Dämmerschlaf. Wieder tanzten die Atome vor meinen Augen. [...] lange Ketten, oft enger miteinander verbunden, waren alle in Bewegung, ineinander verschlungen wanden sie sich wie Schlangen. Aber Achtung, was war das? Eine der Schlangen hatte ihren eigenen Schwanz gepackt und drehte sich vor meinen Augen spöttisch im Kreis. Blitzartig schreckte ich hoch [...]."

Offenbar war der Traum die Fortsetzung des Suchens in den Halbschlaf hinein. Es stellt sich die Frage, worin der Unterschied zwischen der erfolg*losen* Suche im

8 Entnommen aus [Ortoli 98]

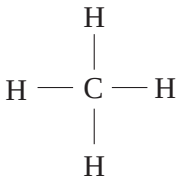
Wachen und der erfolg*reichen* Suche im Halbschlaf bestand. Eine einleuchtende Antwort auf diese Frage könnte helfen, Erfinden zu simulieren.

Im wachen Zustand hat Kekulé sicher alle aus seiner Sicht denkbaren Verkettungen von Wasserstoff- und Kohlenstoff-Atomen (H- und C-Atomen) durchprobiert, in einer Art Puzzlespiel. Wir wollen versuchen, das Puzzeln zu systematisieren und zu algorithmieren. Dazu gehen wir ganz ähnlich vor wie beim Implementieren des Schlussfolgerns. Zunächst suchen wir Fakten und Regeln, nach denen die Molekülbildung erfolgen kann. Dabei gehen wir vom chemischen Allgemeinwissen hinsichtlich unseres Problems aus, das wir in 6 Axiomen aufschreiben.

8 Axiom 1: H-Atome sind einwertig.
Axiom 2: C-Atome sind vierwertig.
Axiom 3: Ein C-Atom kann über jede seiner Valenzen ein H-Atom binden.
Axiom 4: Zwischen zwei C-Atomen sind Einfach- und Doppelbundungen möglich.
Axiom 5: Andere Bindungen gibt es nicht[9].
Axiom 6: Ein Molekül hat keine freien Valenzen.

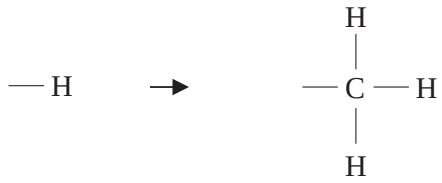
Beim Suchen nach möglichen (den Axiomen nicht widersprechenden) Verbindungen wollen wir etwa so vorgehen, wie auch Kekulé aufgrund seines Wissens vorgegangen sein könnte. Wir beginnen mit einer bekannten Verbindung und verändern sie durch Substitution gemäß "vernünftiger" Regeln. Vernünftig heißt, dass die Regeln dem Wissen und den Vorstellungen der damaligen Zeit entsprechen. Als Ausgangsverbindung wählen wir das Methan (CH_4). Seine Existenz stellt einen "Fakt" dar. Wir notieren:

Fakt

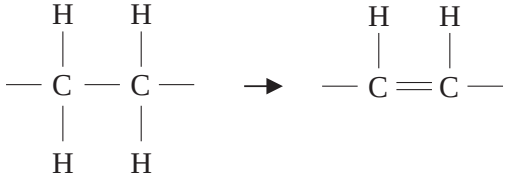


Als Regeln wählen wir

Regel 1



Regel 2



9 Die Möglichkeit von Dreifachbindungen wird außer Betracht gelassen, um die weiteren Überlegungen nicht unnötig zu komplizieren.

Wahrscheinlich wird auch Kekulé diese oder entsprechende Regeln bewusst oder unbewusst verwendet haben.

Ausgehend von diesem Wissen können C- und H-Atome zu linearen und verzweigten Ketten verbunden werden. Regel 1 erlaubt die Substitution eines H-Atoms durch eine CH_3-Gruppe. Regel 2 erlaubt die Substitution einer C_2H_4-Gruppe (Teilkette) durch eine C_2H_2-Gruppe. Das Puzzeln mit den Strukturformeln geht folgendermaßen vor sich. Auf das Methanmolekül lässt sich zunächst nur Regel 1 anwenden. Auf das Ergebnis (C_2H_6, Äthan) lässt sich auch Regel 2 anwenden, was C_2H_4 (Äthylen) ergibt. Auf beide Produkte lässt sich wieder Regel 1 und anschließend Regel 1 oder Regel 2 anwenden und so weiter. Durch (n-1)-malige Awendung von Regel 1 auf Methan wird die Verbindung C_nH_{n+2} "produziert". (Das Wort "produziert" ist mit Hinblick auf Kap.16.4 der Sprechweise der Theorie generativer Grammatiken entlehnt). Hinter dieser Formel verbirgt sich eine Vielfalt molekularer Strukturen. Es ist nämlich nicht gleichgültig, an welcher Stelle das H-Atom substituiert wird, an einem endständigen oder an einem nichtendständigen C-Atom einer Kohlenwasserstoffkette. Dabei können unterschiedliche, verzweigte Strukturen, entstehen, was zu unterschiedlichen chemischen Eigenschaften führen kann.

Der Generierungsalgorithmus ist also doppelt indeterministisch. Es kann zwischen verschiedenen Regeln und zwischen verschiedenen Stellen gewählt werden, an denen eine Regel angewendet werden soll, ganz analog zu den Algorithmen des analytischen Rechnens und des Inferenzierens. Das überrascht sicher nicht sehr. Offensichtlich gilt ganz allgemein: Regelbasierte Algorithmen der *Zeichenmustertransformation* mittels Substitutionsregeln sind zweifach indeterministisch, solange die Wahlmöglichkeiten nicht durch zusätzliche Vorschriften eingeschränkt werden. Im Falle der zweidimensionalen Sprache der chemischen Strukturformeln sind Kompositzeichen i.Allg. keine Zeichenketten, sondern Zeichenmuster aus den elementaren Zeichen (Buchstaben für Atome und Strecken für Valenzen).

Stillschweigend haben wir soeben die Generierung chemischer Verbindungen als *Zeichenmusterverarbeitung* aufgefasst. Wir dürfen sie als *Informationsverarbeitung* auffassen, wenn die Zeichenmuster semantisch belegt, d.h. interpretiert sind. Im Diskursbereich "Chemie" ist das der Fall; sie sind als (durch) Chemikalien interpretiert. Das gilt allerdings nicht für alle Verbindungen, die algorithmisch generierbar sind, denn viele von ihnen existieren nicht, weil sie nicht stabil sind. Stabilitätsbedingungen berücksichtigt unser Algorithmus nicht, sodass er eine teilweise nicht existente (virtuelle) Welt modelliert. Das ist, wie wir aus Kap.6.1 wissen, eine charakteristische Freiheit des sprachlichen Modellierens.

Die vorangehenden Überlegungen weisen den Weg, der zu gehen ist, um den Computer zum Puzzeln zu befähigen. Man erkennt nämlich, dass das Puzzeln in ganz ähnlicher Weise mit Formeln manipuliert, wie das analytische Rechnen in Kap.15.8 und das Inferenzieren in Kap.16.1. Das Regel- und Faktenwissen spielt dabei eine etwas andere Rolle als beim Inferenzieren. Das Faktenwissen, die Strukturformel des Methan, entspricht vielmehr dem Ansatz und das Regelwissen der Formelsammlung

beim Lösen eingekleideter Aufgaben im Mathematikunterricht [15.13]. Von der Fragestellung her entspricht das Puzzeln dem Theorembeweisen bzw. dem Inferenzieren, wenn entschieden werden soll, ob eine Hypothese zutrifft, beispielsweise die Hypothese "Der Vorgestellte ist mein Vater".

Um den Computer zum Puzzeln zu befähigen, muss ein Formelmanipulator implementiert werden, der mit zweidimensional notierten Formeln manipuliert. Wir werden auf die Einzelheiten des "Puzzlealgorithmus" nicht eingehen, sondern nehmen an, dass er implementiert ist, und beauftragen den Computer, solange zu suchen (zu puzzeln), bis er die Verbindung C_6H_6 gefunden hat. Die Aufgabe lässt sich auch anders artikulieren, beispielsweise: Der Rechner soll *entscheiden*, ob die Zeichenkette C_6H_6 durch den Algorithmus *generierbar* ist. Wir geben drei weitere Artikulierungen dieser Frage an:

- Ist der Ausdruck C_6H_6 in dem durch die Regeln festgelegten Kalkül *ableitbar*?
- Ist das *Theorem* "C_6H_6 ist existent" wahr?
- Ist die Zeichenkette C_6H_6 in der *Sprache* enthalten, die durch die Regeln festgelegt ist?

Die letzte Frage ist ein Vorgriff auf das nächste Kapitel. Sie wird verständlich, wenn man weiß, dass in der Theorie formaler Sprachen eine Sprache *extensional* als Menge aller zugelassenen Zeichenketten definiert wird.

Die drei Fragen sollen die Zusammenhänge zwischen verschiedenen Gebieten der KI verdeutlichen, dem *Schließen*, dem *Theorembeweisen* und dem *Übersetzen* oder allgemeiner der formalen *Analyse und Synthese sprachlicher Ausdrücke*. Auf einer ausreichend hohen Abstraktionsebene betreffen die verschiedenen Fragen ein und denselben Sachverhalt; sie werden identisch miteinander (ein Beispiel für begriffliche Konvergenz [8.38]). Infolgedessen ist es auch nicht verwunderlich, dass überall die gleichen Arten von Indeterminismus auftreten, dass überall gesucht werden muss und dass die Methoden des Suchens und seine Darstellung durch Graphen, insbesondere durch Bäume, in den verschiedenen Bereichen der KI ein und dieselben sind.

Wenn Kekulé nach unserem Algorithmus vorgegangen ist, musste seine Suche erfolglos bleiben, solange er sich nicht von dem Algorithmus lösen konnte. Denn die Verbindung C_6H_6 ist nicht generierbar, sie liegt - so die Sprechweise der Mathematiker - außerhalb der *Suchraumes*, der durch die beiden Regeln *aufgespannt* ist[10]. Offenbar beschreibt der Algorithmus die Wirklichkeit nur unvollständig. Ein besserer Algorithmus ist erforderlich, m.a.W. das Regelwissen muss erweitert werden.

Wer die Strukturformel von Benzol kennt, wird eine Regel suchen, die Ringverbindungen produziert. Kekulé wusste es nicht. Der Computer weiß es auch nicht. Kann man ihn trotzdem dazu befähigen, sich eine neue Regel "auszudenken"? Offensichtlich ist das möglich. Es ist nicht sehr schwierig, ein Programm zu entwer-

10 Wenn Dreifachbindungen zugelassen werden, ist C_6H_6 generierbar, allerdings nur formal, nicht tatsächlich im Labor.

fen, das mit Hilfe eines Zufallszahlengenerators Verbindungen aus C- und H-Atomen mit einer oder mehreren freien Valenzen generiert, die den Axiomen nicht widersprechen. Man beachte, dass auch der Programmierer, der das Programm schreibt, nichts von Ringverbindungen zu wissen braucht. Angenommen der Computer verfügt über ein solches Programm und er generiert die Regel,

Regel 3

9

nach der zwei H-Atome durch eine CH_2-Gruppe substituiert wird. Wenn der Computer diese Regel seinem Regelwissen hinzufügt und mit dem erweiterten Wissen puzzelt, wird er früher oder später den Benzolring generieren. Man beachte, dass auch Regel 1 und Regel 2 hätten "erwürfelt" werden können. Tatsächlich lassen sich alle denkbaren Regeln aus den Axiomen *herleiten*, ähnlich wie sich die Rechenregeln eines axiomatisierten Kalküls aus den Axiomen herleiten lassen.

Es stellt sich heraus, dass der puzzelnde Computer "in einem Kalkül rechnet". Das überrascht nicht, denn zu irgendetwas Anderem kann er nicht befähigt werden. Er rechnet sogar in einem axiomatisierten Kalkül. Allerdings ist der Kalkül nicht ordnungsgemäß formal definiert, es sei denn, man akzeptiert die graphische Beschreibung der Zeichenmuster und Regeln als Definition der Syntax der "CH-Sprache" (Sprache der Kohlen-Wasserstoff-Verbindungen). Wenn der Computer die CH-Sprache verstehen soll, muss sie implementiert werden, was die formale Definition ihrer Syntax einschließt.

Kekulé war, wie seine Zeitgenossen, offenbar nicht in der Lage, Regel 3 zu formulieren und gemeinsam mit den anderen Regel systematisch anzuwenden, zumindest nicht im wachen Zustand, weil seiner Phantasie durch Denkgewohnheiten Grenzen gesetzt waren. Denn durch Regel 3 wird eine Kette aus z.B. 5 C-Atomen zu einem Ring aus 6 C-Atomen "zusammengebogen". Wenn sich in dem Ring Einfach- und Doppelbindungen abwechseln, was sich mittels Regel 2 erreichen lässt, ergibt sich der Benzolring. Das lag offenbar jenseits der Grenzen damaliger Vorstellungsgewohnheiten. Im Halbschlaf verschwinden diese Grenzen. Im Traum ist das Ungewohnte, des Unerlaubte, das Ungesetzliche und das Unmögliche erlaubt und möglich. Der *Phantasie* ist freier Lauf gelassen. Hier tritt die gemeinsame Wurzel der Begriffe Intuition, Kreativität, Findigkeit und Phantasie deutlich zutage.

Der Computer verfügt über den Speicherinhalt hinaus über keinerlei Erfahrung, die das Suchen einschränken könnte. Darauf beruht sein Erfolg. Man könnte sagen: Er ist klug, weil er "dumm" ist, d.h. weil er wenig weiß. Dieser Satz kann auch auf Menschen zutreffen. Beispielsweise wäre es gar nicht ausgeschlossen gewesen, dass ein intelligenter Chemielaborant die Strukturformel von Benzol schneller hätte finden können, weil das Denken eines Laboranten weniger durch Spezialwissen eingeschränkt ist als das eines Professors.

Man überzeugt sich leicht, dass auch Regel 3 den durch die Axiome aufgespannten Suchraum noch einengt. Beispielsweise lässt sich gedanklich ein Molekül konstruieren, das aus einem Ring von C-Atomen besteht, die über Doppelbindungen miteinander verbunden sind. Ein solcher Ring enthält kein einziges H-Atom. Er widerspricht nicht den Axiomen

Zweifellos ließe sich ein Programm schreiben, nach dem der Computer schrittweise *sämtliche* Strukturen generiert, die sich mit den Axiomen im Einklang befinden. Der Computer könnte also beispielsweise beauftragt werden, aus *m* Kohlenstoffatomen alle erlaubten Strukturen aufzubauen, beginnend mit einer minimalen Anzahl von H-Atomen.

Bei Ausführung dieses Programms würden zunächst alle von den Axiomen her erlaubten reinen C_n-Moleküle generiert werden mit $n \leq m$. Derartige Verbindungen sind vor gar nicht langer Zeit im Labor gefunden worden; sie heißen **Fullerene**. Beispielsweise ist das Molekül C_{60} ein bekanntes Fulleren[11]. Der Computer würde auch die (topologische) Struktur des Diamanten komponieren. Allerdings könnte er nicht erkennen, dass es sich um ein *räumliches Gitter* handelt. Dazu müsste er mehr wissen. Irgendwann würde der Computer auf das Benzolmolekül stoßen. Die drei Verkettungsregeln benötigt er dafür nicht. Hätte es zu Kekulés Zeiten schon die KI gegeben, dann hätte ihm der Computer zuvorkommen können.

Das Benzolbeispiel zeigt, dass Findigkeit - sei es die eines Forschers oder die eines Rätselraters - entscheidend von der Fähigkeit abhängt, im "*richtigen*" *Suchraum* zu suchen. Dieser darf nicht zu eng, aber auch nicht zu weit sein, weil sonst das Suchen entweder erfolglos oder zu langwierig ist. Als sehr anschauliche Illustration eines zu engen Suchraumes sei folgende bekannte Denksportaufgabe angeführt.

Denksportaufgabe 2: Problem der Verbindungslinie

Die 9 Punkte in Bild 16.4a sind durch einen einzigen Streckenzug (einen Zug von vier geraden Linien ohne abzusetzen) zu verbinden. Solange man auf den Bereich innerhalb der Punkte fixiert ist, bleiben alle Versuche erfolglos, wie z.B. der in Bild 16.4a dargestellte. Erst wenn man über diesen Bereich hinausdenkt, kann man die in Bild 16.4b gezeigte Lösung finden. Der zu enge Suchraum enthält als Kandidaten für die Teilstrecken, aus denen sich der gesuchte Streckenzug zusammensetzt, nur diejenigen Geraden, auf de-

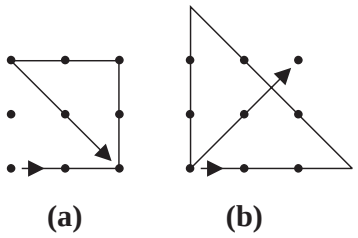


Bild 16.4 Denksportaufgabe 2.

11 Die Bezeichnung ist von dem Namen des amerikanischen Architekten R. Buckminster Fuller abgeleitet, dessen Kuppelbauten den C_n-Molekülen ähnlich sehen.

nen je 3 der 9 Punkte liegen. Der erweiterte Suchraum enthält außerdem diejenigen, auf denen 2 Punkte liegen.[12]

Es stellt sich die Frage, ob für die Lösungsfindung tatsächlich Intuition erforderlich ist, ob es sich also um *Erfinden* handelt oder im Grunde doch "nur" um *Schlussfolgern*. Das Benzolbeispiel legt folgende Antwort nahe: Das *regelbasierte Suchen* erfordert keine Intuition, während das *Formulieren der Regeln* nicht ohne Intuition möglich zu sein scheint.

Aber der Schein trügt, wie wir bereits wissen. Denn beispielsweise ergeben sich die Regeln, Regel 3 eingeschlossen, unmittelbar aus den Axiomen, worauf bereits hingewiesen wurde. Man braucht sich bei ihrer Formulierung nicht vorzustellen, zu welchen Konfigurationen sie führen können, insbesondere braucht man sich keine Ringverbindung vorzustellen (das wäre *externe* Semantik). Der Computer liefert sie automatisch, wenn er "stupide" die Axiome - aber eben nur diese - anwendet. Demnach ist zum Finden der Struktur des Benzolringes keine Intuition erforderlich. Andrerseits ist es ganz sicher gerechtfertigt, zu sagen, dass Kekulés Leistung auf Intuition beruhte. Seine Intuition war die *Erweiterung des Suchraumes*.

Um diesen Gedanken klarer zu artikulieren, führen wir den Begriff des Beschreibungsraumes ein. *Der **Beschreibungsraum** eines Originalbereiches ist die Menge aller Aussagen über den Originalbereich, die einer Menge von Grundaussagen nicht widersprechen. Die Menge der Grundaussagen nennen wir **Grundwissen***. Das Grundwissen spannt den Beschreibungsraum auf, ebenso wie das **Suchwissen** den **Suchraum** aufspannt. Das Suchwissen ist die Gesamtheit von Faktenwissen und Regelwissen. Wenn Grundwissen und Suchwissen zusammenfallen, fallen auch Beschreibungsraum und Suchraum zusammen. 10

Betrachtet man nun noch einmal kritisch das Benzolbeispiel, erkennt man, dass die Regeln 1 und 2 bzw. 1, 2 und 3 einen kleineren bzw. einen größeren Unterraum des Beschreibungsraumes festlegen, der seinerseits durch das Grundwissen (H ist einwertig; C ist vierwertig) festgelegt ist. Die drei Räume sind gewissermaßen ineinandergeschachtelt. Der Übergang in den nächst größeren Raum kann aus den Grundaussagen *abgeleitet*, also *ohne* Intuition gefunden werden.

Wenn der gedankliche Zugang in einen übergeordneten Suchraum durch Denkgewohnheiten versperrt ist, aber dennoch gefunden wird, ist offensichtlich Intuition erforderlich. Doch muss diese Art der Intuition von der "unabdingbar notwendigen Intuition" unterschieden werden. Darum vereinbaren wir: *Wenn eine Aussage auf intuitivem Wege gefunden worden ist, obwohl sie durch Ableitung hätte gefunden werden können, wenn also die intuitive Leistung auf Ableiten zurückgeführt werden kann, sprechen wir von **reduzibler Intuition***. 11

Ob jede Suchraumerweiterung ableitbar ist, erscheint zweifelhaft. Vielmehr setzt sie oft hohe intuitive Intelligenz voraus. Die Fähigkeit zum Erweitern des Suchrau-

12 Das Beispiel und seine formale Darstellung ist in [Schefe 87] behandelt.

mes durch Überwindung von Denkgewohnheiten war für die Fortschritte der Physik von hervorragender Bedeutung. Das ist verständlich, denn der Beschreibungsraum, in dem alle Beobachtungen auffindbar sein müssen, ist durch "eherne Naturgesetze" festgelegt, an deren Gültigkeit man sich gewöhnt und die man deswegen für absolut gültig hält. Doch kann sich herausstellen, dass sie relativiert werden müssen, um neue Beobachtungen erklären zu können. An einige Beispiele sei erinnert.

NICOLAUS KOPERNIKUS überwand die Vorstellung, dass die Sonne um die Erde kreist, und postulierte den umgekehrten Sachverhalt, wodurch sich die beobachtete Bewegung der Planeten am Himmel einfacher erklären ließ. Unter Erweiterung des Suchraums für die Planetenbahnen von Kreisen auf Ellipsen konnte JOHANNES KEPLER aus dem von TYCHO DE BRAHE bereitgestellten Faktenwissen seine Gesetze ableiten. Diese Leistung vollbringt heute bereits der Computer, abgesehen von der Festlegung des Beschreibungsraums. Wir kommen darauf noch einmal in Kap. 21.4 zurück.

MAX PLANCK überwand die Denkgewohnheit, dass die Energie einen kontinuierlichen Wertebereich besitzt, und ALBERT EINSTEIN machte sich von der Vorstellung frei, dass Geschwindigkeiten sich einfach vektoriell addieren, z.B. die Geschwindigkeit einer Person oder eines Gegenstandes *in* einem Zug (relativ zum Zug) und die Geschwindigkeit des Zuges (relativ zu einem ruhenden Bezugssystem, z.B. zur Erdoberfläche).

Die Frage ist erlaubt, ob bzw. wann der Computer zu derartigen Leistungen fähig ist. Die Antwort ist eigentlich trivial: Wenn er im Suchraum keine Lösung findet, aber über einen Beschreibungsraum verfügt, der über den Suchraum hinausgeht, kann man ihn per Programm veranlassen, den Suchraum ("seinen Horizont") zu erweitern. Das "Erfinden" neuer Regeln wird zum Ableiten, denn es erfolgt seinerseits *regelbasiert*, und zwar auf der Basis des Grundwissens. Dem entspräche im Benzolbeispiel das Ableiten der Regel 3. Das wäre die Simulation reduzibler Intuition, zu der Kekulé nur im Halbschlaf fähig war.

In diesem Kapitel ist es uns im Grunde nicht gelungen, die Grenzen der KI zugunsten des Computers zu verschieben. Wir haben zwar gesehen, dass bestimmte Leistungen, von denen gesagt wird, dass sie Intuition erfordern, simulierbar sind. Die genaue Analyse hat jedoch gezeigt, dass es sich dabei um Suchraumerweiterungen handelt, die auch ohne "echte" Intuition hätten gefunden werden können.

Oft ist aber nicht *reduzible*, sondern *echte* Intuition gefragt, die durch keinerlei Schlussfolgern ersetzt werden kann. Das gilt beispielsweise für das Ödipusrätsel. Wie ließe sich in diesem Fall der Beschreibungsraum und wie ließen sich Suchräume festlegen? Offenbar versagt die Methode. Bisher haben wir kein Rezept gefunden, diejenige Intelligenz zu simulieren, die notwendig ist, um derartige Rätsel zu lösen.

Es ist an der Zeit, die Ergebnisse, zu denen wir gelangt sind, zu resümieren. Ein kritischer Blick auf das bisherige Kapitel 16 zeigt, dass es sich im Grunde um eine Ergänzung des Kapitels 15.8 handelt. Denn in Kap.16 wurden Sonderfälle des analytischen Rechnens behandelt, für das in Kap.15.8 eine allgemeine Methode auf der Grundlage der Formelmanipulation erarbeitet worden ist. Der Unterschied zwi-

schen den Kapiteln 15 und 16 besteht darin, dass in Kap.15 Probleme behandelt wurden, mit deren Stellung auch der Kalkül vorgegeben war, in dessen Rahmen sie zu lösen sind. Dagegen musste für die Lösung der in Kap.16 gestellten Probleme ein passender Kalkül erst gefunden werden. Aus diesem Grunde werden die Probleme des Kapitels 15 "automatisch" als *mathematische* Probleme klassifiziert, während die Probleme des Kapitels 16 von unvoreingenommenen Lesern natürlicherweise als *nichtmathematische* Probleme klassifiziert werden.

Der tiefere Grund dieser Unterscheidung liegt darin, dass die Werte der Variablen in den Problemen des Kapitels 15 Zahlen sind, was auf die Probleme des Kapitels 16 nicht zutrifft. Mit anderen Worten, die sprachlichen Modelle in Kap.15 beruhen auf *Messen* und *Zählen*, in Kapitel 16 dagegen auf *Klassifizieren* ohne Bezugnahme auf Zahlen, d.h. ohne Quantifizierung. In Kapitel 16 sind wir in gewissem Sinne "zu den Wurzeln" der Rechentechnik zurückgekehrt, zur "Klassenlogik" des ARISTOTELES (vgl. Kap.11.1). Aus dieser Sicht hätte die Behandlung der boolesche Algebra in Kap.16 eingeordnet werden müssen. Sie musste vorgezogen werden, weil wir die boolesche Algebra als hardwaremäßig zu realisierenden Kalkül ausgewählt hatten.

Für drei der vier Grundideen des elektronischen Rechnens [8.42] haben wir uns Realisierungsmöglichkeiten überlegt; es fehlt die vierte, das automatische Übersetzen. Aus der Sicht des übersetzenden Menschen, beispielsweise eines Dolmetschers, ist Übersetzen eine nichtmathematische Leistung menschlicher Intelligenz. Um das Übersetzen zu automatisieren, muss es kalkülisiert werden. Insofern gehört das Übersetzungsproblem, dem wir uns nun zuwenden, in das Kapitel "Lösen nichtmathematischer Probleme".

16.4 Übersetzen und die vierte Grundidee des elektronischen Rechnens

Bei unseren Überlegungen darüber, wie der Computer nichtmathematische Aufgaben lösen kann, ging es um die Kalkülisierung verschiedener Denkprozesse, speziell des Schlussfolgerns und des Erfindens. Dabei bestand das Kalkülisieren darin, dass eine geeignete formalisierte Sprache definiert wurde und der Denkprozess auf das Ableiten von neuen aus alten (als wahr vorausgesetzten) Sätzen dieser Sprache nach vorgegebenen Regeln zurückgeführt wurde. Das Ergebnis ist ein spezieller Kalkül, genauer eine spezielle Beschreibung von Denkprozessen in Form eines Kalküls. Wir haben ihn *speziellen Denkkalkül* genannt.

Damit ist der erste Teil von Ziel 2, wie es zu Beginn von Kap.13 [13.1] formuliert wurde, erfüllt. Der zweite Teil steht noch aus, die Überführung des speziellen Denkkalküls in den Maschinenkalkül, m.a.W. die Übersetzung der speziellen Kalkülsprache in die Maschinensprache unter Erhaltung der Semantik. Die Idee, die Übersetzung vom Computer selber ausführen zu lassen, nennen wir die **vierte**

Grundidee der elektronischen Rechnens. In diesem Kapitel wollen wir uns überlegen, wie sich die Idee verwirklichen lässt.

Es wurde wiederholt darauf hingewiesen, dass das schwierigste Problem die Erhaltung der Nutzersemantik ist (der Semantik, "in" oder "mit" welcher der Nutzer denkt). Wenn ein Nutzer dem Computer einen Auftrag in der Sprache eines speziellen Denkkalküls erteilt, muss gewährleistet sein, dass der Auftrag das Beabsichtigte bewirkt, m.a.W. dass der Computer den Auftrag "richtig versteht". Wir wollen uns noch einmal klarmachen, was das bedeutet.

Im Falle der Denksportaufgabe 1 (Verwandtschaftsproblem) lautet die Frage "Welche Verwandtschaftsbeziehung besteht zwischen `Er` und `Ich`?" Dafür haben wir die Kurznotation `?(er,ich)` vereinbart. Im Benzolbeispiel lautet die Frage "Wie ist die Strukturformel von C_6H_6?", in Kurznotation z.B. `struk?(6,6)`. Man könnte auch die Frage stellen "Welche Kohlenwasserstoffmoleküle sind denkbar?" in Kurznotation z.B. `moleküle?(C,H)`.

12 Die Sprache, in der dem Computer derartige Fragen gestellt werden, hatten wir *Anfragesprache* genannt. Allgemein werden wir eine Sprache, in der einem Computer ein Auftrag erteilt wird (das kann auch eine Frage oder der Auftrag zur Prüfung einer Hypothese sein), **Auftragssprache** nennen. Im Falle der Verwandtschaftsaufgabe war die Auftragssprache an die Sprache des Prädikatenkalküls angelehnt. Im Falle eines Expertensystems für Chemie wird man sie an die Fachsprache für chemische Verbindungen und Reaktionen anlehnen. Der Computer muss die Ausdrücke der vereinbarten Auftragssprache *verstehen*.

In der Alltagskommunikation zwischen Menschen sagt man, dass eine Person B verstanden hat, was eine Person A gesagt hat, wenn B ausreichend genau weiß, was A gemeint hat, oder in der Sprechweise von Kap.2 und Bild 2.1, wenn B einem von A empfangenen Zeichenrealem ein Idem zuordnet, das ausreichend genau mit dem Idem übereinstimmt, das A artikulieren wollte. Ob die Genauigkeit der Interpretation ausreichend war, zeigt sich im Verhalten (im Tun und Sagen) von B. Wenn A von B nicht ausreichend genau "verstanden" worden ist, kann sich das sofort, eventuell aber auch erst nach langer Zeit oder nie herausstellen. A kann die Genauigkeit des Verstehens kontrollieren, wenn B die Nachricht mit seinen eigenen Worten wiederholt.

Die Interpretation einer Nachricht durch den Empfänger, die *Empfängersemantik* der Nachricht, manifestiert sich also letztlich im Handeln (Tun und Sagen) des Empfängers. Sehr deutlich wird das in den Worten der Mutter: "Hast du mich *verstanden*?!" Die Frage ist im Grunde eine nachdrückliche Mahnung: "*Tu*, was ich dir gesagt habe!"

Im Hinblick auf den Computer ist Verstehen definitionsgemäß identisch mit Handeln, denn als *trägerinterne* Semantik hatten wir denjenigen Prozess bezeichnet, der von einem Zeichenrealem (z.B. einem Befehl oder Programm) im Träger (im Computer) ausgelöst wird [5.7]. Interpretieren durch den Computer, d.h. Zuordnen

der internen Semantik, ist identisch mit Abarbeiten. *Der Computer "**versteht**" einen Auftrag, indem er ihn **ausführt**. Er versteht ihn **richtig**, wenn er ihn richtig ausführt.*

Hiergegen kann eingewendet werden, dass der Computer einen Auftrag "verstanden" haben muss, *bevor* er mit seiner Ausführung beginnen kann. Der Einwand enthält einen wahren Kern. Denn wenn dem Computer beispielsweise die Frage `struk?(6,6)` gestellt wird, kann er dieses Zeichenrealem zunächst einmal nur Zeichen für Zeichen lesen. In der eingelesenen Zeichenfolge kann er dann ihm bekannte Teilfolgen erkennen. Dazu muss er eine **lexikale Analyse** durchführen (wie der Assembler in Kap.15.4). Dabei erkennt er z.B., dass die Buchstabenfolge `struk` ein Lexem der Auftragsprache darstellt. Damit hat er aber noch nicht den eigentlichen Sinn des Auftrags erfasst, den der Mensch in ihn hineinlegt.

Die lexikale Analyse führt zwar zu einem gewissen "Verständnis" insofern, als die Analyse *metasprachliche* Aussagen liefert, also Aussagen *über* den betreffenden Satz, jedoch nicht zu einem Verständnis auf *objektsprachlicher* Ebene; die Bedeutung des Satzes kann nicht erschlossen werden. Was das heißt, wollen wir uns an einem Beispiel - wieder aus der Schule, diesmal aus dem Englischunterricht - klarmachen. Max soll folgenden Satz übersetzen, obwohl er kein Wort Englisch kann:

Peter understands this sentence. (16.1)

Max ist ein findiges Bürschchen und übersetzt: "Peter unterstand die Sentenz". Nach kurzem Nachdenken berichtigt er: "Peter *ver*stand die Sentenz". Was mag dabei alles in Maxens Kopf vorgegangen sein? Das ist ein weites Feld, das wir im Augenblick nicht betreten wollen. Nachdem Max ein Wörterbuch erhalten und einige Zeit darin gesucht hat, liefert er die richtige Übersetzung, wobei er ganz selbstverständlich das Objekt "Satz" in den Akkusativ setzt. Wenn Max nichts über englische Grammatik weiß und weniger findig ist, braucht er ein morphologisches Wörterbuch, aus dem er z.B. erfährt, dass "understands" die Wortform für die dritte Person Präsens von "understand" ist.

Die Vokabelsuche hätte der Computer eventuell schneller erledigt. Er hätte aber - selbst mit einem morphologischen Wörterbuch - nicht ohne Weiteres feststellen können, in welchem Fall die Wörter "Peter" und "sentence" stehen und welches von ihnen Subjekt und welches Objekt ist. Vielmehr muss er eine **syntaktische Analyse** 13
durchführen. Auch der Mensch kommt zuweilen nicht ohne bewusstes Analysieren der Syntax aus, insbesondere, wenn er die Sprache schlecht beherrscht.

