Informatik I

Rudolf Berghammer

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Olshausenstraße 40
24098 Kiel

Skriptum zu der gleichnamigen Vorlesungen, abgehalten
im Wintersemester 2002 / 03 an der
Christian-Albrechts-Universität zu Kiel.

Einleitung

Informatik ist eine sehr junge Wissenschaft, die an deutschen Hochschulen erst ab dem Ende der sechziger Jahre gelehrt wird. Ihr Gegenstand ist die systematische und maschinelle Verarbeitung, Speicherung und Übertragung von Information. Dies ist sehr abstrakt ausgedrückt. Das vielschichtige Wesen der Informatik wird klarer, wenn man sich mit ihrer Aufteilung in die verschiedenen Teilbereiche befaßt, wie sie sich im Laufe der Zeit herausgebildet hat. Wir haben da etwa die folgende Dreiteilung:

- **Technische Informatik:** Diese beschäftigt sich seit den Anfängen der Informatik mit der Architektur, dem Aufbau und der Organisation von Rechenanlagen sowie den physikalischen und elektrotechnischen Grundlagen dazu. In neuerer Zeit hinzugekommen sind Themen wie die Einbettung von Computern in technische Systeme / computerbasierte Systeme (etwa Verkehrssysteme, Telefone), Robotik und Prozeßrechnertechnik.
- **Praktische Informatik:** Hier befaßt man sich mit dem Einsatz der Rechenanlagen zur Informations- und Datenverarbeitung. Dies betrifft insbesondere die Grundlagen von Programmiersprachen und Programmierung, Übersetzerkonstruktion, Betriebssysteme und Rechnernetze sowie Systeme zur Informationsverwaltung (speziell: Datenbanken), alles Gebiete, die zur sogenannten Kerninformatik zählen. Sie existieren ebenfalls seit den Anfängen der Informatik, haben aber gerade in den letzten Jahren moderne Entwicklungen mitgetragen, wie etwa die Automatisierung von Programmentwicklungsmethoden oder multimediale Datenbanken.
- **Theoretische Informatik:** In diesem Teilbereich geht es um theoretische Fragestellungen der Informatik, die es natürlich auch in den anderen Teilbereichen gibt und die sich oft aus technischen und praktischen Fragestellungen entwickelt haben. Ein Beispiel ist die Theorie der Automaten (und Formalen Sprachen), die in den sechziger und siebziger Jahren wesentliche Anregungen aus der Übersetzerkonstruktion bezog und nun, unter Erweiterung des klassischen Automatenbegriffs, etwa eine große Rolle bei den sogenannten reaktiven Systemen spielt. Deren Aufgabe ist es nicht, Werte zu berechnen, sondern auf Ereignisse und Daten aus ihrer Umwelt geeignet zu reagieren.

Bezüglich des ersten und vieler Fragestellungen des zweiten Punktes ist Informatik eine Ingenieur-Wissenschaft, d.h. ihrem Charakter nach konstruierend und technikorientiert. Betrachtet man hingegen Informatik hauptsächlich aus dem Blickwinkel des dritten Punktes, so hat man es mit einer Struktur- und Grundlagenwissenschaft zu tun, also mehr mit analysierendem Vorgehen, mit teilweise sehr großer Nähe zur Mathematik.

Im Informatik-Hauptstudium, d.h. nach dem Vordiplom, werden einzelne Themen der obigen drei Punkte in Spezialvorlesungen behandelt. In einer Informatik-Grundvorlesung, d.h. vor dem Vordiplom, wird in Deutschland derzeit noch üblicherweise ein Einstieg in und Überblick über ihre Teilbereiche gegeben. Dies erleichtert den Einstieg in das in

der Regel neue Studienfach, erlaubt es auch, im Hauptstudium oft einen mehr „abstrakten“ und top-down-orientierten Standpunkt einzunehmen, durch welchen viele Zusammenhänge klarer werden. Auch an der Universität Kiel wird so vorgegangen, wobei der Schwerpunkt von Informatik I auf dem Gebiet der Programmierung liegt.

Das Hauptanliegen dieser Vorlesung ist es, die Studierenden in die fundamentalen Konzepte und Paradigmen von Programmiersprachen und die Grundlagen des methodischen Programmierens und der systematischen Programmentwicklung einzuführen. Die Untersuchung und Vorstellung der Konzepte führt unter anderem zum Begriff der Semantik, d.h. der Unterscheidung zwischen dem Programm als einem (syntaktischen) Text und der dadurch berechneten (mathematischen) Funktion. Hier werden wir meist einen naiven Standpunkt einnehmen und uns am termmäßigen, operationellen Ablauf der Berechnungen orientieren. Dieses Vorgehen entspricht dem schon in der Schule geübten Vereinfachen von komplizierten Ausdrücken gemäß fester Regeln. Sowohl der denotationelle als auch der axiomatische semantische Standpunkt werden nur exemplarisch erklärt; ihre genauere Behandlung erfolgt später insbesondere in Spezialvorlesungen im Hauptstudium. Der zweite Hauptpunkt der Vorlesung – methodisches Programmieren und systematische Programmentwicklung – beinhaltet unter anderem formale Beschreibungen der gestellten Aufgaben, die semantikerhaltende Transformation von Programmen in effizientere Form, Programmverifikationen (d.h. mathematische Beweise, daß Programme die gestellten Aufgaben tatsächlich lösen) und das Finden von geeigneten Abstraktionen zur Konstruktion komplexerer Systeme sowie ihre programmiersprachliche Realisierung. Auch diese Themen werden später insbesondere im Hauptstudium vertieft behandelt.

Eine grobe Gliederung des – dieser kurzen allgemeinen Einführung in die Gegenstände der Informatik folgenden – Rests der Vorlesung Informatik I sieht wie folgt aus, wobei jeder Punkt genau einem Kapitel entspricht:

- Einige mathematische Grundbegriffe.
- Information und ihre Darstellung bzw. Verarbeitung.
- Einführung in die funktionale Programmierung.
- Modularisierung, Datenabstraktion, interne Zustände und Objektorientierung.
- Einführung in die objektorientierte imperative Programmierung.
- Einige abschließende Bemerkungen.

Dabei werden die Konzepte und Paradigmen der funktionalen Programmierung anhand der Programmiersprache ML erklärt. Als begleitende Literatur empfehlen wir die Bücher „Grundlagen funktionaler Programmierung“ von M. Erwig (Oldenburg-Verlag, 1999) und „Elements of ML Programming“ von J.D. Ullmann (Prentice Hall, 2. Auflage, 1998). Aber auch das Buch „ML for the working programmer“ von L.C. Paulson (Cambridge University Press, 2. Auflage, 1996) ist zur Vorlesungsbegleitung sehr gut geeignet.

Für die objektorientierte imperative Programmierung verwenden wir anschließend die Sprache Java. Zwei gut geeignete Bücher zur Einführung in Java sind „Java lernen: Anfangen, anwenden, verstehen“ von J. Bishop (Addison-Wesley, 2. Auflage, 2001) und „Lehrbuch der Programmierung mit Java“ von K. Echtle und M. Goedicke (dpunkt-Verlag, 2000). Einen sehr schönen konzeptuellen Zugang zu Objektorientierung, der ebenfalls Java verwendet, findet man in den Büchern „Konzepte objektorientierter Programmierung“ von A. Poetzsch-Heffter (Springer-Verlag, 2000) und „Einführung in die objektorientierte Programmierung mit Java“ von E.-E. Doberkat und S. Dißmann (Oldenburg-Verlag, 2. Auflage, 2002).

Inhaltsverzeichnis

1 Einige mathematische Grundbegriffe

Informatik stützt sich wie jede moderne Ingenieur- und Naturwissenschaft auf mathematische Notationen und Schlußweisen. Dies gilt insbesondere für die theoretische Informatik. Im folgenden stellen wir die grundlegendsten Begriffe der Mathematik vor, die wir immer wieder benutzen werden. Speziellere Begriffe werden später an den Stellen eingeführt, an denen sie erstmals benötigt werden.

1.1 Relationen und Funktionen

Im folgenden setzen wir den in den mathematischen Anfängervorlesungen benutzten naiven Mengenbegriff voraus. Eine **Menge** ist also eine Zusammenfassung von wohlunterscheidbaren Objekten (Dingen), den **Elementen**. Weiterhin verwenden wir die folgende Symbolik (wobei „$:\Longleftrightarrow$“ heißt „ist wahrheitsäquivalent definiert mittels“):

$$\begin{array}{lcl} x \in M & :\Longleftrightarrow & x \text{ ist ein Element der Menge } M \\ x \notin M & :\Longleftrightarrow & x \text{ ist kein Element der Menge } M \end{array}$$

Die Angabe einer Menge kann in mehreren Arten erfolgen. Wir benötigen am Anfang zwei Darstellungen:

a) **Explizite Darstellung:** Hier zählt man die Elemente auf und klammert sie mit den Mengenklammern „{“ und „}“. Dabei können bei unendlichen Mengen auch die berühmten drei Punkte „...“ verwendet werden, wenn das Bildungsgesetz klar erkennbar ist.

b) **Deskriptive Darstellung:** Ist P eine Eigenschaft (ein **Prädikat** oder eine **Formel**), so bezeichnen wir die Menge aller Objekte, für die P gilt, mit

$$\{\ x\ :\ P(x)\ \}.$$

Beim Hinschreiben von Prädikaten und Formeln verwenden wir dabei die „üblichen“ logischen Symbole, genannt Quantoren (erste Spalte) bzw. Junktoren (restliche Spalten) mit den folgenden sprachlichen Bedeutungen:

$\forall$	für alle	$\wedge$	und (Konjunktion)	$\rightarrow$	impliziert	$\neg$	nicht
$\exists$	es existiert	$\vee$	oder (Disjunktion)	$\leftrightarrow$	äquivalent		

Wir gehen dabei aber nicht nach einer strengen Syntax wie in der mathematischen Logik vor, sondern lassen oft auch umgangssprachliche Redewendungen zur Vereinfachung und zum besseren Verständnis zu.

Alle logischen Symbole treten oft in zwei Ebenen auf: in Prädikaten, die etwa Mengen definieren oder gewisse Sachverhalte beschreiben, und in Beweisen, in denen diese verwendet werden. Wir versuchen diese Ebenen zu unterscheiden, indem wir verschiedene Symbole wählen, etwa „$\rightarrow$“ auf der Formelebene und „$\Longrightarrow$“ in den Beweisen, oder statt der Symbole die umgangssprachlichen „Langfassungen“ verwenden.

1.1.1 Beispiele (Mengen von Zahlen)

Im folgenden führen wir einige fundamentale Mengen von Zahlen ein, die auch Zahlenbereiche genannt werden. Als erstes Beispiel betrachten wir die natürlichen Zahlen (wobei das Symbol „:=“ heißt „ist als Name / Bezeichnung definiert mittels“):

$$\mathbb{N} := \{\ 0, 1, 2, 3, \ldots\ \}$$

Neben den natürlichen Zahlen setzen wir auch die ganzen Zahlen als bekannt voraus und bezeichnen sie mit dem Symbol $\mathbb{Z}$. Wir haben also die folgende explizite Darstellung:

$$\mathbb{Z} := \{\ 0, 1, -1, 2, -2, 3, -3, \ldots\ \}$$

Als einen dritten Bereich von Zahlen betrachten wir $\mathbb{Q}$, die Menge der rationalen Zahlen und definiert durch:

$$\mathbb{Q} := \{\ x\ :\ \exists m, n \in \mathbb{Z} :\ n \neq 0 \wedge x = \frac{m}{n}\ \} = \{\ \frac{m}{n}\ :\ m, n \in \mathbb{Z} \wedge n \neq 0\ \}$$

Die reellen Zahlen bezeichnen wir mit dem Symbol $\mathbb{R}$. Sie können nicht mehr so einfach explizit oder deskriptiv dargestellt werden wie die Mengen $\mathbb{N}$, $\mathbb{Z}$ und $\mathbb{Q}$. Bezüglich ihrer Definition, etwa durch Cauchy-Folgen oder Dedekindsche Schnitte, müssen wir auf die Grundvorlesungen in Mathematik verweisen. ■

Beim naiven Mengenansatz ist bei der deskriptiven Darstellung immer Vorsicht geboten, da man Dinge niederschreiben kann, die nicht mehr vernünftig interpretiert werden können. Ein klassisches Beispiel ist $\{\ x\ :\ x \notin x\ \}$ („Russelsche Antimonie“), eine Konstruktion, die, als Menge betrachtet, zu einem Widerspruch führt.

1.1.2 Definition (Grundoperationen auf Mengen)

a) Mit $\emptyset$ bezeichnen wir die **leere Menge**, die kein Element enthält. Für alle x gilt also $x \notin \emptyset$. In deskriptiver Schreibweise haben wir

$$\emptyset = \{\ x\ :\ \mathit{false}\ \},$$

mit einem Prädikat *false*, welches niemals zutrifft, etwa definiert als $x \neq x$.

b) Die sogenannten **Booleschen** oder **verbandstheoretischen Operationen** auf Mengen sind wie folgt definiert:

$$\begin{aligned} M \cup N &:= \{\ x\ :\ x \in M \vee x \in N\ \} && \textbf{Vereinigung} \\ M \cap N &:= \{\ x\ :\ x \in M \wedge x \in N\ \} && \textbf{Durchschnitt} \\ M \backslash N &:= \{\ x\ :\ x \in M \wedge x \notin N\ \} && \textbf{Komplement} \end{aligned}$$

Ist die Menge M aus dem Zusammenhang klar, so schreibt man auch $\overline{N}$ statt $M \backslash N$ (**absolutes Komplement** bezüglich des Universums M). Die zwei wichtigsten **Vergleiche** auf Mengen sind:

$$\begin{aligned} M \subseteq N &:\Longleftrightarrow \forall x\ :\ x \in M \rightarrow x \in N && \textbf{Inklusion} \\ M = N &:\Longleftrightarrow \forall x\ :\ x \in M \leftrightarrow x \in N && \textbf{Gleichheit} \end{aligned}$$

Gilt $M \subseteq N$, so heißt M eine **Teilmenge** von N und N eine **Obermenge** von M.

c) Schließlich existieren auf Mengen noch die folgenden drei Operationen:

$$\begin{aligned} 2^M &:= \{\ X\ :\ X \subseteq M\ \} && \textbf{Potenzmenge} \\ M \times N &:= \{\ \langle x, y\rangle\ :\ x \in M \wedge y \in N\ \} && \textbf{direktes Produkt} \\ |M| &:= \text{Anzahl der Elemente von M} && \textbf{Kardinalität} \end{aligned}$$

Die Konstruktion $\langle x, y\rangle$ heißt ein **Paar**. Ist $|M| \in \mathbb{N}$, so nennt man M **endlich**. Für endliche Mengen gilt $\left|2^M\right| = 2^{|M|}$, daher rührt die spezielle Bezeichnung der Potenzmenge. In vielen Büchern findet man auch die Schreibweise $\mathfrak{P}(M)$ statt 2^M. ■

Im nächsten Beispiel führen wir eine weitere wichtige Menge ein.

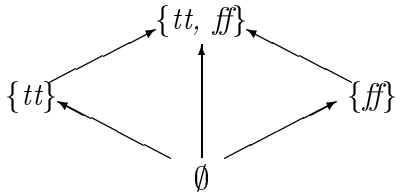
1.1.3 Beispiel (Wahrheitswerte)

Die Menge $\mathbb{B}$ der **Wahrheitswerte** wird definiert durch

$$\mathbb{B} := \{\mathit{ff},\ \mathit{tt}\},$$

wobei *ff* für „falsch“ und *tt* für „wahr“ steht. Auch andere Bezeichnungen für falsch und wahr sind geläufig, so O und L bzw. *false* und *true*.

Es gilt $|\mathbb{B}| = 2$ und somit $\left|2^{\mathbb{B}}\right| = 2^{|\mathbb{B}|} = 2^2 = 4$. Nachfolgend ist die Inklusionsbeziehung aller Elemente von $2^{\mathbb{B}}$ graphisch wiedergegeben:



Dabei führt ein Pfeil von A nach B, falls A in B als Teilmenge enthalten ist. Solche zeichnerischen Darstellungen für kleine Potenzmengen (allgemeiner: für geordnete Mengen, wobei dieser Begriff erst später eingeführt wird) sind wegen ihrer Übersichtlichkeit in der Literatur üblich. ■

Weitere Begriffe der Mengentheorie werden in den Grundvorlesungen der Mathematik gebracht. Wir brauchen davon im folgenden „Relation“ und „Funktion“. Das Wort „Relation“ beschreibt umgangssprachlich eine Beziehung zwischen Dingen, beispielsweise

die Zugspitze **ist höher als** der Watzmann

als Beziehung zwischen zwei sehr bekannten bayerischen Bergen oder

Kiel **liegt an der** Ostsee

als Beziehung zwischen Städten und Gewässern. In der Mathematik formalisiert man Relationen wie folgt.

1.1.4 Definition (Relationen)

Eine Teilmenge R von $M \times N$ heißt (binäre) **Relation** auf M und N mit **Argumentbereich** (oder **Vorbereich**) M und **Resultatbereich** (oder **Nachbereich**) N. Gilt $M = N$, so nennt man R **homogen** (oder **quadratisch**). ■

Trifft die Beziehung $\langle x, y\rangle \in R$ zu, so stehen die Elemente x, y in der Relation R. Man schreibt dafür auch xRy, etwa wenn R die übliche Ordnung auf den natürlichen Zahlen ist, oder R_{xy}, wenn man R als Boolesche Matrix interpretiert.

1.1.5 Definition (Grundoperationen auf Relationen)

a) Die leere Menge $\emptyset \in 2^{M\times N}$ heißt **Nullrelation** und wird auch mit O bezeichnet, das Universum $M \times N \in 2^{M\times N}$ heißt **Allrelation** und wird auch mit L bezeichnet.

b) Da Relationen Mengen sind, haben wir auf $2^{M\times N}$ die schon eingeführten **verbandstheoretischen Operationen** $\cup, \cap, \overline{}$ und die **Vergleichsoperationen** $\subseteq$ und $=$ (siehe Definition 1.1.2.b).

c) Die **Transposition** $R^\top \in 2^{N\times M}$ einer Relation $R \in 2^{M\times N}$ ist definiert durch

$$R^\top := \{\ \langle y, x\rangle \in N \times M\ :\ \langle x, y\rangle \in R\ \}.$$

d) Das **Produkt** (oder die **Komposition**) $RS \in 2^{M\times P}$ von zwei Relationen $R \in 2^{M\times N}$ und $S \in 2^{N\times P}$ ist definiert durch

$$RS := \{\ \langle x, z\rangle \in M \times P\ :\ \exists\, y \in N : \langle x, y\rangle \in R \wedge \langle y, z\rangle \in S\ \}.$$

e) Die **identische Relation** $\mathsf{I} \in 2^{M\times M}$ ist durch

$$\mathsf{I} := \{\ \langle x, y\rangle \in M \times M\ :\ x = y\ \} = \{\ \langle x, x\rangle\ :\ x \in M\ \}$$

definiert. Diese homogene Relation ist die relationale Darstellung des Gleichheitsprädikats auf M. ■

Man beachte, daß nicht beliebige Relationen miteinander verknüpft werden dürfen. Bei der Vereinigung, dem Durchschnitt und den Vergleichsoperationen müssen sowohl der Argumentbereich als auch der Resultatbereich beider Relationen übereinstimmen. Will man ein Produkt RS bilden, so ist dies nur möglich, wenn der Resultatbereich von R gleich dem Argumentbereich von S ist. Die beiden einstelligen Operationen der Komplementbildung und der Transposition sind für alle Relationen definiert.

Natürlich erfüllen die eben eingeführten speziellen Relationen und Operationen auch spezielle Eigenschaften. Beispielsweise ist das Produkt assoziativ und hat I als neutrales Element. Als Gleichungen schreibt sich dies wie folgt:

$$(QR)S = Q(RS) \qquad\qquad \mathsf{I}R = R = R\mathsf{I}$$

Man beachte, daß in den rechten Gleichungen in $\mathsf{I}R$ die identische Relation auf dem Argumentbereich von R und in $R\mathsf{I}$ die identische Relation auf dem Resultatbereich von R gemeint ist. Solche Überlagerungen von Bezeichnungen sind in der Mathematik aus Gründen der Lesbarkeit durchaus üblich. Auf die Gesetze der Relationen gehen wir an dieser Stelle noch nicht genauer ein, sondern erwähnen sie erst dort, wo sie verwendet werden.

1.1.6 Definition (Spezielle homogene Relationen)

Eine homogene Relation $R \in 2^{M \times M}$ heißt

$$
\begin{array}{lll}
\textbf{reflexiv} & :\Longleftrightarrow & \forall\, x \in M : \langle x, x\rangle \in R \\
 & \Longleftrightarrow & \mathsf{I} \subseteq R \\
\textbf{symmetrisch} & :\Longleftrightarrow & \forall\, x, y \in M : \langle y, x\rangle \in R \to \langle x, y\rangle \in R \\
 & \Longleftrightarrow & R^\top \subseteq R \\
\textbf{transitiv} & :\Longleftrightarrow & \forall\, x, y, z \in M : \langle x, y\rangle \in R \wedge \langle y, z\rangle \in R \to \langle x, z\rangle \in R \\
 & \Longleftrightarrow & RR \subseteq R \\
\textbf{antisymmetrisch} & :\Longleftrightarrow & \forall\, x, y \in M : \langle x, y\rangle \in R \wedge \langle y, x\rangle \in R \to x = y \\
 & \Longleftrightarrow & R \cap R^\top \subseteq \mathsf{I}.
\end{array}
$$

Eine **Ordnung** oder **Ordnungsrelation** ist eine reflexive, antisymmetrische und transitive Relation. Als **Äquivalenzrelation** bezeichnet man eine reflexive, symmetrische und transitive Relation ■

Üblicherweise verwendet man bei Ordnungen das Zeichen $\leq$ und die sogenannte Infix-Schreibweise $x \leq y$. Dann steht $x < y$ als Abkürzung für $x \leq y \wedge x \neq y$. Gilt $x \leq y$, so heißt x kleiner oder gleich y; im Falle $x < y$ heißt x kleiner y. Daß zwei Elemente x und y in einer Äquivalenzrelationen enthalten sind, bezeichnet man oft mit $x \equiv y$.

Die im nächsten Punkt eingeführten speziellen Relationen führen zu einem Begriff, der fundamental für die gesamte Mathematik ist.

1.1.7 Definition (Funktionen)

Eine Relation $R \in 2^{M \times N}$ heißt

$$
\begin{array}{lll}
\textbf{eindeutig} & :\Longleftrightarrow & \forall\, x \in M\ \forall\, y, z \in N : \langle x, y\rangle \in R \wedge \langle x, z\rangle \in R \to y = z \\
 & \Longleftrightarrow & R^\top R \subseteq \mathsf{I} \\
\textbf{total} & :\Longleftrightarrow & \forall\, x \in M\ \exists\, y \in N : \langle x, y\rangle \in R \\
 & \Longleftrightarrow & R\mathsf{L} = \mathsf{L}.
\end{array}
$$

Eine eindeutige Relation heißt auch **partielle Funktion**; eine eindeutige und totale Relation heißt auch **Funktion**. N^M ist die **Menge der Funktionen**[1] von M nach N. Statt Funktion verwendet man auch das Wort **Abbildung**. ■

[1] Manchmal steht N^M auch für alle partiellen Funktionen von M nach N.

Bei Funktionen schreibt man nicht $R \in 2^{M \times N}$ und $\langle x, y \rangle \in R$, sondern $R : M \longrightarrow N$ bzw. $R \in N^M$ und $R(x) = y$ und sagt, daß x durch R auf y abgebildet wird. Auch bezeichnet man sie meist mit besonderen kleinen Buchstaben wie f, g und h. Bei partiellen Funktionen verwendet man zusätzlich noch die Notationen „$f(x) = \text{undef}$“ bzw. „$f(x) \neq \text{undef}$“ um anzuzeigen, daß kein bzw. ein Element zu x in Relation steht. Obwohl „undef“ formal kein Objekt einer Menge ist, werden wir es manchmal doch in Schreibweisen wie „$x * \text{undef} = \text{undef}$“ oder Argumentationen wie „$\ldots$ da $f(x) \neq \text{undef}$ folgt $\ldots$“ benutzen, um die verwendeten Schlüsse deutlicher zu machen.

1.1.8 Beispiele (für Funktionen)

Zuerst betrachten wir die Nachfolgerbildung auf den natürlichen Zahlen. Als Relation haben wir hier:

$$succ := \{ \langle 0, 1 \rangle, \langle 1, 2 \rangle, \ldots \} = \{ \langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \ : \ y = x + 1 \}$$

Diese Relation ist eine Funktion und sieht in der üblichen Schreibweise wie folgt aus:

$$succ : \mathbb{N} \longrightarrow \mathbb{N} \qquad\qquad succ(x) = x + 1$$

Als ein zweites Beispiel befassen wir uns noch mit der Vorgängerbildung auf den natürlichen Zahlen. Als Relation schreibt sich diese wie folgt:

$$pred := \{ \langle 1, 0 \rangle, \langle 2, 1 \rangle, \ldots \} = \{ \langle x, y \rangle \in \mathbb{N} \times \mathbb{N} \ : \ y = x - 1 \}$$

Die Relation *pred* ist eindeutig, aber nicht total, da die Null keinen Vorgänger besitzt. Wir haben es also mit einer partiellen Funktion zu tun, die als

$$pred : \mathbb{N} \longrightarrow \mathbb{N} \qquad\qquad pred(x) = \begin{cases} x - 1 & : \quad x \neq 0 \\ \text{undef} & : \quad \text{sonst} \end{cases}$$

in der üblichen Schreibweise definiert wird. ■

Im Falle von Funktionen wird das Relationenprodukt fg zur **Funktionskomposition** (mit Vertauschung der Argumente) $g \circ f$, welche definiert ist durch $(g \circ f)(x) := g(f(x))$.

1.1.9 Definition (Spezielle Funktionen)

Eine Funktion $f : M \longrightarrow N$ heißt

$$\begin{array}{lll} \textbf{injektiv} & :\Longleftrightarrow & \forall\, x, y \in M : f(x) = f(y) \rightarrow x = y \\ \textbf{surjektiv} & :\Longleftrightarrow & \forall\, y \in N \ \exists\, x \in M : f(x) = y. \end{array}$$

Eine **bijektive Funktion** oder **Bijektion** ist definiert als injektive und surjektive Funktion. Existiert eine Bijektion von M nach N, so heißen diese Mengen **gleichmächtig** oder **isomorph**. ■

In Definition 1.1.2.c hatten wir das direkte Produkt $M \times N$ von zwei Mengen eingeführt. Dies verallgemeinert sich zum **n-fachen direkten Produkt** von $n \geq 0$ Mengen wie folgt:

$$\prod_{i=1}^{n} M_i := \{\ \langle m_1, \ldots, m_n \rangle\ :\ \forall\, i : 1 \leq i \leq n \to m_i \in M_i\ \}$$

Für $n = 1$ ist dieses direkte Produkt $\prod_{i=1}^{1} M_i$ isomorph zur Menge M_1 vermöge der bijektiven Funktion $f : \prod_{i=1}^{1} M_i \longrightarrow M_1$ mit $f(\langle m_1 \rangle) = m_1$, und wird in der Regel mit dieser Menge identifiziert; für $n = 0$ besteht das direkte Produkt nur aus dem **leeren Tupel** $\langle\rangle$. Somit gilt also $|\prod_{i=1}^{0} M_i| = |\{\langle\rangle\}| = 1$.

Sind alle Mengen M_i eines direkten Produkts identisch und gleich einer Menge M, so schreibt man vereinfachend M^n statt $\prod_{i=1}^{n} M$. Also gilt:

$$M^n := \{\ \langle m_1, \ldots, m_n \rangle\ :\ \forall\, i : 1 \leq i \leq n \to m_i \in M\ \}$$

Die Paare erweiternde Konstruktion $\langle m_1, \ldots, m_n \rangle$ heißt ein **n-Tupel**, wobei für $n = 3$ bzw. $n = 4$ auch die Bezeichnungen Tripel bzw. Quadrupel üblich sind. Eine (partielle) Funktion $f : \prod_{i=1}^{n} M_i \longrightarrow N$ nennt man **n-stellig**. Statt $f(\langle m_1, \ldots, m_n \rangle)$ schreibt man hier vereinfachend nur eine Klammerung $f(m_1, \ldots, m_n)$. Alle obigen Begriffe übertragen sich somit, denn eine n-stellige (partielle) Funktion ist eine (partielle) Funktion $f : M \longrightarrow N$ im bisherigen Sinne, mit $M = \prod_{i=1}^{n} M_i$ als speziellem Argumentbereich.

Natürlich gibt es auch (partielle) Funktionen mit einem direkten Produkt als Wertebereich. Man spricht dann von **tupelwertigen** (partiellen) Funktionen.

1.2 Induktion und induktive Definitionen

Induktion (oft auch in der Form von Rekursion, etwa bei rekursiven Funktionen) spielt in der Informatik und der Mathematik eine entscheidende Rolle. Sie ist eng verbunden mit speziellen Ordnungen und Beweisprinzipien – den sogenannten Induktionsprinzipien. Auf den natürlichen Zahlen $\mathbb{N}$ kennt man beispielsweise von der Schule her die **vollständige Induktion**. Sie wird dargestellt durch die nachfolgende Inferenzregel:

$$\frac{\begin{array}{c} P(0) \\ \forall\, x \in \mathbb{N}\ :\ P(x) \to P(x+1) \end{array}}{\forall\, x \in \mathbb{N}\ :\ P(x)} \qquad (Ind_1)$$

So eine Inferenzregel besteht aus Oberformeln (über dem Strich), auch **Prämissen** oder Voraussetzungen genannt, und einer Unterformel (unter dem Strich), auch **Konklusion** genannt, und drückt aus, daß die Konjunktion der Oberformeln die Unterformel impliziert.

Ein weiteres Induktionsprinzip auf den natürlichen Zahlen ist die **noethersche Induktion**, welcher die nachfolgende Inferenzregel entspricht:

$$\frac{\begin{array}{c} P(0) \\ \forall\, x \in \mathbb{N} \setminus \{0\}\ :\ (\forall\, y \in \mathbb{N}\ :\ y < x \to P(y)) \to P(x) \end{array}}{\forall\, x \in \mathbb{N}\ :\ P(x)} \qquad (Ind_2)$$

Es kann relativ einfach gezeigt werden, daß die beiden Induktionsprinzipien gleichwertig sind, d.h. aus der Gültigkeit von vollständiger Induktion die Gültigkeit von noetherscher Induktion folgt, und umgekehrt. Für die Anwendung ist noethersche Induktion notwendig, wenn zu einem $n \in \mathbb{N}$ der rekursive Rückgriff, etwa in einer rekursiven Funktionsdefinitionen, auch auf Elemente ungleich $n-1$ erfolgt. Dies wird später auch bei funktionaler Programmierung von Bedeutung sein.

Die vollständige Induktion orientiert sich am Aufbau jeder natürlichen Zahl n als endliche Summe der Form $(\ldots((0+1)+1)+\ldots)+1$ mit genau n Einsen; die noethersche Induktion verwendet eine spezielle Eigenschaft der Ordnung $\leq$ auf $\mathbb{N}$. Diese Zusammenhänge sollen nun allgemein untersucht werden. Dabei beginnen wir mit dem zweiten Prinzip. Wir brauchen einige neue Begriffe:

1.2.1 Definition (Ordnungsbegriffe)

a) Eine **geordnete Menge** oder eine **Ordnung** ist ein Paar $(M, \leq)$ mit einer Menge $M \neq \emptyset$ und einer Ordnungsrelation $\leq \in 2^{M \times M}$.

b) Es seien $(M, \leq)$ eine geordnete Menge und $N \subseteq M$. Ein Element $x \in N$ heißt

$$\begin{array}{lll}
\textbf{größtes Element} \text{ von } N & :\Longleftrightarrow & \forall\, y \in N \;:\; y \leq x \\
\textbf{kleinstes Element} \text{ von } N & :\Longleftrightarrow & \forall\, y \in N \;:\; x \leq y \\
\textbf{maximales Element} \text{ von } N & :\Longleftrightarrow & \forall\, y \in N \;:\; x \leq y \rightarrow x = y \\
\textbf{minimales Element} \text{ von } N & :\Longleftrightarrow & \forall\, y \in N \;:\; y \leq x \rightarrow y = x
\end{array}$$

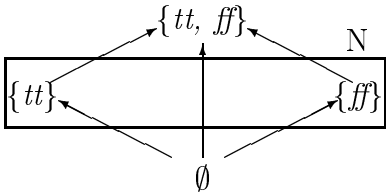
c) Eine Folge $(m_i)_{i \in \mathbb{N}}$ heißt eine **(abzählbar-unendliche aufsteigende) Kette**, falls $m_i \leq m_{i+1}$ für alle $i \in \mathbb{N}$ gilt. Die Kette wird **stationär**, falls es ein $j \in \mathbb{N}$ gibt, so daß $m_j = m_{j+k}$ für alle $k \in \mathbb{N}$ gilt. ■

Ketten schreibt man oft auch in der Form $m_0 \leq m_1 \leq m_2 \leq \ldots$ und eine Kette wird stationär, wenn sie von der Form $m_0 \leq \ldots \leq m_j = m_{j+1} = \ldots$ ist. Die in Punkt b) definierten speziellen Elemente müssen nicht immer existieren. Gibt es ein größtes Element, so ist dies eindeutig. Gleiches gilt für kleinste Elemente. Maximale und minimale Elemente müssen hingegen nicht eindeutig sein.

1.2.2 Beispiele (für Ordnungsbegriffe)

a) Wir betrachten $(\mathbb{N}, \leq)$ versehen mit der üblichen Ordnung und $G \subseteq \mathbb{N}$ als die Teilmenge der geraden Zahlen $0, 2, 4, \ldots$ in $\mathbb{N}$. Dann gelten offensichtlich die folgenden Eigenschaften: Die Teilmenge G hat kein größtes Element und auch kein maximales Element. (Die Existenz eines größten Element würde die eines maximalen Elements nach sich ziehen.) Hingegen hat die Teilmenge G die Null als kleinstes Element. Diese Zahl ist auch das einzige minimale Element.

b) Für das zweite Beispiel, die durch die Inklusion geordnete Potenzmenge von $\mathbb{B}$, d.h. also die Ordnung $(2^{\mathbb{B}}, \subseteq)$, betrachten wir die Teilmenge $N := \{tt, ff\}$. Graphisch haben wir dann



und aus diesem Bild erkennt man sofort die zwei folgenden Tatsachen: Die Teilmenge N hat kein größtes Element und auch kein kleinstes Element. Sowohl maximal als auch minimal in der Teilmenge N sind die beiden Elemente (hier einelementigen Mengen) $\{tt\}$ und $\{ff\}$. ■

Nach diesen Beispielen können wir nun die für die noethersche Induktion entscheidende Eigenschaft festlegen.

1.2.3 Definition (Noethersche Ordnung)

Die geordnete Menge $(M, \leq)$ heißt **noethersch geordnet**, falls jede Teilmenge $N \subseteq M$ mit $N \neq \emptyset$ ein minimales Element besitzt. ■

Beispielsweise ist $(\mathbb{N}, \leq)$ noethersch. Hingegen ist $(\mathbb{R}, \leq)$ nicht noethersch. Man kann noethersch auch anders beschreiben:

$$(M, \leq) \text{ noethersch } \iff \text{ Jede Kette } (m_i)_{i \in \mathbb{N}} \text{ in } (M, \geq) \text{ wird stationär}$$

Dabei steht, wie üblich, $x \geq y$ für $y \leq x$. Eine Ordnung ist also genau dann noethersch, wenn in ihr jede abzählbar-unendliche absteigende Kette $m_0 \geq m_1 \geq \ldots$ stationär wird. Letzteres heißt, daß es keine abzählbar-unendlichen echt absteigenden Ketten $m_0 > m_1 > \ldots$ gibt. Damit ist auch sofort klar, daß die reellen Zahlen nicht noethersch geordnet sind. Beispielsweise existiert in ihnen die unendliche echt absteigende Kette der ganzen Zahlen.

Nach diesen Vorbereitungen können wir nun den entscheidenden Satz formulieren und beweisen:

1.2.4 Satz (Prinzip der Noetherschen Induktion)

Ist $(M, \leq)$ eine noethersch geordnete Menge, so gilt das nachfolgende Induktionsprinzip, genannt noethersche Induktion:

$$\frac{\forall\, x \in M : (\forall\, y \in M : y < x \to P(y)) \to P(x)}{\forall\, x \in M : P(x)}$$

Beweis (durch Widerspruch): Angenommen, es gibt ein $x_0 \in M$, für das $P(x_0)$ nicht gilt. Wir betrachten dann die folgende Menge:

$$S := \{\ x \in M\ :\ P(x) \text{ gilt nicht } \}$$

Wegen $x_0 \in S$ gilt $S \neq \emptyset$. Die Ordnung $(M, \leq)$ ist noethersch. Also hat S ein minimales Element $s \in S$. Für alle $y \in M$ mit $y < s$ gilt somit $y \notin S$ und damit auch $P(y)$ nach der Definition von S. Die Oberformel (genannt Induktionsvoraussetzung) zeigt nun aber $P(s)$, also $s \notin S$. Das ist ein Widerspruch zu $s \in S$. ■

Ist das Element $x \in M$ minimal (bezüglich der gesamten Menge M), dann haben wir die nachfolgende Äquivalenz, wobei das in ihr verwendete Prädikat *true* (als Negation des schon eingeführten *false*) immer zutrifft:

$$\begin{array}{lll} (\forall\, y \in M : y < x \to P(y)) \to P(x) & \iff & (\forall\, y \in M :\ \mathit{false}\ \to P(y)) \to P(x) \\ & \iff & (\forall\, y \in M :\ \mathit{true}) \to P(x) \\ & \iff & \mathit{true}\ \to P(x) \\ & \iff & P(x) \end{array}$$

In dieser Rechnung haben wir anfangs benutzt, daß $y < x$ nicht gelten kann, da x minimal in M ist. In den weiteren Schritten werden logische Regeln verwendet, die umgangssprachlich folgendes beschreiben: Aus einer falschen Aussage kann man alles folgern, sind Aussagen immer wahr oder immer falsch, so spielen Quantoren keine Rolle, und eine Aussage folgt aus einer wahren Aussage genau dann, wenn sie wahr ist. Aufgrund der obigen Beziehung hat man für minimale Elemente $x \in M$ die Eigenschaft $P(x)$ explizit zu zeigen. Im Falle der Ordnung $(\mathbb{N}, \leq)$ ist $0 \in \mathbb{N}$ minimal. Daraus resultiert die Voraussetzung $P(0)$ in der Induktionsregel (Ind_2) am Beginn dieses Abschnitts.

Wir kommen nun zum ersten Prinzip, das im Fall von $\mathbb{N}$ zur vollständigen Induktion führt. Jede Zahl $n \in \mathbb{N}$ kann man sich von 0 und der Nachfolgerfunktion $succ : \mathbb{N} \to \mathbb{N}$ erzeugt vorstellen, da $n = succ(succ(\ldots(succ(0))\ldots))$ mit n $succ$-Anwendungen gilt. Dies ist ein Spezialfall einer induktiven Definition. Allgemein hat man folgendes Verfahren:

1.2.5 Definition (Induktive Erzeugung von Mengen)

Gegeben seien eine Menge U, das **Universum**, eine Teilmenge $B \subseteq U$, die **Basis**, und eine Menge C von Funktionen $c : U^{n_c} \longrightarrow U$, die **Konstruktoren** mit Stelligkeit n_c. Die Teilmenge S von U heißt von B und den Konstruktoren c **induktiv erzeugt** (auch: **induktiv definiert**), falls sie die kleinste Menge ist mit

$$(1)\ \ B \subseteq S \qquad\qquad (2)\ \ \forall\, c \in C\ \forall\, x_1, \ldots, x_{n_c} \in S :\ c(x_1, \ldots, x_{n_c}) \in S.$$ ■

Eigenschaften für Elemente induktiv definierter Mengen kann man beweisen, indem man, ausgehend von der Basis B, sich an den Konstruktoren „hochhangelt“. Formal haben wir das folgende Prinzip:

1.2.6 Satz (Prinzip der Strukturellen Induktion)

Ist die Menge S von der Basis B und der Menge C von Konstruktoren induktiv erzeugt, so gilt die folgende Inferenzregel, genannt strukturelle Induktion:

$$\frac{\begin{array}{c}\forall\, x \in B : P(x) \\ \forall\, c \in C\ \forall\, x_1, \ldots, x_{n_c} \in S : \bigwedge_{i=1}^{n_c} P(x_i) \rightarrow P(c(x_1, \ldots, x_{n_c}))\end{array}}{\forall\, x \in S : P(x)}$$

Beweis: Wir definieren die Teilmenge M von S wie folgt:

$$M := \{\ x \in S\ :\ P(x)\ \}$$

Die erste Prämisse des Induktionsschemas zeigt in Verbindung mit Definition 1.2.5 (1) sofort

$$B \subseteq M.$$

Nun sei ein Konstruktor $c \in C$ gegeben. Dann zeigen die zweite Prämisse des Induktionsschemas und Definition 1.2.5 (2), daß

$$\forall\, x_1, \ldots, x_{n_c} \in M : c(x_1, \ldots, x_{n_c}) \in M$$

zutrifft. Also gelten für die Menge M die Eigenschaften (1) und (2) von Definition 1.2.5.

Die Menge S ist von der Basis B und den Konstruktoren aus C induktiv erzeugt und somit die kleinste Menge, die (1) und (2) von Definition 1.2.5 erfüllt. Damit haben wir $S \subseteq M$. Insgesamt gilt also $M = S$ und dies in Kombination mit der Definition von M zeigt $P(x)$ für alle Elemente $x \in S$. ■

Ist die Menge S durch die Basis B und die Konstruktoren C induktiv erzeugt, so formuliert man dies üblicherweise in der Literatur weniger formal (aber dafür vielleicht etwas verständlicher) wie nachfolgend angegeben:

(1) Für alle $x \in B$ gilt auch $x \in S$.

(2) Für alle $c \in C$ und alle $x_1, \ldots, x_{n_c} \in S$ gilt auch $c(x_1, \ldots, x_{n_c}) \in S$.

(3) Es gibt keine Elemente in S außer denen, die (1) und (2) vorschreiben.

Bedingung (3) wird dabei auch noch oft weggelassen, wenn man explizit erwähnt, daß S durch B und C induktiv erzeugt/definiert ist. Bei Induktionsprinzipien und -beweisen haben sich auch gewisse Sprechweisen eingebürgert. Man nennt die erste Oberformel der Inferenzregel des eben bewiesenen Satzes den **Induktionsbeginn** und die zweite Oberformel den **Induktionsschluß**. Die linke Seite der Implikation des Induktionsschlußes wird auch **Induktionshypothese** genannt. Analog spricht man auch bei noetherscher Induktion, der Inferenzregel von Satz 1.2.4, von Induktionsbeginn, Induktionsschluß und Induktionshypothese. Hier verifiziert man beim Induktionsbeginn die Eigenschaft P für alle minimalen Elemente. Beim Induktionsschluß zeigt man dann, daß für ein beliebiges

nichtminimales Element die Eigenschaft P aus der Annahme (der Induktionshypothese) folgt, daß P für alle echt kleineren Elemente gilt.

Im folgenden geben wir einige Beispiele für induktive Definitionen von Mengen, entsprechende Induktionsprinzipien und Anwendungen in Gestalt von induktiven Definitionen von Funktionen bzw. von Induktionsbeweisen an. Weitere Beispiele werden wir später kennenlernen, insbesondere bei den rekursiven Rechenvorschriften in den Teilen des Skriptums, welche die funktionale Programmierung betreffen.

1.2.7 Beispiele (für Induktion)

a) Die **natürlichen Zahlen** $\mathbb{N}$ sind von der Basis $B := \{0\}$ und der Nachfolgerfunktion $succ(x) := x + 1$ induktiv erzeugt, wobei das Universum etwa die Menge $\mathbb{R}$ der reellen Zahlen sein kann. Die konkrete Ausprägung von Satz 1.2.6 ist gerade das Induktionsprinzip (Ind_1).

Aufbauend auf die eben erwähnte Erzeugung kann man nun Funktionen induktiv definieren. Wir geben zwei Beispiele an:

(1) Die Potenzfunktion $pow : \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{N}$ kann man induktiv durch die nachstehenden zwei Gleichungen definieren:

$$pow(x, 0) = 1 \qquad pow(x, n + 1) = x * pow(x, n)$$

Man schreibt üblicherweise x^n statt $pow(x, n)$.

(2) Die iterierte Komposition $pow : M^M \times \mathbb{N} \longrightarrow M^M$ von Funktionen kann man induktiv definieren durch:

$$pow(f, 0) = id \qquad pow(f, n + 1) = f \circ pow(f, n)$$

Dabei ist $id : M \longrightarrow M$ die identische Funktion auf M, welche $id(x) = x$ erfüllt. Man schreibt üblicherweise f^n statt $pow(f, n)$.

Analog zu (2) definiert man auch induktiv die n-te Potenz R^n einer homogenen Relation $R \in 2^{M \times M}$. Wir werden die relationalen Potenzen später im Rahmen von relationalen Hüllen einführen.

b) Wir wollen für die in (1) definierte Potenzfunktion pow nun in allen Einzelheiten beweisen, daß

$$\forall\, x, m, n \in \mathbb{N} : pow(x, m + n) = pow(x, m) * pow(x, n) \qquad (*)$$

gilt, d.h., in der üblichen Schreibweise, $x^{m+n} = x^m * x^n$ für alle $x, m, n \in \mathbb{N}$. Dazu definieren wir ein Prädikat P auf den natürlichen Zahlen durch

$$P(n) \;:\Longleftrightarrow\; \forall\, x, m \in \mathbb{N} : pow(x, m + n) = pow(x, m) * pow(x, n)$$

und zeigen $\forall\, n \in \mathbb{N} : P(n)$, was genau der Behauptung $(*)$ entspricht, durch strukturelle Induktion.

Den **Induktionsbeginn** $P(0)$, die erste Oberformel der Inferenzregel von Satz 1.2.6, verifiziert man wie folgt, indem man die Definition von $pow(x,0)$ verwendet:

$$\begin{array}{rcl} P(0) & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : pow(x,m+0) = pow(x,m) * pow(x,0) \\ & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : pow(x,m) = pow(x,m) * 1 \\ & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : pow(x,m) = pow(x,m) \\ & \Longleftrightarrow & true \end{array}$$

Beim **Induktionsschluß**, der zweiten Oberformel der Inferenzregel von Satz 1.2.6, haben wir nur den Konstruktor *succ* zu betrachten. Sei also $n \in \mathbb{N}$. Dann gilt, unter Verwendung der zweiten Gleichung von *pow*:

$$\begin{array}{rcl} P(n+1) & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : pow(x,m+n+1) = pow(x,m) * pow(x,n+1) \\ & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : x * pow(x,m+n) = pow(x,m) * x * pow(x,n) \\ & \Longleftrightarrow & \forall\, x,m \in \mathbb{N} : x * pow(x,m+n) = x * pow(x,m) * pow(x,n) \\ & \Longleftarrow & \forall\, x,m \in \mathbb{N} : pow(x,m+n) = pow(x,m) * pow(x,n) \\ & \Longleftrightarrow & P(n) \end{array}$$

In der üblichen Sprechweise der Mathematiker haben wir durch die Definition von P in Abhängigkeit von n die (implizit als allquantifiziert angenommene) Gleichung $x^{m+n} = x^m * x^n$ durch Induktion nach n bewiesen. ■

Wir werden im weiteren Verlauf des Skriptums viele Mengen der von uns benötigten Objekte induktiv definieren. Bei der induktiven Definition von Funktionen auf solchen induktiv erzeugten Mengen ist dann allerdings etwas Vorsicht geboten. Keine Probleme gibt es in den folgenden zwei Fällen:

a) Für jedes Element der betrachteten Menge ist der Aufbau mit Hilfe der Basis und der Konstruktoren **eindeutig vorgegeben**, wie beispielsweise im Falle der natürlichen Zahlen $\mathbb{N}$ mit der Basis $\{0\}$ und der Konstruktorfunktion *succ*. Solche Erzeugungen nennt man in der Literatur auch **frei**. Bei der Definition von syntaktischen Objekten, etwa Termen, verwendet man in der Regel Klammern, um freie Erzeugungen zu erhalten.

b) Es können zwar verschiedene Darstellungen existieren, aber für jedes Element gibt es eine ausgezeichnete Darstellung, genannt **Normalform**, in die man jede andere Darstellung durch vorgegebene Regeln überführen kann. Ein Beispiel hierzu ist die Menge $\mathbb{Z}$ der ganzen Zahlen mit den Konstruktoren *pred* und *succ*, wo die Normalformen $succ^n(0)$ und $pred^n(0)$ existieren. Die zur Überführung verwendeten Regeln sind $succ(pred(n)) = n$ und $pred(succ(n)) = n$ für alle $n \in \mathbb{Z}$.

Existieren verschiedene Darstellungen ohne Normalformen, so muß man für die induktiv definierte Funktion die Wohldefiniertheit (d.h. die Funktionseigenschaften „Eindeutigkeit“ und „Totalität“, siehe Definition 1.1.7) explizit zeigen. Für weitere Einzelheiten zu Induktion und induktive Definition verweisen wir auf das Lehrbuch „The foundations of program verification“ von J. Loeckx und K. Sieber, welches beim Teubner Verlag, Stuttgart, erschienen ist.

2 Information, ihre Darstellung und Verarbeitung

Wie schon in der Einleitung erwähnt, ist der Gegenstand der Informatik, grob gesagt, die Verarbeitung, Speicherung und Übertragung von Information. In diesem Kapitel befassen wir uns mit dem ersten Aspekt, der Verarbeitung von Information. Dazu ist es auch notwendig, über ihre Darstellung zu reden. Grundlegend sind hier Zeichenreihen und Terme, denen, neben dem Algorithmusbegriff, das Hauptinteresse dieses Kapitels gilt.

2.1 Zeichenreihen

Für das folgende brauchen wir die Vereinigung einer unendlichen Folge $M_0, M_1, M_2, \ldots$ von Mengen, die wir als $\bigcup_{n\geq 0} M_n$ schreiben. Diese Vereinigung ist die bezüglich der Inklusion kleinste Menge aus dem vorgegebenen Universum, die alle Mengen M_i, $i \geq 0$, enthält. Damit gilt $\bigcup_{n\geq 0} M_n = \{\ x : \exists\, n \in \mathbb{N} : x \in M_n\ \}$.

2.1.1 Definition (Zeichenreihen)

a) Ein **Zeichenvorrat** ist eine endliche, nichtleere Menge, deren Elemente **Zeichen** heißen.

b) Es sei A ein Zeichenvorrat. Zu $n \in \mathbb{N}$ ist A^n die **Menge der Zeichenreihen** über A **der Länge** n und $A^* := \bigcup_{n\geq 0} A^n$ die **Menge der Zeichenreihen** über A.

c) Mit $|w|$ bezeichnet man die **Länge** der Zeichenreihe w, d.h. es gilt $|w| = n$ genau dann, wenn $w \in A^n$ zutrifft. ■

Mathematisch gesehen ist eine Zeichenreihe also ein n-Tupel $\langle a_1, \ldots, a_n\rangle$ von Zeichen. Für $n = 0$ erhalten wir das leere Tupel $\langle\rangle$ und man definiert $A^+ := A^* \setminus \{\langle\rangle\}$. Statt von Zeichenreihen spricht man auch von Zeichenfolgen oder von **Worten**. Da Zeichenvorräte endlich sind, kann man sie in der Art $A = \{a_1, \ldots, a_n\}$ schreiben und damit anordnen, indem man festlegt $a_1 < a_2 < \ldots < a_n$. So angeordnete Zeichenvorräte nennt man auch **Alphabete**. Offensichtlich gewinnt man Zeichenreihen, indem man, ausgehend vom leeren Tupel, von rechts oder von links immer wieder Zeichen anfügt. Wir haben also:

2.1.2 Satz (Erzeugungsprinzipien)

Es sei A ein Zeichenvorrat. Dann ist die Menge A^* induktiv erzeugt durch die Basis $\{\langle\rangle\}$ und die Funktionen ($a \in A$)

$$\mathit{prefix}_a : A^* \longrightarrow A^* \qquad \mathit{prefix}_a(\langle a_1, \ldots, a_n\rangle) = \langle a, a_1, \ldots, a_n\rangle$$

bzw. durch die Basis $\{\langle\rangle\}$ und die Funktionen ($a \in A$)

$$\mathit{postfix}_a : A^* \longrightarrow A^* \qquad \mathit{postfix}_a(\langle a_1, \ldots, a_n\rangle) = \langle a_1, \ldots, a_n, a\rangle.$$ ■

Aus diesem Satz und Satz 1.2.6 bekommt man sofort, daß man über Zeichenreihen strukturelle Induktion nach ihrem Aufbau führen kann. Die entsprechenden Inferenzregeln sind im nachfolgenden Satz angegeben.

2.1.3 Satz (Induktion auf Zeichenreihen)

Auf der Menge A^* der Zeichenreihen über einem vorgegebenen Zeichenvorrat A gilt das Induktionsprinzip

$$\frac{P(\langle\rangle) \qquad \forall\, a \in A, w \in A^* : P(w) \to P(\mathit{prefix}_a(w))}{\forall\, w \in A^* : P(w)}$$

für den strukturellen Aufbau von Zeichenreihen durch das Anfügen eines Zeichens von links und das Induktionsprinzip

$$\frac{P(\langle\rangle) \qquad \forall\, a \in A, w \in A^* : P(w) \to P(\mathit{postfix}_a(w))}{\forall\, w \in A^* : P(w)}$$

für den strukturellen Aufbau von Zeichenreihen durch das Anfügen eines Zeichens von rechts. ■

Um eine Eigenschaft für alle Zeichenreihen durch Induktion zu beweisen, muß man sie also zuerst für das leere Tupel verifizieren und dann zeigen, daß ihre Gültigkeit beim Anfügen eines einzelnen Zeichens von links (erste Inferenzregel) bzw. von rechts (zweite Inferenzregel) erhalten bleibt. Eine Anwendung von Satz 2.1.3 wird im folgenden Beispiel gegeben. Dieses führt auch eine wichtige weitere Funktion auf A^* ein, die Konkatenation (Hintereinanderschreibung) von Zeichenreihen.

2.1.4 Beispiel (Konkatenation)

Die Funktion $\mathit{conc} : A^* \times A^* \longrightarrow A^*$, die sogenannte **Konkatenationsoperation**, ist induktiv wie folgt definiert:

$$\begin{array}{lrcl} (1) & \mathit{conc}(\langle\rangle, w_2) & = & w_2 \\ (2) & \mathit{conc}(\mathit{prefix}_a(w_1), w_2) & = & \mathit{prefix}_a(\mathit{conc}(w_1, w_2)) \end{array}$$

Durch diese beiden Gleichungen ist die Konkatenation von Zeichenreihen als zweistellige Funktion auf so eine Weise festgelegt, daß man sie für gegebene Argumente konkret „ausrechnen“ kann[2]. Wir führen dies nachfolgend anhand einer kleinen Beispielrechnung

[2]Wir haben damit ein Beispiel für einen Algorithmus, obwohl wir uns mit diesem Begriff erst am Ende dieses Kapitels beschäftigen werden. Dies ist aber nicht das erste Algorithmusbeispiel; auch die induktiven Definitionen der Potenzfunktion und der iterierten Funktionskomposition des letzten Kapitels stellen schon Algorithmen dar. Man mache sich dies klar, indem man etwa $\mathit{pow}(4,5)$ mittels der in Beispiel 1.2.7,b angegebenen Gesetze ausrechne.

durch. Dabei schreiben wir abkürzend p_a statt prefix_a.

$$\begin{array}{rcll} \mathit{conc}(\langle A,N\rangle,\langle A,N\rangle) & = & \mathit{conc}(p_A(p_N(\langle\rangle)),\langle A,N\rangle) & \text{Erzeugungsprinzip} \\ & = & p_A(\mathit{conc}(p_N(\langle\rangle),\langle A,N\rangle)) & \text{nach (2)} \\ & = & p_A(p_N(\mathit{conc}(\langle\rangle,\langle A,N\rangle))) & \text{nach (2)} \\ & = & p_A(p_N(\langle A,N\rangle)) & \text{nach (1)} \\ & = & p_A(\langle N,A,N\rangle) & \text{Erzeugungsprinzip} \\ & = & \langle A,N,A,N\rangle & \text{Erzeugungsprinzip} \end{array}$$

Nun führen wir einen Beweis durch Induktion. Wir zeigen, daß die Konkatenation **assoziativ** ist, d.h. die Eigenschaft

$$\forall\, u,v,w \in A^* : \mathit{conc}(\mathit{conc}(u,v),w) = \mathit{conc}(u,\mathit{conc}(v,w))$$

gilt. Der Beweis erfolgt durch strukturelle Induktion mit dem nachfolgenden Prädikat[3]:

$$P(u) \quad :\Longleftrightarrow \quad \forall\, v,w \in A^* : \mathit{conc}(\mathit{conc}(u,v),w) = \mathit{conc}(u,\mathit{conc}(v,w))$$

Im Vergleich zum Beispiel der Potenzfunktion führen wir ihn aber in der normalerweise in der Mathematik üblicheren Form, d.h. mit Gleichungsketten und im umgebenden Text erwähnten Allquantoren, statt mit Umformungen zwischen allquantifizierten Formeln.

Induktionsbeginn: Zum Beweis von $P(\langle\rangle)$ seien $v,w \in A^*$ beliebig vorgegeben. Dann bekommen wir:

$$\begin{array}{rcll} \mathit{conc}(\mathit{conc}(\langle\rangle,v),w) & = & \mathit{conc}(v,w) & \text{nach (1)} \\ & = & \mathit{conc}(\langle\rangle,\mathit{conc}(v,w)) & \text{nach (1)} \end{array}$$

Induktionsschluß: Wir haben zu zeigen, daß für ein beliebiges $a \in A$ die Eigenschaft $P(p_a(u))$ aus der Induktionshypothese $P(u)$ folgt, wobei p_a wieder als Abkürzung für prefix_a steht. Es seien also $a \in A$ und $v,w \in A^*$ beliebig gewählt. Dann gilt:

$$\begin{array}{rcll} \mathit{conc}(\mathit{conc}(p_a(u),v),w) & = & \mathit{conc}(p_a(\mathit{conc}(u,v)),w) & \text{nach (2)} \\ & = & p_a(\mathit{conc}(\mathit{conc}(u,v),w)) & \text{nach (2)} \\ & = & p_a(\mathit{conc}(u,\mathit{conc}(v,w))) & \text{wegen } P(u) \\ & = & \mathit{conc}(p_a(u),\mathit{conc}(v,w)) & \text{nach (2)} \end{array}$$

Also ist auch $P(p_a(u))$ gültig.

Das Gesetz (1) besagt, daß das leere Tupel $\langle\rangle$ **linksneutral** ist. Es gilt auch die **Rechtsneutralität**

$$\mathit{conc}(w_1,\langle\rangle) = w_1$$

des leeren Tupels bezüglich der Konkatenation, was man analog zum Beweis der Assoziativität leicht durch strukturelle Induktion zeigt. ■

Die bisherigen Bezeichnungen bei Zeichenreihen sind etwas schwerfällig und auch sehr buchstabenreich. Deshalb haben sich in der Literatur **einfachere Schreibweisen** etabliert. Wir zählen sie im folgenden auf:

[3]Man beachte, daß $\forall\, u \in A^* : P(u)$ genau die Assoziativität der Konkatenation ist

- Man notiert Zeichenreihen in der Form $a_1 \ldots a_n$ ohne Klammern und Kommata statt $\langle a_1, \ldots, a_n \rangle$. Manchmal schreibt man auch "$a_1 \ldots a_n$".
- Man schreibt ε statt $\langle\rangle$. Die Zeichenreihe ε heißt die **leere Zeichenreihe** oder das **leere Wort**. Für $A^* \setminus \{\varepsilon\}$ schreibt man, wie schon erwähnt, auch A^+.
- Bei der Konkatenation sind die Schreibweisen $w_1 w_2$ oder auch $w_1 \cdot w_2$ statt der funktionalen Notation $conc(w_1, w_2)$ üblich.
- Bei den Anfügeoperationen schreibt man $a \,\&\, w$ statt $prefix_a(w)$ und $w \,\&\, a$ statt $postfix_a(w)$. Gebräuchlich sind auch die Schreibweisen aw bzw. wa, wobei man dann aber nicht mehr zwischen einzelnen Zeichen und einelementigen Zeichenreihen unterscheidet.

Es seien $a \in A$ ein Zeichen und $w_1, w_2 \in A^*$ Zeichenreihen über A. Dann wird die Gleichung (2) von Beispiel 2.1.4 in der vereinfachenden Schreibweise zu

$$(a \,\&\, w_1) w_2 = a \,\&\, (w_1 w_2).$$

Durch eine einfache strukturelle Induktion läßt sich für die Konkatenationsoperation und das Anfügen von rechts die analoge Gleichung

$$w_1 (w_2 \,\&\, a) = (w_1 w_2) \,\&\, a$$

zeigen. Wir können somit also **Klammern sparen**, ohne daß Mehrdeutigkeiten auftreten. Zukünftig verwenden wir in der Regel die eben eingeführten vereinfachenden Notationen ε, & usw.

2.1.5 Definition (Weitere Grundoperationen auf Zeichenreihen)

Neben den bisher eingeführten totalen Operationen & (von links und von rechts) $\cdot$ und $|\ |$ haben wir noch die folgenden (wegen der leeren Zeichenreihe nur partiellen) Grundoperationen auf Zeichenreihen, die wir induktiv definieren:

a) Berechnung des ersten Elements mittels $top : A^* \longrightarrow A$ und Entfernen des ersten Elements mittels $rest : A^* \longrightarrow A^*$:

$$\begin{aligned} top(\varepsilon) &= \text{undef} & \qquad rest(\varepsilon) &= \text{undef} \\ top(a \,\&\, w) &= a & \qquad rest(a \,\&\, w) &= w \end{aligned}$$

b) Berechnung des letzten Elements mittels $bottom : A^* \longrightarrow A$ und Entfernen des letzten Elements mittels $lead : A^* \longrightarrow A^*$:

$$\begin{aligned} bottom(\varepsilon) &= \text{undef} & \qquad lead(\varepsilon) &= \text{undef} \\ bottom(a \,\&\, \varepsilon) &= a & \qquad lead(a \,\&\, \varepsilon) &= \varepsilon \\ w \neq \varepsilon \rightarrow bottom(a \,\&\, w) &= bottom(w) & \qquad w \neq \varepsilon \rightarrow lead(a \,\&\, w) &= a \,\&\, lead(w) \end{aligned}$$

Wir haben in diesen Festlegungen induktiv alles auf die leere Zeichenreihe ε und das Vorneanfügen abgestützt, genau wie bei der Definition der Konkatenation. Gleiches gelingt auch für die Längenoperation und das Anfügen von hinten, denn es gelten die Gesetze

$$|\varepsilon| = 0 \qquad\qquad |a \,\&\, w| = 1 + |w|$$

für alle $a \in A$, $w \in A^*$ und die Längenoperation und

$$\varepsilon \,\&\, a = a \,\&\, \varepsilon \qquad\qquad (b \,\&\, w) \,\&\, a = b \,\&\, (w \,\&\, a)$$

für alle $a, b \in A$, $w \in A^*$ und das Anfügen von hinten. Damit ist die gesamte Theorie der Zeichenreihen auf die Konstante ε und die Funktionen prefix_a abgestützt. Mit ε und den Konstruktoren $\mathit{postfix}_a$ geht dies natürlich auch. ■

Zeichenreihen kann man auch anordnen. Dies ist vom Lexikon oder Telephonbuch her bekannt, wo etwa die Reihenfolge

$$\ldots < \text{Aachen} < \text{Aar} < \ldots < \text{Ziegel} < \text{Ziegelstein} < \ldots$$

gilt, da im linken Fall da c < r zutrifft und im rechten Fall das Wort „Ziegel“ zu „Ziegelstein“ verlängert wurde. Auch in der Informatik ist diese spezielle Relation, die sogar eine Ordnung auf Zeichenreihen ist, von großer Wichtigkeit. Formal kann man sie wie folgt definieren.

2.1.6 Definition (Lexikographische Ordnung)

Es sei A ein Zeichenvorrat mit Ordnung $\leq$. Dann ist die **strikte lexikographische Ordnung** $<_{lex} \;\subseteq A^* \times A^*$ wie folgt definiert:

$$\begin{aligned} w_1 <_{lex} w_2 \;:\Longleftrightarrow\; & (\exists\, r \in A^+ : w_1 r = w_2) \vee \\ & (\exists\, l, r_1, r_2 \in A^* \,\exists\, a_1, a_2 \in A : \\ & \qquad w_1 = (l \,\&\, a_1) r_1 \wedge w_2 = (l \,\&\, a_2) r_2 \wedge a_1 < a_2) \end{aligned}$$

Gilt $w_1 = w_2$ oder $w_1 <_{lex} w_2$, so nennen wir die Zeichenreihe w_1 **lexikographisch kleiner oder gleich** als die Zeichenreihe w_2 und notieren dies als $w_1 \leq_{lex} w_2$. ■

Nehmen wir uns das obige Beispiel noch einmal vor, so haben wir mit der „üblichen“ Ordnung auf dem deutschen Alphabet (ohne Unterscheidung von Groß- und Kleinbuchstaben), d.h. a < b < ... < z, die Beziehung Aachen $<_{lex}$ Aar, weil hier der zweite Teil der Konjunktion der Definition 2.1.6 mit l = Aa, r_1 = hen, $r_2 = \varepsilon$, a_1 = c und a_2 = r eintritt. Weiterhin gilt die Anordnung Ziegel $<_{lex}$ Ziegelstein, denn hier trifft der erste Teil der Konjunktion in der Definition 2.1.6 mit der Zeichenreihe r = stein zu.

Wir werden nicht beweisen, daß die in Definition 2.1.6 eingeführte **lexikographische Ordnung** $\leq_{lex}$ tatsächlich eine Ordnung ist. Der Beweis ist technisch nicht allzu schwierig, aber doch sehr umständlich und langatmig, was durch die umfangreiche Definition mit den beiden Fällen bedingt ist. Nichtsdestoweniger wollen wir den Sachverhalt jedoch festhalten.