Angenommen, Max hat in der letzten Stunde die syntaktische Konstruktion des AcI (Akkusativ mit Infinitiv) und dessen Übersetzung ins Deutsche gelernt. Nun soll er folgenden Satz übersetzen:

I heard him go away. (16.2)

Da ihm der AcI noch nicht in Fleisch und Blut übergegangen ist, übersetzt er ihn nicht "automatisch" richtig, sondern er führt zunächst eine *syntaktische Analyse*

durch, wobei er die gelernte *syntaktische Übersetzungsregel* anwendet. Die Regel sagt aus, dass dem englischen syntaktischen Konstrukt "AcI" das deutsche Konstrukt "Dass-Satz" entspricht, also ein Nebensatz, der mit "dass" beginnt. Dabei wird das Akkusativ-Lexem des AcI zum Subjekt und der Infinitiv zum Prädikat (im grammatikalischen Sinne) des Dass-Satzes.

Wir wollen uns nun überlegen, welche weiteren Hilfsmittel Max benötigt, um den Satz (16.2) richtig übersetzen zu können, *ohne* vorher den AcI gelernt zu haben, und wie er dabei vorzugehen hat. Offensichtlich benötigt Max neben dem morphologischen ein syntaktisches Wörterbuch, in dem zu allen englischen syntaktischen Konstrukten die deutschen Äquivalente angegeben sind, u.a. die obige AcI-Regel. Die syntaktischen Quell- und Zielkonstrukte müssen in einer geeigneten Metasprache artikuliert sein.

Damit ist das Übersetzungsproblem aber noch nicht gelöst. Max kann nämlich nicht sicher sein, ob ein erkanntes Akkusativ-Lexem mit nachfolgendem Infinitiv-Lexem ein AcI ist. Dafür müssen noch andere syntaktische Voraussetzungen erfüllt sein. Grundsätzlich muss die Lexemfolge des *gesamten* Satzes als syntaktisches Konstrukt erkannt sein, das den Syntaxregeln entspricht. Die Grundidee der syntaktischen Satzanalyse ist in Kap.5.3 am Beispiel einfacher deutscher Sätze dargelegt worden. Es besteht darin, dass Teile des Satzes zu *metasprachlichen Variablen* zusammengefasst werden (präziser: zu Klassen, die mit metasprachlichen Variablenbezeichnern benannt werden), die ihrerseits zu metasprachlichen Variablen höherer Ebene zusammengefasst werden, sodass ein Syntaxbaum entsteht (vgl. Bild 5.2). Wenn die Zusammenfassungen der metasprachlichen Variablen durch geeignete Klammerung gekennzeichnet wird, kann die zweidimensionale Beschreibung der sytaktischen Struktur in eine lineare überführt werden, z.B. in die Backus-Naur-Form [5.3]. Zur Illustration betrachten wir Maxens Vorgehen bei der Analyse des Satzes (16.2). Die lexikale Analyse ergibt die Lexemfolge

Pronomen im 1.Fall, Verbform, Pronomen im 4.Fall, Infinitiv.

Diese nichtformalisierte Darstellung ist sicher jedem verständlich. Um die syntaktische Struktur des Satzes zu erkennen, muss Max sich anhand des syntaktischen Wörterbuches davon überzeugen, dass ein Pronomen im 1. Fall die Rolle eines Subjekts und eine Verbform die Rolle eines Prädikats spielen können, dass ein Pronomen im 4. Fall mit nachfolgendem Infinitiv ein AcI sein kann und dass die Folge Subjekt, Prädikat, AcI ein Satz sein kann. Es ergibt sich folgende syntaktische Struktur der Lexemfolge (16.2):

Pronomen im 1.Fall, Verbform, (Pronomen im 4.Fall, Infinitiv),

wobei der AcI in runde Klammern gesetzt ist.

Es fragt sich nun, ob bzw. wie Maxens Methode automatisiert d.h. ob und wie Übersetzen kalkülisiert werden kann. Dass dies im Prinzip möglich ist, erkennt man, wenn man sich das Grundprinzip klarmacht, nach dem Max vorgeht. Es besteht darin,

dass er schrittweise Teilzeichenketten des Quelltextes durch andere Zeichenketten *substituiert*. In jedem Schritt sucht er zuerst nach einer passenden Teilzeichenkette (Lexemfolge) und anschließend substituiert er sie durch eine andere Zeichenfolge gemäß Wörterbuchregel. Genau so geht der Markovalgorithmus, das analytische Rechnen und das Schlussfolgern vor. Damit ist zwar noch nicht die Frage beantwortet, wie man in dem speziellen Fall des automatischen Übersetzens konkret zu verfahren hat, doch das Prinzip ist klar, und wir sind ausreichend vorbereitet, um in großen Zügen verstehen zu können, wie ein Übersetzerprogramm arbeitet.

Die Suche nach effektiven Methoden der Sprachübersetzung hat zur Herausbildung eigenständiger Gebiete der theoretischen und der praktischen Informatik geführt, nämlich der Theorie formaler Sprachen und der Übersetzerprogrammtechnik, speziell des Compilerbaus. Wir werden uns nicht in diese Gebiete vertiefen, sondern verweisen auf die Literatur[13] und begnügen uns mit der Herausarbeitung der zentralen Ideen sowie einer Andeutung ihrer Realisierungsmöglichkeiten.

Die Grundidee übernehmen wir von Max. Übersetzen ist eine Kompositoperation aus drei nacheinander auszuführenden Bausteinoperationen, der lexikalen Analyse, der syntaktischen Analyse und der Substitution. Die drei Operationsvorschriften (Programme) heißen **Scanner**, **Parser** und **Codegenerator**. Letzterer generiert den *Zielcode* (das Zielprogramm).

Bevor wir auf den Scanner eingehen, wollen wir uns überlegen, wie der Parser arbeitet, um zu erkennen, welche Informationen er vom Scanner erhalten muss. Wir beginnen mit folgender Definition: *Eine **Sprache** ist die Gesamtheit ihrer Sätze* oder abstrakter: *Eine Sprache ist die Menge aller zulässigen Zeichenketten*. Welche Zeichenketten (allgemeiner welche Zeichenkörper oder konkreter welche Wörter und welche Sätze[14]) erlaubt sind, wird von den Syntaxregeln (von der Syntax[15]) der Sprache bestimmt.

Wer durch das Wort "Syntax*regeln*" an die Regeln eines Kalküls und speziell an die Substitutionsregeln des Schlussfolgerns erinnert wird, ist der Lösung unseres Problems schon ein Stück näher gekommen. Wenn man nun noch an die *Generierung* der CH-Sprache (der Formeln von Kohlenwasserstoffen) denkt, steht man unmittelbar vor der Erfindung der sogenannten **generativen Grammatiken**. In Kap.16.3 war von der *Generierung* der chemischen Formel des Benzols, also der Zeichenkette C_6H_6 die Rede. Dabei standen dem Computer drei Substitutionsregeln zur Verfügung, durch die alle erlaubten Molekülketten und damit alle "erlaubten Zeichenketten" festgelegt sind, deren Gesamtheit die CH-Sprache bildet.

---

13 Z.B. [Schöning 95], [Schmitt 92], [Aho 92].

14 In der theoretischen Literatur wird statt "Satz" oft "Wort" gesagt und zwischen Wort und Satz nicht unterschieden.

15 In der Sprachlehre wird die Lehre vom Satzbau als Satzlehre oder Syntax bezeichnet (vgl. die Definition zu Beginn von Kap.5.3). Sie ist Bestandteil der Grammatik.

Wir abstrahieren nun von jeglicher Semantik und definieren eine fiktive Sprache ganz formal, beispielsweise durch folgende Regeln:

```
    w → aa
   aa → aaaa
 aaaa → b;
```

`w` steht für "Wort" und stellt das **Spitzensymbol** dar. Gemäß diesen Regeln lassen sich z.B. folgende Wörter generieren: `aa, aab, aaaa, b, bb, aabb`. Die so definierte "*Sprache*" (Menge von Zeichenketten) enthält alle "Wörter", die aus Teilketten aus einer geraden Anzahl des Zeichens `a` und Teilketten aus einer beliebigen Anzahl des Zeichens `b` bestehen. In einer der Backus-Naur-Form ähnlichen Notation könnte dieser Sachverhalt folgendermaßen notiert werden: [[aa]*[b]*]*, wobei der Stern beliebig häufige, auch 0-malige Aneinanderreihung der geklammerten Kette bezeichnet. Man beachte, dass diese Notation keine syntaktische (im Sinne der Syntax einer Sprache), sondern lediglich eine *lexikale Struktur* beschreibt. Man kann die einzelnen Zeichen auch als Lexeme auffassen. Dann beschreiben die Regeln alle Lexemfolgen der betreffenden Sprache.

Auf diese Weise lassen sich sehr einfache Sprachen beschreiben, generieren oder definieren (welches Verb hier passt, hängt vom Standpunkt des Betrachters ab), z.B. die Sprache der arabischen oder der römischen Zahlen. Anhand der Regeln lässt sich auch feststellen, ob eine gegebene Zeichenkette ein Wort bzw. eine Lexemfolge der betreffenden Sprache ist. Für komplexere Sprachen scheint die generative Methode ungeeignet zu sein. Dennoch ist sie auf Grund einer weiteren Idee sogar für natürliche Sprachen erfolgreich einsetzbar.

Um uns dieser Idee zu nähern, vergegenwärtigen wir uns noch einmal, dass das Artikulieren, d.h. das Codieren von Idemen (von Semantik) bewusst oder unbewusst auf zwei Ebenen abläuft. Auf der objektsprachlichen Ebene wird der gedachten und zu artikulierenden Bedeutung (dem Bewusstseinsinhalt, dem Idem) eine Zeichenkette explizit zugeordnet, die eine *lexikale* (in der Regel hierarchische) Struktur besitzt (vgl. Bild 5.1). Dabei werden auf der metasprachlichen Ebene implizit den lexikalen Bestandteilen der Zeichenkette *metasprachliche Variable* zugeordnet wie Subjekt, Prädikat, Objekt. Dadurch erhält die Zeichenkette eine *syntaktische* Struktur.

Die entscheidende Idee besteht nun darin, den Generierungsmechanismus nicht nur auf die lexikale, sondern auch auf die syntaktische Struktur anzuwenden. Damit ist das Prinzip klar, nach dem der Parser vorzugehen hat. Voraussetzung ist, dass für die Quellsprache und für die Zielsprache je ein System von Syntaxregeln existiert, nach dem sich jeder Satz der Sprache syntaktisch beschreiben lässt. Für die deutsche Sprache wird das System u.a. die Regeln (5.1), (5.3) und (5.4) und für eine Maschinensprache die Regel (5.2) enthalten. Für die in Bild 15.3b verwendete Sprache gelten u.a. die Regeln[16]

16 Es ist hier nicht die allgemeine Definition der WHILE-Anweisung angegeben.

WHILE-anweisung → `WHILE` *ausdruck rel zahl* `DO` {*ergibtanweisung*} `ENDWHILE`

ergibtanweisung → *id* := *ausdruck* | *zahl*

ausdruck → *id* | *op(id)* | *id op id*

Im Gegensatz zu (5.2) sind die metasprachlichen Variablen nicht in spitze Klammern gesetzt, sondern kursiv gedruckt. Das Zeichen | steht für "oder" (alternative Möglichkeiten).

Die Lexeme `WHILE`, `DO` und `ENDWHILE` (Ende der WHILE-Anweisung) gehören zur Objektsprache und heißen **Schlüsselwörter**. Variablenbezeichner sind mit *id* (von **Identifikator**) abgekürzt, *op* steht für Operationszeichen und *rel* für Relatopnszeichen. Es handelt sich um Namen für Lexemklassen, z.B. *id* für die Klasse der Identifikatoren. Man kann das Dekomponieren als "Rechnen mit Variablen" auffassen, wobei die Klassennamen die Rolle von Variablen speziell von *metasprachlichen* Variablen spielen, denen Werte (z.B. konkrete Bezeichner) zugeordnet werden. Darum sind sie, wie beim analytischen Rechnen, kursiv gedruckt. Die geschweiften Klammern bedeuten diesmal, dass der geklammerte Teil sich beliebig oft wiederholen darf, aber mindestens einmal auftreten muss.

Die Regelliste ist nicht vollständig. Zum einen fehlen die Regeln für *zahl* und *id*, zum anderen ist die Regel für *ausdruck* unvollständig, wie der Vergleich mit Bild 15.3b zeigt. Die meisten Zeilen des dortigen Programms werden zwar richtig beschrieben, die Ergibtanweisung für `s` jedoch nicht. Wer versucht, die Regel für *Ausdruck* um weitere Alternativen zu ergänzen, wird feststellen, dass dies gar nicht so ganz einfach ist. Wir verzichten auf die notwendige Erweiterung, da die Regel in der verkürzten Form für die weiteren Überlegungen ausreicht.

Um an Hand derartiger Regeln die syntaktische Struktur (die "Syntax") eines Programms zu erkennen, kann der Parser folgendermaßen vorgehen. Beginnend mit dem Spitzensymbol (mit der metasprachlichen Variablen *Program*) "dekomponiert" er schrittweise jede auftretende metasprachliche Variable, indem er sie durch die rechte Seite einer passenden Syntaxregel substituiert. Der Einfachheit halber nehmen wir an, dass ein Programm laut Syntaxregel eine Folge von Anweisungen ist (vgl. (15.6) in Kap.15.4; den Deklarationsteil unterschlagen wir). Die Dekomponierung des Quelltextes in Anweisungen macht keine Schwierigkeiten, wenn die Syntaxregel vorschreibt, dass Anweisungen durch Trennzeichen (z.B. Semikolons) oder durch Schlüsselworte (z.B. ENDWHILE) abzuschließen sind.

Im nächsten Schritt sind die Anweisungen zu dekomponieren. Die entsprechende Syntaxregel stellt eine Reihe spezieller Anweisungen zur Wahl (Ergibtanweisung, bedingte Anweisung, WHILE-Anweisung u.a.m.). Der Parser muss solange probieren, bis er eine Dekomponierung gefunden hat, die mit dem Quelltext vereinbar ist. Wenn ihm das gelungen ist, substituiert er die betreffende metasprachliche Variable durch die rechte Seite (bzw. durch die passende Alternative der rechten Seite) der

betreffenden Regel. Wenn er auf ein `WHILE` stösst, weiß er, dass er die metasprachliche Variable *WHILE-Anweisung* wählen und durch die rechte Seite obiger Syntaxregel, also durch `WHILE` *ausdruck relop zahl* `DO` {*ergibtanweisung*} `ENDWHILE` substituieren muss. Die Analogie zum analytischen Rechnen (Suchen in einer Formelsammlung und nachfolgendes Substituieren) ist offensichtlich.

Der Parser fährt mit dem schrittweisen Dekomponieren der metasprachlichen Variablen fort, bis er die elementaren metasprachlichen Variablen erreicht hat, die sich nicht mehr dekomponieren, sondern nur noch instanzieren, d.h. durch Lexeme der Objektsprache substituieren lassen, beispielsweise *op* durch `ADD` oder *id* durch den betreffenden Bezeichner.

Häufig sind in einem Dekomponierungsschritt mehrere Regeln oder mehrere Alternativen in einer Regel anwendbar, die alle mit dem Quelltext vereinbar sind, sodass mehrere Dekomponierungswege offen stehen. Dann ist derjenige Weg zu wählen, der mit der grössten Wahrscheinlichkeit zum Quellprogramm führt. Wenn eine Bewertung der Wege nicht möglich ist, muss entweder gewürfelt oder nach einer geeigneten Vorschrift verfahren werden, z.B..nach der Vorschrift “Wähle die in der Regel zuerst genannte Alternative”. Wenn keine solche Vorschrift existiert, ist die Syntaxanalyse ein nichtdeterministischer Prozess, der nach einem *nichtdeterministischen Algorithmus* abläuft. In jedem Fall ist die Analyse mit Suchen verbunden und alle diesbezüglichen Überlegungen, die wir beim analytischen Rechnen [15.15] und beim Schlussfolgern angestellt haben, wären hier zu wiederholen. Erinnert sei an das *Backtracking* [15.17], das oft angewendet wird, wenn der Parser in eine Sackgasse geraten ist.

Während der syntaktischen Analyse entsteht Schritt für Schritt der *Syntaxbaum* des zu analysierenden sprachlichen Ausdrucks, z.B. eines Programms oder eines Satzes (siehe Bild 5.2). Dabei wächst der Baum “von oben nach unten”, beginnend mit dem Spitzensymbol. Dieses auf *De*komponierung beruhende Verfahren wird **Abwärts-** oder **Top-down-Analyse** genannt.

Der Parser kann auch in umgekehrter Richtung vorgehen und “unten”, genauer z.B. “unten links” mit der Analyse beginnen und schrittweise die metasprachlichen Variablen “komponieren”. Im ersten Schritt der Aufwärtsanalyse bestimmt der Parser das kürzeste Anfangsstück des Quelltextes (die kürzeste Anfangslexemfolge), das mit der rechten Seite einer Syntaxregel vereinbar ist und substituiert es durch die metasprachliche Variable, die auf der linken Seite der Regel steht. So fährt er schrittweise von links nach rechts und von unten nach oben fort, bis er das Spitzensymbol erreicht hat. Die Syntaxregeln dienen diesmal als Substitutionsregeln, in denen die Pfeile von rechts nach links zeigen. Diese Vorgehensweise heißt **Aufwärts-** oder **Bottom-up-Analyse**.

Wir wissen nun, welche Informationen der Parser vom Scanner erhalten muss. Mit denjenigen Lexemen, die nicht in den Syntaxregeln auftreten, kann er wenig anfangen. Er muss aber wissen, ob ein solches Lexem ein Identifikator oder ein Operator ist. Diejenigen Lexeme, die in den Syntaxregeln auftreten und zur syntak-

tischen Strukturierung der Lexemkette beitragen, müssen dem Parser explizit übergeben werden. Dazu gehören Trennzeichen (z.B. das Semikolon), Klammern und klammernde Schlüsselwörter (z.B. `WHILE...ENDWHILE`), mit deren Hilfe der Programmierer das Quellprogramm syntaktisch strukturieren kann. Je konsequenter er davon Gebrauch macht bzw. die Sprachdefinition ihn dazu zwingt, umso einfacher ist die syntaktische Analyse.

Wer sich an Kap.15.4 erinnert, wird bemerkt haben, dass die Tätigkeit des Scanners identisch ist mit der Tätigkeit des Assemblers in der ersten Phase des Assemblierens [15.3]. Die Vorgehensweise des Assemblers kann also übernommen werden, das Anlegen einer *Symboltabelle* eingeschlossen. Für jeden Bezeichner legt der Scanner eine Zeile in der Symboltabelle an, wo er selber und später der Parser und der Codegenerator alle relevanten Attribute eintragen, z.B. den Gültigkeitsbereich des Bezeichners innerhalb des Programms, den Typ der bezeichneten Variablen (z.B. `real` oder `integer`), den Speicherplatzbedarf und die Adresse des Speicherplatzes, an den die Variable gebunden ist.

Nachdem der Scanner ein Lexem analysiert hat, übergibt er dem Parser entweder den Namen der Klasse, zu der es gehört, oder, falls es ein Lexem ist, das in den Syntaxregeln auftritt, das Lexem selber. Die Symbole, die er übergibt, also Zeichenketten wie WHILE oder id, werden unter der Bezeichnung **Token** zusammengefasst, und die Tokenfolge, die vom Scanner zum Parser "fließt", wird **Tokenstrom** genannt. Während der lexikalen Analyse der Zeilen 3 und 4 des Programms von Bild 15.6 könnte der Scanner den Tokenstrom

id := zahl ; `WHILE` op (id) rel zahl `DO` (16.3)

generieren.

Angenommen, der Parser arbeitet nach dem Aufwärtsverfahren. Dann sind in der Regel mehrere Token erforderlich, um die nächst höhere syntaktische Einheit gemäß Komponierungsregeln bilden und ihre metasprachliche Klasse bestimmen zu können. Zum Treffen der richtigen Entscheidung benötigt der Parser eventuell viele Token, sodass er weit voraus- und zurückschauen, d.h. viel *Kontext* berücksichtigen muss, m.a.W. auch er muss sich, wie der Mensch, einen gewissen, wenn auch im Vergleich zum Menschen sehr begrenzten Überblick verschaffen. Beim Abwärtsverfahren gilt Entsprechendes.

Der Aufwand, den der Parser treiben muss, insbesondere die Zeitdauer, die er für die Analyse braucht, hängt wesentlich davon ab, wie viel Kontext in den einzelnen Analyseschritten berücksichtigt werden muss. Das aber hängt davon ab, wie *übersetzerfreundlich* die Lexik und die Syntax der Sprache festgelegt sind. Von der Sprachdefinition hängt andererseits die *Nutzerfreundlichkeit* der Sprache ab. Die Forderungen hinsichtlich Übersetzerfreundlichkeit und Nutzerfreundlichkeit können sich widersprechen, sodass beim Sprachentwurf ein Kompromiss zwischen ihnen gefunden werden muss.

Mit der syntaktischen Analyse ist der *Analyseteil* des Übersetzens abgeschlossen und es beginnt der *Syntheseteil*, die Codegenerierung. Sie erfolgt nach dem gleichen Prinzip, das auch Max anwandte, als er einen AcI durch einen Dass-Satz aufgrund eines Wörterbucheintrags substituierte. Auch der Codegenerator muss über ein Wörterbuch verfügen, d.h. eine Liste von Regeln, welche die Rolle von Substitutionsregeln spielen. Auf der linken Seite einer Regel steht ein syntaktisches Konstrukt der Quellsprache, auf der rechten eine Tokenfolge der Zielsprache, beispielsweise für die Generierung eines Inkrementierungsbefehls (vgl. die Zeile 7 der Programme von Bild 15.2)

$$\text{id1} := \text{id1+1} \;\rightarrow\; \texttt{INC}\ \text{id1}. \tag{16.4}$$

Der Leser wird erkennen, dass die rechte Seite keine Lexemfolge, sondern eine Tokenfolge darstellt. Die Richtigkeit derartiger Regeln ist aus den Syntaxdefinitionen der höheren Programmiersprache (Quellsprache) und der Assemblersprache (Zielsprache) formal zu beweisen.

Die prinzipielle Vorgehensweise des Codegenerators ist uns inzwischen von anderen Substitutionsprozessen her gut bekannt. Er substituiert schrittweise Stücke des Quellcodes durch Stücke des Zielcodes gemäß den Substitutionsregeln und ersetzt die syntaktischen Bezeichnervariablen (z.B. *id1*) durch die betreffenden Bezeichner. Damit ist der Zielcode erstellt.

Man beachte, dass die Vorgehensweisen von Scanner, Parser und Codegenerator an keine bestimmte Zielsprache gebunden sind, dass sie also nicht auf die Assemblersprache beschränkt sind (man lasse sich nicht durch das Beispiel (16.4) irritieren).

Wenn ein *ladbares* Programm erzeugt werden soll, das in den Hauptspeicher geladen werden kann, um ausgeführt zu werden, ist die Zielsprache eine binär codierte Maschinensprache, und an die Übersetzung muss sich das Assemblieren und das Binden anschließen [15.4]. Im ersten Schritt des Bindens werden relative Adressen eines oder mehrerer gerufener Prozeduren an relative Adressen der rufenden Prozedur gebunden und dadurch die verschiedenen Bausteinprozeduren zu einer Kompositprozedur verbunden. Dieses “Verbinden” durch den Binder wird vom Codegenerator vorbereitet. Wenn eine Prozedur eine andere ruft, muss bei der Abarbeitung der Prozessor die rufende Prozedur unterbrechen und die gerufene Prozedur “anspringen”. Dazu ist ein Sprungbefehl zur Anfangsadresse der gerufenen Prozedur erforderlich. Diesen baut der Codegenerator in den Zielcode ein, wenn der Scanner einen Prozedurruf erkannt hat. Der Befehl bewirkt eine **Unterbrechung** der Abarbeitung der rufenden Prozedur. In Kap.19.5 werden wir eine andere Methode des Verbindens von Prozeduren (ladbaren Programmen) kennen lernen, wobei zunächst ein Systemprogramm gerufen wird, welches das Verbinden einleitet.

Unterbrechungen sind auch in anderen Situationen notwendig, so bei Fehlermeldungen oder wenn Daten von oder zur Peripherie transportiert werden sollen, beispielsweise wenn Daten vom Plattenspeicher benötigt werden. Für die Bearbei-

tung von Unterbrechungen ist i.Allg. ein spezieller Aktionsschritt der zentralen 14
Steuerschleife [13.10] vorgesehen.

Hiermit wollen wir die Überlegungen zur Arbeitsweise eines Übersetzerprogramms abschließen. Wir haben uns nur für Grundprinzipien interessiert. Sehr viele, auch wichtige Einzelheiten sind nicht zur Sprache gekommen. Darum sollen einige Ergänzungen in mehr oder weniger zufälliger Reihenfolge angefügt werden[17].

In Fachbüchern zur Übersetzerprogrammtechnik findet man nach der lexikalen und syntaktischen Analyse in der Regel Ausführungen zur sogenannten **semantischen Analyse**. In der Sprechweise dieses Buches könnte man für "semantische Analyse" auch "*internsemantische Kontrolle*" sagen. Es handelt sich nämlich im Wesentlichen um die Überprüfung der Anbindbarkeit programmiersprachlicher Ausdrücke an die Hardware. Beispielweise wird geprüft, ob ein syntaktisch richtiger Ausdruck wie z.B. `a+b` vom Mikroprozessor auch tatsächlich ausgeführt werden kann. Angenommen, durch das Pluszeichen wird ein Mikroprogramm aufgerufen, das entweder zwei Integerzahlen *oder* zwei Realzahlen addiert. Wenn nun `a` eine Intergerzahl und `b` eine Realzahl ist, kann der Prozessor die Addition nicht ausführen und der Ausdruck wird als falsch zurückgewiesen. Um das zu vermeiden, wird er beim Übersetzen "semantisch" überprüft und nötigenfalls eine sog. **Typanpassung**, in diesem Fall die Überführung einer Integerzahl in eine Realzahl vorgenommen.

Bei der Generierung des Maschinencodes wird oft zunächst ein **Zwischencode** erzeugt, meistens in Form des Maschinencodes einer (fiktiven) Drei-Adress-Maschine. Häufig wird der Zwischencode hinsichtlich des für die Abarbeitung erforderlichen Aufwandes *optimiert*, insbesondere hinsichtlich der Ausführungsdauer.

Ein Übersetzerprogramm, das vollständige Programme geschlossen übersetzt, heißt **Compiler**. Erfolgt die Übersetzung und Ausführung stückweise (in einzelnen übersetz- und ausführbaren Stücken) spricht man von **Interpretieren**. Ein interpretierendes Übersetzerprogramm heißt **Interpreter**. Das Wort "Interpretieren" hat den gleichen Sinn wie in Bild 2.1, nur tritt an die Stelle der *externen* Semantik (der Nutzersemantik, des Idems) die *interne* Semantik (Computersemantik). Die einzelnen Stücke des Quellprogramms, die der Interpreter interpretiert, müssen internsemantisch abgeschlossen sein, d.h. sie müssen eine *vollständige* Beschreibung dessen sein, was im Computer bei der Abarbeitung des Programmstücks vor sich gehen soll. Interpreter kommen vorwiegend in **interaktiven** Systemen, z.B. in **Dialogsystemen** zur Anwendung. In einem *interaktiven* System wechseln sich die Aktivitäten von Computer und Nutzer ab.

Abschließend sei noch einmal auf den enormen Vorteil hingewiesen, den der Mensch dem Computer gegenüber bei der (i.Allg. unbewussten) Analyse und Synthese sprachlicher Ausdrücke besitzt. Dem Menschen macht das Dekomponieren bzw. Komponieren syntaktischer Einheiten kaum Schwierigkeiten, weil er in der

17 Näheres siehe z.B. in [Aho 92].

Lage ist, Sätze (z.B. den Satz in Bild 5.2) "*in Gänze*" zu erfassen. Bei Benutzung der Muttersprache entfällt eine bewusste Syntaxanalyse von Sätzen, zumindest solange es sich nicht um Sätze mit sehr kompliziertem Satzbau handelt. Dem Computer fehlt die Übersicht. Er kann nur Zeichen für Zeichen lesen. Das ist der Grund für den großen programmtechnischen Aufwand des maschinellen Übersetzens. Welchen Aufwand das Gehirn beim Übersetzen treibt, wissen wir nicht.

16.5* Theorie formaler Sprachen

Der Titel von Kap.16.4 könnte die Erwartung geweckt haben, dass vom Übersetzen zwischen natürlichen Sprachen die Rede sein wird, einem sehr aktuellen Gebiet der Informatikforschung. Dass diese Erwartung nicht erfüllt worden ist, hat zwei triftige Gründe, einen negativen und einen positiven. Der negative Grund wird in Kap.17 klar werden, wenn wir uns überlegen, woran das Vorhaben scheitert, dem Computer die Rolle eines ernsthaften oder witzigen Gesprächspartners zu übertragen. Es scheitert am technischen Semantikproblem, dem Problem des Anbindens der *Humansemantik* (*Nutzersemantik*) an die *Maschinensemantik*. Hier liegt die Hauptschwierigkeit des qualifizierten Übersetzens aus einer natürlichen Sprache in eine andere, etwa das Übersetzen eines Gedichtes von Goethe ins Chinesische.

Der positive Grund besteht in der universellen Anwendbarkeit generativer Grammatiken. Sieht man vom technischen Semantikproblem ab, besteht der Kern jedes Übersetzens in der Wörterbucharbeit der lexikalen Analyse und der Codegenerierung und in der analytischen Arbeit des Erkennens der syntaktischen Struktur. Nun ist unschwer einzusehen, dass die Methode der generativen Sprachdefinition bzw. Sprachbeschreibung, die wir bisher nur auf Programmiersprachen angewendet haben, ohne grundsätzliche Veränderungen auch auf natürliche Sprachen anwendbar ist. Das ist nicht verwunderlich angesichts der Ähnlichkeit des syntaktischen Aufbaus natürlichsprachlicher und programmiersprachlicher Konstrukte. Diese Ähnlichkeit liegt der didaktischen Methode zugrunde, die stets zur Anwendung kam, wenn es sich anbot, und die darin besteht, syntaktische Sachverhalte zunächst an natürlichen Sprachen zu demonstrieren und dann auf Programmiersprachen zu übertragen. Man erinnere sich an die Gegenüberstellung der syntaktischen Struktur von Aussagesätzen (5.1) und Befehlen (5.2) in Kap.5.3., oder an die Begriffe der metasprachlichen Variablen und des Syntaxbaumes, die in Kap.5.3 parallel für natürliche Sprachen und für Programmiersprachen eingeführt wurden, und auch an Max, wie er versucht, einen englischen Satz zu übersetzen.

Tatsächlich haben die Informatiker bei der Entwicklung von Übersetzern auf die Ergebnisse der Linguisten zurückgegriffen. Die Idee der generativen Grammatik entstammt nicht der technischen Informatik, sondern der Sprachwissenschaft. Doch erst in der Hand der theoretischen Informatiker entstand eine ausformulierte Theorie, auf die wir einen kurzen Blick werfen wollen. Das ist insofern lehrreich, als an diesem

Beispiel Gemeinsamkeiten und Unterschiede zwischen informatischen Theorien und Theorien der exakten Naturwissenschaften sichtbar werden.

Die linguistischen Begriffe, die wir bisher verwendet haben, waren inhaltlich (d.h. über externe Semantik) eingeführt worden. Sie sind semantisch geladen, z.B. der Satzbegriff mit der Vorstellung konkreter Sätze und deren Aufbau. Jetzt abstrahieren wir von derartiger Semantik und definieren ganz formal:

Eine **Grammatik** ist ein 4-Tupel $G = (V,\Sigma,P,s)$. V und Σ sind zwei endliche Mengen ohne gemeinsame Elemente. V ist die Menge der sog. **Nichtterminalsymbole** und Σ ist die Menge der **Terminalsymbole**. P ist eine endliche Menge von *Regeln*, den sog. **Produktionsregeln**. Die Elemente aus P sind Substitutionsregeln der Form $u \rightarrow v$, worin u und v Ketten von Terminal- und Nichtterminalsymbolen sein können. Das Zeichen s heißt **Startsymbol**; es ist Element aus V. Die Menge aller Terminalsymbolketten, die sich aus s durch Substitutionsfolgen erzeugen lassen, ist die von G erzeugte (definierte, beschriebene) Sprache $L(G)$.

Es ist unschwer zu erkennen, dass sich die Beschreibungs- bzw. Analysemethode konkreter sprachlicher Ausdrücke mit Hilfe semantisch belegter metasprachlicher Begriffe als Interpretation des eingeführten Formalismus auffassen lässt. Die Elemente von V sind Zeichen (Buchstaben, Symbole) der *Metasprache*, in der über die *Objektsprache* gesprochen wird; darum heißen sie *Nichtterminal*symbole. Die Elemente von Σ sind Zeichen (Buchstaben, Symbole) der Objektsprache, die beschrieben (analysiert) wird; darum heißen sie *Terminal*symbole.

Der interessierte Leser kann sich den Inhalt der Grammatikdefinition an folgenden Beipielgrammatiken verdeutlichen, die beide [Schöning 95] entnommen sind, wo sie näher erläutert werden[18]. Der Pfeil in den Produktionsregeln (in den Elementen von P) entspricht dem Pfeil in Syntaxregeln, z.B. in (5.2).

Beispiel 1

$G_1 = (\{E,T,F\}, \{(,),a,+,*\}, P, E)$, wobei:
$P = \{E \rightarrow T,\ E \rightarrow E+T,\ T \rightarrow F,\ T \rightarrow T*F,\ F \rightarrow a,\ F \rightarrow (E)\}$

Mit dieser Grammatik lassen sich die korrekt geklammerten arithmetischen Ausdrücke darstellen. Beispielsweise ist die Zeichenkette $a+a*a*(a+a)$ ein Satz (“Wort”) der Sprache $L(G_1)$. Anstelle von a könnten wir auch *id* schreiben. Es steht für beliebige arithmetische Variablen. Mit den Symbolen *E, T, F können* die Begriffe *Expression* (Ausdruck), *Term* und *Faktor* (im Sinne der üblichen Definition formalisierter Sprachen) assoziiert werden, d.h. ihnen *kann* metasprachliche Semantik zugeordnet werden. Dadurch wird die Grammatik “verständlicher”, sie bekommt

18 Wir behalten die in [Schöning 95] verwendete Notationsweise bei, d.h. die Buchstaben aus V werden groß, die aus Σ werden klein geschrieben.

Sinn. Doch sind diese Assoziationen aus der Sicht der abstrakten Definition der Grammatik überflüssig. Die Situation ähnelt der in Kap.16.1, als wir uns das abstrakte Inferenzieren durch konkretes Inferenzieren verständlicher machten.

Beispiel 2

$G_2 = (V,\Sigma,P,\mathrm{s})$, wobei:
$V = \{\mathrm{s},B,C\}$
$\Sigma = \{a,b,c\}$
$P = \{\mathrm{s}\rightarrow asBC,\ \mathrm{s}\rightarrow aBC,\ CB\rightarrow BC,\ aB\rightarrow ab,\ bB\rightarrow bb,\ bC\rightarrow bc,\ cC\rightarrow cc\}$
Durch diese Grammatik wird z.B. das Wort $aaabbbccc = a^3b^3c^3$ generiert. Allgemein besitzen die Wörter der Sprache $L(G_2)$ die Form $a^nb^nc^n$ mit $n\geq 1$. Die Regeln sind - ebenso wie beim analytischen Rechnen und beim Schlussfolgern - Substitutionsregeln, und die dargestellte Formalisierung der Syntaxanalyse stellt eine Kalkülisierung dar, sodass einer Implementierung nichts im Wege steht.

Vergleicht man die Substitutionsregeln der beiden Grammatiken miteinander, stellt man fest, dass alle Regeln von G_1 auf der linken Seite ausschließlich Nichtterminalsymbole enthalten; jede Regel substituiert genau ein Nichtterminalsymbol. Demgegenüber können die linken Seiten einiger Regeln von G_2 mehrere Nichtterminalsymbole und eventuell zusätzlich ein Terminalsymbol enthalten. Das bedeutet in unserer früheren, semantisch beladenen Sprechweise ausgedrückt, dass zur Analyse ein mehr oder weniger umfangreicher Kontext erforderlich ist. Dementsprechend wird die Grammatik G_2 und die durch sie definierte Sprache **kontextabhängig** oder **kontextsensitiv** genannt, während G_1 **kontextfrei** genannt wird. Die Arbeit mit G_2 demonstriert unsere frühere Einsicht - diesmal rein formal -, dass Kontextabhängigkeit die Analyse erschwert. Die schrittweise Generierung von Terminalketten entspricht der Top-down-Analyse.

Man kann sich andere Grammatiken mit anderen Substitutionsregeln ausdenken, auch mit anderen Vorschriften für den Aufbau der Substitutionsregeln, also mit anderen *Metaregeln*. Um die Analyse (dem Parser die Arbeit) zu erleichtern, könnte man z.B. neben der Kontextfreiheit verlangen, dass die rechte Seite einer Regel mit einem oder mehreren Terminalsymbolen beginnt, gefolgt von einem oder mehreren Nichtterminalsymbolen. Eine solche Grammatik heißt **regulär**.

Im Sinne dieser Überlegungen hat der amerikanische Sprachwissenschaftler NOAM CHOMSKY in den 50er Jahren seine generative Transformationsgrammatik entwickelt und auf dieser Grundlage eine Hierarchie von Sprachen aufgebaut, wobei eine in der Hierarchie höhere Sprache aus der darunter liegenden durch *Präzisierung* hervorgeht, d.h. durch Hinzunahme zusätzlicher Forderungen an den Aufbau der Substitutionsregeln.

Chomsky hatte sich das Ziel gestellt, den Aufbau natürlichsprachiger Ausdrücke mit Hilfe der mathematischen Logik zu beschreiben. Der weltweite Erfolg seiner Ergebnisse bezog sich jedoch weniger auf natürliche als vielmehr auf Programmier-

sprachen, auf deren Definition und Analyse. Die heute gängigen Programmiersprachen sind fast durchweg kontextfreie, oft sogar reguläre Sprachen.

Es ist das natürliche Bestreben der Theoretiker, jede neue formale Beschreibung irgendeines Tatbestandes daraufhin zu untersuchen, ob sich auf sie vielleicht die eine oder andere bereits erprobte mathematische Methode anwenden lässt, um den neuen Formalismus besser handhaben zu können. Der Formalismus von Chomsky machte da keine Ausnahme. In diesem Fall war es die Automatentheorie, die der generativen Transformationsgrammatik ein solides mathematisches Fundament bescherte und damit die Möglichkeit, den Formalismus mit einer Menge von Theoremen zu untermauern und in Richtung Axiomatisierung weiterzuentwickeln.

Auf diese Weise entstand das Gebäude der heutigen **Theorie formaler Sprachen**. Ohne auf Einzelheiten einzugehen, sollen einige Ideen und Begriffe der Theorie erläutert werden. Der Begriff des Automaten, genauer des *abstrakten endlichen Automaten* im Sinne der Automatentheorie, ist in Kap.8.3 eingeführt worden. Wir erinnern uns, dass ein Automat ein sprachlicher Operator mit Gedächtnis ist. Das Gedächtnis dient der Aufbewahrung des inneren Zustandes, den der Automat selber berechnet. Der neue innere Zustand *z'* ist eine Funktion *f*(*x*,*z*) - Folgefunktion genannt - des momentanen Eingabezeichens *x* und des alten Zustandes *z* (vgl. (8.4)). Einer der Zustände ist als *Anfangszustand* ausgezeichnet.

Wenn man die Begriffe Automat und Sprache in ihren ursprünglichen Bedeutungen einander gegenüberstellt, ist kaum zu verstehen, wie sich zwischen ihnen eine Brücke schlagen lässt. Was hat z.B. die Arbeitsweise eines Fahrkartenautomaten mit den Regeln der deutschen Sprache gemeinsam? Zwar haben die Definitionen des endlichen Automaten und der formalen Sprachen bereits zu einer erheblichen Annäherung auf abstrakter Ebene geführt, doch sind weitere Schritte erforderlich.

Der erste Schritt besteht darin, dass wir einige innere Zustände als *Endzustände* markieren. Wenn ein Endzustand erreicht wird, hält der Automat an, in Analogie zur Turingmaschine und zum Markovalgorithmus. Wenn dem Automaten irgendeine Zeichenkette eingegeben wird, geht er *möglicherweise* in einen Endzustand über. Die theorienverbindende Idee ist nun folgende. Die Gesamtheit aller Eingabeketten, die den Automaten aus dem Anfangszustand in einen Endzustand überführen, wird als Sprache aufgefasst (im Sinne einer extensionalen Definition der Sprache als Menge ihrer Wörter).

Man kann nun umgekehrt eine konkrete Sprache vorgeben und einen konkreten Automaten daraufhin untersuchen, ob er die Sprache *akzeptiert*, d.h. ob er nach Eingabe eines beliebigen Wortes der Sprache in einen Endzustand übergeht. Ist das der Fall, wird der Automat **akzeptierender Automat** der betreffenden Sprache genannt. Damit ist die gesuchte Brücke zwischen Automaten- und Sprachtheorie gefunden. Es zeigt sich jedoch, dass der Automatenbegriff einiger Erweiterungen und Präzisierungen bedarf, um effektiv und umfassend bei der Sprachanalyse eingesetzt werden zu können. Sie sind der Inhalt des zweiten Schrittes.

Der zweite Schritt in Richtung Vereinigung (*Konvergenz*) beider Theorien zielt auf die automatentheoretische Erfassung des evtl. nichtdeterministischen Charakters der Syntaxanalyse. Zu diesem Zweck ist der Begriff des **nichtdeterministischen Automaten** erforderlich. Für ihn ist die Folgefunktion $f(x,z)$ keine eindeutige, sondern eine mehrdeutige Abbildung. Im Falle des *Turingautomaten* (der Turingmaschine) ist das gleichbedeutend mit einer Mehrdeutigkeit der Automatentabelle.

Durch diese Erweiterung verliert die Theorie nicht an Wert, genauso wie die Methoden des Rechnens und Schlussfolgerns nicht durch den inhärenten Indeterminismus an Wert verlieren, der sich stets durch zusätzliche Vorschriften überwinden lässt, beispielsweise durch eine deterministische Suchvorschrift. Ebenso lässt sich ein nichtdeterministischer Automat in einen deterministischen überführen. Auch der Indeterminismus der syntaktischen Analyse muss beim Implementieren durch geeignete Vorschriften ausgeräumt werden (wenn kein Zufallszahlengenerator zum Einsatz kommen soll). Dabei spielen ähnliche Optimierungsüberlegungen eine Rolle wie beim analytischen Rechnen nach einem nichtdeterministischen Algorithmus.

Ein anderer wichtiger Begriff der Theorie ist der des Kellerautomaten. Er wurde, ebenso wie der des Kellerspeichers, bereits in Kap.8.4.6 eingeführt. Einen Automaten, dessen Gedächtnis als Kellerspeicher organisiert ist, hatten wir Kellerautomaten genannt (vgl. die Bilder 8.12 und 13.1c). Um die Bedeutung des Kellerautomaten für die Theorie formaler Sprachen zu verstehen, verfolgen wir die Arbeit eines Computers bei der Berechnung des Wertes des Ausdrucks 3+2∗(1+(5-4)). Dieser Ausdruck ist nach den Regeln der kontextfreien Grammatik G_1 aufgebaut. Bei seiner Berechnung muss der Computer nach Erkennen des Multiplikationszeichens die begonnene Addition unterbrechen und zunächst die Multiplikation ausführen, da sie vor der Addition Vorrang hat. Dazu muss er den bereits gelesenen Teil des Additionsbefehls speichern und mit der Berechnung des zweiten Summanden beginnen. Nach Erkennen der ersten öffnenden Klammer muss er wiederum unterbrechen, das inzwischen Gelesene speichern und mit der zweiten Addition beginnen. Beim Erkennen der zweiten öffnenden Klammer wiederholt sich das Spiel zum dritten Mal. Danach kann er vom Ende her mit dem Rechnen beginnen: 5-4=1; 1+1=2; 2∗2=4; 3+4=7.

Dieses Vorgehen, das mit wiederholtem Unterbrechen und Speichern beginnt, bevor die eigentliche Rechnung ausgeführt werden kann, erinnert an die rekursive Berechnung von 5! in Kap.8.4.6. Ein Blick auf Bild 8.11 legt eine wichtige Idee des Computerentwurfs nahe. Beim Berechnen geklammerter Ausdrücke ist die Verwendung eines Kellerspeichers zweckmäßig, denn dann erübrigt sich das aufwendige Speichern und Lesen über Adressen. Der Kellerspeicher liefert bei der abschließenden Berechnung (durch op_2 in Bild 8.11) die Daten in derjenigen Reihenfolge, in der sie benötigt werden. Dieser Mechanismus war bereits in Kap.8.4.6 bei der Berechnung der Fakultät besprochen worden.

Nach dieser Überlegung wird folgendes Resultat der Theorie nicht überraschen: *Kontextfreie Sprachen werden von Kellerautomaten akzeptiert*. Das ist verständlich,

denn ein Kellerspeicher ist in der Lage, das erforderliche Unterbrechen und Speichern zu erledigen, obwohl er nicht über den Adressiermechanismus verfügt.

Damit sind die Einsatzmöglichkeiten des Kellerprinzips nicht erschöpft. Die Vorteile, die es bei der Ausführung eines konsequent *funktional* formulierten Programms bringt, sind offensichtlich, denn dieses stellt einen einzigen geschachtelten Klammerausdruck dar. Aber auch bei der Ausführung imperativer Programme ist es ein wertvolles Hilfsmittel, denn auch hier gibt es das "Klammern" im weiten Sinne des Wortes (ganz abgesehen von geklammerten mathematischen Ausdrücken, die von vielen imperativen Sprachen zugelassen sind). Beispielsweise stellt das Schlüsselwortpaar `WHILE - ENDWHILE` in Bild 15.3b ein Klammerpaar dar, das die Befehle "einklammert", die bei jedem Iterationsschritt der Berechung der Sinusfunktion gemäß (15.5) auszuführen sind. 16

Ein weiteres Beispiel für den Einsatz des Kellerspeichers ist der *Unterprogrammruf* (*Prozedurruf*). In Kap.15.7 [15.10] hatten wir uns überlegt, dass sich das Komponieren von Kompositoperatoren dadurch programmtechnisch realisieren lässt, dass für die Bausteinoperationen Unterprogramme (Prozeduren) geschrieben werden, die vom Hauptprogramm (dem Programm der Kompositoperation) aufgerufen werden. Wenn bei der Abarbeitung des Hauptprogramms ein Unterprogrammruf erkannt wird, muss das Hauptprogramm unterbrochen und alles abgespeichert werden, was zu seiner späteren Fortführung erforderlich ist. Bei tiefer Schachtelung der Unterprogrammrufe (bei hohem Komponierungsgrad) erspart ein Kellerspeicher viel organisatorische Arbeit, denn er liefert nach Beendigung eines Unterprogramms durch Entkellerung stets die gerade erforderlichen Daten zur Fortführung des jeweils aktuellen Programms.

Der Vollständigkeit halber sei noch einmal darauf hingewiesen, dass ein konsequent funktional geschriebenes Programm eine Operatorenhierarchie nicht durch Unterprogramm-Schachtelung, sondern mittels gewöhnlicher Klammerung realisiert. Eben darin besteht die Kernidee des funktionalen Programmierens. Ein geklammerter Ausdruck stellt für den übergeordneten Klammerausdruck denjenigen Wert dar, der sich bei der Berechnung des untergeordneten Klammerausdrucks ergibt.

Betrachtet man nun noch einmal die Tätigkeit eines Computers, der ein Programm zuerst übersetzt und dann ausführt, erkennt man die Rolle des Kellerns in beiden Tätigkeiten. Beim Übersetzen eines Ausdrucks, der Klammern (im weiten Sinne) enthält, muss der Parser die Analyse auf der aktuellen Komponierungsebene, z.B. des Hauptprogramms (im natürlichsprachlichen Fall des Hauptsatzes) unterbrechen, die notwendigen Daten kellern und mit der Analyse des geklammerten Ausdrucks, z.B. des gerufenen Unterprogramms (eines Nebensatzes oder Satzteiles) beginnen. An denjenigen Stellen, an denen der Parser beim Erkennen eines Unterprogrammrufs die Analyse des Hauptprogramms unterbricht, muss der Prozessor die Abarbeitung des übersetzten Hauptprogramms unterbrechen. Er realisiert den Unterprogrammruf dadurch, dass er in den Befehlszähler nicht die Adresse des nächsten Maschinenbefehls, also nicht die inkrementierte alte Adresse, sondern die Anfangsadresse des

Unterprogramms einträgt. In Bild 13.7 entspricht dem die Übergabe der neuen Adresse aus dem Befehlsregister (BR) an den Befehlszähler (BZ). Um später bei der Absprungadresse (bei dem Befehl, an dem das Hauptprogramm verlassen wurde) fortfahren zu können, müssen alle erforderlichen Daten gespeichert, d.h. zweckmäßigerweise gekellert werden.

Schließlich soll noch eine für die KI besonders charakteristische Anwendung des Kellerspeichers erwähnt werden, das Suchen in einem Baum (z.B. in der Form des Backtracking), von dem wiederholt im Zusammenhang mit indeterministischen Verfahren die Rede war (siehe z.B. Kap.15.8). Beim Abstieg im Suchbaum muss der momentane "Zustand" des Suchprozesses in jedem Knoten (an jeder "Wegegabel") abgespeichert werden, um gegebenenfalls später die Suche an diesem Punkt, aber in anderer Richtung wieder aufnehmen zu können. Offensichtlich ist auch hier das Speichern nach dem Kellerprinzip zweckmäßig.

Die vielseitige Verwendung des Kellerprinzips hat uns etwas vom Thema abgelenkt. Wir kehren noch einmal zur Theorie formaler Sprachen zurück und nennen einige ihrer Ergebnisse. In Bild 16.5 ist zusammengefasst, welche Sprachtypen durch welche Automaten akzeptiert werden. In der zweiten Spalte ist die Bezeichnung der generierenden Grammatik bzw. des generierten Sprachtyps und in der dritten Spalte der akzeptierende Automatentyp angegeben. Die vier Sprachtypen bilden eine

Typ	Grammatik,Sprache	akzeptierender Automat
Typ 3	regulär	endlicher Automat
Typ 2	kontextfrei	Kellerautomat
Typ 1	kontextsensitiv	linear beschränkter Turingautomat
Typ 0	ohne Beschränkung	Turingautomat

Bild 16.5 Chomsky-Hierarchie. Spalte 2 - Eigenschaft der generierenden Grammatik; Spalte 3 - Klasse des akzeptierenden Automaten.

Hierarchie, die sog. **Chomsky-Hierarchie**. Jeder Hierarchieebene entspricht ein Sprachtyp. Dieser wird aufsteigend nummeriert (erste Spalte). Sprachen höheren Typs sind in Sprachen niederen Typs enthalten.

Eine Sprache, deren Grammatik den Produktionsregeln keinerlei Beschränkungen auferlegt, heißt vom Typ 0. Sprachen vom Typ 0 werden von Turingautomaten akzeptiert. Kontextsensitive Sprachen (Typ-1-Sprachen) werden von *linear beschränkten* Turingautomaten akzeptiert, das sind solche, die denjenigen Teil des Bandes, auf dem die Eingabe steht, niemals verlassen. Man erkennt, dass ein Aufsteigen von niederen zu höheren Sprachtypen einem Präzisieren entspricht. Das Präzisieren ist nach Bild 5.4 ein begriffsbildende Operation, die den Abstraktionsgrad herabsetzt. Im Gegensatz zu Bild 5.4 entspricht dem Abstrahieren in Bild 16.5 also die Abwärtsrichtung. Weitere Einzelheiten zur Theorie formaler Sprachen können

z.B. in [Schöning 95] nachgelesen werden. Wir wollen es mit der Nennung eines Satzes genug sein lassen, der das sog. Wortproblem löst. Es geht um die Frage, ob man einer Zeichenkette ansehen kann, ob sie ein Satz (ein Wort) einer bestimmten Sprache ist oder nicht.

Die Sprachtheoretiker haben diese Frage das **Wortproblem** genannt und im Rahmen der Theorie formaler Sprachen beantwortet. Die Antwort lautet: Es gibt einen Algorithmus, der die Frage beantwortet, ob eine gegebene Zeichenkette ein Wort einer von einer gegebenen kontextsensitiven Grammatik definierten Sprache ist. Mit anderen Worten, das Wortproblem ist für Typ1-Sprachen (und damit auch für Typ2- und Typ3-Sprachen) entscheidbar. Für Typ0-Sprachen ist das Wortproblem nicht allgemein entscheidbar.

Das Wenige, was über die Theorie formaler Sprachen gesagt worden ist, sollte zum einen eine Vorstellung davon geben, mit welchen theoretischen Problemen sich die Theoretiker u.a. beschäftigen und aus welchen praktischen Aufgabenstellungen diese Probleme erwachsen sind. Zum anderen sollte ein weiteres Beispiel für die Wirkungsweise des wissenschaftlichen Konvergenzprinzips [8.38] [11.1] angeführt werden. Die Wirkungsweise des Prinzips besteht hier darin, dass eine neue Theorie dadurch entsteht (*erfunden* wird), dass Begriffe und Relationen aus ganz unterschiedlichen Bereichen auf höherer Abstraktionsebene "konvergieren", dass sie miteinander kompatibel oder sogar identisch werden (vgl. Kap.8.5 und 11.1). Im vorliegenden Fall handelte es sich um Begriffe und Relationen aus dem Bereich der Sprachen einerseits und dem der Automaten andrerseits. Im Falle der in Kap.11.1 erwähnten "Erfindung" von SHANNON hatte es sich um Begriffe und Relationen aus der Schaltungstechnik und der Aussagenlogik gehandelt. In beiden Fällen gehören die konvergierenden Bereiche der Informatik an, und die Abstraktionen erfolgten innerhalb des Denkgebäudes der Informatik.

Eine nicht weniger große Rolle spielt das Konvergenzprinzip im Bereich der Physik. Beispielsweise sind die maxwellschen Gleichungen das Ergebnis einer Konvergenz zwischen elektrodynamischen und optischen Begriffen und Relationen. Es besteht aber ein chartakteristischer Unterschied in der Wirkungsweise des Prinzips in der Physik einerseits und in der Informatik andrerseits. Die Naturwissenschaften modellieren *die Natur* sprachlich, während die Informatik *das sprachliche Modellieren* sprachlich modelliert. Insofern betrifft das Konvergenzprinzip einmal das Zusammenwachsen (das Identischmachen) unterschiedlicher Begriffe von Objektsprachen, das andere mal unterschiedlicher Begriffe von Metasprachen.

Danach ist zu erwarten, dass eine "*naturwissenschaftliche*" Theorie der Informatik, die gegenwärtig noch nicht existiert, die Konvergenz objektsprachlicher und metasprachlicher Begriffe und die Bildung entsprechender Oberbegriffe zur Voraussetzung haben wird. Das würde ein Abstraktionsniveau erfordern, das oberhalb des Abstraktionsniveaus sowohl gegenwärtiger naturwissenschaftlicher als auch informatischer Theorien liegt.

17 Unterhaltung

Zusammenfassung

Der Computer kann befähigt werden, an umgangssprachlicher Konversation teilzunehmen, zur Zeit allerdings nur unter Verzicht auf die "Beliebigkeit" menschlichen Unterhaltens (Plauderns). Es gibt zwei Wege, den Computer dialogfähig zu machen: semantische Spezialisierung und semantische Verarmung. *Semantische Spezialisierung* besteht in der Beschränkung auf ein enges Spezialgebiet, auf einen kleinen Diskursbereich, sodass die Definition eines speziellen Denkkalküls und damit die Kalkülisierung des Dialogs möglich wird.

Semantische Verarmung beruht auf der Beschränkung auf Nichtssagendes. Viele Alltagsunterhaltungen zeichnen sich hierdurch aus. Der Computer kann an ihnen teilnehmen, ohne dass seine "unnatürliche" Intelligenz auffällt. Wenn er über eine größere Menge von Floskeln verfügt und vielleicht sogar noch über Regeln zur Auswahl passender Floskeln als Reaktion auf das Gerede seines Gesprächspartners, dann hat er Aussicht den *Turingtest* zu bestehen.

Inhaltsvolle Gespräche setzen *rationale Konsensfindung,* d.h. Einigung auf ein Thema, auf einen bestimmten Diskursbereich und dessen spezifische Sprache voraus und beruhen oft auf *emotionaler Konsensfindung,* auf Einstimmen auf gefühlsmäßige Gemeinsamkeiten. Letztere ist dem Computer verschlossen, wenn man davon ausgeht, dass er weder Bewusstsein noch Emotionen besitzt.

Es ist immer möglich, einen Computer in einen bild- oder klanggebenden Übertragungskanal einzubinden und nicht nur zur Umcodierung, sondern auch zur Manipulation derjenigen Bilder und Klänge zu benutzen oder zu missbrauchen, die den Übertragungskanal passieren. Der Computer kann auch am Anfang der Übertragungskette stehen und Klänge und Bilder erzeugen. Er kann fiktive, nichtexistierende Welten hervorbringen und "*virtuelle Realitäten*" schaffen.

Das *technische Semantikproblem* spielt bei der Unterhaltung durch Bilder und Klänge kaum eine Rolle, denn es werden nicht die *Bedeutungen* von Bildern oder Klängen codiert, sondern die Bilder und Klänge *selber* und zwar durch einfache Diskretisierung ohne Anwendung intelligenter Verfahren wie Abstraktion, Klassifikation oder Standardisierung.

Ursache dafür, dass die Informationsgesellschaft kräftige Wurzeln schlägt, ist weniger die künstliche Intelligenz, als vielmehr das massenhafte Eindringen von Computern in unser Leben oder, noch treffender, die Unterwanderung unseres Daseins durch ein Heer von Prozessoren.

Im Computerschach tritt ein charakteristischer Unterschied zwischen natürlicher und künstlicher Intelligenz zutage. Der Mensch begegnet der Komplexität des Schachspiels durch komplexes *Sehen* und komplexes *Denken*. Der Computer begegnet ihr sehr erfolgreich durch hohe Rechengeschwindigkeit. Dabei wird er kaum

"klüger". Der Mensch kann seine Spielqualität durch Übernahme fremder und durch Sammeln eigener Erfahrungen steigern. Die Schacherfahrungen eines Menschen sind vorwiegend globaler Natur und betreffen ganze Klassen von Spielsituationen und deren Vor- und Nachteile. Auch Erkennen und Denken des Schachspielers sind in vieler Hinsicht global. Er erfasst eine Situation und ihre Vorzüge und Gefahren global ("auf einen Blick"). Der Computer dagegen begnügt sich im wesentlichen mit dem systematischen, aber phantasielosen Durchspielen aller möglichen Fortsetzungen einer Partie und dem Vergleich der eigenen Verluste mit denen des Gegners während der verschiedenen Zugfolgen.

Darin spiegelt sich ein grundsätzlicher Unterschied zwischen dem Denken (dem intern codierten, aber nicht extern artikulierten sprachlichen Modellieren) des Menschen und dem "Denken" des Computers wider. Der Mensch ist in der Lage, gleichzeitig viele Fakten und Zusammenhänge im Bewusstsein zu halten und in seinem Denken zu berücksichtigen. Er denkt global. Er erfasst die Dinge im Überblick und trifft Entscheidungen gewissermaßen von einem höheren Gesichtspunkt aus. Der Mensch denkt in komplexen Vorstellungen, in Bildern, in Gestalten. Er *gestaltet* eine Schachpartie, er *gestaltet* einen Prozess, er *gestaltet* sein Leben.

Ein Computer kann eine Schachstellung nur feldweise und ein Bild nur pixelweise, genauer nur computerwortweise betrachten, denn er denkt algorithmisch. Kurz: *Der Computer denkt in Computerwortfolgen, der Mensch kann anschaulich und in globalen Zusammenhängen denken.*

17.1 Der Computer als Gesprächspartner

Wir verlassen nun den sicheren Boden erprobter und anerkannter Methoden der künstlichen Intelligenz und wenden uns Problemen zu, für deren Lösung zwar auch ein umfangreiches Arsenal methodischer und programmtechnischer Hilfsmittel existiert, doch sind die Verfahren oft noch unzureichend theoretisch untermauert und praktisch wenig ausgereift. Die Überlegungen des Kapitels 17 können sich in weit geringerem Maße auf Autoritäten berufen als die bisherigen Überlegungen. Es haftet ihnen ein gewisser spekulativer Zug an. Doch wäre es anders nicht möglich, die Grenzen des gesicherten Wissens und der sicheren Methoden zu überschreiten. Wir müssen sie überschreiten, wenn wir den Computer befähigen wollen, sich *mit uns* zu unterhalten oder *uns zu unterhalten* im ganz alltäglichen Sinne des Wortes "Unterhaltung". Das Kapitel 17 lässt die Exaktheit und logische Folgerichtigkeit der Gedankengänge vorangehender Kapitel vermissen. Doch fällt es nicht nur in dieser inhaltlichen Hinsicht aus dem Rahmen. Das Wort "Unterhaltung" charakterisiert nicht nur den Gegenstand, über den gesprochen wird, sondern auch den Stil, der eher unterhaltsam als streng wissenschaftlich zu nennen ist.

Als Charakterisierung des Gegenstandes hat das Wort "Unterhaltung" zwei Bedeutungen, eine aktive und eine passive. Ein Mensch kann sich, selber aktiv, mit

einem anderen unterhalten; dann spricht man auch von *Konversation* zwischen zwei oder mehreren Gesprächspartnern. Oder ein Mensch kann sich, selber passiv, von einem anderen unterhalten lassen. Falls ein Mensch viele Zuhörer unterhält, z.B. im Fernsehen, werden häufig die englischen Wörter *Entertainer* und *Entertainment* benutzt.

Wenn die Rolle des Computers als Unterhaltungspartner (in beiden Bedeutungen) in einem Umfange behandelt werden soll, der dieser seiner Rolle in der Informationsgesellschaft entspricht, wären dafür mehrere Bücher erforderlich. Wir beschränken uns auf eine kurze Beantwortung der Frage, was Informatik mit Unterhaltung zu tun hat und welche prinzipiellen Probleme zu lösen sind, damit der Computer als Unterhaltungspartner fungieren kann. Wir beginnen mit der Frage, unter welchen Bedingungen und in welchem Maße ein Computer die Rolle eines Gesprächspartners übernehmen kann.

Im Jahre 1950 veröffentlichte der englische Wissenschaftler ALAN M.TURING, ein Pionier der Informatik, der Erfinder der Turingmaschine, einen Artikel unter dem Titel "Computing Machinery and Intelligence" [Turing 50]. Darin schlug er einen Test vor, der nach ihm **Turingtest** genannt wird. Der Test soll auf die Frage Antwort geben, ob bzw. wann es sinnvoll ist zu sagen, dass der Computer *denken* kann. Der Test läuft folgendermaßen ab.

Eine Versuchsperson ist über ein Kommunikationsgerät, z.B. über einen Fernschreiber, mit einem Partner verbunden, den er weder sehen noch hören kann. In einem "Gespräch" soll er feststellen, ob sein Partner ein Mensch oder ein Computer ist. Wenn er in einer Reihe unterschiedlicher Gespräche den Computer nicht als Computer erkennt, sondern annimmt, dass er mit einem Menschen spricht, ist es offenbar gerechtfertigt, zu sagen, dass der Computer denken kann.

Im Jahre 1966 veröffentlichte ein anderer Pionier der Informatik, der in den USA lebende Wissenschaftler J.WEIZENBAUM, ein Programm namens *Eliza* [Weizenbaum 66]. Dieses Programm befähigt einen Computer die Rolle eines Arztes zu spielen, der sich mit einem neuen Patienten über dessen Anliegen, Beschwerden und persönliche Besonderheiten unterhält, so wie es zu Beginn eines Diagnosegesprächs üblich ist. Der Computer spielt seine Rolle überraschend gut, zumindest auf den ersten Blick, sodass man geneigt sein könnte, zu sagen, dass der Computer denkt, da er offenbar in der Lage ist, den Turingtest zu bestehen.

Analysiert man ein Gespräch mit Eliza genauer, kann man den Eindruck gewinnen, dass Weizenbaum sich mit seinem Programm den Spaß gemacht hat, den Turingtest ad absurdum zu führen. Zunächst stellt man fest, dass der Diskursbereich und die Rolle des Computers so definiert sind, dass der Computer im wesentlichen Fragen stellt, wofür ein sehr begrenztes Wissen ausreicht. Freilich muss er seine Fragen "intelligent" stellen, d.h. vernünftig aus der Sicht des Patienten. Um das zu erreichen, wendet das Programm einen einfachen Trick an. Es formuliert die nächste Frage in der Regel durch Konkretisierung oder Verallgemeinerung der vorangehenden Antwort des Patienten.

Auf die Aussage des Patienten “Ich leide sehr an Kopfschmerzen” könnte der Computer beispielsweise fragen “Seit wann haben Sie die Beschwerden?” oder “Wann treten die Beschwerden vorzugsweise auf?” oder “Sind Ihnen ähnliche Fälle in Ihrer Verwandtschaft bekannt?” Für derartige Fragen lassen sich “Produktionsregeln” implementieren, nach denen sich Fragen aus vorherigen Aussagen produzieren lassen, ganz ähnlich wie im Falle der Produktion neuer Aussagen beim Schlussfolgern oder bei der Produktion von Wörtern oder Sätzen einer Sprache nach den Produktionsregeln einer generativen Grammatik.

Zum Produzieren der Fragen oder Aussagen des Arztes benutzt der Computer Wissen, das beispielsweise in Form von Tafeln (in diesem Fall Frage-Antwort-Tafeln) oder von Regeln und Fakten abgespeichert sein kann. Damit der Computer auf möglichst viele Aussagen des Patienten “vernünftig” reagieren kann, muss er sie in eine geeignete Standardform bringen, sodass die Regeln und Tafeln anwendbar werden. Andrerseits sollte er für seine eigenen Standardantworten mehrere Artikulationsmöglichkeiten bereithalten, damit der Patient keinen Verdacht schöpft. Auf diese Weise kann das “Denken und Sprechen” des Arztes kalkülisiert werden, wenn auch nur in sehr begrenztem Umfange.

Im Laufe des Gesprächs kann *Eliza* (d.h. das Programm, das so tut, als sei es ein Arzt) immer besser auf den Patienten eingehen und immer verblüffendere Fragen stellen, denn während des Gesprächs erweitert das Programm Eliza sein einprogrammiertes allgemeines und medizinisches Faktenwissen um spezifisches Wissen, das ihm vom Patienten mitgeteilt wird. Es baut gewissermaßen eine gemeinsame Wissensbasis auf und damit eine gemeinsame externe Semantik, die es so artikulieren kann, dass der Patient Elizas Artikulationen richtig versteht und sie durchaus für vernünftig hält.

Nach dieser Analyse mag es scheinen, das Programm Eliza sei allzu erkünstelt und seine verblüffenden Fähigkeiten seien wenig aussagekräftig hinsichtlich der Möglichkeiten der KI. Dennoch ist der Wert des Programms nicht zu unterschätzen, denn es macht deutlich, wo die entscheidenden Schwierigkeiten liegen, denen die künstliche Intelligenz nicht gewachsen ist, und welches die Tricks sind, mit deren Hilfe sie sich scheinbar überwinden lassen, aber eben nur scheinbar. Bei diesen Tricks handelt es sich um zwei ernst zu nehmende Vereinfachungen, die nicht übersehen werden dürfen, wenn die Möglichkeiten der künstlichen Intelligenz beurteilt werden sollen. Wie wir gleich sehen werden, handelt es sich bei den beiden Tricks um *semantische Spezialisierung* und *semantische Verarmung*.