2.1.7 Satz (Lexikographische Ordnung)

Das Paar $(A^*, \leq_{lex})$ ist eine geordnete Menge und ε ist das kleinste Element von A^*. In A^* gibt es keine maximalen Elemente[4]. ■

Mit Texten kann man arbeiten, indem man Teile durch andere ersetzt, insbesondere ein- oder anfügt. Dies ist in der Informatik typisch für Editoren. Man ist nun interessiert am Arbeiten mit Texten, wobei feste Regeln vorgegeben sind. Die einfachste abstrakte Variante sind die Textersetzungssysteme. Diese führen dann etwa zu Grammatiken und formalen Sprachen, was wichtig ist zur **syntaktischen Beschreibung von Programmiersprachen**, und zu deren automatischer Übersetzung in maschinennähere Sprachen. Im weiteren geben wir die grundlegendsten Begriffe der Textersetzungssysteme an.

2.1.8 Definition (Textersetzung)

Im folgenden sei ein Zeichenvorrat A vorausgesetzt.

a) Ein **Textersetzungssystem** über A ist ein Paar $(A, \mathfrak{R})$, wobei $\mathfrak{R}$ eine Menge von Regeln $l \to r$ mit $l, r \in A^*$ ist. Damit Textersetzung einen Algorithmus beschreibt, nimmt man die Regelmenge $\mathfrak{R}$ als endlich an.

b) Zu einem Textersetzungssystem $(A, \mathfrak{R})$ ist die **Einschritt-Ersetzungsrelation** $\Rightarrow \subseteq A^* \times A^*$ definiert durch:

$$u \Rightarrow v \quad :\Longleftrightarrow \quad \exists\, l \to r \in \mathfrak{R}\, \exists\, a, e \in A^* : u = ale \wedge v = are$$

c) Zu einem Textersetzungssystem $(A, \mathfrak{R})$ ist die **Berechnungsrelation** $\overset{*}{\Rightarrow} \subseteq A^* \times A^*$ definiert durch

$$u \overset{*}{\Rightarrow} v \quad :\Longleftrightarrow$$
$$\exists\, n \in \mathbb{N}\, \exists\, w_0, \ldots, w_n \in A^* : u = w_0 \wedge w_0 \Rightarrow w_1 \wedge \ldots \wedge w_{n-1} \Rightarrow w_n \wedge w_n = v.$$

d) Eine Zeichenreihe $w \in A^*$ heißt eine **Normalform** zum Textersetzungssystem $(A, \mathfrak{R})$, falls es keine Zeichenreihe $u \in A^*$ mit $w \Rightarrow u$ gibt. Man sagt auch, w ist **terminal**. ■

Die Einschritt-Relation ersetzt beim Übergang von u zu v eine Teilzeichenreihe l von u durch die Zeichenreihe r, wobei das Paar l, r eine Regel bilden muß. Man sagt auch, daß v aus u durch Anwendung der Textersetzungsregel $l \to r$ entsteht und nennt den Vorgang **Berechnungsschritt**. Für Zeichenreihen u und v gilt $u \overset{*}{\Rightarrow} v$ genau dann, wenn v von u aus in endlich vielen Berechnungsschritten **herleitbar** (oder **erreichbar**) ist. In Definition 2.1.8.c ist auch $n = 0$ erlaubt, was besagt, daß die Berechnungsrelation $u \overset{*}{\Rightarrow} u$ für alle $u \in A^*$ erfüllt, d.h. reflexiv ist. Offensichtlich ist diese Relation auch transitiv, d.h. $u \overset{*}{\Rightarrow} v$ und $v \overset{*}{\Rightarrow} w$ impliziert $u \overset{*}{\Rightarrow} w$.

[4] Damit existiert auch kein größtes Element, da dieses maximal wäre. Beschränkt man sich hingegen auf Zeichenreihen mit einer festen Maximallänge, so existieren maximale Elemente.

2.1.9 Beispiel (für Textersetzung)

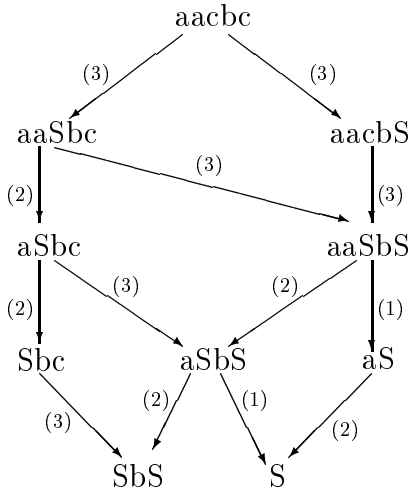
Wir betrachten ein Textersetzungssystem mit dem Zeichenvorrat

$$A = \{\ \mathrm{a}, \mathrm{b}, \mathrm{c}, \mathrm{S}\ \}$$

und der folgenden (durchnumerierten) Menge $\mathfrak{R}$ von Regeln:

$$(1)\ \ \mathrm{aSbS} \to \mathrm{S} \qquad (2)\ \ \mathrm{aS} \to \mathrm{S} \qquad (3)\ \ \mathrm{c} \to \mathrm{S}$$

Die von der Zeichenreihe aacbc ausgehenden Relationenbeziehungen $u \Rightarrow v$ kann man als Diagramm wie folgt darstellen. An den Pfeilen steht dabei die Nummer der beim jeweiligen Berechnungsschritt verwendeten Regel.



Damit haben wir insbesondere, wenn wir das Diagramm von oben nach unten bis zu den Normalformen durchlaufen:

$$\{\ w \in A^* \ :\ \mathrm{aacbc} \overset{*}{\Rightarrow} w \wedge w \text{ ist Normalform}\ \} = \{\mathrm{SbS},\ \mathrm{S}\}$$

Diese Menge der von w aus herleitbaren Normalformen kann man auch als Ausgabe des Textersetzungsalgorithmus zur Eingabe aacbc auffassen. ■

Zwei Fragestellungen in Verbindung mit Textersetzung (und anderen Ersetzungssystemen) sind nun wichtig.

a) **Eindeutigkeit der Ergebnisse aller Berechnungsfolgen:** Gilt für alle Zeichenreihen $v, w_1, w_2 \in A^*$ die Implikation

$$v \overset{*}{\Rightarrow} w_1 \wedge v \overset{*}{\Rightarrow} w_2 \wedge w_1, w_2 \text{ sind Normalformen} \implies w_1 = w_2?$$

b) **Terminierung mindestens einer Berechnungsfolge:** Gilt für alle Zeichenreihen $v \in A^*$ die Eigenschaft

$$\exists\, w \in A^* : v \overset{*}{\Rightarrow} w \wedge w \text{ ist Normalform?}$$

Sind die beiden Punkte a) und b) erfüllt, so beschreibt das vorliegende Textersetzungssystem (sogar wiederum in algorithmischer Weise) eine Funktion, indem man u auf die existierende und eindeutige Normalform v mit $u \overset{*}{\Rightarrow} v$ abbildet. Eindeutigkeit und Existenz einer erreichbaren Normalform müssen aber nicht immer gegeben ein. Das System aus Beispiel 2.1.9 verletzt etwa die Eindeutigkeir, also a). Ergänzt man es um die Regel

$$(4) \quad S \to c,$$

d.h. die Umkehrung der Regel (3), so ist auch der Punkt b) verletzt, d.h. die Terminierung, da für das Wort ca beispielsweise nur die unendliche Sequenz

$$\text{ca} \overset{(3)}{\Rightarrow} \text{Sa} \overset{(4)}{\Rightarrow} \text{ca} \overset{(3)}{\Rightarrow} \text{Sa} \overset{(4)}{\Rightarrow} \ldots$$

von Berechnungsschritten möglich ist. Genauer wird auf die Behandlung von a) und b) im Laufe des Informatikstudiums immer wieder eingegangen werden.

2.2 Darstellung von Daten durch Zeichenreihen

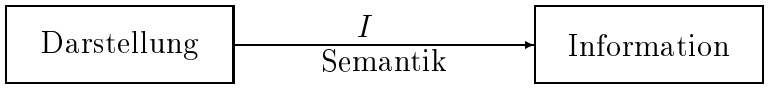
Zur Darstellung von Daten – oder allgemeiner von Information – verwendet man normalerweise Zeichenreihen. Dabei muß diese Darstellung aber nicht eindeutig sein. Folgende Beispiele bezeichnen etwa den gleichen abstrakten Begriff:

siebenundfünfzig	(Zahlwort in deutscher Sprache)
57	(Dezimaldarstellung)
L V II	(Darstellung als römische Zahl)
LLLOOL	(Binärdarstellung)

Man hat also zu unterscheiden zwischen einer Information und der ihr zugeordneten Bedeutung, die ein gewisses Bezugssystem voraussetzt. Dies führt auf die wichtigen Begriffe von Semantik und Interpretation, denen wir uns nun zuwenden wollen.

2.2.1 Erklärung: Semantik und Interpretation

Mit **Semantik** oder **Interpretation** bezeichnet man die Zuordnung von Information zu der gewählten Darstellung unter Verwendung eines **Bezugssystems**. In der Informatik hat man in der Regel die folgende Situation vorliegen:



Syntaktische Gebilde wie Zeichenreihen, Terme, Formeln, Programme, Spezifikationen usw.	Mathematische (semantische) Gebilde wie Mengen, Zahlen, Funktionen, Relationen, Algebren usw.

Ist $I : Syn \longrightarrow Sem$ die in obiger Graphik verwendete Funktion, so heißen $s_1, s_2 \in Syn$ **semantisch äquivalent**, falls $I[\![s_1]\!] = I[\![s_2]\!]$ gilt. Die Feststellung von semantischer Äquivalenz ist eine sehr bedeutende Fragestellung. Bei der Programmierung will man beispielsweise wissen, ob ein Programm semantisch äquivalent zu einer gegebenen Spezifikation ist, also genau das tut, was man von ihm erwartet. Auch die Frage nach der Gültigkeit einer Gleichung (oder einer Formel) ist die Frage nach semantischer Äquivalenz. ■

Bei Semantik hat sich in vielen Bereichen die **Schreibweise** $I[\![s]\!]$ mit Doppelklammerung durchgesetzt. Oft läßt man auch die Bezeichnung I weg und schreibt einfach $[\![s]\!]$ für die Semantik von s. Auch weitere Schreibweisen sind noch gebräuchlich, insbesondere in der mathematischen Logik.

Wir wollen im folgenden den Unterschied von Syntax und Semantik anhand der Darstellung von Zahlen demonstrieren. Dazu beginnen wir mit dem Zahlensystem, das für die technische Informatik fundamental ist.

2.2.2 Beispiel (Binärdarstellung, Dualsystem)

Die Menge der Binärzahlen (oder Zahlen des Dualsystems) ist gegeben durch $\{\mathrm{O}, \mathrm{L}\}^+$. Durch diese Festlegung sind offensichtlich auch führende Nullen wie in der Zeichenreihe OOOLL erlaubt. Wir kommen nun zur Semantik der Binärzahlen. Sie ist gegeben durch eine Funktion

$$[\![.]\!] : \{\mathrm{O}, \mathrm{L}\}^+ \longrightarrow \mathbb{N},$$

deren nachfolgende induktive Definition genau dem syntaktischen Aufbau der Binärworte durch das Anfügen von rechts folgt.

$$\begin{array}{lrcll}
(1) & [\![\mathrm{O}]\!] & = & 0 & \\
(2) & [\![\mathrm{L}]\!] & = & 1 & \\
(3) & [\![w\mathrm{O}]\!] & = & [\![w]\!] * 2 & \text{für } w \in \{\mathrm{O}, \mathrm{L}\}^+ \\
(4) & [\![w\mathrm{L}]\!] & = & [\![w]\!] * 2 + 1 & \text{für } w \in \{\mathrm{O}, \mathrm{L}\}^+
\end{array}$$

Auf der rechten Seite dieser Definition verwenden wir die für Zahlen übliche Dezimaldarstellung als Bezugssystem. Die Festlegung der Semantik durch die Gleichungen / Regeln (1) bis (4) entspricht somit genau der Umwandlung vom Dualsystem in das Dezimalsystem. Als Beispiel nehmen wir wiederum die Binärdarstellung von 57. Wir erhalten dann die Umwandlung gemäß der Rechnung

$$\begin{array}{lll}
[\![\mathrm{LLLOOL}]\!] = [\![\mathrm{LLLOO}]\!] * 2 + 1 & & \text{Regel (4)} \\
= [\![\mathrm{LLLO}]\!] * 2 * 2 + 1 & & \text{Regel (3)} \\
= [\![\mathrm{LLL}]\!] * 2 * 2 * 2 + 1 & & \text{Regel (3)} \\
= ([\![\mathrm{LL}]\!] * 2 + 1) * 2 * 2 * 2 + 1 & & \text{Regel (4)} \\
= (([\![\mathrm{L}]\!] * 2 + 1) * 2 + 1) * 2 * 2 * 2 + 1 & & \text{Regel (4)} \\
= ((1 * 2 + 1) * 2 + 1) * 2 * 2 * 2 + 1 & & \text{Regel (2)} \\
= (3 * 2 + 1) * 8 + 1 & & \\
= 7 * 8 + 1 & & \\
= 57, & &
\end{array}$$

wobei die letzten drei Schritte nur mehr der Vereinfachung dienen. ■

Dieses eben gebrachte Beispiel läßt sich einfach zur sogenannten p-adischen Zahldarstellung erweitern.

2.2.3 Beispiel (Die p-adische Zahldarstellung)

Gegeben seien $p > 0$ verschiedene Zeichen $z_0, \ldots, z_{p-1}$. Dann ordnet man jeder nichtleeren Zeichenreihe, die nur aus diesen Zeichen besteht, durch die Funktion

$$[\![.]\!] : \{\ z_0, \ldots, z_{p-1}\ \}^+ \longrightarrow \mathbb{N}$$

eine natürliche Zahl wie folgt zu:

$$\begin{array}{lrcll} (1) & [\![z_i]\!] & = & i & \text{für alle } i, 0 \leq i \leq p-1 \\ (2) & [\![w]\!] & = & [\![lead(w)]\!] * p + [\![bottom(w)]\!] & \text{falls } |w| > 1 \end{array}$$

In diesen Gleichungen sind *lead* und *bottom* die Funktionen auf Zeichenreihen, wie sie in Definition 2.1.5 eingeführt wurden. Man beachte weiterhin, daß in Gleichung (1) die einzelnen Ziffern als Zeichenreihen der Länge 1 aufgefaßt werden; ansonsten benötigt man eine Verbindung zwischen den Ziffern und den entsprechenden Zahlen mittels einer Funktion, etwa *val* genannt, und hat (1) durch $[\![z_i]\!] = val(z_i)$ zu ersetzen. Für $p = 2$ und $z_0 = \mathrm{O}, z_1 = \mathrm{L}$ erhalten wir genau die Binärdarstellung. ■

Die Umwandlung von Zahldarstellungen ist eine der einfachsten Formen von Semantik und Interpretation. Weitere Arten sind etwa die Definition des Werts eines Terms (Ausdrucks), die Festlegung, wann eine Formel gültig ist und die verschiedenen Semantiken von Programmiersprachen. Mit dem ersten Punkt werden wir uns im nächsten Abschnitt näher befassen. Der zweite Punkt wird grundlegend in der Logik und verwandten Gebieten behandelt, so in der Informatik beispielsweise bei den sogenannten algebraischen Spezifikationen. Die verschiedenen Semantiken von Programmiersprachen sind schließlich das Thema einer entsprechenden Vorlesung im Hauptstudium. Nichtsdestoweniger werden wir auch in diesem Skriptum bei der Einführung von ML und Java etwas auf ihre semantischen Festlegungen eingehen, jedoch nicht in großer mathematischer Gründlichkeit.

2.3 Signaturen und Terme

Terme – auch Ausdrücke genannt – sind Zeichenreihen, die nach bestimmten Gesetzen aufgebaut sind. Etwa sind die nachfolgenden Zeichenreihen Terme („über Zahlen“):

$$(3 * 4) + 12 \qquad (8 * 7) + 23 * 19 - 1 \qquad 3^3 - (2 * 1)$$

Jedoch wird man die drei folgenden Zeichenreihen nicht als Terme anerkennen, da sie nicht „wohlgeformt“ sind, was an den entsprechenden Stellen jeweils durch den senkrechten Pfeil

angezeigt ist:

$$\begin{array}{lll} (3*4)+ & (8*7\ +23*19-1 & {}^{3}-(2*1) \\ \uparrow & \uparrow & \uparrow \end{array}$$

In diesem Abschnitt legen wir formal fest, was Terme sind, und geben die Semantik (hier: den Wert oder die Wertberechnung) an. Auch auf die Manipulation von Termen durch Termersetzung, eine Erweiterung von Textersetzung, gehen wir ein. Am Anfang der Termfestlegung steht der Begriff der Signatur. Durch Signaturen wird das Material für den Termaufbau bereitgestellt.

2.3.1 Definition (Signaturen)

Eine **Signatur** $\Sigma = (S, K, F)$ besteht aus drei disjunkten Mengen S, K, F, wobei die folgenden Eigenschaften gelten:

a) Die Menge S ist nicht leer. Ein Element $s \in S$ heißt **Sorte** (oder auch **Typ** oder, in etwas älterer Sprechweise, **Art**).

b) Jedem $c \in K$ ist genau ein Element $s \in S$ zugeordnet. Das Element c heißt dann **Konstantensymbol** (kürzer auch **Konstante**) der Sorte s.

c) Jedem $f \in F$ ist genau ein Element $w \in S^+$ und genau ein Element $s \in S$ zugeordnet. Das Element f heißt dann **Funktionssymbol** (kürzer auch **Operation**) mit **Argumentsorten** w, **Resultatsorte** s und **Funktionalität** $w \to s$. ■

Zur Darstellung von Signaturen gibt es verschiedene Schreibweisen. Im folgenden geben wir bei Signaturen bei den Mengen K und F die jeweilige Sorte bzw. Funktionalität eines Elements immer in der Form $c : s$ bzw. $f : s_1, \ldots, s_n \to s$ mit an. Diese Schreibweise der durch den Doppelpunkt abgetrennten **nachgestellten Typisierung** ist auch bei der von uns später verwendeten Programmiersprache ML üblich. Die Sprache Java stellt hingegen die Typen voran.

2.3.2 Beispiele (für Signaturen)

a) Betrachten wir die natürlichen Zahlen mit Null und der Nachfolgerbildung, so führt dies auf eine Signatur $\Sigma_{\mathbb{N}} = (S, K, F)$, deren Mengen wie folgt aussehen:

$$\begin{array}{lcl} S & = & \{\ nat\ \} \\ K & = & \{\ zero : nat\ \} \\ F & = & \{\ succ : nat \to nat\ \} \end{array}$$

Die oben beschriebene beabsichtigte Bedeutung der Bestandteile dieser Signatur haben wir durch deren Namensgebung deutlich gemacht.

b) Nun betrachten wir Zeichenreihen aus A^* mit den in Definition 2.1 eingeführten Operationen. Setzen wir für den Zeichenvorrat A genau die Zeichen c_1 bis c_n voraus, so führt dies zu der nachfolgenden Signatur $\Sigma_{A^*} = (S, K, F)$:

$$\begin{aligned} S &= \{\ \mathit{char}, \mathit{string}\ \} \\ K &= \{\ c_1 : \mathit{char}, \\ &\qquad \ldots \\ &\qquad c_n : \mathit{char}, \\ &\qquad \mathit{empty} : \mathit{string}\ \} \\ F &= \{\ \mathit{top} : \mathit{string} \to \mathit{char}, \\ &\qquad \mathit{bottom} : \mathit{string} \to \mathit{char}, \\ &\qquad \mathit{rest} : \mathit{string} \to \mathit{string}, \\ &\qquad \mathit{lead} : \mathit{string} \to \mathit{string}, \\ &\qquad \mathit{prefix} : \mathit{char}, \mathit{string} \to \mathit{string}, \\ &\qquad \mathit{postfix} : \mathit{string}, \mathit{char} \to \mathit{string}\ \} \end{aligned}$$

Bei Signaturen verwendet man in der Zeichenreihe der Argumentsorten normalerweise Kommata zur Separierung, da man es oft mit längeren Sortennamen zu tun hat. ■

Relationen kann man mit Signaturen auch darstellen, indem man $R \subseteq M \times N$ durch die **charakteristische Funktion**

$$\mathcal{X}_R : M \times N \longrightarrow \mathbb{B} \qquad \mathcal{X}_R(x, y) = tt \quad :\Longleftrightarrow \quad \langle x, y \rangle \in R$$

beschreibt. Ist r das Funktionssymbol zu $\mathcal{X}_R$ und sind m und n die Sorten zu M und N, so führt dies in der Menge F der Funktionssymbole der vorliegenden Signatur zu der Angabe $r : m, n \to \mathit{bool}$ mit der Sorte $\mathit{bool} \in S$ für die Wahrheitswerte. Mit den hier verwendeten Sprechweisen wie „das Funktionssymbol bzw. die Sorte zu“ haben wir genaugenommen schon auf die Semantik von Funktionssymbolen und Sorten Bezug genommen. Im Detail wird dies etwas später behandelt.

Ist eine Signatur gegeben, so kann man darüber Terme aufbauen, indem man, ausgehend von den Konstantensymbolen und gewissen vorgegebenen Variablen, iterativ die Funktionssymbole „anwendet“. Beispiele für Terme sind

$$\mathit{zero} \qquad \mathit{succ}(\mathit{zero}) \qquad \mathit{succ}(\mathit{succ}(\mathit{zero}))$$

über der Signatur $\Sigma_{\mathbb{N}}$, oder auch

$$\mathit{top}(\mathit{prefix}(c_i, \mathit{empty})) \qquad \mathit{postfix}(\mathit{prefix}(c_i, \mathit{empty}), c_j)$$

über der Signatur Σ_{A^*}. Dabei beschreiben die ersten drei Terme Zahlen, der vierte Term ein Zeichen und der fünfte Term eine Zeichenreihe. Die Terme sind also jeweils mit einer Sorte versehen, man sagt auch typisiert.

Im folgenden definieren wir Terme induktiv. Aus Gründen der Lesbarkeit verwenden wir dabei die nach Satz 1.2.6 eingeführte leichter verstehbare „übliche“ Schreibweise und lassen auch die letzte Bedingung „es gibt keine weiteren Terme ...“ weg.

2.3.3 Definition (Terme)

Es seien $\Sigma = (S, K, F)$ eine Signatur und X eine zu S, K, F disjunkte Menge von **Variablen**, wobei jedem $x \in X$ genau eine Sorte zugeordnet ist[5]. Die Menge $\mathcal{T}_\Sigma^s(X)$ der **Σ-Terme der Sorte $s \in S$ mit Variablen aus X** ist induktiv wie folgt definiert:

a) Für jedes Konstantensymbol c der Sorte s gilt $c \in \mathcal{T}_\Sigma^s(X)$.

b) Für jede Variable x der Sorte s gilt $x \in \mathcal{T}_\Sigma^s(X)$.

c) Sind f ein Funktionssymbol der Funktionalität $s_1 \dots s_k \to s$ und $t_i \in \mathcal{T}_\Sigma^{s_i}(X)$ für alle $i, 1 \leq i \leq k$, so gilt $f(t_1, \dots, t_k) \in \mathcal{T}_\Sigma^s(X)$. ■

Nach a) und b) sind alle Variablen und alle Konstanten also Terme und aus Termen bekommt man, nach c), durch Anwendungen von Funktionssymbolen wiederum Terme. Beim Aufbau von Termen nach dieser Definition müssen in Punkt c), der Anwendung eines Funktionssymbols, dabei jedoch sowohl die Stelligkeit (Anzahl der Argumente) als auch die Typisierung (Sorten der Argumente) passen.

2.3.4 Bemerkung (zur Darstellung von Termen)

In der Mathematik und in vielen Programmiersprachen haben sich spezielle Schreibweisen und Konventionen eingebürgert, welche die Lesbarkeit von Termen verbessern. Nachfolgend geben wir zwei Beispiele hierzu an:

a) Man verwendet **Infix-, Präfix- und Postfixnotationen** usw. statt der funktionalen Notation der Definition. So schreibt man etwa vereinfachend

$$\begin{array}{lll} 23 - (22 * 12) & \text{statt} & minus(23, mult(22, 12)) \\ (\neg\varphi) \wedge \psi & \text{statt} & and(neg(\varphi), \psi) \\ (R^\top) \cap S & \text{statt} & meet(transp(R), S). \end{array}$$

b) Zur Verringerung der Klammeranzahl legt man zusätzlich **Vorrangregeln** fest, wie beispielsweise das von der Schule her bekannte „Punkt vor Strich“. Mit ihrer Hilfe schreibt man etwa vereinfachend

$$\begin{array}{lll} 23 - 22 * 12 & \text{statt} & 23 - (22 * 12) \\ \neg\varphi \wedge \psi & \text{statt} & (\neg\varphi) \wedge \psi \\ R^\top \cap S & \text{statt} & (R^\top) \cap S. \end{array}$$ ■

Auch wir werden in Zukunft bei den „konkreten“ Beispielen die gewohnte mathematische Notation verwenden. Programmiersprachen halten sich teilweise auch an diese Notationen, teilweise weichen sie hingegen auch von ihnen ab. Nachfolgend geben wir einige Beispiele für die Darstellung von Termen in einigen bekannten Programmiersprachen an.

[5] Man sagt dann, daß $x \in X$ eine Variable der Sorte $s \in S$ ist.

2.3.5 Beispiele (für Termdarstellung)

Wir betrachten, in der gewohnten mathematischen Notation, den Term

$$2^5 - 1.$$

In der Programmiersprache Algol 60, einem Vorläufer von Java aus den sechziger Jahren, ist eine Potenzierung syntaktisch vorgesehen. Obiger Term schreibt sich hier als

$$2 * * 5 - 1.$$

Nun zu der funktionalen Sprache Lisp, die ebenfalls in den sechziger Jahren entwickelt wurde und, im Gegensatz zu Algol 60, derzeit immer noch verwendet wird[6]. Auch hier enthält die Syntax eine Konstruktion zur Potenzierung. Da sich die Sprache strikt an die funktionale Notation mit äußerer Klammerung hält, bekommen wir für den obigen Term

$$(SUB\ (POWER\ 2\ 5)\ 1)$$

als Darstellung. In den Sprachen ML und Java, die wir später verwenden werden, ist keine syntaktisch vordefinierte Konstruktion für die Potenzierung vorhanden. Sie ist nur möglich durch den Aufruf einer benutzerdefinierten Rechenvorschrift mit Namen (einem „Bezeichner"), sagen wir, *power*. Für den Beispielterm erhalten wir also die Darstellung

$$power(2, 5) - 1.$$

Natürlich darf statt *power* auch ein anderer Bezeichner verwendet werden. ■

In der folgenden Definition führen wir einige weitere Bezeichnungen und Abkürzungen für gewisse Mengen von Termen ein, die wir später noch verwenden werden.

2.3.6 Definition (Gewisse Termmengen)

a) Die Vereinigung $\bigcup_{s \in S} \mathcal{T}_\Sigma^s(X)$ heißt die **Menge aller Σ-Terme** und wird mit $\mathcal{T}_\Sigma(X)$ bezeichnet.

b) Ein Term $t \in \mathcal{T}_\Sigma(\emptyset)$ heißt **geschlossen**, **ohne Variable** oder ein **Σ-Grundterm**.

c) $\mathcal{T}_\Sigma^s$ ist die Menge der **Σ-Grundterme** der Sorte $s \in S$ und $\mathcal{T}_\Sigma = \bigcup_{s \in S} \mathcal{T}_\Sigma^s$ die Menge der **Σ-Grundterme**. ■

Terme beschreiben Werte. Etwa beschreibt $3 + 4 * 5$ die Zahl 23. Sie können aber auch undefiniert sein, wie etwa $\frac{1}{0}$ oder $\log 0$. Schließlich muß man auch noch die Variablen betrachten. Der Wert von $x + 25$ kann nur in Abhängigkeit des Werts von x berechnet werden. Belegt man beispielsweise x mit 5, so bekommt man 30 als Wert. All diese Phänomene hat man bei einer formalen Definition der Semantik eines Terms zu berücksichtigen. Beginnen wir mit der Basis, der Interpretation einer Signatur.

[6] Die Hauptanwendungsgebiete von Lisp sind die symbolische Datenverarbeitung und die sogenannte künstliche Intelligenz. Im Unterschied dazu, wurde Algol 60 mehr zur Lösung von numerischen Problemen konzipiert.

2.3.7 Definition (Σ-Algebren)

Es sei $\Sigma = (S, K, F)$ eine Signatur. Eine **Σ-Algebra** $A = (S^A, K^A, F^A)$ besteht aus drei Mengen, nämlich

a) einer Menge $S^A = \{\ s^A\ :\ s \in S\ \}$, wobei jedes s^A eine nichtleere (Träger-)Menge ist,

b) einer Menge $K^A = \{\ c^A\ :\ c \in K\ \}$, wobei jedes c^A ein Element aus s^A mit s als Sorte von c ist,

c) einer Menge $F^A = \{\ f^A\ :\ f \in F\ \}$, wobei jedes Element f^A eine partielle Funktion $f^A : \prod_{i=1}^{k} s_i^A \longrightarrow s^A$ mit $s_1, \ldots, s_k \rightarrow s$ als der Funktionalität von f ist. ■

Man hat also bei den Objekten der Signaturen die folgende Zuordnung, welche man als Interpretation der Signaturen bezeichnet:

Syntax	Semantik
Sorte	Trägermenge
Konstantensymbol	Element aus der entsprechenden Trägermenge
Funktionssymbol	Partielle Funktion zwischen den entsprechenden Trägermengen

Zur Interpretation von Termen mit Variablen brauchen wir noch Belegungen, die jeweils einer Variablen einen Wert zuordnen. Wir legen diese wie folgt fest:

2.3.8 Definition (Belegungen)

Es seien $\Sigma = (S, K, F)$ eine Signatur, $A = (S^A, F^A, K^A)$ eine Σ-Algebra und X eine Menge von Variablen mit $x \in X$ jeweils von einer Sorte $s \in S$. Eine **Belegung** (von X in A) ist eine Funktion $b : X \longrightarrow \bigcup_{s \in S} s^A$ mit der Eigenschaft, daß $b(x) \in s^A$ genau dann, wenn s die Sorte von x ist, für alle $x \in X$ gilt. ■

Eine Belegung b ordnet also einer Variablen x ein (semantisches) Objekt $b(x)$ zu, so daß die Typisierung erhalten bleibt. Nach diesen Vorbereitungen können wir nun formal definieren, was der Wert eines Terms ist.

2.3.9 Definition (Werte von Termen)

Es seien die Signatur Σ, die Σ-Algebra A und die Variablenmenge X wie in Definition 2.3.8 vorausgesetzt. Der **Wert** $I[t]_b^A$ eines Terms t zur Belegung b in A ist induktiv wie folgt definiert:

a) Für alle Konstantensymbole $c \in K$ gilt

$$I[c]_b^A = c^A.$$

b) Für alle Variablen $x \in X$ gilt

$$I[x]_b^A = b(x).$$

c) Für alle zusammengesetzten Terme der Form $f(t_1, \ldots, t_k)$ gilt

$$I[f(t_1, \ldots, t_k)]_b^A = f^A(I[t_1]_b^A, \ldots, I[t_k]_b^A). \qquad \blacksquare$$

Ist t ein Term der Sorte s, so ist der Wert von t ein Element aus s^A oder undefiniert. In der Definition 2.3.9 stützen wir uns in c) auf die mathematische Festlegung der Funktionsanwendung bei möglicherweise partiellen Funktionen, wie sie sich aus der Definition von ihnen als spezielle Relationen ergibt. Definiertheit des Werts gilt im Falle eines zusammengesetzten Terms also genau dann, falls erstens alle Argumente definierte Werte haben und zweitens zusätzlich auch die Anwendung der Interpretation des Funktionssymbols für diese Werte definiert ist.

Um zu demonstrieren, wie man mit den bisher gebrachten formalen Definitionen mathematisch argumentieren kann, beweisen wir einen bekannten Satz über Terme, der besagt, daß der Wert eines Terms nur von der Belegung der in ihm vorkommenden Variablen abhängt.

2.3.10 Satz (Koinzidenzlemma)

Gilt die Gleichung $b_1(x) = b_2(x)$ für alle Variablen x, die in dem Term $t \in \mathcal{T}_\Sigma(X)$ vorkommen, so hat man

$$I[t]_{b_1}^A = I[t]_{b_2}^A.$$

Beweis: Wir führen eine strukturelle Induktion über den Aufbau von t.

Induktionsbeginn: Ist t eine Konstante $c \in K$, so zeigt die Definition des Werts von c die Gleichung

$$I[c]_{b_1}^A = c^A = I[c]_{b_2}^A.$$

Der zweite Fall des Induktionsbeginns ist, daß t eine Variable $x \in X$ darstellt. Dann kommt x natürlich in x vor, denn es gilt sogar Gleichheit. Also erhalten wir, wegen der Annahme $b_1(x) = b_2(x)$, das gewünschte Resultat

$$I[x]_{b_1}^A = b_1(x) = b_2(x) = I[x]_{b_2}^A.$$

Induktionsschluß: Nun habe t die Form $f(t_1, \ldots, t_k)$. Dann bekommen wir

$$\begin{array}{rcll} I[f(t_1, \ldots, t_k)]_{b_1}^A & = & f^A(I[t_1]_{b_1}^A, \ldots, I[t_k]_{b_1}^A) & \text{Definition des Werts} \\ & = & f^A(I[t_1]_{b_2}^A, \ldots, I[t_k]_{b_2}^A) & \text{wegen der Induktionshypothese} \\ & = & I[f(t_1, \ldots, t_k)]_{b_2}^A & \text{Definition des Werts,} \end{array}$$

da die Voraussetzung offensichtlich auch für die Terme t_1 bis t_k gilt. ■

Insbesondere ist also für einen Grundterm $t \in \mathcal{T}_\Sigma$ der Wert $I[t]_b^A$ unabhängig von b. Man schreibt deshalb bei Grundtermen kürzer $I[t]^A$. Oft wird auch das Symbol I noch weggelassen. Dann hat man t^A als den Wert des Grundterms t.

Zum Schluß dieses Abschnitts gehen wir jetzt noch auf die Termersetzung ein, die eine Erweiterung der Textersetzung darstellt. Sie wird später für das operationelle Verständnis der funktionalen Sprache ML (in etwas abgewandelter Form) eine große Rolle spielen. Wir beginnen mit der formalen Definition solcher Systeme:

2.3.11 Definition (Termersetzungssysteme)

Ein **Termersetzungssystem** ist ein Tripel $(\Sigma, X, \mathfrak{R})$ mit

a) einer Signatur Σ,

b) einer Variablenmenge X, wobei die Sorten der Variablen $x \in X$ aus der Sortenmenge S von Σ sind, und

c) einer Menge $\mathfrak{R}$ von Regeln der Form $l \to r$ mit $l, r \in \mathcal{T}_\Sigma(X)$, so daß die Terme l, r gleiche Sorte haben und jede Variable von r auch in l vorkommt. Wie in Definition 2.1.8 nimmt man aus algorithmischen Gründen die Regelmenge $\mathfrak{R}$ wiederum als endlich an. ■

Eine Bemerkung zur sogenannten **Variablenbedingung** in Punkt c) dieser Definition ist angebracht. Würde in der rechten Seite r einer Regel $l \to r$ eine im Vergleich zu linken Seite l neue Variable vorkommen, so wäre bei einer algorithmischen Abarbeitung – und daran ist man bei Termersetzungssystemen, wie bei den Textersetzungssystemen auch, hauptsächlich interessiert – nicht klar, was man später für diese zu setzen hat. Deshalb verbietet man durch diese Forderung die sogenannten **Extravariablen** in der rechten Seite einer Regel.

Einschritt-Berechnungsrelation und Berechnungsrelation sind bei Termersetzung analog zur Textersetzung definiert, nur daß man jetzt Teilterme anhand der Regeln verändert. Dazu brauchen wir den Begriff der Substitutionen.

2.3.12 Definition (Substitutionen)

a) Zu einem Termersetzungssystem $(\Sigma, X, \mathfrak{R})$ heißt eine Funktion $\sigma : X \longrightarrow \mathcal{T}_\Sigma(X)$ eine **Variablensubstitution**, falls $\sigma(x)$ von gleicher Sorte wie x ist. Ist X unendlich, so fordert man noch, daß $\sigma(x)$ nur für endlich viele $x \in X$ ungleich x ist.

b) Eine Variablensubstitution σ heißt **Grundsubstitution**, falls $\sigma(x) \in \mathcal{T}_\Sigma$ für alle $x \in X$ mit $\sigma(x) \neq x$ gilt. ■

Im Gegensatz zu einer Belegung ordnet eine Substitution einer Variablen ein syntaktisches Objekt (gleicher Sorte) zu. Die nächsten Begriffe, die zur formalen Festlegung von Termersetzung notwendig sind, erklären wir nur informell. Sie können formal alle induktiv über den Aufbau der Terme definiert werden.

2.3.13 Einige Erklärungen zur Termersetzung

a) Mit $t\sigma$ bezeichnen wir die **Anwendung der Variablensubstitution** σ auf den Term t. Sie ersetzt simultan in t jede Variable $x \in X$ durch den ihr durch σ zugeordneten Term $\sigma(x)$. Haben wir beispielsweise $X = \{\ x, y, z\ \}$ als Variablenmenge,

$$t = f(x, g(x, y), h(c))$$

als Term und die Substitution σ mit

$$\sigma(x) = h(c) \qquad \sigma(y) = g(y, c) \qquad \sigma(z) = z,$$

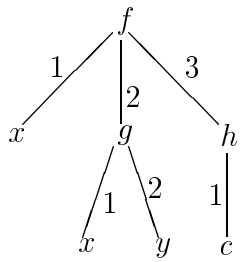
so bekommen wir als Resultat der Anwendung

$$t\sigma = f(h(c), g(h(c), g(y, c)), h(c)).$$

b) Mit $Occ(t)$ bezeichnen wir die sogenannten **Stellen** eines Terms t, das sind Sequenzen von natürlichen Zahlen, die alle Teilterme von t bezeichnen. Für den speziellen Term t aus a) bekommen wir

$$Occ(t) = \{\ \varepsilon, 1, 2, 21, 22, 3, 31\ \}.$$

Dies wird besonders klar, wenn man den Term graphisch als sogenannten Baum darstellt. Man bekommt alle Stellen, wenn man alle möglichen Wege in diesem Bild vom Einstiegspunkt f (der „Wurzel“) ausgehend durchläuft und dabei die an den Kanten stehenden Markierungen aufsammelt.



c) Zu einer Stelle $z \in Occ(t)$ bezeichnet t/z den **Teilterm** von t an der Stelle z. Wiederum für den Beispielterm t von a) und seine in b) angegebene Stellenmenge haben wir beispielsweise

$$t/2 = g(x, y) \qquad t/31 = c \qquad t/\varepsilon = f(x, g(x, y), h(c)).$$

In der Baumdarstellung ergeben sich für diese drei Teilterme drei Teilbäume des Baums von Punkt b), wobei der Teilbaum zu $t/31$ zu einem einzelnen mit dem Konstantensymbol c markierten „Knoten“ entartet.

d) Zu einer Stelle $z \in Occ(t)$ und einem Term $t_1 \in \mathcal{T}_\Sigma(X)$, so daß t/z und t_1 die gleiche Sorte haben, bezeichnet $t[t_1/z]$ die **Ersetzung** von t/z in t durch t_1. Wählen wir wiederum unseren Beispielterm, so gelten etwa die Gleichungen

$$t[h(x)/1] = f(h(x), g(x,y), h(c)) \qquad t[h(x)/21] = f(x, g(h(x), y), h(c)). \quad \blacksquare$$

Nach diesen Erklärungen sind wir nun in der Lage, formal zu definieren, was Termersetzung heißt. Da man Zeichenreihen als spezielle Terme auffassen kann, indem man sie beispielsweise durch ein Funktionssymbol nach und nach aufbaut, ist auch klar, daß Textersetzung ein Spezialfall von Termersetzung ist.

2.3.14 Definition (Termersetzungsrelationen)

Gegeben sei ein Termersetzungssytem $(\Sigma, X, \mathfrak{R})$. Man legt fest:

a) Eine Regel $l \to r \in \mathfrak{R}$ ist **anwendbar** auf einen Term $t \in \mathcal{T}_\Sigma$, falls es eine Stelle $z \in Occ(t)$ und eine Variablensubstitution $\sigma : X \longrightarrow \mathcal{T}_\Sigma$ gibt, mit $l\sigma = t/z$. Man sagt auch, daß die Textersetzungsregel $l \to r$ in t auf t/z paßt.

b) Die **Einschritt-Ersetzungsrelation** $\Rightarrow \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ ist definiert durch

$$\begin{aligned} t_1 \Rightarrow t_2 \quad :\Longleftrightarrow \\ \exists\, l \to r \in \mathfrak{R}\, \exists\, z \in Occ(t_1)\, \exists\, \sigma : X \longrightarrow \mathcal{T}_\Sigma : l\sigma = t_1/z \wedge t_2 = t_1[r\sigma/z]. \end{aligned}$$

c) Die **Berechnungsrelation** $\overset{*}{\Rightarrow} \subseteq \mathcal{T}_\Sigma \times \mathcal{T}_\Sigma$ auf den Grundtermen über der Signatur Σ ist definiert durch

$$\begin{aligned} t_1 \overset{*}{\Rightarrow} t_2 \quad :\Longleftrightarrow \\ \exists\, n \in \mathbb{N}\, \exists\, e_0, \ldots, e_n \in \mathcal{T}_\Sigma : t_1 = e_0 \wedge e_0 \Rightarrow e_1 \wedge \ldots \wedge e_{n-1} \Rightarrow e_n \wedge e_n = t_2. \end{aligned} \quad \blacksquare$$

Die Konstruktion in Punkt c) dieser Festlegung ist vollkommen analog zur Konstruktion in Definition 2.1.8.c bei der Textersetzung. Dem liegt ein allgemeines Prinzip auf Relationen zugrunde, das wir nun einführen.

2.3.15 Definition (Relationale Hüllenbildungen)

Ist R eine homogene Relation, so definiert man induktiv $R^0 := \mathsf{I}$ und $R^{i+1} := RR^i$ als **Potenzen** von R und nennt $R^* := \bigcup_{n \geq 0} R^n$ die **reflexiv transitive Hülle von** R. Die Relation $R^+ := RR^* = \bigcup_{n \geq 1} R^n$ heißt die **transitive Hülle** von R. $\blacksquare$

Es gilt $\langle x, y \rangle \in R^*$ genau dann, wenn es $n \in \mathbb{N}$ und $z_0, \ldots, z_n$ mit $z_0 = x, z_n = y$ und $\langle z_i, z_{i+1} \rangle \in R$ für alle $i, 0 \leq i \leq n-1$, gibt. Man beachte auch, daß $\langle x, x \rangle \in R^*$ für alle Elemente x zutrifft. Hier ist $n = 0$ und $x = z_0 = y$. Die Allquantifizierung gilt

auch, da der Indexbereich leer ist. Die reflexiv-transitive Hülle R^* ist die kleinste reflexive und transitive Relation, die R enthält; die transitive Hülle R^+ ist die kleinste transitive Relation, die R enthält.

Nach diesem Einschub kehren wir nun zurück zu den Termersetzungssystemen. Wir geben nachfolgend zur Verdeutlichung einige Beispiele an, in denen auch ihr algorithmischer Charakter deutlich zum Ausdruck kommt. Zuvor sollte jedoch noch bemerkt werden, daß der Begriff einer **Normalform** bei Termersetzung vollkommen analog zur Texteresetzung definiert ist. Weiterhin übertragen sich auch die bei der Textersetzung eingeführten Sprechweisen wie etwa Berechnungsschritt, Herleitbarkeit usw. Schließlich sind natürlich auch die bei der Textersetzung gestellten Fragen der Eindeutigkeit der Ergebnisse aller Berechnungsfolgen und der Terminierung mindestens einer Berechnungsfolge für Termersetzung wesentlich, wenn man beabsichtigt, durch Termersetzungssysteme algorithmisch Funktionen zu realisieren.

2.3.16 Beispiele (für Termersetzung)

a) Wir setzen eine Signatur für die natürlichen Zahlen voraus mit der Sorte nat, gewissen Konstantensymbolen $0, 1, \ldots$ der Sorte nat als Darstellungen für Zahlen, gewissen Funktionssymbolen $succ, +, *$ usw. (jeweils mit der üblichen Funktionalität versehen) für die gängigsten arithmetischen Operationen, sowie einem speziellen Funktionssymbol $fac : nat \to nat$. Die Menge der Variablen beinhalte eine Variable n der Sorte nat. Als Termersetzungsregeln seien schließlich vorhanden:

(1) $fac(0) \to 1$
(2) $fac(succ(n)) \to succ(n) * fac(n)$
(3) „Vereinfachungsregeln“ der Arithmetik

Eine Beispielrechnung sieht wie folgt aus (mit s als Abkürzung für $succ$):

$$
\begin{aligned}
fac(s(s(s(s(0))))) &\overset{(2)}{\Rightarrow} s(s(s(s(0)))) * fac(s(s(s(0))))\\
&\overset{(2)}{\Rightarrow} s(s(s(s(0)))) * s(s(s(0))) * fac(s(s(0)))\\
&\overset{(2)}{\Rightarrow} s(s(s(s(0)))) * s(s(s(0))) * s(s(0)) * fac(s(0))\\
&\overset{(2)}{\Rightarrow} s(s(s(s(0)))) * s(s(s(0))) * s(s(0)) * s(0) * fac(0)\\
&\overset{(1)}{\Rightarrow} s(s(s(s(0)))) * s(s(s(0))) * s(s(0)) * s(0) * 1\\
&\vdots \quad \text{Vereinfachungsschritte}\\
&\overset{(3)}{\Rightarrow} 24
\end{aligned}
$$

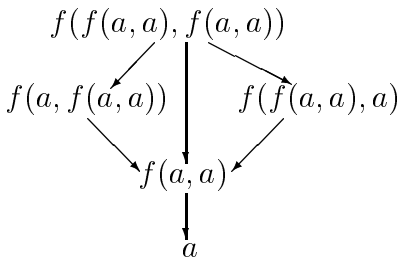
Durch die Regeln (1) und (2) des Termersetzungssystems wird offensichtlich die Fakultätsfunktion $f(n) = n!$ algorithmisch realisiert.

b) Nun beinhalte die vorausgesetzte Signatur gewisse, nicht näher spezifizierte Konstantensymbole der Sorte m und ein einziges Funktionssymbol $f : m, m \to m$. Als

Variablen setzen wir mindestens x von der Sorte m voraus. Schließlich gehen wir von nur einer Regel aus, die folgendes Aussehen hat:

$$(1) \quad f(x,x) \to x$$

Bei diesem Termersetzungssystem sind mehrere Ersetzungswege zu einer Eingabe möglich. Für den Term $f(f(a,a),f(a,a))$ bekommt man beispielsweise das nachfolgende Berechnungsdiagramm:

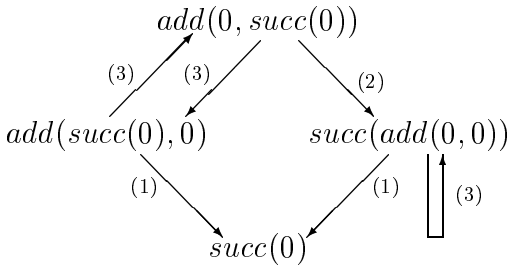


Etwa gilt für die obere mittlere Ersetzung $z = \varepsilon$ und $\sigma(x) = f(a,a)$. Für die obere rechte Ersetzung verwendet man die Stelle $z = 2$ und die Substitution $\sigma(x) = a$.

c) Im folgenden sei die Signatur die von Beispiel 2.3.2.a mit $nat, zero$ und $succ$, wobei wir, wie schon in a), die gewohnte 0 statt $zero$ schreiben, erweitert um ein Funktionsymbol $add : nat, nat \to nat$. Die Variablenmenge enthalte x und y, beide von der Sorte nat. Als Regeln des Termersetzungssystem habe man schließlich:

$$\begin{array}{ll} (1) & add(x,0) \to x \\ (2) & add(x,succ(y)) \to succ(add(x,y)) \\ (3) & add(x,y) \to add(y,x) \end{array}$$

Eine Beispielrechnung (mit Markierung der Übergänge durch die verwendeten Regeln) sieht dann wie folgt aus:



Die Kommutativitätsregel (3) läßt also nicht nur Verzweigungen in den Textersetzungsberechnungen zu, sondern **zerstört auch die Terminierung**. Entfernt man die Regel (3), so gilt offensichtlich

$$add(succ^n(0), succ^m(0)) \stackrel{*}{\Rightarrow} succ^{n+m}(0),$$

wobei $succ^k(0)$ für den Term $succ(succ(\ldots succ(0)\ldots))$ mit k Anwendungen von $succ$ steht. Das Termersetzungssystem mit den Regeln (1) und (2) realisiert somit die Addition. ■

Neben den bisherigen Systemen mit Regeln der Form $l \to r$ gibt es als Erweiterung noch **bedingte Termersetzungssysteme** mit Regeln der Form

$$l_1 = r_1 \wedge \ldots \wedge l_n = r_n \Rightarrow l \to r.$$

Hier darf in dem Term t zu einer Stelle $z \in Occ(t)$ und einer Substitution σ mit $l\sigma = t/z$ zum Term $t[r\sigma/z]$ nur übergegangen werden, wenn $l_i\sigma \overset{*}{\Leftrightarrow} r_i\sigma$ für alle $i, 1 \leq i \leq n$, gilt, wobei $t_1 \Leftrightarrow t_2$ genau dann gilt, falls $t_1 \Rightarrow t_2$ oder $t_2 \Rightarrow t_1$. Letzteres heißt, daß man von $l_i\sigma$ nach $r_i\sigma$ kommt, indem man die rechten Seiten $l \to r$ der Regeln des Systems in beiden Richtungen, d.h. sowohl von l nach r als auch von r nach l, anwendet.

2.4 Algorithmen und ihre Beschreibung

Die Verarbeitung von Information geschieht durch Verfahren, die man auch Algorithmen nennt. Im bisherigen Verlauf dieses Skriptums sind uns Algorithmen schon in der Gestalt von Textersetzung bzw. Termersetzung begegnet. Auch von der Schule her kennen wir schon einige Algorithmen, etwa die Lösung eines linearen Gleichungssystems mit Hilfe des Verfahrens von Gauß oder verschiedene geometrische Konstruktionen (Halbieren einer Strecke, Fällen eines Lots, Dreieckskonstruktionen usw.), die alle nach einem festen „Strickmuster“ ablaufen. Auch im täglichen Leben tauchen Tätigkeiten auf, die man durchaus als das Ausführen von Algorithmen verstehen kann. In diesem Abschnitt geben wir einige grundsätzliche Gedanken zum Algorithmusbegriff an, bleiben jedoch dabei auf einer informellen Ebene. Auch die Grenzen der Algorithmisierbarkeit werden wir anhand eines Beispiels aufzeigen.

2.4.1 Informelle Beschreibung von Algorithmen

Informell gesagt, ist ein **Algorithmus** ein allgemeines Verfahren zur Lösung einer Klasse verwandter Probleme, das den folgenden Eigenschaften genügt:

a) Es ist durch eine (endliche) **Zeichenreihe** beschrieben – auch bei der Behandlung von potentiell unendlichen Objekten.

b) Die **Einzelschritte** des Verfahrens, d.h. die als elementar vorausgesetzten Bestandteile der Aufschreibung, müssen effektiv ausführbar sein. Dabei ist dies relativ zu gewissen Grundlagen gemeint.

c) Aus der schriftlichen Darstellung muß sich die **Abfolge** der Einzelschritte von b) präzise ergeben. ■

Algorithmen kann man in vielerlei Arten beschreiben. Dabei ist **menschliche Lesbarkeit** für das Aufstellen und Verstehen von Algorithmen von kardinaler Bedeutung. Im folgenden geben wir einige Darstellungen an.

2.4.2 Einige Darstellungsformen für Algorithmen

a) **Umgangssprachliche Formulierung**: Beispiele hierzu sind Bastelanleitungen, Spielregeln und vor allem Kochrezepte.

b) **Graphische Darstellungen**: Hier sind die Zeichen der darstellenden Zeichenreihe graphische Zeichen, d.h. Bilder. Ein Beispiel ist etwa die Bedienungsanleitung zum Telefonieren, die man in jedem Telefonhäuschen findet.

c) **Formalisierte Darstellungen**: Diese Darstellungen von Algorithmen sind vor allem für die Mathematik sowie die theoretische und praktische Informatik typisch. So sind etwa Text- und Termersetzungssysteme formalisierte Darstellungen von Algorithmen. Weitere Beispiele sind die Algorithmen, welche in der Gestalt von Programmen einer speziellen Programmiersprache bekannt sind. Programme sind normalerweise Zeichenreihen; es gibt für sie aber auch formalisierte graphischen Darstellungen wie Flußdiagramme oder Struktogramme.

d) **Realisierung durch Geräte**: Beispiele hierzu sind die alten mechanischen **Kurbelrechner**, wie man sie im Deutschen Museum in München noch findet, moderne **Mikroprozessoren** oder **VLSI-Schaltungen**. Solche Realisierungen sind für die technische Informatik typisch. ■

Algorithmen arbeiten auf Daten, Objekten o.ä. **In diesem Skriptum interessieren wir uns nur für solche Verfahren, die eine Eingabe und eine Ausgabe besitzen**, wie beispielsweise die Textersetzungssysteme. Diese berechnen Zeichenreihen aus Zeichenreihen. Ein Algorithmus $\mathcal{A}$ mit Eingaben aus einer Menge M und Ausgaben aus einer Menge N beschreibt im allgemeinsten Fall eine Relation $R_{\mathcal{A}} \subseteq M \times N$, in der Regel jedoch eine (partielle) Funktion $f_{\mathcal{A}} : M \longrightarrow N$. Aus der Logik weiß man jedoch, daß man nicht jede Relation oder (partielle) Funktion durch einen Algorithmus beschreiben kann. Ein Beispiel für eine nicht berechenbare Relation / Funktion werden wir später noch angeben.

2.4.3 Einige Eigenschaften von Algorithmen

Wir betrachten zuerst das **Ablaufverhalten der Einzelschritte**:

a) Werden die Einzelschritte stets nacheinander ausgeführt, so spricht man von einem **sequentiellen** Algorithmus. Ist es hingegen möglich, daß manche Einzelschritte auch gleichzeitig ausgeführt werden, so ist der Algorithmus **parallel** oder **nebenläufig**.

b) Liegt bei der Abarbeitung der nächste Schritt genau fest, so ist der Algorithmus **deterministisch**; sonst nennt man ihn **nichtdeterministisch.** Diese beiden Eigenschaften kann man an der Aufschreibung ablesen.

Nun befassen wir uns mit Eigenschaften, die das **Gesamtablaufverhalten** betreffen:

c) Bei einem **terminierenden** Algorithmus endet jede Ausführung nach endlich vielen Schritten. Er beschreibt damit eine totale Relation.

d) Beschreibt ein Algorithmus eine partielle Funktion, d.h. führt jede Ausführung höchstens zu einem Resultat, so nennt man ihn **determiniert**; sonst heißt er **nichtdeterminiert**. Offensichtlich sind nichtdeterministische Algorithmen auch nichtdeterminiert, die Umkehrung gilt jedoch nicht.

Die letzte Gruppe von Eigenschaften betrifft nicht einen einzelnen Algorithmus, sondern den **Vergleich von Algorithmen**:

e) Löst ein Algorithmus $\mathcal{A}$ auch jedes Problem, das der Algorithmus $\mathcal{B}$ löst, so heißt $\mathcal{A}$ **universeller** als $\mathcal{B}$.

f) Löst hingegen $\mathcal{A}$ die gleichen Aufgaben wie $\mathcal{B}$, aber mit weniger Aufwand, so heißt $\mathcal{A}$ **effizienter** als $\mathcal{B}$. Dabei gibt es verschiedene Effizienzmaße, beispielsweise

- Laufzeiteffizienz (weniger Schritte),
- Speicherplatzeffizienz (weniger Speicherplatz),
- Parallelitätsgrad (weniger Prozessoren),

die man auch kombinieren kann. ■

Mit den beiden Programmiersprachen ML und Java werden wir im weiteren Verlauf dieses Skriptums nur sequentielle und deterministische Algorithmen (hier: Programme) formulieren. Typische nichtdeterministische Algorithmen sind Text- und Termersetzungssysteme. Ein Beispiel zu den Eigenschaften 2.4.3.f sind Algorithmen zur Lösung von linearen Gleichungsystemen. Diese löst man mit dem Gauß-Verfahren wesentlich effizienter als mit der Cramer'schen Regel. Das Studium der Effizienz von Algorithmen ist der Inhalt einer Vorlesung über Komplexitätstheorie, die Teil des Hauptstudiums ist.

Wir haben bisher nur eine informelle Beschreibung des Begriffs „Algorithmus“ angegeben, der damit noch keiner mathematischen Behandlung zugängig ist. Um eine solche zu ermöglichen, wurde in der mathematischen Grundlagenforschung in der ersten Hälfte des Jahrhunderts versucht, den Algorithmusbegriff mathematisch zu präzisieren. Die wichtigsten Ansätze sind nachfolgend genannt:

- Allgemein-rekursive Funktionen (Herbrand, Gödel, 1934).
- λ-definierbare Funktionen (Church, Kleene, 1936).
- Maschinenmodelle, z.B. Turingmaschine (Turing 1937).
- Markov-Algorithmen (Markov 1951).

Die ersten beiden Algorithmenansätze sind sehr bedeutend für funktionale Programmierung. Das im dritten Punkt genannte Maschinenmodell ist das Standardmodell in einer Vorlesung über Berechenbarkeit.

Zum Schluß dieses Abschnitts sind noch einige Worte zu den **Grenzen der Algorithmisierbarkeit** angebracht. Zunächst gab es unter den Mathematikern die Auffassung, daß alle mathematisch exakt formulierten Probleme auch mit mathematischen Hilfsmitteln lösbar seien. Nach der Formalisierung des Algorithmusbegriffs wurden

- alle oben erwähnten Ansätze (und noch einige mehr) als äquivalent gezeigt, d.h. gegenseitig simulierbar, so daß man diese jetzt als Arbeitshypothese nimmt[7]
- aber auch bald Probleme exakt formuliert, zu deren Lösung **bewiesenermaßen kein Algorithmus** existiert.

Die Suche nach solchen unlösbaren Problemen kann angesehen werden als Spezialfall der Frage nach absoluten Grenzen, so wie die Lichtgeschwindigkeit eine prinzipielle Grenze in der Physik darstellt. Im Fall des Algorithmusbegriffs haben diese Forschungen das Konzept des programmierbaren Rechners vorbereitet; hier war also die Theorie Schrittmacher für eine Umsetzung in die Praxis.

Wir geben nachfolgend ein berühmtes algorithmisch nicht lösbares Problem in der Form eines Entscheidungsproblems mit „ja“ und „nein“ als möglichen Antworten an. Da man jedem solchen Problem eine Funktion in die Wahrheitswerte $\mathbb{B}$ zuordnen kann, liefert das Probleme auch ein Beispiel für eine nicht berechenbare Funktion.

2.4.4 Beispiel (für ein unlösbares Problem)

Das folgende Problem, genannt **Hilberts 10. Problem**, ist unlösbar:

Eingabe:	Eine Polynomgleichung $p(x_1, \ldots, x_n) = 0$ mit $n > 0$ Unbekannten und ganzzahligen Koeffizienten, z.B. $3 * x_1 + 2 * x_2^2 + x_1^2 * x_2 * x_3 + 1 = 0$.
Frage:	Gibt es $b_1, \ldots, b_n \in \mathbb{Z}$ mit $p(b_1, \ldots, b_n) = 0$?

Dieses Problem wurde vom deutschen Mathematiker Hilbert im Jahr 1900 im Rahmen eines Vortrags als 10tes einer Liste von 23 Problemen vorgestellt, auf die sich, nach Hilberts Vorschlag, die zukünftige mathematische Forschung konzentrieren solle. Dies war mit einer der Anstöße zur Präzisierung des Algorithmusbegriffs. ■

Auch die Terminierung von Programmen ist allgemein algorithmisch nicht feststellbar. Dies heißt, daß es keinen Algorithmus gibt, der **für alle Programme** das Problem der Terminierung löst. In Spezialfällen ist Terminierung natürlich algorithmisch lösbar. Verfahren zum Beweis von Terminierung werden wir im Laufe der Skriptums noch kennenlernen.

[7] Man nennt dies die **Church'sche These**. Sie besagt in ihrer am weitesten verbreiteten Formulierung, daß der intuitive Algorithmusbegriff exakt beschrieben ist durch die Turingmaschine.

3 Grundkonzepte funktionaler Programmierung

Programmiersprachen zerfallen in zwei große Klassen: **imperative** und **deklarative** Sprachen. Der Unterschied zwischen den beiden Klassen liegt in den Mitteln, die sie zur Beschreibung von Algorithmen und Datenstrukturen zur Verfügung stellen. Zu den deklarativen Sprachen gehören insbesondere die **funktionalen** Sprachen, welche als Grundidee auf dem mathematischen, seiteneffektfreien Funktionsbegriff aufbauen und Funktionen wie gewöhnliche Datenobjekte, also etwa Zahlen oder Zeichenreihen, behandeln. Die Programmierung mittels funktionaler Sprachen nennt man **funktionale Programmierung**. Manchmal bezeichnet man funktionale Sprachen auch als **applikativ** und spricht dann von **applikativer Programmierung**[8]. Dieses Kapitel gibt mittels der Programmiersprache ML eine Einführung in die funktionale Programmierung.

3.1 Funktionen als Grundlage des Programmierens

In der Mathematik besteht eine Funktion $f : M \longrightarrow N$ aus drei Dingen, nämlich

- dem **Argumentbereich** M,
- dem **Resultatbereich** N,
- einer **Abbildungsvorschrift**, die jedem Element $x \in M$ genau ein Element $y \in N$ zuordnet und normalerweise durch einen Term beschrieben ist.

Von ihrer Natur her sind Funktionen also **statisch** und damit in der Regel nichtalgorithmisch. Sie erlauben jedoch auch, **Dynamik** (d.h. Berechnungen, Abläufe, Algorithmen) auszudrücken, indem man von der Originalform zu einer gleichwertigen rekursiven Form übergeht und dann jene zur dynamischen Berechnung der Ergebnisse aus den Argumenten mittels Termersetzung verwendet. Wir wollen dies im folgenden an zwei einfachen Beispielen aus der Arithmetik demonstrieren.

3.1.1 Beispiel (einer rekursiv beschriebenen Funktion)

Eine natürliche Zahl $n \in \mathbb{N}$ heißt **perfekt**, falls sie die Summe aller ihrer echten Teiler ist, also die Gleichung $\sum_{t|n,t<n} t = n$ gilt. Wir betrachten die Funktion

$$isperfect : \mathbb{N} \longrightarrow \mathbb{B} \qquad\qquad isperfect(n) = (\sum_{t|n,t<n} t = n),$$

[8] Von einer applikativen Sprache spricht man in der Regel, wenn Rechenvorschriften (diesen Begriff werden wir bald einführen) hauptsächlich mittels Funktionsapplikation, d.h. in der Form $f(x) = t$, definiert werden. Bei echt funktionalen Programmen hat man es hingegen bei der Definition von Rechenvorschriften mit Verknüpfungen von funktionalen Termen zu tun, d.h. mit Definitionen der Gestalt $f = t$. Ist beides möglich, so liegt, einem sehr verbreiteten Sprachgebrauch folgend, funktionale Programmierung vor. Die eben beschriebene Charakterisierung wird aber nicht überall in der Literatur so gehandhabt.

die spezifiziert, ob eine natürliche Zahl n perfekt ist, oder nicht. Wie kann man nun Perfektsein algorithmisch testen? Dazu definieren wir zuerst eine Verallgemeinerung

$$isp : \mathbb{N} \times \mathbb{N} \times \mathbb{N} \longrightarrow \mathbb{B} \qquad\qquad isp(n, x, y) = (\sum_{t|n, t<x} t = y)$$

der obigen Funktion *isperfect* und erhalten daraus durch eine Spezialisierung der letzten beiden Argumente sofort die Eigenschaft

$$isperfect(n) = isp(n, n, n). \tag{P_1}$$

Wie man auf eine solche Verallgemeinerung kommt, ist natürlich eine andere Frage, auf die wir aber jetzt noch nicht eingehen wollen. Aufgrund der Gleichung (P_1) haben wir das Problem, die Funktion *isperfect* zu algorithmisieren, darauf reduziert, einen Algorithmus für die Funktion *isp* anzugeben. Als ein erstes Resultat in diese Richtung betrachten wir die Gleichung $\sum_{t \in \emptyset} t = 0$ (das ist eine Festlegung, man vergleiche mit $\sum_{i=j}^{k} a_i = 0$ im Fall $k < j$) und bekommen mit ihrer Hilfe

$$isp(n, 0, y) = (0 = y). \tag{P_2}$$

Es verbleibt also noch der Fall $isp(n, x, y)$ mit $x \neq 0$. Hier ist $x - 1$ definiert und wir versuchen im folgenden deshalb, die Berechnung von *isp* mit zweitem Argument x auf die Berechnung von *isp* mit zweitem Argument $x - 1$ zurückzuführen. Dazu unterscheiden wir zwei Fälle:

a) Es sei $x-1$ ein Teiler von n. Dann gilt trivialerweise $\sum_{t|n,t<x} t = x-1+\sum_{t|n,t<x-1} t$. Nun können wir wie folgt vorgehen:

$$\begin{array}{rcll} isp(n, x, y) & = & (\sum_{t|n,t<x} t = y) & \text{Definition von } isp \\ & = & (x - 1 + \sum_{t|n,t<x-1} t = y) & \text{Gleichung von eben} \\ & = & (\sum_{t|n,t<x-1} t = y - (x - 1)) & \text{Eigenschaft der Subtraktion} \\ & = & isp(n, x - 1, y - x + 1) & \text{Definition von } isp \end{array}$$

Lassen wir die Zwischenrechnungen in dieser Gleichungskette weg und notieren nur das Endergebnis, so ist dies die rekursive Gleichung

$$isp(n, x, y) = isp(n, x - 1, y - x + 1). \tag{P_3}$$

b) Ist hingegen $x - 1$ kein Teiler von n, so gilt $\sum_{t|n,t<x} t = \sum_{t|n,t<x-1} t$. Dies bringt sofort die nachfolgende Gleichungskette:

$$\begin{array}{rcll} isp(n, x, y) & = & (\sum_{t|n,t<x} t = y) & \text{Definition von } isp \\ & = & (\sum_{t|n,t<x-1} t = y) & \text{Gleichung von eben} \\ & = & isp(n, x - 1, y) & \text{Definition von } isp \end{array}$$

Lassen wir wiederum die Zwischenrechnungen weg, so erhalten wir als Rekursion

$$isp(n, x, y) = isp(n, x - 1, y). \tag{P_4}$$

Mit Hilfe der eben bewiesenen vier Gleichungen (P_1) bis (P_4) kann man nun sehr einfach Zahlen $n \in \mathbb{N}$ auf Perfektheit testen; etwa erhält man $isperfect(6) = tt$ wegen

$$\begin{array}{rcll} isperfect(6) & = & isp(6,6,6) & \text{wegen } (P_1) \\ & = & isp(6,5,6) & \text{wegen } (P_4),\ \text{da } 5 \not| 6 \\ & = & isp(6,4,6) & \text{wegen } (P_4),\ \text{da } 4 \not| 6 \\ & = & isp(6,3,3) & \text{wegen } (P_3),\ \text{da } 3 \mid 6 \\ & = & isp(6,2,1) & \text{wegen } (P_3),\ \text{da } 2 \mid 6 \\ & = & isp(6,1,0) & \text{wegen } (P_3),\ \text{da } 1 \mid 6 \\ & = & isp(6,0,0) & \text{wegen } (P_4),\ \text{da } 0 \not| 6 \\ & = & (0 = 0) & \text{wegen } (P_2) \\ & = & tt. & \end{array}$$

Der Zusammenhang zwischen den Gleichungen (P_1) bis (P_4) und der Berechnung des Resultats tt des Aufrufs $isperfect(6)$ mit ihrer Hilfe und Termersetzungstechniken ist offensichtlich. ■

Wir geben noch ein zweites Beispiel an, das etwas anders gelagert ist als das letzte. Dies betrifft sowohl die Spezifikation der betrachteten Funktion als auch die Form der abgeleiteten Rekursion.