In der medizinischen Praxis kommen zunehmend interaktive Diagnosesysteme zur Anwendung. Ein solches System schlägt dem Arzt geeignete Fragen an den Patienten vor. Die Antwort wird über ein Eingabegerät (evtl. über einen akustischen Eingang einschließlich Spracherkenner) eingespeichert. Die Antworten werden klassifiziert, d.h. jede Antwort wird einer abgespeicherten Standardantwort zugeordnet. Das System stellt seine Diagnose. Gegebenenfalls stellt es mehrere Diagnosen mit Wahrscheinlichkeitsangaben. Die Hilfe des Diagnosesystems ist umso wertvoller, je

weitgehender das Denken des Arztes kalkülisiert (algorithmisiert) ist und je mehr Wissen und Erfahrungen abgespeichert sind. Zweifellos spielt die Erfahrung beim Diagnostizieren eine hervorragende Rolle.

Die Verwendbarkeit des Computers als Dialogpartner des Arztes beruht - ebenso wie im Falle von Eliza - auf Kalkülisierung, die durch *Klassifizierung* und *Standardisierung* ermöglicht wird. In erster Linie werden die Patientenantworten standardisiert. Das medizinische Wissen wird klassifiziert, um es schnell abrufbar einspeichern zu können. Ferner können auch Erfahrungen des Arztes klassifiziert und gespeichert werden. Das Kalkülisieren betrifft das Ableiten von Diagnosen aus den (standardisierten) Aussagen des Patienten, aus dem medizinischen Wissen und möglicherweise aus der Erfahrung des Arztes. Voraussetzung für eine sinnvolle Standardisierung ist inhaltliche, d.h. **semantische Spezialisierung**, d.h. die Beschränkung auf ein enges Spezialgebiet, auf einen kleinen Diskursbereich. 1

Derartige computerunterstützte Diagnosesysteme können als Erweiterung von Eliza aufgefasst werden. Der Kern beider Systeme ist - wie der aller schlussfolgernden Systeme - das regelbasierte Produzieren neuer Aussagen (bzw. Fragen) aus bekannten Aussagen. Voraussetzung ist, wie gesagt, Klassifizierung und Standardisierung der Aussagen, m.a.W. Festlegung einer endlichen Menge zugelassener Aussagen. Die Endlichkeit der Aussagemengen ist im Grunde nichts anderes als die Endlichkeit der zu berechnenden Funktionstafeln.

An dieser Stelle ist eine Zwischenbemerkung hinsichtlich der Standardisierung am Platze, die wir als Voraussetzung für den Mensch-Maschine-Dialog erkannt haben. Tatsächlich setzt jeder Dialog zwischen Menschen und überhaupt jede sprachliche Kommunikation Standardisierung voraus. Sie beginnt mit dem Hervorbringen von *Standardlauten* durch Tiere z.B. bei Gefahr. Der Mensch begann, Standardlaute zu Wörtern und Sätzen zu komponieren. Später erfand er *Standardzeichen*, die er den Standardlauten zuordnete. Davon war in Kap.5.1 [5.1] im Zusammenhang mit der Idemobjektivierung die Rede.

Die Herausbildung der Buchstabenschrift aus ursprünglichen bildlichen, d.h. analogen (in zweifachem Sinne des Wortes: “entsprechenden” und “nichtsprachlichen”) Darstellungen über Bilderschriften zeigt, dass zwischen analoger nichtsprachlicher und digitaler sprachlicher Artikulierung von Bewusstseinsinhalten, m.a.W. zwischen Urrealemen und Zeichenrealemen ein kontinuierlicher Übergang besteht (siehe Bild 2.1 und 3.1). Die Grenze zwischen sprachlicher und nichtsprachlicher “Codierung” ist also nicht scharf. Beispielsweise ist die Frage berechtigt, ob die malerischen Motive bei CHAGALL (Eselskopf, Geiger) oder die klanglichen Motive bei Richard WAGNER (Ringmotiv, Siegfriedmotiv) analoge Darstellungen oder codierende Zeichen sind.

Nach diesem Exkurs über die Wurzeln des Standardisierens kehren wir zurück zum Computer als Gesprächspartner. Auf Alltagsunterhaltungen, die jeden Diskursbereich berühren können, ist das Verfahren der Klassifizierung und Standardisierung kaum anwendbar. Nichtsdestoweniger kann der Computer auch hier als Dialogpart-

ner auftreten, allerdings aus einem ganz anderen Grund. Tatsächlich kommt Elizas Methode im Alltag gar nicht so selten zum Einsatz. Beispielsweise kann sie erfolgreich von jemandem angewendet werden, der verheimlichen will, dass er nicht weiß, wer die Person ist, die ihn soeben als alten Bekannten begrüßt und in ein Gespräch verwickelt hat. Er wird möglichst nichtssagende Antworten geben und selber Fragen stellen, deren Beantwortungen seinem Gedächtnis auf die Sprünge helfen soll. Ähnlich nichtssagend können Gespräche sein, mit denen nichts weiter bezweckt wird, als das Redebedürfnis zu befriedigen oder Zeit totzuschlagen, z.B. auf Kaffeekränzchen oder am Stammtisch. In solchen Gesprächen ist der Wert dessen, was gesagt wird, oft vernachlässigbar gering, die Gespräche sind kaum "informativ". Ein Computer könnte sich nach der *Eliza-Methode* prächtig an solchen Gesprächen beteiligen.

2 Die Verwendbarkeit des Computers als Gesprächspartner beruht diesmal auf einer relativen Sinn*losigkeit*, das heißt, auf einer Armut an Sinn, an Semantik, auf **semantischer Verarmung**. *Damit haben wir zwei Wege erkannt, den Computer dialogfähig zu machen:* ***semantische Spezialisierung*** *und* ***semantische Verarmung***. Das eine wie das andere ermöglicht die notwendige Standardisierung. Keiner der beiden Wege kann die Lösung sein, wenn man den Computer befähigen will, sinn*volle* (inhaltsvolle) Gespräch zu führen. Einem solchen Unterfangen stehen nach wie vor zwei miteinander verkoppelte Schwierigkeiten entgegen. Die Gesprächspartner müssen eine gemeinsame Wissensbasis besitzen und eine gemeinsame Sprache sprechen, in der sie sich unterhalten. Auf beide Schwierigkeiten soll in aller Kürze eingegangen werden.

Zunächst soll am Beispiel von Homonymen (Wörtern mit mehreren Bedeutungen) illustriert werden, wie wichtig die Gemeinsamkeit des Diskursbereiches ist, über den sich zwei Menschen unterhalten. Den Satz "Der Absatz gefällt mir" wird jedermann sofort richtig verstehen, allerdings völlig unterschiedlich, je nachdem, ob von einem Buch oder einem Schuh die Rede ist. Ähnliche Beispiele lassen sich für andere Homonyme anführen. Die Richtigkeit der Interpretation beruht auf dem Umstand, dass Gesprächspartner sich automatisch auf eine, dem Diskursbereich entsprechende "gemeinsame Sprache" einigen, genauer gesagt auf die Bedeutungen (Bewusstseinsinhalte, Ideme), die durch die Sprache artikuliert werden.

Da zwischen den Idemen der Partner ausreichende Entsprechung vorausgesetzt werden muss, können wir von Semantik sprechen [5.5]. Gesprächspartner müssen sich semantisch aufeinander *einstimmen*, sie müssen einen *semantischen Konsens* finden ("semantisch" als Adjektiv zu "Konsens" ist genau genommen überflüssig). Die Konsensfindung betrifft ausschließlich die Partner und die Dauer des Gesprächs. Insofern unterscheidet sie sich von *semantischer Objektivierung*, die universellen Charakter hat.

Wenn der Computer die Rolle eines Dialogpartners beim Lösen mathematischer wie nichtmathematischer Aufgaben spielen kann, so liegt der Grund dafür in der semantischen Objektivierung durch Anbindung an eine formale Semantik, wodurch

der "semantische Konsens" zwischen Mensch und Maschine gewährleistet ist. Andrerseits ist die Notwendigkeit der semantischen Konsensfindung gerade der Grund dafür, dass der Computer in der Regel *nicht* in der Lage ist, die Rolle eines Gesprächspartners zu spielen.

Man beachte, dass der Konsens nicht die Sprache selber betrifft, sondern das Wissen über den Diskursbereich. Allerdings kann die gegenseitige Einstimmung über den Bereich des *Wissens* hinausgehen und auch das *Gefühl* betreffen, beispielsweise in einem Gespräch über die Schönheit der Natur, in einem Gespräch zwischen Streithähnen oder zwischen Liebenden. Hier muss der Computer wohl vollständig seine Segel streichen. Wenn durch Sprache Emotionen "mitgeteilt" werden, wollen wir von *emotionaler Semantik* sprechen. Auch hinsichtlich emotionaler Semantik müssen Gesprächspartner einen Konsens finden, um sich zu verstehen. Man könnte von "Konsensibilisierung" sprechen. Wir vereinbaren: *Wenn Konsensfindung das gemeinsame Wissen von Gesprächspartnern betrifft, sprechen wir von* ***rationaler Konsensfindung****, wenn sie gemeinsame Gefühle betrifft, von* ***emotionaler Konsensfindung***. Letztere ist dem Computer verschlossen, wenn man davon ausgeht, dass er weder Bewusstsein noch Emotionen besitzt. 3

Es ist aber nicht nur die Semantik, sondern auch die Syntax der Sprache, die es dem Computer eventuell sehr schwer macht, an Alltagsgesprächen teilzunehmen. Das hat vor allem zwei Ursachen, die relativ komplizierte Grammatik natürlicher Sprachen und Verstöße gegen sie.

Natürliche Sprachen als Produkte der Evolution richten sich nicht nach Chomskys Erkenntnissen, sie sind leider nicht von Typ 3, 2 oder 1 (siehe Bild 16.5 in Kap. 16.5). Die syntaktische Analyse natürlichsprachiger Sätze stößt auf erhebliche Schwierigkeiten. Doch gibt es keinen stichhaltigen Grund, der ihre Möglichkeit prinzipiell ausschließt, abgesehen von dem notwendigen Aufwand an Speicherplatz und Rechenzeit. Die technische Entwicklung hat immer wieder scheinbar Unmögliches möglich gemacht.

Um den Computer konversationsfähig zu machen, könnte man auf die Idee kommen, eine Umgangssprache mit einfachen Syntaxregeln zu entwickeln. Doch erscheint es zweifelhaft, ob auf diese Weise die Ausdrucksstärke natürlicher Sprachen erreicht werden kann. Gegen diesen Zweifel sprechen die Erfolge der sogenannten *Plansprachen*. Das sind planmäßig konstruierte Umgangssprachen mit relativ einfacher Grammatik, sodass sie leicht analysierbar und erlernbar sind.

Die bekannteste Plansprache ist **Esperanto**. Sie wurde vor etwa 100 Jahren von dem polnischen Augenarzt LUDWIG ZAMENHOF entwickelt mit dem Ziel, der Welt eine internationale Kommunikationssprache zur Verfügung zu stellen. Dieser Gedanke kann heute beim Zusammenwachsen der Welt, was eine einheitliche Sprache erforderlich macht, politische Bedeutung erlangen. Denn die Einsetzung einer ethnischen Sprache, z.B des Englischen, als internationale Verkehrssprache könnte nationale Gefühle verletzen und das Zusammenwachsen behindern, während Esperanto es eher fördern würde.

SHAKESPEARE, GOETHE und PUSCHKIN sind in Esperanto übersetzt worden. Die Syntax lässt das trotz ihrer Einfachheit zu. Sogar ein in Esperanto übertragenes Gedicht kann etwa so verstanden werden, wie das Original. Offenbar hängt die Breite der Interpretationsmöglichkeit nicht oder nur wenig vom Kompliziertheitsgrad der Syntax der verwendeten Sprache ab. Das ist insofern verständlich, als die Interpretation auf der Grundlage rationaler und emotionaler Konsensfindung des Lesers mit dem Dichter erfolgt, was nicht Angelegenheit der Sprache, sondern des interpretierenden Menschen ist.

Ein anderer Weg zu einer breiten internationalen Kommunikation wäre die maschinelle Übersetzung zwischen allen beteiligten ethnischen Sprachen. Bei n beteiligten Sprachen wären $2n(n\text{-}1)$ Übersetzer erforderlich. Diese Zahl lässt sich erheblich herabsetzen, wenn eine einheitliche Zwischensprache eingeführt wird. Ein Quelltext wäre zunächst in die Zwischensprache und anschließend aus dieser in die Zielsprache zu übersetzen. In diesem Fall wären nur $2n$ Übersetzer erforderlich. An der Verwirklichung dieser Idee wird gearbeitet, wobei auch Esperanto als Zwischensprache benutzt wird.

Angenommen, alle grammatikalischen und lexikalischen Probleme des maschinellen Sprachverstehens seien gelöst. Dann bleibt immer noch eine große Schwierigkeit bestehen, von rationaler und emotionaler Konsensfindung ganz abgesehen. Im alltäglichen Gespräch kümmert man sich nämlich wenig um grammatikalische Regeln. Auf Schritt und Tritt wird gegen sie verstoßen, allerdings vorwiegend gewohnheitsmäßig, sodass auch für die Verstöße Regeln aufgestellt und implementiert werden können. Nehmen wir als Beispiel die Replik: “Das kannst du doch nicht machen” - “Doch” - “Nein”. Diese Worte reichen für die Verständigung in einer bestimmten Situation aus, auch wenn nur Bruchstücke ganzer Sätze artikuliert werden. Die Antwort “Doch” steht für den vollständigen Satz: “Das kann ich *doch* machen”, der gedanklich ergänzt wird. Was das erste “Das” bedeutet, geht aus der Situation (aus dem Kontext) hervor und wird ebenfalls gedanklich ergänzt. Die richtigen Ergänzungen sind aufgrund der gemeinsamen Einstimmung auf einen bestimmten Diskursbereich und aufgrund der Kenntnis der vorangegangenen Replik möglich.

Es gibt keinen Grund anzunehmen, dass der Computer prinzipiell nicht in der Lage ist, derartige Ergänzungen vorzunehmen, freilich wieder unter der Voraussetzung, dass ausreichend Zeit und Speicherplatz zur Verfügung steht. Wie groß der Aufwand werden kann, wird klar, wenn man bedenkt, dass jedes Wissenselement unter einer Adresse abgespeichert werden muss, und dass der Prozessor nur über Adressen auf sein Wissen zugreifen kann.

Die Evolution hat weit effektivere Methoden hervorgebracht, die es dem Menschen erlauben, nach Belieben und scheinbar unmittelbar und uneingeschränkt mit seinem gesamten Wissen zu hantieren. Die Hantierung mit Gedächtnisinhalten scheint der entscheidende Punkt zu sein, in dem sich Mensch und Maschine hinsichtlich ihrer Fähigkeiten zur Konversation unterscheiden. Es stellt sich die Frage, ob

dies vielleicht überhaupt der entscheidende Punkt ist hinsichtlich der unterschiedlichen intellektuellen Fähigkeiten des Menschen und des Computers. Wir kommen darauf in Kap.17.3 zurück.

17.2 Der Computer als Unterhalter

Die Rolle des Computers als Unterhalter, als Spielgefährte oder einfach als Zeittotschläger ist schon heute groß und sattsam bekannt. Sie wird sicher noch erheblich zunehmen, bis der menschliche Selbsterhaltungstrieb sich ernsthaft dagegen zur Wehr setzen wird. Wir werden uns nicht für die vielen Facetten dieser Rolle des Computers interessieren, sondern lediglich für die Frage, was Unterhaltung per Computer mit Informatik zu tun hat.

Wir hatten die Informatik als Lehre vom aktiven sprachlichen Modellieren definiert und erinnern an den Auskunftsautomat, der gewünschte Informationen schriftlich oder mündlich erteilt, als Beispiel für aktive (d.h. vom System selbst vollzogene) Artikulierung von Aussagen, m.a.W. für aktives sprachliches Modellieren eines bestimmten Diskursbereiches.

Wir erweitern das Beispiel um eine CD, auf der beispielsweise ein Gesundheitslexikon oder Goethes Faust abgespeichert ist. Der Besitzer der CD samt eines entsprechenden Computers kann in dem Lexikon oder im Faust herumblättern und nach Belieben lesen. Dabei gibt der Computer Zeichen aus, die der Nutzer interpretieren kann. Dieser empfängt also *Information*, genauer: er empfängt Zeichenketten, die Bewusstseinsinhalte aktivieren (Ideme auslösen) und dadurch zu Information werden [1.3]. Die Tätigkeit des Computers (die technische “Informationsverarbeitung”) beschränkt sich dabei im wesentlichen auf das Auffinden der gewünschten Textstellen und das Umcodieren aus dem Binärcode der CD in den Buchstabencode des Bildschirms. (Wir nehmen an, dass die CD nicht analog sondern digital geprägt ist. Beides ist möglich.)

Was aber findet statt, wenn der Computer stehende oder bewegte Bilder zeigt oder wenn er Musik macht, wenn also seine Mitteilungen nicht sprachlicher, sondern analoger (nichtsprachlicher) Natur sind? Um das Problem in seiner ganzen Breite zu erkennen, gehen wir davon aus, dass unser Computer Video- und Audio-Ausgänge besitzt. Er kann also Bitketten ausgeben, welche die Frequenzen und Amplituden von Licht- und Schallwellen codieren. Wenn diese Informationen auf einem peripheren Speicher des Computers gespeichert sind, z.B. auf einer CD, und wenn der Computer mit einem Fernsehgerät verbunden wird, spielt er die Rolle eines Videorekorders.

Wir interessieren uns zunächst für die visuelle Information und fragen, welche Formen sie auf dem Wege von der CD bis zum Nervensystem annimmt. Genauer gesagt, wir fragen nach der Transformation der Realeme, die letzten Endes vom Menschen interpretiert und dadurch zu Information werden. Auf der CD und im

Computer ist die Information binär codiert. Auf dem Bildschirm ist sie durch die Lage der beteiligten Bildpunkte (*Pixel*), durch diskrete Wellenlängenbereiche und Helligkeitswerte, mit denen die Pixel leuchten und - im Falle bewegter Bilder - durch diskrete Zeitpunkte codiert, zu denen sie leuchten. Eine Fernsehkamera zerlegt einen aufgenommenen Vorgang in eine diskrete Folge von Bildern und jedes Bild in eine diskrete Folge von Pixeln, die zeilenweise das Bild überdecken. Man nennt diese Art der Diskretisierung *Rasterung*. Farbe und Helligkeit der Pixel werden i.d.R. sequenziell übertragen und auf dem Bildschirm des Empfängers wieder zu dem gesendeten Bild zusammengefügt.

Sämtliche bildgebenden Größen (Zeitpunkt, Ort, Farbe, Helligkeit) werden so fein, in so kleinen Schritten diskretisiert (digitalisiert), dass der Betrachter sie als kontinuierliche (analoge) Größen empfindet, er nimmt kontinuierlich verlaufende Linien, Färbungen, Schattierungen und Bewegungen wahr. Das Entsprechende gilt für die Wiedergabe von Musik. Die Introspektion sagt uns, dass unsere auditiven und visuellen Eindrücke und Empfindungen in jedem Fall kontinuierlicher Natur sind, einerlei, wie sie zustande kommen und welcher Natur ihre Quellen (z.B. die reale Umgebung oder der Fernseher) sind.

Die Kontinuität der Empfindungen entspricht aber nicht der physiologischen Realität. Das kontinuierliche Bild der Außenwelt oder auch das scheinbar kontinuierliche Bild des Fernsehers, das die Linse auf die Netzhaut projiziert, wird durch die Struktur der Netzhaut diskretisiert. Es wird ein Bild perzipiert, das ähnlich wie das des Fernsehers aus Pixeln besteht. Jedem Stäbchen bzw. Kolben der Netzhaut entspricht ein Pixel. Die “Digital-Analog-Konvertierung” zu einer kontinuierlichen Empfindung ist das Produkt der Gehirntätigkeit. Ob eine solche Konvertierung neurophysiologisch tatsächlich stattfindet, ist zu bezweifeln. Angesichts der Funktionsweise der Neuronen ist es wahrscheinlich, dass eine *Umcodierung* stattfindet, wobei das Gehirn sowohl statisch als auch dynamisch codiert [9.1]. Experimentell ist dies gegenwärtig nicht eindeutig belegbar. Mit anderen Worten, es fehlt der experimentelle Beweis, dass Interpretieren (die Pfeile 1 und 5 in Bild 2.1) mit Codieren verbunden ist, dass also Urideme materielle Träger in Form neuronaler Zustände besitzen.

Unversehens ist unsere Frage zu einer sehr grundsätzlichen geworden: *Entsprechen **mentalen** Zuständen **nervale** codierende Zustände?* Im Nachwort werden einige philosophische Aspekte dieser Frage beleuchtet, ohne sie zu beantworten. Wir wissen also nicht, ob es sich bei den Pfeilen 2, 4 und 5 in Bild 2.1 um Verarbeitung von Zeichenrealemen handelt. Hingegen beinhaltet Pfeil 2 mit Sicherheit Codieren und Pfeil 3 Zeicheninformationsverarbeitung. Anders ausgedrückt, der internen wie externen Codierung liegt Standardisierung zugrunde. Die zu artikulierenden Bewusstseinsinhalte werden durch Abstraktion in Begriffe, in *Standardbedeutungen* und diese durch Benennung in *Standardzeichen* überführt. Mit diesem “Trick” löst die Natur den “*Widerspruch zwischen der kontinuierlichen Natur des Gedachten und der nichtkontinuierlichen Natur des Denkens*” (teleonomisch gesprochen).[1]

Ebenso wie in Kap.17.1 [1] kommen wir zu dem Ergebnis, dass Standardisierung Voraussetzung der Kommunikation ist. Früher hatten wir die Idemobjektivierung [5.1] als Voraussetzung genannt. Offenbar handelt es sich um zwei Seiten ein und desselben Sachverhaltes. Aus der Sicht der Kommunikation zwischen Menschen lag es näher, von Idemobjektivierung bzw. von semantischer Objektivierung [5.6] zu sprechen. Aus der Sicht der *technischen* Kommunikation liegt der Begriff der Standardisierung näher. Im Falle *menschlicher* Kommunikation schließt sich an die interne codierte Verarbeitung das externe Codieren (Pfeil 3 in Bild 2.1) an, z.B. durch Sprechen oder durch Schreiben mit der Hand, also durch kontinuierliche (analoge) Prozesse.

Die letzten Überlegungen zur Kommunikation gehen bereits über die ursprüngliche Fragestellung hinaus, die das Artikulieren nicht einschloss, wie z.B. das Nacherzählen einer gehörten CD oder einer Sendung. Doch demonstriert die Einbeziehung des Artikulierens besonders anschaulich den wiederholten Wechsel zwischen analoger und digitaler Informationsdarstellung, ja sogar die gleichzeitige Verwendung beider Darstellungsformen. Handschriftlicher Text ist z.B. eine kontinuierliche Darstellung von sprachlich codierten Bewusstseininhalten (Idemen), und das gerasterte Fernsehbild ist für den Zuschauer eine kontinuierliche Darstellung.

Der Wechsel zwischen analoger und digitaler Darstellung war bereits am Ende von Kap.4.2 diskutiert worden, jedoch aus rein technischer Sicht. Dort war auch der Begriff der Konvertierung zwischen den beiden Darstellungsformen eingeführt worden. Wir sahen, dass Konvertierung (in beiden Richtungen) das Zusammenschalten von Analog- und Digitalrechnern und so den Einsatz von Digitalrechnern bei der Lösung analoger Steuerungsaufgaben ermöglicht. Wir wollen uns nun überlegen, welche Möglichkeiten das Konvertieren dem Computer als Unterhalter eröffnet.

Die entscheidende Einsicht ist diese: Es ist immer möglich, einen Computer in einen bild- oder klanggebenden Übertragungskanal einzubinden. Diese Einsicht legt den Gedanken nahe, den Computer nicht nur zum Umcodieren und Suchen zu benutzen, sondern auch zur Manipulation derjenigen Bilder und Klänge, die ihn passieren. Per Programm lässt sich alles machen, aus einem X ein U, aus gelb grün, aus schön häßlich, aus Bach Jazz, aus meinem Gesicht das eines Papagein und aus seiner Stimme meine.

Der Computer kann auch am Anfang der Übertragungskette stehen und “selber”, durch Abarbeitung entsprechender Programme, Klänge und Bilder erzeugen. Er kann fiktive, nichtexistierende Welten hervorbringen. Die Illusion, dass es sich um *wirkliche* Welten handelt, kann dadurch bedeutend gesteigert werden, dass dem Betrachter die Möglichkeit gegeben wird, auf die vom Computer produzierte Welt einzuwirken, mit ihr in Kontakt zu treten und sich in ihr zu bewegen. Man spricht dann von **virtueller Realität**, und der Raum, in dem man sich bewegt, wird **Cyberspace**

1 Zitat aus Kap.4.1 [4.1].

genannt. Bereits der selbstvergessene Computerspieler befindet sich in einer anderen "Wirklichkeit", selbst wenn die Möglichkeiten, auf diese einzuwirken, sehr begrenzt sind.

Die virtuelle Realität kann noch erheblich realer erscheinen, wenn auch die anderen Sinnesorgane einbezogen werden. Hinsichtlich der taktilen Rezeption hat man es zu einigen Erfolgen und in bestimmter Hinsicht sogar zu beachtlichen Erfolgen gebracht. So kann der Computer in bereitwilligen Versuchspersonen unter Verwendung geeignet konstruierter Vorrichtungen den Orgasmus auslösen, natürlich auf optimierte Weise. Ob das eine positive Entwicklung ist und ob der Mensch sich auf diesem Wege dem Ebenbild Gottes nähert, ist fragwürdig.

Mit diesen Bemerkungen sollte nur ein kleines Schlaglicht auf das geworfen werden, was der Menschheit bevorstehen könnte, wenn sie alle Unterhalterpotenzen des Computers ausschöpfen würde.

So interessant die Unterhalterrolle des Computers in psychologischer und sozialer Hinsicht auch sein mag, aus der Sicht der Informatik als Wissenschaft vom aktiven sprachlichen Modellieren ist sie relativ uninteressant, denn die zentralen Fragen werden kaum berührt, weder die der Semantik noch die der künstlichen Intelligenz. Das technische Semantikproblem spielt bei der Unterhaltung durch Bilder und Klänge keine Rolle, denn es werden nicht die *Bedeutungen* von Bildern oder Klängen codiert, sondern die Bilder und Klänge *selber* und zwar durch einfache Diskretisierung ohne Anwendung intelligenter Verfahren wie Abstraktion, Klassifikation oder Standardisierung. Der Programmierer teilt dem Computer kaum etwas von seiner eigenen Intelligenz mit, im Gegensatz zu den in den Kapiteln 15 und 16 dargelegten KI-Methoden, in denen der Computer durch *Intelligenztransfer* per Programm zum Helfer beim Lösen von Aufgaben befähigt wurde.

Das Gesagte beinhaltet eine gewisse Abwertung des Computers als Unterhalter, zumindest eine Abwertung der wissenschaftlichen Bedeutung diesbezüglicher Erfolge. Daraus darf jedoch nicht der Schluss gezogen werden, die Bearbeitung von Bildern und Klängen sei grundsätzlich für die Informatik nicht von wissenschaftlichem Interesse. Innerhalb der Informatik hat sich ein selbständiger Wissenschaftszweig unter der Bezeichnung "Bildverarbeitung" etabliert, der sich mit der computerinternen Darstellung und Bearbeitung von Bildern befasst. Allein die Erwähnung der automatischen Auswertung von Lichtbildern macht die praktische Bedeutung dieser Arbeitsrichtung deutlich. Auch wird sofort klar, welch große Rolle bei der Bildverarbeitung das Klassifizieren und Standardisieren spielt, beispielsweise bei der rechnergestützten Überführung von Luftaufnahmen in Landkarten. In diesem Zusammenhang ist auch das Komponieren von Musik durch den Computer zu nennen, ein gerade aus *wissenschaftlicher* Sicht höchst interessanter Prozess.

Die vorangehenden Überlegungen haben gezeigt, dass der Computer allerhand kann, ohne besonders intelligent zu sein. Gerade darum können die gewonnenen Einsichten helfen, sich ein begründetes Urteil hinsichtlich Ursachen, Möglichkeiten und Gefahren der Informationsgesellschaft zu bilden. Unsere Einsichten lassen sich

in folgendem Sastz zusammenfassen. *Ursache dafür, dass die Informationsgesellschaft kräftige Wurzeln schlägt, ist weniger die künstliche Intelligenz, als vielmehr das massenhafte Eindringen von Computern in unser Leben oder noch treffender:* ***die Mitgestaltung unseres Daseins durch ein Heer von Prozessoren***.

Prozessoren mischen sich nachdrücklich in unseren Alltag ein. Am "aufdringlichsten" im sogenannten **Multimediabereich**, d.h. im Bereich des computergestützten Kommunizierens und Informierens unter Verwendung verschiedener Kommunikations*medien*, wie Druck, Bild und Ton. Oft weniger auffällig mischen sich Prozessoren in all unser Tun ein, primär in das weniger intelligente Tun wie das Ausführen häufig sich wiederholender manueller Tätigkeiten (z.B. das Bedienen von Geräten und Maschinen) oder das Anstellen häufig notwendiger Überlegungen und Rechnungen (z.B. in Verbindung mit Haushalt und Einkauf oder mit dem Ausfüllen von Formularen, wie Anträgen oder Steuererklärungen). Vor allem aber mischen sie sich in die Unterhaltung ein und vervielfachen die nutzlos verbrachte Zeit, obwohl sie eigentlich dazu bestimmt sind, Zeit zu sparen.

Dass der Computer dabei eine unerhörte Penetranz entwickelt und keinen Lebensbereich auslässt, ist eine Folge der Ausnutzung menschlicher Grundeigenschaften durch die Marktwirtschaft. Denn Hard- und Software bringt Geld, und das offenbar umso mehr, je "niederer" die Bedürfnisse sind, die sie befriedigen, d.h. je mehr diesen Bedürfnissen Instinkte zugrunde liegen.

17.3 Der Computer als Schachpartner

Wir wollen nun noch eine sehr spezielle Rolle des Computers untersuchen, die Rolle eines Schachpartners. Man kann sie als Kombination der Rolle eines Unterhaltungspartners mit der eines Problemlösungspartners auffassen. Anhand des Schachspielens wollen wir noch einmal der immer wieder gestellten Frage nach den Grenzen der künstlichen Intelligenz nachgehen, diesmal konkret in der Form:

Wieweit lässt sich Schachintelligenz simulieren?[2]

Diese Frage ist im Grunde ebenso sinnlos wie die Frage nach den Grenzen der künstlichen Intelligenz überhaupt. Dennoch wird sie uns weiterhelfen. Zunächst rüsten wir das Problem ab, indem wir, ebenso wie früher bei der Behandlung anderer Partnerrollen des Computers, pragmatisch vorgehen und uns mit dem Erfinden (Nacherfinden) einiger konkreter Möglichkeiten, den Computer schachintelligent zu machen, begnügen. Wir wollen "schlaue" Algorithmen erfinden. Dabei gilt in besonderem Maße das zu Beginn des Kapitels 17.1 Gesagte: Es handelt sich weitgehend um spekulative Überlegungen in dem Sinne, dass wir uns auf keine Autoritäten

2 Es sei auf den sehr aufschlussreicher Artikel über das Computerschach von JÜRG NIEVERGELT [Nievergelt 96] hingewiesen.

und auf keine ausgebauten Theorien stützen können. Solche existieren nur in sehr begrenztem Umfang. Auch die Spieltheorie kann uns kaum helfen. Wir werden uns auch nicht auf existierende Schachprogramme stützen, denn die "wollen gewinnen", während wir erkennen wollen. Wir wollen erkennen, worin *natürliche* Schachintelligenz besteht, um sie kalkülisieren und implementieren zu können. Warum existierende Schachprogramme uns nur wenig helfen können, wird später begründet.

In Kap.21.3 werden die Mechanismen (Algorithmen) der Schachintelligenz, die wir uns anschicken zu erfinden, auf andere Arten des sprachlichen Modellierens verallgemeinert. In diesem Zusammenhang sei an zwei Vereinbarungen erinnert.

Erste Vereinbarung. Wir hatten jedes *codierende* Modellieren, also auch gedankliches Modellieren, *sprachliches* Modellieren genannt und die Fähigkeit zum sprachlichen Modellieren hatten wir *Intelligenz* genannt. Schachspielen stellt einen fortlaufenden *Problemlösungsprozess* dar. Das Nachdenken über den nächsten Zug (Lösen des aktuellen Problems) ist eine spezielle Form des sprachlichen Modellierens. Die spezielle Fähigkeit, über eine Schachpartie *nachzudenken*, d.h. die Partie sprachlich (gedanklich) zu modellieren, ist eine spezielle Art von Intelligenz. Wir nennen sie **Schachintelligenz**. Sie befähigt den Spieler, das *Schachproblem* so gut er kann zu lösen, d.h. die für ihn besten Züge zu erkennen.

Zweite Vereinbarung. Unter *Simulieren* hatten wir das Modellieren von Eigenschaften und Verhaltensweisen irgendwelcher Objekte oder Systeme auf einem Computer verstanden, also sprachliches Modellieren mittels Computer. In diesem Kapitel wird ein spezielles Simulieren behandelt, das Modellieren ("Nachmachen") menschlicher Schachintelligenz auf einem Computer, genauer auf einem *Prozessor*computer. Wir benutzen das Wort Simulieren sowohl hinsichtlich des Programmierers, der das *Simulations*programm schreibt, als auch hinsichtlich des Computers, der das Programm ausführt.

Nach diesen Rückerinnerungen und Vereinbarungen wenden wir uns der gestellten Frage zu, wollen sie aber zuspitzen auf die Frage: *Kann der Computer Schachweltmeister werden?* Die Frage ist nicht, ob es möglich ist, dass ein Computer in der Lage ist, eine Schachpartie gegen den Weltmeister zu gewinnen, sondern ob künstliche Schachintelligenz produziert werden kann, die grundsätzlich der natürlichen überlegen ist. Diese Frage ist sinnvoll, denn es wird nicht nach prinzipiellen Grenzen der Intelligenz gefragt, sondern nach dem Unterschied zwischen natürlicher und künstlicher Intelligenz hinsichtlich eines ganz bestimmten Problems.

Den Sieg eines Computers über einen Schachweltmeister könnte ein Journalist mit der Schlagzeile kommentieren: "Maschinelle Intelligenz besiegt menschliche Intelligenz". Das wäre ein typisches Beispiel für Furoremachen durch Irreführung, selbst dann, wenn die Aussage an sich stimmt. Die Irreführung ist eine doppelte. Zum einen ist die Schlagzeile eine Binsenwahrheit. Denn seit langem gibt es Schachprogramme, die in der Lage sind, mittelmäßige Schachspieler zu besiegen. Zum anderen provoziert sie die unerlaubte Verallgemeinerung, der Computer sei intelligenter als der Mensch, während er lediglich "*schach*intelligenter" ist. Genauer gesagt bedeutet

der Sieg des Computers lediglich, dass er in einem bestimmten Fall schachintelligenter gewesen ist als derjenige Mensch, der vom Computer geschlagen wurde. Betrachtet man die Ergebnisse der Schachturniere "Mensch contra Computer" und setzt den Trend in die Zukunft fort, muss man allerdings zu dem Schluss kommen, dass schließlich wohl der Computer als Sieger aus dem Duell hervorgehen wird

Wodurch zeichnet sich Schachintelligenz aus? Wie spielt man Schach? Welche intelligenten Operationen führt man aus? Diese Fragen wird derjenige stellen und zu beantworten versuchen, der ein Schachprogramm schreiben will. Denn offenbar muss man das Vorgehen des Menschen genau *verstanden* haben, bevor man es simulieren, d.h. kalkülisieren und algorithmieren kann[3].

Die erste Idee, die sicher jedem kommt, der Schachspielen simulieren will, ist das gedankliche *Vorausspielen*, das Suchen nach dem besten Zug durch *Probieren*. So sind auch zwei Pioniere der Informatik und des Computerschach vorgegangen, TURING und SHANNON (siehe [Nievergelt 96]). Es ist ein *Suchproblem* zu lösen. Auf ein ganz anderes und dennoch im Prinzip ähnliches Suchproblem waren wir in Kap.15.8 im Zusammenhang mit dem analytischen Rechnen gestoßen. Beim analytischen Rechnen sucht sich der Computer in einem *Suchgraphen* (im Labyrinth aller möglichen Ableitungswege) seinen Weg. Die Knoten des Graphen waren damals mathematische Ausdrücke; jetzt sind es Stellungen auf dem Spielbrett. Die Kanten waren dort Transformationen gemäß Formeln; jetzt sind es Züge gemäß Regeln. Ein wesentlicher Unterschied besteht darin, dass Schach ein Zweipersonenspiel ist, und dass man die Züge des Gegners nicht vorhersehen kann.

Angesichts dieser Unbestimmtheit könnte man bei der Spieltheorie Hilfe suchen. So verfährt jedoch kein Schachspieler. Der benutzt vielmehr seine Intelligenz, d.h. seine Fähigkeit zum sprachlichen (gedanklichen) Modellieren dazu, die Partie *in Gedanken* weiterzuspielen und dabei verschiedene Züge des Gegners ins Kalkül zu ziehen.

Für ein solches Vorgehen scheint der Computer wegen seiner Schnelligkeit prädestiniert zu sein. Bevor er einen Zug ausführt, probiert er "in Gedanken" alle möglichen Züge durch, alle möglichen Antwortzüge des Gegners, alle möglichen eigenen Antworten auf die Züge des Gegners und so fort. Diese Methode des *vollständigen Durchmusterns* lässt sich zwar ohne besondere Schwierigkeiten implementieren, doch ist die Anzahl der Züge einer vorausgespielten Zugfolge, die sogenannte *Tiefe* des Vorausspielens, auf einige wenige Züge begrenzt, wie folgende grobe Abschätzung zeigt.

Beim Eröffnungszug hat Weiß 20 Möglichkeiten. Danach hat Schwarz 20 Möglichkeiten, sodass nach zwei Zügen, Zug und Gegenzug), 20^2=400 verschiedene Stellungen möglich sind. Nach drei Zügen wären näherungsweise 20^3 und nach n Zügen 20^n verschiedene Stellungen möglich. Tatsächlich liefert diese Extrapolation

3 Der Einsatz neuronaler Netze wird hier nicht betrachtet.

für kleine *n* (Eröffnungsspiel) zu niedrige Werte, da die Anzahl der Zugmöglichkeiten im Mittel mit *n* zunimmt, für große *n* hingegen zu hohe Werte, zum einen infolge des Ausscheidens von Figuren, zum anderen weil jede Stellung auf vielen Wegen erreicht werden kann. Im Endspiel verliert die Näherung ihren Sinn. Die Anzahl der möglichen Züge kann vom Gegner gezielt herabgesetzt werden, im Grenzfall auf Null (vollständiges Blockieren).

Bereits für relativ kleine *n* ergibt die Abschätzung eine riesige Zahl (man spricht von "kombinatorischer Explosion"), für $n = 10$ beispielsweise $20^{10} \approx 10^{13}$ mögliche Stellungen. Allein um sie aufzuzählen benötigt ein Computer, der pro Sekunde, sagen wir, eine Million Stellungen generieren kann, etwa 4 Monate. Dabei ist die Näherung 20^n nach 10 Zügen eher zu niedrig als zu hoch. Eine in [Reischuk 90] angegebene Abschätzung besagt, dass bei einer Generierungsgeschwindigkeit von einer Milliarde Stellungen pro Sekunde 100 Millionen Jahre nicht ausreichen, um alle legalen Spielstellungen (das sind nach Reischuk über 10^{24}) aufzuzählen.

Die Frage nach dem richtigen Schachzug kann offensichtlich weder der Mensch noch der Computer durch vollständiges Durchmustern, d.h. durch vollständiges Vorausspielen der Restpartie lösen. Das Problem (der Suchgraph) ist *zu komplex*. In Kap.21.2 wird ein Problem, dessen Lösungsaufwand mit einem charakteristischen Parameter, dem sog. Problemumfang (in unserem Fall mit *n*) exponentiell zunimmt,
4 als Problem mit *exponentieller Komplexität* bezeichnet. Derartige Probleme werden schon für relativ kleinen Problemumfang unlösbar, d.h. in zulässiger Zeit nicht berechenbar. Die sog. *Berechnungskomplexität* ist zu hoch.

Wir werden den Begriff der Berechnungskomplexität (im Weiteren kurz Komplexität genannt) schon in diesem Kapitel benutzen, obwohl er erst in Kap.21.2 definiert wird. Dabei wird ein intuitives Verständnis des Begriffs beim Leser vorausgesetzt. Damit können wir unser Ergebnis in folgende Worte fassen: Das vollständige Durchmustern (das Vorausspielen aller Restpartien) scheitert an der Komplexität des Problems. Dabei handelt es sich nicht um *prinzipielle*, sondern um *praktische* Unmöglichkeit. Je schneller die Computer werden, umso weiter (tiefer) können sie in zulässiger Zeit vorausrechnen. Es ist ein Kampf *Rechengeschwindigkeit contra Komplexität* im Gange.

Die Komplexität des Problems wird noch offensichtlicher, wenn man bedenkt, dass jede Stellung nicht nur generiert (gestellt), sondern auch bewertet werden muss. Der Frage nach der Bewertung waren wir bereits beim analytischen Rechnen begegnet. An jeder Wegegabel im Labyrinth (an jeder Verzweigung im Suchgraph) musste beurteilt werden, welcher Weg (die Anwendung welcher Formel) eher zum Ziel führt. Zu diesem Zweck wurde der *Zielabstand* [15.19] eingeführt. Diese Methode ist beim Schach kaum anwendbar (abgesehen vom Endspiel). Wenn die Intelligenz des Spielers (des Menschen oder des Computers) einzig und allein im "stupiden" Suchen besteht, kann eine Stellung nur dadurch bewertet werden, dass die Partie bis zum "bitteren Ende" durchgespielt wird, was jedoch aus Zeitgründen unmöglich ist.

Der Spieler könnte sein Vorausspielen beim Erreichen einer Stellung abbrechen, in der er ohne eigenen Verlust eine Figur des Gegners schlagen kann (Erreichen eines Teilziels). Doch kann auch das schon zu lange dauern. Abgesehen davon ist diese Taktik aus zwei weiteren Gründen fragwürdig. Zum einen kann das Schlagen einer gegnerischen Figur zu einer für den Schlagenden ungünstigen, vielleicht sogar tödlichen Stellung führen. Zum anderen kann das Opfern einer eigenen Figur zu einem Stellungsvorteil führen, der den Verlust überwiegt. Das Vorausspielen muss also fortgesetzt werden.

Der Spieler muss sich entscheiden, warauf er mehr Zeit verwendet, auf das Vorausspielen, also auf das Generieren möglicher Folgestellungen, oder auf die Analyse der generierten Stellungen hinsichtlich ihres taktischen und strategischen Wertes. Das gilt auch für den Computer. Dabei tritt ein auffallender Unterschied zutage. Hinsichtlich der Generierung möglicher Stellungen ist der Computer dem Menschen überlegen, während bei ihrer Bewertung der Mensch dem Computer überlegen ist. Wenn Mensch und Computer ihre Kräfte messen, ist es für den Computer angesichts seiner Schnelligkeit sicher vorteilhafter, das Vorausspielen zu bevorzugen, für den Menschen dagegen das Analysieren und Bewerten. Die Praxis des Computerschach bestätigen dies. Schachprogramme sind i.d.R. auf schnelles Vorausspielen gezüchtet. Aber nicht nur von der Software, sondern auch von der Hardware hängt die Rechengeschwindigkeit des Computers ab. In den vergangenen Jahren hat gerade die Hardware die Spielqualität des Computers erheblich gesteigert. Die heutigen Hochleistungs-Schachcomputer sind Spezialcomputer, die über viele Prozessoren verfügen. In Kap.19.5.4 wird darauf kurz eingegangen.

Weil der Mensch langsamer ist als der Computer, muss er “intelligenter” sein; er muss effektiver suchen und den Wert einer Stellung “auf einen Blick”, ohne langes Analysieren, also *intuitiv* erkennen können. Tatsächlich kann er das; aber *wie macht er das*? Wir erinnern uns an die Bemerkung in Kap.15.8 [15.20], dass ein ahnungsloser Zuschauer den Eindruck haben kann, dass der erfahrene Schachspieler, den er beobachtet, *intuitiv* die richtigen Züge macht, weil er dermaßen schnell zieht, dass zum Nachdenken kaum Zeit bleibt. Wieder die Frage: *Wie macht er das*? Wir wollen versuchen, die Antwort durch Introspektion zu finden. Wir werden dem Denken und Trainieren des Schachspielers nachgehen und uns überlegen, wieweit sich beides simulieren lässt.

Eine sehr einfache Antwort auf die Frage nach der Wurzel der Intuition des Spielers wäre, dass er die Partie auswendig kennt. Für einen erfahrenen Spieler kann das zutreffen, zumindest hinsichtlich des Eröffnungsspiels, das er unzählige Male und in allen Varianten durchexerziert hat. Aus dem gleichen Grund kann er auch in der Lage sein, das Endspiel “automatisch” zu absolvieren. Wenn in solchen Fällen ein Spieler *intuitiv* zu spielen scheint, so verbirgt sich hinter der Intuition weiter nichts als *Erfahrung* und zwar *bewusste* Erfahrung. In diesem Fall sprechen wir von **scheinbarer Intuition**.

Auch im Mittelspiel kann Erfahrung helfen, dort allerdings mehr punktuell, hinsichtlich bestimmter Stellungen, die der Spieler sich eingeprägt hat mitsamt dem jeweils günstigsten folgenden Zug. Tatsächlich ist das Einprägen von Stellungen und sogar ganzer Partien eine Methode, die eigene Spielqualität zu erhöhen. Je mehr Partien ein Spieler im Kopf hat, umso öfter wird er eine aktuelle Stellung *wiedererkennen* und *wissen*, welches der nächste "richtige" Zug ist, d.h. welcher Zug in der aktuellen Situation schon einmal mit Erfolg ausgeführt worden ist. Ein solches Vorgehen, das sich an konkreten Stellungen, an konkreten "*Fällen*" orientiert, wird fallorientiert oder **fallbasiert** (englisch: case based) genannt.

6 Ein fallbasierter und infolgedessen schneller Schachzug wird auf denjenigen, der die Erfahrung des Spielers nicht kennt, den Eindruck der *Intuition* machen. Dieser Eindruck wird verstärkt, wenn der Spieler selber seinen Zug nicht genau erklären kann, weil die zugrundeliegende Erfahrung nicht in sein Bewusstsein getreten ist, was durchaus verständlich wäre. Wenn ein Spieler wiederholt mit ein und derselben Stellung (oder einer ähnlichen) konfrontiert wird, liegt die Annahme nahe, dass sich - in entfernter Analogie zu PAWLOVS Hundeexperiment - so etwas wie ein bedingter Reflex ausbildet in dem Sinne, dass der Entscheidungsprozess nicht mehr ins Bewusstsein tritt, dass er "*interiorisiert*" wird. Es stellen sich folgende Fragen:

- Ist Schacherfahrung auf den Computer übertragbar?
- Ist das Sammeln von Erfahrung simulierbar?
- Ist das Anwenden von Erfahrung simulierbar?

Wir werden die Fragen ausführlich beantworten.

Ein Anfänger wird nicht dadurch Schachspielen lernen, dass er Großmeister kopiert, sondern in erster Linie dadurch, dass er übt, also spielt und am eigenen Spiel *lernt*. Er sammelt Erfahrung. Er merkt sich typische Fehler und Erfolge; er merkt sich typische *Fälle*. Er prägt sich nicht fremde sondern eigene Partien ein. Er lernt Fälle, und er lernt aus Fällen. Letzteres bedeutet, dass er seine Erfahrungen aus vielen Fällen zu **Erfahrungsregeln** verdichtet, indem er aus einer Reihe von Fällen, die in bestimmter Hinsicht einander *ähnlich* sind, eine Erfahrungsregel extrahiert, z.B. die Regel, dass man einen Springer nicht auf ein Randfeld stellen soll. Diese Regel hat einen so durchsichtigen Grund, dass man sie auch ohne Erfahrung unmittelbar aus den *Spiel*regeln ableiten kann. Auf den Randfeldern hat ein Springer nämlich höchstens halbsoviele Zugmöglichkeiten, wie auf den meisten anderen Feldern (vorausgesetzt, die Zielfelder sind nicht von eigenen Figuren besetzt).

Die *Ähnlichkeit* aller Stellungen, für welche die eben genannte Erfahrungsregel gilt, besteht darin, dass sie alle durch das *Merkmal* "eigener Springer auf einem Randfeld" charakterisiert sind. Wenn man zwei Stellungen (allgemein zwei Objekte) einander *ähnlich* nennt, wird damit zum Ausdruck gebracht, dass sie in einem oder mehreren Merkmalen übereinstimmen. In je mehr Merkmalen sie übereinstimmen, umso ähnlicher sind sie sich. Wir kommen darauf später genauer zurück.

Es gibt viele derartige Erfahrungsregeln für einander ähnliche Stellungen. Der Anfänger muss sie nicht alle selber finden. Sie sind in Schachlehrbüchern zusam-

mengestellt. *Wenn ein Spieler sich aufgrund einer Erfahrungsregel für einen Zug entscheidet, sprechen wir von* ***regelbasiertem Entscheiden*** *im Gegensatz zum* ***fallbasierten Entscheiden***. Eine Regel ist in vielen Spielsituationen anwendbar, sie gilt für viele *Fälle*.

Wir wiederholen das Gesagte noch einmal mit anderen Worten. Nachdem sich der Anfänger (der *Lernende*) die *Spielregeln* angeeignet hat, stehen ihm folgende Wege offen, seine Spielqualität zu erhöhen (das *Lernen* fortzusetzen): Übernahme fremder Erfahrung (*Reproduktion von Wissen*) und Sammeln eigener Erfahrung (*Produktion von Wissen*). Das Wissen betrifft sowohl *Fälle* (ganz bestimmte Stellungen) als auch *Erfahrungsregeln*. Die Qualifizierungswege (die Lernmethoden) setzen *reproduktive* bzw. *produktive Intelligenz* voraus [3.1].

Es sind also vier Arten des Wissenserwerbs, vier Qualifikationswege zu unterscheiden, *die* ***Produktion*** *von* ***Fall-*** *und* ***Regelwissen*** *und die* ***Reproduktion*** *(Übernahme) von* ***Fall-*** *und* ***Regelwissen***. *Die Interiorisierung von Wissen führt zum Erscheinungsbild der* ***Intuition***. Wenn nichts Gegenteiliges gesagt wird, ist im Weiteren unter *Regelwissen* nicht *Spiel*regelwissen, sondern *Erfahrungs*regelwissen zu verstehen.

Es erhebt sich die Frage, welche Arten des Wissenserwerbs dem Computer offen stehen, m.a.W. welche diesbezüglichen Fähigkeiten der natürlichen Intelligenz simulierbar sind. Da Implementieren Algorithmieren und Algorithmieren Kalkülisieren voraussetzt, lässt sich die Frage ganz allgemein, aber ziemlich nichtssagend folgendermaßen beantworten: *Die vier Qualifikationswege lassen sich - wie jede Fähigkeit der natürlichen Intelligenz - soweit simulieren, wie man sie kalkülisieren und in einen Algorithmus überführen kann.* Ob die Anwendung simulierter Fähigkeiten zum Erscheinungsbild der Intuition führt, hängt in erster Linie von der Rechenzeit ab, die der Computer für eine Entscheidung benötigt. Es bleibt allerdings dahingestellt, ob das Problem der Simulierbarkeit der Intuition damit aus der Welt geschafft ist. Doch lässt es sich, nachdem wir die vier Qualifizierungswege herausgearbeitet haben, konkretisieren. Unter diesem Gesichtspunkt wollen wir die vier Wege der Reihe nach untersuchen.

1. Übernommenes Fallwissen. Um fremdes Wissen (fremde Erfahrung) anwenden zu können, muss es zunächst abgespeichert werden. Wir gehen davon aus, dass Fallwissen bestimmten Spielstellungen bestimmte Züge zuordnet, sodass es eine Funktion darstellt. Jeder Fall (jede Spielstellung) stellt einen Argumentwert und der dazugehörige Zug den entsprechenden Funktionswert dar. Das Fallwissen stellt also eine *Funktionstafel* dar und kann *strukturell* (z.B. als Kombinationsschaltung) oder *adressiert* (z.B. in einem peripheren Speicher) abgespeichert werden. Fallwissen kann auf positiver wie auf negativer Erfahrung beruhen, d.h. auf erfolgreichen Zügen, die zu empfehlen sind, sowie auf Zügen, die zum Misserfolg geführt haben und darum besser zu unterlassen sind.

Um das abgespeicherte Fallwissen anwenden zu können, muss der Computer über ein Programm verfügen, das ihn befähigt, eine aktuelle Stellung *wiederzuerkennen*,

das heißt zu erkennen, dass sie im Erfahrungswissen (in der Funktionstafel) enthalten ist. Dazu muss er Stellungen miteinander Feld für Feld vergleichen können. Das zu realisieren bereitet keine prinzipiellen Schwierigkeiten, abgesehen vom Zeitaufwand. Je mehr Fälle die Tafel (das Fallwissen) enthält, umso besser spielt der Computer, umso höher ist seine Schachintelligenz, aber umso länger braucht er für jeden Zug. Der Zeitaufwand lässt sich u.a. dadurch herabsetzen, dass nicht die vollständige Stellung (das ganze Schachbrett), sondern nur ein Teil in Betracht gezogen wird. Diese Möglichkeit werden wir weiter unten aufgreifen.

Der Computer würde perfekt spielen, wenn die Funktionstafel sämtliche denkbaren Fälle umfasste, also - gemäß oben genannter Näherung - mindestens $10^{24} \approx 2^{80}$ Fälle (siehe Formel (9.4b)). Für ihre adressierte Abspeicherung wäre eine Adresse von mindestens 80 Bit Länge und für die strukturelle Abspeicherung eine Kombinationsschaltung mit 80 Eingängen erforderlich. Die Schaltung müsste eine 80-stellige boolesche Funktion berechnen. All das ist praktisch nicht zu verwirklichen.

Schnelles fallbasiertes Spielen würde als intuitives Spielen erscheinen. Es scheitert, ebenso wie das reine Durchsuchen, an der Komplexität des Problems. Dabei handelt es sich wiederum nicht um ein prinzipielles, sondern um ein praktisches Scheitern. Auf diese Weise lässt sich der perfekte Schachcomputer also *nicht* realisieren, nicht nur wegen der Adress- bzw. Wortlänge, sondern auch wegen der Notwendigkeit, zuvor für jede der 10^{24} Stellungen den günstigsten Zug zu ermitteln. Fallwissen kann also nur eine sehr kleine Auswahl aller möglichen Stellungen umfassen. Die Auswahl muss der Programmierer treffen, oder er muss den Computer befähigen, selber Erfahrung zu sammeln.

2. Selbstproduziertes Fallwissen. Damit der Computer eigene Erfahrung sammeln kann, muss er über ein Programm verfügen, das ihn befähigt, Züge zu bewerten und entsprechend der Bewertung zusammen mit der Spielstellung als positive oder negative Erfahrung abzuspeichern. Damit stoßen wir wieder auf das bereits besprochene Problem der Bewertung und ihrer Simulierbarkeit. Offensichtlich ist es günstiger, das vom Menschen akkumulierte Fallwissen zu übernehmen und zu nutzen.

3. Übernommenes Regelwissen. Regelwissen unterscheidet sich von Fallwissen dadurch, dass nicht das ganze Spielbrett betrachtet wird, sondern nur ein Ausschnitt,
8 nur ein Teil des Brettes und nur ein Teil der Figuren. Einen solchen Ausschnitt nennen wir **Konfiguration**. Eine Konfiguration kann eine, zwei oder mehrere Figuren enthalten. Eine Konfiguration aus einer Figur ist z.B. “weißer König steht auf a1.” (Wir nehmen uns die Freiheit, von *Kon*figuration aus *einer* Figur zu sprechen, obwohl die Bezeichnung paradox ist.) Zunächst überlegen wir uns, wie der Mensch Erfahrungsregeln produziert, was sie beinhalten und wie sie in Worte gefasst werden können.

Zu einer **Erfahrungsregel** gelangt man durch Zusammenfassung von Konfigurationen zu einer *Klasse* und zwar solcher Konfigurationen, die einander so ähnlich sind, dass sie zu ein und derselben Empfehlung führen, einen Zug auszuführen bzw. zu unterlassen. Eine Empfehlung kann auch mehrere gute und schlechte Züge

enthalten, wodurch die Entscheidung nichtdeterministisch wird. Umgangssprachlich wird eine Konfigurationsklasse i.Allg. durch einen Aussagesatz beschrieben, beispielsweise durch den Satz “Springer steht auf Randfeld” oder “König ist auf Grundlinie eingemauert”, d.h. die eigenen Figuren hindern ihn daran, die Grundlinie zu verlassen. Die Empfehlung wird umgangssprachlich natürlicherweise als Imperativsatz artikuliert, z.B. “Verlasse das Randfeld!” oder “Öffne die Mauer!”

Die “Mauerregel” kann durch Bewegen einer von drei (am Rande von zwei) Figuren befolgt werden, vorausgesetzt, es stehen keine weiteren eigene Figuren im Wege. Welcher konkrete Zug auszuführen ist, hängt von dem Feld ab, auf dem der König steht, und von den mauernden Figuren, aber auch von der Gesamtstellung. Diese kann den Spieler sogar veranlassen, die Regel zu negieren.

Wie lassen sich Erfahrungsregeln und ihre Anwendung implementieren? Hinsichtlich der Form der Abspeicherung der Regeln steht einem, wenn man keine bessere Idee hat, die *extensionale Notlösung* offen, d.h. die Aufzählung aller Konfigurationen, die zu der betreffenden Klasse gehören, und aller Züge der betreffenden Empfehlung. Doch wird für umfangreichere Konfigurationen die Anzahl der Elemente einer Klasse sehr schnell so groß, dass deren Abspeicherung auf Schwierigkeiten stößt oder sogar unmöglich wird. Wieder ist es die Komplexität des Problems, an der eine allgemeine Lösung, d.h. die Implementierung eines *universellen* Algorithmus, der in jedem Falle ausreichend schnell terminiert, scheitert (kombinatorische Explosion). Es handelt sich also auch hier um ein praktisches, nicht um ein prinzipielles Scheitern.

Gegen die kombinatorische Explosion kann man sich eventuell dadurch zur Wehr setzen, dass man Variable für Felder und/oder Figuren einführt und eine Konfigurationsklasse durch eine konkrete Konfiguration und ein Prädikat festlegt und zwar durch eine Formel, nach der die Konfigurationen der Klasse berechnet werden können. Auf diese Weise lassen sich z.B. alle Konfigurationen generieren, die sich durch *Translation* (z.B. durch Verschieben des Königs auf der Grundlinie) oder durch *Spiegelung* einer gegebenen Konfiguration an einer senkrechten oder waagerechten Gitterlinie des Schachbretts entstehen. Aber auch diesem Weg ist bei zunehmender Komplexität früher oder später eine Grenze gesetzt.

Das Anwenden einer Regel muss offensichtlich ähnlich beginnen, wie das Anwenden von Fallwissen, nämlich mit dem Vergleich der aktuellen Stellung mit den Bedingungen der Regeln, diesmal mit dem Ziel, herauszufinden, ob eine Regel anwendbar ist, d.h. ob die aktuelle Stellung eine Konfiguration enthält, die eine konkrete Realisierung (ein *Element* in der Sprechweise der Mathematik) der Konfigurationsklasse einer Regel ist. Dieser Satz weckt beim Leser möglicherweise eine Assoziation. Das Suchen nach einer passenden Regel war der entscheidende Schritt im Markovalgorithmus und beim analytischen Rechnen. Die aktuelle Stellung entspricht einer zu transformierenden Zeichenkette und eine Konfiguration einer zu substituierenden Teilkette. Weiter unten kommen wir auf diese Analogie zurück.

Das Suchen nach einer passenden Regel kann als Klassifizieren aufgefasst werden, genauer als *Klassieren*, denn die aktuelle Stellung wird in eine *bekannte* Klasse eingeordnet und zwar in die Klasse derjenigen Stellungen, die der Bedingung der Regel entsprechen, d.h. die eine Konfiguration enthalten, die in die Konfigurationsklasse der Bedingung der Regel fällt. In diesem Sinne kann man die Konfigurationsklasse als *Merkmal* der ihr entsprechenden *Stellungsklasse* auffassen.

Das Klassi*fizieren* (das "Machen" der Konfigurations- und Stellungsklassen) ist Aufgabe desjenigen, der die Regel erstellt. Er legt die Konfigurationsklassen fest. Die entsprechenden Stellungsklassen ergeben sich durch Vervollständigungen der Konfigurationen zu allen möglichen Stellungen. Insofern beruht das Erstellen einer Regel auf *Abstraktion*. Es werden nur Ausschnitte von Stellungen betrachtet, vom Rest der jeweiligen Stellung wird abstrahiert.

Hier bietet es sich an, noch einmal auf den Begriff der Ähnlichkeit zurückzukommen. Zwei Schachstellungen sind einander umso ähnlicher, in je mehr Merkmalen sie übereinstimmen. Da die Konfigurationsklassen und die Empfehlungen aufgrund taktischer Überlegungen festgelegt werden und ihre Befolgung gewissermaßen die Taktik des Spielers bestimmt, ist es gerechtfertigt, von *taktischer Ähnlichkeit* zu sprechen.

Hat der Computer eine anwendbare Regel gefunden, muss er entscheiden, ob er sie anwendet oder nicht und gegebenenfalls, welchen der angebotenen Züge er ausführt. Dazu muss er die möglichen resultierenden Stellungen bewerten, was, wie wir wissen, nur sehr bedingt möglich ist. Wiederum ist es die Komplexität, an der eine allgemeine, praktikable Lösung scheitert. Die Güte machbarer Regelanwendungen hängt von der Rechengeschwindigkeit des Computers ab.

Die Regelanwendung kann dadurch beschleunigt werden, dass die Regeln nicht adressiert, sondern strukturell gespeichert werden. Eine Regel hat die uns von der Implikation und von den Entscheidungstabellen her bekannte Form eines Wenn-dann-Satzes: "Wenn die und die Bedingungen gegeben sind, dann führe die und die Aktion aus". Regelwissen kann also als *Entscheidungstabelle* formuliert und beispielsweise als ROM-Schaltung realisiert werden (vgl.Kap.12.3.4 und Formel (12.5)). Wenn die Empfehlung der Erfahrungsregel (die Aktion der Entscheidungsregel) mehrere Züge enthält, sind sie alle dem entsprechenden Leiter der ROM-Schaltung einzuprägen, doch nur eine ist jeweils auszuwählen, eventuell nach Bewertungsmerkmalen.

9 Sowohl den Empfehlungen als auch den Konfigurationsklassen können *Bewertungsmerkmale* zugeordnet werden, z.B. das Merkmal "vorteilhaft" oder "gefährlich". Der eingemauerte König war ein Beispiel für eine gefährliche Konfiguration. Gefährliche Konfigurationen können zu einer Klasse zusammengefasst werden. Jede Stellung, die Element dieser Klasse ist, wäre als gefährlich zu bewerten. Nach diesem Rezept können *Bewertungsalgorithmen* entworfen werden.

Wir wollen nun auf den oben angedeuteten gedanklichen Brückenschlag zum Markovalgorithmus und zum analytischen Rechnen eingehen. Die hervorstechende

Gemeinsamkeit, das Suchen in einem Suchgraph, war bereits erwähnt worden. Jetzt kommt es uns in erster Linie auf die Verschiedenheiten der drei Methoden an.

Wenn eine Regel/Formel auf eine Zeichenkette (einen analytischen Ausdruck, eine Spielstellung) anwendbar ist, kann eine Transformation der Zeichenkette (des Ausdrucks, der Stellung) vorgenommen werden, die in der Regel/Formel angegeben ist. Dabei darf man im Falle des analytischen Rechnens aus allen anwendbaren Formeln eine beliebige auswählen. Diese Freiheit lässt einem der Markovalgorithmus nicht. Dagegen geht die Freiheit beim Schachspielen noch weiter als beim analytischen Rechnen. Man ist nicht an die Regeln gebunden, sondern hat bei der Wahl seines Zuges volle Entscheidungsfreiheit und darf seiner *Phantasie* völlig freien Lauf und seine eigene *Intuition* "zum Zuge" kommen lassen.

Der Schachspieler ist also nur an die Spielregeln gebunden, nicht an die Erfahrungsregeln. Blinde Befolgung der Erfahrungsregeln könnte sogar gefährlich werden, denn die Vernachlässigung aller nicht betrachteten Figuren birgt die Gefahr in sich, dass die Anwendung einer Regel zu einer ungünstigen Stellung führt. Der erfahrene Spieler unterliegt dieser Gefahr kaum, denn er hat das gesamte Brett im Blick und erkennt, ob es zweckmäßig ist, eine Regel anzuwenden oder nicht.

Die Annahme liegt nahe, dass die Überlegenheit des Menschen über den Schachcomputer und allgemein der natürlichen über die künstliche Intelligenz in der Entscheidungsfreiheit, also letzten Endes in der *Willensfreiheit* des Menschen liegt, und dass die Überlegenheit eine prinzipielle ist, weil der Computer keine Willensfreiheit besitzt. Er kann nur Programme ausführen.

Die "Unfreiheit" des Computers kann durch Einbau eines Zufallszahlengenerators aufgehoben werden. Man lässt den Computer den nächsten Zug "würfeln". Doch werden seine Gewinnchancen dadurch eher verringert als erhöht. Der Computer wird dadurch nicht intelligenter, denn die gewonnene Freiheit ist nichts anderes als *Unabhängigkeit*, sie ist keine *Willensfreiheit* , denn sie kann nichts wollen, sie kann sich keine Ziele stellen. Der Computer gewinnt keine *intuitive* Intelligenz, denn ***Intuition*** *ist immer* ***zielgerichtet***.

Menschliche Intuition hat aber nicht nur mit der *Existenz* eines Zieles zu tun, sondern auch damit, dass der Mensch sich seine Ziele frei wählen kann. Er kann auch verlieren wollen. Er kann "wollen, was er will", alles, was ihm seine *Phantasie* eingibt. Damit öffnet sich neben der interiorisierten Erfahrung eine weitere Quelle der Intuition, die *Phantasie*, und das Wort erlangt diejenige Bedeutung, in der es umgangssprachlich vorwiegend benutzt wird, beispielsweise hinsichtlich des künstlerischen Schaffens, aber auch hinsichtlich des Schachspielens. Schach ist eine *Kunst*. Damit geht unsere Frage in eine andere über: *Lässt sich Phantasie simulieren?* Wir lassen diese Frage im Raume stehen, fügen aber noch eine Bemerkung an. Die Begründung der Überlegenheit des Menschen über den Computer durch die Willensfreiheit wird gegenstandslos, wenn man in der Willensfreiheit keinen objektiven Tatbestand, sondern eine subjektive Überzeugung sieht.

4. Selbstproduziertes Regelwissen. Im Vergleich zum übernommenen Regelwissen spielt selbstproduziertes Regelwissen - ebenso wie selbstproduziertes Fallwissen - für das Computerschach eine untergeordnete Rolle. Bis ein bestimmter Computer (ein Computer mit installiertem Schachprogramm) das im Laufe von Jahrhunderten angesammelte Regelwissen selber produziert haben könnte, ist er lange "gestorben", d.h. sein Programm ist zum alten Eisen geworfen worden. Dennoch wollen wir uns der Vollständigkeit halber überlegen, wie sich dieser Weg simulieren ließe und welche Schwierigkeiten dabei auftreten könnten, denn für die KI ganz allgemein spielt selbstproduziertes Regelwissen durchaus eine Rolle.