3.1.2 Beispiel (einer rekursiv beschriebenen Funktion)

Es sei $n \in \mathbb{N}$. Dann ist der **ganzzahlige Anteil** $isqrt(n)$ **der Quadratwurzel** von n gegeben durch die nachfolgende, implizit definierte Funktion:

$$isqrt : \mathbb{N} \longrightarrow \mathbb{N} \qquad\qquad isqrt(n)^2 \leq n < (isqrt(n)+1)^2$$

Wie kann man die Zahl $isqrt(n)$ aufbauend auf diese Spezifikation algorithmisch berechnen? Offensichtlich gilt $0^2 \leq 0 < (0+1)^2$ und dies zeigt sofort

$$isqrt(0) = 0. \qquad (Q_1)$$

Da für die Null damit das Resultat bekannt ist, sei nun $n \neq 0$. Wir zielen wiederum auf eine Rekursion ab und nehmen deshalb $x = isqrt(n-1)$ an, d.h. $x^2 \leq n-1 < (x+1)^2$. Geleitet durch die implizite Definition der Funktion $isqrt$ unterscheiden wir nun die beiden folgenden Fälle:

a) Es sei $(x+1)^2 \leq n$. Dann haben wir

$$\begin{array}{rll} (x+1)^2 & \leq n & \text{Voraussetzung} \\ & < n - 1 + 2 * x + 3 & \\ & < (x+1)^2 + 2 * x + 3 & \text{da } n-1 < (x+1)^2 \\ & = x^2 + 2 * x + 1 + 2 * x + 3 & \text{Binomische Formel} \\ & = (x+2)^2 & \text{Binomische Formel} \end{array}$$

und dies zeigt, nach der Definition von $isqrt$, die Gleichung $x + 1 = isqrt(n)$ bzw., nach der Definition von x, die Rekursion

$$isqrt(n) = isqrt(n-1) + 1. \qquad (Q_2)$$

b) Nun sei $n < (x+1)^2$. Hier gelten die Abschätzungen $x^2 \leq n - 1 < n < (x+1)^2$, welche, nach der Definition von $isqrt$, gleichwertig sind zu $x = isqrt(n)$ bzw., nach der Definition von x, zur Rekursion

$$isqrt(n) = isqrt(n-1). \qquad (Q_3)$$

Durch die Eigenschaften (Q_1), (Q_2) und (Q_3) ist wiederum eine Berechnung des Werts $isqrt(n)$ möglich, wobei jedoch, im Vergleich zu Beispiel 3.1.1, nun die Termersetzungsvorgehensweise nicht mehr gar so offensichtlich ist. Dies liegt daran, daß, im Gegensatz zum obigen Beispiel, wo die „inneren" Bedingungen $(x-1) \mid n$ bzw. $(x-1) \nmid n$ nicht von der durch die Rekursion beschriebenen Funktion isp abhängen, nun die inneren Bedingungen $(isqrt(n-1)+1)^2 \leq n$ und $n < (isqrt(n-1)+1)^2$ in Abhängigkeit von $isqrt$ definiert sind. ■

Nach diesen motivierenden Beispielen, die auch schon andeuten, wie ein funktionales Programm aussieht und wie man funktionale Programme entwickeln kann, wollen wir nun kurz und sehr allgemein auf die im weiteren Verlauf dieses Skriptums verwendete funktionale Programmiersprache eingehen.

3.1.3 Die funktionale Programmiersprache ML

Wie schon in der Einleitung erwähnt, werden wir ML als funktionale Programmiersprache verwenden. Um einen ersten Eindruck davon zu geben, wie ML-Programme aussehen, greifen wir die beiden letzten Beispiele noch einmal auf.

Das erste Beispiel 3.1.1, der Test, ob eine natürliche Zahl perfekt ist, formuliert sich in ML wie folgt als Deklaration von zwei sogenannten **Rechenvorschriften**:

```
fun isp (n : int, x : int, y : int) : bool =
  if x = 0 then y = 0
           else if divides(x-1,n) then isp(n,x-1,y-x+1)
                                  else isp(n,x-1,y);

fun isperfect (n : int) : bool = isp(n,n,n);
```

Man sieht, daß es sich eigentlich nur um eine Übertragung der Gleichungen (P_1) bis (P_4) samt der zugehörigen Bedingungen von Beispiel 3.1.1 in eine spezielle Notation handelt. Dabei sind `int` und `bool` die Sorten für $\mathbb{Z}$ bzw. $\mathbb{B}$ und der Teilbarkeitstest $(x-1) \mid n$ von n durch $x - 1$ wird durch die (nicht ausformulierte) Rechenvorschrift `divides` realisiert. Die Verwendung der Sorte `int` in diesen beiden Rechenvorschriften statt einer Sorte für natürliche Zahlen ist dadurch bedingt, daß es in ML keine (vordefinierte) Sorte für $\mathbb{N}$ gibt.

Betrachten wir das zweite Beispiel 3.1.2, die Berechnung des ganzahligen Anteils der Quadratwurzel, so erhalten wir durch eine Übertragung in die Programmiersprache ML das nachfolgende funktionale Programm:

```
fun isqrt (n : int) : int =
  if n = 0 then 0
           else let val x : int = isqrt(n-1)
                in  if (x+1) * (x+1) <= n then x+1
                                          else x
                end;
```

Neu im Vergleich zum obigen Programm ist hier die Verwendung einer sogenannten Objektdeklaration, durch die der Bezeichner `x` als Abkürzung für den Term `isqrt(n-1)` eingeführt wird.

ML entstand Ende der 70er Jahre des letzten Jahrhunderts an der Universität Edinburgh und war ursprünglich nur die Implementierungssprache für das Programmsystem LCF. Deshalb wurde auch der Name gewählt, denn ML steht für **meta language** (für LCF). Es wurde jedoch bald klar, daß ML auch als Programmiersprache von allgemeinem Interesse ist. Mittlerweile liegen auch Implementierungen von kommerziellen Anbietern vor mit komfortablen Entwicklungsumgebungen und Hilfen zur Fehlersuche („Debugger"). Speziell verwenden wir in dem Skriptum **Standard ML of New Jersey**, SML/NJ in der neuesten Version von 1997. Dieses System ist normalerweise interpretierend, erlaubt jedoch durch spezielle Kommandos auch eine Kompilation zur Effizienzsteigerung. ■

Wir haben die beiden obigen ML-Programme in Schreibmaschinenschrift gesetzt, ebenso Teile daraus im umgebenden Text. Diese Schreibweise werden wir auch im weiteren Verlauf des Skriptums beibehalten.

3.2 Elementare Datenstrukturen

In Abschnitt 2.3 hatten wir Signaturen $\Sigma = (S, K, F)$ eingeführt, darüber Σ-Terme definiert und dann erklärt, wie man dies alles über Σ-Algebren interpretiert. Dieses Prinzip ist auf die Programmiersprache ML übertragbar:

a) Es gibt eine Signatur zu ML, welche die vorimplementierten Sorten, wie `bool` und `int`, Konstantensymbole, wie `0` und `1`, und Funktionssymbole, wie `+` und `<=`, zur Verfügung stellt. Wir fassen im folgenden immer Sorten, Konstanten- und Funktionssymbole zu „passenden" Klassen zusammenen und nennen diese dann die **elementaren Datenstrukturen**. In diesem Zusammenhang werden die Konstanten- und Funktionssymbole auch oft **Konstanten** bzw. **Operationen** genannt.

b) Es gibt eine Algebra zur Signatur von ML, welche die Sorten, Konstanten und Operationen interpretiert, und zwar so, daß die durch die Namensgebung bzw. verwendeten Symbole beabsichtigte Zuordnung tatsächlich gegeben ist. Für diese Algebra

A gelten also etwa die Gleichungen $\texttt{bool}^A = \mathbb{B}$, $\texttt{int}^A = \mathbb{Z}$, $\texttt{true}^A = tt$, $\texttt{0}^A = 0$, $x \texttt{+}^A y = x + y$ und so fort.

c) Über der Signatur, d.h. den elementaren Datenstrukturen, werden Terme aufgebaut. Der ML-Termbegriff ist im Vergleich zur Definition 2.3.3 jedoch stark erweitert; es werden auch noch Fallunterscheidung `if ... then ... else ...`, Objektdeklaration `val ...`, Rekursion und einige weitere Möglichkeiten zugelassen. Insbesondere verkompliziert Rekursion die Semantik. Die einfache Festlegung des Werts in Definition 2.3.9 ist nur für Terme ohne Rekursion möglich.

Aufbauend auf die Terme ist es dann möglich, Rechenvorschriften zu deklarieren, welche die einfachsten Formen von Programmen in ML sind. Dazu kommen wir im nächsten Abschnitt. In diesem Abschnitt geben wir als Vorbereitung die elementaren Datenstrukturen von ML (ausschnittsweise) an. Man beachte, daß bei den nachfolgend aufgeführten Datenstrukturen zu jeder Sorte m ungleich `real` zusätzlich immer noch `=` : m,m `-> bool` als **Gleichheits-** bzw. `<>` : m,m `-> bool`, als **Ungleichheitsoperation** definiert sind.

3.2.1 Datenstruktur des einelementigen Bereichs

Sorte:	`unit`	Einelementige Menge
Konstanten:	`() : unit`	Einziges Element
Operationen:	Keine	

■

Der einelementige Bereich wird später unter anderem dazu benötigt, um aus den elementaren Datenstrukturen durch in ML fest vorgegebene Konstruktionsmechanismen neue, zusammengesetzte Datenstrukturen zu erzeugen und Rechenvorschriften ohne Argumente zu typisieren.

3.2.2 Datenstruktur der Wahrheitswerte

Sorte:	`bool`	Wahrheitswerte $\mathbb{B}$
Konstanten:	`true : bool`	Symbol für tt
	`false : bool`	Symbol für ff
Operationen:	`not : bool -> bool`	Negation
	`andalso : bool,bool -> bool`	Sequentielle Konjunktion
	`orelse : bool,bool -> bool`	Sequentielle Disjunktion

■

Die Schreibweisen für Negation, Konjunktion und Disjunktion sind analog zur Mathematik vorgeschrieben, also Präfixform bzw. funktionale Schreibweise im ersten Fall und Infixform in den restlichen zwei Fällen.

An dieser Stelle ist eine Bemerkung zu den Operationen der elementaren Datenstrukturen von ML angebracht. Normalerweise sind sie **strikt** definiert, d.h. eine Anwendung liefert keinen definierten Wert, wenn eines der Argumente undefiniert ist. Die einzigen

Ausnahmen sind `andalso` und `orelse`, die von links nach rechts, also **sequentiell ausgewertet** werden. Im Fall von `andalso` heißt dies: Hat das erste Argument den Wert *ff*, so ist der Wert der Anwendung ebenfalls *ff*, unabhängig vom Wert des zweiten Arguments. Bei `orelse` gilt folgendes: Ist der Wert des ersten Arguments *tt*, so ist auch der Wert der Anwendung *tt*, ebenfalls unabhängig vom zweiten Argument.

3.2.3 Datenstruktur der ganzen Zahlen

Sorte:	`int`	Ganze Zahlen $\mathbb{Z}$
Konstanten:	`0`, `1`, ...	Symbole für 0, 1 ...
Operationen:	$\sim$ `: int -> int`	Einstelliges Minus
	`+, -, * : int,int -> int`	Arithmetik
	`div : int,int -> int`	Ganzzahlige Division
	`mod : int,int -> int`	Rest bei ganzzahliger Division
	`Int.min, Int.max : int,int -> int`	Minimum bzw. Maximum
	`abs : int -> int`	Absolutbetrag
	`<, <=, >, >= : int,int -> bool`	Vergleichsoperationen ■

Zu den Schreibweisen ist folgendes zu bemerken: Das einstellige Minus wird in Präfixform notiert. Die arithmetischen Operationen und die Vergleichsoperationen werden wie üblich in Infixnotation geschrieben. Gleiches gilt für die Operationen `div` und `mod`. Die funktionale Schreibweise ist schließlich vorgeschrieben für `Int.min`, `Int.max` und `abs`. Mit Ausnahme von `div` und `mod` sind alle Operationen total. Eine Anwendung der eben genannten Operationen ist nur dann definiert, falls der Wert des zweiten Arguments nicht Null ist. Zur Semantik ist noch zu bemerken, daß, genaugenommen, die Sorte `int` natürlich nicht durch die ganzen Zahlen interpretiert wird, sondern nur durch ein Intervall davon, das die Null als „Mitte“ enthält. Die Größe dieses Intervalls ist maschinenabhängig. Auf solche technische Einzelheiten wollen wir in diesem Skriptum nicht eingehen, obwohl diese in vielen Bereichen durchaus wichtig sind.

3.2.4 Datenstruktur der reellen Zahlen

Sorte:	`real`	Reelle Zahlen $\mathbb{R}$
Konstanten:	Alle Gleitkommadarstellungen	Beispielsweise `1.0`, `10.83E3`
Operationen:	$\sim$ `: real -> real`	Einstelliges Minus
	`+, -, *, / : real,real -> real`	Arithmetik
	`abs : real -> real`	Absolutbetrag
	`Math.sqrt : real -> real`	Quadratwurzel
	`Math.sin, Math.cos : real -> real`	Sinus bzw. Cosinus
	`Math.exp, Math.ln : real -> real`	e-Funktion bzw. natürl. Log.
	`real : int -> real`	Injektive Einbettung
	`floor : real -> int`	Abhackung
	`<, <=, >, >= : real,real -> bool`	Vergleichsoperationen ■

Auch bei dieser Datenstruktur legen wir eine idealisierende Interpretation zugrunde, obwohl tatsächlich nur die sogenannten Maschinenzahlen mit ihren Operationen vorliegen. Im eben aufgeführten Beispiel `10.83E3` steht der Buchstabe `E` für die Exponentiation. Bei Verwendung der üblichen mathematischen Notation schreibt sich die angegebene ML-Notation als $10.83 * 10^3$. Diese spezielle Zahldarstellung nennt man auch **Gleitkommadarstellung** mit **Mantisse** 10.83 und **Exponent** 3. Genaueres zu den Darstellungen von reellen Zahlen und das Rechnen mit ihnen erfährt man in einer Vorlesung über numerische Mathematik.

Mit Ausnahme des einstelligen Minus, das, wie im Falle der ganzen Zahlen, prefix-geschrieben wird, werden alle einstelligen Operationen in funktionaler Schreibweise notiert. Die arithmetischen Operationen und die Vergleichsoperationen werden, wie üblich, in Infixnotation geschrieben. Wie bei den ganzen Zahlen, so entspricht auch die Interpretation der Operationen der allgemeinen mathematischen Vorstellung, d.h. für die Operationen `/`, `Math.sqrt` und `Math.ln` gelten die üblichen Undefiniertheiten. Zu den beiden Operationen `real` und `floor` ist noch eine Bemerkung angebracht, die wir, wegen ihrer allgemeinen Bedeutung für Programmiersprachen, als eigenen Punkt notieren.

3.2.5 Bemerkung (zur Sortenanpassung)

In vielen Programmiersprachen sind zwischen Sorten sogenannte **Teilsortenrelationen** festgelegt, die Teilmengenbeziehungen auf der semantischen Ebene entsprechen. Sind m und n Sorten und ist n eine Teilsorte von m, so kann dann eine Operation $f : m \to p$ auch auf einen Term t der Sorte n angewendet werden. Für die Programmiersprache ML gilt jedoch der Grundsatz: **Es gibt keine Teilsortenrelationen**. Alle Sortenanpassungen müssen somit in ML **explizit** vorgenommen werden, so von `int` nach `real` durch die Operation `real` der Datenstruktur der reellen Zahlen bzw. durch spezielle Schreibweisen wie `1.0`, `2.0` usw. Die Umwandlung einer reellen Zahl in eine ganze Zahl geschieht durch die Operation `floor`, durch die der Nachkommateil abgehackt wird. Die Anwendung dieser Operation entspricht also keiner Rundung!

Andere Programmiersprachen, beispielsweise Java, welches wir später noch besprechen werden, besitzen Teilsortenrelationen, und damit geschieht, bei einem Term $f(t)$ mit einem Funktionssymbol f und einem Term t wie eben vorausgesetzt, bei der Anwendung $f(t)$ eine **implizite Sortenanpassung**. ■

Wir kommen nun zur letzten elementaren Datenstruktur von ML, die wir vorstellen wollen, den Zeichenreihen. Im folgenden sei Z der Zeichenvorrat von ML, bestehend aus den kleinen und großen lateinischen Buchstaben, den Ziffern und einigen Sonderzeichen wie & , _, % und $. ($Z$ entspricht genau dem sogenannten ASCII-Zeichenvorrat.)

3.2.6 Datenstruktur der Zeichenreihen

Sorten:	`char`	Zeichenvorrat Z
	`string`	Zeichenreihen Z^*

Konstanten:	Zeichen der Form `#"`a`"`	z.B. `#"R"`
	Zeichenreihen mit `"` begrenzt	z.B. `"Rudolf"`
Operationen:	`^ : string,string -> string`	Konkatenation
	`size : string -> int`	Länge
	`str : char -> string`	Zeichen wird Zeichenreihe
	`String.sub : string,int -> char`	n-tes Zeichen
	`substring : string,int,int -> string`	Teilzeichenreihe
	`<, <=, >, >= : string,string -> bool`	Lexikograph. Vergleiche ■

Für die Konkatenation und die vier lexikographischen Vergleichsoperationen ist die Infixschreibweise vorgesehen. Die zwei einstelligen Operationen `size` und `str` werden, wie auch die zweistellige Operation `String.sub` und die dreistellige Operation `substring`, in der funktionalen Schreibweise notiert. Ein Aufruf von `String.sub` ist nur dann definiert, wenn der Wert n des zweiten Arguments zwischen Null und der Länge des ersten Arguments minus Eins liegt. In diesem Fall wird das Zeichen nach der Position n geliefert. Durch die Operation `substring` ist es möglich, Teilzeichenreihen zu erhalten. Ist s eine Zeichenreihe und sind $i, j \in \mathbb{N}$ mit $i+j \leq |s|$, so liefert `substring` mit diesen Argumenten die Teilzeichenreihe $s_{i+1} \ldots s_{i+j}$ von s. Andernfalls ist das Resultat undefiniert.

Mit Hilfe der Operation `substring` kann man durch `substring(s,1,size(s)-1)` in sehr einfacher Weise die Restbildung auf Zeichenreihen realisieren. Auch das erste Element einer Zeichenreihe kann durch sie als Zeichenreihe der Länge Eins sehr einfach berechnet werden, nämlich mittels `substring(s,0,1)`. Will man hingegen das erste Element einer Zeichenreihe wirklich als Zeichen, d.h. als ein Objekt der Sorte `char` geliefert bekommen, so ist dies durch den Aufruf `String.sub(s,0)` möglich.

Damit wollen wir die Vorstellung der elementaren Datenstrukturen von ML abschließen. Wie man aus ihnen mittels genau festgelegter Konstruktionsprinzipen komplexe Datenstrukturen aufbaut, werden wir später kennenlernen.

3.3 Rekursive Rechenvorschriften

Das zentrale Konzept einer funktionalen Programmiersprache ist die rekursive Rechenvorschrift, die eine **Berechnung von einer einzelnen Eingabe auf eine Klasse von Eingaben abstrahiert** und eine mathematische Funktion beschreibt. Oft werden Rechenvorschriften auch Funktionen, Funktionsprozeduren oder Methoden genannt, genauer ML-Funktionen, Pascal-Funktionsprozeduren, Java-Methoden usw. Wir unterscheiden in der Sprechweise jedoch präziser und bezeichnen als

- **(partielle) Funktionen** diejenigen mathematischen Gebilde $f : M \longrightarrow N$, die in Abschnitt 1.1 als spezielle Relationen eingeführt wurden,
- und als **ML-Rechenvorschriften** gewisse Teile von ML-Programmen, die einer speziellen Syntax genügen und deren Semantik (Wert, Interpretation) jeweils eine (partielle) Funktion in dem eben beschriebenen mathematischen Sinne ist.

So ist etwa die Semantik der ML-Rechenvorschrift `isperfect` von Beispiel 3.1.3 zum Testen perfekter Zahlen die partielle Funktion

$$isperfect_{\mathbb{Z}} : \mathbb{Z} \longrightarrow \mathbb{B} \qquad isperfect_{\mathbb{Z}}(n) = \begin{cases} \sum_{t|n,t<n} t = n & : \quad n \geq 0 \\ \text{undef} & : \quad n < 0, \end{cases}$$

denn die termmäßige Abarbeitung dieser Rechenvorschrift terminiert offensichtlich nicht für negative Eingaben. Semantik und Terminierung von ML-Rechenvorschriften werden im Laufe dieses Skriptums noch genauer betrachtet. Wir beginnen im folgenden mit der Syntax einer einzelnen Rechenvorschrift.

3.3.1 Einzelne Rechenvorschriften in ML

Eine **ML-Rechenvorschrift** besteht im wesentlichen aus

(1) ihrem **Namen**, einem Bezeichner, der frei wählbar ist,

(2) ihrer **Funktionalität**, das sind die frei wählbaren Namen/Bezeichner der (formalen) Parameter zusammen mit ihren Sorten sowie die Sorte des zu berechnenden Resultats,

(3) ihrem **Rumpf**, das ist ein Term, der besagt, wie das Ergebnis in Abhängigkeit von den Parametern berechnet wird.

Syntaktisch **deklariert** man eine Rechenvorschrift mit Namen F, Parametern x_i jeweils von der (Argument-)Sorte m_i $(1 \leq i \leq k)$, Resultatsorte n und Rumpf t wie folgt:

$$\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{,...,}x_k\ \texttt{:}\ m_k\texttt{)}\ \texttt{:}\ n\ \texttt{=}\ t\texttt{;} \qquad (R)$$

Dabei ist das **Semikolon als Abschlußzeichen Teil der Deklaration**. Auch der Fall $k = 0$ ist erlaubt und führt zur Form `fun` F `()` $:$ n `=` t`;` von (R) mit – per Definition – einziger Argumentsorte `unit`.

In einer Deklaration wie (R) wird als sogenannte Kontextbedingung noch gefordert, daß die Parameter alle paarweise verschieden und ungleich F sind und diese Namen auch nicht mit den Bezeichnern aus der ML-Signatur bzw. schon andersweitig eingeführten Bezeichnern kollidieren. Beim Aufbau des Rumpfes t einer Rechenvorschrift F der in (R) angegebenen Gestalt sind (beim derzeitigen Stand des Skriptums) erlaubt:

a) Die **Konstanten** der elementaren Datenstrukturen des letzten Abschnitts 3.2.

b) Die **Parameter** x_i, $1 \leq i \leq k$, der Funktionalität.

c) Anwendungen (Sprechweise ist auch: Aufrufe) von **Operationen** der elementaren Datenstrukturen und Aufrufe von (schon deklarierten) **anderen Rechenvorschriften**. Dabei müssen die Argumente ebenfalls dem Aufbauprinzip genügen, das gerade erklärt wird, und zusätzlich noch Typisierung und Stelligkeit passen.

d) **Fallunterscheidungen** in der Form von `if` b `then` e_1 `else` e_2. Dabei müssen b, die **Bedingung**, und e_1 und e_2, der **then-** bzw. **else-Zweig**, ebenfalls dem Aufbauprinzip genügen, das gerade erklärt wird. Weiterhin muß b von der Sorte `bool` sein und die Sorten von e_1 und e_2 müssen übereinstimmen.

e) **Rekursive Aufrufe**, d.h. Anwendungen der Rechenvorschrift F. Dabei müssen, wie in c), die Argumente ebenfalls dem Aufbauprinzip genügen, das gerade erklärt wird, und zusätzlich noch Typisierung und Stelligkeit passen. ■

Wir haben uns aus Gründen der Lesbarkeit für eine umgangssprachliche Beschreibung der Syntax von Rechenvorschriften entschieden und werden diese Beschreibungsart auch bei den später noch folgenden Erweiterungen von ML bzw. bei der syntaktischen Beschreibung von Java beibehalten. Es ist dann jeweils eine einfache Übungsaufgabe, die umgangssprachliche Syntax zu formalisieren, etwa mittels einer induktiven Definition. Auf die Semantik gehen wir später genauer ein. Was wir aber jetzt schon festhalten wollen ist, daß sich eine Fallunterscheidung zum then-Zweig auswertet, wenn der Wert der Bedingung *tt* ist, und zum else-Zweig, wenn der Wert der Bedingung *ff* ist.

Bei der Definition von Rechenvorschriften in der Festlegung 3.3.1 handelt es sich nicht um die gesamte Sprache ML. Später werden wir beispielsweise in der Funktionalität statt Sorten auch Bezeichner für komplexere Datenstrukturen u. ä. und beim Aufbau des Rumpfes noch ausgereiftere Termkonstruktionen zulassen.

Zur Verdeutlichung der bisher zugelassenen Form von Rechenvorschriften geben wir nachfolgend einige Beispiele an. Dabei zeigen wir auch, wie man rekursive Rechenvorschriften aus gegebenen Spezifikationen durch mathematische Überlegungen „herrechnen“ kann. Auf die formale Semantik von Rechenvorschriften gehen wir jetzt noch nicht ein, sondern unterstellen ein intuitives termmäßiges Verständnis der Abarbeitung von Rekursion, wie es durch die beiden einführenden Beispiele 3.1.1 und 3.1.2 und ihre Übertragung nach ML motiviert wurde.

3.3.2 Beispiele (für ML-Rechenvorschriften)

a) Das **Volumen eines Kegelstumpfs** mit Höhe h, kleinem Radius r und großem Radius R berechnet sich nach der Formel

$$V = \frac{\pi * h}{3} * (r^2 + r * R + R^2).$$

Aus dieser Formel bekommt man sofort eine nichtrekursive ML-Rechenvorschrift `V` zur Berechnung des Volumens, die wie folgt aussieht:

```
fun V (r : real, R : real, h : real) : real =
  1.0472 * h * (r * r + r * R + R * R);
```

In dieser Rechenvorschrift stellt die im Rumpf verwendete Zahl `1.0472` eine Näherung des Bruchs $\frac{\pi}{3}$ auf vier Stellen nach dem Dezimalpunkt dar.

b) Die **Fakultät** $n!$ einer natürlichen Zahl n ist definiert als $n! := \prod_{i=1}^{n} i$. Die Zahl 1 ist neutral bezüglich der Multiplikation und damit per Definition gleich jedem Produkt mit leerer Indexmenge. Im Falle der Fakultät liefert dies die folgende Gleichung:

$$0! = \prod_{i=1}^{0} i = 1 \qquad (F_1)$$

Nun sei $n \neq 0$. Dann gilt, nach der induktiven Definition des Produktsymbols und der Definition der Fakultät, die nachfolgende rekursive Beziehung:

$$n! = \prod_{i=1}^{n} i = n * \prod_{i=1}^{n-1} i = n * (n-1)! \qquad (F_2)$$

Aus den beiden Gleichungen (F_1) und (F_2) erhalten wir durch einen Übergang zur Programmiersprache ML und unter Verwendung der Sorte `int` der ganzen Zahlen sofort die folgende ML-Rechenvorschrift:

```
fun fac (n : int) : int =
  if n = 0 then 1
           else n * fac(n-1);
```

Offensichtlich berechnet, eine naive Auswertung durch Termersetzung analog zu Beispiel 3.1.1 vorausgesetzt, diese Rechenvorschrift für ein nichtnegatives Argument n die Fakultät von n und terminiert für alle negativen Argumente nicht. D.h. die Semantik von `fac` ist die folgende partielle Funktion:

$$fac_{\mathbb{Z}} : \mathbb{Z} \longrightarrow \mathbb{Z} \qquad fac_{\mathbb{Z}}(n) = \begin{cases} n! & : \ n \geq 0 \\ \text{undef} & : \ \text{sonst} \end{cases}$$

c) Es sei A ein Zeichenvorrat. Wir betrachten die Funktion $reverse : A^* \longrightarrow A^*$ zum **Revertieren von Zeichenreihen**. Es gibt mehrere Möglichkeiten, diese Funktion zu spezifizieren. Betrachten wir eine Zeichenreihe als ein n-Tupel, wie in Definition 2.1.1 eingeführt, so können wir festlegen:

$$reverse(s_1 \ldots s_n) = s_n \ldots s_1 \qquad (R_1)$$

Diese Definition verwendet die berühmten drei Punkte. Sie können aber auch vermieden werden, indem man beispielsweise die implizite Spezifikation

$$\forall\, i \in \mathbb{N} : 1 \leq i \leq |s| \rightarrow s_i = reverse(s)_{|s|-i+1} \qquad (R_2)$$

wählt, oder aber die Funktion *reverse* induktiv durch

$$reverse(\varepsilon) = \varepsilon \qquad reverse(a \,\&\, w) = reverse(w) \,\&\, a \qquad (R_3)$$

festlegt. Im Prinzip entspricht die Festlegung (R_3) der Herrechnung einer rekursiven Formulierung aus einer der obigen Definitionen (R_1) oder (R_2).

Ist man an einer Realisierung der Funktion *reverse* in ML interessiert, so ist hierfür die Spezifikation (R_3) der geeignetste Ausgangspunkt. Aufbauend auf die Bemerkungen am Ende von Abschnitt 3.2 und (R_3) erhalten wir sofort das folgende ML-Programm für *reverse*:

```
fun top (s : string) : string =
  substring(s,0,1);

fun rest (s : string) : string =
  substring(s,1,size(s)-1);

fun reverse (s : string) : string =
  if size(s) <= 1 then s
                  else reverse(rest(s)) ^ top(s);
```

Die letzte Rechenvorschrift `reverse` bildet eine Zeichenreihe s mit $|s| \leq 1$ auf sich selbst ab. Andernfalls wird durch `rest` und den rekursiven Aufruf von `reverse` zuerst der Rest der Eingabe revertiert und dann, mittels `top` und der Konkatenationsoperation, das erste Zeichen der Eingabe rechts angefügt. ■

Unsere bisherigen Festlegungen erlauben, ML-Programme zu schreiben, die aus Deklarationen von Rechenvorschriften bestehen. In so einem Programm

$$\texttt{fun } F_1 \ldots\texttt{;}\ \texttt{fun } F_2 \ldots\texttt{;}\ \ldots\ \texttt{fun } F_k \ldots\texttt{;} \qquad (*)$$

kann man natürlich F_1 in F_2 bis F_k, F_2 in F_3 bis F_k usw. verwenden, jedoch nicht etwa F_2 in F_1. Das Semikolon legt eine Reihenfolge fest. Erst nach ihm ist der entsprechende Name bekannt und kann von anderen Rechenvorschriften verwendet werden. Wird ein Bezeichner F_i mehrmals verwendet, was nicht sinnvoll aber erlaubt ist, so wird durch eine Deklaration jeweils die zuletzt gültige Version von F_i überschrieben.

Bei ($*$) handelt es sich um die einfachste Form eines ML-Programms. Was noch bleibt, ist die Eingabe der Argumente. Befindet man sich im interaktiven Modus des SML97-Systems, so gibt man einfach den auszuwertenden Term ein, etwa:

```
isperfect(6);
```

Auch hier ist das Semikolon als Abschlußzeichen der Benutzereingabe notwendig. Daraufhin antwortet das System mit

```
val it = true : bool
```

und man kann dann weitermachen. Vor der Argumenteingabe müssen natürlich die Deklarationen von ($*$) zur Verfügung gestellt werden. Dies kann beispielsweise dadurch geschehen, daß man sie eintippt. Bei größeren Programmen, die man auch wiederverwenden will, empfiehlt es sich hingegen, die Deklarationen in eine Datei, etwa `isperfect.ml`, zu schreiben und diese mittels des Systemkommandos `use "isperfect.ml";` zu laden.

3.3.3 Systeme von Rechenvorschriften

Es gibt Situationen, in denen Rekursion, wie man sagt, **verschränkt** wird, wie etwa im Falle der Tests auf Gerade- und Ungeradesein. Die Funktionen zu diesen Tests, genannt

$$iseven : \mathbb{N} \longrightarrow \mathbb{B} \qquad\qquad isodd : \mathbb{N} \longrightarrow \mathbb{B},$$

erfüllen die nachfolgenden Rekursionen mit jeweils wechselseitigen Aufrufen:

$$iseven(n) = \begin{cases} tt & : \quad n = 0 \\ isodd(n-1) & : \quad n \neq 0 \end{cases}$$

$$isodd(n) \;\; = \begin{cases} ff & : \quad n = 0 \\ iseven(n-1) & : \quad n \neq 0 \end{cases}$$

Eine direkte Überführung dieser rekursiven Gleichungen in die Programmiersprache ML ist mit den bisher zugelassenen Sprachmitteln nicht möglich. Hat man sich beispielsweise dazu entschieden, mit der Deklaration

```
fun iseven (n : int) : bool =
  if n = 0 then true
           else isodd(n-1);
```

zu beginnen, so kommt in ihrem Rumpf im else-Zweig die Bezeichnung `isodd` vor, die aber weder eine Operation aus den elementaren Datenstrukturen noch der Name einer schon deklarierte Rechenvorschrift ist. Das SML97-System meldet also einen Fehler.

Grundsätzlich ist die Zusammenfassung von mehreren Rechenvorschriften mit gegenseitiger Verschränkung der Rekursion in ML zu sogenannten Systemen möglich, erfordert jedoch eine andere Syntax als bisher. Rechenvorschriften werden zu einem **System mit erlaubter verschränkter Rekursion** zusammengefaßt, indem man in der Syntax $(*)$ der Deklarationsfolgen jeweils Zeichenreihen der Gestalt „ ...; `fun` ...“ durch „... `and` ...“ ersetzt. Also haben wir insbesondere

```
fun iseven (n : int) : bool =
  if n = 0 then true
           else isodd(n-1)

and isodd (n : int) : bool =
  if n = 0 then false
           else iseven(n-1);
```

für das gewählte Beispiel der Tests auf Gerade- und Ungeradesein zu schreiben. ■

Nach dieser syntaktischen Erweiterung besteht also beim jetzigen Stand des Skriptums ein ML-Programm aus einer Folge von Deklarationen, jeweils mit einem Semikolon als Anschlußzeichen, wobei entweder einzelne Rechenvorschriften der obigen Gestalt (R) oder Systeme von Rechenvorschriften mit der eben beschriebenen Syntax deklariert werden. Auch hier wird die Verwendungsreihenfolge durch die Aufschreibung festgelegt.

3.3.4 Taxonomie der Rekursivität

Charakteristisch für die Situation bei (Systemen von) rekursiven Rechenvorschriften sind zwei Strukturen:

a) **Mikroskopische Struktur**: Diese betrifft eine einzelne Rechenvorschrift und die syntaktische Gestalt der rekursiven Aufrufe. Die folgenden Sprechweisen sind üblich: Eine Rechenvorschrift F ist

- **rekursiv**, falls sie in ihrem Rumpf aufgerufen wird,
- **linear-rekursiv**, falls sie rekursiv ist, aber in jedem Zweig einer Fallunterscheidung höchstens ein Aufruf erfolgt,
- **repetitiv**, falls sie linear-rekursiv ist und zusätzlich kein Aufruf von F in einem Argument einer Operation der elementaren Datenstrukturen oder einer anderen Rechenvorschrift vorkommt,
- **geschachtelt-rekursiv**, falls ein Teilterm der Gestalt $F(\ldots F(\ldots)\ldots)$ in ihrem Rumpf existiert,
- **kaskardenartig-rekursiv**, falls ein Teilterm $h(\ldots, F(\ldots), \ldots, F(\ldots)\ldots)$ in ihrem Rumpf vorkommt.

b) **Makroskopische Struktur**: Diese betrifft mehrere Rechenvorschriften und ihre gegenseitigen Aufrufe. Hier hat man die folgenden Bezeichnungen: Eine Rechenvorschrift F ist

- **direkt-rekursiv**, falls es im Rumpf von F einen Aufruf von F gibt,
- **indirekt-rekursiv**, falls sie Teil eines Systems im oben beschriebenen Sinne ist, kein eigener Aufruf im Rumpf vorkommt, sie aber mittelbar durch eine andere Rechenvorschrift des Systems wieder aufgerufen wird. ■

Im Hinblick auf eine maschinelle Abarbeitung von Rekursion ist die repetitive Rekursion die günstigste Form, da sie die effizienteste Übersetzung in Maschinenprogramme mit Sprungbefehlen erlaubt. Solche Rekursionen kann man oft auch durch die Schleifenkonstrukte von imperativen Programmiersprachen beschreiben, wie wir später noch sehen werden. Geschachtelte und kaskadenartige Rekursionen sind oft schrecklich ineffizient, müssen es aber nicht sein. Nachfolgend geben wir einige Beispiele für die eben erklärten Begriffe an. Wir verbinden dies mit etwas Programmiermethodik und zeigen, wie man Rechenvorschriften in günstigere Rekursionsformen überführen kann.

3.3.5 Beispiele (für Struktur von Rekursion)

a) Ein Beispiel für eine geschachtelt-rekursive Rechenvorschrift ist:

```
fun S (n : int) : int =
  if n = 0 then 1
           else S(S(n-1)-1) + 1;
```

Sie bestimmt, natürlich auf sehr umständliche Weise, den Nachfolger einer natürlichen Zahl. Es ist eine einfache Übungsaufgabe, dies durch Induktion zu zeigen. Der **Induktionsbeginn** verwendet den then-Zweig von `S`:

```
S(0) = if 0 = 0 then 1
                else S(S(0-1)-1) + 1
     = 1
```

Im **Induktionsschluß** `n` $\neq$ `0` wird der else-Zweig von `S` verwendet:

```
S(n) = if n = 0 then 1
                else S(S(n-1)-1) + 1
     = S(S(n-1)-1) + 1
     = S(n-1) + 1                          Induktionshypothese
     = n + 1                               Induktionshypothese
```

b) Die oben angegebenen Rechenvorschriften `iseven` und `isodd` sind, im System betrachtet, indirekt-rekursiv. Man gelangt aber leicht in beiden Fällen zu direkt-rekursiven Varianten. Wir führen die entsprechenden Programmumformungen nur für die Rechenvorschrift `iseven` durch:

```
iseven(n) = if n = 0 then true
                     else isodd(n-1)
          = if n = 0 then true
                     else if n-1 = 0 then false
                                     else iseven(n-2)
          = if n = 0 then true
                     else if n = 1 then false
                                   else iseven(n-2)
```

Das erste und das letzte Glied dieser Kette beschreiben eine Rekursion, die, vervollständigt zu einer direkt-rekursiven Rechenvorschrift, von folgender Gestalt ist:

```
fun iseven (n : int) : bool =
  if n = 0 then true
           else if n = 1 then false
                         else iseven(n-2);
```

c) Wir betrachten nun die Rechenvorschrift `fac` von Beispiel 3.3.2.b. Sie ist linear-rekursiv, aber nicht repetitiv, da im Rumpf der rekursive Aufruf `fac(n - 1)` als Argument der Multiplikation auftritt.

Unser Ziel ist es, sie mit Hilfe einer repetitiv-rekursiven Rechenvorschrift auszudrücken. Dazu starten wir mit

```
fun facr (n : int, res : int) : int =
  fac(n) * res;
```

und erhalten durch einfache (und intuitiv auch gültige) Gesetze die folgende Gleichungskette zwischen ML-Termen:

```
facr(n,res) = (if n = 0 then 1
                        else n * fac(n-1)) * res
            =  if n = 0 then 1 * res
                        else n * fac(n-1) * res
            =  if n = 0 then res
                        else fac(n-1) * n * res
            =  if n = 0 then res
                        else facr(n-1, n*res)
```

Offensichtlich gilt auch `fac(n)` = `facr(n,1)`, und damit können wir von der Rechenvorschrift `fac` von Beispiel 3.3.2.b zu den zwei Deklarationen

```
fun facr (n : int, res : int) : int =
  if n = 0 then res
           else facr(n-1,n*res);

fun fac (n : int) : int = facr(n,1);
```

übergehen, wobei die nun die eigentliche Berechnung durchführende Rechenvorschrift `facr` repetitiv ist. ■

Man sagt, daß die Rechenvorschrift `fac` durch die Vorgehensweise des letzten Beispiels (mittels einer Einbettung) **entrekursiviert** wurde. Diese Sprechweise hat sich eingebürgert, weil bei einer Transformation des speziellen repetitiven Rekursionstyps in Schleifen und Sprünge die rekursiven Aufrufe nicht mehr explizit auftreten.

In den Beispielen 3.3.5 zeigte sich auch, wie einfach und doch mathematisch präzise man funktionale Programme manipulieren kann. Dies ist auch ein großer Vorteil von funktionaler Programmierung: **Im Vergleich zu imperativer Programmierung werden Manipulationen und Beweise** wesentlich vereinfacht. Dies liegt im wesentlichen daran, daß man keine Seiteneffekte hat, **also ein Bezeichner (Name) für genau einen Wert steht und immer nur für diesen**. Diese wichtige Eigenschaft funktionaler Programmiersprachen wird **referenzielle Transparenz** genannt.

3.4 Betrachtungen zu Semantik und Programmentwicklung

Wegen der Rekursion muß man bei der Auswertung von funktionalen Sprachen und auch bei Programmanipulationen trotz aller Einfachheit doch etwas aufpassen. Was dahinter steckt, wird unter anderem in diesem letzten Abschnitt dieses Kapitels besprochen. Das Hauptziel dieses Abschnitts ist zu zeigen, wie man die Semantik von funktionalen Sprachen formal festlegen kann. Dazu gibt es mehrere Möglichkeiten. Wir besprechen hier zwei Arten. Dies geschieht aber nicht in der mathematischen Präzision, wie es heutzutage

möglich ist. Eine exakte Semantikbeschreibung ist Teil einer entsprechenden Vorlesung nach dem Vordiplom. Für das weitere betrachten wir der Einfachheit halber nur eine einzelne Rechenvorschrift der Form (R) von Abschnitt 3.3 Auch nehmen wir an, daß in ihr keine Aufrufe anderer Rechenvorschriften vorkommen.

Zum besseren Verständnis der im Anschluß zu behandelnden ersten Art von Semantik, der Termersetzungssemantik, diskutieren wir zunächst zwei elementare Umformungen von funktionalen Programmen.

3.4.1 Erklärung: Expandieren und Komprimieren

a) **Expandieren** eines Aufrufs einer Rechenvorschrift ist seine Ersetzung durch den Rumpf, wobei gleichzeitig die Parameter durch die Argumentterme des Aufrufs textuell substituiert werden.

b) **Komprimieren** eines Terms ist seine Ersetzung durch den Aufruf einer Rechenvorschrift, deren Rumpf nach der textuellen Substitution der Parameter durch die Aufrufargumente mit ihm übereinstimmt. ■

In der englischsprachigen Literatur spricht man statt von Expandieren und Komprimieren auch von **unfold** und **fold**. Bei den Rechnungen in Beispiel 3.3.5 haben wir Expandieren und Komprimieren fortlaufend verwendet. Nach diesen Erklärungen sind wir nun in der Lage, die erste Semantik funktionaler Sprachen anzugeben.

3.4.2 Termersetzungssemantik von Rechenvorschriften

Wir setzen voraus, daß eine einzelne Rechenvorschrift F der Form (R) von Abschnitt 3.3 ohne Aufrufe anderer Rechenvorschriften gegeben ist, sowie ein Term e, bei dessen Aufbau nur die Konstanten und Operationen der elementaren Datenstrukturen und Aufrufe von F vorkommen. Ausgehend von e wird eine Folge von Termen $d_0, d_1, \ldots$ wie folgt konstruiert:

a) Starte mit $d_0 := e$.

b) Bilde den Term d_{i+1} aus seinem Vorgänger d_i in zwei Schritten:

   (1) **Expandiere** zuerst „gewisse“ Aufrufe der Rechenvorschrift F im Term d_i simultan.

   (2) **Vereinfache** dann den entstehenden Term mit Hilfe der Regeln der elementaren Datenstrukturen[9] und der zwei Regeln

$$\texttt{if true then } e_1 \texttt{ else } e_2 = e_1 \qquad \texttt{if false then } e_1 \texttt{ else } e_2 = e_2$$

[9] Diese Regeln werden üblicherweise nicht exakt spezifiziert. Das Vereinfachen von arithmetischen und Booleschen Ausdrücken lernt man schon in der Schule und für die Zeichenreihen setzen wir ein intuitives Verständnis für Vereinfachung voraus. Formalisiert handelt es sich bei Vereinfachung um die Berechnung von Normalformen von Termersetzungssystemen.

der Fallunterscheidung so weit wie möglich.

Ist die so konstruierte Folge endlich, weil keine Expansionen mehr möglich sind, und mit dem letztem Folgenglied d_N, so ist dieser geschlossene Term d_N das **Ergebnis** der Berechnung. Ansonsten, d.h. die beiden Schritte von b) sind immer wieder durchführbar, ist das Resultat per Definition undefiniert.

Zur Präzision ist noch festzulegen, was im Expansionsschritt, der **Parameterübergabe**, mit „gewisse“ gemeint ist. Hier gibt es drei Hauptregeln:

- **Leftmost-innermost-Substitution**: Expandiere von allen innersten F-Aufrufen (in deren Argumenten also F nicht mehr vorkommt) den linkesten (im Aufschreibungssinne).
- **Leftmost-outermost-Substitution**: Expandiere von allen äußersten F-Aufrufen (die also nicht in einem Argument eines F-Aufrufs vorkommen) den linkesten (im Aufschreibungssinne).
- **Volle Substitution**: Expandiere alle F-Aufrufe.

Die erste Regel, kurz auch LI-Regel genannt, besagt, daß bei einem Aufruf einer Rechenvorschrift zuerst alle Argumente auszuwerten und dann ihre Werte zu übergeben sind. Man spricht deshalb auch von einer **Call-by-value** Parameterübergabe. Eine Anwendung der zweiten Regel, die auch als LO-Regel bezeichnet wird, übergibt die Argumente textmäßig. Deshalb hat sich auch die Bezeichnung **Call-by-name** Parameterübergabe eingebürgert. Die Wahl des linkesten Aufrufs ist dabei durch die Fallunterscheidung bedingt, wo, als Zeichenreihe, die Bedingung links des then- und else-Zweigs steht. Aufrufe in der Bedingung sind natürlich vor denen des then- und des else-Zweigs auszuwerten. ■

Die bei der Termersetzungssemantik 3.4.2 erwähnten ersten zwei Substitutionen expandieren jeweils genau einen Aufruf und sind deshalb technisch einfach zu realisieren. Es war lange nicht klar, daß verschiedene Parameterübergabe-Mechanismen zu verschiedenen Ergebnissen führen können. Entdeckt wurde dieser Sachverhalt erst am Ende der 60er Jahre des letzten Jahrhunderts, als man sich intensiver mit Semantik beschäftigte. Nachfolgend geben wir ein berühmtes Beispiel an, das auf J.H. Morris (1968) zurückgeht.

3.4.3 Beispiel (Call-by-value ist ungleich Call-by-name)

Die folgende Rechenvorschrift zeigt, daß die Regeln Call-by-value und Call-by-name der Termersetzungssemantik 3.4.2 nicht immer das gleiche Resultat liefern müssen:

```
fun F (x : int, y : int) : int =
  if x = 0 then 1
           else F(x-1,F(x-y,y));
```

Wir betrachten den Aufruf `F(1,0)` und beginnen mit einer Auswertung nach der Call-by-value-Strategie. Der Beginn der Termfolge der Termersetzungssemantik 3.4.2 sieht dann wie folgt aus, wobei dasjenige Vorkommen von `F`, dessen Aufruf expandiert wird, durch Fettdruck hervorgehoben ist:

d_0 :	**`F(1,0)`**	Start
	`if 1 = 0 then 1 else  F(1-1,F(1-0,0))`	Expansion
d_1 :	`F(0,`**`F(1,0)`**`)`	Vereinfachung
	`F(0,if 1 = 0 then 1 else F(1-1,F(1-0,0)))`	Expansion
d_2 :	`F(0,F(0,`**`F(1,0)`**`))`	Vereinfachung
	$\vdots$	

Wie sofort ersichtlich ist, hat die Folge $d_0, d_1, d_2, \ldots$ kein letztes Glied; das Resultat der Berechnung von `F(1,0)` nach der LI-Regel ist also per Definition undefiniert.

Nun werten wir den gleichen Aufruf nach der Call-by-name-Regel aus. Hier bekommen wir als Folge von Termen:

d_0 :	**`F(1,0)`**	Start
	`if 1 = 0 then 1 else  F(1-1,F(1-0,0))`	Expansion
d_1 :	**`F(0,F(1,0))`**	Vereinfachung
	`if 0 = 0 then 1 else F(0-1,F(0-F(1,0),F(1,0)))`	Expansion
d_2 :	`1`	Vereinfachung

Damit ist die Folge $d_0, d_1, d_2, \ldots$ endlich mit letztem Glied $d_2 =$ `1`. Das Ergebnis der Berechnung ist somit der Term `1`. ■

Für die von uns in diesem Skriptum benutzte funktionale Sprache **ML ist per Semantikdefinition Call-by-value**, d.h. leftmost-innermost-Substitution, als Mechanismus zur Parameterübergabe vorgeschrieben.

Um den Studierenden den Zugang zur Termersetzungssemantik zu erleichtern, wurde in den letzten Jahren am Institut für Informatik und Praktische Mathematik der Universität Kiel das KIEL-System entwickelt. Es erlaubt, die termmäßige Auswertung von funktionalen Programmen in ML-Syntax benutzergesteuert mittels Einzelschritten, semiautomatisch oder sogar vollautomatisch zu visualisieren, und enthält als Optionen genau die drei oben genannten Substitutions-Mechanismen. Einzelheiten findet man auf der Web-Seite des Systems mit der URL www.informatik.uni-kiel.de/~progsys/kiel.shtml.

Wir haben bisher, der Einfachheit halber, die Termersetzungssemantik nur für eine einzelne Rechenvorschrift betrachtet. Sie kann natürlich auch auf mehrere Rechenvorschriften ausgedehnt werden. Für die Sprache ML heißt dies, daß bei mehreren Rechenvorschriften $F_1, \ldots, F_k$ derjenige Aufruf $F_i(A_1, \ldots, A_n)$ einer Rechenvorschrift F_i expandiert wird, in dessen Argumenten A_j, $1 \leq j \leq n$ kein Aufruf einer der vorliegenden Rechenvorschriften mehr vorkommt und welcher der linkeste im Aufschreibungssinne mit dieser Eigenschaft ist. Wegen der Call-by-value-Semantik von ML ermöglicht dies, die Mehrfachauswertung

von identischen Teiltermen durch Einführungen von Hilfsrechenvorschrift und Komprimieren zu verhindern. Dies kann zu einer beträchtlichen Steigerung der Effizienz führen, wie das nachfolgende Beispiel zeigt.

3.4.4 Beispiel (Verhinderung von Mehrfachauswertung bei Call-by-value)

Wir betrachten die nachfolgend gegebene linear-rekursive ML-Rechenvorschrift `power` zur Potenzierung einer natürlichen Zahl (erstes Argument), wobei das zweite Argument den Exponenten der Potenz angibt:

```
fun power (x : int, n : int) : int =
  if n = 0 then 1
           else x * power(x,n-1);
```

Für diese Rechenvorschrift gilt die Gleichung

$$\texttt{power(x,m+n)} = \texttt{power(x,m)} * \texttt{power(x,n)},$$

die man sehr einfach durch Induktion nach `n` beweist. Zwei unmittelbare Folgerungen dieser Gleichung sind

$$\texttt{power(x,n)} = \texttt{power(x,n div 2)} * \texttt{power(x,n div 2)},$$

falls das zweite Argument nicht Null und gerade ist, und

$$\texttt{power(x,n)} = x * \texttt{power(x,n div 2)} * \texttt{power(x,n div 2)},$$

falls das zweite Argument nicht Null und ungerade ist. Somit bekommen wir die folgende kaskadenartig-rekursive Variante von `power`:

```
fun power (x : int, n : int) : int =
  if n = 0 then 1
           else if n mod 2 = 0
                  then power(x,n div 2) * power(x,n div 2)
                  else x * power(x,n div 2) * power(x,n div 2);
```

Ein Aufruf dieser Rechenvorschrift führt bei einer Zweierpotenz 2^k als zweitem Argument zu $1 + 2 + 4 + \ldots + 2^{k+1}$ Aufrufen, also, nach der Summenformel der geometrischen Reihe, zu insgesamt zu $2^{k+2} - 1$ Aufrufen. Eine Effizienzverbesserung erhält man, indem man eine Rechenvorschrift `quad` zum Quadrieren von ganzen Zahlen einführt und die Quadrierungen im Rumpf von `power` komprimiert:

```
fun quad (n : int) : int = n * n;

fun power (x : int, n : int) : int =
  if n = 0 then 1
           else if n mod 2 = 0 then quad(power(x,n div 2))
                               else x * quad(power(x,n div 2));
```

Nun hat man bei 2^k als zweitem Argument nur mehr $k+2$ Aufrufe von `power` und $k+1$ Aufrufe von `quad`. Nachfolgend ist die Termfolge des Aufrufs `power(2,16)` angegeben:

$$
\begin{array}{rcl}
d_0 & : & \texttt{power(2,16)} \\
d_1 & : & \texttt{quad(power(2,8))} \\
d_2 & : & \texttt{quad(quad(power(2,4)))} \\
d_3 & : & \texttt{quad(quad(quad(power(2,2))))} \\
d_4 & : & \texttt{quad(quad(quad(quad(power(2,1)))))} \\
d_5 & : & \texttt{quad(quad(quad(quad(2 * quad(power(2,0))))))} \\
d_6 & : & \texttt{quad(quad(quad(quad(2 * quad(1)))))} \\
d_7 & : & \texttt{quad(quad(quad(quad(2))))} \\
d_8 & : & \texttt{quad(quad(quad(4)))} \\
d_9 & : & \texttt{quad(quad(16))} \\
d_{10} & : & \texttt{quad(256)} \\
d_{11} & : & \texttt{65536}
\end{array}
$$

Die genaue Anzahl der bei einem Aufruf von `power` auszuführenden Multiplikationen hängt von der Binärdarstellung des zweiten Arguments ab. Wir können aber nicht genauer auf diesen Sachverhalt eingehen. Festgehalten werden sollte aber, daß die letzte Version von `power` wesentlich effizienter ist als die erste Version, die sich direkt aus der induktiven Definition der Potenzen mittels $x^0 = 1$ und $x^{n+1} = x * x^n$ ergibt. ■

Soweit zur ersten Semantik. Da sie auch als ein Abarbeiten des Programms auf einer hypothetischen Maschine betrachtet werden kann – hier: einer Termersetzungsmaschine – wird sie als **operationelle Semantik** bezeichnet. Bei der nun folgenden zweiten Art wird einer Rechenvorschrift „direkt“ eine (partielle) Funktion zugeordnet. Diese Art von Semantik ohne den Umweg einer Abarbeitung heißt **denotationell**.

3.4.5 Beispiel (zur Motivation)

Wir betrachten noch einmal die Rechenvorschrift `fac` von Beispiel 3.3.2.b zur Berechnung der Fakultät einer natürlichen Zahl:

```
fun fac (n : int) : int =
  if n = 0 then 1
           else n * fac(n-1);
```

Durch ihre Rekursion wird eine (eventuell partielle) Funktion $f : \mathbb{Z} \longrightarrow \mathbb{Z}$ beschrieben, welche die folgenden zwei Bedingungen erfüllt:

$$(1)\quad f(0) = 1 \qquad\qquad (2)\quad \forall\, n \in \mathbb{Z} : n \neq 0 \rightarrow f(n) = n * f(n-1)$$

Mit Hilfe eines **Funktionals**[10], d.h. einer Funktion, die (partielle) Funktionen auf (partielle) Funktionen abbildet, lassen sich die beiden Formeln (1) und (2) in eine Gleichung

[10] Ein Funktional kennt man schon von der höheren Schule, auch wenn der Begriff da nicht explizit verwendet wird, nämlich die Bildung der Ableitung.

zusammenfassen. Definiert man nämlich ein Funktional $\tau_{\mathtt{fac}} : \mathbb{Z}^{\mathbb{Z}} \longrightarrow \mathbb{Z}^{\mathbb{Z}}$, das einer (partiellen) Funktion $f : \mathbb{Z} \longrightarrow \mathbb{Z}$ eine (partielle) Funktion $\tau_{\mathtt{fac}}[f] : \mathbb{Z} \longrightarrow \mathbb{Z}$ zuordnet, durch die elementweise Definition (mit den eckigen Klammern für die Anwendung des Funktionals, dies hat sich so eingebürgert)

$$\tau_{\mathtt{fac}}[f](n) = \begin{cases} 1 & : \quad n = 0 \\ n * f(n-1) & : \quad n \neq 0 \end{cases}$$

von $\tau_{\mathtt{fac}}[f]$, so sind die beiden Formeln (1) und (2) äquivalent zu Gleichung

$$\forall\, n \in \mathbb{Z} : f(n) = \tau_{\mathtt{fac}}[f](n).$$

Nun wenden wir, um auch noch den Allquantor zu entfernen, die Definition der Gleichheit von (partiellen) Funktionen an. Diese besagt[11], daß zwei (partielle) Funktionen genau dann gleich sind, wenn sie für jedes Argument das gleiche Resultat liefern. Folglich erhalten wir (1) und (2) äquivalenterweise als sogenannte **Fixpunktgleichung**

$$f = \tau_{\mathtt{fac}}[f].$$

Rekursive Rechenvorschriften induzieren also Fixpunktgleichungen. Diese müssen aber keinesfalls eine eindeutige Lösung besitzen. Zur Angabe von zwei Lösungen einer Fixpunktgleichung betrachten wir wiederum unser Fakultäts-Beispiel:

a) Offensichtlich gilt $\tau_{\mathtt{fac}}[f_1] = f_1$ (man sagt: f_1 ist ein **Fixpunkt** von $\tau_{\mathtt{fac}}$) für die partielle Funktion

$$f_1 : \mathbb{Z} \longrightarrow \mathbb{Z} \qquad\qquad f_1(n) = \begin{cases} n! & : \quad n \geq 0 \\ \text{undef} & : \quad n < 0. \end{cases}$$

Der Beweis erfolgt einfach durch Einsetzung. Man hat drei Fälle. Ist $n > 0$, so gilt:

$$\begin{array}{lcll} \tau_{\mathtt{fac}}[f_1](n) & = & n * f_1(n-1) & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & n * (n-1)! & \text{Definition von } f_1 \\ & = & n! & \\ & = & f_1(n) & \text{Definition von } f_1 \end{array}$$

Nun gelte $n = 0$. Hier bekommen wir:

$$\begin{array}{lcll} \tau_{\mathtt{fac}}[f_1](n) & = & 1 & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & n! & \\ & = & f_1(n) & \text{Definition von } f_1 \end{array}$$

Als letzter Fall bleibt noch $n < 0$. Auch hier gilt die beabsichtigte Gleichung, wie die nachstehende Rechnung zeigt:

$$\begin{array}{lcll} \tau_{\mathtt{fac}}[f_1](n) & = & n * f_1(n-1) & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & n * \text{undef} & \text{Definition von } f_1 \\ & = & \text{undef} & \text{Striktheit der Basisoperationen} \\ & = & f_1(n) & \text{Definition von } f_1 \end{array}$$

Also gilt $\tau_{\mathtt{fac}}[f_1](n) = f_1(n)$ für alle $n \in \mathbb{Z}$, und das ist genau $\tau_{\mathtt{fac}}[f_1] = f_1$ nach der Definition der Gleichheit von partiellen Funktionen.

[11] Man vergleiche dies mit der Definition von (partiellen) Funktionen als spezielle Relationen, also Mengen, in Abschnitt 1.1.

b) Es gilt die Fixpunkt-Eigenschaft $\tau_{\texttt{fac}}[f_2] = f_2$ aber auch für die nachfolgend angegebene Funktion, welche die Fakultätsfunktion auf die negativen Zahlen konstant mit Null fortsetzt:

$$f_2 : \mathbb{Z} \longrightarrow \mathbb{Z} \qquad\qquad f_2(n) = \begin{cases} n! & : \quad n \geq 0 \\ 0 & : \quad n < 0 \end{cases}$$

Dazu muß man nur noch $\tau_{\texttt{fac}}[f_2](n) = f_2(n)$ für alle $n \in \mathbb{Z}$ mit $n < 0$ nachrechnen, was aber trivial ist:

$$\begin{array}{lcll} \tau_{\texttt{fac}}[f_2](n) & = & n * f_2(n-1) & \text{Definition von } \tau_{\texttt{fac}} \\ & = & n * 0 & \text{Definition von } f_2 \\ & = & 0 & \\ & = & f_2(n) & \text{Definition von } f_2 \end{array}$$

Vergleicht man die eben erzielten Ergebnisse mit der Termersetzungssemantik 3.4.2, so entspricht die termmäßige Auswertung der Fakultätsrechenvorschrift offensichtlich der Funktion f_1, da für ein negatives Argument `n` die Folge

$$\texttt{fac(n)}, \texttt{n * fac(n-1)}, \texttt{n * (n-1) * fac(n-2)}, \ldots$$

der von der Termersetzungssemantik produzierten Terme nicht endlich ist. ■

Nach diesen Vorbemerkungen sind wir nun in der Lage, die denotationelle Semantik von ML-Rechenvorschriften zu beschreiben. Bei dieser Semantik spielen Mengen von Funktionen eine entschiedende Rolle und die übliche Darstellung N^M erweist sich als hinderlich. Stattdessen verwendet man deshalb in der Literatur $[M \to N]$ **als Bezeichnung für die Menge der (partiellen) Funktionen** von M nach N. Auch wir werden im folgenden diese Notation verwenden.

Das Vorgehen bei der denotationellen Semantik von ML-Rechenvorschriften ist wie folgt. In einem ersten Schritt interpretiert man die ML-Sorten s, wie am Anfang von Abschnitt 3.2 erwähnt, durch Mengen s^A aus der Algebra A zur Signatur von ML. Dann ordnet man im zweiten Schritt einer Rechenvorschrift ein Funktional zu, das auf partiellen Funktionen operiert. Nachfolgend geben wir diese Konstruktion formal an, wobei die Definition so gewählt ist, daß sie mit der operationellen Call-by-value Termersetzungssemantik 3.4.2 verträglich ist.

3.4.6 Definition (Induziertes Funktional)

Wir gehen aus von einer ML-Rechenvorschrift der Form (R) von Abschnitt 3.3

$$\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{,}\ldots\texttt{,}x_k\ \texttt{:}\ m_k\texttt{)}\ \texttt{:}\ n\ \texttt{=}\ t\texttt{;}$$

in der, wie vorausgesetzt, keine Aufrufe einer anderen Rechenvorschrift vorkommen. Dann ist zu den Sorteninterpretationen $m_i^A, 1 \leq i \leq k$, und n^A in der zugrundeliegenden Algebra

das **von F induzierte Funktional**

$$\tau_F : [\prod_{i=1}^{k} m_i^A \to n^A] \longrightarrow [\prod_{i=1}^{k} m_i^A \to n^A]$$

für eine partielle Funktion $f : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ und ein Tupel $v := \langle v_1, \ldots, v_k \rangle \in \prod_{i=1}^{k} m_i^A$ festgelegt als

$$\tau_F[f](v) = [\![t]\!][f](v),$$

wobei $[\![t]\!][f](v)$, die Semantik des Terms t, induktiv wie folgt definiert ist:

a) Ist der Term t eine **Konstante** c der elementaren Datenstrukturen, so gilt:

$$[\![c]\!][f](v) = c^A$$

b) Ist der Term t einer der **Parameter** x_i, $1 \leq i \leq k$, der Rechenvorschrift F, so gilt:

$$[\![x_i]\!][f](v) = v_i$$

c) Ist der Term t von der Form $g(t_1, \ldots, t_n)$ mit einer **Operation** g der elementaren Datenstrukturen, so gilt:

$$[\![g(t_1, \ldots, t_n)]\!][f](v) = g^A([\![t_1]\!][f](v), \ldots, [\![t_n]\!][f](v))$$

d) Ist der Term t eine **Fallunterscheidung** `if` b `then` e_1 `else` e_2, so gilt:

$$[\![\texttt{if } b \texttt{ then } e_1 \texttt{ else } e_2]\!][f](v) = \begin{cases} [\![e_1]\!][f](v) & : \quad [\![b]\!][f](v) = tt \\ [\![e_2]\!][f](v) & : \quad [\![b]\!][f](v) = ff \\ \text{undef} & : \quad [\![b]\!][f](v) = \text{undef} \end{cases}$$

e) Hat der Term t die Form $F(t_1, \ldots, t_k)$ eines **rekursiven Aufrufs**, so gilt:

$$[\![F(t_1, \ldots, t_k)]\!][f](v) = f([\![t_1]\!][f](v), \ldots, [\![t_k]\!][f](v))$$

Dabei wird in c) durch g^A die Interpretation der Operation g in der Algebra zur Signatur von ML bezeichnet. Im allgemeinen ist g^A eine partielle Funktion und die Funktionsanwendung ist somit undefiniert, falls es ein i, $1 \leq i \leq n$, mit $[\![t_i]\!][f](v) = \text{undef}$ gibt. Bei den rekursiven Aufrufen in e) trifft die gleiche Aussage wie bei den Operationen unter c) zu: Gilt also für ein i, $1 \leq i \leq k$, die Gleichung $[\![t_i]\!][f](v) = \text{undef}$, so folgt daraus $f([\![t_1]\!][f](v), \ldots, [\![t_k]\!][f](v)) = \text{undef}$. ■

Man beachte, daß wir in der Definition 3.4.6 des von einer Rechenvorschrift induzierten Funktionals alles auf die Funktionsanwendung bei partiellen Funktionen zurückgeführt haben, die **strikt** ist in dem Sinne, daß **ein undefiniertes Argument ein undefiniertes Resultat** impliziert. Daher rührt die Verträglichkeit der späteren denotationellen Semantik 3.4.10 mit der Call-by-value Termersetzungssemantik.

An dieser Stelle ist eine Bemerkung zur Call-by-name-Semantik angebracht. Zur Definition einer solchen Semantik auf denotationelle Art ist die strikte Funktionsanwendung bei partiellen Funktionen nicht geeignet. Stattdessen erweitert man die Interpretationen der Sorten um ein Element $\perp$, das für „undefiniert“ steht, verwendet nur (totale) Funktionen und definiert dann die Fehlerfortpflanzung so, daß sie einer Call-by-name-Semantik entspricht.

Es sei F die in Definition 3.4.6 betrachtete Rechenvorschrift. Wie schon das motivierende Beispiel 3.4.5 zeigt, hat man nun in einem dritten Schritt einen Fixpunkt des induzierten Funktionals τ_F als denotationelle Semantik F zu wählen. Dabei stellt sich die folgende Frage: Hat so ein induziertes Funktional überhaupt Fixpunkte und wenn ja, welchen Fixpunkt muß man wählen, damit sich eine Übereinstimmung zwischen denotationeller und operationeller Semantik ergibt? Eine Lösung dieses Problems wird im weiteren Verlauf dieses Abschnitts entwickelt. Wir starten dabei wie folgt:

3.4.7 Definition (Beziehung zwischen Funktionen)

Es seien $f, g : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ zwei partielle Funktionen. Man definiert, mit v als Abkürzung für das Tupel $\langle v_1, \ldots, v_k \rangle \in \prod_{i=1}^{k} m_i^A$, eine Beziehung $\leq$ zwischen ihnen durch

$$f \leq g \quad :\Longleftrightarrow \quad \forall\, v \in \prod_{i=1}^{k} m_i^A : f(v) \neq \text{undef} \rightarrow f(v) = g(v). \qquad \blacksquare$$

Betrachtet man f und g in dieser Definition als eindeutige Relationen, so gilt $f \leq g$ genau dann, wenn $f \subseteq g$ zutrifft. Offensichtlich folgt daraus sofort der nachstehende Satz. Man kann diesen aber auch ohne Mühe direkt auf Definition 3.4.7 aufbauend beweisen; dem Leser sei dies zur Übung empfohlen.

3.4.8 Satz (Funktionsordnung)

Die durch Definition 3.4.7 festgelegte Relation $\leq$ ist eine Ordnungsrelation auf der Menge $[\prod_{i=1}^{k} m_i^A \rightarrow n^A]$ der partiellen Funktionen. von $\prod_{i=1}^{k} m_i^A$ nach n^A. $\blacksquare$

Zwei partielle Funktionen f und g stehen in der Relation $f \leq g$ genau dann, wenn ihre definierten Werte übereinstimmen, aber g eventuell für mehr Argumente definiert ist als f. Beispielsweise stehen die beiden partiellen Funktionen $f_1, f_2 : \mathbb{Z} \longrightarrow \mathbb{Z}$ von Beispiel 3.4.5 in der Ordnung $f_1 \leq f_2$. Dieses Beispiel motiviert auch, daß man als denotationelle Semantik der Rechenvorschrift F den bezüglich dieser Ordnung kleinsten Fixpunkt des induzierten Funktionals zu wählen hat. (Eine formale Rechtfertigung dieser Wahl ist im Rahmen dieser Einführungsvorlesung natürlich nicht möglich.) So ein kleinster Fixpunkt von τ_F existiert immer. Der folgende Satz gibt sogar ein einfaches iteratives Berechnungsverfahren an. Sein exakter mathematischer Hintergrund ist die Theorie der vollständig geordneten Mengen (englisch: complete partial orders), auf die wir aber nicht genauer eingehen können, da

uns derzeit die entsprechenden mathematischen Mittel noch nicht zur Verfügung stehen. Deshalb wird der Satz auch nicht bewiesen.

3.4.9 Satz (Fixpunktsatz)

Für eine ML-Rechenvorschrift F, wie in Definition 3.4.6 betrachtet, gelten die folgenden zwei Aussagen:

a) Das von F induzierte Funktional τ_F besitzt bezüglich der in Definition 3.4.7 festgelegten Ordnung $\leq$ einen kleinsten Fixpunkt, der mit $\mu(\tau_F)$ bezeichnet wird.

b) Mit Hilfe der überall undefinierten partiellen Funktion $\Omega(v_1, \ldots, v_k) :=$ undef von $\prod_{i=1}^{k} m_i^A$ nach n^A ergibt sich der kleinste Fixpunkt $\mu(\tau_F)$ als der Grenzwert der Kette $\Omega \leq \tau_F[\Omega] \leq \tau_F[\tau_F[\Omega]] \leq \ldots$ der iterierten Anwendungen des Funktionals τ_F auf die partielle Funktion Ω. ■

Faßt man partielle Funktionen als eindeutige Relationen, also Mengen, auf, so ist der Grenzwert der Kette $\Omega \leq \tau_F[\Omega] \leq \tau_F[\tau_F[\Omega]] \leq \ldots$ genau die Vereinigung $\bigcup_{i \geq 0} \tau_F^i[\Omega]$. Allgemein handelt es sich bei dem Grenzwert um das Supremum, also eine (partielle) Funktion, die größer oder gleich jedem Glied der Kette ist aber kleiner oder gleich jeder anderen (partiellen) Funktion mit dieser Eigenschaft.

Nach all diesen Vorbereitungen können wir nun endlich festlegen, was die denotationelle Semantik einer ML-Rechenvorschrift ist.

3.4.10 Denotationelle Semantik von Rechenvorschriften

Für eine ML-Rechenvorschrift F, wie in Definition 3.4.6 betrachtet, ist die **denotationelle Semantik** definiert als der kleinste Fixpunkt $\mu(\tau_F) : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ des von F induzierten Funktionals τ_F. ■

Nach dieser formalen Definition der denotationellen Semantik einer ML-Rechenvorschrift greifen wir das motivierende Beispiel 3.4.5 noch einmal auf und zeigen, daß die in ihm angegebene Rechenvorschrift tatsächlich die partielle Funktion f_1 als denotationelle Semantik besitzt.

3.4.11 Beispiel (zur denotationellen Semantik)

Für die ML-Rechenvorschrift `fac` von Beispiel 3.4.5 zur Berechnung der Fakultät einer natürlichen Zahl haben wir das induzierte Funktional $\tau_{\texttt{fac}}$ schon in diesem Beispiel aufgeführt. Nachfolgend berechnen wir das Funktional $\tau_{\texttt{fac}}$ mit Hilfe der in Definition 3.4.6

angegebenen formalen Festlegung aus dem Rumpf der Rechenvorschrift:

$$\begin{aligned}
\tau_{\texttt{fac}}[f](n) &= [\![\texttt{if n = 0 then 1 else n * fac(n-1)}]\!][f](n) \\
&= \begin{cases} [\![\texttt{1}]\!][f](n) & : \ [\![\texttt{n = 0}]\!][f](n) = tt \\ [\![\texttt{n * fac(n-1)}]\!][f](n) & : \ [\![\texttt{n = 0}]\!][f](n) = ff \\ \text{undef} & : \ [\![\texttt{n = 0}]\!][f](n) = \text{undef} \end{cases} \\
&= \begin{cases} 1 & : \ [\![\texttt{n}]\!][f](n) = 0 \\ [\![\texttt{n}]\!][f](n) * [\![\texttt{fac(n-1)}]\!][f](n) & : \ [\![\texttt{n}]\!][f](n) \neq 0 \\ \text{undef} & : \ [\![\texttt{n}]\!][f](n) = \text{undef} \end{cases} \\
&= \begin{cases} 1 & : \ n = 0 \\ n * f([\![\texttt{n-1}]\!][f](n)) & : \ n \neq 0 \end{cases} \\
&= \begin{cases} 1 & : \ n = 0 \\ n * f([\![\texttt{n}]\!][f](n) - 1) & : \ n \neq 0 \end{cases} \\
&= \begin{cases} 1 & : \ n = 0 \\ n * f(n-1) & : \ n \neq 0 \end{cases}
\end{aligned}$$

Zur Bestimmung des kleinsten Fixpunktes iterieren wir nun gemäß dem Fixpunktsatz 3.4.9. Dabei betrachten wir nur die nichtnegativen Argumente, da für negative Argumente das Resultat immer undefiniert ist. Wir starten mit der überall undefinierten partiellen Funktion, deren „nichtnegatives Anfangsstück“ wir tabellarisch nachstehend angeben:

Ω

...	0	1	2	3	4	5	...
...	undef	undef	undef	undef	undef	undef	...

Im ersten Schritt berechnen wir $\tau_{\texttt{fac}}[\Omega]$. Es gilt

$$\begin{aligned}
\tau_{\texttt{fac}}[\Omega](0) &= 1 \\
\tau_{\texttt{fac}}[\Omega](1) &= 1 * \Omega(0) = 1 * \text{undef} = \text{undef}
\end{aligned}$$

und damit $\tau_{\texttt{fac}}[\Omega](n) = \text{undef}$ für alle $n \geq 1$. Wir erhalten somit in unserer tabellarischen Darstellung das folgende Resultat:

$\tau[\Omega]$

...	0	1	2	3	4	5	...
...	1	undef	undef	undef	undef	undef	...

Als nächstes geben wir auch noch den zweiten Schritt $\tau^2_{\texttt{fac}}[\Omega] = \tau_{\texttt{fac}}[\tau_{\texttt{fac}}[\Omega]]$ der iterativen Berechnung des kleinsten Fixpunktes an. Dazu berechnen wir

$$\begin{aligned}
\tau^2_{\texttt{fac}}[\Omega](0) &= 1 \\
\tau^2_{\texttt{fac}}[\Omega](1) &= 1 * \tau_{\texttt{fac}}[\Omega](0) = 1 * 1 = 1 \\
\tau^2_{\texttt{fac}}[\Omega](2) &= 2 * \tau_{\texttt{fac}}[\Omega](1) = 2 * \text{undef} = \text{undef}
\end{aligned}$$

und bekommen daraus $\tau^2_{\mathtt{fac}}[\Omega](n) = \text{undef}$ für alle $n \geq 2$. Dies führt, in unserer tabellarischen Darstellung, zu der folgenden partiellen Funktion:

$\tau^2_{\mathtt{fac}}[\Omega]$

...	0	1	2	3	4	5	...
...	1	1	undef	undef	undef	undef	...

Nach diesen beiden Schritten ist schon ersichtlich, daß jeder weitere Schritt der Iteration für genau eine zusätzliche natürliche Zahl die Fakultät berechnet. Schließlich landet man beim Grenzwert $\tau^\infty_{\mathtt{fac}}[\Omega]$, der in unserer tabellarischen Darstellung wie folgt aussieht:

$\tau^\infty_{\mathtt{fac}}[\Omega]$

...	0	1	2	3	4	5	...
...	1	1	2	6	24	120	...

Dies ist genau die in Beispiel 3.4.5 angegebene Funktion f_1.

Berechnet man mit Hilfe der Iteration des Fixpunktsatzes einige Glieder der zum kleinsten Fixpunkt führenden Kette, so bekommt man oft genügend viele Hinweise, um dessen allgemeine Gestalt zu bestimmen. Man vergleiche mit den obigen Tabellen. Nachfolgend stellen wir nun beispielhaft noch zwei Methoden vor, mit denen man von einer gegebenen (partiellen) Funktion oft zeigen kann, daß sie der kleinste Fixpunkt eines gegebenen Funktionals ist.

Behauptung: Die partielle Funktion

$$f_1 : \mathbb{Z} \longrightarrow \mathbb{Z} \qquad f_1(n) = \begin{cases} n! & : \quad n \geq 0 \\ \text{undef} & : \quad n < 0. \end{cases}$$

ist der kleinste Fixpunkt $\mu(\tau_{\mathtt{fac}})$ des Funktionals $\tau_{\mathtt{fac}}$.

Beweis: Die Fixpunkteigenschaft $f_1 = \tau_{\mathtt{fac}}[f_1]$ wurde schon in Beispiel 3.4.5.a gezeigt. Wir haben also noch zu verifizieren, daß f_1 der kleinste Fixpunkt des Funktionals $\tau_{\mathtt{fac}}$ zur Fakultäts-Rechenvorschrift ist.

a) Eine **erste (direkte) Beweismethode** nimmt einen weiteren Fixpunkt f des Funktionals $\tau_{\mathtt{fac}}$ an und zeigt die Implikation

$$f_1(n) \neq \text{undef} \quad \Longrightarrow \quad f_1(n) = f(n) \tag{$\dagger$}$$

für alle $n \in \mathbb{Z}$, da dies gleichwertig zu $f_1 \leq f$ ist. Beim Beweis von ($\dagger$) unterscheiden wir zwei Fälle.

Es sei $n < 0$. Dann gilt die Implikation ($\dagger$), denn wegen $f_1(n) = \text{undef}$ ist die Voraussetzung falsch.

Nun sei $n \geq 0$. Hier ist $f_1(n) \neq \text{undef}$ wahr, und die Implikation ($\dagger$) gilt genau dann, falls $f_1(n) = f(n)$ zutrifft. Wir verwenden zum Beweis dieser Gleichung das Prinzip der noetherschen Induktion:

Den **Induktionsbeginn** $n = 0$ zeigt man wie folgt:

$$\begin{array}{rcll} f_1(n) & = & \tau_{\mathtt{fac}}[f_1](n) & f_1 \text{ ist Fixpunkt} \\ & = & 1 & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & \tau_{\mathtt{fac}}[f](n) & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & f(n) & f \text{ ist Fixpunkt} \end{array}$$

Zum **Induktionsschluß** seien nun eine natürliche Zahl n mit $n \neq 0$ gegeben. Dann können wir wie folgt schließen:

$$\begin{array}{rcll} f_1(n) & = & \tau_{\mathtt{fac}}[f_1](n) & f_1 \text{ ist Fixpunkt} \\ & = & n * f_1(n-1) & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & n * f(n-1) & \text{Induktionshypothese} \\ & = & \tau_{\mathtt{fac}}[f](n) & \text{Definition von } \tau_{\mathtt{fac}} \\ & = & f(n) & f \text{ ist Fixpunkt} \end{array}$$

b) Zu einem **zweiten Beweis durch Widerspruch** nehmen wir an, daß f_1 nicht der kleinste Fixpunkt von $\tau_{\mathtt{fac}}$ sei, sondern die echt kleinere partielle Funktion f. Also gibt es ein $n_0 \in \mathbb{Z}$ mit $f(n_0) = \text{undef}$ und $f_1(n_0) \neq \text{undef}$. Nach der Definition von f_1 und der Fixpunktgleichung $\tau_{\mathtt{fac}}[f] = f$ muß $n_0 > 0$ gelten. Wir nehmen nun zusätzlich an, daß n_0 bezüglich der üblichen Ordnung auf $\mathbb{N}$ minimal gewählt ist. Somit gilt die Ungleichung $f(n_0 - 1) \neq \text{undef}$ und dies bringt

$$\begin{array}{rcll} f(n_0) & = & \tau_{\mathtt{fac}}[f](n_0) & f \text{ ist Fixpunkt} \\ & = & n_0 * f(n_0 - 1) & \text{Definition von } \tau_{\mathtt{fac}} \\ & \neq & \text{undef} & \text{nach Voraussetzung,} \end{array}$$

also einen Widerspruch zu $f(n_0) = \text{undef}$. ■

Aus der Festlegung der Semantik einer Rechenvorschrift F als den kleinsten Fixpunkt $\mu(\tau_F)$ des von F induzierten Funktionals τ_F ergibt sich eine wichtige Konsequenz bezüglich des Entwickelns von Rekursionen bzw. des Umformens / Manipulierens von Rechenvorschriften. Wir zeigen die Problematik am ersten Punkt und hier speziell am Beispiel der Fakultät auf.

3.4.12 Beispiel (für ein Problem beim Gewinnen von Rekursionen)

Bisher hatten wir zu einer gegebenen (partiellen) Funktion zur Gewinnung eines funktionalen Programms in einem ersten Schritt immer eine rekursive Darstellung entwickelt, d.h. genaugenommen eine Fixpunktgleichung bewiesen. So führt die Spezifikation

$$fac_{\mathbb{Z}} : \mathbb{Z} \longrightarrow \mathbb{Z} \qquad fac_{\mathbb{Z}} = \begin{cases} n! & : \quad n \geq 0 \\ \text{undef} & : \quad n < 0 \end{cases} \qquad (Spec)$$

der partiellen Fakultätsfunktion zu $fac_{\mathbb{Z}}(0) = 1$ und $fac_{\mathbb{Z}}(n) = n * fac_{\mathbb{Z}}(n-1)$ für alle $n \neq 0$. Unter Verwendung des Fakultätsfunktionals wird dies nun zur Fixpunktgleichung $fac_{\mathbb{Z}} = \tau_{\texttt{fac}}[fac_{\mathbb{Z}}]$. In einer Form mit Funktionsanwendung schreibt sich diese als

$$fac_{\mathbb{Z}}(n) = \tau_{\texttt{fac}}[fac_{\mathbb{Z}}](n) = \begin{cases} 1 & : \quad n = 0 \\ n * fac_{\mathbb{Z}}(n-1) & : \quad n \neq 0 \end{cases} \qquad (Rec)$$

für alle $n \in \mathbb{Z}$. Damit haben wir eine Gestalt, von der man sofort durch einen Übergang nach ML, und das war dann jeweils der zweite Schritt der Programmentwicklung, die funktionale Rechenvorschrift

```
fun fac (n : int) : int =
  if n = 0 then 1
           else n * fac(n-1);
```
$$(Prog)$$

bekommt. Insbesondere ist nach der Herleitung die in (*Spec*) ursprünglich betrachtete partielle Funktion ein Fixpunkt der Gleichung (*Rec*). Da das Funktional dieser Gleichung genau dem von der Rechenvorschrift aus (*Prog*) induzierten Funktional entspricht, bekommen wir $\tau_{\texttt{fac}}[fac_{\mathbb{Z}}] = fac_{\mathbb{Z}}$, woraus $\mu(\tau_{\texttt{fac}}) \leq fac_{\mathbb{Z}}$ folgt, also, nach Definition der Funktionsordnung,

$$\mu(\tau_{\texttt{fac}})(n) \neq \text{undef} \quad \Longrightarrow \quad \mu(\tau_{\texttt{fac}})(n) = fac_{\mathbb{Z}}(n)$$

für alle $n \in \mathbb{Z}$. Die Semantik der Rechenvorschrift `fac` stimmt nach dieser Implikation mit der Spezifikation $fac_{\mathbb{Z}}$ somit nur auf ihrem definierten Bereich überein. Gewünscht ist natürlich ein Beweis, daß die Semantik der Rechenvorschrift mit der Spezifikation überall übereinstimmt. ■

Damit jetzt, von dem eben im Beispiel gezeigten speziellen Falle abstrahierend, die Originalspezifikation und die Semantik der aus einer abgeleiteten Rekursion sich ergebenden Rechenvorschrift überall übereinstimmen, hat man zusätzlich in einem dritten Schritt noch zu verifizieren, daß die Definiertheit erhalten bleibt. Formal wird dies in dem nachfolgenden Satz bewiesen.

3.4.13 Satz

Für eine ML-Rechenvorschrift F, wie in Definition 3.4.6 betrachtet, und eine partielle Funktion $f : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ folgt aus

$$f = \tau_F[f]$$

und der Implikation („Erhaltung der Definiertheit“)

$$f(v_1, \ldots, v_k) \neq \text{undef} \quad \Longrightarrow \quad \mu(\tau_F)(v_1, \ldots, v_k) \neq \text{undef}$$

für alle $\langle v_1, \ldots, v_k \rangle \in \prod_{i=1}^{k} m_i^A$ die Gleichheit $\mu(\tau_F) = f$.

Beweis: Gegeben sei ein k-Tupel $\langle v_1, \ldots, v_k \rangle \in \prod_{i=1}^{k} m_i^A$. Zum Beweis der Gleichung $\mu(\tau_F)(v_1, \ldots, v_k) = f(v_1, \ldots, v_k)$ unterscheiden wir zwei Fälle:

Es sei $\mu(\tau_F)(v_1, \ldots, v_k) \neq$ undef. Aus der ersten Prämisse $f = \tau_F[f]$ folgt sofort die Beziehung $\mu(\tau_F) \leq f$, also

$$\mu(\tau_F)(v_1, \ldots, v_k) \neq \text{undef} \implies \mu(\tau_F)(v_1, \ldots, v_k) = f(v_1, \ldots, v_k)$$

nach der Definition der Ordnung auf den partiellen Funktionen. Da die linke Seite dieser Implikation nach der gemachten Voraussetzung wahr ist, gilt auch die rechte Seite.

Nun sei $\mu(\tau_F)(v_1, \ldots, v_k) =$ undef. Nach der zweiten Prämisse folgt daraus durch Kontraposition, d.h. Negation der Formeln der Implikation bei gleichzeitiger Vertauschung, daß $f(v_1, \ldots, v_k) =$ undef, also auch $\mu(\tau_F)(v_1, \ldots, v_k) = f(v_1, \ldots, v_k)$. ■

Aus Satz 3.4.13 folgt unmittelbar das nachfolgende Prinzip bei der Gewinnung von rekursiven Rechenvorschriften aus gegebenen (partiellen) Funktionen: Rechnet man für eine (partielle) Funktion eine Rekursion aus, so ist der Übergang zur entsprechenden rekursiven Rechenvorschrift nur dann erlaubt, wenn die Definiertheit erhalten bleibt, d.h. die Definiertheit der (partiellen) Funktion die Termininierung der Rechenvorschrift (für das gleiche Argument) impliziert. Gleiches gilt auch für Rechenvorschriften bei deren Manipulation: Berechnet man durch Expandieren und Komprimieren eine neue Version einer Rechenvorschrift, so sind beide nur dann semantisch gleichwertig, wenn die Terminierung des Originals die der neuen Version impliziert.

Bei all unseren bisherigen Rechnungen in den Beispielen 3.1.1, 3.1.2, 3.3.2, 3.3.5.b und 3.3.5.c waren diese Bedingungen erfüllt und somit die Übergänge korrekt. Dies muß aber nicht immer so sein. Nachfolgend geben wir ein Beispiel an.

3.4.14 Beispiel (für eine inkorrekte bzw. korrekte Entwicklung)

Wir betrachten die folgende partielle Funktion als Spezifikation der ganzzahligen Division von zwei ganzen Zahlen:

$$div : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z} \qquad div(m,n) = \begin{cases} max\{\ x \in \mathbb{N} \ :\ x \leq \frac{m}{n}\ \} & : \ m \geq 0, n > 0 \\ \text{undef} & : \ \text{sonst} \end{cases}$$

Im folgenden berechnen wir eine rekursive Darstellung für div. Dazu seien $m, n \in \mathbb{Z}$ mit $m \geq 0$ und $n > 0$ vorgegeben. Wir berechnen:

$$\begin{array}{rcll} div(m,n) & = & max\{\ x \in \mathbb{N} \ :\ x \leq \frac{m}{n}\ \} & \text{Definition von } div \\ & = & max\{\ x \in \mathbb{N} \ :\ x \leq \frac{m+n}{n} - 1\ \} & \text{Arithmetik} \\ & = & max\{\ x \in \mathbb{N} \ :\ x \leq \frac{m+n}{n}\ \} - 1 & \text{Arithmetik} \\ & = & div(m+n,n) - 1 & \text{Definition von } div \end{array}$$

Diese Rekursion gilt auch für die restlichen Paare $m, n \in \mathbb{Z}$ und die zu ihr gehörende ML-Rechenvorschrift

```
fun divide (m : int, n : int) : int =
  divide(m+n,n) - 1;
```

induziert zwar ein Funktional

$$\tau_{\texttt{divide}} : [\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}] \longrightarrow [\mathbb{Z} \times \mathbb{Z} \to \mathbb{Z}] \qquad \tau_{\texttt{divide}}[f](m,n) = f(m+n,n) - 1$$

mit $\tau_{\texttt{divide}}[div] = div$, wegen $\mu(\tau_{\texttt{divide}}) = \Omega \neq div$ ist sie jedoch zur Berechnung von *div* nicht geeignet. Die Rechenvorschrift `divide` terminiert offensichtlich für kein Argument.

Man braucht zur algorithmischen Realisierung der partiellen Divisionsfunktion *div* eine Rechenvorschrift, die mindestens für die Argumente terminiert, für die auch *div* definierte Werte liefert. Dazu kann man wie folgt vorgehen: Seien $m \geq 0$ und $n > 0$. Ist $m < n$, so gilt offensichtlich

$$\begin{array}{rcll} div(m,n) & = & max\{\ x \in \mathbb{N}\ :\ x \leq \frac{m}{n}\ \} & \text{Definition von } div \\ & = & 0 & \text{da } 0 \leq \frac{m}{n} < 1. \end{array}$$

Im anderen Falle $m \geq n$ schließt man wie folgt:

$$\begin{array}{rcll} div(m,n) & = & max\{\ x \in \mathbb{N}\ :\ x \leq \frac{m}{n}\ \} & \text{Definition von } div \\ & = & max\{\ x \in \mathbb{N}\ :\ x \leq \frac{m-n}{n} + 1\ \} & \text{Arithmetik} \\ & = & max\{\ x \in \mathbb{N}\ :\ x \leq \frac{m-n}{n}\ \} + 1 & \text{Arithmetik} \\ & = & div(m-n,n) + 1 & \text{Definition von } div \end{array}$$

Dies bringt die folgende, ebenfalls mit `divide` bezeichnete, ML-Rechenvorschrift, welche genau dann terminiert, wenn das erste Argument nichtnegativ und das zweite Argument positiv ist:

```
fun divide (m : int, n : int) : int =
  if (m>=0) andalso (n>0) then if m<n then 0
                                      else divide(m-n,n) + 1
                          else 1 div 0;
```

Die Semantik dieser Rechenvorschrift ist exakt die partielle Divisionsfunktion *div* wie oben eingeführt. ■

Zum Schluß dieses Abschnitts wollen wir uns nun noch mit dem Nachweis der Terminierung von rekursiven Rechenvorschriften beschäftigen, da dieser Begriff mitentscheidend ist beim Umgang mit Rekursion, wie sich in den letzten Beispielen gezeigt hat. Bisher hatten wir den Begriff „Terminierung“ nur informell benutzt. Präzisiert haben wir die folgende Festlegung der Terminierung von Rechenvorschriften:

3.4.15 Definition (Terminierung)

Gegeben sei eine ML-Rechenvorschrift der folgenden Form:

$$\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{,}\ldots\texttt{,}x_k\ \texttt{:}\ m_k\texttt{)}\ \texttt{:}\ n\ \texttt{=}\ t\texttt{;}$$

a) Der **Aufruf** $F(e_1, \ldots, e_k)$ **terminiert**, falls die bei der Termersetzungssemantik 3.4.2 konstruierte Folge $d_0, d_1, \ldots$ mit $F(e_1, \ldots, e_k)$ als dem ersten Folgenglied endlich und der Wert des letzten Terms d_N definiert ist.

b) Die **Rechenvorschrift terminiert unter der Vorbedingung** B, falls jeder ihrer Aufrufe $F(e_1, \ldots, e_k)$ mit definierten Grundtermen $e_i, 1 \leq i \leq k$, terminiert, sofern die Eigenschaft $B(e_1, \ldots, e_k)$ gilt.

c) Die **Rechenvorschrift terminiert**, falls sie unter der Vorbedingung *true* terminiert. ■

Beispielsweise terminiert die in Beispiel 3.3.2 angegebene Rechenvorschrift `fac` zur Berechnung der Fakultät einer natürlichen Zahl unter der Vorbedingung B, wobei $B(n)$ festgelegt ist als $n \geq 0$. Dies ist auch die allgemeinste Vorbedingung zur Terminierung, denn Aufrufe mit negativem Argument terminieren offensichtlich nicht.

Man beachte, daß in Definition 3.4.15 das eventuell existierende letzte Folgenglied ein Grundterm ist und man somit den Wert gemäß Definition 2.3.9 festlegen kann. Im allgemeinen ist das Feststellen von Terminierung algorithmisch nicht lösbar. Diese Tatsache besagt aber nur, daß es **keinen allgemeinen Algorithmus gibt, der alle Terminierungsprobleme löst**. In Spezialfällen kann man mittels spezieller Ansätze jedoch oft erfolgreich die Terminierung von (Aufrufen von) Rechenvorschriften zeigen. Wir geben nachfolgend ein Verfahren an.

3.4.16 Satz (Hinreichendes Kriterium für Terminierung)

Es sei F eine ML-Rechenvorschrift der Form von Definition 3.4.15. Diese Rechenvorschrift terminiert unter einer Vorbedingung, falls die folgenden Eigenschaften gelten:

a) Unter der Gültigkeit der Vorbedingung und der Bedingungen der Fallunterscheidungen[12] sind die Anwendungen der Operationen der elementaren Datenstrukturen in den jeweiligen Zweigen definiert.

b) Es gibt eine noethersch geordnete Menge $(M, \leq)$ und eine **Terminierungsfunktion** $\delta : \prod_{i=1}^{k} m_i^A \longrightarrow M$, so daß für jeden Aufruf von F im Rumpf t die Werte der Argumente bezüglich δ echt vermindert werden, falls die Vorbedingung und die Bedingungen der umgebenden Fallunterscheidungen gültig sind.

Beweis: Nach a) und b) ist die Folge $d_0, d_1, \ldots$ der Termersetzungssemantik 3.4.2 endlich, da jede echt absteigende Kette in noethersch geordneten Mengen $(M, \leq)$ endlich ist, und das letzte Folgenglied d_N ist auch definiert. ■

[12] Genaugenommen heißt dies, daß die Bedingung wahr ist, falls man sich im then-Zweig befindet, und deren Negation wahr ist, falls man sich im else-Zweig befindet.

Auf die in Satz 3.4.16 als a) aufgeführten Definiertheitsforderungen kann nicht verzichtet werden. Ein einfaches Gegenbeispiel ist der Aufruf `F(10)` der folgenden Rechenvorschrift:

```
fun F (x : int) : int =
  if x < 0 then x
           else if x = 0 then 10 div x
                         else F(x-1) + x;
```

In diesem Fall erhalten wir als die endliche Folge $d_0, d_1, \ldots$ der Termersetzung die nachstehende Folge von 12 Termen:

$$\begin{array}{ll} d_0 & : \texttt{F(10)} \\ d_1 & : \texttt{F(9) + 10} \\ d_2 & : \texttt{F(8) + 19} \\ \vdots & \quad\vdots \\ d_{10} & : \texttt{F(0) + 55} \\ d_{11} & : \texttt{(10 div 0) + 55}, \end{array}$$

Der Wert des letzten Terms ist wegen der Semantik der Basisoperation `div` offensichtlich undefiniert. Hier ist der Punkt a) des Satzes 3.4.16 verletzt, da aus der Gültigkeit der Bedingungen `x >= 0` und `x = 0` nicht die Definiertheit der Anwendung von `div` im inneren then-Zweig folgt.

Wir schließen dieses Kapitel über die Grundlagen der funktionalen Programmierung ab mit einigen Beispielen zur Terminierung von Rechenvorschriften.

3.4.17 Beispiele (für Terminierungsbeweise bzw. -probleme)

a) Wir beginnen mit einem **einfachen Terminierungsbeweis**. Dazu betrachten wir die folgende ML-Rechenvorschrift, die den ganzzahligen Anteil des Logarithmus zur Basis 2 berechnet:

```
fun blog (n : int) : int =
  if n = 1 then 0
           else 1 + blog(n div 2);
```

Wir wollen zeigen, daß `blog` unter der Vorbedingung B terminiert, falls man $B(n)$ als $n > 0$ definiert. Zuerst überprüfen wir Voraussetzung a) von Satz 3.4.16, die Anwendungen der Operationen: Die Addition ist total und die ganzzahlige Division durch 2 ist ebenfalls immer definiert.

Wir kommen nun zu Voraussetzung b) des Satzes 3.4.16. Wir betrachten die noethersche Ordnung $(\mathbb{N}, \leq)$ und die Terminierungsfunktion

$$\delta : \mathbb{Z} \longrightarrow \mathbb{N} \qquad \delta(n) = \begin{cases} n & : \quad n \geq 0 \\ 0 & : \quad \text{sonst.} \end{cases}$$

Es sei nun $n \in \mathbb{Z}$ mit $n > 0$ (Vorbedingung) und $n \neq 1$ (Bedingung der Alternative). Ist n gerade, so haben wir

$$\delta(\frac{n}{2}) = \frac{n}{2} < n = \delta(n),$$

und ist n ungerade, so haben wir

$$\delta(\frac{n-1}{2}) = \frac{n-1}{2} < n = \delta(n).$$

b) Nun betrachten wir einen **schwierigen Terminierungsbeweis**. Für das folgende setzen wir, der Einfachheit halber, in der Sprache ML eine Sorte `nat` für die natürlichen Zahlen $\mathbb{N}$ voraus, zusammen mit den entsprechenden Operationen und Konstanten. Um zu zeigen, daß die Rechenvorschrift

```
fun F (n : nat) : nat =
  if n mod 2 = 1 then F((3*n+1) div 2)
                 else n div 2;
```

(wobei der Aufruf von `mod` in der Bedingung von `F` testet, ob `n` ungerade ist, oder nicht) terminiert, haben wir wiederum die Voraussetzungen a) und b) von Satz 3.4.16 zu betrachten.

Vorausetzung a) ist trivial. Zu Voraussetzung b) von Satz 3.4.16 betrachten wir die noethersche Ordnung $(\mathbb{N}, \leq)$ und die Terminierungsfunktion

$$\delta : \mathbb{N} \longrightarrow \mathbb{N} \qquad \delta(n) = \begin{cases} 0 & : \ n \text{ gerade} \\ 1 + \delta(\frac{n-1}{2}) & : \ n \text{ ungerade.} \end{cases}$$

Offensichtlich terminiert die Rekursion von δ; die Funktion ist also tatsächlich total[13]. Eine echte Verminderung der Argumente bezüglich δ beim rekursiven Aufruf zeigt man wie nun folgt: Sei $n \in \mathbb{N}$ ungerade. Dann gilt die Beziehung

$$\delta(\frac{3 * n + 1}{2}) < \delta(n).$$

(Beachte: $(3 * n + 1)\ div\ 2 = \frac{3*n+1}{2}$, weil $3 * n + 1$ gerade ist.) Der Beweis erfolgt durch noethersche Induktion. Zum **Induktionsbeginn** gelte $n = 1$. Dies impliziert

$$\begin{array}{lll} \delta(\frac{3*1+1}{2}) = \delta(2) & & \text{Arithmetik} \\ = 0 & & \text{Definition } \delta \\ < 1 & & \\ = 1 + \delta(0) & & \text{Definition } \delta \\ = \delta(1) & & \text{Definition } \delta. \end{array}$$

Für den **Induktionsschluß** setzen wir die Ungleichung $n > 1$ voraus und betrachten dann zwei Fälle.

[13] Es bleibt natürlich immer noch die Frage, wie man so ein δ findet. Hier spielt Erfahrung und die Einsicht in den operationellen Ablauf der Rekursion eine große Rolle.

Ist $\frac{3*n+1}{2}$ gerade, so erhalten wir ohne Verwendung der Induktionshypothese die gewünschte Abschätzung wie folgt:

$$\begin{aligned}
\delta(\tfrac{3*n+1}{2}) &= 0 && \text{Definition } \delta \\
&< 1 \\
&\leq 1 + \delta(\tfrac{n-1}{2}) \\
&= \delta(n) && \text{Definition } \delta
\end{aligned}$$

Nun sei $\frac{3*n+1}{2}$ ungerade. Hier benötigen wir zum Beweis der gewünschten Abschätzung die Induktionshypothese:

$$\begin{aligned}
\delta(\tfrac{3*n+1}{2}) &= 1 + \delta(\tfrac{\frac{3*n+1}{2}-1}{2}) && \text{Definition } \delta \\
&= 1 + \delta(\tfrac{3*\frac{n-1}{2}+1}{2}) && \text{Arithmetik} \\
&< 1 + \delta(\tfrac{n-1}{2}) && \text{Induktionshypothese, } \tfrac{n-1}{2} < n \\
&= \delta(n) && \text{Definition } \delta
\end{aligned}$$

c) Als ein letztes Beispiel betrachten wir noch die nachfolgende ML-Rechenvorschrift, deren Rekursion auf L. Collatz zurückgeht. Dabei setzen wir, analog zu b), ebenfalls eine ML-Sorte `nat` und entsprechende Operationen und Konstanten voraus.

```
fun F (n : nat) : nat =
  if n <= 1 then 0
            else if n mod 2 = 1 then F(3*n+1)
                                else F(n div 2);
```

Es ist ein **ungelöstes Problem**, ob die Rechenvorschrift `F` für alle Argumente terminiert, obwohl man bisher mit Hilfe von Computern einen extrem großen Zahlbereich getestet hat und dabei immer Terminierung herrschte. ■

Die in Teil b) dieses Beispiels angegebene Rechenvorschrift stellt gewissermaßen einen Sonderfall dar. Normalerweise sind nämlich Terminierungsprobleme von zweigestaltiger Natur. Entweder sind sie relativ einfach lösbar, oder aber sie erfordern zur Lösung einen unheimlichen Aufwand (bzw. sind noch nicht gelöst wie die Collatz-Rekursion). Glücklicherweise sind fast alle in der praktischen Programmierung vorkommenden Probleme aus der ersten Kategorie.

Etwas anders ist die Situation bei den mit funktionaler Programmierung eng verwandten Termersetzungssystemen, wo man ja ebenfalls an Terminierung interessiert ist. Üblicherweise zeigt man hier Terminierung auch, indem man zuerst eine Terminierungsfunktion in eine noethersche Ordnung definiert und anschließend beweist, daß durch jede Regelanwendung der Wert dieser Funktion echt vermindert wird. Im Gegensatz zu Programmen liegt bei Termersetzungssystemen oft keine Interpretation der Sorten vor. Man hat also „nur syntaktisches Material zur Verfügung“ und dies zieht oft nach sich, daß man Terminierungsbeweise mit relativ komplexen notherschen Ordnungen zu führen hat.

3.5 Einschub: Zur Benutzung des ML-Systems

In diesem abschliessenden Abschnitt der Einführung in die funktionale Programmierung wird nun noch beschrieben, wie das schon erwähnte SML97-System (Standard ML of New Jersey, Version 1997) zur Ausführung von ML-Programmen in der praktischen Programmierung verwendet werden kann. Wir stellen dabei natürlich nicht alle Möglichkeiten des Systems vor, sondern nur diejenigen, welche in diesem Skriptum bis zum jetzigen Zeitpunkt gebraucht werden. Insbesondere gehen wir im weiteren nur auf den ML-Interpreter im Rahmen des SML97-Systems ein und nicht auch auf die Möglichkeiten der Übersetzung von ML-Programmen durch SML97. Mit den nachfolgend beschriebenen Möglichkeiten kann man im Prinzip (d.h. unter Berücksichtigung der entsprechenden Erweiterungen von ML) auch alle ML-Programme des nächsten Kapitels über die vertiefte funktionale Programmierung ausführen.

3.5.1 Aufruf des Systems

Das SML97-System an den Rechnern des Instituts für Informatik und Praktische Mathematik der Universität Kiel wird mittels des folgenden Befehls aufgerufen:

```
/home/smlnj/bin/sml
```

Um Aufrufe von SML97 einfacher zu gestalten, kann man unter dem von den Rechnern verwendeten Betriebssystem Sun Solaris (einer Abart des bekannten Unix) bei Verwendung der bash-Shell als „Befehlssprache“ des Betriebssystems ein sogenanntes „Alias“ einführen, beispielsweise wie in der folgenden Form:

```
alias sml97=/home/smlnj/bin/sml
```

Mit Hilfe dieses Solaris-Befehls wird `sml97` als Abkürzung für den Originalbefehl zum Aufruf des SML97-Systems definiert. Damit genügt es, `sml97` einzugeben, um das SML97-System zu starten.

Auf anderen Rechnern und/oder Betriebssystemen wird das SML97-System (oder ein anderes System zur Ausführung von ML-Programmen) in der Regel mittels eines anderslautenden Befehls gestartet. Auch das nachfolgend beschriebene Arbeiten mit ML ist natürlich systemabhängig und gilt in der beschriebenen Weise nur für SML97.

3.5.2 Das Arbeiten mit dem SML97-System

Ruft man das SML97-System auf, so erscheint der folgende Text auf dem Bildschirm:

```
Standard ML of New Jersey, Version 110.0.3, January 30, 1998
val use = fn : string -> unit
-
```

Durch das Minuszeichen „-“ zeigt das System an, daß es eine Benutzereingabe erwartet. Man nennt „-“ auch den Prompt von SML97.

Nun kann man beispielsweise mittels des `use`-Kommandos eine Datei mit einem ML-Programm in SML97 laden, so wie es in Abschnitt 3.3 schon beschrieben wurde. Wir gehen im folgenden, wie ebenfalls in diesem Abschnitt angenommen, davon aus, daß in einer Datei mit dem Namen `isperfect.ml` das aus den drei Rechenvorschriften `divides`, `isp` und `isperfect` bestehende ML-Programm zum Testen von perfekten Zahlen enthalten ist. Gibt man dann nach dem Prompt den Text

```
use "isperfect.ml";
```

ein, so antwortet das SML97-System wie nachfolgend angegeben:

```
[opening isperfect.ml]
val divides = fn int * int -> bool
val isp = fn int * int -> bool
val isperfect = fn int -> bool
val it = () : unit
-
```

Damit werden die drei geladenen Rechenvorschriften mit ihren Namen und jeweils auch den Sorten der Parameter und der Sorte des Resultats aufgezählt. Zusätzlich wird noch eine Konstante `it` als Abkürzung für das einzige Objekt `()` der Sorte `unit` definiert

Nun kann man nach dem Prompt einen geschlossenen Term eingeben, der aufgebaut sein darf aus den Konstanten und Operationen der elementaren ML-Datenstrukturen und den Rechenvorschriften, die eben geladen wurden. Ein Beispiel hierfür ist:

```
isperfect(6);
```

Das SML97-System antwortet auf diese Eingabe mit der folgenden Ausgabe:

```
val it = true : bool
-
```

SML97 berechnet also den Wert `true` des Terms `isperfect(6)` und gibt auch die entsprechende Sorte `bool` mit aus. Weiterhin wird die oben erwähnte Konstante `it` nun als Abkürzung für den berechneten Wert definiert. Nach dieser Auswertung kann man entweder weitere Terme zur Auswertung eingeben, oder mittels `use` weitere Programme aus Dateien laden, um die darin enthaltenen ML-Programme analog zum Term `isperfect(6)` auszuwerten. Es ist aber auch möglich, direkt nach dem Prompt von SML97 ML-Programme einzutippen statt sie aus Dateien in das System zu laden. Das Verlassen des SML97-Systems ist schließlich durch das gleichzeitige Drücken der beiden Tasten `Control` und `D` möglich. Um eine nicht terminierende Berechnung zu stoppen, hat man gleichzeitig die Tasten `Control` und `C` zu drücken.

Beim SML97-System muß jede Benutzereingabe mit einem Semikolon abgeschlossen werden. Wird dieses vergessen, d.h. etwa nur `isperfect(6)` statt `isperfect(6);` getippt, so antwortet das System wie folgt:

```
=
```

Dieses ausgegebene Gleichheitszeichen besagt, daß das SML97-System die Eingabe als noch nicht beendet ansieht. Der Benutzer kann daraufhin die noch fehlenden Zeichen, im Beispiel also das Semikolon, eingeben, um die Eingabe zu vervollständigen.

Ist ein ML-Programm syntaktisch nicht korrekt, so werden die Fehler durch das SML97-System zusammen mit der Angabe der entsprechenden Zeilen und Spalten aufgeführt. Vergißt man beispielsweise in der Rechenvorschrift `isp` das Gleichheitszeichen nach der Resultatsorte `bool`, so antwortet SML97 wie folgt:

```
isperfect.ml:6.3 Error: syntax error: inserting  EQUALOP
```

In so einem Fall hat man das Programm in der Datei entsprechend zu verändern und die Datei nochmals zu laden. Analoges gilt natürlich auch bei interaktiv eingetippten Programmen.

3.5.3 Vermeidung von Abkürzungen

Berechnet man mit SML97 den Wert eines Terms der Sorte `string`, der eine Zeichenreihe w von sehr großer Länge ist, so wird von w in der Standardeinstellung des Systems nur ein gewisses Anfangsstück auf dem Bildschirm gezeigt. Man kann aber die Ausgabelänge von Zeichenreihen bis zu einer fest vorgegebenen Obergrenze festsetzen, beispielsweise auf eine Million mittels der folgenden Eingabe:

```
Compiler.Control.Print.stringDepth := 1000000;
```

Analoges gilt auch für die im nächsten Kapitel behandelten Objekte, die durch Typen mit Konstruktoren eingeführt werden, und lineare Listen, welche ebenfalls im nächsten Kapitel behandelt werden. Auch hier wird in der Standardeinstellung von SML97 jeweils nur ein gewisser Teil der Ausgabe auf dem Bildschirm angezeigt. Die genaue Festlegung der Ausgabe auf beispielsweise die „Schachtelungstiefe“ 100 erfolgt im ersten Fall mittels

```
Compiler.Control.Print.printDepth := 100;
```

und eine ungekürzte Ausgabe von linearen Listen mit bis zu einer Million Elementen auf dem Bildschirm geschieht nach

```
Compiler.Control.Print.printLength := 1000000;
```

als Eingabe. Es empfiehlt sich, die eben genannten drei Befehle in einer Datei mit dem festen Namen `startup.ml` zusammenzufassen und diese jeweils mittels `use` in SML97 zu laden, wenn zu erwarten ist, daß die Standardeinstellung die zu erwartende Ausgabe abschneidet.

4 Vertiefung der funktionalen Programmierung

In Kapitel 3 wurde mittels der Programmiersprache ML eine einführende Übersicht über Prinzipien und Techniken des funktionalen Programmierens gegeben. Dabei wurde nur ein geringer Teil des Sprachumfangs von ML aufgezeigt, der sicherlich noch nicht geeignet ist zur Programmierung komplizierterer Algorithmen und größerer Programmsysteme. In diesem Kapitel führen wir einige weitere Sprachkonstrukte von ML ein – sowohl auf der Seite der Datenstrukturen als auch der Kontrollstrukturen. Wir verbinden diese Spracherweiterung von ML auch mit einer Vertiefung der Prinzipien und Techniken des funktionalen Programmierens.

4.1 Zusammengesetzte Datenstrukturen

Bisher kennen wir von ML nur die elementaren Datenstrukturen. Diese sind gegeben durch die Sorten `unit`, `bool`, `int`, `real`, `char`, `string`, und die entsprechenden Operationen (siehe Abschnitt 3.2), also durch die Signatur der Sprache. Der Aufbau von weiteren Datenstrukturen geschieht nun induktiv und orientiert sich gänzlich an den in der denotationellen Semantik üblichen Techniken zur schrittweisen Definition der dort erforderlichen speziellen Mengen. Diese Techniken sind Produkt- und Summenbildung, der Übergang zu Mengen von Funktionen und Rekursion.

4.1.1 Definition (Typen)

Es seien S die Menge der Sorten von ML und A die Algebra zu deren Interpretation, wie am Anfang von Abschnitt 3.2 erklärt. Dann sind die **Typen** von ML induktiv definiert durch die Menge S der Sorten als Basis und die beiden **Typ-Konstruktoren** `*` und `->`, d.h. durch die folgenden Regeln:

(1) Jede Sorte $s \in S$ ist ein Typ.

(2) Sind $m_1, \ldots, m_k$ Typen, so ist auch m_1 `*` $\ldots$ `*` m_k ein Typ, genannt **Produkttyp** mit den Komponententypen m_i, $1 \leq i \leq k$.

(3) Sind m und n Typen, so ist auch m `->` n ein Typ, genannt **Funktionstyp** mit Argumenttyp m und Resultattyp n.

Aufbauend auf die Interpretation s^A der Sorten $s \in S$ in der Algebra A ist die **Interpretation der ML-Typen** wie folgt induktiv definiert:

(4) $I[s] = s^A$ für $s \in S$

(5) $I[m_1 \texttt{ * } \ldots \texttt{ * } m_k] = \prod_{i=1}^{k} I[m_i]$

(6) $I[m \texttt{ -> } n] = [I[m] \to I[n]] = \{\ f\ :\ f$ partielle Funktion von $I[m]$ nach $I[n]\ \}$ ■

Beim Typaufbau sind in ML auch die Klammern `(` und `)` erlaubt. Um solche zu sparen, haben wir in den syntaktischen Festlegungen der ML-Typen unterstellt, daß die **Produkttypbildung stärker bindet als die Funktionstypbildung**. Will man eine andere Bindung beim Aufbau von Typen erreichen, so hat man zusätzlich zu klammern, wie etwa im Beispiel `int * (int -> int)`.

In Kapitel 3 hatten wir ein ML-Programm erklärt als eine Folge von Deklarationen, wobei jede Deklaration mit einem Semikolon als Abschlußzeichen endet und entweder aus einer einzelnen Rechenvorschrift oder aus einem System von Rechenvorschriften besteht. Man vergleiche noch einmal mit Abschnitt 3.3. Als Erweiterung dieser Syntax lassen wir nun auch noch die Deklaration von Typen in ML-Programmen zu. Dabei darf die Deklaration von Rechenvorschriften und von Typen in diesen erweiterten Programmen **in einer beliebigen Reihenfolge** erfolgen. Wichtig ist nur, daß eine Deklaration eines Begriffs vor seiner ersten Verwendung erfolgt und daß sich keine Namenskonflikte mit den Bezeichnern der ML-Signatur bzw. den schon deklarierten Rechenvorschriften und Typen ergeben. Auf diese „offensichtlichen“ Kontextbedingungen werden wir im weiteren zur Vereinfachung nicht mehr eingehen.

Es gibt in der Programmiersprache ML zwei Arten von Typdeklarationen. Wir beginnen mit der einfacheren der beiden Arten.

4.1.2 Typdeklarationen I: Typen als Abkürzungen

Es seien m ein Bezeichner (für einen Typ, kurz: ein **Typbezeichner**) und n ein Typ (der aus den Sorten und schon früher deklarierten Typen mittels der Typkonstruktoren `*` und `->` kompliziert aufgebaut sein kann[14]). Durch die Typdeklaration

$$\texttt{type}\ m\ \texttt{=}\ n\texttt{;} \qquad (T_A)$$

führt man den Bezeichner m **als Abkürzung** für den Typ n ein. Nach einer Deklaration der Gestalt (T_A) darf man im restlichen ML-Programm überall den Bezeichner m statt des eventuell komplizierteren Typausdrucks n schreiben. ■

Wichtiger als die eben erklärte erste Art von Typdeklarationen in der Sprache ML, die im Jargon wegen des verwendeten Schlüsselworts `type` auch **type-binding** genannt wird, ist die später noch folgende zweite Art. Im ML-Jargon nennt man diese, nach dem verwendeten Schlüsselwort `datatype`, auch **datatype-binding**. Vor ihrer formalen Definition ist noch der mathematische Begriff der disjunkten Vereinigung oder direkten Summe von Mengen notwendig, dem wir uns nun kurz zuwenden.

Wir starten der Einfachheit halber mit dem Fall $n = 2$, also mit zwei Mengen M und N. Im Gegensatz zur üblichen Vereinigung $M \cup N$ von M und N werden bei ihrer disjunkten Vereinigung $M \uplus N$ die beiden Mengen so vereinigt, daß bei einem eventuellen nichtleeren Schnitt die in beiden Mengen enthaltenen Teile nicht identifiziert werden. Dies kann man mathematisch dadurch erreichen, daß man jedes Element der disjunkten Vereinigung mit

[14] In diesem Zusammenhang spricht man dann auch von „Typausdrücken“ statt von Typen.

einem sogenannten **Diskriminator** versieht, der angibt, aus welcher der Originalmengen es stammt. In Hinblick auf eine spätere Erweiterung auf die n-fache disjunkte Vereinigung verwendet man bei zwei Mengen die Zahlen 1 und 2 als Diskriminatoren. Dies führt zur folgenden Definition:

$$M \uplus N := (M \times \{1\}) \cup (N \times \{2\}) = \{\ \langle x, 1\rangle\ :\ x \in M\ \} \cup \{\ \langle x, 2\rangle\ :\ x \in N\ \}$$

Aus dieser Festlegung ergeben sich auch sofort durch die Definitionen

$$\imath_1(x) = \langle x, 1\rangle \qquad\qquad \imath_2(x) = \langle x, 2\rangle$$

zwei injektive Funktionen $\imath_1 : M \longrightarrow M \uplus N$ und $\imath_2 : N \longrightarrow M \uplus N$ zur Einbettung von M bzw. N in ihre disjunkte Vereinigung.

4.1.3 Definition (Disjunkte Mengenvereinigung)

Als Verallgemeinerung des eben beschriebenen binären Falls definiert man nun zu $n \in \mathbb{N}$ beliebig und Mengen $M_1, \ldots, M_n$ die n-fache **disjunkte Vereinigung** (oft auch n-fache **direkte Summe** genannt) als Menge mittels

$$\biguplus_{i=1}^{n} M_i := \bigcup_{i=1}^{n} (M_i \times \{\, i \,\})$$

und, für k mit $1 \leq k \leq n$, die k-te **kanonische Injektion** $\imath_k : M_k \longrightarrow \biguplus_{i=1}^{n} M_i$ durch

$$\imath_k(x) = \langle x, k\rangle.$$ ■

Für $n = 1$ ist $\biguplus_{i=1}^{n} M_i$ isomorph zu M_1 und wird in der Regel mit dieser Menge identifiziert. Im Falle $n = 0$ hat man $\biguplus_{i=1}^{n} M_i = \emptyset$. Haben die Mengen $M_1, \ldots, M_n$ paarweise einen leeren Durchschnitt (sind also paarweise disjunkt), so ist ihre disjunkte Vereinigung $\biguplus_{i=1}^{n} M_i$ isomorph zur gewöhnlichen Vereinigung $\bigcup_{i=1}^{n} M_i$ und man identifiziert deshalb auch hier beide Mengenbildungen.

Nach diesen Vorbereitungen kehren wir nun zum eigentlichen Thema, dem **datatype-binding** der Programmiersprache ML, zurück.

4.1.4 Typdeklarationen II: Typen mit Konstruktoren

In der Programmiersprache ML hat eine **Typdeklaration mit Konstruktoren** syntaktisch die nachfolgende Form:

$$\texttt{datatype}\ m\ \texttt{=}\ con_1\ \text{of}\ n_1\ \texttt{|}\ \ldots\ \texttt{|}\ con_k\ \text{of}\ n_k\,\texttt{;} \qquad (T_K)$$

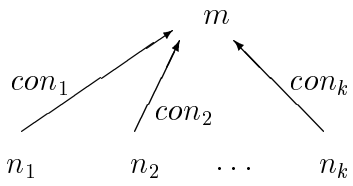
Dabei ist k eine positive natürliche Zahl. Weiterhin sind m und con_i, $1 \leq i \leq k$, neue Bezeichner (für einen Typ bzw. für Operationen) und die n_i, $1 \leq i \leq k$, sind Typen (wiederum aufgebaut aus den Sorten und schon früher deklarierten Typen mittels der

Typkonstruktoren * und ->). Durch eine Deklaration der Gestalt (T_K) wird ein neuer Typ m definiert, dessen Interpretation $I[m]$ die disjunkte Vereinigung $\biguplus_{i=1}^{k} I[n_i]$ der Interpretationen der Typen $I[n_i]$, $1 \leq i \leq k$, ist. Weiterhin werden für alle i mit $1 \leq i \leq k$ Operationen (mit nachgestellter Funktionalität)

$$con_i \ : \ n_i \ \texttt{->} \ m$$

eingeführt, die sogenannten **Konstruktoren**. Die Semantik eines jeden Konstruktors con_i, $1 \leq i \leq k$, ist festgelegt als die kanonische Injektion von $I[n_i]$ in die disjunkte Vereinigung $I[m]$. ■

Bildlich hat man bei der Deklaration mit Konstruktoren der Gestalt (T_K) die nachfolgende Situation vorliegen, wobei die Pfeile die Funktionalitäten der Konstruktoren anzeigen:



Nach diesen abstrakten und vielleicht auch etwas ungewohnten Begriffen behandeln wir nun mit ihrer Hilfe ein konkretes Beispiel.

4.1.5 Beispiel (Geradendarstellungen)

Eine von der Schule her bekannte Art, eine Gerade g als Teilmenge der Euklidschen Ebene $\mathbb{R} \times \mathbb{R}$ darzustellen, ist die **Normalform**

$$g = \{ \ \langle x, y \rangle \in \mathbb{R} \times \mathbb{R} \ : \ a * x + b * y + c = 0 \ \}.$$

Damit ist eine Gerade als Teilmenge von $\mathbb{R} \times \mathbb{R}$ durch drei reelle Zahlen $a, b, c \in \mathbb{R}$ bestimmt. In der Programmiersprache ML liest sich dies als Deklaration mit einem Konstruktor wie folgt:

```
datatype normform = mnf of real * real * real;
```

Hierdurch führt man `normform` als Typ für Geraden in Normalform ein, sowie eine Operation `mnf`, die, auf drei reelle Zahlen angewendet, ein Objekt des Typs `normform` (also eine Gerade in Normalform) erzeugt.

Eine weitere, auch von der Schule her bekannte Art, Geraden als Teilmengen von $\mathbb{R} \times \mathbb{R}$ darzustellen, ist die **Vektorform**

$$g = \{ \ \boldsymbol{x} + \lambda * \boldsymbol{y} \ : \ \lambda \in \mathbb{R} \ \}$$

mit einem sogenannten **Ortsvektor** $\boldsymbol{x} = \langle x_1, x_2 \rangle \in \mathbb{R} \times \mathbb{R}$ und einem sogenannten **Richtungsvektor** $\boldsymbol{y} = \langle y_1, y_2 \rangle \in \mathbb{R} \times \mathbb{R}$. Bei dieser Darstellung ist die Gerade g durch das Paar

$\boldsymbol{x}$ und $\boldsymbol{y}$ von Vektoren bestimmt. In ML kann man dies beispielsweise wie folgt realisieren:

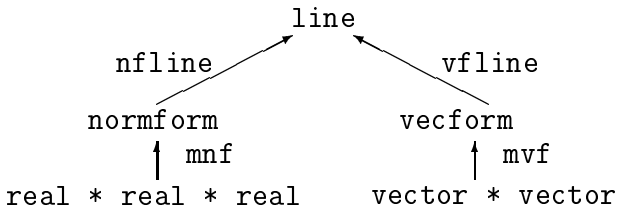
```
type vector      = real * real;
datatype vecform = mvf of vector * vector;
```

Diese Deklarationen führen folgendes ein: einen Typ `vector` als Abkürzung für den Produkttyp `real*real`, einen Typ `vecform` für die Geraden in Vektorform und eine Operation `mvf`, die, auf zwei Vektoren angewendet, ein Objekt des Typs `vecform` (also eine Gerade in Vektorform) erzeugt.

Will man nun eine Gerade sowohl in der Normal- als auch in der Vektorform darstellen, so hat man beide Definitionsvarianten „disjunkt“ zu vereinigen. Dies führt in der Programmiersprache ML zur nachfolgenden Deklaration:

```
datatype line = nfline of normform |
                vfline of vecform;
```

Damit erhält man einen neuen ML-Typ `line` und zwei neue Konstruktoren `nfline` („fasse die Normalform als Gerade auf“) und `vfline` („fasse die Vektorform als Gerade auf“). Insgesamt stehen uns also nach den obigen Typdeklarationen die Typen und Konstruktoren des folgenden Diagramms zur Verfügung:



Nun betrachten wir eine konkrete Gerade, nämlich die identische Relation auf $\mathbb{R}$. Es sei also $g = \{\ \langle x, x\rangle : x \in \mathbb{R}\ \}$. Wir implementieren nachfolgend g mit Hilfe der eben erklärten Darstellungsarten in ML. Bei einer Verwendung von $\langle 1, -1\rangle$ als Normalenvektor bekommt man als Normalform

$$g = \{\ \langle x, y\rangle \in \mathbb{R} \times \mathbb{R}\ :\ 1 * x + (-1) * y + 0 = 0\ \}.$$

Dem entspricht in der Programmiersprache ML der Term `mnf(1.0,~1.0,0.0)` des Typs `normform`. Um g als ML-Objekt des Typs `line` zu erhalten, ist noch eine Anwendung des Konstruktors `nfline` erforderlich. Dies sieht dann wie folgt aus:

```
nfline(mnf(1.0,~1.0,0.0))
```

Nun betrachten wir g in der Vektorform-Darstellung. Wählen wir $\langle 0, 0\rangle$ als Ortsvektor und $\langle 1, 1\rangle$ als Richtungsvektor, so erhalten wir

$$g = \{\ \langle 0, 0\rangle + \lambda * \langle 1, 1\rangle\ :\ \lambda \in \mathbb{R}\ \}.$$

In ML wird diese Vektorform durch den Term `mvf((0.0,0.0),(1.0,1.0))` des Typs `vecform` dargestellt. Hierbei sind sowohl `(0.0,0.0)` als auch `(1.0,1.0)` Terme des Typs

`vector`, also des Typs `real*real`. Wie schon im Falle der Normalform, so liefert auch hier eine weitere Anwendung des entsprechenden Konstruktors `vfline`, d.h. der Term

```
vfline(mvf((0.0,0.0),(1.0,1.0))),
```

die Vektorform-Gerade g als ML-Objekt des Typs `line`. ■

Dieses Beispiel zeigt auch, daß durch die Typen und deren Konstruktoren die Möglichkeiten erweitert werden, in der Sprache ML Terme aufzubauen. Die Auswirkungen auf die Rechenvorschriften fassen wir nachfolgend zusammen.

4.1.6 Bemerkung (Erweiterung der Rechenvorschriften)

In Abschnitt 3.3 waren in einer Rechenvorschrift, die auch Teil eines Systems sein konnte, in der Funktionalität bisher nur Parameter $x : m$ mit einer Sorte m zugelassen worden. Auch die Art des Resultats war durch eine Sorte bestimmt. Aufgrund der bisherigen Erweiterung der Sprache um Typen und Typdeklarationen machen wir uns nun von dieser Einschränkung frei und legen fest, daß **Parameter einer Rechenvorschrift als auch das Resultat einen beliebigen Typ haben dürfen**.

Diese Verallgemeinerung der Funktionalität zieht nach sich, daß beim Aufbau des Rumpfes einer Rechenvorschrift auch die bisher (durch die Punkte 3.3.1.a bis 3.3.1.e) gegebenen Konstruktionsmöglichkeiten zu erweitern sind. Wir erlauben im folgenden als zusätzliche Konstruktionen

f) Anwendungen von **Konstruktoren** von Typdeklarationen und von **Parametern**, sofern sie von einem Funktionstyp sind, und

g) **Tupelbildung** $(t_1, \ldots, t_k)$ von Termen. ■

Für die eben angegebenen syntaktischen Erweiterungen sind natürlich auch die in Abschnitt 3.4 definierten Semantiken geeignet zu erweitern. Im Falle von Tupeln ist dies nicht schwierig. Die Semantik eines Tupels $(t_1, \ldots, t_k)$ ist genau dann definiert, wenn die Semantiken aller Komponenten t_i, $1 \leq i \leq k$, definiert sind, und stimmt in diesem Falle mit dem Tupel der Komponentensemantiken überein. Es ist jedoch nicht ganz einfach, die Call-by-value-Semantik für funktionale Konstruktionen anzugeben. Bezüglich dieses Punktes müssen wir auf eine Vorlesung über Semantik nach dem Vordiplom verweisen.

4.2 Rekursive Datenstrukturen und Typen

Neben der Produkt-, Funktionen- und Summenbildung läßt die Sprache ML noch Rekursion als Prinzip zur Definition von Datenstrukturen zu. Viele interessante Datenstrukturen der Informatik sind durch Induktion und/oder Rekursion beschrieben. Wir geben nachfolgend einige Beispiele an.