Wir fragen: Wie kann aus Fallwissen Regelwissen abgeleitet werden? Das Problem ist zwar schwieriger als die zuvor besprochenen Qualifikationswege, doch spielt die Zeit, die benötigt wird, um aus Fallwissen Regelwissen abzuleiten, eine zweitrangige Rolle, da neue Regeln nicht online (während des Spiels) abgeleitet werden müssen. Der Computer kann sich damit beschäftigen, wenn er "Zeit hat". Das Gleiche gilt für das Aneignen fremden Wissens.

Wir wollen einen Algorithmus entwerfen, der aus einer Menge von Fällen eine Regel extrahiert oder "abstrahiert". Wir werden sehen, dass das Wort *abstrahieren* hier durchaus am Platze ist. Beim Entwurf gehen wir davon aus, dass eine Regel einer Reihe von Stellungen ein und dieselbe Empfehlung zuordnet. Demzufolge beginnt der Algorithmus zweckmäßigerweise mit der Wahl einer Empfehlung aus dem Fallwissen. Sodann muss er diejenigen Stellungen, die zu dieser Empfehlung führen, auflisten und aus ihnen diejenigen herausfinden, die eine bestimmte Konfiguration enthalten. Wenn ihm das gelingt, hat er eine Regel gefunden. Aber nach welcher Konfiguration soll er suchen?

Bei der Beantwortung dieser Frage kann der Programmierer dem Computer dadurch helfen, dass er typische Bestandteile von Konfigurationen vorgibt, z.B. "Eigener König steht in einer Ecke" oder "Eigener König und gegnerischer Turm stehen auf eigener Grundlinie" oder "Eigener König ist eingemauert". Es sind solche Teilkonfigurationen auszuwählen, die zwar noch zu arm sind, um eine Empfehlung angeben zu können, die aber für den erfahrenen Spieler ein Achtungszeichen darstellen.

Ausgerüstet mit einer Liste derartiger *beachtenswerter* (gefahren- oder erfolgsträchtiger) Teilkonfigurationen kann der Computer nun das Fallwissen nach diesen durchsuchen, zunächst ohne die Empfehlungen zu berücksichtigen. Findet er für eine Teilkonfiguration mehrere Fälle mit ein und derselben Empfehlung, kann er sie zu einer Regel zusammenfassen. Findet er viele Fälle, aber mit unterschiedlichen Empfehlungen, kann er die Teilkonfiguration um ein Feld oder eine Figur vergrößern und untersuchen, ob in der zuvor gefundenen Untermenge des Fallwissens eine kleinere Untermenge von Fällen enthalten ist, welche die vergrößerte Teilkonfiguration enthalten und außerdem alle zu ein und derselben Empfehlung führen. Ist die Suche erfolgreich, kann eine Regel formuliert werden.

Nebenbei sei darauf hingewiesen, dass die Hinzunahme weiterer Felder oder 11
Figuren als Hinzunahme weiterer Merkmale aufgefasst werden kann, dass es sich also um *Präzisieren* im Sinne von Bild.5.4 handelt, d.h. um eine begriffsbildende Operation.

Man könnte nun die Idee des Vorausspielens mit der Idee der beachtenswerten Konfigurationen kombinieren, indem man das Vorausspielen auf solche Züge beschränkt, die zu einer erfolgsträchtigen Konfiguration führen bzw., falls die eigene Stellung eine gefahrenträchtige Konfiguration enthält, diese entschärfen. Damit würde man dem Vorgehen des Menschen sicher näher kommen. Man könnte sich viele andere Suchstrategien einfallen lassen. Doch wollen wir es genug sein lassen und an dieser Stelle das Erfinden von Schachalgorithmen abbrechen.

Unsere algorithmischen Erfindungen können als Bestätigung folgender oft geäußerten und intuitiv plausiblen Behauptung aufgefasst werden: *Was man verstanden hat, kann man simulieren*. Es bedarf allerdings der Präzisierung, was die Worte "Etwas verstehen" genau bedeuten. Bezüglich der Computerschachintelligenz und der künstlichen Intelligenz überhaupt bedeuten sie soviel wie *"Zurückführung einer intelligenten Leistung auf einen Mechanismus"*. Dabei bezeichnet das Wort Mechanismus eine *endliche Kette diskreter* (genauer *kausaldiskreter) Schritte, die von einem Computer ausgeführt werden können*. Mit diesen Präzisierungen ist die ursprüngliche Behauptung nicht nur plausibel, sondern zu einer Selbstverständlichkeit geworden.

Es würde naheliegen, die Behauptung durch ein "nur" zu verschärfen: *Nur was man verstanden hat, kann man simulieren.* Dieser Satz folgt aus keiner unserer Überlegungen, und tatsächlich trifft er in dieser allgemeinen Form nicht zu. Das folgt bereits aus dem Wenigen, was über neuronale Netze gesagt wurde. Wir kommen darauf noch einmal in Kap.21 zurück.

Unser Erfinden von Schachalgorithmen veranlasst den einen oder anderen Leser vielleicht, selber erfinderisch zu sein. Sicherlich wird er nachfühlen können, welchen Spaß es machen muss, Schachprogramme zu entwickeln, und wie *unheimlich* (wörtlich gemeint) spannend es sein muss, an einem Schachprogramm zu basteln, das versierte Schachspieler besiegt, vielleicht sogar eines Tages Schachweltmeister wird. Ein Sieg eines Computers über einen Meister des Schachspiels ist immer ein Sieg folgsamer Pedanterie und exakter Befehlsausführung über die freie Phantasie. Darin liegt nichts Ungewöhnliches. Die Evolution scheint sogar solche Siege zu lieben, speziell die kulturelle Evolution.

Aus der langen Geschichte des Computerschach zieht Nievergelt in dem oben erwähnten Artikel [Nievergelt 96] folgende Lehre: *"Aber eine Weisheit möge man sich merken. Hunderte von äußerst kompetenten, hochmotivierten Hackern und Forschern haben ein halbes Jahrhundert gebraucht, um künstliche Schachexperten zu bauen, die an menschliche Spitzenleistung herankommen. Dabei ist der Wissensbereich "Schach" eher kleiner, homogener als derjenige vieler "Expertensysteme" (oder "Novizensysteme"?) für kommerzielle Anwendungen. Man darf also nicht*

erwarten, dass ein kleines Team von Programmierern in kurzer Zeit ein nützliches Expertensystem hervorzaubern kann."

Diese Gegenüberstellung von Aufwand und erreichter sehr partieller künstlicher Intelligenz eines Schachcomputers ist ein Schlag gegen alle Euphorie hinsichtlich Expertensystemen und hinsichtlich der KI überhaupt. Nievergelts Schlussfolgerung zeigt, wie weit wir von einer künstlichen Intelligenz entfernt sind, die es mit der menschlichen Intelligenz in voller Breite aufnehmen kann.

Abschließend soll noch einmal der entscheidende Punkt herausgestellt werden, in dem der Mensch dem Schachcomputer überlegen ist, ohne dabei auf so schwierige Begriffe wie Phantasie und Willensfreiheit Bezug zu nehmen. Der Mensch besitzt die Fähigkeit, beim Denken sehr viele Fakten gleichzeitig im Bewusstsein zu halten und zu berücksichtigen. Er denkt *global*, sein Denken ist *komplex*; es erfasst die Dinge im Überblick und trifft Entscheidungen gewissermaßen von einem höheren Gesichtspunkt aus. *Der Mensch denkt in Komplexen.* Vielleicht gewinnt dieser Satz an assoziativer Kraft, wenn das Wort "Komplex" durch das vielschichtige deutsche Wort "Gestalt" ersetzt wird: *Der Mensch denkt **in Gestalten**, er denkt gestalthaft*. Er *gestaltet* eine Schachpartie, er *gestaltet* eine Konfiguration und erkennt eine Konfiguration *gestalthaft*, als Gestalt. Im Gegensatz dazu kann der Computer eine Schachstellung nur feldweise "betrachten" und nur computerwortweise bearbeiten, denn er
12 denkt algorithmisch. Verkürzt aber prägnant formulieren wir: *Der Computer denkt in **Computerwortfolgen**, der Mensch kann **anschaulich** und **in globalen Zusammenhängen** denken*.

Aus dieser Feststellung könnte der Schluss gezogen werden, dass die künstliche Intelligenz für alle Zeiten weit unter dem Niveau der natürlichen Intelligenz bleiben wird. Tatsächlich gibt es für eine solche Schlussfolgerung keinen zwingenden Grund. Zum einen haben wir gesehen, zu welchen erstaunlichen Leistungen der Prozessorcomputer durch die ständige Weiterentwicklung seiner Hard- und Software bereits
13 befähigt worden ist; und diese Entwicklung ist in keiner Weise abgeschlossen. Zum anderen wissen wir aus Kap.9.2.2, dass der genannte Unterschied zwischen dem Denken des Menschen und dem "Denken" des Computers zwar hinsichtlich des Prozessorcomputers, nicht aber hinsichtlich des Neurocomputers besteht. Insbesondere die Lernfähigkeit des Neurocomputers [9.11] gibt Anlass zu einer optimistischeren Sicht auf die Zukunft der KI.

18 Evolution der Programmiersprachen

Zusammenfassung

Sprachen, in denen sich Menschen sowohl mündlich als auch schriftlich zum Zwecke des Informationsaustausches artikulieren können, heißen *Humansprachen*. Humansprachen sind Laut- und meistens gleichzeitig Schriftsprachen. Sie sind bidirektional, d.h. zwischen zwei Kommunikationspartnern in beiden Richtungen verwendbar. Programmiersprachen sind unidirektionale Schriftsprachen. Humansprachen wie Programmiersprachen sind *lineare* (eindimensionale) Sprachen, d.h. als komposite Zeichenrealeme werden ausschließlich Zeichenketten verwendet. Die Linearität der Humansprachen ist die Ursache für eine Diskrepanz zwischen Denken und Sprechen, denn der Mensch denkt vorwiegend bildhaft und netzorientiert. Das entspricht der geometrischen Struktur und kausalen "Vernetzung" der Welt. *Der Mensch denkt netzorientiert und spricht satzorientiert*.

Zwischen dem vernetzten Original und dem vom modellierenden Menschen gedachten vernetzten Modell, dessen Träger ein *neuronales Netz* (das Gehirn des modellierenden Menschen) ist, liegt eine linearsprachliche, satzorientierte Schicht. Dieser Sachverhalt liegt auch beim Modellieren mit Hilfe des Computers vor, nur ist der Träger des vernetzten Modells kein neuronales, sondern ein *boolesches Netz*, die zentrale Computerhardware. Die satzorientierte Schicht zwischen Original und Modell ist in diesem Fall die Maschinenebene, wobei die Sätze der linearen Sprache die Befehle der Maschinensprache sind. Die satzorientierte Schicht heißt *satzorientierte Schnittstelle* oder anschaulicher *linearsprachlicher Flaschenhals des sprachlichen Modellierens*. Durch den engen Flaschenhals muss sich die Artikulierung einer netzorientierten (z.B. räumlichen) Vorstellung satzweise "hindurch zwängen", wobei die Schnittstelle den Informationsfluss zwischen Denken und Sprechen in ähnlicher Weise behindern kann wie der von-neumannsche Flaschenhals den Informationsfluss zwischen Hauptspeicher und Prozessor behindert.

Eine starke Triebkraft der Evolution der Programmiersprachen ist das Bestreben der Programmierer, sich von den Einschränkungen der Maschinensprache zu befreien und die semantische Lücke zwischen Gedachtem und Programmiertem zu schließen. Das sichtbarste Ergebnis ist die Herausbildung des *funktionalen* und des *logischen* (relationalen) *Programmierparadigmas*. Funktionale bzw. logische Programmiersprachen erleichtern das "Umcodieren" funktional bzw. relational formulierter Modelle, die von funktional bzw. relational denkenden Modellierern erstellt (erdacht, evtl. aber noch nicht artikuliert) worden sind, in eine Programmiersprache.

Eine zweite Triebkraft der Entwicklung ist der Wunsch nach Erhöhung der Ausdrucksstärke, der semantischen Dichte von Programmiersprachen. Die *internsemantische Dichte* eines programmiersprachlichen Eingabetextes ist das Verhältnis

der Anzahl der auszuführenden Maschinenbefehle zur Länge des Textes (Anzahl der Bits oder Lexeme). Damit lässt sich die Idee der *Begriffsbildung* als Mittel der semantischen Verdichtung auf Programmiersprachen übertragen, wobei es sich um *internsemantische* Verdichtung handelt.

Die semantische Verdichtung von Programmiersprachen und Programmen beruht auf komponierender und klassifizierender Abstraktion hinsichtlich Operationen (*prozedurale Abstraktion*) und Operanden (*Datenabstraktion*). Ein wichtiger Aspekt der prozeduralen Abstraktion ist das "data hiding", das Schützen prozedureigener Daten gegen den Zugriff durch andere Prozeduren. Das Wort "*Kapselung*" bringt dieses Anliegen anschaulich zum Ausdruck. Speziell auf *klassifizierender* Abstraktion beruht die Idee der "*Vererbung*". Sie besteht darin, in einer Klassenhierarchie von Prozeduren auf jeder Hierarchieebene nur dasjenige explizit zu programmieren, was beim Abstieg in der Hierarchie hinzukommt, also die zusätzlichen (als Vorschrift formulierten) "Merkmale", die beim Übergang von der Oberklasse zur Unterklasse die Präzisierung ausmachen. Die Merkmale (Vorschriften) der Oberklasse werden an alle Unterklassen vererbt.

Die Vereinigung von Kapselung und Vererbung haben zur Herausbildung des Sprachelements "Objekt" und des *objektorientierten* Programmierparadigmas geführt. Ähnlich wie ein Denkobjekt (Idem) ein relativ abgeschlossener Bewusstseinsausschnitt ist, so ist ein *Objekt* ein relativ abgeschlossener Programmteil, der mit anderen Objekten in Wechselwirkung treten, z.B. Aufträge (*Direktiven*) ausführen kann, ohne seine Identität zu verlieren. Die gemeinsame Eigenschaft von Objekten und Operatoren ist die Fähigkeit, Netze zu bilden, Netze kooperierender Partner oder Akteure. Betrachtet man ein aus dem Netz herausgelöstes Objekt, einen isolierten Akteur, so führt dieser eine Folge von Direktiven aus, die den "Imperativen" (Befehlen) imperativer Sprachen entsprechen. Das objektorientierte Paradigma vereinigt in sich Vorteile der Datenflussprogrammierung einerseits und der Aktionsfolgeprogrammierung andrerseits.

18.1 Der Flaschenhals des linearsprachlichen Modellierens

Zunächst vergegenwärtigen wir uns noch einmal, was in diesem Buch unter "Sprache" und "sprachlichem Modellieren" verstanden wird. Im Untertitel des Buches ist mit sprachlichem Modellieren jedes *codierende* Modellieren gemeint. *Ein* ***sprachliches Modell*** *eines Originals ist dessen Beschreibung mit Hilfe codierender Zeichen*, wobei "Beschreibung" nicht auf *Schreiben* und "sprachlich" nicht auf *Sprechen* eingeschränkt ist. Sprachliche Modelle können Sätze einer natürlichen oder künstlichen Sprache sein, z.B. Formeln, es können auch zweidimensionale, man sagt auch *ebene* (in der "Zeichenebene" dargestellte) Gebilde sein wie Graphen, chemische Strukturformeln oder Bilder.

Eine Sprache dient der Artikulierung von *Aussagen über die Welt*. Eine "Aussage" sagt etwas über ein oder mehrere Objekte aus, indem sie ihnen Merkmalswerte (Attribute) zuordnet. Im Falle von Bildern kann die Zuordnung quasi uncodiert, d.h. anschaulich erfolgen. Extensional ist eine Sprache als Menge von auditiven oder visuellen *Kompositzeichen* definiert, die aus elementaren Zeichen nach den syntaktischen Regeln der Sprache gebildet werden. Im Rahmen der Sprachlehre und auch umgangssprachlich wird der Sprachbegriff i.Allg. enger gefasst, indem als Kompositzeichen nur Zeichenketten zugelassen werden. Eine Zeichenkette ergibt sich durch *Verkettung* (durch Voranstellen oder Anhängen, d.h. durch Komponieren "entlang einer gedachten *Linie*") von Zeichen oder Zeichenketten. Aus diesem Grunde spricht man von *eindimensionalen* oder **linearen** Sprachen.

Nach dieser Wiederholung führen wir für die folgenden Überlegungen den Begriff der "Humansprache" ein und vereinbaren: *Als **Humansprache** wird eine Sprache bezeichnet, in der sich Menschen sowohl mündlich als auch schriftlich zum Zwecke des Informationsaustausches artikulieren können, die also als **Lautsprache** (**auditive** Sprache) und als **Schriftsprache** (**visuelle** Sprache) verwendet wird.* Eine Humansprache dient denjenigen, die sie beherrschen, als zweiseitiges oder **bidirektionales** Kommunikationsmittel , d.h. jeder kann Mitteilungen *senden* und *empfangen*, jeder kann sich in der Sprache *artikulieren* und jeder kann sie *interpretieren*.

Demgegenüber ist eine *Programmiersprache* ein einseitiges oder **unidirektionales** Kommunikationsmittel. In ihr artikulieren Menschen sprachliche Operatoren (Operationsvorschriften). Der Computer interpretiert Programme, die ihm in einer Programmiersprache, die er "versteht", zur Ausführung übergeben werden. Er artikuliert sich aber nicht in der Programmiersprache.

Der Leser hat sicherlich richtig verstanden, was hier mit "Interpretieren durch den Computer" gemeint ist, nämlich das Zuordnen *maschineninterner* Semantik zu einer empfangenen Zeichenkette. Ursprünglich hatten wir den Begriff des Interpretierens als "Interpretieren durch den Menschen" definiert, und zwar als Zuordnen von *externer* Semantik (Zuordnen eines *Idems*) zu einer empfangenen Zeichenkette (einem *Zeichenrealem*) und das "Artikulieren durch den Menschen" als Zuordnen eines Zeichenrealems zu einem Idem (vgl. Bild 2.1).

Dementsprechend ist konsequenterweise unter "*Artikulieren* durch den Computer" das Zuordnen einer Ausgabezeichenkette zur *maschineninternen* Semantik, d.h. zu einem codierenden internen Zustand zu verstehen, was natürlich möglich ist. Insofern könnte man sagen, dass der Computer sich artikulieren kann. Doch wäre das irreführend. Der Computer artikuliert sich nicht in dem Sinne, dass er "seinen", mit *externer* Semantik behafteten Idemen Zeichenrealeme zuordnet. Sein "Artikulieren" ist lediglich ein Überführen aus seiner internen in eine externe *Darstellung*. Was der Computer ausgibt, legt i.d.R. der Programmierer fest. Wenn dieser einen umgangssprachlichen Satz (mit externer Semantik) einprogrammiert, kann die Illusion entstehen, der Computer "denke in externer Semantik" und artikuliere *sich* umgangssprachlich (vgl. das Fallexperiment in Kap.15.5 [15.7]).

Trotz dieses gravierenden Unterschiedes zwischen Human- und Programmiersprachen besitzen sie eine fundamentale Eigenschaft gemeinsam. *Humansprachen wie Programmiersprachen sind* ***lineare Sprachen***. Auf den ersten Blick scheint das eher eine Selbstverständlichkeit als eine "fundamentale" Eigenschaft zu sein. In Kap.18.3 wird sich zeigen, dass die Linearität den wohl wichtigsten Motor für die Evolution sowohl der Humansprachen als auch der Programmiersprachen darstellt. Zunächst fragen wir nach dem *Grund* der Linearität.

Aus der Sicht des Menschen ist die Linearität von *Programmiersprachen* notwendig, damit er bei der Kommunikation mit dem Computer seine humansprachlichen Artikulierungsgewohnheiten beibehalten kann. Diese Forderung lag letzten Endes unserer sehr speziellen Zielstellung zugrunde, ein Gerät zur Verarbeitung von *Bitketten* zu konstruieren. Warum aber sind *Humansprachen* linear? Der Grund ist offenbar in der Funktionsweise des Hörapparates zu suchen. Dieser registriert den "*eindimensionalen*" Verlauf, des Druckes, den die Luft auf das Trommelfell ausübt. Die "einzige Dimension" ist die Zeit. Auf dieser Grundlage kann sich nur eine *eindimensionale* Sprache entwickeln. Demgegenüber registriert der Sehapparat eine zeitliche Folge ebener (genauer sphärischer, jedenfalls zweidimensionaler) "Bilder", die von der Linse auf die Netzhaut projiziert werden.

Das Fehlen geometrischer Dimensionen bei der auditiven Perzeption der Welt (abgesehen von einem sehr groben Stereohören) ist nicht etwa ein Mangel des Hörapparates, sondern hat prinzipielle physikalische Ursachen, die mit der großen Länge von Schallwellen (im Vergleich zur Länge von Lichtwellen) zusammenhängen.

Mit dem Menschen, seinen Sinnesorganen, seinem Bewegungsapparat und seinem Gehirn hatte die Evolution die Möglichkeit der Entwicklung von Sprachen geschaffen. Infolge der physikalischen und physiologischen Voraussetzungen entwickelten sich zunächst auditive (gesprochene) Sprachen und einfache visuelle, in erster Linie gestische Zeichensprachen. Aus den *eindimensionalen* auditiven Sprachen gingen später (unter Einbeziehung des visuellen und motorischen Apparates) *eindimensionale* Schriftsprachen hervor, indem Folgen gedachter Objekte (Idemfolgen) und später Phonemfolgen in Zeichenfolgen "übersetzt" wurden, wobei die einzige Dimension nicht mehr die *zeitliche*, sondern *eine räumliche* Dimension ist. Da die Zeichen selber i.d.R. ebene (zweidimensionale) Gebilde (Schriftzüge auf einer Schreibebene) darstellen, bezeichnet man Schriftsprachen besser als *linear* und nicht als eindimensional.

Hier soll eine Zwischenbemerkung eingeschoben werden, die über den eigentlichen Gegenstand des Buches hinausgeht. Eine Humansprache kann als Lautsprache und als Schriftsprache verwendet werden. Daran scheint nichts Verwunderliches zu sein. Tatsächlich ist dieser Doppelcharakter ein erstaunliches Phänomen. Denn um es zu erklären, muss man annehmen, dass das Gehirn sowohl mit statischer als auch mit dynamischer Codierung umgehen kann [9.1]. Denn *auditive Sprachen verwenden* ***dynamische*** *Codierung, während Schriftsprachen* ***statische*** *Codierung verwenden.*

Die Linearität der Humansprachen ist die Ursache für eine Diskrepanz zwischen Denken und Sprechen, auf die etwas ausführlicher eingegangen werden soll, da sie dem Menschen nur relativ selten zum Bewusstsein kommt, nämlich dann, wenn er sich "nicht ausdrücken kann".

Die hervorragende Bedeutung der visuellen Perzeption der Welt durch den sehenden Menschen hat dazu geführt, dass die *Bildhaftigkeit* des "Eindrucks", den die Welt auf die Netzhaut macht, sich auf das Denken übertragen hat. *Der Mensch denkt vorwiegend* ***bildhaft***, d.h. in zweidimensionalen, eventuell auch in dreidimensionalen Vorstellungen.

Eine erste Verarbeitung der Sinneseindrücke durch das Gehirn führt zur Heraustrennung einzelner Objekte, so dass sich die Welt dem Menschen als Menge von Objekten darstellt, zwischen denen räumliche und zeitliche Beziehungen bestehen. Infolge einer weiteren Verarbeitung, in die gewissermaßen die gesamte Menschheitserfahrung eingeht, stellt sich die Welt dem Menschen, seinem analytischen Verstande, als *Netz* von Objekten dar, zwischen denen Ursache-Wirkungs-Beziehungen bestehen. Die Welt ist *kausal vernetzt*. Mit "kausal vernetzt" soll zum Ausdruck gebracht werden, dass die Objekte der Welt gegenseitig aufeinander *wirken*, sodass die Ereignisse in der Welt *zeitliche Ursache-Wirkungsfolgen* bilden. Der Mensch "denkt" die Welt (modelliert die Welt gedanklich) als räumliches und kausales Netz. Das bringen wir mit dem Satz zum Ausdruck: *Der Mensch denkt* ***netzorientiert***. Die kausale Modellierung impliziert die Dimension der Zeit. 1

Ob die kausale Vernetzung eine Eigenschaft der Welt oder eine Eigenschaft des menschlichen Denkens ist, sei dahingestellt. Jedenfalls sind wir Menschen gewohnt, die Welt als kausales Netz anzusehen. Das Denken in Ursache-Wirkungs-Zusammenhängen liegt nicht nur dem naturwissenschaftlichen, sondern jedem Modell der Welt, jedem "*Weltbild*" zugrunde, auch einem religiösen, mystischen, magischen oder esoterischen.

Das Wort "vernetzt" steht als *bildhafter* Ausdruck für "*miteinander in Beziehung stehend*". Im vorangehenden Kapitel haben wir das Denken des Menschen *gestalthaft* genannt. In diesem vielsinnigen Wort finden die Wörter "bildhaft" und "netzorientiert" eine geglückte inhaltliche Synthese mit hoher praktischer Relevanz, wie die Bezeichnung "*Gestaltpsychologie*" zeigt.

Während der Mensch netzorientiert denkt, spricht er "linear". Da die *semantische Einheit* von Humansprachen der Satz ist, kann man prägnant, aber nicht ganz exakt (s.u.) sagen: *Der Mensch denkt* ***netzorientiert*** *und spricht* ***satzorientiert***. Zwischen Denken und Sprechen besteht also ein charakteristischer Unterschied und die *interne* Codierung des Denkens unterscheidet sich wesentlich von der *externen* Codierung des Sprechens.

Die Überführung von dem einem in den anderen Code ist Aufgabe des Sprachzentrums. Dem externen Artikulieren von Vorstellungen muss ein i.Allg. unbewusstes satzorientiertes internes Artikulieren, also ein Denken in Sätzen vorausgegangen sein. Es ist also nicht verwunderlich, dass sich mit der Zeit auch ein *bewusstes Denken*

und *Nachdenken in Sätzen* herausgebildet hat. Der Mensch kann also nicht nur netzorientiert, sondern auch satzorientiert denken. Der Überführung interncodierter (gedachter) in externcodierte (gesprochene oder geschriebene) Sätze entspricht der Pfeil 3 in Bild 2.1, wobei "Zeichenidem" und "Zeichenrealem" zu "Satzidem" und "Satzrealem" zu konkretisieren sind.

Die Ergebnisse psychologischer Experimente sprechen dafür, dass der Träger des gestalthaften, netzorientierten Denkens vorzugsweise in der rechten und der Träger des satzorientierten Denkens vorzugsweise in der linken Gehirnhälfte liegt, wo sich normalerweise das Sprachzentrum befindet. Die Überführung des gestalthaft Gedachten in gesprochene Sprache kann auf Schwierigkeiten stoßen, mit denen das im Unterbewusstsein arbeitende Sprachzentrum nicht fertig wird und die es "an das Bewusstsein delegiert". Man *merkt* dann, dass man "sich nicht ausdrücken kann", dass es schwierig ist, seine Vorstellungen (das gestalthaft Gedachte) in Worte zu fassen, d.h. linearsprachlich zu artikulieren. In solchen Fällen bedient man sich gerne einer Zeichensprache, etwa indem man Stift und Papier nimmt und "zeichnet" oder indem man den Körper, z.B. die Hände "sprechen" lässt. Man denke an die Wendelbewegung, die ein Mensch mit seiner Hand ausführt, um zu verdeutlichen, was eine Wendeltreppe ist.

Dass sich die lineare Umgangssprache eventuell als recht unbeholfen erweist, ist nicht verwunderlich, insbesondere beim Artikulieren komplexer Vorstellungen. Denn das vorgestellte Objekt, z.B. eine Wendeltreppe, wird zwar gedanklich als Ganzes erfasst, muss aber sprachlich als Folge von Sätzen beschrieben werden. Bildlich gesprochen muss beim *Artikulieren* eine komplexe ganzheitliche Vorstellung, ein gestalthaftes ("voluminöses") Objekt zu einem langen dünnen Strang, zu einer Folge von Sätzen verformt werden, die ihrerseits Folgen von Morphemen oder Buchstaben sind.

Dieses Bild vor Augen sprechen wir metaphorisch vom *Nadelöhr* oder noch anschaulicher vom **Flaschenhals des linearsprachlichen Modellierens**. Das Wort "Flaschenhals" soll - ähnlich wie im Falle des von-neumannschen Flaschenhalses - die Vorstellung wecken, dass der Inhalt einer Flasche durch den Hals hindurch *geschleust* werden muss, um "nach außen" treten und verwendet werden zu können, und dass er dabei aus einer voluminösen Form zu einem dünnen Strahl, Strang oder Strom verformt werden muss. Der "lange, dünne Strom" der Rede verlässt den Sprechenden durch den "Hals" (den physischen "Flaschenhals" Kehlkopf und Mund) und tritt durch das Ohr des Hörenden (den physischen Flaschenhals des Empfängers) in den Flaschenbauch (das Gehirn), wo es wieder eine "voluminöse" Form annimmt, d.h. durch Interpratation zu einer ganzheitlichen, gestalthaften Vorstellung (z.B. zur Vorstellung einer sich wendelnden Treppe) wird. Aus einem "Rinnsal" von Worten wird ein "Meer" von Vorstellungen.

2 Neben der *räumlichen* (geometrischen) *Komplexität* des Gedachten gibt es einen zweiten Grund, Gedanken mehrdimensional, speziell zweidimensional zu artikulieren, die *logische Komplexität*. Das sogenannte logische Denken, z.B. Rechnen oder

Schlussfolgern, vollzieht sich vorzugsweise linearsprachlich, auch dann, wenn es *nicht* extern, sondern gedanklich artikuliert wird. Dies ist der Grund dafür, dass logisches Denken zuweilen als schwierig oder unangenehm (weil unanschaulich) empfunden wird.

Hinzu kommt, dass logisches Denken in größeren Zusammenhängen eine große Kapazität des Kurzzeitgedächtnisses erfordert. Denn alle Objekte und alle Beziehungen zwischen ihnen müssen zugriffsbereit im Bewusstsein gehalten werden, und der Zugriff erfolgt nicht über räumliche Vorstellungen, wie beim vorstellenden Denken, sondern über gedachte Namen (interne Codierungen; vgl. Bild 2.1). Wenn dies allzu schwierig wird, benutzt man gerne einen externen Speicher, z.B. Papier und Stift und codiert extern, wobei die Papierebene netzorientiertes Codieren erlaubt. Ein Beispiel hierfür ist der Verwandtschafts*graph* von Bild 16.1. Er unterstützt das Zurechtfinden in der Verzwicktheit oder Vertracktheit (in der logischen Komplexität) der linearsprachlichen Beschreibung des verwandtschaftlichen Beziehungs*netzes*.

Schließlich gibt es einen dritten Grund für die Verwendung einer ebenen Sprache, die *kausale Komplexität*. Auch hier leisten graphische Darstellungen gute Dienste. Ein Beispiel sind die Operatorennetze der USB-Methode. Die Methode dient der "anschaulichen" zweidimensionalen, sprachlichen (codierten) Modellierung von Ursache-Wirkungsbeziehungen, die auch dann, wenn sie hohe Komplexität aufweisen, "überschaubar" bleiben. Die netzorientierte *kausale* Vorstellungs- und Beschreibungsweise war an den unterschiedlichsten Beispielen illustriert worden, in Bild 8.3 am Beispiel eines Fertigungsprozesses, in Kap.13.7 am Beispiel des Straßenverkehrs [13.15] und der Strömung von Flüssigkeiten [13.16], und in Kap.15.7 am Beispiel eines Unternehmens [15.11], das man sich als Operatorenhierarchie, also als übereinander geschichtete Netze vorstellen kann. In Kap.8.2.2 hatten wir eine Methode kennen gelernt, die es gestattet, die kausale Vernetzung der Welt auf sehr abstrakter Ebene mit Hilfe von *Petrinetzen* zu beschreiben.

Bisher war zwischen dem von-neumannschen Flaschenhals und dem Flaschenhals des linearsprachlichen Modellierens lediglich eine rein metaphorische Analogie zu erkennen. Es besteht aber eine ganz konkrete Analogie. Um sie herauszuarbeiten, geben wir zunächst eine genaue Bestimmung des bereits verwendeten Begriffs der *semantischen Einheit*. *Eine Zeichenkette, die als Objekt-Merkmals-Zuordnung, also als Aussage interpretierbar ist, heißt* ***semantische Einheit****. Eine semantische Einheit heißt* ***syntaktisch vollständig****, wenn alle Objekte, die an der Merkmalszuordnung beteiligt sind, explizit genannt werden*. Die semantischen Einheiten von Humansprachen sind Sätze. Die semantischen Einheiten von Maschinensprachen sind Befehle.

Hiergegen könnte eingewendet werden, dass die Definition der semantischen Einheit nur auf Aussagesätze zutrifft. Darauf wäre zu erwidern, dass auch Imperativsätze bzw. Maschinenbefehle und auch Fragesätze Objekt-Merkmals-Zuordnungen, also Aussagen artikulieren. Imperativsätze bzw. Maschinenbefehle *schreiben Zuordnungen vor*, Fragesätze *fragen nach Zuordnungen*. Insofern trifft die Definition auf jeden Satz und jeden Maschinenbefehl zu.

Die Analogie, auf die wir hinauswollen, beruht darauf, dass die semantischen Einheiten sowohl maschinensprachlicher als auch natürlichsprachlicher Ausdrücke syntaktisch vollständig sind. Ein Befehl nennt die beteiligten Objekte, also die Ein- und Ausgabeoperanden, durch Angabe ihrer Bezeichner bzw. Adressen. Ein Satz nennt sie durch entsprechende Satzglieder; es sind i.d.R. Substantive oder Pronomen. Sowohl Humansprachen als auch Maschinensprachen befolgen die Regel, dass semantische Einheiten syntaktisch vollständig sein müssen[1]. Angesichts dieser Gemeinsamkeit vereinbaren wir: *Syntaktisch vollständige semantische Einheiten, also humansprachliche Sätze und Maschinenbefehle werden unter der Bezeichnung* ***Satz im weiten Sinne (i.w.S.)*** *zusammengefasst*.