4.2.1 Beispiele (für rekursive Datenstrukturen)

a) Es sei A ein Zeichenvorrat. Dann kann man die Menge A^* der **Zeichenreihen** über A darstellen als

$$\begin{array}{rcll} A^* & = & \{\varepsilon\} \cup \{\ w \in A^* \ :\ w \neq \varepsilon\ \} & \\ & = & \{\varepsilon\} \cup \{\ a \,\&\, w \ :\ a \in A, w \in A^*\ \} & \text{Erzeugungsprinzip} \\ & = & \{\varepsilon\} \cup \{\ \langle a, w\rangle \ :\ a \in A, w \in A^*\ \} & \text{Definition von } \& \\ & = & \{\varepsilon\} \cup A \times A^*. & \end{array}$$

Da die Mengen $\{\varepsilon\}$ und $A \times A^*$ disjunkt sind (eine Zeichenreihe ist entweder leer oder nichtleer), bekommen wir für die Menge A^* die Rekursion

$$A^* = \{\varepsilon\} \uplus A \times A^*,$$

indem wir die gewöhnliche Vereinigung und die disjunkte Vereinigung (wie im letzten Abschnitt bemerkt) identifizieren[15].

b) Nun sei A eine nichtleere Menge (von Markierungen). Dann ist die Menge $\mathsf{Btree}(A)$ der **Binärbäume mit Knotenmarkierungen** aus A durch die nachstehenden zwei Eigenschaften induktiv definiert:

(1) Für jedes $a \in A$ gilt $a \in \mathsf{Btree}(A)$.

(2) Gilt $l, r \in \mathsf{Btree}(A)$ und $a \in A$, so folgt daraus $\langle l, a, r\rangle \in \mathsf{Btree}(A)$.

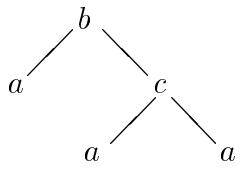
Ein Baum der ersten Form heißt atomarer Baum. Hingegen nennt man einen Baum $\langle l, a, r\rangle$ der zweiten Form zusammengesetzt mit Wurzelmarkierung a, linkem Teilbaum l und rechtem Teilbaum r. Die obige induktive Definition der Menge $\mathsf{Btree}(A)$ läßt sich mit Hilfe einer disjunkten Vereinigung sofort in die Rekursion

$$\mathsf{Btree}(A) = A \uplus \mathsf{Btree}(A) \times A \times \mathsf{Btree}(A)$$

übertragen, wobei wir wiederum die gewöhnliche Vereinigung und die disjunkte Vereinigung identifizieren.

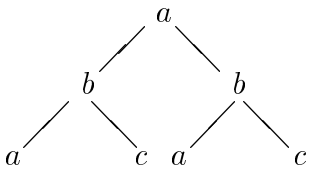
Durch die obige mathematische Definition sind die Elemente aus $\mathsf{Btree}(A)$ Klammerschachtelungen über Elementen von A. Zeichnerisch stellt man diese als Diagramme mit Linien (**Kanten**) dar, welche die Markierungen (**Knoten**) verbinden. Einige Beispiele sind nachfolgend angegeben, wobei $A = \{\ \bullet, \mathrm{a,b,c,d}\ \}$ als Menge von Markierungen unterstellt ist:

$\langle a,b,\langle a,c,a\rangle\rangle$

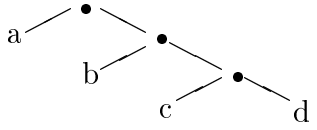


[15] Genaugenommen haben wir schon bei der Herrechnung der Rekursion von A^* eine Identifizierung vorgenommen, da wir das $n+1$-Tupel $\langle a, a_1, \ldots, a_n\rangle$ im dritten Schritt gleichsetzten mit dem Paar $\langle a, \langle a_1, \ldots, a_n\rangle\rangle$, dessen zweite Komponente ein n-Tupel ist.

$\langle\langle a,b,c\rangle, a, \langle a,b,c\rangle\rangle$



$\langle$a,•, $\langle$b,•, $\langle$c, • ,d$\rangle\rangle\rangle$



Aus diesen graphischen Darstellungen leitet sich auch die Bezeichnung „Binärbaum“ ab. Sich weiter an botanischen Begriffen orientierend, nennt man den obersten Knoten jeweils die **Wurzel** und die untersten Knoten, die nur eine Kante berühren, die **Blätter**[16].

c) Es sei, analog zu b), wiederum eine nichtleere Menge A (von Markierungen) vorausgesetzt. Dann ist die Menge $\mathsf{Lisp}(A)$ der **Binärbäume mit Blattmarkierungen** aus A induktiv wie folgt definiert:

(1) Für jedes $a \in A$ gilt $a \in \mathsf{Lisp}(A)$.

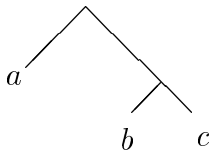
(2) Gilt $l, r \in \mathsf{Lisp}(A)$, so auch $\langle l, r\rangle \in \mathsf{Btree}(A)$.

Diese Definition führt sofort zur der folgenden Rekursion:

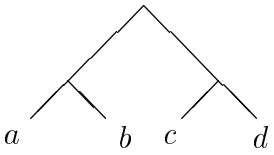
$$\mathsf{Lisp}(A) = A \uplus \mathsf{Lisp}(A) \times \mathsf{Lisp}(A)$$

Den Unterschied zu den knotenmarkierten Binärbäumen $\mathsf{Btree}(A)$ sieht man am besten anhand einiger graphischer Darstellungen von blattmarkierten Binärbäumen, wobei wir als Markierungsmenge $A = \{\ a, b, c, d, e\ \}$ voraussetzen:

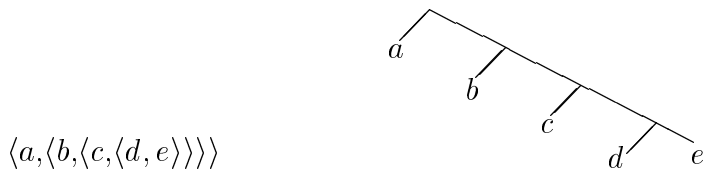
$\langle a, \langle b, c\rangle\rangle$



$\langle\langle a, b\rangle, \langle c, d\rangle\rangle$



[16] Im Gegensatz zur Natur ist es in der Informatik üblich, die Wurzel eines Baums oben und seine Blätter unten zu zeichnen. Dazu bemerkte einmal zweideutig der Münchner Professor Klaus Samelson, einer der Begründer der deutschen Informatik: In der Informatik wachsen die Bäume nicht in den Himmel.



Im Vergleich zu den Binärbäumen mit Knotenmarkierungen sind die Verzweigungsknoten („inneren Knoten“) nun nicht mehr markiert, sondern nur noch die Blätter. Daher rührt auch der Name dieser Datenstruktur. ■

Bis zum jetzigen Stand des Skriptums werden Typen aufgebaut mit Hilfe von Sorten, Produkttypbildung und Funktionstypbildung. Bei den zwei möglichen Arten der Deklaration von Typen hatten wir weiterhin stillschweigend vorausgesetzt, daß der zu deklarierende Typ auf der rechten Seite der Deklaration nicht vorkommt. Diese Kontextbedingung wird in ML aber nur für Typdeklarationen als Abkürzungen verlangt.

Im Falle einer Typdeklaration mit Konstruktoren der Form (T_K) kann der Bezeichner m auch rechts vorkommen, was zu einer bedeutenden Unterart der zweiten Art von Typdeklarationen führt, die wir nun besprechen wollen.

4.2.2 Typdeklarationen III: Rekursive Typen

Kommt in einer Typdeklaration mit Konstruktoren der Form

$$\texttt{datatype}\ m\ \texttt{=}\ con_1\ \text{of}\ n_1\ \texttt{|}\ \ldots\ \texttt{|}\ con_k\ \text{of}\ n_k\,\texttt{;}$$

der Typbezeichner m auch beim Aufbau der Typen n_i, $1 \leq i \leq k$, der rechten Seite vor, so heißt m **rekursiver Typ** oder **rekursiv definierter Typ**. ■

Zur Bestimmung der Interpretation $I[m]$ eines rekursiv definierten Typs m geht man in einem ersten Schritt von der ML-Deklaration zu einer semantischen Gleichung

$$X = \mathcal{E}(X) \qquad (*)$$

über. Hierbei ist X die Unbekannte, eine Menge, und $\mathcal{E}(X)$ ein sich aus der rechten Seite der Typdeklaration ergebender Mengenausdruck, in dem X vorkommt. Geleitet durch die induktive Definition 1.2.5 von Mengen, die kleinste Mengen spezifiziert, beweist man dann, daß die Gleichung $(*)$ eine kleinste Lösung D als Grenzwert der Kette

$$\emptyset \subseteq \mathcal{E}(\emptyset) \subseteq \mathcal{E}(\mathcal{E}(\emptyset)) \subseteq \ldots$$

besitzt. In einem dritten Schritt definiert man schließlich die Interpretation $I[m]$ als D und, darauf aufbauend, die Semantik der Konstruktoren con_i, $1 \leq i \leq k$, als die entsprechenden kanonischen Injektionen von $I[n_i]$ nach D.

Das eben skizzierte Vorgehen ist verblüffend ähnlich dem der denotationellen Semantik von Rechenvorschriften. Es ist natürlich noch genau festzulegen, wie man von der Typdeklaration zur Gleichung $(*)$ auf Mengen kommt und auch zu zeigen, daß die angegebene Kette tatsächlich den kleinsten Fixpunkt der Mengenfunktion $\mathcal{E}$ bezüglich der Inklusion „berechnet“. Beim ersten Punkt verwendet man eine ähnlichen Technik wie in Definition 3.4.6 bei der Festlegung des von einer Rechenvorschrift induzierten Funktionals. Für den zweiten Punkt bedarf es eines Satzes ähnlich zum Fixpunktsatz 3.4.9. Es würde den Rahmen dieses Skriptums sprengen, wenn wir auf diese beiden Punkte in voller mathematischer Strenge eingehen würden. Im Detail wird dies alles in einer entsprechenden Vorlesung des Hauptstudium gezeigt; wir begnügen uns mit der Demonstration der Vorgehensweise anhand der Beispiele 4.2.1.

Im Vergleich zur Originalsyntax von Typdeklarationen machen wir in den folgenden Beispielen von einer erlaubten **syntaktischen Vereinfachung** von ML Gebrauch und schreiben in der Typdeklaration nur den Konstruktor *con* statt *con* `of unit`. Wendet man diese Vereinfachung in einer Deklaration an, so ist natürlich im restlichen Programm auch überall *con* statt *con*`()` zu schreiben, da nun *con* für ein Element und nicht für eine kanonische Injektion steht.

4.2.3 Beispiele (Weiterführung von 4.2.1)

Im folgenden sei m ein Typ, dessen Interpretation jeweils die Menge A aus den Beispielen 4.2.1 sei. Dann übertragen sich die drei Rekursionsgleichungen unter der Einführung von Konstruktoren wie folgt in rekursive ML-Typen.

a) Wir beginnen mit den **Zeichenreihen**. Eine auf Beispiel 4.2.1.a aufbauende rekursive Definition von Zeichenreihen von Objekten des Typs m in der Programmiersprache ML, in diesem Zusammenhang wegen der vordefinierten ML-Sorte `string` besser **lineare Listen** genannt, ist:

```
datatype sequ = empty |
                append of m * sequ;
```

Nach dieser Typdeklaration hat man auf den linearen Listen (mit Angabe des jeweiligen Typs bei der Konstante und dem Konstruktor) zur Verfügung:

Typ	:	`sequ`
Konstante	:	`empty : sequ`
Konstruktor	:	`append :` m `* sequ -> sequ`

In Erweiterung des Begriffs von Definition 2.3.1, bezeichnet man dies auch als die Signatur der linearen Listen.

Es sei nun die Menge A die Interpretation des Typs m und die einelementige Menge $E = \{\langle\rangle\}$ die Interpretation der vordefinierten Sorte `unit`. Dann ist die zur Deklaration der Listen gehörende Gleichung

$$X = E \uplus A \times X.$$

Wir iterieren nun, beginnend mit der leeren Menge $\emptyset$, die durch ihre rechte Seite beschriebene Mengenfunktion $\mathcal{E}(X) = E \uplus A \times X$ und erhalten die Kette

$$\emptyset \subseteq E \subseteq E \cup A \subseteq E \cup A \cup A^2 \subseteq E \cup A \cup A^2 \cup A^3 \subseteq \ldots,$$

wenn wir isomorphe Mengen als gleich auffassen. Unter Nichtbeachtung der Klammerung in den Tupeln ist der Grenzwert dieser Kette, also die kleinste Lösung obiger Mengengleichung, die Menge $A^* = \bigcup_{n\in\mathbb{N}} A^n$. Als eine Konsequenz erhalten wir die Semantik des Konstruktors `append` als diejenige Funktion von $A \times A^*$ nach A^*, die ein Element a und eine Liste w durch das Anfügen von links zur Liste $\langle a, w\rangle$ zusammenfaßt.

b) Nun betrachten wir die **knotenmarkierten Binärbäume**, wobei m der Typ für die Knotenmarkierungen sei. In der Sprache ML schreibt sich dann die induktive Definition bzw. die Rekursionsgleichung aus Beispiel 4.2.1.b wie folgt:

```
datatype btree = atom of m |
                 make of btree * m * btree;
```

Damit bekommt man als Signatur der knotenmarkierten Binärbäume:

```
Typ           : btree
Konstruktoren : atom : m -> btree
                make : btree * m * btree -> btree
```

Die zu dieser Typdeklaration gehörende Mengengleichung ist

$$X = A \uplus X \times A \times X,$$

mit der Menge A als Interpretation des Typs m. Ihre kleinste Lösung bestimmt sich zu $\mathsf{Btree}(A)$[17]. Die Semantik des Konstruktors `atom` ist somit die identische Einbettung von A in $\mathsf{Btree}(A)$ und als Semantik von `make` erhalten wir die Funktion von $\mathsf{Btree}(A) \times A \times \mathsf{Btree}(A)$ nach $\mathsf{Btree}(A)$, die l, a und r zu einem Baum $\langle l, a, r\rangle$ zusammensetzt.

c) Es sei m wiederum ein ML-Typ für Markierungen. Bei den **blattmarkierten Binärbäumen** sieht die Übertragung der induktiven Definition bzw. Rekursion aus Beispiel 4.2.1.c in ML wie folgt aus:

```
datatype lisp = atom of m |
                cons of lisp * lisp;
```

Durch diese Typdeklaration werden für das Restprogramm als Signatur der blattmarkierten Binärbäume zur Verfügung gestellt:

[17] Um dies einzusehen, berechnet man am einfachsten wiederum einige Glieder der den Fixpunkt approximierenden Kette. Wegen der Klammerstruktur der Bäume muß man dabei aber auch alle zweifachen direkten Produkte vollständig klammern, d.h. darf $A \times (A \times A)$ und $(A \times A) \times A$ nicht identifizieren.

Typ	:	`lisp`
Konstruktoren	:	`atom` : m `->` `lisp`
		`cons` : `lisp * lisp -> lisp`

Als Mengengleichung zu dieser Typdeklaration bekommt man

$$X = A \uplus X \times X.$$

Ist A die Interpretation von m, so hat die Gleichung die Menge $\mathsf{Lisp}(A)$ als kleinste Lösung und die Semantik der zwei Konstruktoren `atom` und `cons` ist offensichtlich gegeben durch die identische Einbettung von A in $\mathsf{Lisp}(A)$ bzw. die Funktion von $\mathsf{Lisp}(A) \times \mathsf{Lisp}(A)$ nach $\mathsf{Lisp}(A)$, die l und r zum Baum $\langle l, r \rangle$ zusammenfaßt. ■

Für eine spezielle Art von Rekursion, nämlich die der linearen Listen, gibt es so viele Anwendungen, daß sie in ML typmäßig vordefiniert ist. Im folgenden gehen wir genauer auf die Datenstruktur der linearen Listen in ML ein.

4.2.4 Die Datenstruktur der linearen Listen

Gegeben sei in ML ein Typ m. Dann ist auch m `list` ein ML-Typ und seine Interpretation ist die Menge der **linearen Listen** (Zeichenreihen, Folgen) mit Elementen aus der Interpretation von m. Beispielsweise beschreibt somit der Typ `int list` die Menge der linearen Listen von ganzen Zahlen.

Nachfolgend ist ein Ausschnitt der Signatur der linearen Listen angegeben:

Konstante:	`nil` : m `list`	Leere Liste
Operationen:	`::` : m `*` m `list` `->` m `list`	Anfügen von links
	`hd` : m `list` `->` m	Erstes Listenelement
	`tl` : m `list` `->` m `list`	Liste ohne erstes Listenelement
	`null` : m `list` `->` `bool`	Test auf Leersein
	`length` : m `list` `->` `int`	Länge einer Liste
	`@` : m `list` `*` m `list` `->` m `list`	Listenkonkatenation
	`rev` : m `list` `->` m `list`	Listenrevertierung

Alle Anwendungen der einstelligen Operationen dieser Signatur werden in der funktionalen Schreibweise notiert, alle Anwendungen der binären Operationen in der infix-Schreibweise. Mit Ausnahme von `hd` und `tl` sind alle Operationen total. Eine Anwendung dieser zwei Operationen ist nur dann definiert, wenn das Argument nicht `nil` ist.

Zum Aufbau einer Liste ist die Schreibweise mittels `nil` und der Anfügeoperation `::` etwas umständlich. Es gibt deshalb eine sogenannte Standarddarstellung. Anstelle von

$$a_k \texttt{ :: } a_{k-1} \texttt{ :: } \ldots \texttt{ :: } a_1 \texttt{ :: nil}$$

schreibt man abkürzend

$$\texttt{[}a_k\texttt{,}a_{k-1}\texttt{,}\ldots\texttt{,}a_1\texttt{]}.$$

Insbesondere hat man für $k = 0$ die Darstellung `[]` für die leere lineare Liste `nil`. ■

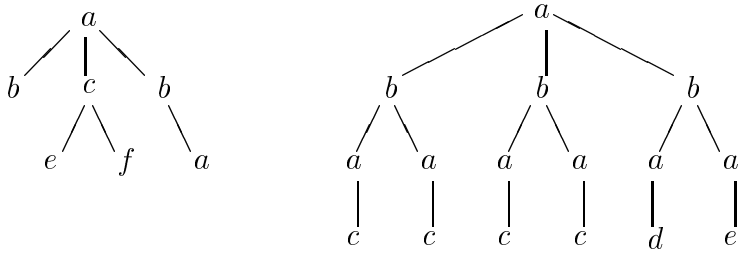
In der obigen Signatur sind nicht alle Operationen auf linearen Listen in ML aufgeführt. Weitere Operationen findet man im Manual zum SML97-System, das üblicherweise mit dem System ausgeliefert wird.

Durch obige Festlegung sind lineare Listen von Objekten beliebiger Sorte in unser bisheriges Typsystem eingeführt, und man kann nun mit den bisher vorgestellten Mitteln weiterkonstruieren. Dies führt etwa zu Listen von Listen, Bäumen von Listen, Listen von Bäumen usw., also zu einem sehr allgemeinen Konzept von Typen und Datenstrukturen. Wir betrachten dazu einen wichtigen Einzelfall.

4.2.5 Beispiel (Allgemeine Bäume)

In Beispiel 4.2.1 haben wir Binärbäume behandelt, bei denen jeder Knoten entweder zwei Nachfolger hat, oder keinen. Diese Bäume sind Spezialfälle von sogenannten **n-fach vergabelten Bäumen**, bei denen jeder Verzweigungsknoten genau n Nachfolger hat. Noch weniger einschränkend ist ein Baumbegriff, bei denen die Nachfolgerzahl der Verzweigungsknoten beliebig ist.

Zwei graphische Beispiele für solche allgemeine knotenmarkierte Bäume mit Markierungen aus der Menge der ersten sechs Kleinbuchstaben des Alphabets sind nachstehend angegeben; man vergleiche auch mit der Erklärung einiger Begriffe zur Termersetzung in Abschnitt 2.3, wo wir auch schon solche Bilder verwendeten:



Nun sei A eine nichtleere Menge von Markierungen. Dann kann man die Menge $\mathsf{Tree}(A)$ der **allgemeinen** oder **beliebig-vergabelten Bäume mit Knotenmarkierungen** aus A induktiv wie folgt definieren:

(1) Es ist $a \in \mathsf{Tree}(A)$ für alle $a \in A$.

(2) Aus $a \in A$ und $w \in \mathsf{Tree}(A)^*$ folgt $\langle a, w\rangle \in \mathsf{Tree}(A)$.

Einem Knoten wird also die Liste seiner Nachfolger zugeordnet. Verwenden wir für Listen die ML-Notation mit eckigen Klammern [und], um sie von der Paarbildung mittels der spitzen Klammern $\langle$ und $\rangle$ zu unterscheiden, so erhalten wir in dieser formalen Klammerdarstellung für das Beispiel oben links:

$$\langle a, [b, \langle c, [e, f]\rangle, \langle b, [a]\rangle]\rangle$$

Das Beispiel oben rechts sieht in der Klammerschreibweise wie folgt aus:

$$\begin{array}{l}\langle a, [\ \langle b, [\langle a, [c]\rangle, \langle a, [c]\rangle]\ \rangle, \\ \quad\ \langle b, [\langle a, [c]\rangle, \langle a, [c]\rangle]\ \rangle, \\ \quad\ \langle b, [\langle \mathrm{a}, [d]\rangle, \langle a, [e]\rangle]\ \rangle\ ]\rangle\end{array}$$

In der Programmiersprache ML schreiben sich die beiden Forderungen (1) und (2) als Typdeklaration mit zwei Konstruktoren `atom` und `succs` wie folgt:

```
datatype tree = atom of m |
                succs of m * tree list;
```

■

Bei den Rechenvorschriften lernten wir verschränkte Rekursion kennen. Dies führte zu den Systemen von Rechenvorschriften. Verschränkte Rekursion ist auch auf der Ebene der Datenstrukturen möglich und führt hier ebenfalls zu Systemen.

4.2.6 Typdeklarationen IV: Systeme von rekursiven Typen

Wir betrachten noch einmal die Datenstruktur der beliebig-vergabelten Bäume. Will man diese in ML ohne Verwendung der vorimplementierten Listen bewerkstelligen, d.h. mit der rekursiven Listendeklaration wie in Beispiel 4.2.3.a mit dem Typ `tree` an Stelle von m, so führt dies zu einer verschränkt-rekursiven Situation, denn die Deklaration von `tree` verwendet den Typ `sequ` und in der rekursiven Deklaration dieses Typs `sequ` kommt aber, neben `sequ`, auch noch der Typ `tree` der Bäume auf der rechten Seite vor.

Wie bei den Rechenvorschriften, so sind in ML zur Behebung dieser Situation deshalb auch verschränkt-rekursive Typdeklarationen möglich, indem man zu Systemen übergeht und dies wiederum durch das Schlüsselwort „`and`“ kennzeichnet. Insbesondere ist durch

```
datatype tree     = atom of m |
                    succs of m * treesequ
     and treesequ = empty |
                    append of tree * treesequ;
```

eine verschränkt-rekursive Definition beliebig-vergabelter Bäume mit Knotenmarkierungen des Typs m gegeben. ■

4.3 Verwendung von funktionalen Termen

Ein Term der Sprache ML heißt ein **funktionaler Term**, falls sein Typ ein Funktionstyp ist. Insbesondere haben wir, daß die Operationen der elementaren ML-Datenstrukturen und die Namen von Rechenvorschriften funktionale Terme sind. Ihr Typ ergibt sich hierbei jeweils direkt aus der Funktionalität, was wir als Schreibweise auch schon in den Signaturen der Beispiele 4.2.3 verwendet haben. Die bedeutendste Art funktionaler Terme sind

jedoch die sogenannten Funktionsabstraktionen, denen später in diesem Abschnitt unser Hauptinteresse gelten wird. Erst die Verwendung von funktionalen Termen ist „eigentlich“ funktionales Programmieren. Im folgenden betrachten wir drei Möglichkeiten. Zur Verbesserung der Lesbarkeit verwenden wir in ihnen wiederum $[A \to B]$ als Notation für die Menge der (ggf. partiellen) Funktionen von A nach B.

4.3.1 Erste Verwendung: Funktionale Parameter

Wir haben mittlerweile in den Funktionalitäten von Rechenvorschriften auch Parameter $f : m$ `->` n zugelassen, d.h. Parameter eines Funktionstyps oder **funktionale Parameter**. Besitzt eine Rechenvorschrift so einen Parameter, so sind beim Aufbau des Rumpfes auch Aufrufe von f erlaubt. ■

Durch die Verwendung von funktionalen Parametern kann man viele algorithmische Probleme elegant und methodisch sauber programmieren. Sie erlauben auch, Programme oder Programmstücke so zu formulieren, daß sie vielfach **wiederverwendet** werden können. Wir geben nachfolgend zwei bekannte Beispiele auf den lineare Listen an.

4.3.2 Beispiele (Listenoperationen mit Funktionsparametern)

Es bezeichne A^* die Menge aller linearen Listen mit Elementen aus A, d.h. die Menge der n-Tupel $\langle a_1, \ldots, a_n \rangle$ mit $n \geq 0$ und $a_i \in A$ für alle i mit $1 \leq i \leq n$. Auf A^* haben wir die in Abschnitt 2.1 für Zeichenreihen eingeführten Operationen $\&$, *top*, *rest* usw. Mit ε bezeichnen wir die leere Liste $\langle\rangle \in A^*$.

a) Als ein erstes Beispiel für einen funktionalen Parameter betrachten wir eine Funktion

$$map : [A \to B] \times A^* \longrightarrow B^*,$$

die eine gegebene Funktion $f : A \longrightarrow B$ auf jedes Element einer gegebenen Liste $w \in A^*$ anwendet. Aufbauend auf die mathematische Definition von Listen als n-Tupel kann man *map* wie folgt definieren:

$$map(f, \langle a_1, \ldots, a_n \rangle) = \langle f(a_1), \ldots, f(a_n) \rangle$$

Offensichtlich gilt für die leere Liste ε, nach der üblichen Konvention bei der Tupelschreibweise, die Gleichung

$$map(f, \varepsilon) = \varepsilon. \qquad (M_1)$$

Nun sei $w \in A^*$ mit $w = \langle a_1, \ldots, a_n \rangle \neq \varepsilon$, d.h. $n \geq 1$. Dann erhalten wir mittels der schon mehrfach verwendeten Technik

$$\begin{array}{rcll} map(f, w) & = & \langle f(a_1), \ldots, f(a_n) \rangle & \text{Definition von } map \\ & = & f(a_1) \,\&\, \langle f(a_2), \ldots, f(a_n) \rangle & \text{Definition von } \& \\ & = & f(a_1) \,\&\, map(f, \langle a_2, \ldots, a_n \rangle) & \text{Definition von } map \\ & = & f(top(w)) \,\&\, map(f, rest(w)) & \text{Definition von } top, rest, \end{array}$$

also die nachfolgende rekursive Gleichung:

$$map(f, w) = f(top(w)) \,\&\, map(f, rest(w)) \qquad (M_2)$$

Die eben hergeleitete Rekursion für die Funktion *map* terminiert offensichtlich. Man erhält also durch eine Übertragung nach ML die Rechenvorschrift

```
fun map (f : m->n, w : m list) : n list =
  if null(w) then nil
             else f(hd(w)) :: map(f,tl(w));
```

als die programmiersprachliche Fassung von (M_1) und (M_2).

b) Als zweites Beispiel betrachten wir eine Funktion

$$\mathit{filter} : [A \to \mathbb{B}] \times A^* \longrightarrow A^*,$$

so daß $\mathit{filter}(p, w)$, unter Beibehaltung der Reihenfolge, aus der Liste $w \in A^*$ genau die Elemente $a \in A$ mit $p(a) = tt$ herausfiltert. Im Gegensatz zur Funktion *map* können wir *filter* formal nicht mehr explizit durch eine Gleichung $\mathit{filter}(p, w) = \ldots$ definieren. Stattdessen bietet sich eine Induktion nach dem Aufbau von w an.

Für die Basis, die leere Liste, legen wir fest:

$$\mathit{filter}(p, \varepsilon) = \varepsilon \qquad (F_1)$$

Bei der Anwendung des Konstruktors $\&$, dem Anfügen von a an eine Liste w von links, haben wir zwei Fälle zu betrachten. Gilt $p(a) = tt$, so definieren wir

$$\mathit{filter}(p, a \,\&\, w) = a \,\&\, \mathit{filter}(p, w). \qquad (F_2)$$

Im verbleibenden Falle $p(a) = \mathit{ff}$ legen wir fest, daß

$$\mathit{filter}(p, a \,\&\, w) = \mathit{filter}(p, w) \qquad (F_3)$$

zutrifft. Die induktive Definition der Funktion *filter* mittels der Gleichungen (F_1) bis (F_3) und der dazugehörigen Fallunterscheidungen überträgt sich unmittelbar in die Programmiersprache ML und wir bekommen

```
fun filter (p : m->bool, w : m list) : m list =
  if null(w) then nil
             else if p(hd(w)) then hd(w) :: filter(p,tl(w))
                              else filter(p,tl(w));
```

als die der Funktion *filter* entsprechende Rechenvorschrift.

Sowohl die Rechenvorschrift `map` als auch die Rechenvorschrift `filter` besitzen mit `f` bzw. `p` einen funktionalen Parameter. ■

Als nächste Verwendung funktionaler Terme behandeln wir die Funktionsabstraktion, die es ermöglicht, durch die Bindung von frei vorkommenden Bezeichern („Variablen“) in Termen zu funktionalen Termen zu gelangen. Die Funktionsabstraktion ist die bedeutendste Möglichkeit, in der Sprache ML mit funktionalen Objekten zu arbeiten.

Zur Motivation dieses Sprachkonstrukts betrachten wir den Funktionsbegriff in der Mathematik. Hier werden (partielle) Funktionen üblicherweise definiert durch die Notation (auch „explizite“ Definition genannt)

$$f : M \longrightarrow N \qquad\qquad f(x) = \ldots$$

und damit ist es notwendig, ihnen einen Namen zu geben. Ein konkretes einfaches Beispiel ist die folgende Funktion:

$$square : \mathbb{R} \longrightarrow \mathbb{R} \qquad\qquad square(n) = n * n$$

Will man nun eine (partielle) Funktion **anonym** definieren, d.h. ohne ihr einen Namen zu geben, so muß man sich – beim derzeitigen Stand des Skriptums – auf ihre ursprüngliche Definition als eindeutige und totale (oder nur eindeutige) Relation rückbesinnen. Anonym schreibt sich *square* in der originalen Auffassung als Menge

$$\{\ \langle n, n * n\rangle\ :\ n \in \mathbb{R}\ \}.$$

Diese mengentheoretische Darstellung anonymer Funktionen ist nicht sehr lesbar, deshalb wurden in der Mathematik und auch in der Informatik günstigere Schreibweisen eingeführt, wie beispielsweise die folgenden:

$$(n\ :\ \mathbb{R})\ :\ \mathbb{R}\ \bullet\ n * n \qquad\qquad (\ n \mapsto n * n \mid n \in \mathbb{R}\ )$$

Auch in der Sprache ML ist es möglich, Funktionsobjekte anonym durch eine spezielle Termkonstruktion, nämlich genau die Funktionsabstraktion, einzuführen.

4.3.3 Zweite Verwendung: Funktionsabstraktion

Es seien $x_1, \ldots, x_k$ paarweise verschiedene Bezeichner, $m_1, \ldots, m_k$ deren Typen und t ein Term des Typs n. Dann ist ist auch

$$\texttt{fn (}x_1\ \texttt{:}\ m_1\texttt{, }\ldots\texttt{, }x_k\ \texttt{:}\ m_k\texttt{) => }t \qquad (FA)$$

ein Term, genannt **Funktionsabstraktion**, und sein Typ ist m_1 `*` $\ldots$ `*` m_k `->` n. Sich an die Sprechweise bei den Rechenvorschriften anpassend, nennt man die x_i, $1 \leq i \leq k$, auch die **Parameter** und t den **Rumpf** von (FA). ■

Durch die Hinzunahme der Funktionsabstraktion zum Termaufbau ergibt sich eine **erneute Erweiterung des Begriffs einer Rechenvorschrift** in ML.

Nach der Syntax der Funktionsabstraktion wollen wir nun auch etwas auf ihre Semantik eingehen. Im einfachsten Falle, daß im Term t nur Aufrufe von Operationen der elementaren Datenstrukturen vorkommen, ist die Semantik von (FA) eine (partielle) Funktion

$$f : \prod_{j=1}^{k} I[m_j] \longrightarrow I[n],$$

die einem k-Tupel $\langle u_1, \ldots, u_k \rangle \in \prod_{j=1}^{k} I[m_j]$ den Wert von t unter einer Belegung σ mit $\sigma(x_i) = u_i$ im Sinne von Definition 2.3.9 zuordnet. Z.B. ist dann die Semantik von

```
fn (m : real, n : real) => m * n
```

die Funktion $f : \mathbb{R} \times \mathbb{R} \longrightarrow \mathbb{R}$, welche sich, mit A als der Algebra zur Signatur der Programmiersprache ML, zu

$$f(u_1, u_2) = I[\texttt{m * n}]^A_\sigma = I[\texttt{m}]^A_\sigma * I[\texttt{n}]^A_\sigma = \sigma(\texttt{m}) * \sigma(\texttt{n}) = u_1 * u_2$$

berechnet. In dieser Gleichungskette folgt die letzte Gleichung aus den Festlegungen $\sigma(\texttt{m}) = u_1$ und $\sigma(\texttt{n}) = u_2$.

Unkritisch ist auch der Fall, wenn in t ein Aufruf einer Rechenvorschrift F vorkommt, dieser jedoch nicht rekursiv ist. Hier wird bei der Auswertung F wie ein Funktionssymbol aus der ML-Signatur behandelt und durch seine Semantik $\mu(\tau_F)$ interpretiert. Ist etwa `fac` die Fakultäts-Rechenvorschrift aus Beispiel 3.3.2.b, so erhalten wir als Semantik von

```
fn (m : int, n : int) => fac(m) + n
```

die Funktion $f : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{Z}$ mit (wegen der partiellen Funktion $fac_{\mathbb{Z}}$ vergleiche man wiederum mit Beispiel 3.3.2.b)

$$f(u_1, u_2) = I[\texttt{fac(m) + n}]^A_\sigma = fac_{\mathbb{Z}}(I[\texttt{m}]^A_\sigma) + I[\texttt{n}]^A_\sigma = fac_{\mathbb{Z}}(\sigma(\texttt{m})) + \sigma(\texttt{n}) = fac_{\mathbb{Z}}(u_1) + u_2.$$

Kritisch wird es, wenn die Funktionsabstraktion (FA) im Rumpf einer rekursiven Rechenvorschrift F vorkommt und t einen rekursiven Aufruf von F enthält. Eine denotationelle Semantik für diesen Fall anzugeben, erfordert eine Erweiterung von Definition 3.4.6, was den Rahmen dieses Skriptums aber überschreitet.

4.3.4 Bemerkung (β-Konversion)

Eine sehr gute operationelle Vorstellung von der Funktionsabstraktion bekommt man durch die Gleichung

$$\texttt{(fn (}x_1 \texttt{ : } m_1\texttt{,} \ldots \texttt{,}x_k \texttt{ : } m_k\texttt{) => } t\texttt{)(}e_1\texttt{,} \ldots \texttt{,}e_k\texttt{)} = t^{e_1 \ldots e_k}_{x_1 \ldots x_k}, \qquad (\beta)$$

in welcher der Term $t^{e_1 \ldots e_k}_{x_1 \ldots x_k}$ der rechten Seite aus dem Rumpf t der Funktionsabstraktion dadurch entsteht, indem man für alle i, $1 \leq i \leq k$, simultan den Bezeichner x_i textuell jeweils durch den Term e_i ersetzt. Beispielsweise haben wir

```
(fn (m : real, n : real) => m * n)(3.0,4.0) = 3.0 * 4.0 = 12.0.
```

In der Literatur wird (β) die **β-Konversion** genannt. Für eine Sprache mit Call-by-name-Semantik gilt (β) immer. Ist hingegen, wie bei ML, Call-by-value als Parameterübergabemechanismus vorgeschrieben, so ist (β) nur dann gültig, falls die Werte der Argumente $e_1, \ldots, e_k$ der Funktionsabstraktion definiert sind. ■

Nachfolgend geben wir ein Beispiel für die Verwendung der bisher vorgestellten funktionalen Konzepte der Programmiersprache ML an. Wir entwickeln schrittweise ein ML-Programm, das einen bekannten Primzahl-Test realisiert.

4.3.5 Beispiel (Primzahltest mittels der Siebmethode)

Nach Eratosthenes bekommt man die Liste der ersten N Primzahlen aus der Liste der Zahlen von 2 bis N wie folgt: Man markiert die Zahl 2 und streicht in der Liste alle echten Vielfachen von 2. In dem nachfolgenden Bild ist die Markierung durch ein Kästchen um die Zahl angezeigt und die Streichung durch einen Unterstrich.

$$\boxed{2}\ 3\ \underline{4}\ 5\ \underline{6}\ 7\ \underline{8}\ 9\ \underline{10}\ 11\ \ldots$$

Dann markiert man in der neuen Liste die kleinste ungestrichene und unmarkierte Zahl, also die Zahl 3, und streicht deren echten Vielfachen. In unserer graphischen Darstellung sieht dies wie folgt aus:

$$\boxed{2}\ \boxed{3}\ \underline{4}\ 5\ \underline{6}\ 7\ \underline{8}\ \underline{9}\ \underline{10}\ 11\ \ldots$$

Diesen letzten Schritt wiederholt man immer wieder. Das Verfahren terminiert schließlich, wenn alle Zahlen der Liste entweder markiert oder gestrichen sind. Am Schluß sind genau die Primzahlen der Liste markiert.

Für eine Programmierung der Siebmethode bietet es sich an, die Liste, in der gestrichen werden soll, wie folgt in ML darzustellen; zu einer Liste `s` wird die Zahl `n` genau dann als ungestrichen betrachtet, wenn `s(n)` gleich `true` ist.

```
type sequ = int -> bool;
```

Aufbauend auf diese Darstellung kann man nämlich das Streichen einer Zahl `n` in einer Liste `s` sehr einfach durch eine Rechenvorschrift mit einer Funktionsabstraktion realisieren:

```
fun crossout (s : sequ, n : int) : sequ =
  fn (k : int) => if k = n then false
                           else s(k);
```

In gewohnter mathematischer Notation würde man die Rechenvorschrift `crossout` als Funktion *crossout* wie folgt punktweise definieren:

$$crossout(s,n)(k) = \begin{cases} \mathit{ff} & : \ k = n \\ s(k) & : \ k \neq n \end{cases}$$

Mittels `crossout` ist es nun einfach, eine Rechenvorschrift `cross` anzugeben, die in der Liste `s` alle Zahlen der Form `i`, `i+p`, `i+2*p`, `i+3*p` usw. zwischen `i` und `n` streicht:

```
fun cross (s : sequ, i : int, n : int, p : int) : sequ =
  if i > n then s
           else cross(crossout(s,i),i+p,n,p);
```

Die folgende Rechenvorschrift `next` sucht nun in der Liste `s` zu einer Zahl `n` die nächste nichtgestrichene Zahl, die größer als oder gleich `n` ist. Ausgehend von der gegebenen Zahl `n` erfolgt die Suche dabei schrittweise von unten.

```
fun next (s : sequ, n : int) : int =
   if s(n) then n
           else next(s,n+1);
```

Mit Hilfe der beiden letzten Rechenvorschriften `next` und `cross` kann nun eine simple Rechenvorschrift realisiert werden, die den Siebalgorithmus zur gegebenen Liste `s` zwischen zwei Zahlen `i` und `n` ausführt:

```
fun sieve (s : sequ, i : int, n : int) : sequ =
  if i > n then s
           else (fn (t : sequ) => sieve(t,next(t,i+1),n))
                (cross(s,i+i,n,i))
```

In dieser Rechenvorschrift wird die Funktionsabstraktion im else-Fall dazu verwendet, in Verbindung mit der Call-by-value-Semantik von ML eine zweifache Berechnung des Aufrufs `cross(s,i+i,n,i)` zu verhindern, da natürlich der Siebalgorithmus in der neuen Liste die nächste nichtgestrichene Zahl sucht. Wir haben an dieser Stelle bewußt auf eine Vereinfachung der Schreibweise mittels einer Objektdeklaration verzichtet, da dieses Sprachkonstrukt erst in Abschnitt 4.5 behandelt wird.

Der endgültige Siebalgorithmus streicht nun zwischen `2` und der gegebenen Zahl `N`, da kleinere Zahlen als `2` per Definition keine Primzahlen sind, und beginnt mit einer Liste, bei der noch keine Zahl gestrichen ist. Wegen der speziellen Darstellung der Listen, läßt sich letztere mittels einer speziellen Funktionsabstraktion beschreiben und wir erhalten:

```
fun eratosthenes (N : int) : sequ =
  sieve(fn (k : int) => true,2,N);
```

Damit liefert der Aufruf `eratosthenes(N)(k)` für zwei Eingaben `N` und `k`, die größer als `1` sind und `k <= N` erfüllen, genau dann `true`, wenn `k` eine Primzahl ist. ■

Wir kommen nun zu einer dritten Möglichkeit, in ML funktionale Objekte zu verwenden. Wie bei der Funktionsabstraktion, werfen wir auch hier zuerst einen Blick auf die Mathematik. In ihr ist es üblich, Funktionen mit Funktionen als Resultat punktweise zu definieren. Wir haben das schon in Abschnitt 3.4 bei den Funktionalen in Form von $\tau_F[f](v)$ ausgiebig verwendet. Auch die Funktion *crossout* des letzten Beispiels wurde

so erklärt. Ein weiteres Beispiel, bei der die punktweise Definition sogar in einer noch höheren Stufe auftritt, ist die Funktion

$$curry : [A \times B \to C] \longrightarrow [A \to [B \to C]],$$

die einer zweistelligen Funktion $f : A \times B \longrightarrow C$ durch die punktweise Festlegung

$$curry(f)(a)(b) = f(a, b) \qquad (*)$$

eine einstellige Funktion $curry(f) : A \longrightarrow [B \to C]$ zuordnet. Man beachte insbesondere, daß $curry(f)(a)$ für alle $a \in A$ eine Funktion von B nach C ist, die für ein Element $b \in B$ durch die Gleichung $(*)$ definiert ist.

Nach diesen Vorbemerkungen betrachten wir nun die punktweise Definition von Rechenvorschriften in ML, die eine **nochmalige Erweiterung** dieses Begriffs darstellt. Bei den Typen führt dies zu Funktionstypen, deren Resultattypen wiederum Funktionstypen sind. Um Klammern zu sparen, nimmt deshalb das ML-System bei Funktionstypen eine **Rechtsklammerung** an, d.h. der Typ m `->` n `->` p steht als Abkürzung für den vollständig geklammerten Typ m `->` `(`n `->` p`)`.

4.3.6 Dritte Verwendung: Punktweise Definition einer Rechenvorschrift

Eine punktweise Definition einer Rechenvorschrift in ML hat, mit paarweise verschiedenen Parametern x_i, deren Typen m_i $(1 \leq i \leq k)$ und einem Term t, die folgende Form:

$$\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{)(}x_2\ \texttt{:}\ m_2\texttt{)}\ldots\texttt{(}x_k\ \texttt{:}\ m_k\texttt{)}\ \texttt{=}\ t\texttt{;} \qquad (PD)$$

Ist n der Typ des Terms t, so ist m_1 `->` m_2 `->` $\ldots$ `->` m_k `->` n der Typ dieser Rechenvorschrift. ■

Punktweise Definitionen kann man als syntaktische Varianten von Funktionsabstraktionen auffassen, denn die Semantik von (PD) is per Definition gleich der Semantik der nachfolgenden Rechenvorschrift:

$$\begin{array}{l}\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{)}\ \texttt{:}\ m_2\ \texttt{->}\ \ldots\ \texttt{->}\ m_k\ \texttt{->}\ n\ \texttt{=} \\ \quad \texttt{fn (}x_2\ \texttt{:}\ m_2\texttt{)}\ \texttt{=>}\ \texttt{(fn}\ \ldots\ \texttt{=>}\ \texttt{(fn (}x_k\ \texttt{:}\ m_k\texttt{)}\ \texttt{=>}\ t\texttt{)}\ldots\texttt{);}\end{array}$$

Statt nur einzelner Parameter `(`x_i `:` m_i`)` sind in (PD) auch Listen von Parametern `(`x_{i_1} `:` m_{i_1}`,`$\ldots$`,`x_{i_s} `:` m_{i_s}`)` erlaubt; der Typ einer solchen Rechenvorschrift ergibt sich dann aus dem Typ m_1 `->` m_2 `->` $\ldots$ `->` m_k `->` n, indem man m_i durch den entsprechenden Produkttyp m_{i_1} `*` $\ldots$ `*` m_{i_s} ersetzt. Es ist offensichtlich, wie die zu einer solchen punktweise definierten Rechenvorschrift semantisch gleichwertige Rechenvorschrift mit einer Funktionsabstraktion als Rumpf aussieht.

Man beachte, daß im Gegensatz zu den punktweisen Fassungen in den Fassungen mit Funktionsabstraktionen auch die Resultattypen angegeben sind, d.h. die volle Funktionalität aus der Deklaration ersichtlich ist.

Wir betrachten die oben angegebene Funktion *curry* noch einmal. Sind m, n und p die ML-Typen für die Mengen A, B und C, so schreibt sich die punktweise Definition von *curry* wie folgt als ML-Rechenvorschrift:

```
fun curry (f : m*n->p)(x : m)(y : n) = f(x,y);
```

Unter Verwendung von zwei geschachtelten Funktionsabstraktionen statt der punktweisen Definition bekommt man offensichtlich sofort

```
fun curry (f : m*n->p) : m->n->p =
  fn (x : m) => (fn (y : n) => f(x,y));
```

als Version dieser Rechenvorschrift.

4.4 Polymorphie

Wie wir bisher gesehen haben, ist in der Programmiersprache ML jedem Term ein Typ zugeordnet, insbesondere auch den Operationen der elementaren Datenstrukturen und den Rechenvorschriften. Gleichzeitig ist es in der Sprache aber auch möglich, als Erweiterung der bisher vorgestellten Sprachkonstruktionen, etwa eine Rechenvorschrift F zu deklarieren, deren Argumente $t_1, \ldots, t_k$ beim Aufruf `F(`t_1`,...,`t_k`)` mehrere Typen annehmen dürfen. Solche Rechenvorschriften nennt man polymorph. Polymorphie tritt in Programmiersprachen, so auch in unserer Vorlesungssprache ML, im wesentlichen in zweierlei Gestalten auf, die wir nun kurz beschreiben wollen.

4.4.1 Ad-hoc Polymorphie in ML

Ad-hoc Polymorphie, die erste Variante von Polymorphie, ist die **Überladung von Bezeichnungen**. Hier steht derselbe Name oder dasselbe Symbol für eine Reihe von Operationen und/oder Rechenvorschriften unterschiedlicher Funktionalität, von denen bei der Auswertung eines Terms, je nach Kontext, die zutreffende ausgesucht wird.

In ML tritt Ad-hoc Polymorphie nur bei den Operationen der elementaren Datenstrukturen auf. Beispielsweise werden sowohl die Addition auf den ganzen Zahlen als auch die Addition auf den reellen Zahlen mit dem Symbol `+` bezeichnet. Liegt nun der Term `3+5` vor, so ist aus den Darstellungen `3` und `5` der Argumente zu entnehmen, daß mit `+` die Basisoperation des Typs `int*int->int` gemeint ist. Hingegen wird bei der Auswertung von `3.0+5.0`, wegen der durch die Argumente `3.0` und `5.0` gegebene Kontextinformation, die Basisoperation `+` des Typs `real*real->real` genommen. ■

Ad-hoc Polymorphie tritt eigentlich in allen gängigen Programmiersprachen auf. Die zweite Art, parametrische Polymorphie, wird hingegen in der Regel nur von funktionalen Programmiersprachen zur Verfügung gestellt. Wir beginnen eine Erklärung dieses Begriffs am besten mit einem Beispiel.

4.4.2 Beispiel (für eine parametrisch polymorphe Rechenvorschrift)

In der ML-Rechenvorschrift `map` von Beispiel 4.3.2.a sind m und n gegebene ML-Typen. Man sieht sofort ein, daß der konkrete Typ jeweils gar nicht wesentlich ist; für m und n sind eigentlich beliebige ML-Typen zulassen[18]. Dies kann man in ML dadurch ausdrücken, daß man in der Funktionalität der Rechenvorschrift `'m` und `'n` statt m und n schreibt, also `map` wie folgt deklariert:

```
fun map (f : 'm->'n, w : 'm list) : 'n list =
  if null(w) then nil
             else f(hd(w)) :: map(f,tl(w));
```

Damit werden `'m` und `'n` zu **Typvariablen**, für die bei der konkreten Anwendung jeder Typ eingesetzt werden kann. So ist es wegen der Verwendung von Typvariablen in obiger Rechenvorschrift `map` möglich, den Term

```
map(fn (n : int) => n*n, [1,2,3])
```

hinzuschreiben, dessen Wert sich zur linearen Liste (mit Typangabe)

```
[1,4,9] : int list
```

ergibt. Es ist aber beispielsweise auch ein Aufruf von `map` mit einer Operation auf Zeichenreihen und einer Liste von Zeichenreihen als Argumenten erlaubt. So liefert der Aufruf

```
map(size, ["ab","abc","a"]),
```

mit der Basisoperation `size` zur Berechnung der Länge einer Zeichenreihe, die lineare Liste (mit Typangabe)

```
[2,3,1] : int list.
```

Eine Rechenvorschrift wie `map`, die gleichzeitig auf eine ganze Klasse von Objekten wirken kann, nennt man (parametrisch) polymorph. ■

In der Programmiersprache ML hat man zwei Ausprägungen von Polymorphie, die wir nun genauer beschreiben wollen.

4.4.3 Parametrische Polymorphie in ML

a) Die Sprache ML erlaubt **parametrische Typen** bzw. Datenstrukturen: Bei der Typdeklaration mit Konstruktoren sind, als Erweiterung ihrer Festlegung 4.1.4, auch Typvariable `'a`, `'m` usw. erlaubt. Syntaktisch werden diese, wie schon in Beispiel 4.4.2.a bei der Rechenvorschrift `map` verwendet, dem eigentlichen Typ vorangestellt.

[18] Wir haben bisher versucht, in ML-Programmen beliebig wählbare Teile dadurch hervorzuheben, daß wir eine kursive Schrift statt Schreibmaschinenschrift verwendeten.

Damit erhalten wir beispielsweise einen parametrischen Typ für die linearen Listen des Typs `'a` durch die folgende Deklaration:

```
datatype 'a sequ = empty |
                   append of 'a * 'a sequ;
```

Offensichtlich handelt es sich hier um eine Verallgemeinerung von Beispiel 4.2.3.a.

b) In ML ist es erlaubt, **polymorphe Rechenvorschriften** zu deklarieren: Dies ist einerseits möglich, indem man, wie in obiger Rechenvorschrift `map`, als Erweiterung des bisherigen Begriffs, bei den Deklarationen von Rechenvorschriften in den Funktionalitäten auch Typvariablen `'a`, `'m` usw. oder parametrische Typen verwendet. Damit ist die Anwendung für jede Ersetzung der jeweiligen Typvariablen durch explizite Typen definiert.

Polymorphie wird in ML auch dadurch erreicht, indem man in den Funktionalitäten Typangaben wegläßt. In diesem Falle berechnet der Typisierungsalgorithmus von ML, in der Regel unter Verwendung von Typvariablen `'a`, `'m` usw., den allgemeinsten Typ, so daß Aufrufe ohne Typfehler ausgewertet werden können. Ein einfaches Beispiel ist die Rechenvorschrift

```
fun id (x) = x;
```

zur Realisierung der identischen Funktion auf einer beliebigen Typinterpretation. Als allgemeinsten Typ für `id` liefert das System `'a->'a`. ■

Neben den Typvariablen `'a`, `'m` usw., die für einen beliebigen ML-Typ stehen, gibt es noch die **Typvariablen mit Gleichheit** `''a`, `''m` usw., die für einen ML-Typ mit zugehöriger Gleichheits- und Ungleichheitsoperation, also für eine Sorte der elementaren Datenstrukturen ungleich `real`, stehen. Eine Anwendung ist die nachfolgende Rechenvorschrift:

```
fun eqlist (s : ''m list, t : ''m list) : bool =
  if null(s) then null(t)
             else hd(s) = hd(t) andalso eqlist(tl(s),tl(t));
```

Durch sie ist es möglich, die Gleichheit von linearen Listen von Objekten der Sorten der elementaren Datenstrukturen von ML zu testen.

4.5 Objektdeklarationen und Abschnitte

Bis zum jetzigen Aufbau des Skriptums besteht ein ML-Programm aus einer Folge von Deklarationen, wobei deklariert werden können (i) einzelne Rechenvorschriften oder Systeme von Rechenvorschriften, (ii) Typen als Abkürzungen und (iii) Typen mit Konstruktoren oder Systeme von Typen mit Konstruktoren. Als vierte Deklarationsart führen wir jetzt noch die Objektdeklarationen ein.

4.5.1 Objektdeklarationen

In der Programmiersprache ML sieht eine **Objektdeklaration** in ihrer allgemeinsten Form syntaktisch wie folgt aus:

$$\texttt{val (}x_1\ \texttt{:}\ m_1\texttt{,}\ldots\texttt{,}x_k\ \texttt{:}\ m_k\texttt{) = }t\texttt{;} \qquad (OD)$$

Dabei sind die x_i Bezeichner, die m_i ihre jeweiligen Typen ($1 \leq i \leq k$) und t ist ein Term des Typs $m_1 * \ldots * m_k$. Für den Fall $k = 1$ darf man in (OD) auf der linken Seite auf die Klammerung verzichten. ■

Durch die Konstruktion (OD) wird dem Bezeichnertupel der linken Seite der Wert der rechten Seite t zugeordnet. Es handelt sich also, wie bei der Deklaration von Typen durch type-binding, um eine **Abkürzung**. Insbesondere darf man im Fall $k = 1$, sofern der Wert von t definiert ist[19], überall im Restprogramm den Bezeichner x_1 statt des eventuell komplizierten Terms t schreiben. Wegen der Auffassung von Objektdeklarationen als Abkürzungen ist, wie beim type-binding, **keine Rekursion** möglich. Kommt einer der Bezeichner x_i der linken Seite von (OD) in der rechten Seite t vor, so muß x_i schon vorher im Programm durch eine andere Objektdeklaration eingeführt worden sein, und bei der Auswertung von t wird für x_i dann der dort berechnete Wert genommen.

Normalerweise wird man in einem ML-Programm allen deklarierten Dingen verschiedene Namen geben. Sind jedoch zwei Dinge gleicher Art auch gleich benannt, so wird durch die zweite Deklaration die erste „überschrieben“. Nachfolgend geben wir ein Beispiel an:

4.5.2 Beispiel (für Mehrfachdeklaration)

Wir betrachten das folgende einfache ML-Programm, in dem nacheinander zwei Rechenvorschriften mit jeweils gleichem Namen und zwei Objektdeklarationen mit jeweils gleicher linker Seite deklariert werden:

```
fun F (n : int) : int =
  if n = 0 then 1
           else n * F(n - 1);

val r : int = F(5);

fun F (n : int) : int =
  if n = 0 then 0
           else n + F(n - 1);

val r : int = F(r);
```

Durch die erste Deklaration wird `F` die auf $\mathbb{Z}$ erweiterte partielle Fakultätsfunktion als Semantik zugeordnet. Die zweite Deklaration bindet nun 5! an den Bezeichner `r`, d.h. `r` bekommt 120 zugeordnet. In der dritten Deklaration wird dem Bezeichner `F` als Semantik

[19] Dies ist eine Folge der Call-by-value-Semantik von ML.

eine neue partielle Funktion zugewiesen, nämlich die auf $\mathbb{Z}$ erweiterte Summe $\sum_{i=0}^{n} i$ für eine gegebene Zahl $n \in \mathbb{N}$ zu bestimmen. In der vierten Deklaration wird zuerst der Wert von `F(r)`, mit `F` und `r` wie bisher gegeben, bestimmt. Dies ergibt $\sum_{i=0}^{120} i = 7260$. Dann wird dieser Wert an den Bezeichner `r` gebunden. ■

Bisher haben wir Deklarationen nur global betrachtet. Es ist jedoch auch sinnvoll, und wir haben in Abschnitt 3.1 mit der Rechenvorschrift `sqrt` schon ein Beispiel dafür angegeben, Dinge lokal zu deklarieren. Solche lokalen Deklarationen erweitern nochmals die Möglichkeiten, in ML Terme zu definieren.

4.5.3 Lokale Deklarationen und Abschnitte

Es seien D_1 bis D_k Deklarationen der bisher betrachteten Gestalt, also von einzelnen (oder Systemen von) Rechenvorschriften, Typen (als Abkürzungen oder mit Konstruktoren, auch in Systemen) oder Objekten. Weiterhin sei t ein Term des Typs m. Dann ist

$$\texttt{let}\ D_1\texttt{;}\ \ldots\ D_k\texttt{;}\ \texttt{in}\ t\ \texttt{end} \qquad (A)$$

ebenfalls ein Term des Typs m. Ein Term der Form (A) wird **Abschnitt** mit **Deklarationsteil** (lokalen Deklarationen) $D_1\texttt{;}\ \ldots\ D_k\texttt{;}$ und **Rumpf** t genannt. ■

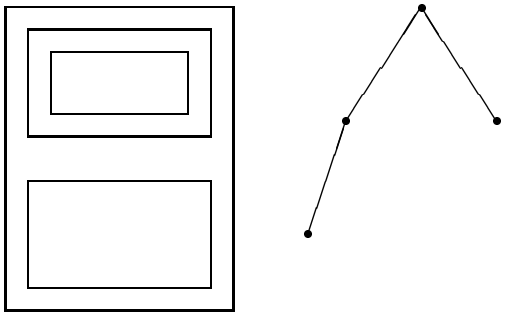
Als Konvention dürfen in (A) die die Deklarationen abschließenden Semikola auch weggelassen werden. Wir tun dies in Zukunft nur **mit dem letzten Semikolon** vor dem Symbol `in` und fassen damit das Semikolon als Trennzeichen zwischen und nicht als Abschlußzeichen von Deklarationen auf. Da Abschnitte per Definition wieder Terme sind, kann man sie beliebig schachteln. Nachfolgend geben wir so eine Schachtelung an. Sie dient zugleich als Motivation für Fragen, die sich im Zusammenhang mit der Semantik von Abschnitten ergeben.

4.5.4 Beispiel (für geschachtelte Abschnitte)

Wir betrachten den nachfolgenden ML-Term, der einen Abschnitt darstellt, in dem sich wiederum drei Abschnitte befinden:

```
let val x : int = 3
in  let fun F (y : int) : int = x + y
    in  let val x : int = 2
        in  F(1) + x
        end
    end
    * x *
    let val x : int = 2
    in  1 + x
    end
end
```

Graphisch kann man die geschachtelte Struktur der Abschnitte dieses Terms mittels mehrerer Möglichkeiten darstellen, wobei zwei davon nachfolgend angegeben sind.



Man bezeichnet die linke Darstellung mit den ineinandergeschachtelten Rechtecken auch als **Abschnittsstruktur** und die rechts von ihr angegebene baumartige Darstellung als den **Abschnittsstrukturbaum**. Diese Begriffe kommen eigentlich aus den sogenannten blockorientierten imperativen Programmiersprachen; wir haben sie hier auf den Fall der funktionalen Sprachen übertragen. ■

Schon das eben angegebene einfache Beispiel zeigt auf, daß man bei der Definition der Semantik von Abschnitten etwas aufpassen muß. So kommt etwa im Rumpf der Rechenvorschrift `F` ein Bezeichner `x` vor, der weder ein Parameter von `F` ist, noch im Rumpf von `F` im Rahmen einer Objektdeklaration eingeführt wurde. Solche Bezeichner nennt man **global** gegenüber `F`. Was soll man nun bei der Auswertung des Aufrufs `F(1)` als Wert für den globalen Bezeichner `x` nehmen? In der Praxis der Programmiersprachen werden bei globalen Bezeichnern zwei verschiedene Möglichkeiten von Bindung / Sichtbarkeit angewendet, die sich im konkreten Falle des Beispiels 4.5.4 wie folgt darstellen:

(1) Die erste Möglichkeit wählt den Bezeichner `x` des äußeren, der Baumwurzel entsprechenden, Abschnitts. Man wertet also `F(1)` mit einem `x` aus, welches mit `3` initialisiert wurde.

(2) Die Alternative nimmt bei der Auswertung von `F(1)` den Bezeichner `x` des Abschnitts, in dem die Rechenvorschrift aufgerufen wurde und welcher dem linkem Blatt im Abschnittsstrukturbaum entspricht. Hier wird `x` mit dem Wert `2` initialisiert.

Je nach der Wahl des Vorkommens von `x` erhalten wir 54 oder 45 als Resultat des Abschnitts von Beispiel 4.5.4. Wir gehen bei der allgemeinen Klärung der Bindung / Sichtbarkeit bei Abschnitten in kleinen Schritten vor. Zuerst reduzieren wir den allgemeinen Fall auf einen Spezialfall.

4.5.5 Reduktion auf spezielle Abschnitte

In der Programmiersprache ML ist für einen allgemeinen Abschnitt

$$\texttt{let}\ D_1\texttt{;}\ \ldots\ D_k\ \texttt{in}\ t\ \texttt{end}$$

der obigen Form (A) mit $k > 1$ Deklarationen die Semantik per Definition gleich der Semantik der geschachtelten Abschnitte

$$\texttt{let } D_1 \texttt{ in let } D_2 \texttt{ in } \ldots \texttt{ let } D_k \texttt{ in } t \texttt{ end} \ldots \texttt{end},$$

in denen jeweils genau eine Deklaration auftritt. ■

Nach dieser Reduktion des allgemeinen Falles verbleibt noch die Aufgabe, für einen Abschnitt mit genau einer Deklaration festzulegen, was seine Semantik ist. Der nachfolgende Begriff ist dafür fundamental. Man beachte, daß durch ihn auch der Fall erfaßt wird, daß der erwähnte Bezeichner x Teil eines Systems (von Rechenvorschriften oder Typen mit Konstruktoren) ist.

4.5.6 Definition (Gültigkeitsbereiche und Verschattung)

Gegeben sei ein Abschnitt der Form `let` D `in` t `end`, wobei in der Deklaration D ein Bezeichner x eingeführt wird. Dann ist der **Gültigkeitsbereich** (auch: **Sichtbarkeitsbereich**) von x, als Zeichenreihe[20] über dem Zeichenvorrat, aus dem ML-Programme aufgebaut werden können, dadurch gegeben, daß man im Rumpf t

a) alle Abschnitte entfernt, in denen x nochmals als Rechenvorschrift oder Typ mit Konstruktoren deklariert wird,

b) alle Abschnitte mit Ausnahme der rechten Seite der Deklaration entfernt, in denen x nochmals als Objekt oder Typabkürzung deklariert wird.

Die Unterscheidung in a) und b) rührt daher, daß in ML Deklarationen von Rechenvorschriften und Typen mit Konstruktoren Rekursion erlauben, Deklarationen von Objekten oder Typabkürzungen hingegen nicht. Den nach a) oder b) in einem Abschnitt entfernten Anteil von t nennt man auch den Bereich, in dem die Deklaration von x durch D in t **verschattet** oder **nicht sichtbar** ist. ■

Die folgenden zwei Beispiele machen den Unterschied bei den Gültigkeitsbereichen im Falle von rekursiven bzw. nichtrekursiven Deklarationen klar. Dabei sind die entfernten Teile jeweils kursiv dargestellt. Diese beiden Beispiele zeigen auch, daß Gültigkeitsbereiche in der Regel keine ML-Terme sind.

4.5.7 Beispiele (für Gültigkeitsbereiche)

a) Bei den geschachtelten Abschnitten

[20] Man beachte, daß ein Abschnitt ein Term ist und in Abschnitt 2.3 Terme als Zeichenreihen mit einer speziellen Struktur definiert wurden.

```
let fun F (x : int) : int = 1
in  F(let  fun F (y : int) : int = 2
       in  F(3)
       end) + F(1)
end
```

mit einer inneren (eventuell rekursiven) Deklaration einer Rechenvorschrift wird der gesamte innere Abschnitt entfernt, da in ihm die in der äußeren Deklaration eingeführte Rechenvorschrift `F` verschattet ist.

b) Hingegen wird bei den geschachtelten Abschnitten

```
let val x : int = 1
in  let  val x : int = x + 1
     in  x + 2
     end * x + 3
end
```

mit einer inneren (nichtrekursiven) Objektdeklaration die rechte Seite der inneren Deklaration von `x` nicht entfernt, da hier mit dem `x` der in der äußeren Deklaration eingeführte Bezeichner gemeint ist. ■

Nach diesen Vorarbeiten können wir nun festlegen, was die Semantik eines Abschnitts mit genau einer Deklaration ist.

4.5.8 Semantik von Abschnitten

Es werde im folgenden Abschnitt durch die Deklaration D ein Bezeichner x eingeführt:

`let` D `in` t `end`

Dann ist die **Semantik des Abschnitts** der Wert des Terms t, wobei für jedes Vorkommen von x im Gültigkeitsbereich der x in D zugewiesene Wert genommen wird. ■

Wir greifen zu einer Demonstration dieser Semantikfestlegung das motivierende Beispiel 4.5.4 am Anfang dieses Abschnitts noch einmal auf.

4.5.9 Beispiel (Weiterführung von 4.5.4)

Nach der Festlegung der Semantik von Abschnitten und dem Gültigkeitsbereich des ersten äußeren Vorkommens von `x` ist die Semantik des Terms von Beispiel 4.5.4 gleich dem Wert des nachfolgenden Terms; dieser Term entspricht genau dem Rumpf des äußeren

Abschnitts von Beispiel 4.5.4, bei dem jedes Vorkommen von `x` im Gültigkeitsbereich durch `3` ersetzt wurde.

```
let fun F (y : int) : int = 3 + y
in  let val x : int = 2
    in  F(1) + x
    end
end
* 3 *
let val x : int = 2
in  1 + x
end
```

In dem obigen Programmstück wird kein Bezeichner mehr verschattet und wir können die Werte der beiden Terme `F(1) + x` und `1 + x` mit `F(1) = 3 + 1` und `x = 2` im ersten Fall bzw. `x = 2` im zweiten Fall problemlos berechnen. Dies bringt

```
((3 + 1) + 2) * 3 * (1 + 2),
```

was sich schließlich durch Ausrechnen zu `54` vereinfacht. ■

Die durch die Reduktion 4.5.5 und die Semantik 4.5.8 festgelegte Bedeutung, bei der man bei globalen Größen in Rechenvorschriften den Abschnittsstrukturbaum in Richtung Wurzel zurückgeht und die erste Deklaration wählt, die man antrifft, heißt in der Literatur auch **statische Bindung** oder **statische Sichtbarkeitsregel**. Für die funktionale Sprache **ML ist per Semantikdefinition statische Bindung** festgelegt. Von **dynamischer Bindung** bzw. **dynamischer Sichtbarkeitsregel** spricht man hingegen, wenn von den nach dem Beispiel 4.5.4 aufgezeigten Möglichkeiten (2) gewählt wird. Wie das Beispiel 4.5.4 zeigt, müssen beide Bindungsmechanismen nicht immer das gleiche Resultat liefern. So wertet sich bei dynamischer Bindung der Abschnitt von Beispiel 4.5.4 zu `((2+1)+2)*3*(1+2)`, also zu `45` aus.

Nach diesen theoretischen Untersuchungen zur Semantik von Abschnitten wollen wir nun noch ihre praktischen Anwendungen bei der Programmierung studieren. Die Bedeutung hierin ist in vielerlei Hinsicht gegeben; nachfolgend demonstrieren wir einige anhand von ausgewählten Beispielen.

Die Verwendung von Abschnitten kann **Mehrfachauswertung von Termen verhindern und damit die Effizienz steigern**. Im nächsten Beispiel greifen wir auf die Anfänge der funktionalen Programmierung in diesem Skriptum zurück.

4.5.10 Beispiele (für Effizienzsteigerung durch Abschnitte)

a) In Beispiel 3.1.2 haben wir für den ganzzahligen Anteil der Quadratwurzel einer natürlichen Zahl die nachfolgende Rekursion entwickelt:

$$isqrt(n) = \begin{cases} 0 & : \quad n = 0 \\ isqrt(n-1)+1 & : \quad n \neq 0, (isqrt(n-1)+1)^2 \leq n \\ isqrt(n-1) & : \quad \text{sonst} \end{cases}$$

Dies führte, mittels einer Objektdeklaration, anschließend zu einer ML-Rechenvorschrift, die den ganzzahligen Anteil der Quadratwurzel von n durch n rekursive Aufrufe bestimmt. Verzichtet man auf die lokale Objektdeklaration, d.h. überträgt man die obige Rekursion direkt in die Sprache ML, so liefert dies:

```
fun isqrt (n : int) : int =
  if n = 0 then 0
           else if (isqrt(n-1)+1) * (isqrt(n-1)+1) <= n
                  then isqrt(n-1) + 1
                  else isqrt(n-1);
```

Im Falle eines Arguments $n > 0$ zieht jeder Aufruf dieser Rechenvorschrift `isqrt` drei unmittelbare weiteren rekursiven Aufrufe von `isqrt` nach sich. Insgesamt führt die Berechnung von $isqrt(n)$ somit zu

$$1 + 3 + 3^2 + \ldots + 3^n = \sum_{i=0}^{n} 3^i = \frac{3^{n+1} - 1}{2}$$

Aufrufen von `isqrt`. Der Gegensatz zur programmiersprachlichen Realisierung der Funktion *isqrt* in Abschnitt 3.1, wo in der entsprechenden Rechenvorschrift die Mehrfachauswertung durch einen Abschnitt verhindert wird, stellt sich durch einige Zahlenbeispiele für die Aufrufe wie folgt dar:

n	obige Fassung	Fassung aus 3.1
1	1	1
5	364	5
10	88 573	10
15	21 523 360	15
20	5 230 176 601	20

Besonders eindrucksvoll wird der Vergleich, wenn man zur Abarbeitung eines Aufrufs eine reale Zeit annimmt, sagen wir eine Millisekunde. Dann braucht die Version ohne Abschnitt zur Bestimmung von $isqrt(20)$ ungefähr 70 Tage, während die Originalversion von Abschnitt 3.1 mit 20 Millisekunden auskommt.

b) Nun betrachten wir die Folge der sogenannten Fibonacci-Zahlen, die schon im Mittelalter aufgestellt wurde und überraschenderweise bei der Lösung des in Beispiel 2.4.4 genannten 10. Hilbertschen Problems eine entscheidende Rolle spielte. Die Folge der Fibonacci-Zahlen $\langle fib(n) \rangle_{n \in \mathbb{N}}$ ist rekursiv wie folgt definiert:

(1) $fib(0) = 1$ (2) $fib(1) = 1$ (3) $fib(n+2) = fib(n+1) + fib(n)$

Aus dieser Definition ergibt sich sofort eine kaskadenartig-rekursive ML-Rechenvorschrift zur Berechnung der Fibonacci-Zahlen:

```
fun fib (n : int) : int =
  if n = 0 orelse n = 1 then 1
                        else fib(n-1) + fib(n-2);
```

Diese Rechenvorschrift ist jedoch schrecklich ineffizient, da während der Abarbeitung viele Aufrufe mehrfach berechnet werden. Der Leser überprüfe dies an einigen selbstgewählten kleinen Beispielen, etwa `fib(7)`. Wie kommt man nun zu einer effizienteren Version? Dazu betrachten wir, geleitet durch die Rekursion der Fibonacci-Zahlen, die Funktion $G : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N}$, definiert mittels

$$G(n) = \langle \mathit{fib}(n), \mathit{fib}(n+1) \rangle. \qquad (G_1)$$

Aufgrund der Gleichungen (1) und (2) gilt dann die Gleichung

$$G(0) = \langle 1, 1 \rangle. \qquad (G_2)$$

Wir wollen, wegen dieses Terminierungsfalles, die Berechnung von $G(n)$ rekursiv auf die Berechnung von $G(n-1)$ zurückführen. Zu diesem Zwecke sei $\langle x, y \rangle = G(n-1)$ vorausgesetzt. Dann bekommen wir

$$\begin{array}{rcll} G(n) & = & \langle \mathit{fib}(n), \mathit{fib}(n+1) \rangle & \text{Definition von } G \\ & = & \langle \mathit{fib}(n), \mathit{fib}(n-1) + \mathit{fib}(n) \rangle & \text{Fibonacci-Gleichung (3)} \\ & = & \langle y, x+y \rangle & \text{Definition von } G, x, y, \end{array}$$

also die rekursive Beziehung

$$G(n) = \langle y, x+y \rangle \quad \text{mit} \quad \langle x, y \rangle = G(n-1). \qquad (G_3)$$

Die Eigenschaften (G_2) und (G_3) lassen sich nun sofort als ML-Rechenvorschrift `G` mit einem Abschnitt formulieren. Verwenden wir auch noch die Eigenschaft (G_1) zur Einbettung der Original-Rechenvorschrift `fib` in `G`, so bekommen wir

```
fun G (n : int) : int*int =
  if n = 0 then (1,1)
           else let val (x : int, y : int) = G(n-1)
                in  (y,x+y)
                end;

fun fib (n : int) : int =
  let val (x : int, y : int) = G(n)
  in  x
  end;
```

als effizientes ML-Programm zur Berechnung von Fibonacci-Zahlen. ■

Abschnitte erlauben die Einführung von Hilfsbezeichnungen, wie Hilfsrechenvorschriften und -typen, die nach außen verborgen bleiben. Damit kann bei Programmumformungen die **Schnittstelle**, d.h. die Namen der deklarierten Bezeichner plus deren Typ bzw. Funktionalität erhalten werden. Lokale Hilfsbezeichnungen ermöglichen es oft auch, **Programme besser zu strukturieren** und dadurch lesbarer zu programmieren. Wir könnten all dies demonstrieren, indem wir die bisherigen Beispiele nochmals betrachten würden. Wir haben uns jedoch zu neuen Beispielen entschlossen, da durch diese auch zwei der für die Informatik insgesamt immens wichtigen Sortierverfahren vorgestellt werden können.

4.5.11 Beispiele (für die Strukturierung durch Abschnitte; Sortieren)

Wir betrachten das Problem, eine Liste von ganzen Zahlen in aufsteigender Ordnung zu sortieren. Eine mathematische Präzisierung der Sortierfunktion

$$sort : \mathbb{Z}^* \longrightarrow \mathbb{Z}^*$$

hat zweierlei Dinge zu verlangen, nämlich, daß (i) für jedes Argument s das Resultat $sort(s)$ aufsteigend sortiert ist bezüglich der Ordnung $\leq$ und (ii) für jedes Argument s das Resultat $sort(s)$ die gleichen Elemente in der gleichen Anzahl wie s enthält, d.h. eine Permutation von s ist. Dies führt etwa zu der folgenden formalen Festlegung der Sortierfunktion *sort*, im Jargon auch (formale) **Spezifikation des Sortierproblems** genannt:

$$\begin{array}{ll} (1) & \forall\, i : 1 \leq i \leq |sort(s)| - 1 \rightarrow sort(s)_i \leq sort(s)_{i+1} \\ (2) & \forall\, n \in \mathbb{Z} : |\{\ i \in \mathbb{N}\ :\ s_i = n\ \}| = |\{\ i \in \mathbb{N}\ :\ sort(s)_i = n\ \}| \end{array}$$

Wie kann man nun in einer funktionalen Programmiersprache wie ML sortieren, d.h. einen Sortieralgorithmus programmieren? Nachfolgend geben wir zwei Verfahren an. Dabei verzichten wir jeweils auf den Korrektheitsbeweis, d.h. den Nachweis, daß die Semantik der eben gegebenen Spezifikation genügt. Insbesondere ein formal geführter Korrektheitsbeweis für das zweite Verfahren würde den Rahmen dieses Skriptums weit überschreiten.

a) Ein erster Ansatz zum Sortieren in ML ist, sich auf eine Hilfsrechenvorschrift zu stützen, die ein Element `y` **in eine sortierte Liste** `w` **einsortiert**. Eine rekusive Formulierung solch einer Rechenvorschrift

```
fun insort (y : int, w : int list) : int list =...
```

durch eine (geschachtelte) Fallunterscheidung ist offensichtlich.

(1) Ist die lineare Liste `w` leer, so liefert man als Resultat die einelementige Liste `[y]` ab.

(2) Andernfalls vergleicht man das einzusortierende Element `y` mit dem ersten Element von `w`. Ist `y` kleiner oder gleich `hd(w)`, so fügt man `y` von vorne an `w` an. Ansonsten sortiert man `y` in den Rest `tl(w)` von `w` ein und fügt dann `hd(w)` vorne an.

Aufbauend auf die Rechenvorschrift `insort` kann nun wie folgt rekursiv sortiert werden: Ist die Liste leer, so ist nichts zu tun also die Eingabe gleich der Ausgabe. Ansonsten sortiert man das erste Element in die sortierte Restliste ein. Formal erhalten wir die folgende ML-Rechenvorschrift `sort` zum Sortieren, wobei die Hilfsrechenvorschrift `insort` lokal deklariert und damit nach außen nicht sichtbar ist.

```
fun sort (s : int list) : int list =
  let fun insort (y : int, w : int list) : int list =
        if null(w) then [y]
                   else if y <= hd(w) then y :: w
                                      else hd(w) :: insort(y,tl(w))
  in  if null(s) then s
                 else insort(hd(s),sort(tl(s)))
  end;
```

b) Eine weitere Möglichkeit, eine Liste w von ganzen Zahlen durch ein rekursives Verfahren zu sortieren, kann man umgangssprachlich wie folgt beschreiben:

(1) Ist die vorliegende Liste leer, so mache nichts, d.h. liefere auch die leere Liste als Resultat zurück.

(2) Ist die Liste nicht leer, so wähle irgendein Element a aus ihr und teile die Liste (unter Beibehaltung der Reihenfolge der Elemente) in drei Teillisten auf, nämlich die Liste *lo* der Elemente echt kleiner als a, die Liste *eq* der Elemente gleich a und die Liste *up* der Elemente echt größer als a. Sortiere dann die zwei Listen *lo* und *up* und liefere als Resultat die Konkatenation der sortierten Liste *lo* mit der Liste *eq* und der sortierten Liste *up*.

Anhand eines kleinen Beispiels wird diese Vorgehensweise dem Leser hoffentlich schnell klar. Wir betrachten die nachfolgende Liste:

$$13, 15, 28, 5, 12, 44, 7, 9, 44, 12, 12, 33$$

Die Wahl der Zahl 12 führt im ersten Schritt zur folgenden Unterteilung, welche wir durch zwei Leerräume andeuten:

$$5, 7, 9, \qquad 12, 12, 12, \qquad 13, 15, 28, 44, 44, 33$$

Beim zweiten Schritt terminiert beispielsweise das Verfahren bei der Wahl von 7 für die linkeste Teilliste; die zufällige Wahl von 28 führt zu folgenden Unterteilung der rechtesten Teilliste, die wir durch etwas kleinere Leerräume andeuten:

$$5, \quad 7, \quad 9, \qquad 12, 12, 12, \qquad 13, 15, \quad 28, \quad 44, 44, 33$$

Damit müssen wir nur noch die rechteste Teilliste sortieren. Wählen wir etwa das zweite Vorkommen der Zahl 44 zur Unterteilung, was wir durch noch kleinere Leerräume andeuten, so terminiert das Verfahren mit folgender Sortierung:

$$5, \quad 7, \quad 9, \qquad 12, 12, 12, \qquad 13, 15, \quad 28, \quad 33, \ 44, 44$$

Das „Herausfiltern“ der drei Teillisten *lo*, *eq* und *up* aus der Eingabe w kann in ML sehr einfach mit Hilfe der Rechenvorschrift `filter` von Beispiel 4.3.2.b in Kombination mit einer Funktionsabstraktion bewerkstelligt werden. In Verbindung mit drei Objektdeklarationen erhalten wir schließlich als Umsetzung des oben beschriebenen Verfahrens die nachfolgende ML-Rechenvorschrift:

```
fun quicksort (w : int list) : int list =
  if null(w)
    then w
    else let val lo : int list = filter(fn (n : int) => n<hd(w),w);
             val eq : int list = filter(fn (n : int) => n=hd(w),w);
             val up : int list = filter(fn (n : int) => n>hd(w),w)
         in  quicksort(lo) @ eq @ quicksort(up)
         end;
```

Der Name **Quicksort** für dieses Sortierverfahren ist allgemein üblich. Er wurde deshalb gewählt, weil es sich um das derzeit schnellste Sortierverfahren handelt. Da wir in ML keine Möglichkeit haben, ein beliebiges Element zu wählen, wird in der obigen Rechenvorschrift `quicksort` das erste Listenelement zur Unterteilung verwendet.

Vergleicht man Quicksort mit beliebiger Wahl von a mit den in Abschnitt 2.4 vorgestellten Eigenschaften von Algorithmen, so stellt man fest, daß es sich um einen nichtdeterministischen, aber determinierten Algorithmus handelt. In der ML-Version wurde der Nichtdeterminismus durch die Wahl von a als `hd(w)` entfernt. ■

Wir haben uns bei den gerade präsentierten Sortierprogrammen der Einfachheit halber auf das Sortieren von linearen Listen ganzer Zahlen beschränkt. Es ist offensichtlich, daß alle Programme mit Hilfe eines funktionalen Parameters des Typs `'m*'m->bool` polymorph auf beliebigen Listen von Objekten einer geordneten Menge formuliert werden können, solange in dieser Ordnung jeweils zwei Elemente vergleichbar sind. Dabei wird die Menge durch die Typvariable dargestellt und die Ordnung durch den funktionalen Parameter.

Schon diese einfachen Beispiele zeigen, daß Polymorphie und funktionale Terme zwei fundamentale Begriffe für die **Wiederverwendbarkeit von Programmen** sind. Leider können wir auf diesen Punkt nicht näher eingehen und müssen den Leser deshalb auf weiterführende Literatur verweisen.

4.6 Rechenvorschriften und Muster

In Abschnitt 4.2 haben wir gezeigt, wie man in der Sprache ML rekursive Datenstrukturen durch Anwendung einer Deklaration mit Konstruktoren definieren kann. Dies entspricht im Prinzip der induktiven Definition von Mengen, wie sie in Abschnitt 1.2 vorgestellt wurde. Dort haben wir auch demonstriert, wie die induktive Erzeugung von Mengen die induktive Definition von Funktionen auf ihnen erlaubt. Für dieses Vorgehen stellt die

Programmiersprache ML durch die Definition von Rechenvorschriften mittels sogenannter Muster die entsprechenden Sprachmittel bereit. Der Leser mache sich klar, daß die bisher vorgestellten ML-Sprachmittel zum Aufbau von Rechenvorschriften noch nicht erlauben, sinnvolle Programme über Typen mit Konstruktoren, etwa rekursiven Datenstrukturen, zu formulieren, da man Objekte solcher Typen bisher noch nicht in kleinere Bestandteile zerlegen kann.

4.6.1 Definition von Rechenvorschriften durch Musteranpassung

Gegeben sei eine Deklaration eines Typs m mit den Konstruktoren b_i, $1 \leq i \leq k$, und c_j, $1 \leq j \leq n$, in der folgenden schematischen Form, wobei bei den b_i die in Abschnitt 4.2 vorgestellte und den Typ `unit` betreffende syntaktische Vereinfachung verwendet wird:

```
datatype m = b1 |
             ...
             bk |
             c1 of m11 * ... * m1s1 |
             ...
             cn of mn1 * ... * mnsn ;
```

So eine Deklaration ist das programmiersprachliche Gegenstück zu einer induktiven Erzeugung einer Menge im Sinne von Definition 1.2.5. Die Konstruktoren b_1 bis b_k werden zu Konstanten und diese entsprechen genau der Basis B, während die Konstruktoren c_1 bis c_n genau der Menge C der Konstruktorfunktionen entsprechen.

Analog zur induktiven Definition von Funktionen auf induktiv erzeugten Mengen ist es in ML möglich, eine Rechenvorschrift des Typs $\ldots m \ldots$ `->` p wie folgt über den termmäßigen Aufbau der Objekte des Typs m zu deklarieren:

```
fun F (...,b1 : m,...) : p = e1
  ...
  | F (...,bk : m,...) : p = ek
  | F (...,c1(t11,...,t1s1) : m,...) : p = ek+1
  ...
  | F (...,cn(tn1,...,nnsn) : m,...) : p = ek+n;
```

Dabei sind die Argumente t_{ij} der Konstruktoren jeweils Terme passenden Typs, die aufgebaut sind mit Hilfe von frei wählbaren Bezeichnern (wiederum Parameter genannt) sowie den Konstruktoren b_i und c_j der Typdeklaration von m. Auch die **rechten Seiten** e_l nach dem Gleichheitssymbol sind Terme des Typs p, die wie bisher aufgebaut sind.

Deklariert man eine Rechenvorschrift F in der eben aufgezeigten Art und Weise, so ist jeder Term der Form F`(...,`b_i`,...)` oder F`(...,`c_j`(`t_{j1}`,...,`t_{js_j}`),...)` sozusagen ein **Muster** für einen erwarteten Aufruf. Man spricht deshalb auch von der Deklaration bzw. Definition von Rechenvorschriften durch **Musteranpassung**. Um Mehrdeutigkeiten zu verhindern, verbietet ML die mehrfache Verwendung eines Parameters in einer **linken Seite** F`(...,`b_i`,...)` oder F`(...,`c_j`(`t_{j1}`,...,`t_{js_j}`),...)`. ■

Die Deklaration von Rechenvorschriften durch Musteranpassungen ist in funktionalen Programmiersprachen ein allgemeinübliches Konzept und nicht nur auf die programmiersprachliche Beschreibung von induktiv erzeugten Mengen und darauf induktiv definierten Funktionen beschränkt. Nachfolgend geben wir in ML, auch abgestützt auf frühere Typdeklarationen, einige Anwendungsbeispiele an.

4.6.2 Beispiele (für Muster bei Typdeklarationen mit Konstruktoren)

a) In Beispiel 4.1.5 haben wir Geraden in zwei verschiedenen Darstellungen behandelt. Es sei g eine Gerade. Dann ist g **parallel zur Abszisse** im Falle der Normalform-Darstellung $a * x + b * y + c = 0$ genau dann, wenn $a = 0$ gilt, und im Falle der Vektorform-Darstellung $\boldsymbol{x} + \lambda * \boldsymbol{y}$ mit $\boldsymbol{x} = \langle x_1, x_2 \rangle$ und $\boldsymbol{y} = \langle y_1, y_2 \rangle$ genau dann, wenn $y_2 = 0$ zutrifft. Unter Abstützung auf die Deklarationen von Beispiel 4.1.5 können wir somit in der Programmiersprache ML durch eine mittels Musteranpassung deklarierte Rechenvorschrift wie folgt feststellen, ob eine Gerade parallel zu der Abszisse verläuft:

```
fun par_x (nfline(mnf(a,b,c)) : line) : bool =
    abs(a) <= 0.001
  | par_x (vfline(mvf((x1,x2),(y1,y2))) : line) : bool =
    abs(y2) <= 0.001;
```

Dabei haben wir die Tests, ob `a` gleich Null ist bzw. ob `y2` gleich Null ist, durch `abs(a) <= 0.001` bzw. durch `abs(y2) <= 0.001` ausgedrückt, da für die Sorte `real` kein Gleichheitstest existiert.

b) In Beispiel 4.2.5 haben wir die Menge $\mathsf{Tree}(A)$ der allgemeinen oder beliebig-vergabelten Bäume mit Markierungen aus einer Menge A definiert. Nun wollen wir die **Anzahl** der in einem Baum **vorkommenden Knoten** zählen. Eine einfache Möglichkeit ist, die entsprechende Funktion

$$count : \mathsf{Tree}(A) \longrightarrow \mathbb{N}$$

induktiv über den Aufbau der Bäume zu definieren: Den Induktionsanfang bilden dabei die atomaren Bäume mit der Festlegung

$$count(a) = 1 \qquad (C_1)$$

für alle $a \in A$. In den anderen Fällen haben wir jeden Baum als ein Paar $\langle a, w \rangle$ mit $a \in A$ als atomaren Baum und $w \in \mathsf{Tree}(A)^*$ als Liste von Bäumen vorliegen. Nun unterscheiden wir weiter: Gilt nämlich $w = \varepsilon$, d.h. a ist ein Knoten / Baum ohne Nachfolger, so legen wir fest

$$count(\langle a, \varepsilon \rangle) = 1, \qquad (C_2)$$

und im Falle $w \neq \varepsilon$, d.h. $w = b \,\&\, s$, definieren wir

$$count(\langle a, b \,\&\, s \rangle) = count(b) + count(\langle a, s \rangle). \qquad (C_3)$$

Die Gleichung (G_3) drückt aus, daß man zuerst die Knotenanzahl des ersten Nachfolgerbaums von a bestimmt und zu dieser Zahl dann die Knotenanzahlen der restlichen Bäume addiert.

Nun setzen wir eine polymorphe rekursive Deklaration von beliebigen Bäumen voraus, wie sie sich als Erweiteung der früheren Typdeklaration durch eine Typvariable `'m` sofort ergibt, sowie einen polymorphen rekursiven Datentyp für Listen mit `nil` für die leere Liste und `append` für das Anfügen eines Elements von links. Dann führen die Eigenschaften (C_1), (C_2) und (C_3) zur Deklaration

```
fun count (atom(a) : 'm tree) : int =
    1
  | count (succs(a,nil) : 'm tree) : int =
    1
  | count (succs(a,append(t,s)) : 'm tree) : int =
    count(t) + count(succs(a,s));
```

einer Rechenvorschriften `count` zum Knotenzählen bei beliebig-vergabelten Bäumen durch Musteranpassung. ■

Die Definition von Rechenvorschriften durch Musteranpassung kann natürlich auch im Falle der vorimplementierten linearen Listen verwendet werden, da hier die Standarddarstellungen wie `[]` oder `[3,5,6]` als Abkürzungen für bestimmte Terme aufzufassen sind. In dem nachfolgenden Beispiel 4.6.3 geben wir zuerst zwei solche Rechenvorschriften an, die anschließend in einfacher Weise zu einem weiteren schnellen Sortierprogramm in der Programmiersprache ML kombiniert werden. In der Literatur ist das durch dieses Programm implementierte Verfahren als „Sortieren durch Mischen“ oder **Mergesort** bekannt.

4.6.3 Beispiel (für Muster bei linearen Listen; Mergesort)

Wir betrachten zuerst die nachfolgende, durch Musteranpassung auf den linearen Listen definierte polymorphe Rechenvorschrift `split`, die eine lineare Liste in zwei Teillisten aufteilt, welche als Resultatpaar abgeliefert werden:

```
fun split (nil : 'm list) : 'm list * 'm list =
    (nil,nil)
  | split ([x] : 'm list) : 'm list * 'm list =
    ([x],nil)
  | split (x :: y :: s : 'm list) : 'm list * 'm list =
    let val (u : 'm list, v : 'm list) = split(s)
    in  (x :: u,y :: v)
    end;
```

Die algorithmische Idee hinter `split` ist „ein Element in die erste Liste und ein Element in die zweite Liste“. Ist die Länge der Eingabe ungerade, so enthält die erste Liste wegen

des zweiten Musters von `split` genau ein Element mehr als die zweite. Beispielsweise gilt also im Fall von ganzen Zahlen

```
split([2,4,6,1,3,7,7]) = ([2,6,3,7],[4,1,7]).
```

Als zweite auf den vordefinierten linearen Listen durch Musteranpassung definierte Rechenvorschrift betrachten wir die nachfolgend angegebene:

```
fun merge (nil : int list, t : int list) : int list =
    t
  | merge (s : int list, nil : int list) : int list =
    s
  | merge (x :: s : int list, y :: t : int list) : int list =
    if x <= y then x :: merge(s, y :: t)
              else y :: merge(x :: s, t);
```

Man beachte, daß wegen des im Rumpf vorkommenden Größenvergleichs `x <= y` die Rechenvorschrift `merge` nicht mit Hilfe einer Typvariablen an Stelle des Typs `int` formuliert werden kann. Sind die linearen Listen `s` und `t` von ganzen Zahlen sortiert, so ist offensichtlich auch die durch den Aufruf `merge(s,t)` entstehende lineare Liste sortiert. Weiterhin enthält sie die gleichen Elemente wie die Konkatenation von `s` und `t` und in gleicher Anzahl. Wir geben auch hier ein Beispiel an:

```
merge([1,3,5,7],[1,3,5,7,9,11]) = [1,1,3,3,5,5,7,7,9,11]
```

Aufbauend auf die beiden eben vorgestellten Rechenvorschriften `split` und `merge` ist es nun sehr einfach, eine ML-Rechenvorschrift `mergesort` anzugeben, die eine lineare Liste von ganzen Zahlen sortiert. Das durch diese Rechenvorschrift beschriebene Sortierverfahren teilt eine lineare Liste mit mindestens zwei Elementen in zwei etwa gleichlange Teillisten auf, sortiert diese rekursiv, und mischt schließlich die beiden sortierten Teillisten zu einer Sortierung der Originalliste zusammen:

```
fun mergesort (s : int list) : int list =
  if length(s) <= 1 then s
                    else let val (u : int list, v : int list) = split(s)
                         in  merge(mergesort(u),mergesort(v))
                         end;
```

Offensichtlich kann das in Beispiel 4.5.11.a behandelte Sortierverfahren durch Einsortieren in eine sortierte lineare Liste als Spezialfall des Sortierens durch Mischen verstanden werden, wenn die Aufteilung der linearen Liste durch die Variante

```
fun split (nil : 'm list) : 'm list * 'm list =
    (nil,nil)
  | split (x :: s : 'm list) : 'm list * 'm list =
    ([x],s);
```

der obigen Rechenvorschrift `split` in das erste Listenelement (als Liste der Länge 1) und die Restliste erfolgt. ■

Wir geben noch ein letztes Beispiel für eine typische Verwendung von Mustern an, nämlich, wenn das Resultat einer Berechnung von zweierlei Gestalt ist. Diese Situation tritt in der Praxis sehr häufig auf.

4.6.4 Beispiel (für die Realisierung eines zweigestaltigen Resultats)

Wir betrachten die Aufgabe, zu zwei Zeichenreihen $s \in A^*$ und $p \in A^+$ festzustellen, ob p eine Teilzeichenreihe von s ist, und bei einer positiven Antwort den kleinsten Index i mit $s_i \ldots s_{i+|p|-1} = p$ abzuliefern. Eine negative Antwort ist durch ein anderes geeignetes Resultat anzuzeigen.

Eine elegante Formalisierung des Problems ist gegeben durch die Funktion

$$pm : A^* \times A^+ \longrightarrow \mathbb{N}_\infty ,$$

welche definiert ist mittels der Festlegung

$$pm(s,p) = min\{\ i \in \mathbb{N}_\infty \ :\ s_i \ldots s_{i+|p|-1} = p\ \}.$$

Dabei sind die natürlichen Zahlen um ein zusätzliches größtes Element ∞ erweitert, d.h. $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$, und ∞ wird als das Minimum der leeren Menge definiert. Somit ist genau durch ∞ die negative Antwort „p ist keine Teilzeichenreihe von s“ dargestellt. Weiterhin wird die Zeichenreihe $s_i \ldots s_j$ für $j < i$ oder $s = \varepsilon$ als ε definiert.

Erweitert man die Nachfolgerbildung von $\mathbb{N}$ auf $\mathbb{N}_\infty$ in einer naheliegenden Weise mittels $\infty + 1 := \infty$, so bekommt man durch eine einfache Rechnung, in der die Tatsache, daß p nicht leer ist, wesentlich eingeht, die folgende terminierende Rekursion:

$$pm(s,p) = \begin{cases} \infty & : \ s = \varepsilon \\ 1 & : \ s \neq \varepsilon \ \wedge \ s_1 \ldots s_{|p|} = p \\ pm(rest(s),p) + 1 & : \ s \neq \varepsilon \ \wedge \ s_1 \ldots s_{|p|} \neq p \end{cases}$$

Üblicherweise formuliert man diese Rekursion in einer programmiersprachlichen Notation, indem man eine „große“ und nicht mehr als Resultat in Frage kommende Zahl als Ersatz für ∞ nimmt. In ML sieht dies, etwa mit `10000` als Ersatz für ∞, wie folgt aus:

```
fun lp (p : string, s : string) : bool =
  size(p) <= size(s) andalso substring(s,0,size(p)) = p;

fun succ (x : int) : int =
  if x = 10000 then x else x + 1;

fun pm (s : string, p : string) : int =
  if s = "" then 10000
            else if lp(p,s) then 1
                            else succ(pm(rest(s),p));
```

Dabei realisiert ein Aufruf der Rechenvorschrift `lp` den Test, ob eine Zeichenreihe Anfang einer anderen Zeichenreihe ist, wie er in der Rekursion der Funktion pm durch die

Gleichung $s_1 \ldots s_{|p|} = p$ beschrieben ist, und ein Aufruf der (schon früher angegebenen) Rechenvorschrift `rest` entfernt das erste Zeichen.

Eine Formulierung, die Abstand nimmt von der Hilfskonstruktion „∞ ist eine große Zahl“, ist möglich durch die Typdeklaration

```
datatype res = infty | intres of int;
```

und eine Rechenvorschrift, die `res` an Stelle von `int` als Resultattyp besitzt. Wir haben dazu im Prinzip nur die Nachfolgeroperation auf `res` zu realisieren und in der Rechenvorschrift `pm` die entsprechenden Anpassungen durchzuführen. Das Resultat (ohne die schon erklärte Rechenvorschrift `lp`) ist:

```
fun succ (infty : res) : res = infty
  | succ (intres(x) : res) : res = intres(x+1);

fun pm (s : string, p : string) : res =
  if s = "" then infty
            else if lp(p,s) then intres(1)
                            else succ(pm(rest(s),p));
```

Um die wahre Bedeutung der Konstanten `infty` besser zu betonen, bietet sich schließlich noch an, einen zutreffenderen Bezeichner zu wählen, etwa `no_substring`. ■

In den letzten Beispielen haben wir in allen Rechenvorschriften auf der linken Seite jeweils nur verschiedene Muster. Dies ist auch der Normalfall bei Deklarationen von Rechenvorschriften durch Musteranpassung und eine **Auswertung eines Aufrufs zu einem Argument entspricht operational dann genau einer Termersetzungberechnung mit jeweils eindeutiger Regelanwendung** solange, bis eine Normalform erreicht ist. Dabei entstehen die Termersetzungsregeln durch das Richten der einzelnen Fälle der Rechenvorschrift von links nach rechts.

Nicht sinnvoll, wenn auch in der Programmiersprache ML erlaubt, ist es, auf der linken Seite einer durch Musteranpassung deklarierten Rechenvorschrift ein Muster mehrmals anzugeben oder sich überlappende Muster zu wählen. Um einen Nichtdeterminismus zu vermeiden, wie er sich bei allgemeiner Termersetzung in so einer Situation in der Regel ergibt, werden in allen funktionalen Programmiersprachen mit Musterkonzept die Muster bei der Auswertung einer Rechenvorschrift gesteuert angepaßt. Die nachstehende Regelung gilt ohne Ausnahme für alle diese Sprachen.

4.6.5 Regel (zur gesteuerten Musteranpassung)

Man geht die einzelnen Fälle der vorliegenden Rechenvorschrift textuell von oben nach unten der Reihe nach durch und nimmt die Musteranpassung innerhalb jedes einzelnen Falles von links nach rechts vor. ■

Das Terminierungsverhalten einer durch Musteranpassung deklarierten Rechenvorschrift kann somit von der Reihenfolge der Tests bezüglich des anzupassenden Musters abhängen.

Eine gute Regel ist, die Muster bei der Deklaration einer Rechenvorschrift so anzuordnen, daß sie von oben nach unten immer allgemeiner werden. Damit wird bei der Abarbeitung zuerst immer der speziellere Fall genommen und ein allgemeinerer Fall, bei dem man vielleicht etwas übersehen hat, zurückgestellt.

Wir haben bisher Muster nur in Verbindung mit der Deklaration von Rechenvorschriften durch Musteranpassung betrachtet. Die Programmiersprache ML erlaubt jedoch die Verwendung von Mustern in sehr vielfältiger Weise. Wir wollen im folgenden noch kurz auf einen solchen Fall eingehen, der insbesondere für die praktische Programmierung sehr nützlich sein kann. Er verallgemeinert die uns schon bekannte Fallunterscheidung von zwei auf beliebig viele Fälle.

4.6.6 Fallunterscheidung durch Musteranpassung

Wir betrachten wiederum eine Deklaration eines Typs m mit den Konstruktoren b_i und c_j in der folgenden schematischen Form:

$$
\begin{array}{l}
\texttt{datatype}\ m\ \texttt{=}\ b_1\ \texttt{|} \\
\qquad\qquad \ldots \\
\qquad\qquad b_k\ \texttt{|} \\
\qquad\qquad c_1\ \texttt{of}\ m_{11}\ \texttt{*}\ \ldots\ \texttt{*}\ m_{1s_1}\ \texttt{|} \\
\qquad\qquad \ldots \\
\qquad\qquad c_n\ \texttt{of}\ m_{n1}\ \texttt{*}\ \ldots\ \texttt{*}\ m_{ns_n}\texttt{;}
\end{array}
$$

Es sei a ein Term des Typs m. Dann hat die allgemeine **Fallunterscheidung durch Musteranpassung** die syntaktische Form

$$
\begin{array}{l}
\texttt{case}\ a\ \texttt{of}\ b_1\ \texttt{=>}\ e_1 \\
\qquad\qquad \texttt{|}\ \ldots \\
\qquad\qquad \texttt{|}\ b_k\ \texttt{=>}\ e_k \\
\qquad\qquad \texttt{|}\ c_1\texttt{(}t_{11}\texttt{,}\ldots\texttt{,}t_{1s_1}\texttt{)}\ \texttt{=>}\ e_{k+1} \\
\qquad\qquad\quad \ldots \\
\qquad\qquad \texttt{|}\ c_n\texttt{(}t_{n1}\texttt{,}\ldots\texttt{,}t_{ns_n}\texttt{)}\ \texttt{=>}\ e_{k+n}
\end{array}
$$

mit Termen t_{ij} von jeweils passendem Typ, die aufgebaut sind mit Hilfe von frei wählbaren Bezeichnern und den Konstruktoren b_i und c_j der Typdeklaration, und Termen e_l, die alle den gleichen Typ besitzen müssen. Dieser Typ ist dann auch als der Typ der Fallunterscheidung durch Musteranpassung definiert.

Die Bedeutung einer allgemeinen Fallunterscheidung durch Musteranpassung ist operational analog zur Bedeutung der mittels Musteranpassung definierten Rechenvorschriften festgelegt. Man geht also die einzelnen Muster b_1 bis $c_n\texttt{(}t_{n1}\texttt{,}\ldots\texttt{,}t_{ns_n}\texttt{)}$ textuell von oben nach unten der Reihe nach durch und nimmt als Wert der Fallunterscheidung den Wert des ersten Terms e_{l_0}, für den eine Anpassung des Musters a im Sinne von Termersetzung erfolgreich ist.

Wegen dieser Übereinstimmung der Semantik der allgemeinen Fallunterscheidung durch

Musteranpassung mit der Semantik der durch Musteranpassung deklarierten Rechenvorschriften kann man jede solche Rechenvorschrift in der schematischen Form

```
fun F (...,b1 : m,...) : p = e1
  ...
  | F (...,bk : m,...) : p = ek
  | F (...,c1(t11,...,t1s1) : m,...) : p = ek+1
  ...
  | F (...,cn(tn1,...,nnsn) : m,...) : p = ek+n;
```

mit Hilfe einer Fallunterscheidung durch Musteranpassung und unter Verwendung eines Parameters x an Stelle des obigen Terms a gleichwertig schematisch wie folgt formulieren:

```
fun F (...,x : m,...) : p =
   case x of b1 => e1
           | ...
           | bk => ek
           | c1(t11,...,t1s1) => ek+1
             ...
           | cn(tn1,...,tnsn) => ek+n;
```

Diese Form ist manchmal etwas übersichtlicher als die Originalform. ■

Die bisher immer verwendete binäre Fallunterscheidung kann als ein Spezialfall der eben eingeführten allgemeinen Fallunterscheidung durch Musteranpassung angesehen werden. Wenn man nämlich die Sorte `bool` als Typ mit den beiden Konstruktoren `true` und `false` auffaßt, so ergibt sich, daß die binäre Fallunterscheidung `if` b `then` e_1 `else` e_2 und die durch Musteranpassung `case` b `of true =>` e_1 `| false =>` e_2 definierte Fallunterscheidung mit zwei Zweigen äquivalent sind.

5 Datenabstraktion und Objektorientierung

Abstraktion ist ein fundamentales Prinzip von Programmiersprachen und Programmierung. Bisher haben wir es u.a. in der Form von Rechenvorschriften bei Berechnungen angewendet, also **Abstraktionen mittels Rechenvorschriften konstruiert**. In diesem Kapitel behandeln wir nun die Abstraktion zwei Ebenen höher. Die erste Ebene wird **Datenabstraktion** genannt. Dieser Mechanismus ist ein wesentliches Hilfsmittel bei der inkrementellen Konstruktion großer Programmsysteme. Wir geben zuerst eine knappe Einführung in das Gebiet der Datenabstraktion – zusammen mit einigen historischen Betrachtungen – und beschreiben dann etwas genauer, welche Möglichkeiten unsere Modellsprache ML bietet. Anschließend befassen wir uns noch mit den Grundkonzepten der objektorientierten Programmierung und zeigen, wie man diese in ML realisieren kann. Damit bereiten wir den Einstieg in das nächste Kapitel des Skriptums vor. Objektorientierung ist eine Abstraktionsstufe, die noch über Datenabstraktion hinausgeht. Man spricht hier auch von einer **Konstruktion von Abstraktionen mittels Objekten**. Das Thema Datenabstraktion und Objektorientierung ist natürlich viel zu umfassend, als daß wir es im Rahmen dieses Skriptums erschöpfend behandeln können.

5.1 Einige allgemeine und historische Bemerkungen

Bei der Konzeption und Programmierung von großen Programmsystemen – beim sogenannten „programming in the large"– ist es die Regel, daß gewisse Systemteile relativ unabhängig voneinander sind und deshalb auch oft von verschiedenen Personen oder Gruppen konzipiert und programmiert werden. Aus dieser Einsicht heraus erlaubte schon die erste höhere Programmiersprache Fortran die Auftrennung eines großen Programms in einzelne Teile, sogenannte **Übersetzungseinheiten**, die dann getrennt übersetzt werden konnten. Dabei erfolgte jedoch beim Zusammenwirken der einzelnen Teile noch keine Überprüfung der sogenannten **Schnittstellen**, etwa ob eine Rechenvorschrift aus Teil A mit einem Parameter des Typs m in Teil B auch mit einem passenden Term auf der entsprechenden Argumentposition aufgerufen wurde. Dies führte dazu, daß solche verschiedene Teile betreffende Fehler nur sehr schwer zu lokalisieren waren.

5.1.1 Die Grundidee der Datenabstraktion

Datenabstraktion (oft auch nur Abstraktion oder, mit Betonung einer etwas anderen Sichtweise, auch Modularisierung genannt) stellt eine Erweiterung des Prinzips der separaten Übersetzung dar.

Die Grundidee von Datenabstraktion ist die folgende: Man **gruppiert** unabhängig zu programmierende Systemteile zu abgeschlossenen Einheiten zusammen und stellt in kontrollierter Weise deren Daten und die dazugehörenden Grundoperationen zur Programmierung anderer Einheiten zur Verfügung. Weiterhin **kapselt** man die einzelnen Einheiten ab. Dies heißt, daß nur die Typbezeichnungen und die Funktionalitäten der Operationen

angegeben werden, die tatsächlichen Realisierungen aber unzugänglich sind und verborgen bleiben. ■

Diese eben noch genannte **Daten- und Operationenkapselung** – man spricht auch von dem **Geheimhaltungsprinzip** – schützt vor unerlaubten Manipulationen und erlaubt ein Auswechseln der Realisierung, etwa durch eine effizientere. Die Wirkungsweisen der Operationen auf den Daten werden implementierungsunabhängig beschrieben. Möglichkeiten dazu sind etwa aussagekräftige Namen oder ausführliche Kommentare, falls man sich in einem informellen Rahmen bewegt. Eine formale Angabe kann im Fall von Prozeduren etwa erfolgen durch Vor- und Nachbedingungen, wie wir sie im nächsten Kapitel kennenlernen werden, und im Fall von Rechenvorschriften durch logische Formeln (Sprechweisen sind hier: Axiome, Gesetze). Die letzte Form führt zum speziellen Ansatz der algebraischen Spezifikationen.

In den sogenannten **objektorientierten Programmiersprachen**, beispielsweise in Java (das wir im nächsten Kapitel einführen werden), Eiffel, C++ oder Smalltalk, wird das Prinzip der Daten- und Operationenkapselung noch erweitert, indem die Operationen syntaktisch und auch logisch den Typen zugeordnet sind. **Klassen** werden verwendet, um äquivalente Objekte zusammenzufassen, und **Vererbung** wird schließlich benutzt, um Ähnlichkeiten zwischen Klassen auszudrücken.

Bei Datenabstraktion unterscheidet man also zwischen **Spezifikation** und **Implementierung** einer Einheit; dies wird sich auch später deutlich bei den das Prinzip realisierenden Sprachmitteln von ML zeigen.

5.1.2 Einige Programmiersprachen mit Datenabstraktion

In den ersten Programmiersprachen war Datenabstraktion mit Schnittstellenüberprüfung nicht vorgesehen; das Prinzip taucht zuerst in Simula 67 als Class-Konzept auf. Datenabstraktion ist nun in sehr vielen Programmiersprachen möglich, wobei die Sprechweisen für die geschlossenen Einheiten variieren. Beispiele für verschiedene Sprechweisen sind Module (Modula-2, Euclid), Package (Ada), Cluster (Clu), Structure (ML), Class (Java) und Form (Alphard). Die Programmiersprache Pascal sieht im Standard keine Datenabstraktion vor, jedoch ist sie in den gängigen Implementierungen vorhanden. ■

Bei den eben aufgezählten Programmiersprachen ist Datenabstraktion natürlich nicht immer gleich mächtig. So erlauben etwa Ada und ML, im Gegensatz zu Modula-2, die Definition von „generischen“ Einheiten, etwa von Listen, bei denen der Elementtyp noch nicht festgelegt ist. Damit werden solche Einheiten in vielfacher Weise wiederverwendbar.

5.1.3 Erklärung: Exportation und Importation

Eine geschlossene Einheit eines großen Programmsystems repräsentiert eine Sammlung von Datentypen, Programmvariablen (im imperativen Fall), Konstanten und Operationen (Prozeduren und Rechenvorschriften).

(1) **Exportation** ist die Kennzeichnung der Programmteile, die eine Einheit nach außen zur Verwendung in anderen Einheiten zur Verfügung stellt. Diese Programmteile werden auch **sichtbare oder öffentliche Konstituenten** genannt.

(2) **Importation** ist hingegen die Kennzeichnung der Programmteile, die eine Einheit von außen benötigt.

Programmteile, die in einer Einheit deklariert, aber nicht exportiert werden, heißen auch **verborgene oder private Konstituenten**. ■

Beim Entwurf der Einheiten, die ein Gesamtsystem realisieren, ist nun zu beachten, daß kein zyklischer Import vorkommt. Um dies zu überprüfen, stellt man den sogenannten Import-Graph auf und testet, ob dieser kreisfrei ist.

5.2 Datenabstraktion in ML

Nach der Einführung in die Datenabstraktion skizzieren wir nun im folgenden die Sprachmittel zur Datenabstraktion in ML. In dieser Programmiersprache gibt es zwei Möglichkeiten zur Datenabstraktion, nämlich abstrakte Datentypen und Strukturen. Eng verbunden mit Strukturen sind Signaturen und Funktoren. Erstere dienen zur Beschreibung der Schnittstellen von Strukturen und letztere ermöglichen die Parametrisierung von Strukturen und ihr Zusammenbinden zu einem Gesamtprogramm. Wir beginnen die Beschreibung mit den abstrakten Datentypen.

5.2.1 Abstrakte Datentypen

Ein abstrakter Datentyp in ML ist ein Sprachmittel zur Definition eines Datentyps bestehend aus einem Typ sowie dazugehörenden Rechenvorschriften und Konstanten. Die Syntax eines abstrakten Datentyps sieht schematisch wie folgt aus:

```
abstype m = ≫Typausdruck≪
  with ≫Liste von Deklarationen≪ end;
```

Dabei heißt m der **abstrakte Typ** und ≫ *Typausdruck* ≪ seine **Implementierung**. Die zwischen `with` und `end` deklarierten Rechenvorschriften und Konstanten heißen das **Interface**. Exportiert wird nur das Interface, nicht jedoch die Konstruktoren des Typausdrucks. Dies erlaubt ein Auswechseln der Implementierung des Typs m und auch der entsprechenden Rechenvorschriften und Konstanten. ■

In ML gibt es keine vordefinierte Sorte für natürliche Zahlen. Ein Typ `nat` für natürliche Zahlen mit einer Konstante `zero` für die Null und einer Rechenvorschrift `succ` zur Nachfolgerbildung kann jedoch leicht eingeführt werden, wie das folgende Beispiel zeigt. Analoges gilt auch für die rationalen und komplexen Zahlen, indem man die üblichen mathematischen Definitionen als Paare in ML nachbildet.

5.2.2 Beispiel (für einen abstrakten Datentyp; natürliche Zahlen)

Wir betrachten den abstrakten Datentyp `nat` für natürliche Zahlen, definiert durch das folgende ML-Programm:

```
abstype nat = mkzero of unit | mknext of nat
  with val zero : nat = mkzero();
       fun succ (x : nat) : nat = mknext(x);
       fun add (mkzero() : nat, y : nat) : nat = y
         | add (mknext(x) : nat, y : nat) : nat = succ(add(x,y));
       fun mult (mkzero() : nat, y : nat) : nat = zero
         | mult (mknext(x) : nat, y : nat) : nat = add(mult(x,y),y);
       fun nattoint(mkzero() : nat) : int = 0
         | nattoint(mknext(x) : nat) : int = nattoint(x) + 1;
  end;
```

Dieser abstrakte Datentyp faßt eine natürliche Zahl n als Strichzahl $|||\ldots|||$ mit n Strichen auf, eine Auffassung, die in den sogenannten Peano-Axiomen auch zur induktiven Definition von $\mathbb{N}$ verwendet wird und zum Prinzip der vollständigen Induktion führt. Insbesondere hat man `mkzero()` als die leere Folge von Strichen. Durch den Aufruf `mknext(`n`)` fügt man an eine Strichzahl n genau einen Strich an, d.h. liefert die nächste natürliche Zahl.

Mit Hilfe der beiden Rechenvorschriften `mkzero` und `mknext` kann man eine Konstante `zero` für die Null und eine Rechenvorschrift `succ` für die Nachfolgeroperation durch Musteranpassung definieren, und folglich die gesamte Arithmetik der natürlichen Zahlen. Dies wird an den Beispielen `add` und `mult` der Additions- bzw. Multiplikations-Rechenvorschrift demonstriert. Die Rechenvorschrift `nattoint` bettet den Typ `nat` schließlich injektiv in die Sorte `int` ein. Man beachte, daß `mkzero` und `mknext` nicht sichtbar sind. ■

An dieser Stelle sollte bemerkt werden, daß in abstrakten Datentypen auch Typvariablen verwendet werden dürfen, d.h. die Programmiersprache ML generische abstrakte Datentypen erlaubt.

Nach den abstrakten Datentypen kommen wir nun zur zweiten Möglichkeit, in ML Datenabstraktion zu betreiben. Diese benutzt drei Sprachmittel, genannt Struktur, Signatur und Funktor. Im Gegensatz zu den abstrakten Datentypen werden die letztgenannten Sprachmittel nicht zum Kern von ML gerechnet, sondern stellen die sogenannte **Modulsprache** von ML als Erweiterung des ML-Kerns dar. Wir beginnen mit den Strukturen.

5.2.3 Strukturen

Durch eine Struktur werden in ML Deklarationen zu einer manipulierbaren Einheit (zu **Umgebungen**, engl. environments) zusammengefaßt. Ihre schematische Syntax ist:

```
structure S =
  struct ≫ Liste von Deklarationen ≪ end;
```

Dabei ist S der Name der Struktur. Im Unterschied zu den abstrakten Datentypen wird bei den Strukturen im Rahmen einer Weiterverwendung alles sichtbar gemacht, also auch die tatsächliche Darstellung der deklarierten Typen. Dies widerspricht eigentlich dem Prinzip der Datenabstraktion. Wie wir später jedoch sehen werden, ist Kapselung bei Strukturen mittels Signaturen möglich. ■

Der Zugriff auf eine Konstituente k einer Struktur S ist bei einer Importation durch die **Punktnotation** $S.k$ möglich. Analog zum Sprachkonstrukt `with` in imperativen Sprachen wie Pascal und Modula-2 zur Vereinfachung von Selektionen bei Verbunden existiert in der Sprache ML eine Abkürzung für Strukturzugriffe. Beispielsweise entspricht der normalen Schreibweise $h(S.f(S.c,A))$ die abkürzende Schreibweise

$$\texttt{let open } S \texttt{ in } h(f(c,A)) \texttt{ end}$$

mittels einer Verallgemeinerung der Objektdeklarationen, welche sich natürlich erst bei großen Ausdrücken auswirkt. Noch einfacher ist, die „Anweisung“ `open` S; zu schreiben. Dann stehen ab dieser Stelle im Gesamtprogramm die in S deklarierten Konstituenten unter den abkürzenden Schreibweisen zur Verfügung.

In ML können Strukturen natürlich wiederum in Strukturen deklariert werden, d.h. man bekommt eine hierarchische Gliederung des zu erstellenden Gesamtsystems. Das führt, wie bei den bisher in ML behandelten Deklarationen, auf die üblichen Fragen nach Gültigkeit usw, welche es bei den Zugriffen zu beachten gilt. Darauf soll jedoch im Rahmen dieses Abschnitts nicht weiter eingegangen werden.

Nachfolgend greifen wir das Beispiel der natürlichen Zahlen noch einmal auf und geben eine Implementierung von $\mathbb{N}$ als rekursiven Typ `nat` mit der Konstante und den Rechenvorschriften von Beispiel 5.2.2 als Struktur `NAT` an.

5.2.4 Beispiel (für eine Struktur; natürliche Zahlen)

Das folgende ML-Programm zeigt die Übersetzung des abstrakten Datentyps von Beispiel 5.2.2 in eine Struktur mit Namen `NAT`:

```
structure NAT =
  struct datatype nat = mkzero of unit | mknext of nat;
         val zero : nat = mkzero();
         fun succ (x : nat) : nat = mknext(x);
         fun add (mkzero() : nat, y : nat) : nat = y
           | add (mknext(x) : nat, y : nat) : nat = succ(add(x,y));
         fun mult (mkzero() : nat, y : nat) : nat = zero
           | mult (mknext(x) : nat, y : nat) : nat = add(mult(x,y),y);
         fun nattoint(mkzero() : nat) : int = 0
           | nattoint(mknext(x) : nat) : int = nattoint(x)+1;
  end;
```

Bei einer Verwendung der Struktur `NAT` in anderen Programmteilen sind, wegen der Nichtkapselung, auch Zugriffe der Formen `NAT.mkzero()` und `NAT.mknext(`x`)` möglich. Dies verletzt das Geheimhaltungsprinzip und sollte deshalb unterbunden werden. Wie dies möglich ist, wird anschließend gezeigt. ■

Weitere Beispiele für Strukturen geben wir später an. Zuerst behandeln wir jedoch die noch ausstehenden Sprachmittel der ML-Modulsprache.

5.2.5 Signaturen passend zu Strukturen

Die Programmiersprache ML ist streng getypt, d.h. jedem Sprachkonstrukt ist ein Typ zugeordnet. Der Typ einer Struktur ist nun kein ML-Typ im bisherigen Sinne, sondern die Ansammlung der Typen der in ihr deklarierten Programmteile. Diese Ansammlung wird üblicherweise Signatur genannt (vergl. mit Definition 2.3.1).

In ML kann nun eine **Signatur** A **passend zu einer Struktur** S eigens deklariert werden. Die Syntax einer Signatur sieht schematisch wie folgt aus:

```
signature A =
  sig ≫ Liste der Namen und Typen der Konstituenten von S ≪ end;
```

Eine Signatur A passend zu einer Struktur S teilt dem Benutzer nichts außer den Namen der Konstituenten von S und deren Typen mit; sie beschreibt somit die Schnittstelle des durch die Struktur gegebenen Teils des Gesamtprogramms. ■

Natürlich ist es in ML auch möglich, eine Signatur als eigenständiges Programmstück, d.h. ohne eine explizite Verbindung zu einer Struktur, zu deklarieren. Dies verwendet man, um verborgene Konstituenten festzulegen.

5.2.6 Kapselung bei Strukturen mittels Signaturen

Unter Verwendung von Signaturen erlaubt die Sprache ML auch Daten- und Operationenkapselung. Dies soll im weiteren erklärt werden. Angenommen, es sei in einem Gesamtsystem, geschrieben in der Sprache ML, eine Struktur S deklariert. Weiterhin sei in diesem System auch eine Signatur A deklariert, die nur einen Teil der Namen und ggf. zugeordneten Typen der in S deklarierten Typen, Konstanten und Rechenvorschriften enthält. Ändert man nun die (in der originalen Syntax gegebene) Deklaration von S zu

```
structure S : A =
  struct ≫ Liste von Deklarationen ≪ end;
```

ab, d.h. stellt man dem Namen durch einen Doppelpunkt abgetrennt den Signaturnamen nach, so heißt nun S eine **durch** A **beschränkte Struktur**. Von den in S deklarierten Typen, Konstanten und Rechenvorschriften sind dann nur die in A aufgeführten Konstituenten nach außen sichtbar. Der Rest bleibt verborgen.

Als Beispiel betrachten wir noch einmal die natürlichen Zahlen. Beschreibt man sie nicht durch eine unbeschränkte Signatur wie in Beispiel 5.2.4, sondern durch ein Paar

```
signature NAT_SIG =
  sig type nat;
      val zero : nat;
      val succ : nat -> nat;
      val add : nat*nat -> nat;
      val mult : nat*nat -> nat;
      val nattoint : nat -> int;
  end;

structure NAT_STRUCT : NAT_SIG =
  struct datatype nat = mkzero of unit | mknext of nat;
         val zero : nat = mkzero();
         fun succ (x : nat) : nat = mknext(x);
         fun add (mkzero() : nat, y : nat) : nat = y
           | add (mknext(x) : nat, y : nat) : nat = succ(add(x,y));
         fun mult (mkzero() : nat, y : nat) : nat = zero
           | mult (mknext(x) : nat, y : nat) : nat = add(mult(x,y),y);
         fun nattoint(mkzero() : nat) : int = 0
           | nattoint(mknext(x) : nat) : int = nattoint(x)+1;
  end;
```

bestehend aus einer Signatur und der durch sie beschränkten Struktur von Beispiel 5.2.4, so werden nun die Konstituenten `mkzero` und `mknext` verborgen. ■

Ist eine Struktur S in der eben beschriebenen Form als durch eine Signatur A beschränkt deklariert, so werden S und A durch das System verglichen und bei Inkonsistenz wird ein Fehler gemeldet. Wird dabei bei den Deklarationen in S auf die Typangaben (teilweise) verzichtet, so beachtet das ML-System die allgemeine Regel, daß der Typ einer Konstanten oder Rechenvorschrift der Struktur nicht spezieller sein darf als der der sie beschränkenden Signatur.

Bei der Verwendung von ML zur Programmierung von großen Programmsystemen zeigt sich oft, daß dabei die eigentlich zu bewältigende Aufgabe nicht die Programmierung von einzelnen Strukturen ist, sondern die Klärung ihres Zusammenwirkens, d.h. die sogenannte **Systemarchitektur**. Dieses Zusammenwirken wird durch Funktoren beschrieben, denen wir uns nun zuwenden wollen.

5.2.7 Funktoren

In ML kann aus einer Struktur mit Hilfe eines Funktors eine neue Struktur erhalten werden. Funktoren entsprechen Rechenvorschriften auf dem Niveau von Strukturen. Ihre schematische Syntax ist

$$\texttt{functor}\ F\ \texttt{(}s\ \texttt{:}\ A_1\texttt{)}\ \texttt{:}\ A_2\ \texttt{=}\ \gg \mathit{Rumpf} \ll\texttt{;}$$

mit zwei Signaturen A_1 und A_2 und einem formalen Parameter s für eine Struktur passend zu oder beschränkt durch A_1. Im Rumpf des Funktors F wird mit Hilfe von s eine Struktur definiert, die zur Signatur A_2 paßt oder durch diese beschränkt wird. ■

In ML sind, wie bei Rechenvorschriften, auch parameterlose Funktoren erlaubt. Dies heißt, daß in der eben angegebenen Syntax von Funktoren nach dem Bezeichner F das Klammerpaar `()` geschrieben wird und ein Aufruf von F dann in der Form F`()` erfolgt.

Das folgende Beispiel soll die Benutzung von Funktoren zur Parametrisierung von Strukturen in der Programmiersprache ML verdeutlichen. Wir brauchen dazu die sogenannten Ausnahmen von ML, die in der englischsprachlichen ML-Literatur **exceptions** genannt werden. Eine Ausnahme ist als **Konstante** des ausgezeichneten ML-Tys `exn` beschrieben durch einen Namen e und wird, als Erweiterung der bisher zugelassenen Deklarationsarten, mittels `exception` e`;` eingeführt. Nach so einer Deklaration ist es nun möglich, eine partielle Rechenvorschrift „künstlich“ zu totalisieren, indem man in allen Fällen, in denen bisher kein Resultat abgeliefert würde, eine Ausnahme als Resultat festlegt. Dies geschieht mittels des Terms `raise` e.

5.2.8 Beispiel (für einen Funktor)

Wir betrachten eine Datenstruktur *DICT* (eine Art „Wörterbuch“) zum Speichern von Datenbeständen. Dabei unterstellen wir, daß der Zugriff auf die eigentlich interessierende Information eines Datums durch einen sogenannten Schlüssel erfolgt und Schlüssel auf Gleichheit getestet werden können. Neben der leeren Datenstruktur ohne Datum betrachten wir als Operationen das Einfügen einer Information unter einem Schlüssel und das Auffinden einer solchen zu einem gegebenen Schlüssel.

In der Modulsprache von ML ist die eben genannte Schnittstellenbeschreibung durch die folgende Signatur `DICT` gegeben:

```
signature DICT =
  sig type key;
      type info;
      type dict;
      val empty : dict;
      val insert : key * info * dict -> dict;
      val find : key * dict -> info;
  end;
```

Eine ML-Struktur, welche durch diese Signatur beschränkt wird, besteht nun mindestens aus den Typdeklarationen für `key`, `info` und `dict`, einer Objektdeklaration für `empty` und zwei Rechenvorschriftsdeklarationen für `insert` und `find`.

Als eine spezielle Realisierung der Datenstruktur *DICT* betrachten wir nun die folgende Struktur, welche durch die Signatur `DICT` beschränkt wird. Es sollte betont werden, daß diese äußerst einfache Realisierung eines speziellen Wörtebuchs durch einen Funktionstyp

und Funktionsabstraktionen – also, semantisch gesehen, durch eine Funktion von der Schlüssel- in die Informationsmenge – nur aus Darstellungsgründen verwendet wird.

```
structure IntStringDict : DICT =
  struct type key = int;
         type info = string;
         type dict = key -> info;
         exception no_info;
         fun empty (k : key) : info =
           raise no_info;
         fun insert (k : key, c : info, d : dict) : dict =
           fn (j : key) => if j = k then c else d(j);
         fun find (k : key, d : dict) : info =
           d(k);
  end;
```

In der Praxis arbeitet man bei Wörterbüchern und ähnlichen Strukturen natürlich wesentlich effizienter, z. B. unter Verwendung von balancierten Suchbäumen oder Streuspeicherung, Begriffen, die in Informatik II im Detail behandelt werden.

Man hätte nun natürlich gerne eine Realisierung von *DICT*, die auf einfache Weise für jeden Schlüssel- und Informationstyp wiederverwendbar ist. In ML ist dies mit Hilfe der strukturabbildenden Funktoren möglich. Dazu definiert man für unser Beispiel zuerst eine Signatur für die drei Parameter Schlüsseltyp, Informationstyp und Schlüsselvergleich der Datenstruktur *DICT* wie folgt:

```
signature PARAM =
  sig type key;
      type info;
      val iseq : key * key -> bool;
  end;
```

Anschließend abstrahiert man dann die obige Struktur `IntStringDict` zu einem Funktor `Inst`, der Strukturen passend zu `PARAM` auf Strukturen beschränkt durch `DICT` abbildet oder, wie man auch sagt, **instantiiert**. In ML sieht dies dann so aus:

```
functor Inst (P : PARAM) : DICT =
  struct type key = P.key;
         type info = P.info;
         type dict = key -> info;
         exception no_info;
         fun empty (k : key) : info =
           raise no_info;
         fun insert (k : key, c : info, d : dict) : dict =
           fn (j : key) => if P.iseq(j,k) then c else d(j);
         fun find (k : key, d : dict) : info =
           d(k);
  end;
```

Der Zugriff auf eine Komponente der Parameter-Struktur `P` wird dabei, wie schon erwähnt, durch `P.key` usw. notiert.

Zu beliebigem Schlüssel- und Informationstyp ist es nun einfach, das entsprechende Wörterbuch zu erstellen. Man gibt zuerst eine konkrete Struktur für die Parameter an, etwa

```
structure IS : PARAM =
  struct type key = int;
         type info = string;
         fun iseq (j : key, k : key) : bool =
           (j = k);
  end;
```

im Fall von ganzzahligen Schlüsseln und Zeichenreihen als eigentlich interessierende Informationen, und wendet anschließend, etwa wie nachfolgend beschrieben, den Funktor `Inst` auf die Struktur `IS` im Rahmen einer sogenannten **Strukturdeklaration** an:

```
structure IntStringDict : DICT = Inst(IS);
```

Damit erhalten wir das schon oben erwähnte Wörterbuch `IntStringDICT`. Um ein Wörterbuch mit z.B. Zeichenreihen als Schlüssel und Tripel von Zeichenreihen eigentlich interessierende Informationen zu erhalten, hat man nur den Funktor `Inst` auf eine Struktur anzuwenden, die aus `IS` dadurch entsteht, daß man die Deklaration der zweiten Zeile zu `type key = string;` und die der dritten Zeile zu `type info = string * string * string;` abändert. ■

5.3 Objektorientierung

Wir haben am Anfang dieses Kapitels schon erwähnt, daß in objektorientierten Programmiersprachen das Prinzip der Daten- und Operationenkapselung noch erweitert wird. Dieser Abschnitt gibt nun eine Einführung in die grundlegendsten Begriffe der Objektorientierung. Dies geschieht in abstrakter, sprachunabhängiger Weise. Wie man den Ansatz dann in der Sprache ML umsetzen kann, wird im nächsten Abschnitt gezeigt. Zuerst führen wir die entscheidenden Konzepte „Objekt“ und „Methode“ als technische Begriffe ein.

5.3.1 Erklärung: Objekte und Methoden

Ein **Objekt** O ist ein Datum, welches

(1) einen **Zustand** besitzt, der zeitabhängig und intern (also von außen nicht sichtbar) ist, und

(2) **Aktionen** ausführen kann, die durch einen Satz von sogenannten **Methoden** beschrieben sind.

Der Zustand von O ist gegeben durch die Werte der **Attribute** von O, das sind gewisse Größen, die für O bezeichnend sind. Die in Punkt (2) genannten Aktionen sind in der Lage, **den Zustand von O zu verändern**, in Abhängigkeit vom gegebenen Zustand von O **Werte zu berechnen** und **andere Objekte zur Ausführung von Aktionen zu veranlassen**. ■

Beispielsweise kann man ein Bankkonto als Objekt ansehen. In einer sehr einfachen Darstellung ist dann der interne Zustand eines Kontos etwa durch den aktuellen Kontostand und den aktuellen Überziehungskredit gegeben. Ausführbare Aktionen eines Kontos sind beispielsweise das Einzahlen und das Abheben, welche den Zustand veränderen, und die Abfrage des Kontostandes, welche einen Wert berechnet.