Die Forderung syntaktischer Vollständigkeit mag überraschen, lässt sich aber plausibel erklären. Eine ihrer Ursachen liegt in beiden Fällen in der Speicherung, im Falle der Humansprachen in der Kapazität des Kurzzeitgedächtnisses und im Falle der Maschinensprachen in der Zentralisierung der Speicherplätze im Hauptspeicher, die hardwaremäßig zu der Verbindung zwischen HK und K in Bild 13.7, d.h. zum von-neumannschen Flaschenhals führt und die softwaremäßig das Programmieren in Aktionsfolgen verlangt. Damit ist die zunächst rein metaphorische Analogie zwischen dem von-neumannschen und dem linearsprachlichen Flaschenhals zu einer konkreten Analogie geworden, die im materiellen Träger ihre Ursache hat.

Auch die Metapher vom "Flaschen*bauch*" lässt sich vom materiellen Träger her deuten, in welchem aus dem eintretenden "Rinnsal" von Sätzen bzw. Befehlen ein "Meer" von Wirkungen wird. Die Wirkungen im "Flaschenbauch" sind die Prozesse, die im Gehirn bzw. in der Zentraleinheit (CPU) ausgelöst werden. Beide "Flaschenbäuche" sind *Netze*, entweder *Neuronennetze* oder *boolesche Netze*.

Hinsichtlich der syntaktischen Vollständigkeit sei daran erinnert, dass auch für Algorithmen im ursprüngliche Sinne, d.h. für *imperative* Algorithmen, die syntaktische Vollständigkeitsforderung gilt. Schließlich sei noch erwähnt, dass in Humansprachen Pronomen die Rolle von Subjekten und Objekten übernehmen können. Das setzt eine eindeutige Zuordnung zwischen Nomen und Pronomen und ein entsprechendes Kurzzeitgedächtnis voraus. Ähnlich verfahren Maschinensprachen. Die Rolle des Kurzzeitgedächtnisses können der Akkumulator oder andere CPU-Register übernehmen. Zeiger auf diese Register können mit Pronomen in humansprachlichen Sätzen verglichen werden.

Unsere ursprüngliche Idee war es, das Komponieren von Operatoren von der Hardware in die Software zu übernehmen, d.h. in die Software hinein fortzusetzen. Der linearsprachliche Flaschenhals vereitelt die unmittelbare Verwirklichung dieser Idee. Das hatten wir bereits in Kap.13.5.3 erkannt und mit dem bedingten Sprungbefehl einen Ausweg gefunden, der die Fortsetzung des Komponierens ermöglicht,

---

1 Dabei ist von sog. *Ellipsen* abgesehen; das sind Auslassungen von Redeteilen humansprachlicher Sätze, die der Interpretierer "automatisch" ergänzt.

wenn auch nicht in Form von Operatorennetzen, so doch in Form verzweigter Aktionsfolgen, konkreter in Form von Maschinenprogrammen, welche bedingte Sprungbefehle enthalten, oder allgemeiner in Form imperativer Programme mit bedingten Anweisungen. Der Preis für die softwaremäßige Fortsetzung des Komponierens ist die Aufgabe des Netzparadigmas zugunsten des imperativen Paradigmas. Dieser Paradigmenwechsel ist in Kap.13.7 ausführlich besprochen worden.

Hinsichtlich des Modellierens "der vernetzten Welt" steht der Programmierer vor der gleichen Schwierigkeit wie jeder Mensch, der Zusammenhänge der "vernetzten Welt" in Worte fassen will. Zwischen dem vernetzten Original und dem vernetzten Modell, dessen Träger entweder ein boolesches Netz oder ein neuronales Netz sein kann, liegt eine linearsprachliche, und zwar eine *satzorientierte* Schicht. Das Wort "satzorientiert" bedeutet hier, dass die Bausteine dieser Schicht Sätze im weiten Sinne sind. Die Evolution der Programmiersprachen ist von dem - als Selektionsdruck wirkenden - Bestreben der Programmierer geprägt, sich von den Einschränkungen zu befreien, die ihm das satzorientierte Artikulieren auferlegt. Diese Entwicklung wollen wir in großen Zügen nachvollziehen. Dabei können wir auf den Einsichten dieses Kapitels aufbauen, die wir noch einmal zusammenfassen.

Es hat sich folgendes Bild ergeben. Zwischen der *externen* Netzstruktur der Außenwelt und der *internen* Netzstruktur des Computers bzw. des Gehirns liegt eine *linearsprachliche, satzorientierte* Schicht. Sie ist für die humansprachliche Modellierung der Welt ebenso notwendig wie für die Modellierung durch Prozessorrechner (Simulation). Wir nennen sie ***satzorientierte Schnittstelle*** oder anschaulicher ***linearsprachlichen Flaschenhals*** *des sprachlichen (d.h. codierenden) Modellierens.*

18.2 Semantische Verdichtung

In Kap.5.1 hatten wir uns Gedanken über die Evolution natürlicher Sprachen gemacht. Der Versuch liegt nahe, unsere dortigen Einsichten auf Programmiersprachen zu übertragen. Dies ist angesichts der Unterschiede in den Ursachen und in den Mechanismen der Evolution der beiden Sprachklassen sicher nur sehr bedingt möglich. Humansprachen (natürliche Laut- und Schriftsprachen) sind unter dem Evolutionsdruck in Richtung höherer Überlebenschancen entstanden, ohne dass der Mensch sie *bewusst* "konstruiert" hätte. Programmiersprachen sind das Produkt *bewusster, zielgerichteter, konstruktiver* Tätigkeit der Menschen, wobei auch hier ein Druck in Richtung höherer "Überlebenschancen" eine Rolle gespielt hat und weiterhin spielt, besser gesagt, in Richtung höherer Durchsetzungschancen im wirtschaftlichen oder beruflichen Wettbewerb.

Die Evolution von Sprachen ist ein Beispiel dafür, wie sehr die Evolution durch die Teilnahme der menschlichen Intelligenz am Evolutionsprozess beschleunigt wird. Das betrifft nicht nur die Programmiersprachen, sondern jede Sprache, an deren Entwicklung der Mensch bewusst teilnimmt. Dabei handelt es sich um eine opera-

tionale Zirkularität. Die Intelligenz, also die Fähigkeit zum sprachlichen Modellieren, schmiedet sich ihr eigenes Werkzeug, die Sprache. Dieser Prozess ist im Laufe der Menschheitsgeschichte immer mehr aus dem unbewussten in den bewussten Bereich menschlicher Tätigkeit aufgestiegen.[2]

Der Umstand, dass ein und derselbe physiologische Apparat (das Gehirn) Sprache *benutzt* und Sprache *schafft*, lässt erwarten, dass trotz aller Unterschiede zwischen der Evolution von Humansprachen und Programmiersprachen Ähnlichkeiten bestehen, denn ein Sprachentwickler denkt über die Programmiersprache und deren Syntax, die er definieren will, in "*seiner eigenen*" (durch lebenslange Benutzung "zu eigen" gewordenen, interiorisierten) Humansprache nach.

Es gibt noch andere Gründe, die Ähnlichkeiten erwarten lassen. So wird von universellen Programmiersprachen ebenso wie von Humansprachen verlangt, dass ihr Umfang endlich ist und dass ihre Artikulationen räumlich und zeitlich endlich sind, dass mit ihrer Hilfe aber trotzdem "unendlich viel" ausgedrückt werden kann, dass "unendlich viele" Originale sprachlich modelliert werden können. Schließlich müssen programmiersprachliche Ausdrücke ebenso wie humansprachliche eindeutig und schnell interpretierbar sein.

In Kap.5.1 hatten wir die "Reaktion" der Evolution von Humansprachen auf die genannten Forderungen durch drei Evolutionsprodukte charakterisiert:

- Begriffsbildung,
- Grammatik,
- Idemobjektivierung.

Da programmiersprachliche Ausdrücke an sich (ohne menschlichen Interpretierer) keine externe, sondern nur maschineninterne Semantik besitzen, entfällt die Idemobjektivierung, denn interne Semantik *ist* durch den technischen Träger objektiviert. Die Rolle der Grammatik von Programmiersprachen ist in den Kapiteln 16.4 und 16.5 behandelt worden. Es stellt sich die Frage, ob die Begriffsbildung für Programmiersprachen eine ähnliche Bedeutung besitzt wie für Humansprachen.

Es scheint so, als müsse die Frage verneint werden, denn humansprachliche Begriffsbildung ist an externe Semantik gebunden. Ein wesentlicher "Zweck" der Begriffsbildung ist die Erhöhung der Ausdrucksstärke, der *semantischen Dichte*. Die ***semantische Dichte*** eines Textes war in Kap.5.5 [5.14] (nicht sehr exakt) definiert worden als *Umfang des im Mittel pro Buchstaben (oder auch pro Wort) assoziierbaren Gedächtnisinhaltes* (der assoziierbaren Denkobjekte oder Ideme). Man erinnere sich an die Wörter "Tierreich", "Erde", "Gott", die in Kap.5.5 als Beispiele für Wörter mit hoher semantischer Dichte genannt wurden.

Da die Begriffe "Idem" und "Denkobjekt" nicht exakt definiert sind, gilt dies auch für den Begriff der semantischen Dichte. Das ändert sich, wenn man den Begriff auf Programmiersprachen überträgt. Das ist möglich, wenn unter semantischer Dichte

2 Siehe z.B. [Klix 80].

nicht *extern*semantische, sondern *intern*semantische Dichte verstanden wird. Die ***internsemantische Dichte*** eines programmiersprachlichen Eingabetextes lässt sich definieren als die *mittlere Anzahl der Maschinenbefehle pro Bit (oder auch pro Lexem) des Eingabetextes*, die bei dessen Abarbeitung (Interpretation) ausgeführt werden. Damit lässt sich auch die Idee der Begriffsbildung auf Programmiersprachen übertragen. Sie dient der *internsemantischen Verdichtung*.

Zu diesem Zwecke werden durch Abstraktion neue Begriffe gebildet, die sich nun aber nicht auf die zu modellierende Welt, sondern auf die modellierende Sprache beziehen und diese beschreiben. Die Begriffsbildung betrifft also, ebenso wie die Begriffsbildung der Schulgrammatik, *metasprachliche* Begriffe. In Kap.5.5 war das Vorgehen am Beispiel des Syntaxbaumes eines deutschsprachigen Satzes (siehe Bild 5.2) veranschaulicht worden. In der Backus-Naur-Form der Syntaxregeln einer Sprache treten die metasprachlichen Begriffe als *metasprachliche Variablen* auf, z.B. "*Satz*" in (5.1) oder "*Befehl*" in (5.2). Jetzt wollen wir den Weg verfolgen, den die Entwickler von Programmiersprachen beim Erfinden neuer Sprachkonstrukte und neuer metasprachlicher Begriffe gegangen sind.

In einem humansprachlichen oder programmiersprachlichen Text entsprechen den metasprachlichen Begriffen bzw. Variablen konkrete Wörter oder Konstruktionen (Sprachkonstrukte), z.B. der AcI in (16.2) [16.13] oder die WHILE-Anweisung in Bild 15.3b. Der Weg zum abstrakten metasprachlichen Begriff beginnt mit der *Erfahrung*, dass gewisse komplexe Operationen oder Operanden häufig in ähnlicher Weise auftreten. Wenn derartige Fälle erkannt sind, kann es zweckmäßig sein, ein kompakteres (ausdrucksstärkeres, *semantisch dichteres*) Sprachelement zu definieren, beispielsweise eine Laufanweisung anstelle eines Zyklus, wie die Programme der Bilder 15.2 und 15.3 demonstrieren. Wenn ein auf diese Weise entstandenes Sprachkonstrukt als Sprachelement in eine Programmiersprache aufgenommen wird, erscheint in der Syntax der Sprache eine neue *metasprachliche Variable*.

Es ist aber auch eine semantische Verdichtung durch den Programmierer selbst (den Benutzer einer gegebenen Programmiersprache) möglich. Viele Sprachen erlauben dem Programmierer, sich für den eigenen Gebrauch Konstrukte, die von der Sprache nicht zur Verfügung gestellt werden, selber zu definieren und Bezeichner für sie zu vereinbaren. Das gilt insbesondere für Kompositoperationen (Operationsvorschriften). Wenn der Bezeichner einer solchen selbstdefinierten Operation in einem Programm seinerseits als Name eines *Programms* auftritt, sprechen wir von *Unterprogramm*. Dann kann auf das Resultat der Operation über einen speziellen Bezeichner zugegriffen werden. Wenn der Bezeichner in einem Programm (evtl. auch in einem an sich imperativen Programm) als Name einer *Funktion* auftritt, sprechen wir von *Unterfunktion*. Der Bezeichner stellt dann den Wert der Unterfunktion, also das Resultat der Operation dar. Unterprogramme und Unterfunktionen können über ihre Namen eventuell auch von anderen Anwendern aufgerufen werden.

Die beschriebene Art der semantischen Verdichtung von Operationsvorschriften beruht auf *komponierender Abstraktion* (vgl. Bild 5.5) bezogen auf das Komponieren

sprachlicher Operatoren. In Kap.15.7 [15.12] war dafür der Begriff der **prozeduralen Abstraktion** eingeführt worden. Diese Bezeichnung beruht auf folgendem Sachverhalt. Ein Programmierer, der für eine Operation, beispielsweise für die Multiplikation von Matrizen, ein Programm (eine Prozedur) geschrieben hat, darf für die Prozedur einen Bezeichner, z.B. MATRMUL, vereinbaren und im Weiteren wie einen Operationscode verwenden. Wenn er davon beim Programmieren Gebrauch macht, *abstrahiert* er von den Datails der Operation. Der Bezeichner steht für einen "schwarzen Kasten", dessen Inhalt nicht interessiert, sondern nur das, was er ausgibt.

Der Leser vergleiche das Vorgehen mit Bild 5.2. Dort handelt es sich um die Einführung metasprachlicher Variablen für die metasprachliche Beschreibung von Sätzen der deutschen Sprache. In Kap.5.5 hatten wir erkannt, dass die grammatikalische Begriffsbildung (die Einführung metasprachlicher Begriffe) auf *komponierender* und *klassifizierender* bzw. *generalisierender* Abstraktion beruht. Die komponierende Abstraktion ist in Bild 5.2 in senkrechter, die klassifizierende (generalisierende) in waagerechter Richtung dargestellt. Weiter unten werden wir sehen, dass auch bei der Entwicklung von Programmiersprachen nicht nur die komponierende, sondern auch die klassifizierende Abstraktion eine wichtige Rolle spielt. Zunächst wollen wir unser Verständnis für die Rolle der komponierenden Abstraktion erweitern und vertiefen.

Auf einer sehr hohen Komponierungs- und Abstraktionsebene kommen *Anwenderprogrammpakete* zur Anwendung, die dem Computeranwender fachspezifische Unterstützung anbieten. Beispielsweise kann ein Dachdeckermeister seinen Computer beauftragen, alle Berechnungen durchzuführen, die mit der Deckung eines Daches zusammenhängen, von der Materialbestellung bis zum Ausdruck der zu bezahlenden Rechnung. Dabei benutzt er eine *ebene* (zweidimensionale) Sprache. In einem *Fenster* auf dem Bildschirm bietet der Computer *Aktionen* in Form von Symbolen (meist kleinen Bildchen) an. Die Gesamtheit der angebotenen Aktionen heißt **Menü**. Die gewünschten Aktionen ruft der Anwender durch "*Mausklick*" auf, beispielsweise die Berechnung der Dachfläche und anschließend die Berechnung der erforderlichen Anzahl an Dachziegeln. Dazu "klickt" er mit der Maus der Reihe nach die betreffenden Symbol an, d.h. er führt mit der Maus den Mauszeiger, einen mit der Maus verschiebbaren Lichtpunkt, auf eines der Symbole und drückt die Maustaste. Auf diese Weise kann er ein "Aktionsfolgeprogramm programmieren". Ganz ähnlich verfährt ein Statistiker, der ein Statistikprogramm benutzt, oder ein Schriftsteller, der ein Textverarbeitungsprogramm benutzt.

Das *Fensterprinzip* als Kommunikationsmethode zwischen Mensch und Maschine hat sich in weiten Bereichen der Computeranwendung durchgesetzt, nicht zuletzt durch das *Windows-Betriebssystem* (Window = Fenster). In der Regel vereinigt ein Fenster in sich die Funktion eines Menüangebots und eines Fragebogens, in den der Nutzer über die Tastatur seine Eingaben "einträgt".

Wenn die Erteilung eines Auftrags an den Computer über ein Fenster durch Mausklick oder durch Eintragen einer Antwort besteht, ist es kaum noch gerechtfer-

tigt, von einer *Programmier*sprache zu reden, denn der Nutzer programmiert nicht den Computer, sondern er wird vom Computer *abgefragt*. Auch die Bezeichnung *Dialog*sprache für die Kommunikation nach dem Fensterprinzip zwischen Mensch und Computer ist unpassend, weil sie die Vorstellung wecken kann, dass beide Partner ihre Repliken *selber artikulieren*, was für den Computer nicht zutrifft. Passender wäre die Bezeichnung *Abfragesprache* oder, falls die Eingabe ausschließlich per Mausklick erfolgt, *Menüsprache*. Es existiert allerdings keine scharfe Grenze zwischen Programmier- und Abfragesprache, und häufig kommen sie gemeinsam zur Anwendung. Beide Sprachklassen fassen wir unter dem Begriff "**Auftragssprache**" zusammen. Diese Bezeichnung ist bereits in Kap.16.4 [16.12] eingeführt worden.

Auftragssprachen und speziell Programmiersprachen können nach ihrem *Komponierungsgrad* klassifiziert werden, genauer nach dem maximalen Komponierungsgrad der Operationen, für welche die Sprache einen Bezeichner bzw. ein Menüsymbol enthält. Beispielsweise muss der Komponierungsgrad einer Auftragssprache für die Arbeit mit einem umfassenden Mathematiksystem, das u.a.. Differenzialgleichungen lösen kann, sehr hoch sein (siehe dazu Kap.19.4 [19.4] und das Kapitel "Computeralgebrasysteme" in [Bronstein 95]). Im Laufe der Jahre sind Programmiersprachen mit immer höheren Komponierungsgraden entwickelt worden. In diesem Zusammenhang wurde der Begriff der *Sprachgeneration* in Analogie zur Rechnergenerationen vorgeschlagen. Er beruht auf einer sehr abstrakten Analogie und hat sich nicht durchsetzen können.

Dennoch enthält diese abstrakte Generationenanalogie zwischen Hardware- und Softwarekomponierung einen *realen* Kern und zwar die Tendenz zur Spezialisierung. Sie geht naturgemäß mit der Herausbildung immer komplexerer Kompositoperatoren einher. Softwaremäßig erfolgt sie durch fortschreitende prozedurale Abstraktion auf immer höherer Komponierungsebene, hardwaremäßig durch die Realisierung immer komplexere Operationen als mikroelektronische Schaltungen. Jeder Operator, der sich *sprachlich* komponieren lässt, kann auch *real*, d.h. als Schaltung komponiert werden (man denke an die ROM-Hierarchie [13.7]). Viele Rechner verfügen z.B. über eine Schaltung für die Multiplikation von Matrizen. Sowohl für das reale (schaltungsmäßige) als auch für das sprachliche Komponieren gilt, dass mit dem Komponierungsgrad in der Regel auch die Spezialisierung zunimmt, dass also der Anwendungsbereich des Computers bzw. der Sprache eingeschränkt wird. Beispielsweise wird einem Statistiker wahrscheinlich wenig mit einem Vektorrechner gedient sein, der auf das Lösen partieller Differenzialgleichungen spezialisiert ist. Ebenso wenig wird einem Elektriker ein Programmpaket nützen, das für Dachdecker konzipiert ist.

Besonders sinnfällig ist die Generationenalogie im Falle der Bezeichnung "vierte Generation", die sich aber weder für Rechner noch für Sprachen durchgesetzt hat. In den 70er Jahren wurde vorgeschlagen, Computer, die Operatoren hoher Komponierungsstufe hardwaremäßig zur Verfügung stellen, als Rechner der vierten Generation

zu bezeichnen. Ein solcher Rechner stellt ein Operatorennetz aus vielen Prozessoren oder kompletten Computern (sog. Workstations) mit variabler Kopplungsstruktur des Netzes dar. Er arbeitet massiv parall. Dieser Computertyp kommt gegenwärtig wegen seines günstigen Preis/Leistungsverhältnisses zunehmend zum Einsatz. Es werden Rechenleistungen bis nahe an 1 Tflops (10^{12} Gleitkommaoperationen pro Sekunde) erreicht. Derartige Systeme werden als Spezialrechner für ganz bestimmte Aufgaben entworfen, beispielsweise für die Wettervorhersage. In Analogie dazu wurde ebenfalls in den 70er Jahren vorgeschlagen, Sprachen, die für bestimmte Anwendungsgebiete (Diskursbereiche) spezielle Operationen hoher Komponierungsstufe zur Verfügung stellen, als Sprachen der vierten Generation zu bezeichnen.

Von den Maschinensprachen über die soeben charakterisierten “Sprachen der vierten Generation” bis zu den Menüsprachen erstreckt sich ein Spektrum von Sprachen mit steigendem Komponierungsgrad. Man beachte, dass der Begriff des Komponierungsgrades nicht an ein bestimmtes Programmierparadigma gebunden ist und nicht nur für imperative Sprachen Sinn hat. Zwar stellen die gängigen funktionalen Sprachen - abgesehen von einigen Ausnahmen - keine hochkomponierten Funktionen zur Verfügung, dafür aber elegante sprachliche Mittel zur Komponierung von Funktionen, die sich mehr oder weniger deutlich an die Idee des Lambda-Kalküls von A. CHURCH anlehnen (siehe Kap.8.4.7). Das gilt insbesondere für die Sprache *Lisp*. Eine Ausnahme bildet u.a. die Sprache APL, eine funktionale Programmiersprache relativ hohen Komponierungsgrades.

Demgegenüber existieren viele *logische* Sprachen mit hohem Komponierungsgrad, z.B. Anfragesprachen von Datenbanksystemen, die gleichzeitig die Möglichkeit demonstrieren, dass nicht nur die Operationen, sondern auch die Operanden hoch komponiert sein können. Das trifft z.B. für Merkmalswertetupel, Datensätze und Dateien zu, die alle die Rolle von Operanden spielen können.

Man könnte nun erwarten, dass die Sprachevolution nicht nur für Operationen, sondern auch für Daten metasprachliche Begriffe auf der Grundlage komponierender Abstraktion hervorgebracht hat. Tatsächlich hat sich als Pendant zur prozeduralen Abstraktion der Begriff der **Datenabstraktion** herausgebildet. Wir wollen uns überlegen, worum es sich dabei handeln könnte.

Aus der Sicht eines “Rechners”, der mit Zahlen rechnet, kann es sich offensichtlich nur um “Zahlenabstraktion” handeln. Jeder, der sich ein wenig mit Mathematik oder Programmieren beschäftigt hat und der sich fragt, worin Abstraktion bezüglich Zahlen bestehen kann, wird wahrscheinlich sofort an die *Klassen* der ganzen Zahlen, der rationalen und der reellen Zahlen denken. Der “Datentyp integer” (Klasse der ganzen Zahlen) und der “Datentyp real” (Klasse der reellen Zahlen; “real” wird ebenso wie “integer” englisch ausgesprochen) sind so ziemlich das Erste, was ein Programmieranfänger lernt.

Wie man sieht, handelt es sich bei dieser Art von Datenabstraktion nicht um komponierende, sondern um *klassifizierende* Abstraktion, also um diejenige Abstraktion, die in Bild 5.2 in waagerechter Richtung dargestellt ist. Als Klassifizieren

hatten wir das Zusammenfassen verschiedener Objekte zu einer *Klasse* bezeichnet, die in bestimmten Merkmalen übereinstimmen. Bei der klassifizierenden Abstraktion hinsichtlich Daten spricht man i.Allg. nicht von Klassen, sondern von *Typen*, genauer von **Datentypen**. In der Typangabe ist alles enthalten, was der Computer wissen muss, um mit "*typisierten*" (oder "*getypten*") Daten umzugehen, u.a. um für sie den erforderlichen Speicherplatz bereitzustellen. Typisierung bewirkt semantische Verdichtung.

Bevor wir das Zusammenspiel klassifizierender und generalisierender Abstraktion in der Evolution der Programmiersprachen genauer analysieren, fassen wir unsere bisherigen Einsichten in folgender Feststellung zusammen: *Die **semantische Verdichtung** von Programmiersprachen und Programmen kann durch komponierende Abstraktion hinsichtlich Operationen (Prozeduren) und klassifizierende Abstraktion hinsichtlich Operanden (Daten) erreicht werden*. Doch wird damit die Komplexität der Sprachevolution nicht voll erfasst. An der *prozeduralen* Abstraktion kann nämlich auch *Klassifizieren* und an der *Datenabstraktion* kann auch *Komponieren* beteiligt sein. Die Ursache hierfür liegt in der inhaltlichen Verknüpfung von Operationen und Operanden. Dieser Sachverhalt tritt bei mathematischen Operationen sehr anschaulich zutage.

Wenn der Computer als "hochqualifizierter Rechner", als mächtiges mathematisches Werkzeug benutzt werden soll, ist es zweckmäßig, eine Sprache mit hohem Komponierungsgrad zur Verfügung zu stellen, die es gestattet, nicht nur Operationen mit booleschen oder numerischen Werten (Zahlen) und Variablen, sondern auch mit Vektoren, Matrizen, Tensoren, Tupeln, Aussagen, Prädikaten oder Mengen in kompakter Form auszudrücken, wie es in der Mathematik üblich ist. Die dafür notwendige prozedurale Abstraktion kann freilich nur dann vom Anwender effektiv genutzt werden, wenn sie durch eine entsprechende Datenabstraktion begleitet wird. Die erforderlichen Datentypen stellen häufig *Kompositdaten* dar, die gemeinsam mit den Kompositoperationen definiert werden, wie z.B. die komplexen Zahlen gemeinsam mit ihrer Addition und Multiplikation.

Das Gleiche gilt für die Komposition *nicht*mathematischer Operationen und Daten, wie das Beispiel der Datenbanken zeigt. Datensätze und Dateien sind *Kompositdaten*, die ohne entsprechende Kompositoperationen (Suchen, Aktualisieren) wenig Sinn hätten. Doch obwohl sie durch Komponierung entstehen, beruht die Definition entsprechender syntaktischer Variablen, also die Definition des Datentyps "*Datensatz*" oder "*Datei*", auf *klassifizierender* Abstraktion. Das gleiche gilt für die Definition des Datentyps "*komplexe Zahl*", "*Array*" oder "*Menge*". Das klassische Beispiel ist der Datentyp "*record*", den NIKLAUS WIRTH mit der Definition seiner Sprache *Pascal* eingeführt hat und der nichts anderes darstellt als einen Datensatz-Typ.

Die Möglichkeit, Komposit*daten* zu Klassen (Typen) zusammenzufassen, wirft die Frage auf, ob sich in Analogie dazu auch aus Komposit*operationen* Klassen (Typen) bilden lassen. Offenbar ist das möglich. Beispielsweise können die Additio-

nen ganzer bzw. reeller bzw. komplexer Zahlen, hinter denen sich unterschiedliche konkrete Operationen verbergen, zur "*Klasse der Additionen*" zusammengefasst werden. Wo es Sinn hat, kann man auch die Addition von Vektoren und von Matrizen hinzunehmen, eventuell sogar auch die "Addition" (Disjunktion) boolescher Werte oder die "Addition" (Vereinigung) von Mengen.

Genau genommen handelt es sich dabei nicht um Klassifizieren im Sinne von Bild 5.5 (Übergang vom Exemplar zur Klassen), sondern um *Generalisieren* (Übergang Unterklasse - Oberklasse), denn die Addition stellt bereits eine Klasse dar, deren Exemplare die Additionen konkreter Summanden sind. Die Generalisierung kann fortgesetzt werden, indem beispielsweise alle Operationen mit ganzen oder reellen Zahlen zur Klasse der *arithmetischen Operationen* zusammengefasst werden. Auf diese Weise kann sich eine *Hierarchie* von Operationstypen ergeben. Hinsichtlich *Daten*typen kann man analog verfahren. Ganze Zahlen, reelle Zahlen und komplexe Zahlen stellen drei Datentypen mit zunehmendem **Generalisierungsgrad** dar. Die allgemeinste Klasse von Operationen bzw. von Daten, der wir begegnet sind, hatten wir mit *op* bzw. mit *od* oder *id* bezeichnet; im Rahmen der USB-Methode hatten wir die Bezeichnung *od* und im Zusammenhang mit der syntaktischen Analyse (Kap.16.4) die Bezeichnung *id* (von Identifikator) verwendet.

Generalisierte Klassen (Typen) werden in der Programmierungstechnik - in Abhängigkeit von Realisierungsnuancen - auch als **generische** oder **polymorphe Typen** bezeichnet. Ihre Verwendung bewirkt eine zusätzliche semantische Verdichtung, eine Erhöhung der Ausdrucksstärke der Sprache. Bei der Übersetzung eines Programms, das derartige Typen verwendet, muss das Übersetzerprogramm bei der syntaktischen Analyse feststellen, welcher konkrete Typ im gegebenen Fall vorliegt.

Interessanterweise ist die Evolution der Programmiersprachen *nicht* in Richtung zunehmender Generalisierung von Operationstypen einerseits und Datentypen andrerseits verlaufen, sondern in Richtung ihrer Kombination. Dabei verfolgten die Sprachentwickler zwei Ziele, eine noch effizientere Programmierung durch zusätzliche semantische Verdichtung und gleichzeitig die Sicherung einer fehlerfreien Programmausführung. Das Prozedurkonzept birgt nämlich die Gefahr in sich, dass die Abarbeitungsprozesse verschiedener Prozeduren sich gegenseitig stören. Diese Gefahr besteht immer dann, wenn zwei oder mehrere Prozeduren dieselben Daten benutzen, wenn sie also während ihrer Abarbeitung auf dieselben Speicherplätze zugreifen. Dieses Problem führt uns in die Domäne des *Betriebssystems*, das für die Prozessorganisation zuständig ist. Darum verschieben wir eine ausführlichere Behandlung auf Kap.19.5. Doch soll schon jetzt der Gang der weiteren Entwicklung skizziert werden, die durch zwei Ideen geprägt ist, die mit den Worten *Kapselung* und *Vererbung* benannt werden.

Kapselung. Die erste der beiden Ideen richtet sich gegen die genannte Gefahr der gegenseitigen Störung von Prozessen. Die Idee wird durch den oben erwähnten Umstand nahegelegt, dass Operationen und ihre Operanden nur *gemeinsam* Sinn ergeben und insofern eine Einheit bilden. Wenn dafür gesorgt wird, dass auf dieje-

nigen Daten, mit denen eine Prozedur arbeitet, nur diese Prozedur selber zugreifen kann, lassen sich störende Wechselwirkungen zwischen den Ausführungsprozessen verschiedener Prozeduren vermeiden. Das ist allerdings nicht restlos zu verwirklichen, da zwischen den Prozeduren, die als Bausteine eines oder auch mehrerer Programme dienen, eventuell Daten übergeben werden müssen. Dennoch ist es möglich, dass ein Prozedur ihre Daten weitgehend vor anderen Prozeduren "*verbirgt*" (sog. **data hiding**).

Diese Idee führte zur Erfindung neuer Sprachelemente durch Zusammenfassung von Operanden und Operationen (Daten und Prozeduren) zu programmierungstechnischen Einheiten. Für diese Einheiten wurden je nach Sicht des Erfinders und in Abhängigkeit von Realisierungsnuancen verschiedene Bezeichnungen vorgeschlagen wie **abstrakter Datentyp**, **Datenkapsel** und **Modul**. Der Kern ist in jedem Fall das "data hiding", also das *Schützen* prozedureigener Daten gegen den Zugriff durch andere Prozeduren. Das Wort "*Kapselung*" bringt dieses Anliegen anschaulich zum Ausdruck.

Vererbung. Die zweite Idee besteht darin, in einer *Klassenhierarchie* von Prozeduren auf jeder Hierarchieebene nur dasjenige explizit zu programmieren, was beim Abstieg in der Hierarchie hinzukommt, also die zusätzlichen (als Vorschrift formulierten) "*Merkmale*", die beim Übergang von der Oberklasse zur Unterklasse die *Präzisierung* ausmachen. Die Merkmale (Vorschriften) der Oberklasse werden an alle Unterklassen *vererbt*, wodurch Programmtext eingespart und so ein Verdichtungseffekt erzielt wird. Das Vorgehen soll am Beispiel der Implementierung des Operatorennetzes von Bild 8.1 (der Funktion (8.1)) erläutert werden. 4

Jeder Operator, jeder Flussknoten und jeder Operandenplatz hat seine Ein- und Ausgänge. In jedem Abarbeitungstakt des zu erstellenden Programms muss festgestellt werden, wo welche Operanden übergeben werden, m.a.W. ob und gegebenenfalls welcher Operand einen bestimmten Eingang bzw. Ausgang passiert. Diese Operation kann für die "*Klasse aller Bausteine des Netzes*" einheitlich programmiert werden. Diese allgemeinste *Klasse* wird im folgenden Präzisierungsschritt in die *Klassen* der Operatoren, Flussknoten und Plätze zerlegt. Im nächsten Präzisierungsschritt kann z.B. die Klasse der Operatoren in arithmetische und trigonometrische Operatoren und die Klasse der Flussknoten in starre und steuerbare Flussknoten zerlegt werden.