Wenn ein Objekt ein anderes Objekt veranlaßt, eine Aktion auszuführen, so bedeutet dies in der Sicht der Objektorientiertheit, folgendes:

5.3.2 Erklärung: Botschaften

Eine **Botschaft** ist eine Nachricht, die von einem Objekt O_1 an ein anderes Objekt O_2 geschickt wird, um O_2 zum Ausführen einer ihrer Aktionen zu veranlassen. ■

Aufgrund des Austauschens von Botschaften taucht nun eine neue Fragestellung auf: Soll ein Objekt nach dem Versenden einer Nachricht untätig warten, bis eine Antwort eintrifft, oder, alternativ, zwischenzeitlich weiterarbeiten, d.h. weitere Aktionen ausführen? Im ersten Fall ist immer genau ein Objekt aktiv; dies ist einfach zu realisieren. Der zweite Fall führt zu einer Parallelarbeit von Objekten, was wesentlich komplizierter zu realisieren ist. Beide Fälle tauchen in objektorientierten Sprachen auf.

Betrachten wir das oben eingeführte Bankbeispiel noch einmal. Bekanntlich gibt es in einer Bank eine Unmenge von Konten. Viele davon sind gleichartig, d.h. besitzen die gleichen Attribute und den gleichen Vorrat von Aktionen. Man kann deshalb von den einzelnen Konten abstrahieren und nur mehr vom abstrakten Begriff des Kontos einer bestimmten Klasse und seinen konkreten Ausprägungen sprechen. Dies ist ähnlich wie in der Mathematik, wo man z.B. auch den abstrakten Begriff einer Gruppe $(G, \circ, e)$ untersucht und dann daraus auf die Eigenschaften von konkreten Gruppen wie $(\mathbb{Z}, +, 0)$, Permutationsgruppen, Gruppen von Drehungen usw. schließt. All dies stellt sich in der Objektorientierung wie folgt dar:

5.3.3 Erklärung: Klassen und Instanzen

Eine **Klasse** ist die Beschreibung von gleichartigen Objekten, d.h. von Objekten mit den gleichen Attributen und den gleichen Methoden. Ist O ein durch die Klasse C beschriebenes Objekt, so heißt O **Objekt der Klasse** C oder **Instanz** von C. Zu jeder Klasse gibt es immer eine Operation zum Erzeugen einer Instanz und häufig auch (nicht jedoch in Java) eine Operation zum Löschen einer Instanz.

Kann die Ausführung einer in der Klasse C beschriebenen Methode M von einem anderen Objekt veranlaßt werden, so heißt M **sichtbar** oder **öffentlich**, ansonsten **verborgen** oder **privat**. ■

Insbesondere ist also jedes Objekt Instanz einer gewissen Klasse. Weiterhin sind Objekte manipulierbare Dinge, was sie zu dem macht, was man in der englischsprachlichen Informatikliteratur **first-class citizens** nennt.

Bei Banken gibt es etwa die Klasse der Girokonten, die Klasse der Privatkonten und die Klasse der Geschäftskonten. Um ein Girokonto, ein Privatkonto oder ein Geschäftskonto zu eröffnen, wendet man die Erzeuge-Operation der jeweiligen Klasse an. Analoges gilt auch für das Schließen von Konten. Hier wird die jeweilige Lösche-Operation angewendet.

Auch der nun noch ausstehende wichtige Begriff der Objektorientierung, die Vererbung[21], läßt sich gut mit Hilfe des Beispiels der Bankkonten motivieren. Obwohl es in einer Bank durchaus verschiedene Klassen von Konten gibt, haben diese oft vieles gemeinsam. Beispielsweise kann man fast bei jedem Konto Ein- und Auszahlungen vornehmen und den Kontostand abfragen. Zusätzlich zu diesen Basismöglichkeiten besitzt ein Girokonto jedoch andere Möglichkeiten als ein Privatkonto und dieses wieder andere als ein Geschäftskonto. Vererbung ist nun ein Mechanismus, um Ähnlichkeiten von und Hierarchienen zwischen den Aktions-Möglichkeiten von Objekten auszudrücken.

5.3.4 Erklärung: Vererbung

Eine Klasse C_2 heißt eine **Unterklasse** der Klasse C_1 (oder C_1 **Oberklasse** der Klasse C_2), wenn jede sichtbare Methode von C_1 auch Methode von C_2 ist (aber dort nicht als sichtbar erklärt sein muß). Bekommt man C_2 dadurch, daß C_1 um zusätzliche Attribute und Methoden erweitert wird, so nennt man C_2 aus C_1 durch (einfache) **Vererbung** entstanden. ■

Man beachte, daß der Unterklassenbegriff nicht mit den üblicherweise in der Mathematik verwendeten Hierarchiebegriffen übereinstimmt. Unterklassenbildung heißt oft Erweiterung, während in der Mathematik mit dem Präfix „Unter-“ normalerweise Einschränkung gemeint ist. Neben der einfachen Vererbung gibt es noch die **Mehrfachvererbung**, bei der eine Unterklasse C_2 durch die Erweiterung von mehreren Oberklassen $C_1^{(1)}, \ldots, C_1^{(k)}$ entsteht.

5.4 Umsetzung des Ansatzes in ML

Nachdem im letzten Abschnitt die grundlegendsten Begriffe der Objektorientierung sprachunabhängig erklärt wurden, wollen wir nun zeigen, wie man den Ansatz in der Sprache

[21] In Büchern über Objektorientierung wird, neben den in diesem Abschnitt genannten Begriffen, in der Regel noch Polymorphie als grundlegend für den Ansatz angesehen. Polymorphie kennen wir aber schon von ML und Objektorientierung verwendet sie in sehr ähnlicher Weise.

ML mit Hilfe der Modulsprache umsetzen kann. Dazu greifen wir das bisher nur skizzierte Beispiel der Bankkonten noch einmal auf, und demonstrieren, wie man es objektorientiert in ML realisieren kann.

Zustände von Objekten sind zeitabhängig. Dies können wir bisher in ML nicht beschreiben. Aufgrund der referenziellen Transparenz steht nämlich ein Bezeichner ab seiner Deklaration bis zu einer eventuellen Neudeklaration immer für den gleichen Wert. Es gibt jedoch in ML ein Sprachmittel, die Referenzen, das von diesem Prinzip abweicht. Referenzen ermöglichen eine eingeschränkte Art von imperativer Programmierung. Wir werden sie nachfolgend nur soweit einführen, wie wir sie zur Realisierung von Objektorientierung brauchen.

5.4.1 Referenzen

Ist m ein ML-Typ, so ist auch m `ref` ein ML-Typ, genannt der Typ der **Referenzen** auf Objekte[22] des Typs m. Die Bedeutung einer Referenz r kann bildlich beschrieben werden als ein Behälter, der mit r beschriftet ist und ein Objekt des Typs m aufnehmen kann. Diese Interpretation orientiert sich am Variablenbegriff der imperativen Programmierung. Statt den Behälter mit r zu beschriften, kann man sich die Referenz r aber auch als einen mit r beschrifteten Pfeil vorstellen, der auf den Behälter zeigt. Dieser bildlichen Darstellung liegt das Konzept der Zeiger der imperativen Sprachen zugrunde.

Auf den Referenzen gibt es drei Operationen, die nachfolgend aufgezählt werden:

a) **Erzeugung:** Ist t ein Term des Typs m, so wird durch `ref` t eine anonyme Referenz erzeugt, deren Behälter den Wert von t enthält. Eine Namensgebung ist somit durch eine ML-Objektdeklaration wie folgt möglich:

$$\texttt{val}\ x\ \texttt{:}\ m\ \texttt{ref = ref}\ t\texttt{;} \qquad (*)$$

Dadurch wird, wiederum bildlich gesprochen, der Behälter mit dem Wert von t als Inhalt mit dem Bezeichner x beschriftet (bzw., in der zweiten Auffassung, der Pfeil, der auf diesen Behälter zeigt, mit x beschriftet). In Anlehnung an die Sprechweise der imperativen Programmierung nennt man einen durch $(*)$ eingeführten Bezeichner x auch **Referenzvariable**.

b) **Dereferenzierung:** Ist x ein Bezeichner des Typs m `ref`, etwa eine durch $(*)$ deklarierte Referenzvariable oder ein Parameter einer Rechenvorschrift, dann ist

$$!x$$

ein Term des Typs m und sein Wert ist definiert als der Inhalt des mit x beschrifteten Behälters. Wiederum in Anlehnung an die Sprechweise der imperativen Programmierung heißt $!x$ **Dereferenzierung** oder **Inhalt** von x.

[22] Wir verwenden hier das Wort Objekt im ursprünglichen Sinne von Kapitel 4.

c) **Modifikation:** Es sei x nochmals ein Bezeichner des Typs m `ref`. Weiterhin sei e ein Term des Typs m. Dann ist auch

$$x \texttt{ := } e$$

ein Term. Seine Sorte ist `unit` und damit sein Wert das einzige Element der Interpretation von `unit`. Bei der Berechnung dieses Werts wird als **Seiteneffekt** der Inhalt des mit x beschrifteten Behälters durch den Wert von e ausgewechselt. Statt von einer Modifikationen spricht man auch von einer **Zuweisung** x `:=` e, denn sie entspricht genau dem entsprechenden imperativen Sprachkonstrukt.

Mit Hilfe von Zuweisungen x_i `:=` e_i, $1 \leq i \leq n$, kann man eine **Anweisung** als Terme der Sorte `unit` in ML wie folgt definieren:

$$(x_1 \texttt{ := } e_1\texttt{; } \ldots \texttt{; } x_n \texttt{ := } e_n)$$

Auch bei der Berechnung des Werts solch einer Anweisung tritt ein **Seiteneffekt** auf. Er ergibt sich durch die Hintereinanderausführung der Seiteneffekte der einzelnen Zuweisungen x_1 `:=` e_1, dann x_2 `:=` e_2 usw. bis x_n `:=` e_n. ■

Durch eine Kombination der Referenzen mit den Sprachmitteln der Modulsprache von ML, d.h. mit Signaturen, Strukturen und Funktoren, sind wir nun in der Lage, Objektorientierung in ML auszudrücken. Wir beginnen mit den Objekten:

5.4.2 Realisierung von Objekten und Methoden

Ein Objekt O im Sinne der Objektorientierung kann in ML durch eine Struktur S_O realisiert werden. Die Attribute von O werden dabei durch Referenzvariablen dargestellt und die Methoden von O durch Rechenvorschriften. Um den Zustand und ggf. einige Methoden von O nicht sichtbar zu machen, braucht man ferner eine Signatur A, die S_O auf die den sichtbaren Methoden von O entsprechenden Rechenvorschriften beschränkt. ■

Wir wollen diese ML-Realisierung am Beispiel der im letzten Abschnitt skizzierten simplen Bankkonten demonstrieren.

5.4.3 Beispiel (für die objektorientierte Realisierung von Bankkonten)

Ist der interne Zustand eines Bankkontos durch den aktuellen Kontostand und den aktuellen Überziehungskredit gegeben und bestehen die ausführbaren Aktionen eines Bankkontos aus der Abfrage nach dem noch verfügbaren Geldbetrag, der Abfrage des aktuellen Kontostands, dem Setzen des Überziehungskredits, dem Einzahlen, dem Abheben und dem Einziehen von Kontoführungsgebühren, so kann man ein einzelnes Bankkonto A in ML als Struktur `A` wie nachfolgend angegeben definieren. Damit der interne Zustand, in

A dargestellt durch die Referenzvariablen st (Kontostand) und cr (Überziehungskredit), verborgen bleibt, ist A noch durch die Signatur

```
signature ACCOUNT =
  sig val available : unit -> int;
      val credit_balance : unit -> int;
      val set_credit : int -> unit;
      val deposit : int -> unit;
      val withdraw : int -> unit;
      val charge : int -> unit;
  end;
```

zu beschränken. Aus der Namensgebung in dieser Signatur ist unschwer die Zuordnung der Aktionen/Methoden zu den Rechenvorschriften zu erkennen. Die Argumentsorte unit von available und credit_balance ergibt sich dabei aus der Tatsache, daß diese zwei Rechenvorschriften nullstellig sind; die Resultatsorte unit von set_credit, deposit, withdraw und charge ergibt sich hingegen aus den Rümpfen dieser Rechenvorschriften. Insgesamt erhalten wir die Realisierung von A als ML-Struktur A wie folgt:

```
structure A : ACCOUNT =
  struct val st : int ref = ref 0;
         val cr : int ref = ref 0;
         fun available () : int =
           !st + !cr;
         fun credit_balance () : int =
           !st;
         fun set_credit (n : int) : unit =
           cr := n;
         fun deposit (n : int) : unit =
           st := !st + n;
         fun withdraw (n : int) : unit =
           if n <= !st + !cr then st := !st - n
                             else ();
         fun charge (n : int) : unit =
           st := !st - n;
  end;
```

Will man in das Bankkonto A beispielsweise 500 Geldeinheiten einzahlen, so hat man die Methode „Einzahlen“ von A mit dem Argument 500 auszuführen. In der ML-Realisierung A von A entspricht dies dem Aufruf A.deposit(500). ■

Um ein weiteres Bankkonto B anzulegen, könnte man nun den obigen Code duplizieren und dabei den Namen der Struktur zu B auswechseln. Wesentlich sinnvoller ist es aber, dem objektorientierten Ansatz folgend, von einem Bankkonto zu der Klasse der Bankkonten zu abstrahieren. Auch für diesen Schritt besitzt ML mit den Funktoren ein geeignetes Sprachmittel.

5.4.4 Realisierung von Klassen und Instanzen

Die Beschreibung einer Klasse C von gleichartigen Objekten kann in der Sprache ML erhalten werden, indem man von der Realisierung eines speziellen Objekts O von C als ML-Struktur, sagen wir mit Namen S_O, zu einem Funktor übergeht, dessen Rumpf sich direkt aus der Deklaration von S_O ergibt. Die Erzeugung einer Instanz von C als ML-Struktur ergibt sich dann durch einen Funktoraufruf in einer Strukturdeklaration. Der Normalfall führt auf Funktoren ohne Argumente. Bei der Realisierung von Klassen durch Funktoren ist aber sogar, durch die Hinzunahme von formalen Parametern in den Funktoren, eine Parametrisierung von Klassen möglich. ■

Wir wollen die eben beschriebene Vorgehensweise ebenfalls am Beispiel der simplen Bankkonten verdeutlichen.

5.4.5 Beispiel (Weiterführung von 5.4.3)

Es ist offensichtlich, wie sich die ML-Struktur `A` von Beispiel 5.4.3 für ein spezielles Bankkonto A zu einem parameterlosen Funktor `GenAccount` verallgemeinert, der die Klasse aller von uns betrachteten simplen Bankkonten beschreibt. Man hat nur die erste Zeile von `A` geringfügig abzuändern. Dies bringt:

```
functor GenAccount () : ACCOUNT =
  struct val st : int ref = ref 0;
         val cr : int ref = ref 0;
         fun available () : int =
           !st + !cr;
         fun credit_balance () : int =
           !st;
         fun set_credit (n : int) : unit =
           cr := n;
         fun deposit (n : int) : unit =
           st := !st + n;
         fun withdraw (n : int) : unit =
           if n <= !st + !cr then st := !st - n
                             else ();
         fun charge (n : int) : unit =
           st := !st - n;
  end;
```

Die Erzeugung eines speziellen Kontos, sagen wir B, als ML-Struktur `B` ist dann durch einen Aufruf von `GenAccount` in Verbindung mit der Strukturdeklaration

```
structure B : ACCOUNT = GenAccount();
```

zur Namensgebung möglich. Nach dieser Erzeugung kann man nun beliebige Anweisungen ausführen, die den internen Zustand des Bankkontos B ändern, beispielsweise:

```
(B.set_credit(1000);
 B.deposit(500);
 B.deposit(300);
 B.withdraw(200) )
```

Damit wird zuerst der Überziehungskredit festgelegt. Dann werden zwei Einzahlungen gefolgt von einer Auszahlung vorgenommen. Wertet man nach der Ausführung dieser Anweisung die Terme `B.available()` bzw. `B.credit_balance()` aus, so bekommt man die Resultate `1600` (noch verfügbarer Geldbetrag) bzw. `600` (Kontostand). ■

Als letzten Punkt wollen wir schließlich noch auf die Modellierung von Vererbung in der Sprache ML eingehen.

5.4.6 Realisierung von Vererbung

Will man eine Unterklasse C_2 durch Vererbung aus einer Oberklasse C_1 gewinnen, und ist die Klasse C_1 in ML durch einen Funktor realisiert, so kann man Vererbung dadurch modellieren, daß man den zu C_1 gehörenden Funktor bei der Konstruktion des Funktors zu C_2 aufruft. ■

Auch dieses Vorgehen wird am Beispiel der simplen Bankkonten demonstriert.

5.4.7 Beispiel (Weiterführung von 5.4.5)

Zur Verdeutlichung betrachten wir nun ein spezielles Bankkonto für Studierende (oder Schüler). Für solche speziellen Konten verlangen die Banken in der Regel keine Kontoführungsgebühren. In ML kann man etwa mittels

```
functor GenStudAccount () : ACCOUNT =
  struct structure A : ACCOUNT = GenAccount();
         fun available () : int =
           A.available();
         fun credit_balance () : int =
           A.credit();
         fun set_credit (n : int) : unit =
           A.set_credit(n);
         fun deposit (n : int) : unit =
           A.deposit(n);
         fun withdraw (n : int) : unit =
           A.withdraw(n);
         fun charge (n : int) : unit =
           A.charge(0);
  end;
```

die Konstruktion eines „Studierenden-Kontos“ aus einem beliebigen Konto durch einen Funktor ausdrücken. Bis auf die Rechenvorschrift `charge` entsprechen alle Konstituenten des Rumpfs dieses Funktors den Konstituenten von `GenAccount`. Nur `charge` ist umdefiniert; ein Aufruf dieser Rechenvorschrift hat nun keinerlei Auswirkung auf den internen Zustand. An Stelle von `A.charge(0)` hätte man natürlich auch `()` als Rumpf von `charge` wählen können. ■

Bei der Konstruktion des Studierenden-Kontos aus einem beliebigen Konto handelt es sich um eine **Spezialisierung mittels Vererbung**, was ein sehr häufig auftretender Fall ist. Zwei weitere häufige Fälle von Vererbung sind noch **verhaltensgleiche Vererbung**, bei der die Unterklasse alle Attribute und Methoden der Oberklasse ohne Einschränkungen übernimmt, und **implementierende Vererbung**, bei der die Unterklasse die Konzepte der Oberklasse darstellt. Ein Beispiel für das Letzte ist etwa die Implementierung einer (Unter-)Klasse für Mengen durch eine (Ober-)Klasse für lineare Listen.

In ML ist es durch das `include`-Sprachkonstrukt sehr einfach möglich, eine bei einer Vererbung anfallende Erweiterung einer Signatur A_1 zu A_2 syntaktisch zu formulieren. Man hat A_2 wie folgt zu deklarieren:

```
signature A2 =
  sig include A1;
      ≫ Liste der zusätzlichen Konstituenten ≪
  end;
```

Auf der Ebene der Strukturen (und damit auch der Funktoren) gibt es ein analoges Sprachkonstrukt, nämlich das schon erwähnte `open`. Eine Erweiterung wird hier in der Regel, wie anhand von Beispiel 5.4.7 gezeigt, mit Hilfe einer Strukturdeklaration bewerkstelligt. An Stelle der Redeklarationen der „alten“ Rechenvorschriften und der Deklarationen der „neuen“ Rechenvorschriften öffnet man jedoch nun die deklarierte Struktur und deklariert dann die geänderten und neuen Rechenvorschriften. Damit werden die aus der Struktur stammenden Rechenvorschriften überschrieben. Auf diese Weise ergibt sich

```
functor GenStudAccount () : ACCOUNT =
  struct structure A : ACCOUNT = GenAccount();
         open A;
         fun charge (n : int) : unit =
           ();
  end;
```

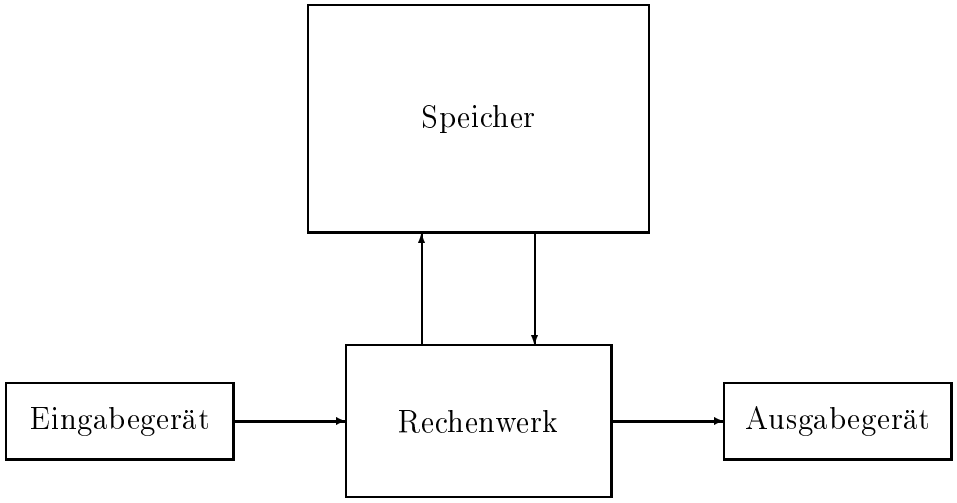
als vereinfachte Form des obigen Funktors `GenStudAccount` zur Erzeugung von Studierendenkonten.

6 Grundkonzepte imperativer Programmierung

Wie schon in der Einleitung von Kapitel 3 erwähnt, teilen sich die Programmiersprachen auf in die imperativen und die deklarativen Sprachen. Bisher haben wir uns mit der wichtigsten Teilklasse der deklarativen Sprachen beschäftigt, den funktionalen Programmiersprachen[23]. Wir behandelten dieses Thema anhand unserer Modellsprache ML. Das letzte technische Kapitel dieses Skriptums ist nun der Einführung in die imperative Programmierung gewidmet. Imperative Programmierung wird in der Praxis derzeit der funktionalen Programmierung – hauptsächlich aus Effizienzgründen – immer noch vorgezogen. Als Modellsprache für imperative Programmierung wählen wir die Programmiersprache Java, eine Entwicklung der 90er Jahre des letzten Jahrhunderts, welche den Ansatz der Objektorientierung sehr konsequent verfolgt.

6.1 Grundlegendes

Programmiersprachen sind Kunstsprachen zur Formulierung von Algorithmen, die auf einem Rechner abgearbeitet werden. Obwohl es verschiedene Typen von Rechnern gibt, ist die heute vorherrschende **Rechnerarchitektur** immer noch die auf **J. von Neumann zurückgehende Architektur**. Eine ganz grobe Vorstellung von ihr gibt das folgende Bild, welches heutzutage aber eher von historischem Interesse ist:

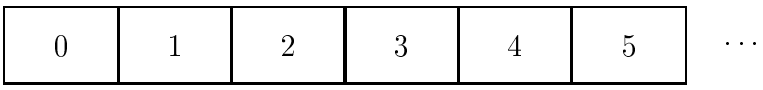


Ein Rechenwerk wird von einem Eingabegerät (z.B. einer Tastatur) mit Daten versorgt, verarbeitet diese mit Hilfe eines Speichers und übermittelt die Resultate an ein Ausgabegerät (z.B. einen Drucker oder Bildschirm). Dabei kann der Speicher sowohl Daten als auch Programme aufnehmen.

Eine sehr simple, aber dennoch auch sehr brauchbare Vorstellung von einem **Speicher**

[23] Es sollte an dieser Stelle erwähnt werden, daß, neben den funktionalen Programmiersprachen, noch weitere deklarative Sprachklassen praktisch relevant sind. Es sei an dieser Stelle insbesondere die weitverbreitete logische Programmiersprache Prolog erwähnt.

ist die eines sogenannten **linearen Speicherbandes** mit numerierten Feldern, wie es nachfolgend bildlich dargestellt ist:



Die Zahlen heißen **Adressen**, in einem mit einer Adresse versehenen Feld kann ein Datum (z.B. eine Zahl) untergebracht werden. Statt Feld sagt man oft auch **Speicherzelle**.

Imperative Programmiersprachen orientieren sich genau an den eben gezeichneten Bildern. Ein wesentlicher Gedanke dabei ist, daß eine Speicherzelle nicht nur ein Datum aufnehmen kann, sondern in ihr ein **früheres Datum auch sehr einfach ersetzt** werden kann. Höhere und modernere imperative Sprachen verwenden auch keine expliziten Adressen mehr, sondern symbolische Bezeichner. Ein solcher Bezeichner wird gewöhnlich auch **Variable** genannt.

Der Mechanismus zur Ersetzung eines Datums in der der Variablen x zugeordneten Speicherzelle durch ein neues Datum, welches sich durch Auswerten eines Terms t ergibt, ist die **Zuweisung**. Diese haben wir bei den ML-Referenzen schon kennengelernt. In Java hat sie x `=` t`;` als syntaktische Form.

Eine Zuweisung ist ein Spezialfall einer **Anweisung**, wie wir ebenfalls von ML her schon wissen. Anweisungen sind die Bausteine, aus denen imperative Programme in der einfachsten Art aufgebaut sind. Sie verändern durch ihre Abarbeitung bestimmte Zellen des Speichers, also den **Speicherzustand**. Spezielle Anweisungen dienen auch zum Einlesen und/oder Ausgeben von Daten. Ausgeben verändert dabei den Speicherzustand nicht.

6.1.1 Beispiel (Kegelstumpf)

Wir wollen, analog zu Beispiel 3.3.2.a, das Volumen eines Kegelstumpfes mit Höhe h, kleinem Radius r und großem Radius R berechnen, dabei die Formel

$$V = \frac{\pi * h}{3} * (r^2 + r * R + R^2)$$

jedoch zur Gewinnung eines imperativen Programms benutzen. So ein Programm hat drei Schritte auszuführen. Diese sind nachfolgend angegeben:

(1) Einlesen der Daten r, R, h.

(2) Berechnung von V nach obiger Formel.

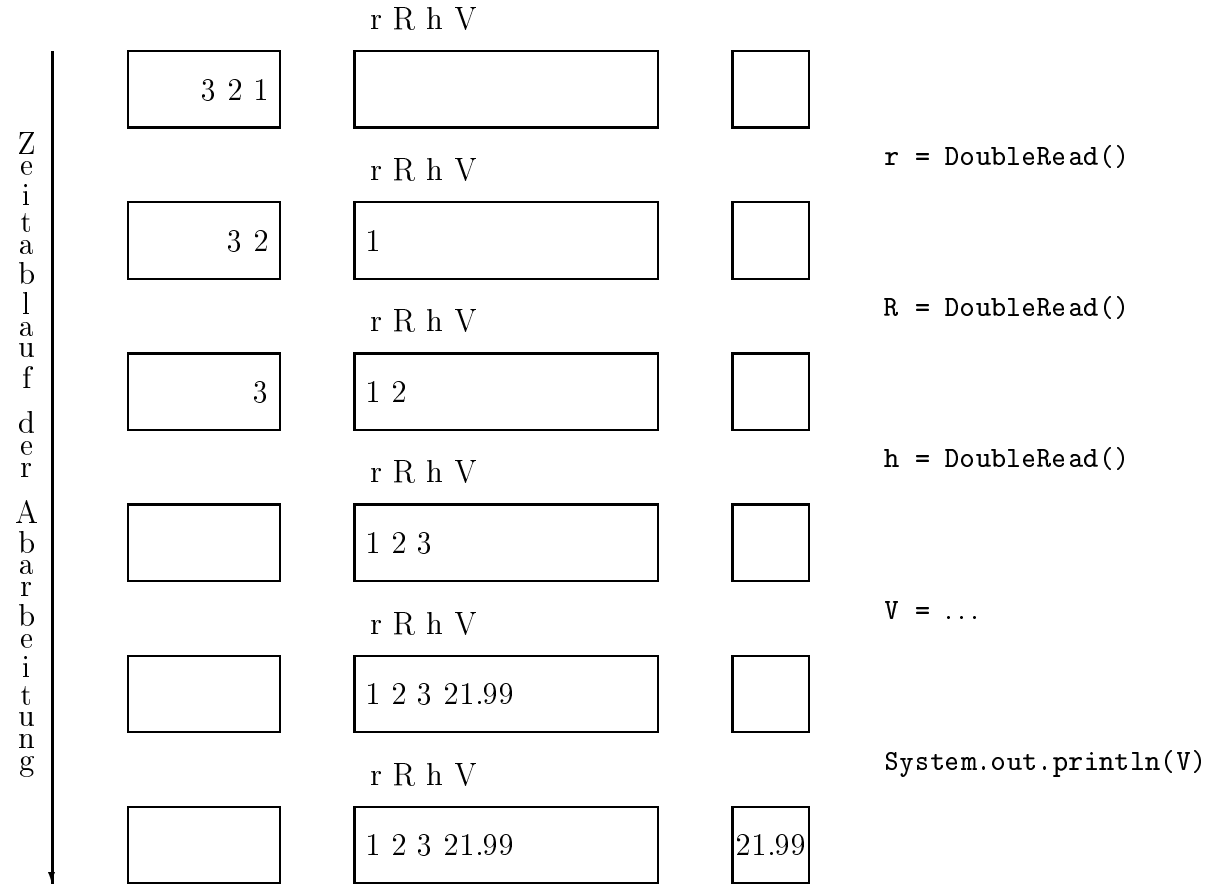
(3) Ausdrucken von V.

Wir benötigen also vier Speicherzellen, die wir durch die Variablen `r`, `R`, `h` und `V` bezeichnen. Dann kann man die Schritte (1), (2) und (3) wie folgt als Folge von vier Zuweisungen

und einer Ausgabe-Anweisung in der Syntax der Sprache Java formalisieren:

```
r = DoubleRead();
R = DoubleRead();
h = DoubleRead();
V = 1.0472 * h * (r*r + r*R + R*R);
System.out.println(V);
```

Diese Folge von Anweisungen stellt noch kein komplettes Java-Programm sondern nur einen Teil davon dar. Die ersten drei Zuweisungen lesen die Daten ein und speichern sie in den mit `r`, `R` bzw. `h` bezeichneten Speicherzellen ab, die vierte Zuweisung berechnet das Volumen und speichert es in der Speicherzelle mit dem Namen `V` ab, und die abschließende Ausgabe-Anweisung druckt den Inhalt dieser Speicherzelle. Bei einer auf dem Eingabegerät anstehenden Eingabe 1 (für den kleinen Radius r), 2 (für den großen Radius R) und 3 (für die Höhe h) führt ihre Abarbeitung zu der folgenden Sequenz von noch anstehender Eingabe, aktuellem Speicherzustand und schon gedruckter Ausgabe:



Dabei haben wir 21.99 statt dem genauen Wert von $21.9912 = 1.0472 * 3 * (1 + 2 + 4)$ für V geschrieben, wobei 1.0472 wiederum $\frac{\pi}{3}$ approximiert. ■

Bisher haben wir idealisierterweise angenommen, daß eine Variable genau einer Speicherzelle entspricht. In der Praxis ist dies natürlich nicht so, da ein zu speicherndes Datum,

etwa eine lineare Liste oder ein Binärbaum, welche wir beide von ML her schon kennen, zu komplex für eine einzelne Speicherzelle sein kann. In diesem Fall bezeichnet eine Variable eine Menge von Speicherzellen oder einen Verweis (eine Referenz) auf eine solche Menge. Genaueres zur Organisation des Speichers erfährt man beispielsweise in einer Vorlesung über Rechnerorganisation oder Betriebssysteme. Nichtsdestoweniger hält man an der Sprechweise fest, daß ein Datum D in (oder unter) einer Variablen x abgespeichert ist. Synonym dazu ist auch die Sprechweise „D ist der **Wert** oder der **Inhalt** von x“, die wir ebenfalls schon von ML her kennen.

6.1.2 Die imperative Programmiersprache Java

Wie schon in der Einleitung zu diesem Kapitel erwähnt, werden wir in diesem Skriptum Java als imperative Programmiersprache verwenden. Die Folge von Anweisungen aus Beispiel 6.1.1 stellt, wie schon erwähnt, einen Teil eines Java-Programms dar. Vervollständigen wir sie, so erhalten wir das folgende Java-Programm:

```
public class cone {
  private static double DoubleRead () {
    String s = "";
    try {s = new java.io.DataInputStream(System.in).readLine();}
    catch(java.io.IOException e) {;}
    return java.lang.Double.parseDouble(s); }

  public static void main (String[] args) {
    double R, r, h, V;
    R = DoubleRead();
    r = DoubleRead();
    h = DoubleRead();
    V = 1.0472 * h * (r*R + r*R + R*R);
    System.out.println(V); }
  }
```

Dieses Programm besteht aus einer Klasse `cone`, in der zwei Methoden `DoubleRead` und `main` deklariert werden. Die erste Methode dient, unter Verwendung einer Ausnahmebehandlung mittels `try` und `catch`, zum Einlesen von reellen Zahlen, die doppelt so genau dargestellt werden, wie üblicherweise reelle Zahlen in Java dargestellt werden. Aus diesem Grund wurde die entsprechende Sorte in Java `double` genannt. Die zweite Methode ist die Hauptmethode, deren Ausführung die eigentliche Arbeit leistet.

Die Sprache Java wurde ab dem Jahr 1991 von Mitarbeitern der Firma Sun Microsystems entwickelt. Zuerst sollte nur eine Sprache zur Programmierung von elektronischen Geräten der Konsumgüterindustrie (Recorder, Toaster usw.) geschaffen werden. Die Änderung der Zielrichtung hin zu einer allgemeinen objektorientierten Programmiersprache, die bewährte Konzepte anderer Sprachen (wie Java, C++ und Smalltalk) in einer bisher nicht bekannten Kombination zusammenfaßt, erfolgte 1993. Im Jahr 1995 wurde Java der Öffentlichkeit vorgestellt. Seit dieser Zeit bietet Sun den Kern eines Java-Programmiersystems

(genannt JDK als Kurzform von „Java Development Kit“) zusammen mit einer Implementierung des Java-Interpreters (der „Java Virtual Machine“, kurz JVM) kostenlos an. Die meisten Java-Systeme benutzen JDK als Kernsystem. ■

In der Praxis hat sich die Sprache Java erstaunlich rasch durchgesetzt, insbesondere wegen der Plattformunabhängigkeit und der „Internet-Unterstützung“. Für die Integration von Java-Anwendungen in Internet-Seiten – man spricht hier von sogenannten **Applets** – besitzt die Sprache spezielle Sicherheitsmechanismen. Mittlerweile ist Java auch an Universitäten sehr verbreitet und wird auch als Einführungssprache in die imperative Programmierung verwendet. Der Vorteil dieser Vorgehensweise ist, daß man dadurch später nahtlos zu ausgereifteren Konstruktionen und Programmiertechniken übergehen kann, etwa zu objektorientierter und verteilter Programmierung, und dabei nicht die Programmiersprache wechseln muß. Ein Nachteil ist aber auch gegeben: Jedes Java-Programm hat objektorientiert zu sein. Damit werden jedoch die üblicherweise in einer Anfängervorlesung gebrachten kleineren Beispiele in der Regel komplexer als z.B. die entsprechenden Pascal-Versionen, da sie Konstruktionen verwenden, die für die geforderten Aufgabenstellungen eigentlich unnötig und für einen Anfänger in der Programmierung manchmal auch schwer zu verstehen sind.

Im folgenden betrachten wir die Grundstruktur von den Java-Programmen, wie sie uns bis zum Ende des Skriptums begegnen werden. Weitere Details werden wir zu späteren Zeitpunkten nachschieben. Eine Vertiefung der Java-Programmierung wird in der Vorlesung Informatik II bei der Implementierung von effizienten Datenstrukturen und Algorithmen erfolgen.

6.1.3 Klassen als die zentralen Elemente von Java-Programmen

Klassen sind die zentralen Elemente der objektorientierten Programmierung und jedes Java-Programm besteht aus einer Liste von Klassendeklarationen.

Die Syntax einer einzelnen **Klassendeklaration ohne Vererbung** sieht dabei in Java schematisch wie folgt aus:

$$\texttt{class}\ C\ \texttt{\{}\ \gg \textit{Liste von Deklarationen} \ll\ \texttt{\}}$$

Dabei ist C der **Name** der Klasse und die Liste der Deklarationen wird auch der **Rumpf** der Klasse genannt.

Mit Hilfe von zusätzlichen **Modifizierern** vor dem Schlüsselwort `class` können Eigenschaften der Klasse C festgelegt werden. Wir brauchen im folgenden nur den Modifizierer `public`, welcher besagt, daß die Klasse C sichtbar ist. Enthält eine Datei einen Teil eines großen Java-Programms, welcher aus der Deklaration von mehreren Klassen $C_1, \ldots, C_n$ besteht, so darf höchstens eine Klasse C_i als sichtbar erklärt sein und der Name der Datei muß dann C_i`.java` lauten. Sichtbarkeit bei Klassen ermöglicht diese in sogenannten Paketen zu verwenden. Pakete dienen zur Modularisierung. Sie werden in dieser Vorlesung aber nicht behandelt, Auf eine allgemeine Erklärung zur Sichtbarkeit von Klassen in Paketen müssen wir deshalb leider verzichten. ■

Der Rumpf einer Klasse C besteht aus Deklarationen von Variablen, welche den Attributen der durch C beschriebenen Objekte entsprechen, und von Methoden. Auf beide Konstruktionen werden wir später noch genauer eingehen. Jedoch wird vieles hoffentlich auch schon im Laufe dieses Abschnitts durch das vorhergehende Beispiel und die noch folgenden Beispiele klar, insbesondere, wenn man letztere mit den früheren ML-Programmen zu den gleichen Problemstellungen vergleicht.

6.1.4 Vererbung in Java

Die Programmiersprache Java besitzt einige einfache Sprachmittel, um Vererbung auszudrücken. In diesem Skriptum verwenden wir nur die folgende Konstruktion:

`class` C_2 `extends` C_1 `{` $\gg$ *Liste von Deklarationen* $\ll$ `}`

Dadurch entsteht die (neue) Unterklasse C_2 aus der schon deklarierten Oberklasse C_1, indem C_1 um die innerhalb der geschweiften Klammern aufgeführten Attribute/Variablen und Methoden erweitert wird. Modifikatoren vor `class` sind natürlich auch hier erlaubt.

In Java ist es erlaubt, in einer Klasse Methoden mit dem gleichen Namen zu definieren, sofern sie sich in den Funktionalitäten unterscheiden. Bei der Vererbung mittels der eben angegebenen Konstruktion wird eine Methode M der Oberklasse C_1 nur dann überschrieben, wenn in der Liste der neuen Deklarationen ebenfalls eine Methode M der gleichen Funktionalität auftaucht. Ansonsten existieren in der Unterklasse C_2 zwei Methoden mit dem gleichen Namen M, nämlich die Originalversion aus C_1 und die zusätzlich neu deklarierte. Welche der beiden Methoden dann bei einem Aufruf genommen wird, kann erst bei der Übersetzung der Klassen festgestellt werden, die diese Methoden verwenden. Die Wahl ergibt sich dann aus den jeweils vorliegenden konkreten Argumenten[24]. Man nennt in der Literatur die eben beschriebene Vorgehensweise **späte Bindung**. Manchmal spricht man auch von dynamischer Bindung, weil sie Ähnlichkeit hat mit der bei den ML-Objektdeklarationen erwähnten alternativen Vorgehensweise. ■

Eine Überschreibung der Methode M bei der Konstruktion `class` C_2 `extends` C_1 ... kann übrigens verhindert werden, wenn M in C_1 mit dem Modifizierer `final` versehen wird. Sie hat dann, wie man sagt, ihre **finale oder endgültige Implementierung** erhalten. Im nachfolgenden Beispiel demonstrieren wir nun die Bildung von Klassen und Unterklassen mittels Vererbung in Java, indem wir die objektorientierte Realisierung von Bankkonten des Abschnitts 5.4 noch einmal aufgreifen.

6.1.5 Beispiel (Weiterführung der Beispiele 5.4.5 und 5.4.7)

In Beispiel 5.4.5 hatten wir eine Klasse zur Beschreibung von Bankkonten mit dem aktuellen Kontostand und dem aktuellen Überziehungskredit als internen Zustand und Metho-

[24] Genaugenommen ist also nicht die gesamte Funktionalität entscheidend, denn das Ergebnis bleibt unberücksichtigt. Es kommt nur darauf an, daß man anhand der Eingaben entscheiden kann, welche Methode zu wählen ist. Dies genau ist auch die von Java geforderte Eigenschaft.

den zur Abfrage des noch verfügbaren Geldbetrags bzw. des aktuellen Kontostands, zum Setzen des Überziehungskredits, zum Einzahlen, zum Abheben und zum Einziehen von Kontoführungsgebühren als ML-Funktor realisiert. In Java wird diese Klasse, mit dem Namen `account`, wie folgt deklariert:

```
class account {
  private int st;
  private int cr;
  public int available () {
    return st + cr; }
  public int credit_balance () {
    return st; }
  public void set_credit (int n) {
    cr = n; }
  public void deposit (int n) {
    st = st + n; }
  public void withdraw (int n) {
    if (n <= st + cr) st = st - n; }
  public void charge (int n) {
    st = st - n; }
  }
```

Dabei sind die den inneren Zustand realisierenden Variablen `st` und `cr`, wegen der Verwendung des Modifizierers `private`, außerhalb von `account` nicht sichtbar. Aufbauend auf diese Klasse läßt sich nun eine Klasse `studaccount` zur Beschreibung von Studierendenkonten ohne Kontoführungsgebühren in Java wie folgt gewinnen:

```
class studaccount extends account {
  public void charge (int n) {;}
  }
```

Damit wird in `studaccount` eine neue Methode namens `charge` deklariert; die alte Methode von `account` wird, da sich die Funktionalität nicht ändert, überschrieben.

Die Erzeugung von Instanzen einer Klasse in der Programmiersprache Java erfolgt durch den `new`-Operator in Kombination mit einem **Konstruktor**, beispielsweise in der Form

```
account B = new account();
```

zur Erzeugung eines Objekts `B` der Klasse `account`. Dies entspricht genau der ML-Strukturdeklaration `structure B : ACCOUNT = GenAccount()` von Beispiel 5.4.5. Die an diese Deklaration sich anschließende ML-Anweisung schreibt sich in Java wie folgt:

```
B.set_credit(1000);
B.deposit(500);
B.deposit(300);
B.withdraw(200);
```

Um den nach der Ausführung dieses Programmstücks noch verfügbaren Geldbetrag bzw. den Kontostand zu bekommen, was in ML durch die Auswertung von Termen erfolgte, kann man in Java beide Werte mittels der Ausgabe-Anweisungen

```
System.out.println(B.available());
System.out.println(B.credit_balance());
```

am Bildschirm ausgeben. ■

Konstruktoren dienen dazu, zusammen mit dem Operator `new` Instanzen einer Klasse – eventuell auch mit einem gewissen Anfangszustand – zu erzeugen. Sie tragen immer den Namen der Klasse, also etwa in Beispiel 6.1.5 `account` zum Erzeugen einer Instanz der Klasse `account`. Für jede Klasse C stellt Java einen parameterlosen Standard-Konstruktor bereit, dessen Aufruf in `new` C`()` einen speziellen Anfangszustand festlegt (gegeben durch die Initialisierung von neu deklarierten Variablen; man vergleiche mit Abschnitt 6.3) und entsprechenden Speicherplatz reserviert. Konstruktoren mit Parametern müssen hingegen von Benutzer in der entsprechenden Klasse eigens deklariert werden. Erweitert man etwa die Klasse `account` von Beispiel 6.1.5 um den Konstruktor

```
account (int n) { cr = n; },
```

so wird durch `account B = new account(1000);` eine Instanz `B` von `account` erzeugt, dessen Überziehungskredit mit 1000 Geldeinheiten initialisiert ist. Wir wollen nicht näher auf Konstruktoren eingehen, da sie in dieser Vorlesung nicht mehr verwendet werden. Ihre Bedeutung wird sich erst in Informatik II ergeben.

An dieser Stelle ist noch eine wichtige Bemerkung zur praktischen Ausführung von Java-Programmen angebracht. Etwas genauer gehen wir auf die Benutzung des Java-Systems in Abschnitt 6.5 ein.

6.1.6 Bemerkung (zur Ausführung von Java-Programmen)

Wir gehen davon aus, daß das gesamte Java-Programm in eventuell mehreren Dateien untergebracht ist, in jeder Datei genau eine Klasse als sichtbar erklärt ist und alle Dateien übersetzt sind. Eine sichtbare Klasse wird zur **Hauptprogramm-Klasse** erklärt, indem sie eine Methode der folgenden Form enthält:

```
public static void main (String[] args) { ... }
```

Hier ist `main` der fest vorgeschriebene Name. Die – ebenfalls fest vorgeschriebenen – beiden Modifizierer `public` und `static` der Methode besagen, daß `main` sichtbar ist bzw. verwendet werden kann, ohne daß vorher eine Instanz der Hauptprogramm-Klasse gebildet wurde. Mittels der Resultatangabe `void` wird schließlich angezeigt, daß `main` keinen Wert berechnet, sondern nur, ggf. unter Verwendung von Ein- und Ausgabe, den Speicherzustand verändert. Auch das Argument – es handelt sich hier um ein sogenanntes Feld von Zeichenreihen, eine Datenstruktur, die in Informatik II behandelt wird – ist fest vorgeschrieben. Warum, das können wir an dieser Stelle leider nicht erklären.

Der die Hauptprogramm-Klasse enthaltende Teil des gesamten Programms heißt das **Hauptprogramm** und die Ausführung des Gesamtprogramms wird mit der Ausführung des Hauptprogramms gestartet. ■

Im Fall von Beispiel 6.1.5 hat man also beispielsweise die Deklaration der Klasse `account` um die Hauptprogramm-Klasse

```
public class bank {
  public static void main (String[] args) {
    account B = new account();
    B.set_credit(1000);
    B.deposit(500);
    B.deposit(300);
    B.withdraw(200);
    System.out.println(B.available());
    System.out.println(B.credit_balance()); }
  }
```

zu ergänzen, um, sofern sich alles in einer Datei `bank.java` befindet, ein ausführbares Java-Programm zu erhalten.

Stehen hingegen die beiden Klassen `account` und `bank` in zwei verschiedenen Dateien namens `account.java` bzw. `bank.java`, so sollte die Klasse `account` noch durch den vorangestellten Modifizierer `public` als sichtbar erklärt werden. Ohne diesen Modifizierer wird diese Klasse nämlich bei einer Modularisierung mittels Paketen als nicht sichtbar angenommen.

6.2 Elementare Datenstrukturen

Wie im Fall der Programmiersprache ML gibt es auch für Java eine Basis-Signatur $\Sigma = (S, K, F)$, die die vorimplementierten Sorten, Konstantensymbole (wir sagen wiederum kurz: **Konstanten**) und Funktionssymbole (kurz: **Operationen**) zur Verfügung stellt, also die elementaren Datenstrukturen syntaktisch festlegt. Analog ist auch die Semantik der elementaren Datenstrukturen durch die Wahl einer speziellen Σ-Algebra A gegeben, so daß die Menge m^A genau dem entspricht, was durch die Sorte $m \in S$, genauer deren Namen, ausgedrückt wird. So ist etwa durch die Bezeichnung `double` die (idealisierte) Zuordnung $\texttt{double}^A = \mathbb{R}$ in dem ersten Java-Programm dieses Abschnitts gemeint. (Warum `double` und nicht `real` als Name gewählt ist, wird im Laufe dieses Abschnitts noch klar werden.)

Im folgenden geben wir die elementaren Datenstrukturen von Java (ausschnittsweise) an. Man beachte jedoch, daß, entgegen zum Fall von ML, in Java **für jede elementare Datenstruktur** der Test auf **Gleichheit** `==` : m, m `->` `boolean` und auf **Ungleichheit** `!=` : m, m `->` `boolean` zusätzlich zu den nachfolgenden Operationen existieren. Auch beachte man die im Vergleich zu ML veränderte Schreibweise. Die Verwendung des „üblichen“

Gleichheitssymbols bei den Zuweisungen wurde (leider) von der Programmiersprache C übernommen und ist eine häufige Fehlerquelle bei Java-Anfängern.

Wir beginnen mit der Datenstruktur der Wahrheitswerte, wo sich, im Vergleich zur entsprechenden Datenstruktur in ML, nur einige Bezeichnungen ändern.

6.2.1 Datenstruktur der Wahrheitswerte

Sorte:	`boolean`	Wahrheitswerte $\mathbb{B}$
Konstanten:	`true : boolean`	Symbol für *tt*
	`false : boolean`	Symbol für *ff*
Operationen:	`! : boolean -> boolean`	Negation
	`&& : boolean,boolean -> boolean`	Sequentielle Konjunktion
	`|| : boolean,boolean -> boolean`	Sequentielle Disjunktion ■

Die Schreibweise für die Negation `!` ist also Präfix-Notation. Für sequentielle Konjunktion `&&` und sequentielle Disjunktion `||` sieht Java ebenfalls eine Infix-Schreibweise vor. Mit Ausnahme dieser beiden Operationen sind, wie in ML, die restlichen Operationen der elementaren Datenstrukturen strikt.

Als nächstes behandeln wir eine Datenstruktur für ganze Zahlen in der 32-Bit-Darstellung. Neben ihr gibt es noch zwei andere Java-Datenstrukturen für ganze Zahlen mit den Sorten `short` (für eine 16-Bit-Darstellung) bzw. `long` (für eine 64-Bit-Darstellung), welche die gleichen Operationen wie die nachfolgende Datenstruktur besitzen. Der Konstantenvorrat verringert bzw. vergrößert sich natürlich entsprechend.

6.2.2 Datenstruktur für ganze Zahlen

Sorte:	`int`	Ganze Zahlen $\mathbb{Z}$
Konstanten:	`0, 1, ...`	Symbole für 0, 1 ...
Operationen:	`- : int -> int`	Unäres Minus
	`+, -, * : int,int -> int`	Arithmetik
	`/ : int,int -> int`	Ganzzahlige Division
	`% : int,int -> int`	Rest bei ganzz. Division
	`<, <=, >, >= : int,int -> boolean`	Vergleichsoperationen ■

Im Vergleich zur elementaren Datenstruktur der ganzen Zahlen in ML ändert sich bei der Datenstruktur der ganzen Zahlen in Java fast nichts. Die Schreibweisen bleiben gleich und die Semantik der entsprechenden Operationen ist ebenfalls gleich. Es fehlen nur die den ML-Operationen `min` und `max` entsprechenden Java-Operationen.

Für reelle Zahlen gibt es zwei Java-Datenstrukturen, welche den gleichen Operationsvorrat besitzen und sich nur in den Sortenbezeichnungen und den Konstantenvorräten unterscheiden. Hier ist die Datenstruktur mit der Sorte `double`, die wir bereits kennen. In ihr werden Gleitkommazahlen durch 64 Bit dargestellt.

6.2.3 Datenstruktur für reelle Zahlen

Sorte:	`double`	Reelle Zahlen $\mathbb{R}$
Konstanten:	Alle Gleitkommadarstellungen	Z.B. `1`, `2.0`, `2e-3`
Operationen:	`- : double -> double`	Unäres Minus
	`+, -, *, / : double,double -> double`	Arithmetik
	`<, <=, >, >= : double,double -> boolean`	Vergleichsoperat. ■

Wie das Beispiel `2e-3` andeutet, erlaubt die Gleitkommadarstellung von Java, im Gegensatz zu der von ML, auch, daß die Mantisse eine ganze Zahl (ohne Nachkommateil) ist. Wird in der eben angegebenen Datenstruktur die Sorte `float` statt `double` verwendet, so hat man es mit einer Datenstruktur zu tun, die Gleitkommazahlen durch 32 Bit dargestellt und entsprechend weniger Konstanten besitzt.

Im Vergleich zu den elementaren Datenstrukturen für ganze Zahlen wird bei den Datenstrukturen für reelle Zahlen die ganzzahlige Division durch die übliche Division ersetzt und die Restbildung bei der Division entfällt. Die Schreibweisen sind die in der Mathematik gebräuchlichen, d.h. bis auf das präfix-notierte einstellige Minus werden alle Operationen in Infix-Schreibweise geschrieben.

Vergleicht man die Java-Operationen auf den reellen Zahlen mit denen von ML, so fällt auf, daß die Funktionen der Analysis für $\sqrt{x}, sin(x)$ usw. nicht Teil der Basis-Signatur von Java sind. Dem liegt die Philosophie zugrunde, die Sprache klein und schlank zu halten und kontrollierte, individuelle Erweiterungen nur im Rahmen des Klassenkonzeptes zuzulassen. Für Java existiert eine solche vorimplementierter Klasse namens `Math`. Sie enthält beispielsweise Methoden `pow` zur Potenzierung, `sqrt` zur Berechnung der Wurzel und `sin`, `cos`, `tan` für die entsprechenden trigonometrischen Funktionen, wobei Argumente und Resultate von der Sorte `double` sind. Die Aufrufe erfolgen in der üblichen funktionalen Schreibweise, wobei der Methodenzugriff durch die Punktnotation geschieht. Beispielsweise berechnet also der Aufruf `Math.sqrt(22.5)` die Wurzel von 22.5. Eine genauere Beschreibung der Klasse `Math` findet man etwa in Abschnitt 3.1.2 des in der Einleitung beschriebenen Buchs von J. Bishop.

6.2.4 Bemerkung (zur Sortenanpassung)

Bei der Vorstellung der elementaren Datenstrukturen von ML wurde mittels Teilsortenrelationen erklärt, was Sortenanpassung ist, und auch erwähnt, daß in Java eine implizite Sortenanpassung vorgenommen wird. Dies geschieht bei den zahlartigen Sorten durch die **Ausweitung** von `short` über `int`, `long`, `float` zu `double`, da diese Sorten in der durch die Aufzählung angegebenen Teilsortenrelation stehen.

Anpassungen in der entgegengesetzten Richtung müssen auch in Java, wie in ML, durch spezielle Operationen oder Schreibweisen explizit vorgenommen werden. Die allgemeinste Möglichkeit stellen hier die **Sortenumwandlungsoperationen** dar. Ist n eine Teilsorte einer Sorte m für Zahlen und t ein Term der Sorte m, so ist `(`n`)` t ein Term der Sorte n, dessen Wert durch Runden des Werts von t entsteht bzw. durch das Abschneiden von

Stellen in seiner Darstellung. ■

Wir kommen nun zur letzten elementaren Datenstruktur von Java.

6.2.5 Datenstruktur der Zeichen

Sorte:	`char`	Zeichenvorrat von Java
Konstanten:	Zeichen	z.B. `'R'`, Escape-Sequenzen ■

Dabei besitzt Java einen viel größeren Zeichenvorrat als ML. Die Sprache verwendet nämlich nicht die ASCII-Codierung, sondern die Unicode-Codierung, welche derzeit etwa 34000 Zeichen umfaßt. Diese beinhalten, neben dem ASCII-Zeichensatz, z.B. auch griechische und kyrillische Buchstaben und viele mathematische Symbole. Neben der erwähnten Darstellungsart mittels Hochkommata-Klammerung können Zeichen auch direkt durch eine sogenannte **Escape-Sequenz** `\u`*xxxx* dargestellt werden, wobei *xxxx* die 2-Byte Hexadezimal-Codierung des Zeichens ist. Z.B. ergibt `\u005A` den Buchstaben „Z“.

Wir beenden die Vorstellung der elementaren Datenstrukturen von Java mit einigen Bemerkungen zu ihrer Ein- und Ausgabe. Java-Programme einer Einführungsvorlesung in die Programmierung lesen in der Regel interaktiv durch die Tastatur gegebene Daten einer der in diesem Abschnitt vorgestellten Sorten ein und geben ebensolche auf dem Bildschirm aus. Das Ausgeben des Werts eines Term t der Sorte m ist einfach. Man verwendet die Methoden `print` bzw. `println` der vorimplementierten Klasse `System.out`. Im Unterschied zur Ausgabe-Anweisung

`System.out.print(`t`);`

wird bei der Ausgabe-Anweisung

`System.out.println(`t`);`

nach dem Ausdruck auch noch ein Zeilenwechsel vorgenommen. Das Ausgeben einer durch das Zeichen „`"`“ geklammerten Zeichenreihe ist sinnvollerweise ebenfalls auf diese Weise möglich. Die Ausgabe-Anweisung `System.out.print("Rudolf");` ist hierzu ein Beispiel.

Bei Aufrufen der Methoden `print` und `println` können Zeichenreihen und Ausdrücke mittels des Konkatenationsoperators `+` für Zeichenreihen kombiniert werden, wie etwa im folgenden Beispiel, wo eine Methode `fac` zur Berechnung der Fakultät einer natürlichen Zahl angenommen ist:

```
System.out.println("Die Fakultaet der Zahl" +  n  + "ist" fac(n));
```

Bei der Abarbeitung dieses Aufrufs werden die Werte von `n` und `fac(n)` zuerst als Zeichenreihen in der üblichen Dezimaldarstellung dargestellt, diese dann mit den beiden anderen Zeichenreihen `"Die Fakultaet der Zahl"` und `"ist"` in der angegebenen Weise konkateniert und schließlich das Ergebnis der Konkatenation gedruckt.

Hingegen ist die Eingabe eines elementaren Datums mittels der Tastatur in Java für einen Anfänger nicht ganz einfach. Man hat das Datum zuerst als Zeichenreihe einzulesen und diese dann explizit durch eine entsprechende **Umwandlungsmethode** aus einer bestimmten vorimplementierten Klasse in das Datum der gewünschten Sorte umzuwandeln. Aus diesem Grund haben wir im Java-Programm zur Kegelstumpfberechnung des Abschnitts 6.1 das Einlesen von „langen“ reellen Zahlen in einer Methode namens `DoubleRead` verborgen. Aus `DoubleRead` erhält man sofort auch Methoden zum Einlesen anderer zahlartiger Daten durch die Tastatur, indem man die Zeichenreihen `Double` und `double` entsprechend ersetzt. Beispielsweise bewerkstelligt die so entstehende Methode

```
static int IntRead () {
  String s = "";
  try {s = new java.io.DataInputStream(System.in).readLine();}
  catch(java.io.IOException e) {;}
  return java.lang.Integer.parseInt(s); }
```

das Einlesen von ganzen Zahlen in der 32-Bit-Darstellung. Wir werden bis zum Ende des Skriptums immer mit den beiden Methoden `IntRead` und `DoubleRead` arbeiten. Bezüglich einer detaillierten Behandlung der Eingabe in Java müssen wir auf die in der Einleitung genannte Literatur verweisen.

Beim Einlesen von Wahrheitswerten hilft man sich in der Regel dadurch, daß man die Wahrheitswerte *tt* und *ff* durch die Zahlen 1 und 0 codiert. Wenn Wahrheitswerte Resultate von Berechnungen sind, gibt man an Stelle der beiden Konstanten `true` und `false` oft entsprechende erklärende Zeichenreihen aus.

6.3 Variablen und Methoden

In diesem Abschnitt gehen wir genauer auf Variablen und Methoden ein. Insbesondere stellen wir die elementarsten Anweisungen der Sprache Java vor. Diese bestehen aus einer Kategorie von Anweisungen, welche in der Literatur die Sprache der while-Programme genannt werden, ergänzt um Ein- und Ausgabe-Anweisungen und sonstige Aufrufe von vorimplementierten oder benutzerdefinierten Methoden. While-Programme stellen die bedeutendste imperative Modellsprache der Informatik dar, da ihre Anweisungsformen den Kern einer jeden imperativen Programmiersprache bilden.

Wir beginnen mit den Variablen und ihrer Deklaration, welche die Grundlage für das Weitere bildet. Dabei lassen wir uns von dem früher erwähnten simplen Modell leiten, daß eine Variable genau eine bestimmte Speicherzelle bezeichnet.

6.3.1 Variablen und ihre Deklaration

In der Sprache Java sind, wie in ML, alle Variablen typisiert, wobei ein **Typ in Java entweder eine Sorte aus den elementaren Datenstrukturen oder eine Klasse (genauer: deren Name) ist**. Typen der zweiten Art heißen auch **Referenztypen**. Wegen dieser Unterscheidung gibt es auch zwei Arten von Variablen:

(1) Variablen, denen als Typ eine Sorte m der elementaren Datentypen von Abschnitt 6.2 zugeordnet ist: Diese werden auch **elementare Variablen** genannt. Im oben genannten Speichermodell heißt dies, daß in der zugeordneten Speicherzelle ein Element aus m^A abgespeichert ist.

(2) Variablen, denen als Typ eine Klasse C (genauer: deren Name) zugeordnet ist: Bei dieser Art spricht man von **Referenzvariablen**, denn im Speichermodell heißt dies, daß in der zugeordneten Speicherzelle entweder ein Verweis auf einen Speicherbereich steht, in dem eine Instanz der Klasse C untergebracht ist, oder der sogenannte **leere Verweis** `null`[25].

Alle Variablen müssen vor ihrer ersten Verwendung deklariert werden. Dies geschieht in der Form $m\ x$; mit x als der Variablen und m als ihren Typ, wobei noch vorangestellte Modifizierer erlaubt sind, wie Beispiel 6.1.5 zeigt. Durch die Deklaration geschieht auch eine **implizite Initialisierung**, d.h. eine Festlegung des Anfangswerts. Dieser ist 0 für Variablen für Zahlen-Sorten, *ff* für Variablen der Sorte `boolean`, `\u0000` für Variablen der Sorte `char` und `null` für Referenzvariablen. ■

Aufbauend auf die Variablen und Konstanten erfolgt die Bildung von **Termen** eines bestimmten Typs in Java in der uns schon bekannten Weise durch Funktionsapplikationen. Angewendet werden dabei

(1) die Operationen der elementaren Datenstrukturen in den dort jeweils angegebenen Schreibweisen,

(2) Methoden von vorimplementierten oder benutzerdefinierten Klassen in funktionaler Schreibweise, sofern ihre Resultatangabe (vergleiche mit später) ein Typ ist, und

(3) vorimplementierte Operationen auf Objekten, wie etwa die in Beispiel 6.1.5 schon erwähnte Operation `new` zur Erzeugung einer neuen Instanz einer Klasse durch einen Konstruktoraufruf.

Die vorimplementierten Operationen auf Objekten liefern keine Objekte als Resultate ab, sondern nur Verweise auf Objekte, in der zeichnerischen Darstellung also jeweils den entsprechenden Pfeil. Gleiches gilt für Methodenaufrufe, wie wir später noch sehen werden. Somit ist der Wert eines Terms

(1) entweder ein Element aus der Interpretation m^A einer Sorte der Basis-Signatur, wenn nämlich sein Typ m ist, oder

(2) ein Verweis auf ein Objekt einer Klasse C, wenn nämlich sein Typ C ist,

[25] Zeichnerisch stellt man Verweise in der Regel durch Pfeile dar und den leeren Verweis `null` dadurch, daß man in die entsprechende Speicherzelle ein diagonales Kreuz zeichnet. In der Implementierung wird stattdessen die Anfangsadresse des Bereichs für die Instanz in die Speicherzelle geschrieben bzw. ein Sonderelement für `null`.

Mit Hilfe von Variablen und Termen können wir nun festlegen, was Methoden sind. Im folgenden betrachten wir dabei nicht den gesamten Anweisungsvorrat von Java, sondern nur diejenigen Anweisungsformen, welche die Sprache der while-Programme bilden, sowie, damit wir strukturierte komplette Programme mit Ein- und Ausgabe schreiben können, zusätzlich noch Methodenaufrufe.

6.3.2 Methoden in Java

Eine **funktionale Methode** in Java besteht aus der **Resultatangabe** r, dem **Namen** M, der **Liste der formalen Parameter** x_i jeweils des Typs m_i, $1 \leq i \leq k$, und dem **Rumpf**. Syntaktisch deklariert man dies in der folgenden Form:

$$r\ M\ \texttt{(}m_1\ x_1\texttt{,}\ \ldots\texttt{,}m_k\ x_k\texttt{)\{} \gg \mathit{Rumpf} \ll \texttt{\}}$$

Dabei ist r ein Typ. Es ist $k = 0$ erlaubt, d.h. die Parameterliste darf leer sein. Der Rumpf besteht aus einem **Deklarationsteil**, das ist eine Folge von Deklarationen von Variablen, gefolgt von einer **Anweisung** und der **Resultatfestlegung** `return` t`;` mit einem Term t des Typs r. Es dürfen sowohl der Deklarationsteil als auch die Anweisung fehlen.

Die Menge der Anweisungen (die wir in diesem Skriptum betrachten) ist dabei induktiv wie folgt definiert:

a) Die **leere Anweisung** `;` ist eine Anweisung.

b) Ist x eine Variable des Typs m und t ein Term des gleichen Typs, so ist die **Zuweisung** x `=` t`;` mit der **rechten Seite** t eine Anweisung.

c) Sind s_1 und s_2 Anweisungen und ist b ein Term der Sorte `boolean`, so ist auch die **(zweiseitige) bewachte Anweisung** `if (`b`) {`s_1`} else {`s_2`}` mit der **Bedingung** b, dem **then-Fall** `{`s_1`}` und dem **else-Fall** `{`s_2`}` eine Anweisung.

d) Ist s eine Anweisung und b ein Term der Sorte `boolean`, so ist auch die **while-Schleife** `while (`b`) {`s`}` mit der **Schleifenbedingung** b und dem **Schleifenrumpf** `{`s`}` eine Anweisung.

e) Sind s_1 und s_2 Anweisungen, so ist auch deren **sequentielle Komposition** $s_1\, s_2$ eine Anweisung.

f) Ist N eine vorimplementierte oder benutzerdefinierte Methode, deren Resultatangabe `void` ist, so ist auch ein **Methodenaufruf** N`(`a_1`,...,`a_n`);` von N mit den Argumenttermen a_i, $1 \leq i \leq k$, von jeweils passendem Typ eine Anweisung.