In jedem Präzisierungsschritt werden jeweils die notwendigen Programmergänzungen programmiert. Im Falle der Präzisierung der Operatoren sind die Ergänzungen Vorschriften zum Addieren, Multiplizieren bzw. zur Berechnung der Sinusfunktion. In Kap.20.3 ist das Vorgehen vollständig ausprogrammiert.

Den entscheidenden Anstoß zur weiteren Sprachevolution lieferte die Idee, Kapselung und Vererbung zu vereinigen und ein entsprechendes neues Sprachkonstrukt, einen neuen *Typ* zu definieren, der die Bezeichnung **Objekt** erhielt. Diese Wortwahl mag ungeschickt erscheinen, denn das semantisch hochgeladene Wort "Objekt" kann irritieren und missverstanden werden. Gegebenenfalls werden wir der Eindeutigkeit 5

halber von *informatischen* bzw. von *umgangssprachlichen* Objekten sprechen. In Kap.18.3 wird sich die Berechtigung der Wortwahl "Objekt" herausstellen. Eine Programmiersprache, die den Typ "*Objekt*" zur Verfügung stellt, heißt **objektorientierte Sprache**.

Ein Objekt ist ein relativ abgekapselter Programmbaustein, der über eigene Prozeduren, **Methoden** genannt, und über einen privaten Speicher verfügt und damit über eigene Daten, mit denen die Methoden des Objekts operieren. Ein Objekt kann als Spezialist aufgefasst werden, der bestimmte Dinge *kann* (bestimmte Methoden
6 beherrscht). Objekte können sich gegenseitig um **Dienstleistungen** bitten. Die einzelnen Objekte eines objektorientierten Programms bilden gewissermaßen ein Netz von Kooperationspartnern, die sich gegenseitig Aufträge erteilen können. Für diese Art der Auftragserteilung wird auch das Wort *Direktive* verwendet, und es wird von *direktivem* statt von *objektorientiertem* Programmieren gesprochen [Schefe 87].

Durch Verwendung einer objektorientierten Sprache kann nicht nur die *Sicherheit*, sondern eventuell auch die *Effizienz* der Ausführung von Programmen erhöht, d.h. die Laufzeit verkürzt werden, zumindest bei geschickter Programmierung. Da ein Objekt seine Methoden und seinen Speicher selbst verwaltet, führt das objektorientierte Programmieren zu einer Dezentralisierung der Steuerung und alle damit verbundenen Vorteile können genutzt werden. Darüber hinaus bedeutet die Einführung des Typs (der metasprachlichen Variablen) "*Objekt*" einen weiteren Schritt zur Lösung des *technischen Semantikproblems*, wie im folgenden Kapitel gezeigt wird.

18.3 Technisches Semantikproblem und Programmierparadigmen

Die charakteristische Schwierigkeit, vor der jeder Nutzer eines Computers steht, liegt in der Notwendigkeit, das eigene Denken, die eigenen Vorstellungen in einer Sprache auszudrücken, die der Computer "versteht". Den Kern dieser Schwierigkeit hatten wir in Kap.15.5 [15.6] in der *semantischen Anbindung* erkannt, d.h. in der Anbindung der *Nutzersemantik* (der *externen* Semantik, in welcher der Nutzer denkt) über eine *formale Semantik* (die Semantik eines Kalküls bzw. einer Programmiersprache) *an die interne Maschinensemantik*, also an die Prozesse, die im Computer ablaufen. Das Problem der semantischen Anbindung hatten wir das "*technische Semantikproblem*" genannt. Bild 18.1 veranschaulicht die Problematik.

Bild 18.1 enthält die gleichen semantischen Ebenen wie die Bilder 5.3 und 8.6. Doch zeigt Bild 18.1 einen anderen Weg auf, der zu den drei Ebenen führt. Ausgangspunkt sind diesmal verschiedene *Sprach*ebenen. In der linken Spalte sind drei Sprachklassen definiert, denen in der rechten Spalte die jeweilige Semantik zugeordnet ist. Eine Sprache, in der ein Analytiker oder ein Modellierer über das Original nachdenkt und ein erstes Modell entwirft, nennen wir **Modelliersprache**. Eine Modelliersprache trägt die *externe* Semantik des Nutzers, d.h. die Interpretationen

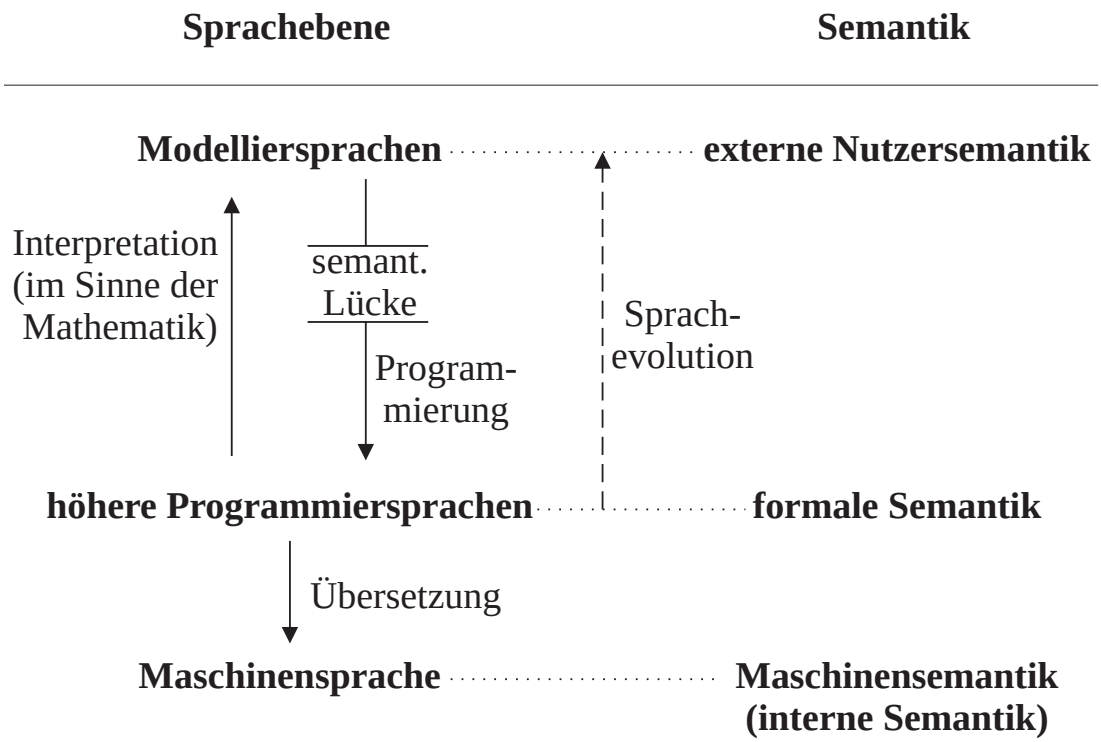


Bild 18.1 Zum technischen Semantikproblem

modelliersprachlicher Ausdrücke sind Vorstellungen (Gedanken, Ideme) des Nutzers. Die Anbindung der externen Nutzersemantik an die interne Maschinensemantik erfolgt normalerweise in zwei Schritten. Im ersten Schritt wird das Modell von einem Programmierer in einer höheren Programmiersprache ausprogrammiert, wodurch das Modell von der oberen in die mittlere Ebene überführt wird. Dabei erfolgt die Anbindung der externen Semantik des Nutzers an die formale Semantik der Programmiersprache. Im zweiten Schritt wird das Programm von einem Übersetzerprogramm in die Maschinensprache übersetzt, wodurch das Modell von der mittleren in die untere Ebene überführt wird. Dabei erfolgt die Anbindung der formalen Semantik an die interne Semantik der Maschine.

In beiden Schritten können Fehler auftreten, die sehr häufig durch unkorrekte semantische Anbindung verursacht sind. Die Folge ist, dass der Computer nicht das “tut” (rechnet), was der Modellierer “will”, was er seiner Meinung nach in der Modelliersprache artikuliert hat. Eine korrekte semantische Anbindung der externen Semantik an die Maschinensemantik kann nur dann garantiert werden, wenn sowohl das Modell der obersten Ebene (und damit die externe Realität) als auch das Maschinenprogramm der untersten Ebene (und damit die computerinterne Realität) Interpretationen des durch die Programmiersprache der mittleren Ebene definierten Kalküls sind. Jeder Programmierer wird zwar bemüht sein, dieser Forderung auf mehr oder weniger systematische Weise nachzukommen, der Nachweis, dass die Bedingung tatsächlich erfüllt ist, wird aber nur selten erbracht.

Beim Übergang von der obersten zur untersten Ebene müssen nicht unbedingt alle drei Schritte explizit ausgeführt werden. Der zweite Schritt entfällt, wenn der Programmierer in der Maschinensprache programmiert. Der erste Schritt entfällt, wenn Modelliersprache und Programmiersprache zusammenfallen, wenn also die Sprache, in welcher der Modellierer gewohnt ist zu denken und das Gedachte zu artikulieren, implementiert ist. Für ein mathematisch formuliertes Modell ist das durchaus möglich. Denn es existieren umfangreiche Mathematiksysteme, deren Eingabesprachen (Auftragssprachen) weitgehend an mathematische Sprachen angelehnt sind, sodass der Modellierer sein Modell praktisch in der ihm gewohnten Sprache "programmieren" und unmittelbar vom Rechner ausführen lassen kann. Falls eine *physikalische Theorie* implementiert ist, erübrigen sich möglicherweise beide Schritte des semantischen Anbindens.

Auch der umgekehrte Weg ist möglich, der darin besteht, dass nicht die Modelliersprache an die implementierte Kalkülsprache angelehnt wird, sondern dass die Kalkülsprache an die - evtl.verbale - Modelliersprache angepasst wird. Der **Fuzzy-Kalkül** macht diesen Weg gangbar. Er erlaubt es, dem Computer teilweise verbale (man sagt auch "linguistische") Aussagen mit unscharfer Semantik anzubieten. In Kap.21.3.2 wird die Idee der Methode skizziert.

Unsere Überlegungen zeigen, dass der Begriff des sprachlichen Modells in recht unterschiedlichen Bedeutungen verwendet wird. Wir hatten ihn sehr allgemein als eine *durch idealisierende Abstraktion vereinfachte Beschreibung eines Originals* eingeführt [2.1], wobei die Beschreibung aus Aussagen über Merkmale des Originals besteht. Um die Unterschiede in der Verwendung dieses Begriffs deutlicher herauszuarbeiten, wollen wir die Rolle der mittleren Ebene beim Modellieren ins Auge fassen und untersuchen, was die Kalkülisierung genau beinhaltet und wie der Kalkül in das Modell eingeht.[3] Um uns nicht im Abstrakten zu verlieren, betrachten wir die folgenden konkreten Modelle.

1. Computermodell eines Produktionsbetriebes (Kap.15.7)
2. Modell verwandschaftlicher Beziehungen (Kap.16.1)
3. Modell der Kohlenwasserstoff-Moleküle (Kap.16.3)
4. Mathematisches Modell des Planetensystems (Kap.4.2).

Geht man die vier Modelle der Reihe nach durch, erkennt man, dass der jeweils verwendete Kalkül eine immer grundsätzlichere Rolle im Modell spielt; aus einer *punktuellen* Rolle im ersten Modell wird eine *globale* im letzten. Dabei ist unter "punktuell" die unzusammenhängende Interpretation des Kalküls durch einzelne Modellaussagen zu verstehen und unter "global" die ganzheitliche Interpretation des Kalküls durch das Modell und damit durch das Original. Mit der schrittweisen

3 Die Anregung zu den folgenden Überlegungen gab der Artikel "Softwaretechnik und Erkenntnistheorie" von PETER SCHEFE [Schefe 99], in dem der Unterschied zwischen naturwissenschaftlichen und programmiersprachlichen Modellen aus erkenntnistheoretischer Sicht diskutiert wird.

Globalisierung werden die kausalen Zusammenhänge des Originals durch das Modell immer tiefer erfasst. Durch die Interpretation des Kalküls (durchgezogener Aufwärtspfeil in Bild 18.1) werden Aussagen der formalen Ebene mit externer Semantik belegt. Das Gesagte soll anhand der vier genannten Modelle illustriert werden.

Das Computermodell eines Produktionsbetriebes möge der *quantitativen Beschreibung* des Produktionsablaufs dienen, also der zahlenmäßigen (numerischen) Berechnung von Werten interessierender Merkmale. Die prozedurale Abstraktion bzw. die objektorientierte Programmierung gibt die Möglichkeit, das Modell als Operatorenhierarchie zu formulieren, wobei die hierarchische Struktur des Modells derjenigen des Originals entsprechen sollte [15.11]. Darin liegt ein wichtiger Schritt zur Überwindung der semantischen Lücke. Interessierende Merkmale können u.a. Durchlaufzeiten von Werkstücken durch bestimmte Bearbeitungsabschnitte (z.B. durch das Operatorennetz von Bild 8.3.), die Auslastung von Maschinen, benötigte Materialmengen oder der Gewinn sein. Das Modell verwendet den arithmetischen Kalkül. Dieser wird "punktuell" interpretiert und angewendet, d.h. für die Berechnung der verschiedenen interessierenden Größen werden spezielle Programme geschrieben ("erfunden"). Tiefer liegende kausale Zusammenhänge werden nicht erfasst. In diesem Sinne ist die Kalkülisierung "*oberflächlich*". 7

Im Modell des Planetensystems kommt der Kalkül der Analysis zur Anwendung. Er ist bedeutend "*tiefer*" in das Modell des Diskursbereiches eingebunden. Das Modell erfasst die kausalen Zusammenhänge vollständig, in ganzer *Tiefe*. In diesem Sinne sagen wir, dass der **Kalkülisierungsgrad** des Modells maximal ist. Auch in diesem Modell können einzelne Aussagen berechnet werden, z.B. die Umlaufzeit der Erde um die Sonne. Doch muss dafür nicht eine spezielle Rechenvorschrift (Formel) "ad hoc erfunden" werden; vielmehr lässt sie sich aus dem Modell nach den Regeln des Kalküls "ableiten". Das Modell produziert neue Modellaussagen. Das Original (der Diskursbereich) wird "als Ganzes" oder "global" durch das formale Modell erfasst. Grundlage sind die *Erkenntnisse*, die Newton in seinen Prinzipien der Mechanik formuliert hat, die Hypothese der Gravitationskraft und die Erfindung der Differenzialrechnung. Das Modell ist eine *physikalische Theorie*.

Ähnliche Verhältnisse liegen im Modell der Kohlenwasserstoff-Moleküle vor. Grundlage sind die chemischen *Erkenntnisse*, die als Grundwissen bzw. als Axiome [16.8] formuliert wurden. Aus ihnen können alle interessierenden Aussagen über den Diskursbereich, d.h. "alle möglichen" (von den Axiomen erlaubten) Moleküle aus C- und H-Atomen *abgeleitet* werden. Im Gegensatz zum Planetenmodell stellt dieses Modell keine physikalische Theorie dar. Um das zu erreichen, müsste das Faktenwissen über die Wertigkeiten aus der elektromagnetischen Wechselwirkung zwischen C- und H-Ionen abgeleitet werden.

Auch im Modell der Verwandtschaftsbeziehungen können neue Aussagen aus bekanntem Wissen *abgeleitet* werden. Doch beruht das Wissen nicht, wie in den letzten beiden Modellen, auf der experimentellen Bestimmung (Messung) von Merk-

malswerten, sondern auf dem Umstand, dass jeder Mensch Vater und Mutter hat. Im Übrigen beruht es auf Vereinbarungen.

8 Im Modell eines Produktionsbetriebes beruht das Faktenwissen sowohl auf Messungen (z.B. Messung der Dauer einer Operation) als auch auf Vereinbarungen (z.B. Festlegung von Preisen). Das Wissen hat aber in jedem Falle "punktuellen" Charakter, es betrifft bestimmte Merkmale bestimmter Objekte, z.B. bestimmter Werkstücke oder bestimmter Operationen. In den anderen drei Beispielen hat das Wissen "globalen" Charakter für den gesamten Diskursbereich. Beispielsweise gelten die Bewegungsgleichungen für sämtliche Planeten und die Axiome der C-H-Verbindungen gelten für beliebige Kohlenwasserstoffmoleküle.

Aus den Beispielen lässt sich folgende verbale Begriffsbestimmung extrahieren. *Der **Kalkülisierungsgrad** eines sprachlichen Modells ist der Grad der Durchgängigkeit und Geschlossenheit der Kalkülisierung des Modells*. Je umfassender und geschlossener das Modell kalkülisiert ist, umso allgemeiner sind die Modellaussagen. Das bedeutet, dass mit zunehmendem Kalkülisierungsgrad die Aussagen Variablen betreffen, z.B. Aussagen über (irgendwelche) Planeten und deren Koordinaten oder Aussagen über (irgendwelche) Atome und deren Verbindungen oder Aussagen über (irgendwelche) Personen und deren verwandtschaftliche Beziehungen. Das Modell wird also zunehmend *analytisch*. Damit ergibt sich die Möglichkeit, ein Maß für den

9 Kalkülisierungsgrad anzugeben. *Der **Kalkülisierungsgrad** eines Modells ist das Verhältnis von analytischem zu numerischem Rechenaufwand (gemessen in elementaren Rechenschritten) beim Ableiten von Modellaussagen ohne Berücksichtigung des numerischen Lösens von Gleichungen*.

Die Analyse des Grades der Kalkülisierung stellt eine Ergänzung zur Klassifikation von Modellen in Bild 3.1 dar. Dort hatten wir zwischen formalisierten und nichtformalisierten sprachlichen Modellen unterschieden. Inzwischen wissen wir, dass formalisiertes Modellieren stets die Interpretation eines Kalküls durch das modellierte Original beinhaltet. In Bild 3.1 kann also "formalisiert" durch "kalkülisiert" ersetzt werden. Aufgrund unserer Analyse erkennen wir nun, dass die Unterteilung in zwei Modellklassen, formalisierte und nichtformalisierte, der Vielfalt möglicher sprachlicher Modelle nicht voll gerecht wird, dass vielmehr verschiedene Grade der Formalisierung möglich sind. Es läge nahe, von Formalisierungsgrad zu sprechen, doch ziehen wir den ausdrucksstärkeren Begriff des Kalkülisierungsgrades vor.

Damit beenden wir den Abstecher in die Problematik der Modellklassifikation und wenden uns dem Pfeil "Programmierung" in Bild 18.1 zu. Wie bereits erwähnt, entfällt die Programmierschicht, wenn die Modelliersprache implementiert ist. Das ist jedoch die Ausnahme. Normalerweise liegt zwischen der Modelliersprache und der Programmiersprache die in Kap.15.5 ausführlich behandelte *semantische Lücke*, die der Programmierer "überspringen" muss, um die externe *Nutzersemantik* an die Semantik einer maschinenverständlichen Sprache anzubinden. Jede Verringerung der semantischen Lücke erleichtert ihm die Arbeit. Dadurch entsteht ein Selektions-

druck in Richtung des gestrichelten Pfeils in Bild 18.1, der die Sprachevolution vorantreibt.

Das Wort "Selektionsdruck" ist durchaus berechtigt, denn die Phantasie der Spracherfinder brachte eine unübersehbare Menge von Programmiersprachen hervor, die immer noch wächst. In der praktischen Arbeit mit den Sprachen haben sich manche bewährt und sind weiterentwickelt worden, andere sind ausgestorben. Die Entwicklung ging in verschiedene Richtungen, denen unterschiedliche Konzepte und Denkschemata zugrunde liegen. So entstanden unterschiedliche **Sprachparadigmen**. Mit dem Wort "Paradigma" ist in diesem Zusammenhang die konzeptionelle Grundlage einer Sprache gemeint, die einem bestimmten *Denkschema* des Programmierers angepasst ist, z.B. dem Denken in Algorithmen, in Funktionen oder in Operatorennetzen.[4]

Der abstrakte Begriff des Programmierparadigmas hat sich als "Nebenprodukt" der Bemühungen herausgebildet, die semantische Lücke zu veringern und die Programmiersprachen immer besser den Denkgewohnheiten der Programmierer und, wenn möglich, auch der Programmnutzer anzupassen. Dieses Ziel war die Motivation für viele Überlegungen und Ideen der Kapitel 15, 16 und 17. Die Entwicklung *funktionaler* oder *logischer* (relationaler) Programmiersprachen erleichtert das "Umcodieren" *funktional* bzw. *relational* formulierter Modelle (Modelle, die von "funktional" bzw, "relational" denkenden Modellierern erstellt sind) in eine Programmiersprache.

Angesichts der weiten Verbreitung des funktionalen Denkens in Naturwissenschaft und Technik ist die große Bedeutung des *funktionalen* Programmierparadigmas neben dem ursprünglichen, maschinenorientierten *imperativen* Paradigma nicht verwunderlich. Im Alltagsleben denken die Menschen aber doch relativ selten in Funktionen und weit öfter in Wenn-Dann-Zusammenhängen, d.h. sie denken *relational* oder "*logisch*" im Sinne des *logischen* Programmierparadigmas. Auch wir haben bei der Lösung der Verwandschaftsaufgabe relational gedacht.

Wegen der Netzorientiertheit des menschlichen Denkens hätte man erwarten können, dass die genannten Paradigmen zunehmend durch ein "*datenflussorientiertes Paradigma*" verdrängt werden, d.h dass zumehmend Programme als *Datenflusspläne* formuliert werden. Denn ein Datenflussplan kommt dem netzorientierten Denken am nächsten. Es gab Ansätze in dieser Richtung, die auf der Konzeption entsprechender Hardware in Form sogenannter *Datenflussmaschinen* basierten. Es haben sich aber weder die Datenflussmaschinen noch das datenflussorientierte Paradigma durchsetzen können. Doch ist das datenflussorientierte Paradigma in zwei andere Paradigmen eingeflossen, in das funktionale und in das *objektorientierte* Paradigma.

---

4 Zu den begrifflichen und philosophischen Hintergründen dieser Entwicklung findet der Leser in den Artikeln [Pflüger 94] und [Pflüger 97] interessante Ausführungen.

In Kap. 15.1 [15.1] hatten wir festgestellt, dass dem funktionalen Programmieren die Vorstellung eines Datenflusses zugrunde liegt. Das tritt in dem Umstand zutage, dass in einem funktionalen Programm ebenso wie in einem Datenflussprogramm (Datenflussplan) Zwischenresultate nicht explizit in Erscheinung treten, da sie gewissermaßen automatisch durch den Datenfluss weitergetragen werden. Insofern ist das funktionale Paradigma *datenflussorientiert* zu nennen.

Seit der Mitte der achtziger Jahre hat es sich zunehmend eingebürgert, diejenige Programmierweise, die sich infolge der Verwendung von Objekten (des Datentyps "Objekt") herausgebildet hat, als eigenständiges Paradigma aufzufassen und vom **objekorientierten Programmierparadigma** zu sprechen. Es vereinigt in sich Charakteristiken sowohl des imperativen als auch des funktionalen Paradigmas. Um das zu erkennen, lenken wir zunächst unsere Aufmerksamkeit auf eine spezifische Eigenschaft, die Objekte und Operatoren gemeinsam besitzen. Wir erinnern uns an die in Kap.18.2 [6] eingeführten Begriffe des Objekts und des objektorientierten Programms und zitieren:

"*Die einzelnen Objekte eines objektorientierten Programms bilden gewissermaßen ein Netz von Kooperationspartnern, die sich gegenseitig Aufträge erteilen können*."

Ihm stellen wir einen Satz aus Kap.13.7 [13.14] gegeüber:

"*In der Welt der Operatorennetze lässt sich die Menge der verkoppelten Operatoren als Menge kooperierender Akteure auffassen, die sich gegenseitig Daten zur weitern Verarbeitung zuspielen*."

Die gemeinsame Eigenschaft ist die Fähigkeit, Netze zu bilden, Netze kooperierender Partner oder Akteure. Betrachtet man ein aus dem Netz herausgelöstes Objekt, einen isolierten Akteur, so führt dieser eine Folge von Direktiven (Aufträgen, Befehlen) aus. Hierin liegt die *imperative* Komponente der objektorientierten Programmierung. Demgegenüber stellt die kooperative Bearbeitung eines Auftrages durch die Objekte eines objektorientierten Programms eine *datenflussorientierte* und damit auch eine *funktionale* Komponente dar.

Die auffälligste Ähnlichkeit besteht jedoch zwischen dem (nicht existierenden) datenflussorientierten Paradigma und dem objektorientierten Paradigma, konkreter zwischen Datenflussplänen und objektorientierten Programmen. Das zeigen die beiden zitierten Sätze. Fast könnte man geneigt sein, beide Paradigmen zu identifizieren und "objektorientiert" durch "datenflussorientiert" zu ersetzen. Doch würde damit ein wesentlicher Aspekt des Objektbegriffs verloren gehen, auf den ausführlicher eingegangen werden soll.

Wie bereits bemerkt, mag die Wahl des Wortes "Objekt" ungeschickt erscheinen, weil dieses Wort bereits umgangssprachlich belegt ist, wobei die Bedeutung des umgangssprachlichen Objektbegriffs wenig mit der des informatischen Objektbegriffs, dem die Vorstellung der Kapselung zugrunde liegt, zu tun zu haben scheint. Bei genauerer Betrachtung erkennt man jedoch, dass es gerade die Vorstellung der

relativen Kapselung ist, die den umgangssprachlichen Objektbegriff mit dem informatischen verbindet.

In Kap.5.5 [5.17] hatten wir festgestellt, dass sich im Denken des Menschen durch Beobachtung der Welt einzelne Objekte dadurch "herauskristallisieren", dass sie mit Merkmalswerten in Verbindung gebracht und im Endeffekt durch diese "definiert" und dadurch zu *Denkobjekten* werden [5.18]. Hierauf beruht das "Begreifen" der Welt durch das Neugeborene und dessen intellektuelle Entwicklung. Hierauf beruhen auch die begriffsbildenden Operationen von Bild 5.4, z.B. das Generalisieren und Präzisieren, also diejenigen Operationen, auf denen die Vererbung von Eigenschaften informatischer Objekte beruht. Entscheidend für die intellektuelle und kulturelle Evolution ist die relative Abgeschlossenheit und Stabilität der Denkobjekte. Im Denken treten Denkobjekte miteinander in Wechselwirkung, ohne sich gegenseitig zu stören (zu *zer*stören). Genau diese Eigenschaft bildet die Grundlage des informatischen Objektbegriffs. Hierin liegt eine nicht sofort erkennbare, aber umso stichhaltigere Rechtfertigung der Wortes "Objekt" als Name des so benannten Datentyps.

Eine zweite, sehr sinnfällige Rechtfertigung ergibt sich daraus, dass auf einem ausreichend hohen Abstraktionsniveau der informatische mit dem umgangssprachlichen Objektbegriff verschmilzt (begriffliche Konvergenz). In der Losung "*Jedem Objekt in der Wirklichkeit sein Objekt im Programm!*" hat die Verschmelzung ihren treffenden Ausdruck gefunden. Dahinter verbirgt sich eine neue Möglichkeit der Annäherung zwischen Programmier- und Modellierebene. Ein weiterer Schritt in Richtung des gestrichelten Pfeils in Bild 18.1, also in Richtung der Schließung der semantischen Lücke konnte getan werden. Das Ergebnis dieses Schrittes besteht darin, dass Denkobjekte, mit denen der Modellierer hantiert, in Objekte eines objektorientierten Programms überführt werden können. 11

Das Neue und Bedeutende des objektorientierten Ansatzes im Vergleich zum datenflussorientierten Ansatz (zur Operatorennetzmethode) besteht darin, dass Objekte keine Operatoren, keine Akteure sein müssen, dass sie vielmehr programmiersprachliche Repräsentationen beliebiger Denkobjekte sein können, die in Beziehung zueinander stehen und die sowohl im Denkprozess als auch im Abarbeitungsprozess eines objektorientierten Programms "ungestört" miteinander wechselwirken können.

Dadurch kommt eine neue Dynamik in den Abarbeitungsprozess hinein. Denn die Reaktion eines Objekts auf eine Direktive (auf eine Bitte um eine Dienstleistung) ist nicht durch den Programmierer vollständig vorherbestimmt, sondern kann von den momentanen Gegebenheiten abhängen, insbesondere vom inneren Zustand (vom "Wissenstand") des Objekts. Diese Dynamisierung ist ein entscheidender Vorteil des objektorientierten Paradigmas. Programmierungstechnisch wird sie durch *dynamisches Binden* realisiert [15.5].

Wie man sieht, besitzen Objekte im Vergleich zu den Prozeduren imperativer Programme mehr "Freiheiten", den Prozessverlauf mitzubestimmen, obwohl Objekte - ebenso wie Prozeduren - durch Direktiven (Befehle), die sie empfangen, aktiviert werden. Die Idee liegt nahe, auch *diese* "Unfreiheit" zu beseitigen und dem Objekt

die “Freiheit” zu geben, von sich aus aktiv zu werden, m.a.W. das Objekt mit *Eigeninitiative* auszustatten. Diese Idee ist bereits realisiert. Das Produkt, ein Objekt oder besser ein Akteur, der nicht nur auf Direktiven wartet, sondern selber aktiv werden kann, wird **Agent** genannt. Vertieft man sich in diese Idee, öffnet sich ein weites Feld neuer Perspektiven, die hier nur angedeutet werden sollen.

Wenn die Aktivität eines Agenten sinnvoll sein soll, darf sie nicht rein zufälligen Charakter haben, sondern muss irgendwelche Ziele verfolgen. Ein Agent muss seine Aktivität aus der Aktivität der Umgebung gemäß seiner *eigenen Intentionen* ableiten und sich in diesem Sinne der Umgebung anpassen. Der Unterschied zwischen Objekt und Agent besteht also schlagwortartig ausgedrückt darin, dass ein Objekt sich durch sein Wissen und Können auszeichnet, während ein Agent sich durch sein Wissen, Können *und Wollen* auszeichnet. Alle drei Eigenschaften liegen in der den Agenten implementierenden Software begründet. Sie können vom Programmierer vorgegeben werden oder sich im Wechselwirkungsprozess mit der Umgebung, d.h. mit anderen Agenten entwickeln. In letzterem Fall muss der Programmierer die Fähigkeit des Agenten, sich zu entwickeln, zu lernen, d.h. Erfahrungen zu sammeln und zu nutzen, einprogrammieren.

Was aber kann der Agent wollen? Wenn wir bei unserer ursprünglichen Zielstellung bleiben, ein Gerät zu bauen, das dem Menschen als *Denkassistent* dient, lautet die Antwort: *Ein Agent muss wollen, was sein Nutzer will*, d.h. was der Nutzer des betreffenden “*agentenorientierten*” Programms will. Das können die verschiedensten Aktionen sein. Ein Agent kann die Aufgabe haben, seinen “Herrn” an Termine zu erinnern, die Eingangspost durchzusehen und zu filtern, Schreiben zu verfassen oder seinen Herrn auf Fehler hinzuweisen. Der Bürger der Informationsgesellschaft kann *persönliche Assistenten* “einstellen” und im Internet aktiv werden lassen, um gezielt Informationen einzuholen bzw. zu verteilen oder um sich auf Chancen oder Gefahren aufmerksam machen zu lassen. Aus einem Expertensystem wird ein Agent oder ein Kollektiv von Agenten, von denen jeder ein Experte ist, der die Ziele des Nutzers verfolgt.

Die Eigenschaften und Fähigkeiten eines Agenten in der Form des persönlichen Assistenten überschreiten nicht den Rahmen der künstlichen Intelligenz, wie sie bisher besprochen worden ist. Der Schritt in unbekanntes Terrain wird dann vollzogen, wenn ein Agent “*wollen kann, was er will*”. Das bedeutet, dass der Agent über einen freien Willen und damit über Selbstbewusstsein verfügt.

Es wird den Leser nicht überraschen, dass hier der Punkt erreicht ist, an dem wir unsere Wanderung zu den Grenzen der künstlichen Intelligenz abbrechen. Wir wollten verstehen, wie sich auf der Grundlage des gegenwärtigen Wissensstandes künstliche *Intelligenz* produzieren, nicht aber, wie sich künstliches *Selbstbewusstsein* produzieren lässt. Wie immer brechen wir da ab, wo es besonders interessant, wo es geradezu spannend wird. Den Wanderer ergreift eine Spannung, die durch die Dunkelheit und Ungewissheit erzeugt wird, die über dem Gelände liegt, das er betritt,

falls er die Wanderung fortsetzt. Wir halten uns an den wittgensteinschen Imperativ: *"Wovon man nicht sprechen kann, darüber muss man schweigen."*

Das Ziel des Buches ist erreicht. Der letzte Teil bringt Ergänzungen, die zum einen in die Tiefe technischer Details gehen und zum anderen den eigentlichen Themenkreis des Buches überschreiten. Die Kapitel 19 und 20 richten sich an Leser, die etwas mehr über die technischen Schwierigkeiten erfahren möchten, die auf dem Weg zum gegenwärtigen Stand der Computertechnik und der künstlichen Intelligenz überwunden werden mussten. In Kapitel 21 wird ein Blick auf ein Gebiet geworfen, das die Brücke von der traditionellen Informatik zur *technischen Neuroinformatik*, vom Prozessorcomputer zum *Neurocomputer* schlägt. Im Zentrum steht der Begriff der Komplexität. Es werden Methoden erläutert, die es erlauben, auch solche Systeme und Prozesse zu simulieren, die infolge ihrer Komplexität nicht durchschaubar sind. In Kap.21.4 werden einige in Teil 3 offen gebliebene Fragen hinsichtlich der Simulierbarkeit des menschlichen Denkens noch einmal einer kritischen Analyse unterzogen. Dabei wird an Kap.17.3 angeknüpft.

Diejenigen Leser, welche den langen Weg bis zu diesem Punkt mitgegangen sind, sich aber für die genannten vertiefenden bzw. erweiternden Themen weniger interessieren, können die Kapitel 19 bis 21.3 überspringen. Das Schlusswort enthält einige persönliche Meinungsäußerungen des Autors zur Bedeutung der Informatik und der künstlichen Intelligenz für die Zukunft der menschlichen Gesellschaft.