Eine **prozedurale Methode** ist ähnlich zu einer funktionalen Methode definiert. Die einzigen Unterschiede sind, daß der Rumpf hier nur aus einem Deklarationsteil und einer folgenden Anweisung besteht, wobei wiederum beide fehlen dürfen, und die Resultatangabe `void` ist.

Wie bei den Klassen können auch bei den Methoden zusätzliche Eigenschaften durch Modifizierer festgelegt werden. Wir werden nur `public`, `private` und `static` verwenden. Die Bedeutung von `public` und `static` haben wir schon früher bei der Besprechung von Klassen bzw. in Bemerkung 6.1.6 angegeben; der Modifizierer `private` besagt, daß die Methode außerhalb der Klasse nicht sichtbar (also verborgen) ist. Mit `static` modifizierte Methoden heißen auch **Klassenmethoden**. ■

Die prozeduralen Methoden von Java mit dem Modifizierer `static`, also die prozeduralen Klassenmethoden, entsprechen genau den klassischen Prozeduren, wie man sie etwa von den Sprachen Pascal oder C her kennt. Das einfachte Beispiel ist `static void M () { }`.

Wir haben bei der bewachten Anweisung und bei der while-Schleife den then-Fall und den else-Fall bzw. den Schleifenrumpf mit Klammern versehen, um syntaktische Eindeutigkeit zu erzwingen, wenn diese Anweisungen durch sequentielle Kompositionen entstanden sind. Man nennt eine durch geschweifte Klammern begrenzte sequentielle Komposition von mehreren Anweisungen in der älteren Literatur auch einen **Block**. Bestehen der then-Fall oder else-Fall bzw. der Schleifenrumpf aus einer Anweisung, die nicht eine sequentielle Komposition anderer Anweisungen ist, so kann auf die beiden geschweiften Klammern verzichtet werden.

Weiterhin haben wir bei der Einführung von Methoden, aus methodischen Gesichtspunkten, die später hoffentlich noch klar werden, alle Deklarationen vor die Anweisung gezogen und nur ein Resultatfestlegung `return` t erlaubt. In Java dürfen, dies verallgemeinernd, Deklarationen und Anweisungen sogar gemischt werden und es sind auch mehrere Resultatfestlegungen möglich[26]. Die einzige Forderung ist, daß jede in einer Methode verwendete Variable, die kein formaler Parameter ist, vor dem ersten Auftreten in einer Anweisung deklariert wird. Dabei kann eine Deklaration $m\ x$`;` mit einer anschließenden Zuweisung x `=` t`;` sogar zu $m\ x$ `=` t`;` verschmolzen werden. Wir haben dies schon in Bemerkung 6.1.6 verwendet.

Die informelle Bedeutung der oben aufgezählten einzelnen Anweisungen ergibt sich vielfach direkt aus deren Bezeichnungen. Hier sind die einfachen Fälle a) bis e): Die leere Anweisung hat keine Auswirkungen, verändert also den Speicherzustand nicht und liest bzw. druckt auch keine Daten. Den Fall einer Zuweisung x `=` t`;` haben wir für eine elementare Variable x schon in Abschnitt 6.1 erklärt. Ist x hingegen eine Referenzvariable, so wird die ihr zugeordnete Speicherzelle zu dem sich als Wert von t ergebenden Verweis abgeändert. Der am häufigsten vorkommende Fall ist hier x `=` y`;` mit zwei Referenzvariablen x und y. In der zeichnerischen Darstellung wird dadurch von der Speicherzelle von x ein Pfeil zu dem Bereich gezeichnet, auf den der Pfeil der Speicherzelle von y zeigt. Bei der bewachten Anweisung wird der then-Fall oder der else-Falls ausgeführt, je nachdem, ob der Wert der Bedingung gleich *tt* oder *ff* ist. Im Fall einer while-Schleife wird der Schleifenrumpf solange ausgeführt, bis zum ersten Mal der Wert der Schleifenbedingung gleich *ff* ist. Liegt eine sequentielle Komposition $s_1\ s_2$ vor, so wird schließlich zuerst s_1

[26] Java faßt nämlich genaugenommen `return` t als Anweisung auf, die die Ausführung einer Methode beendet und dabei zusätzlich einen Wert berechnet. Ohne eine Wertangabe wird durch `return` nur die Ausführung beendet.

und dann s_2 ausgeführt.

Diese eben gebrachten informellen Erklärungen betreffen aber nur Abarbeitungen ohne Fehlerfälle. Fehler treten z.B. auf, wenn ein Term undefiniert ist oder eine while-Schleife nicht terminiert (d.h. die Schleifenbedingung niemals den Wert *ff* annimmt).

Nach der obigen Beschreibung ist es klar, was informell die Semantik einer Methode M ist, deren Anweisung keinen Methodenaufruf enthält:

(1) Im Fall einer funktionalen Methode werden zuerst der Speicherzustand und eventuell auch Ein- und Ausgabesequenzen gemäß der Anweisung des Rumpfs von M verändert und dann als Ergebnis der Wert des Terms t des abschließenden `return` t`;` bezüglich des erhaltenen Speichers definiert.

(2) Im Fall einer prozeduralen Methode werden nur der Speicherzustand und eventuell auch Ein- und Ausgabesequenzen gemäß der Anweisung des Rumpfs von M transformiert.

Kommt zusätzlich in der Anweisung des Rumpfs von M ein Aufruf N`(`a_1`,...,`a_n`);` einer Methode N mit formalen Parametern y_i, $1 \leq i \leq n$, und Resultatangabe `void` vor, so geschieht dessen Ausführung mittels der folgenden Schritte:

(1) Zuerst wird jedes Argument a_i von N ausgewertet und sein Wert in einer neuen i-ten Speicherzelle gespeichert, $1 \leq i \leq n$.

(2) Dann wird allen formalen Parametern y_i die i-te neue Speicherzelle zugeordnet, $1 \leq i \leq n$, und mit dieser Zuordnung die Anweisung des Rumpfs von N ausgeführt, d.h. Speicher, Ein- und Ausgabe entsprechend geändert.

Weil man bei diesen beiden Schritten zuerst die Argumente einer Methode auswertet und erst dann mit den in den neuen Speicherzellen untergebrachten Werten weiterrechnet, spricht man hier ebenfalls von einer **Call-by-value Parameterübergabe**. Man beachte jedoch, daß bei Termen eines Referenztyps der Wert ein Verweis auf ein Objekt ist, man solche Argumente eigentlich in einer Weise übergibt, die man in Pascal und ähnlichen Sprachen **Call-by-reference** nennt.

Das folgende Beispiel soll den Unterschied zwischen elementaren Variablen und Referenzvariablen verdeutlichen und zeigen, daß man bei letzteren wegen der Verweise als Werte durchaus auch jene gefährlichen Effekte erhalten kann, die man etwa von den Pascal-Zeigern her kennt.

6.3.3 Beispiel (zur Semantik von Anweisungen)

Wir betrachten die folgenden beiden Anweisungen, wobei in der linken `x` und `y` zwei Variablen der Sorte `int` sind, und in der rechten `A` und `B` zwei Variablen des Referenztyps

account von Beispiel 6.1.5:

```
x = 100;                          A = new account();
y = x;                            A.deposit(500);
y = y+1;                          B = A;
System.out.println(x);            B.withdraw(200);
System.out.println(y);            System.out.println(A.available());
                                  System.out.println(B.available());
```

Die Abarbeitung der linken Anweisung gibt offensichtlich die beiden Werte 100 und 101 aus. Weil bei der rechten Anweisung nach dem Abarbeiten der dritten Zeile `A` und `B` als Wert einen Verweis auf das gleiche Objekt besitzen und in der vierten Zeile das `st`-Attribut dieses Objekts zu 300 geändert wird, lautet die Ausgabe 300 gefolgt von 300 und nicht, wie vielleicht erwartet, 500 gefolgt von 300. ■

Es sei an dieser Stelle der Vollständigkeit halber aber auch erwähnt, daß es in Java eine Operation `clone` gibt, mit deren Hilfe man eine identische Kopie eines Objekts erzeugen kann.

6.3.4 Bemerkung (Sprache der while-Programme)

Neben den oben angeführten Anweisungsformen gibt es in Java noch viele weitere Möglichkeiten, Anweisungen hinzuschreiben. Diese kann man aber fast alle auf die obigen Anweisungsformen a) bis e) zurückführen.

Beispielsweise kann man die **einseitige bewachte Anweisung**

$$\texttt{if (}b\texttt{) \{}s_1\texttt{\}}$$

(die wir schon in Beispiel 6.1.5 verwendet haben) als Abkürzung für die spezielle zweiseitige bewachte Anweisung

$$\texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{;\}}$$

erklären. Ein zweites Beispiel ist die folgende **nichtabweisende Schleife**:

$$\texttt{do \{}s\texttt{\} while (}b\texttt{);}$$

Durch sie wird die Anweisung s solange ausgeführt, bis der Term b der Sorte `boolean` den Wert *ff* liefert. Man kann dieses Sprachkonstrukt somit als eine Abkürzung für das folgende Programmstück auffassen:

$$s\ \texttt{while (}b\texttt{) \{}s\texttt{\}}$$

Besteht s aus einer Anweisung, die nicht eine sequentielle Komposition ist, so kann auf die beiden geschweiften Klammern wiederum verzichtet werden.

Wegen dieser Ausdrucksstärke der Anweisungsformen a) bis e) legt man sie vielen Untersuchungen der Informatik zugrunde und nennt sie die **Sprache der while-Programme**.

Aus theoretischen und methodischen Gründen fordert man für while-Programme zusätzlich noch, daß das **Auswerten von Termen keine Variableninhalte verändert**, es also seiteneffektfrei ist. Dies verbietet etwa Zuweisungen, die einen Aufruf einer Eingabe-Methode enthalten. ■

Bevor wir nun weitere konkrete Beispiele für Java-Programme angeben, wollen wir erst noch das Verhältnis von repetitiver Rekursion – wie wir es von funktionaler Programmierung kennen – und der while-Schleife klären, da dies der Hintergrund für den Kern vieler imperativer Programme ist.

6.3.5 Repetitive Rekursion und while-Schleife

Wir betrachten eine (partielle) Funktion $f : M \longrightarrow N$, die mit Hilfe von drei (partiellen) Funktionen $b : M \longrightarrow \mathbb{B}$, $k : M \longrightarrow M$ und $l : M \longrightarrow N$ wie nachfolgend angegeben durch eine repetitive Rekursion beschrieben ist:

$$f(x) = \begin{cases} f(k(x)) & : \quad b(x) = tt \\ l(x) & : \quad b(x) = ff \end{cases}$$

Nun sei $a \in M$ ein beliebiges Element. Wenn wir dann die obige repetitive Rekursion termmäßig durch die Gleichungskette

$$f(a) = f(k(a)) = f(k^2(a)) = \ldots$$

„abwickeln“, so erhalten wir offensichtlich für den Aufruf $f(a)$ die Gleichung

$$f(a) = \begin{cases} l(k^n(a)) & : \quad n = min\{\, i \in \mathbb{N} : b(k^i(a)) = ff \,\} \\ \text{undef} & : \quad \forall\, i \in \mathbb{N} : b(k^i(a)) \neq ff\,. \end{cases}$$

Wenn wir also weiterhin annehmen, daß f, b, k und l als Operationen gleichen Namens in der Basis-Signatur von Java enthalten sind und die Mengen M und N durch entsprechende Sorten m und n interpretiert werden, so ist, mit Variablen x und y der Sorte m bzw. n und a als einem Term der Sorte m (statt einem Element aus M), die Zuweisung

$$y \texttt{ = } f\texttt{(}a\texttt{);} \tag{1}$$

gleichwertig zum folgenden Programmstück, bestehend aus einer Initialisierung, einer while-Schleife und einer „Nachberechnung“:

$$\begin{array}{l} x \texttt{ = } a\texttt{;} \\ \texttt{while (}b\texttt{(}x\texttt{))} \\ \quad x \texttt{ = } k\texttt{(}x\texttt{);} \\ y \texttt{ = } l\texttt{(}x\texttt{);} \end{array} \tag{2}$$

Auf diese Weise kann man sich durch den Schritt von (1) nach (2) von der Annahme befreien, daß f in der Basis-Signatur vorkommt, d.h. vorimplementiert ist. Eine Wiederholung des Schritts erlaubt schließlich oft, ein Java-Programm zu erreichen, in dem nur mehr

vorimplementierte Operationen vorkommen, d.h. alle sogenannten **Pseudooperationen** – wie oben f – eliminiert sind.

Hat man es mit einer n-stelligen repetitiv-rekursiven Pseudooperation $f : \prod_{i=1}^{n} M_i \longrightarrow N$ zu tun, so sind auch b, l und k n-stellig. Insbesondere haben wir $k : \prod_{i=1}^{n} M_i \longrightarrow \prod_{i=1}^{n} M_i$ und diese (partielle) Funktion ist in der Regel realisiert durch

$$k(x_1, \ldots, x_n) = \langle k_1(x_1, \ldots, x_n), \ldots, k_n(x_1, \ldots, x_n) \rangle$$

mit $k_j : \prod_{i=1}^{n} M_i \longrightarrow M_j$ für alle j mit $1 \leq j \leq n$. Wir bekommen damit, entsprechende Sorten, Operationen und Variablen vorausgesetzt, als Erweiterung der obigen Regel „(1) $\Rightarrow$ (2)“ die Gleichwertigkeit der Zuweisung (mit Termen a_i, $1 \leq i \leq n$)

$$y \texttt{ = } f\texttt{(}a_1\texttt{,} \ldots \texttt{,}a_n\texttt{);} \tag{3}$$

mit dem folgenden Programmstück, in dem die sogenannte **kollaterale Zuweisung** den Variablen links des Symbols „=“ **gleichzeitig** die Werte der rechten Terme zuweist:

$$\begin{array}{l} x_1\texttt{,} \ldots \texttt{,}x_n \texttt{ = } a_1\texttt{,} \ldots \texttt{,}a_n\texttt{;} \\ \texttt{while (}b\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{))} \\ \quad x_1\texttt{,} \ldots \texttt{,}x_n \texttt{ = } k_1\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{),} \ldots \texttt{,}k_n\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{);} \\ y \texttt{ = } l\texttt{(}x_1\texttt{,} \ldots \texttt{,}x_n\texttt{);} \end{array} \tag{4}$$

Die kollaterale Zuweisung, bei der natürlich alle Variablen der linken Seite als paarweise verschieden vorausgesetzt sind, existiert in der Sprache Java nicht. Man muß also auch noch diese **Pseudo-Anweisung** mittels Sequentialisierung durch Java-Zuweisungen (eventuell mit Hilfsvariablen) realisieren. ■

Im folgenden zeigen wir anhand von zwei schon bekannten Beispielen, wie man mit Hilfe der eben vorgestellten Umformungen eine Anweisung mit einer Pseudooperation in eine „echte“ Java-Anweisung überführen kann, welche schließlich ohne größeren intellektuellen Aufwand zu einem kompletten und lauffähigen Java-Programm führt.

6.3.6 Beispiele (für zwei Programmentwicklungen)

a) Wir betrachten zuerst die Zuweisung

```
y = fac(a);
```

mit zwei Variablen `a` und `y` der Sorte `int` und einer Pseudooperation `fac` zur Berechnung der Fakultät einer natürlichen Zahl. Aus Beispiel 3.3.5 wissen wir, daß die Gleichung `fac(a)` = `facr(a,1)` gilt, mit `facr` definiert durch die Repetition

$$\texttt{facr(n,res)} = \begin{cases} \texttt{facr(n-1, n*res)} & : \quad \texttt{n} \neq \texttt{0} \\ \texttt{res} & : \quad \texttt{n} = \texttt{0}. \end{cases}$$

Wir können also von der obigen Zuweisung `y = fac(a);` übergehen zur Zuweisung

```
y = facr(a,1);
```

– mit `facr` als Pseudooperation. Aufgrund der zweiten Regel „(3) ⇒ (4)“ von oben ist diese gleichwertig zu dem folgenden Pseudo-Programmstück:

```
n,res = a,1;
while (n != 0)
  n,res = n-1,n*res;
y = res;
```

Es bleiben noch die zwei kollateralen Zuweisungen zu sequentialisieren. Bei der ersten kann man jede Reihenfolge wählen, bei der zweiten muß erst die Zuweisung an `res` erfolgen, da die Zuweisung an `n` nicht auf `res` Bezug nimmt. Wir erhalten insgesamt gleichwertig zur originalen Zuweisung das folgende Programmstück:

```
res = 1;
n = a;
while (n != 0) {
  res = n * res;
  n = n - 1; }
y = res;
```

Dessen Vervollständigung zur Methode `main` durch einen Deklarationsteil, eine Einlesezuweisung und eine Resultatausgabe ist offensichtlich und liefert schließlich, durch Zusammenfassung mit der Methode `IntRead`, zu einer Klasse das nachstehende Java-Programm zur Berechnung der Fakultät einer natürlichen Zahl:

```
public class fac {
  private static int IntRead () {
    String s = "";
    try {s = new java.io.DataInputStream(System.in).readLine();}
    catch(java.io.IOException e) {;}
    return java.lang.Integer.parseInt(s); }

  public static void main (String[] args) {
    int a, y, n, res;
    a = IntRead();
    res = 1;
    n = a;
    while (n != 0) {
      res = n * res;
      n = n - 1; }
    y = res;
    System.out.println(y); }
  }
```

b) Nun setzen wir eine Pseudooperation `isperfect` zum Testen von Perfektsein voraus, sowie Variablen `a` und `res` von der Sorte `int` bzw. `boolean`. Wir betrachten die folgende Zuweisung:

```
res = isperfect(a);
```

Aufgrund der Ergebnisse von Beispiel 3.1.1 ist diese gleichwertig zu:

```
res = isp(a,a,a);
```

Dabei ist die Pseudooperation `isp` durch die repetitiven Rekursion

$$\texttt{isp(n,x,y)} = \begin{cases} \texttt{isp(n,x-1,H(n,x-1,y))} & : \ \texttt{x} \neq 0 \\ \texttt{y} = 0 & : \ \texttt{x} = 0 \end{cases}$$

definiert, und die Pseudooperation `H` (welche eine geschachtelte Fallunterscheidung in der `isp`-Rekursion verhindert) durch die Fallunterscheidung

$$\texttt{H(n,x,y)} = \begin{cases} \texttt{y-x} & : \ \texttt{x} \text{ teilt } \texttt{n} \\ \texttt{y} & : \ \texttt{x} \text{ teilt nicht } \texttt{n}. \end{cases}$$

Wir können also wiederum den zweiten der oben vorgestellten Übergänge verwenden und bekommen als gleichwertig zu `res = isp(a,a,a);` – also auch zur originalen Zuweisung – das folgende Programmstück:

```
n,x,y = a,a,a;
while (x != 0)
  n,x,y = n,x-1,H(n,x-1,y);
res = (y==0);
```

Die Sequentialisierung der Initialisierung von `n`, `x` und `y` ist beliebig. Hingegen ist bei einer Sequentialisierung der kollateralen Zuweisung des Schleifenrumpfs etwas Vorsicht geboten. Der Schleifenrumpf kann beispielsweise erst das `y` ändern, der Rest ist dann beliebig, oder er kann erst das `x` ändern und dann im Aufruf von `H` den Term `x-1` durch `x` ersetzen. Der Rest ist dann wiederum beliebig. Wählen wir etwa die zweite Möglichkeit, so führt dies zu:

```
x = a; y = a; n = a;
while (x != 0) {
  x = x-1; y = H(n,x,y); n = n; }
res = (y==0);
```

Es verbleibt noch die Aufgabe, in der Zuweisung `y = H(n,x,y);` die Pseudooperation `H` durch Operationen der Java-Signatur zu realisieren. Dazu beachten wir, daß `x` dann und nur dann `n` teilt, wenn `x` $= 0$ impliziert `n` $= 0$ (weil die Null offensichtlich das größte Element der durch die Relation „ist Teiler von“ geordneten natürlichen Zahlen ist) und aus `x` $\neq 0$ folgt, daß der Rest der ganzzahligen Division von `n` durch `x` gleich Null ist.

Dies führt, weil wir `a` $\neq 0$ und damit auch `n` $\neq 0$ annehmen dürfen (0 als Eingabe führt zu keinem Schleifendurchlauf), nach einigen einfachen logischen Umformungen zur folgenden Darstellung von `H`, wobei für die Restbildung bei der ganzzahligen Division schon Java-Notation verwendet wird:

$$\texttt{H(n,x,y)} = \begin{cases} \texttt{y-x} & : \ \texttt{x != 0 \&\& n\%x == 0} \\ \texttt{y} & : \ \text{sonst} \end{cases}$$

Aufgrund dieser Gleichung kann man die Zuweisung `y = H(n,x,y);` offensichtlich mittels der folgenden einseitigen bewachten Anweisung realisieren.

```
if ((x != 0) && (n % x == 0)) y = y - x;
```

Nun setzen wir dies in das obige Programmstück ein, streichen noch die zur leeren Anweisung gleichwertige Zuweisung `n = n;` und erhalten schließlich durch eine Vervollständigung das nachstehende Java-Programm:

```
public class isperfect {
  private static int IntRead () {
    String s = "";
    try {s = new java.io.DataInputStream(System.in).readLine();}
    catch(java.io.IOException e) {;}
    return java.lang.Integer.parseInt(s); }

  public static void main (String[] args) {
    int a, n, x, y;
    boolean res;
    a = IntRead();
    x = a;
    y = a;
    n = a;
    while (x != 0) {
      x = x - 1;
      if ((x != 0) && (n % x == 0)) y = y - x; }
    res = (y == 0);
    if (res)
      System.out.println("Perfekte Zahl");
    else
      System.out.println("Nicht perfekte Zahl"); }
  }
```

Dieses Programm gibt statt eines Wahrheitswerts eine entsprechende Zeichenreihe als Meldung aus. ■

Es ist klar, daß man in den beiden entwickelten Programmen noch einige Variablen einsparen kann. So kann man in `fac` etwa `a` sparen, wenn man mit der Zuweisung `y=fac(y)` statt `y=fac(a)` die Programmherleitung beginnt. Auch die Variable `res` kann beispielsweise im `isperfect` eingespart werden, wenn man die entsprechenden Zeichenreihen `"Perfekte Zahl"` bzw. `"Nicht perfekte Zahl"` direkt in Abhängigkeit vom Wert von `y==0` ausdruckt. Wir haben auf solche Optimierungen verzichtet, da sie erstens keinerlei praktische Auswirkungen auf die Effizienz haben und zweitens durch den Verzicht das prinzipielle Vorgehen anhand der obigen Regeln klarer wird.

6.4 Betrachtungen zu Semantik und Verifikation

In Abschnitt 3.4 hatten wir bei den semantischen Betrachtungen zu ML zwei Arten zur Definition der Semantik kennengelernt, die operationelle und die denotationelle. Auch für imperative Sprachen sind diese beiden Arten der Semantikfestlegung möglich. Wir besprechen im folgenden jedoch eine dritte Art, die **axiomatische Semantik** genannt wird, da sie auf einem (als Axiome auffaßbaren) Regelsystem zur Herleitung von Korrektheitsaussagen basiert. Weiterhin zeigen wir – und auch dies geschieht in Analogie zur funktionalen Programmierung – wie man die axiomatische Semantik bei der Programmentwicklung einsetzen kann. Der Einfachheit halber **beschränken wir uns im weiteren auf die Sprache der while-Programme**.

Im Gegensatz zur operationellen oder denotationellen Semantik orientiert sich die axiomatische Semantik nicht an der konkreten Abarbeitung einer Anweisung s bzw. der durch sie hervorgerufenen Änderung des Speicherzustands, sonderen charakterisiert die durch s bewirkte Speicheränderung durch ein Paar (φ, ψ) von Aussagen, wobei φ den Speicherzustand vor und ψ den Speicherzustand nach der Ausführung von s beschreibt. Formal werden Aussagen durch Formeln ausgedrückt.

Zur Formulierung von Aussagen über while-Programme benutzen wir Formeln, die aufgebaut sind über einer (im allgemeinen nicht näher spezifizierten) Signatur, von der wir nur fordern, daß sie die Sorten, Konstanten und Operationen der Java-Signatur von Abschnitt 6.2 enthält, und einer (auch nicht näher spezifizierten Menge) von Variablen. Ausgehend von den Termen der Sorte `boolean`, werden Formeln aufgebaut mit Hilfe der bekannten logischen Symbole (Junktoren) $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ und der Quantoren $\forall$ und $\exists$.

6.4.1 Definition (Formeln)

Wir definieren induktiv die **Menge aller Formeln** wie folgt:

(1) Jeder Term der Sorte `boolean` über der erweiterten Signatur ist eine Formel.

(2) Ist φ eine Formel, so ist auch die Negation $(\neg\varphi)$ eine Formel.

(3) Sind φ und ψ Formeln, so sind auch die Konjunktion $(\varphi \wedge \psi)$, Disjunktion $(\varphi \vee \psi)$, Implikation $(\varphi \rightarrow \psi)$ und Äquivalenz $(\varphi \leftrightarrow \psi)$ Formeln.

(4) Ist x eine Variable und φ eine Formel, so sind auch die Allquantifizierung $(\forall\, x : \varphi)$ und die Existenzquantifizierung $(\exists\, x : \varphi)$ Formeln. In Formeln dieser Bauart heißt x durch den Quantor $\forall$ bzw. $\exists$ **gebunden**.

Bei der axiomatischen Semantik hat sich auch die Sprechweise **Zusicherung** für Formeln eingebürgert. ■

Durch die induktive Definition 6.4.1 erhalten wir Formeln in vollständiger Klammerung. Im weiteren nehmen wir jedoch die in der Mathematik üblichen Vorrangregeln an, um die Anzahl der Klammern in einer Formel möglichst gering zu halten.

Wir werden später Formeln und Programme in einem gemeinsamen Kontext verwenden. Damit bekommen wir zwei Arten von Variablen, nämlich diejenigen, welche in Programmen vorkommen, d.h. welchen Speicherzellen zugeordnet sind und deren Wert sich ändern kann, und diejenigen, welche Variablen im Sinne der Logik und der funktionalen Programmierung sind, also Platzhalter für bestimmte Daten. Manchmal bezeichnet man die erste Art als **Programmvariablen** und die zweite Art als **logische Variablen**. Wir werden diese Unterscheidung nicht explizit durch die Aufteilung in zwei Mengen sondern implizit treffen. Letzteres wird durch die folgende Forderung ausgedrückt:

6.4.2 Forderung (an die Variablen)

Kommt eine Variable in einem Programm und in einer Formel gleichzeitig vor, so darf sie in der Formel nicht durch einen All- oder Existenzquantor gebunden sein. ■

Für das weitere Vorgehen benötigen wir einen **Gültigkeitsbegriff** für Formeln in einem vorgegebenen Speicherzustand. Wir verzichten an dieser Stelle auf eine formale Definition, da diese im Prinzip nichts anderes bewerkstelligt, als die logischen Symbole in die deutschen Worte „nicht“, „und “, „oder“ usw. mit ihren offensichtlichen Bedeutungen zu transformieren. Beispielsweise ist die Formel $x = 2 \wedge y * y > 3$ in einem Speicherzustand genau dann gültig, wenn der Wert der Variablen x gleich 2 ist und das Quadrat des Werts der Variablen y größer als 3 ist.

Weiterhin benötigen wir einen **Terminierungsbegriff** für while-Programme, ebenfalls bezüglich eines vorgegebenen Speicherzustands. Auch hier verzichten wir auf eine formale Definition und nennen ein while-Programm terminierend, wenn beim gegebenen Speicherzustand

(1) jede while-Schleife nur endlich oft durchlaufen wird und

(2) die Auswertung eines jeden Terms unter den entsprechenden Bedingungen der bewachten Anweisungen bzw. while-Schleifen einen definierten Wert liefert.

Beispielsweise terminiert die Zuweisung `x = 1.0 / z;` für alle Speicherzustände, bei denen der Wert der Variablen `z` ungleich Null ist. Dagegen terminiert `if (z != 0) x = 1.0 / z;` für alle Speicherzustände. Terminiert ein while-Programm bezüglich eines Speicherzustands, so erhalten wir nach der Ausführung einen eindeutig festgelegten neuen Speicherzustand. Die in der folgenden Definition eingeführten Begriffe sind nun wesentlich für das weitere Vorgehen.

6.4.3 Definition (Korrektheitsbegriffe für while-Programme)

Es sei s ein while-Programm. Weiterhin seien φ und ψ zwei Formeln, genannt die **Vor**- bzw. die **Nachbedingung**.

a) Die Anweisung s ist **partiell-korrekt** bezüglich φ und ψ, falls aus der Gültigkeit von φ in einem Speicherzustand vor der Ausführung von s und aus der Terminierung von s bezüglich dieses Speicherzustands folgt, daß ψ in dem Speicherzustand nach der Ausführung von s gilt.

b) Die Anweisung s ist **total-korrekt** bezüglich φ und ψ, falls aus der Gültigkeit von φ in einem Speicherzustand vor der Ausführung von s die Terminierung von s bezüglich dieses Speicherzustands folgt, sowie die Gültigkeit von ψ in dem Speicherzustand nach der Ausführung von s.

Wir schreiben abkürzend $\{\varphi\}\, s\, \{\psi\}$, um auszudrücken, daß s partiell-korrekt bezüglich φ und ψ ist, und $[\varphi]\, s\, [\psi]$, um auszudrücken, daß s total-korrekt bezüglich φ und ψ ist. ■

Man beachte, daß sowohl in Punkt a) als auch in Punkt b) dieser Definition über alle Speicherzustände quantifiziert wird. In der Literatur werden die Speicherzustände bei einer informellen Einführung oder Beschreibung der obigen Begriffe oft unterdrückt. Man sagt dann beispielsweise, daß s partiell-korrekt bezüglich der Vorbedingung φ und der Nachbedingung ψ ist, falls aus der Gültigkeit von φ vor der Ausführung von s und der Terminierung von s die Gültigkeit von ψ nach der Ausführung von s folgt.

Obwohl die Begriffe in Definition 6.4.3 auf die nur informell eingeführten Begriffe „Gültigkeit“ und „Terminierung“ aufbauen, kann man mit Hilfe formaler logischer Umformungen den folgenden, sehr wichtigen Sachverhalt zeigen:

6.4.4 Satz (Totale Korrektheit ist partielle Korrektheit plus Terminierung)

Gegeben seien φ als eine Vorbedingung und ψ als eine Nachbedingung zu einem while-Programm s. Es gilt $[\varphi]\, s\, [\psi]$ genau dann, wenn $\{\varphi\}\, s\, \{\psi\}$ gilt und bezüglich aller Speicherzustände, in denen φ gilt, s auch terminiert.

Beweis: Wir betrachten zu einem gegebenen Speicherzustand die nachstehenden drei Abkürzungen:

V : In dem gegebenen Speicherzustand gilt φ.
N : Nach der Ausführung von s im gegebenen Speicherzustand gilt ψ im neuen Speicherzustand.
T : Das Programm s terminiert bezüglich des gegebenen Speicherzustands.

Der Satz ist dann offensichtlich bewiesen, wenn die folgende Äquivalenz gezeigt ist:

$$V \Longrightarrow (T \wedge N) \quad \Longleftrightarrow \quad ((V \wedge T) \Longrightarrow N) \wedge (V \Longrightarrow T) \qquad (*)$$

Zum Beweis von $(*)$ führen wir eine Fallunterscheidung durch:. Ist T wahr, so gilt die Äquivalenz $(*)$, denn beide Seiten sind gleichwertig zu $V \Longrightarrow N$. Ist hingegen T nicht wahr, so sind beide Seiten von $(*)$ gleichwertig zu $V \Longrightarrow \mathit{false}$, also auch in diesem Fall äquivalent. ■

Wegen Satz 6.4.4 bietet es sich an, statt der totalen Korrektheit die partielle Korrektheit und die Terminierung (bei einer gegebenen Vorbedingung) zu betrachten, d.h. das Abbildungsverhalten und die Totalität der entsprechenden Speichertransformation gesondert zu untersuchen. Dies ist beispielsweise mit Hilfe von denotationeller oder operationeller Semantik möglich. Für die praktische Programmentwicklung und -verifikation aber noch wesentlich besser geeignet ist das folgende Vorgehen:

(1) Man behandelt **Terminierung von while-Programmen** mit Hilfe von Terminierungsfunktionen in noethersch geordnete Mengen, indem man zeigt, daß Variablenwerte echt verringert werden.

(2) Man behandelt **partielle Korrektheit**, indem man ein Regelsystem zur Herleitung von Ausdrücken der Art $\{\varphi\}\, s\, \{\psi\}$ angibt. Die Angabe eines solchen Systems entspricht genau der axiomatischen Definition der Semantik von while-Programmen bei partieller Korrektheit als Korrektheitsbegriff.

Das Regelsystem, welches wir für (2) im weiteren Verlauf dieses Abschnitts angeben werden, geht auf R. Floyd und insbesondere auf C.A.R. Hoare aus dem Jahr 1969 zurück und wird in der Literatur auch **Hoare-Kalkül** oder **Zusicherungskalkül** genannt. Die Regeln haben die allgemeine Form

$$\frac{A_1 \ \ldots \ A_n}{B}$$

mit $n \geq 0$, wobei jedes A_i, $1 \leq i \leq n$, und auch B eine Formel im Sinne von Definition 6.4.1 ist, oder ein Ausdruck der Gestalt $\{\varphi\}\, s\, \{\psi\}$ mit Vorbedingung φ, while-Programm s und Nachbedingung ψ (man sagt zu letzteren auch: eine **Hoare-Formel** oder ein **Hoare-Tripel**). Die Elemente $A_1, \ldots, A_n$ einer Regel heißen **Oberformeln** oder **Prämissen** und das Element B heißt **Unterformel** oder **Konklusion**. Ist $n = 0$, d.h. die Liste der Oberformeln leer, so nennt man die Regel $\frac{}{B}$ ein **Axiom**. Und hier ist nun der angekündigte Hoare-Kalkül für die Sprache fer while-Programme.

6.4.5 Definition (Hoare-Kalkül für while-Programme)

Der **Hoare-Kalkül** (oder die **axiomatische Semantik**) für die Sprache der while-Programme ist gegeben durch das nachfolgende Regelsystem:

a) **Axiome:** Für alle Formeln φ, die **in allen Speicherzuständen gültig sind**[27], ist die folgende Regel ein Axiom:

$$\frac{}{\varphi}$$

Für alle Formeln φ ist die folgende Regel für die leere Anweisung ein Axiom:

$$\frac{}{\{\varphi\}\ ;\ \{\varphi\}}$$

[27] Man nennt so eine Formel **allgemeingültig**.

Es bezeichne φ_x^t die Formel, die aus φ entsteht, indem man die Variable x durch den Term t textuell ersetzt. Dann ist die folgende Regel, genannt **Zuweisungsaxiom**, für alle Formeln φ und Zuweisungen $x = t;$ ein Axiom:

$$\frac{}{\{\varphi_x^t\}\ x = t;\ \{\varphi\}}$$

b) **Sprachabhängige Nicht-Axiome:** Diese bestehen aus der Regel

$$\frac{\{\varphi \wedge b\}\ s_1\ \{\psi\} \qquad \{\varphi \wedge\ !(b)\}\ s_2\ \{\psi\}}{\{\varphi\}\ \texttt{if (}b\texttt{) \{}s_1\texttt{\} else \{}s_2\texttt{\}}\ \{\psi\}}$$

für die bewachte Anweisung, die Regel

$$\frac{\{I \wedge b\}\ s\ \{I\}}{\{I\}\ \texttt{while (}b\texttt{) \{}s\texttt{\}}\ \{I \wedge\ !(b)\}}$$

für die while-Schleife und die Regel

$$\frac{\{\varphi\}\ s_1\ \{\xi\} \qquad \{\xi\}\ s_2\ \{\psi\}}{\{\varphi\}\ s_1\ s_2\ \{\psi\}}$$

für die sequentielle Komposition von Anweisungen, wobei die in den Regeln auftauchenden Formeln, Terme und Anweisungen beliebig sind.

c) **Sprachunabhängige Nicht-Axiome (Konsequenzregeln genannt):** Deren gibt es zwei, nämlich die Regel

$$\frac{\varphi \rightarrow \psi \qquad \{\psi\}\ s\ \{\xi\}}{\{\varphi\}\ s\ \{\xi\}}$$

zur Abschwächung der Vorbedingung φ der Unterformel zur Vorbedingung ψ der Oberformel und die Regel

$$\frac{\{\varphi\}\ s\ \{\psi\} \qquad \psi \rightarrow \xi}{\{\varphi\}\ s\ \{\xi\}}$$

zur Verstärkung der Nachbedingung ξ der Unterformel zur Nachbedingung ψ der Oberformel. Dabei sind die in den Regeln auftauchenden Formeln und Anweisungen wiederum beliebig. ■

Die eben aufgeführten Regeln sind intuitiv sofort einsichtig. In Worten besagt etwa die Regel für die leere Anweisung, daß diese den Speicherzustand nicht verändert. Die erste Konsequenzregel liest sich wie folgt: Impliziert φ die Formel ψ und folgt, für einen gegebenen Speicherzustand, aus der Gültigkeit von ψ und der Terminierung von s die Gültigkeit von ξ, so gilt dies auch für φ an Stelle von ψ. Schließlich besagt die zweite Konsequenzregel, daß partielle Korrektheit erhalten bleibt, wenn man von der Nachbedingung zu einer schwächeren Formel übergeht, die weniger für das Resultat fordert.

Die entscheidende Regel, die etwa bei der Entwicklung von imperativen Programmen aus einer vorgegebenen Problemspezifikation das Einbringen einer (algorithmischen) Idee seitens des Programmierers erfordert, ist die Regel zur while-Schleife. In Worten besagt diese: Läßt die Ausführung des Rumpfs einer Schleife die Formel I invariant (d.h. erhält ihre Gültigkeit im neuen Speicherzustand), so gilt I im Speicherzustand nach Ausführung der Schleife (und natürlich auch die Negation der Schleifenbedingung), falls I im Speicherzustand vor der Ausführung gültig war. Man nennt deswegen I auch eine **Schleifeninvariante**. Auf den Zusammenhang zwischen Hoare-Kalkül und Programmentwicklung gehen wir am Ende dieses Abschnitts genauer ein.

6.4.6 Definition (Herleitungen im Hoare-Kalkül)

Die Menge der **Herleitungen** im Hoare-Kalkül mit jeweiliger **Wurzel** ist induktiv wie folgt definiert:

a) Jedes Axiom $\dfrac{}{B}$ ist eine Herleitung, und die Wurzel ist B.

b) Hat man eine Regel $\dfrac{A_1 \ldots A_n}{B}$ des Hoare-Kalküls und Herleitungen $\mathcal{H}_i,\ 1 \leq i \leq n$, mit jeweiligen Wurzeln $A_i,\ 1 \leq i \leq n$, so ist auch die folgende Figur eine Herleitung mit Wurzel B:

$$\frac{\mathcal{H}_1 \ldots \mathcal{H}_n}{B}$$

Eine Hoare-Formel $\{\varphi\}\, s\, \{\psi\}$ heißt (im Hoare-Kalkül) **herleitbar**, falls es eine Herleitung $\mathcal{H}$ mit Wurzel $\{\varphi\}\, s\, \{\psi\}$ gibt. ■

Herleitungen kann man also auch als Bäume interpretieren, bei denen Formeln oder Hoare-Formeln als Knotenmarkierungen angebracht sind. Diese sogenannten „Herleitungsbäume“ besitzen als Blattmarkierungen nur allgemeingültige Formeln, denn genau die Blätter entsprechen den Axiomen. Weiterhin gilt für jeden anderen Knoten (also jedes Nicht-Blatt), daß man von seiner Markierung die Markierungen seiner unmittelbaren Baumnachfolger bekommt, indem man eine Regel des Hoare-Kalküls von unten nach oben „liest“. Dies entspricht genau einer Regelanwendung. Die Wurzel eines Herleitungsbaums ist schließlich mit der hergeleiteten Formel markiert.

In Definition 6.4.3 haben wir festgelegt, daß die Schreibweise $\{\varphi\}\, s\, \{\psi\}$ dafür steht, daß das while-Programm s partiell-korrekt bezüglich der Vorbedingung φ und der Nachbedingung ψ ist. Dies heißt: **Hat man im Hoare-Kalkül $\{\varphi\}\, s\, \{\psi\}$ hergeleitet, so hat man gezeigt, daß s partiell-korrekt bezüglich φ und ψ ist.** Der Hoare-Kalkül ist also ein Werkzeug, um partielle Korrektheit zu beweisen.

Bevor wir nun konkrete Beispiele für Herleitungen von Hoare-Formeln im Hoare-Kalkül (also Programmverifikation mittels des Hoare-Kalküls) angeben und auch die Verbindung des Hoare-Kalküls zur Programmiermethodik und -entwicklung herstellen, ist noch eine Bemerkung zur Zuweisung angebracht.

6.4.7 Bemerkung (zur kollateralen Zuweisung)

In Abschnitt 6.3 hatten wir die kollaterale Zuweisung $x_1, \ldots, x_n$ `=` $t_1, \ldots, t_n$`;` als eine Pseudo-Anweisung eingeführt, die Programmherleitungen aus Repetitionen erleichtert, aber bei einem Übergang zu „echtem“ Java schließlich noch entfernt werden muß. Auch beim Hoare-Kalkül empfiehlt es sich, zuerst mit kollateralen Zuweisungen zu arbeiten (da dies viele Herleitungen verkürzt) und erst am Ende diese zu entfernen. Wir brauchen somit eine Regel für die kollaterale Zuweisung. Als Erweiterung des dritten Axioms von Definition 6.4.5 legen wir für kollaterale Zuweisungen fest:

$$\frac{}{\{\varphi^{t_1,\ldots,t_n}_{x_1,\ldots,x_n}\}\ x_1, \ldots, x_n \texttt{ = } t_1, \ldots, t_n\texttt{;}\ \{\varphi\}}$$

In diesem **erweiterten Zuweisungsaxiom** ist $\varphi^{t_1,\ldots,t_n}_{x_1,\ldots,x_n}$ diejenige Formel, die aus der Formel φ entsteht, indem man gleichzeitig alle Variablen x_i durch die entsprechenden Terme t_i ersetzt $(1 \leq i \leq n)$. ■

Nach dieser Bemerkung kommen wir nun zum angekündigten Beispiel, in dem, als Verallgemeinerung, auch kollaterale Zuweisungen in while-Programmen verwendet werden.

6.4.8 Beispiel (Verifikation eines gegebenen Programms)

Wir betrachten das nachfolgende while-Programm mit drei Variablen `n`, `a` und `res` jeweils von der Sorte `int`:

```
n,res = a,1;
while (n != 0)
  n,res = n-1,n*res;
```

Nach der Herleitung von Beispiel 6.3.6 wissen wir schon, daß der Wert von `res` nach der Abarbeitung gleich der Fakultät des nicht-negativen Werts von `a` ist. Wir wollen nun mit den neu vorgestellten Mitteln zeigen, daß das Programm total-korrekt ist bezüglich `a` $\geq$ 0 als Vorbedingung und `res` $=$ `a`! als Nachbedingung. Dabei ist die Fakultätsoperation als in der erweiterten Signatur existierend angenommen. Für diese Signatur setzen wir auch eine Produktbildungsoperation voraus, die wir in der üblichen Schreibweise $\prod_{i=a}^{b} x_i$ notieren.

Zum Korrektheitsbeweis verwenden wir Satz 6.4.4 und spalten die gestellte Aufgabe in zwei Teilaufgaben auf.

Die **Terminierung** des obigen Programms in jedem Speicherzustand, in dem `a` ≥ 0 gilt, ist offensichtlich, da der Wert der Variablen `n` in jedem Schleifendurchlauf echt vermindert wird und alle vorkommenden Anwendungen von Operationen (wegen deren Totalität) definiert sind.

Zum Beweis der **partiellen Korrektheit** verfolgen wir die Werte der Variablen des Programms durch einige Durchläufe der Schleife, um eine Gesetzmäßigkeit zu entdecken, die als Invariante geeignet ist. Dann erst führen wir den formalen Beweis im Hoare-Kalkül

durch. Beim Abarbeiten der while-Schleife ergibt sich die folgende Tabelle, in der wir auch gleich den vermuteten allgemeinen Fall angeben:

Nach i-tem Durchlauf	`n`	`res`
0	$\mathtt{a}$	1
1	$\mathtt{a}-1$	$\mathtt{a}$
2	$\mathtt{a}-2$	$(\mathtt{a}-1)*\mathtt{a}$
3	$\mathtt{a}-3$	$(\mathtt{a}-2)*(\mathtt{a}-1)*\mathtt{a}$
$\vdots$	$\vdots$	$\vdots$
i	$\mathtt{a}-i$	$\prod_{j=\mathtt{a}-i+1}^{\mathtt{a}} j$
$\vdots$	$\vdots$	$\vdots$

Dies gibt berechtigten Anlaß zur Vermutung, daß die Gleichung $\mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j$ die Invariante der Schleife ist.

Nun erst starten wir mit dem formalen Beweis von partieller Korrektheit, d.h. der Herleitung der Hoare-Formel

$$\{\mathtt{a} \geq 0\} \quad \begin{array}{l}\mathtt{n,res = a,1;}\\ \mathtt{while\ (n\ !=\ 0)}\\ \quad \mathtt{n,res = n-1, n*res;}\end{array} \quad \{\mathtt{res} = \prod_{j=1}^{\mathtt{a}} j\}$$

im Hoare-Kalkül. Zur Vereinfachung kürzen wir im folgenden die while-Schleife immer ab. Auch werden wir Hoare-Formeln in einer „senkrechten“ Schreibweise notieren und mit Teilen von Herleitungen arbeiten, um mit dem Platz auszukommen.

Als erstes erhalten wir die folgende Figur durch eine Anwendung der Konsequenzregel zur Verstärkung der Nachbedingung der Unterformel:

$$\frac{\begin{array}{l}\{\mathtt{a} \geq 0\}\\ \mathtt{n,res = a,1; while...;}\\ \{\mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j \wedge \mathtt{n == 0}\}\end{array} \qquad \begin{array}{r}\mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j \wedge \mathtt{n == 0}\\ \rightarrow \mathtt{res} = \prod_{j=1}^{\mathtt{a}} j\end{array}}{\{\mathtt{a} \geq 0\}\ \mathtt{n,res = a,1; while...;}\ \{\mathtt{res} = \prod_{j=1}^{\mathtt{a}} j\}}$$

Zur Gewinnung einer Gesamtherleitung brauchen wir also noch jeweils Herleitungen für die Hoare-Formel bzw. die Formel über dem Strich dieser Figur. Der Fall der rechten Oberformel ist einfach. Diese stellt offensichtlich eine allgemeingültige Formel dar und eine Herleitung ist somit das folgende Axiom:

$$\frac{}{\mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j \ \wedge\ \mathtt{n == 0} \rightarrow \mathtt{res} = \prod_{j=1}^{\mathtt{a}} j}$$

Bei der linken Oberformel der obigen Figur bekommen wir durch die Anwendung der Regeln für die sequentielle Komposition bzw. die while-Schleife die folgende Figur, da die doppelte Negation `!(n != 0)` offensichtlich gleichwertig zu $\mathtt{n == 0}$ ist:

$$\dfrac{\begin{array}{ll} & \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n != 0}\} \\ & \texttt{n,res = n - 1,n * res;} \\ & \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\} \\ \hline \{\texttt{a} \geq 0\} & \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\} \\ \texttt{n,res = a,1;} & \texttt{while...;} \\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\} & \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n == 0}\} \end{array}}{\{\texttt{a} \geq 0\}\ \texttt{n,res = a,1; while...;}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n == 0}\}}$$

Somit stehen noch zwei Herleitungen aus. Die linke Hoare-Formel der gerade erreichten Figur wird wie folgt hergeleitet, denn die beiden Oberformeln der nächsten Figur sind nach Definition 6.4.5 Axiome und die Unterformel folgt nach der ersten Konsequenzregel:

$$\dfrac{\dfrac{}{\texttt{a} \geq 0 \rightarrow 1 = \prod_{j=\texttt{a}+1}^{\texttt{a}} j} \quad \dfrac{}{\{1 = \prod_{j=\texttt{a}+1}^{\texttt{a}} j\}\ \texttt{n,res = a,1;}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}}{\{\ \texttt{a} \geq 0\}\ \texttt{n,res = a,1;}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}$$

Auch die obige rechte Hoare-Formel ist mittels dieser Konsequenzregel wie nachfolgend angegeben herleitbar, wobei die Axiom-Eigenschaft des linken Teilbaums trivial ist und im rechten Teilbaum $\texttt{n} - 1 + 1$ gleich zu $\texttt{n}$ vereinfacht wurde:

$$\dfrac{\dfrac{}{\begin{array}{l} \texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n != 0} \\ \rightarrow \texttt{n * res} = \prod_{j=\texttt{n}}^{\texttt{a}} j \end{array}} \quad \dfrac{\dfrac{}{\{\texttt{n * res} = \prod_{j=\texttt{n}}^{\texttt{a}} j\}}}{\begin{array}{l} \texttt{n,res = n - 1,n * res;} \\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\} \end{array}}}{\{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n != 0}\}\ \texttt{n,res = n - 1,n * res;}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}$$

Insgesamt haben wir also die folgende Herleitungsstruktur zur partiellen Korrektheit des oben angegebenen while-Programms zur Berechnung der Fakultät:

$$\dfrac{\dfrac{\dfrac{}{A_5} \quad \dfrac{}{A_6}}{A_3} \quad \dfrac{\dfrac{}{A_8} \quad \dfrac{}{A_9}}{A_7}}{\dfrac{A_4}{\dfrac{A_2}{A_1}}}$$

Die Arbeitsrichtung ist von unten nach oben, in der Sprache der Bäume also von der Wurzel zu den Blättern, was beweistechnisch einer reduzierenden Vorgehensweise entspricht. Die entscheidenden Tätigkeiten passieren dabei in den ersten beiden Schritten und betreffen die Verstärkung der Nachbedingung und das Einführen einer Invariante. ■

Im letzten Beispiel haben wir ein gegebenes Programm bezüglich gegebener Vor- und Nachbedingung verifiziert. Im allgemeinen ist die Situation jedoch anders. Man hat eine **(formale) Problemspezifikation** vorliegen als Paar

- φ : Vorbedingung (was sind die Voraussetzungen an die Eingabe)
- ψ : Nachbedingung (welche Werte sollen berechnet und in welchen Variablen abgespeichert werden)

von Formeln und sucht ein while-Programm s mit $\{\varphi\}\, s\, \{\psi\}$ und welches bei Gültigkeit von φ terminiert. Auch hierzu kann man den Hoare-Kalkül verwenden, indem man die Regeln des Systems „rückwärts liest“ und gemäß der Oberformeln s schrittweise verfeinert. Man entwickelt also ein Programm aus einer Spezifikation Hand in Hand mit seinem Korrektheitsbeweis, wobei die Richtung, wie es weitergeht, normalerweise durch den Beweis bestimmt wird. Diese Vorgehensweise wird auch **Dijkstra-Gries-Methode** genannt, weil sie insbesondere von diesen beiden Informatikern in Form von Lehrbüchern und zahlreichen wissenschaftlichen Aufsätzen propagiert wurde. Der Leser sei etwa auf das Buch „The Science of Computer Programming“ von D. Gries (Springer-Verlag, 1981) verwiesen.

Um das Vorgehen bei der Dijkstra-Gries-Methode zu erleichtern, extrahiert man nun die wesentlichen immer wiederkehrenden Tätigkeiten in Form von Theoremen mit sogenannten **Beweisverpflichtungen** heraus. Diese Theoreme hängen von der Form der Programme ab, die man erzielen will. Ein sehr wichtiges Beispiel, welches eine große Klasse von Programmen abdeckt, ist nachfolgend angegeben.

6.4.9 Satz (Entwurf von Programmen)

Eine Hoare-Formel der Form

$$\{\varphi\}\ \mathit{init}\ \texttt{while (}b\texttt{) \{}\mathit{body}\texttt{\}}\ \{\psi\}$$

mit Anweisungen *init* (der Initialisierung) und *body* (dem eigentlichen Schleifenrumpf) ist herleitbar, falls, mit einer Formel (der Invariante) I die folgenden drei Beweisverpflichtungen gelten:

(1) Die Formel $I \wedge \texttt{!(}b\texttt{)} \rightarrow \psi$ ist allgemeingültig.

(2) Die Hoare-Formel $\{\varphi\}\ \mathit{init}\ \{I\}$ ist herleitbar.

(3) Die Hoare-Formel $\{I \wedge b\}\ \mathit{body}\ \{I\}$ ist herleitbar.

Beweis: Man ergänzt die nach den Beweisverpflichtungen (2) und (3) existierenden Herleitungen zu einer Gesamtherleitung wie folgt:

$$\dfrac{\dfrac{\dfrac{(2)}{\vdots}}{\{\varphi\}\ \mathit{init}\ \{I\}} \qquad \dfrac{\dfrac{\dfrac{(3)}{\vdots}}{\{I \wedge b\}\ \mathit{body}\ \{I\}}}{\{I\}\ \texttt{while (}b\texttt{) \{}\mathit{body}\texttt{\}}\ \{I \wedge \texttt{!(}b\texttt{)}\}}}{\{\varphi\}\ \mathit{init}\ \texttt{while (}b\texttt{) \{}\mathit{body}\texttt{\}}\ \{I \wedge \texttt{!(}b\texttt{)}\}} \qquad \dfrac{}{I \wedge \texttt{!(}b\texttt{)} \rightarrow \psi}}{\{\varphi\}\ \mathit{init}\ \texttt{while (}b\texttt{) \{}\mathit{body}\texttt{\}}\ \{\psi\}}$$

Man beachte dabei, daß, nach der Beweisverpflichtung (1), die rechte Oberformel des ersten Schritts von unten nach oben ein Axiom ist. ■

Dieser Satz beschreibt die folgende Programmentwicklungstechnik: Liegen Vorbedingung und Nachbedingung als Problemspezifikation vor, und hat man sich für eine Programmstruktur entschieden, die aus einer Initialisierung und einer nachfolgenden while-Schleife besteht, so gehe man wie folgt vor:

a) Zuerst entwickle man eine Verstärkung der Nachbedingung, welche die Invariante und die Schleifenbedingung der zu konstruierenden while-Schleife liefert.

b) Anschließend entwickle man eine Initialisierung, welche die Invariante unter der Vorbedingung etabliert.

c) Schließlich entwickle man einen Schleifenrumpf, dessen Ausführung die Gültigkeit der Invariante aufrechterhält.

Eine Technik, die bei Schritt a) oft hilft, ist die **Generalisierung der Nachbedingung** durch Einführung von zusätzlichen Variablen. Diese werden in der Regel an Stelle von (geeigneten) Konstanten verwendet, beispielsweise unteren oder oberen Grenzen von Intervallen. Bei Schritt b) orientiert man sich in der Regel an der Abbruchbedingung der while-Schleife. Schließlich konzentriert man sich normalerweise bei Schritt c) zuerst auf einen Teil des Schleifenrumpfs und versucht dabei, die Terminierung der while-Schleife zu erreichen. Die Abbruchbedingung der while-Schleife ist hier normalerweise wiederum entscheidend. Erst dann rechnet man den Rest des Schleifenrumpfs aus.

Das eben angegebene Verfahren kann natürlich auch bei der Konstruktion der Initialisierung und des Schleifenrumpfs wiederum angewendet werden, so daß beispielsweise auch Programme mit geschachtelten oder hintereinanderstehenden while-Schleifen auf diese Weise entwickelt werden können.

Im nächsten Beispiel zeigen wir, wie man mit dem Hoare-Kalkül und dem eben beschriebenen Verfahren ein Programm korrekt aus einer Spezifikation entwickeln kann, wobei die Spezifikation aus der Vor- und der Nachbedingung besteht.

6.4.10 Beispiel (Entwicklung eines Programms)

Gegeben sei die Aufgabe, ein while-Programm zur Bestimmung von Fibonacci-Zahlen zu entwickeln, also eine Anweisung s, so daß die nachfolgende Hoare-Formel herleitbar ist:

$$\{\mathtt{a} \geq 0\}\ s\ \{\mathtt{x} = \mathit{fib}(\mathtt{a})\}$$

In dieser Hoare-Formel sind `a` und `x` zwei Variablen der Sorte `int` zur Bereitstellung der Eingabe bzw. zum Abspeichern des Resultats und *fib* ist eine Funktion/Operation der erweiterten Signatur zur Berechnung der Folge der Fibonacci-Zahlen, d.h. definiert durch die beiden nachstehenden Gleichungen:

$$\mathit{fib}(0) = \mathit{fib}(1) = 1 \qquad\qquad \mathit{fib}(n+2) = \mathit{fib}(n+1) + \mathit{fib}(n)$$

Man vergleiche auch mit Beispiel 4.5.10.b, wo wir ein ML-Programm zur Berechnung von Fibonacci-Zahlen entwickelt haben. Weiterhin beachte man, daß $\mathit{fib}(\mathtt{a})$ kein Java-Term ist – was wir dadurch anzeigen, daß *fib* und die Klammern bei der Anwendung von *fib* nicht in Schreibmaschinenschrift gesetzt sind.

Zur Berechnung der Werte der Operation *fib* mittels eines while-Programms orientieren wir uns genau an der oben angegebenen Rekursion und verwenden neben `a` und `x` noch zwei weitere Variablen zur Tabellierung der Funktionswerte von *fib* für zwei aufeinanderfolgende Argumente.

Formal setzen wir die Programmentwicklung aus der Vor-/Nachbedingungs-Problemspezifikation $(\mathtt{a} \geq 0, \mathtt{x} = \mathit{fib}(\mathtt{a}))$ an mit der offensichtlich allgemeingültigen Formel

$$\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1) \wedge \mathtt{u == a} \;\rightarrow\; \mathtt{x} = \mathit{fib}(\mathtt{a}),$$

wobei `u` und `y` die neuen Variablen der Sorte `int` sind. Mit Blick auf diese allgemeingültige Formel und auf die Beweisverpflichtung (1) von Satz 6.4.9 können wir nun als Invariante I und als Schleifenbedingung b versuchen

$$I \;\stackrel{\wedge}{=}\; \mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1) \qquad\qquad b \;\stackrel{\wedge}{=}\; \mathtt{u != a},$$

da der Java-Term (die negierte Schleifenbedingung) `!(u != a)` äquivalent ist zur Gleichung `u == a`. Damit gilt die Beweisverpflichtung (1) von Satz 6.4.9 mit $\mathtt{x} = \mathit{fib}(\mathtt{a})$ als Nachbedingung ψ, und unser zukünftiges while-Programm hat die folgende schematische Form (wobei die beiden geschweiften Klammern fehlen dürfen, wenn es uns gelingt, *body* ohne sequentielle Komposition zu realisieren:

init
`while (u != a) {` *body*`}`

Es bleiben noch die Initialisierung und der Schleifenrumpf zu finden, so daß auch die restlichen Beweisverpflichtungen (2) und (3) von Satz 6.4.9 ebenfalls gelten.

Wir setzen für die Initialisierung *init* die nachfolgende kollaterale Zuweisung an:

```
x,y,u = 1,1,0;
```

Der Ansatz $\mathtt{u} = 0$ ist motiviert durch die Bedingung `u != a` der while-Schleife. D.h. man wird sich der Eingabe `a` vom Startpunkt Null ausgehend von „unten“ her nähern. Mit `u = 0` als Start ergibt sich die Initialisierung der restlichen zwei Variablen `x` und `y` mittels `1` unmittelbar aus der Rekursion der Fibonacci-Zahlen. Der formale Beweis der Beweisverpflichtung (2) von Satz 6.4.9 verwendet die erste Konsequenzregel in Verbindung mit dem Terminierungsfall der Fibonacci-Rekursion und sieht wie folgt aus:

$$\dfrac{\mathtt{a} \geq 0 \rightarrow \mathtt{1} = \mathit{fib}(\mathtt{0}) \wedge \mathtt{1} = \mathit{fib}(\mathtt{1}) \qquad \dfrac{}{\{\mathtt{1} = \mathit{fib}(\mathtt{0}) \wedge \mathtt{1} = \mathit{fib}(\mathtt{1})\}\ \mathtt{x,y,u = 1,1,0;}\ \{\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1)\}}}{\{\mathtt{a} \geq 0\}\ \mathtt{x,y,u = 1,1,0;}\ \{\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1)\}}$$

Es bleiben also noch die Entwicklung des Schleifenrumpfs und die Verifikation der Beweisverpflichtung (3) von Satz 6.4.9.

Da Null der Initialwert der Variablen `u` ist, müssen wir von `u` nach `u+1` übergehen, um irgendwann schließlich `a` zu treffen, d.h. Terminierung zu erreichen. Dazu ist die Vorbedingung $\mathtt{a} \geq 0$ natürlich wesentlich. Offensichtlich gilt für diese Abänderung von `u` zu `u+1` die Implikation

$$\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1) \implies \mathtt{y} = \mathit{fib}(\mathtt{u+1}) \wedge \mathtt{x+y} = \mathit{fib}(\mathtt{u+1}+1), \qquad (*)$$

da die Gleichung

$$\mathtt{x}+\mathtt{y} = \mathit{fib}(\mathtt{u}) + \mathit{fib}(\mathtt{u}+1) = \mathit{fib}(\mathtt{u+1}+1)$$

nach der oben angegebenen Fibonacci-Rekursion zutrifft. Also haben wir als Ansatz für den Schleifenrumpf die folgende kollaterale Zuweisung:

```
x,y,u = y,x+y,u+1;
```

Ein formaler Nachweis der Beweisverpflichtung (3) von Satz 6.4.9 verwendet die eben angegebene Implikation $(*)$ und, wie beim Nachweis von (2), wiederum die erste Konsequenzregel und ist nachstehend angegeben:

$$\dfrac{\begin{array}{l}\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1) \wedge \mathtt{u\ !=\ a} \\ \rightarrow \\ \mathtt{y} = \mathit{fib}(\mathtt{u+1}) \wedge \mathtt{x+y} = \mathit{fib}(\mathtt{u+1}+1)\end{array} \qquad \begin{array}{l}\{\mathtt{y} = \mathit{fib}(\mathtt{u+1}) \wedge \mathtt{x+y} = \mathit{fib}(\mathtt{u+1}+1)\} \\ \mathtt{x,y,u\ =\ y,x+y,u+1;} \\ \{\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \mathtt{y} = \mathit{fib}(\mathtt{u}+1)\}\end{array}}{\left\{\begin{array}{l}\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \\ \mathtt{y} = \mathit{fib}(\mathtt{u}+1) \wedge \mathtt{u\ !=\ a}\end{array}\right\}\ \mathtt{x,y,u\ =\ y,x+y,u+1;}\ \left\{\begin{array}{l}\mathtt{x} = \mathit{fib}(\mathtt{u}) \wedge \\ \mathtt{y} = \mathit{fib}(\mathtt{u}+1)\end{array}\right\}}$$

Insgesamt haben wir also für s das folgende while-Programm entwickelt, welches für einen nichtnegativen Eingabewert von `a` terminiert und in dem alle verwendeten Variablen `x`, `y`, `u` und `a` die gleiche Sorte `int` besitzen:

```
x,y,u = 1,1,0;
while (u != a)
  x,y,u = y,x+y,u+1;
```

Die Sequentialisierung der Initialisierung dieses Programms ist trivial. Gleiches gilt auch für den Schleifenrumpf, trotz der zyklischen Abhängigkeit von `x` und `y`. Vervollständigt man das sequentialisierte Programm zu einem Java-Programm, wie wir es schon von den Beispielen 6.3.6 her kennen, so bekommt man schließlich das folgende Resultat:

```
public class fibonacci {
  private static int IntRead () {
    String s = "";
    try {s = new java.io.DataInputStream(System.in).readLine();}
    catch(java.io.IOException e) {;}
    return java.lang.Integer.parseInt(s); }
```

```
public static void main (String[] args) {
  int a, x, y, u, h;
  a = IntRead();
  u = 0;
  y = 1;
  x = 1;
  while (u != a) {
    u = u + 1;
    h = y;
    y = x + y;
    x = h; }
  System.out.println(x); }
}
```

Man beachte, daß zum Gewinnen des Rumpfs von `main` aus dem obigem while-Programm bei der Sequentialisierung der kollateralen Zuweisung der while-Schleife eine Hilfsvariable `h` verwendet wurde. Theoretisch ginge es auch ohne die Variable `h`, indem man

```
x,y,u = y,x+y,u+1;
```

mittels der Zuweisungsfolge

```
u = u+1; y = x+y; x = y-x;
```

sequentialisiert. Solche Tricks machen Programme aber nur undurchsichtig, schwer verständlich und deshalb auch schwer verifizierbar. Sie haben somit in der Informatik nichts verloren. ■

Zum Schluß dieses Abschnitts ist noch eine Bemerkung angebracht: Wir haben in den letzten Beispielen die Verifikation bzw. Herleitung eines Programms sehr formal durchgeführt und jeweils die entsprechenden Herleitungen im Hoare-Kalkül angegeben. In der Praxis der Programmierung wendet man den Kalkül in der Regel aber wesentlich „freier“ an und beweist hier „wie in der Mathematik üblich“. Wenn man den Hoare-Kalkül und seine Verwendung verstanden hat, so ist dies ungefährlich, weil man dann jeden „üblichen“ Beweis ohne große Schwierigkeiten in eine formale Herleitung im Hoare-Kalkül übertragen kann. Ist man hingegen an theoretischen Eigenschaften von Programmverifikation[28] oder an ihrer maschinellen Unterstützung durch entsprechende Werkzeuge interessiert, so wird man um den formalen, kalkülhaften Ansatz nicht vorbeikommen.

Wir wollen dieses Kapitel mit einer Programmverifikation beenden, die nicht im formalen Hoare-Kalkül durchgeführt ist, sondern „wie in der Mathematik üblich“. Mit ihr rechtfertigen wir nachträglich die früher angegebenen Transformationen von repetitiven Rekursionen zu while-Schleifen. Zur Vereinfachung betrachten wir dabei nur den Fall einer einstelligen Funktion.

[28] Ein Beispiel hierzu ist die Frage nach der sogenannten Vollständigkeit: Kann man eine Herleitung von $\{\varphi\}\ s\ \{\psi\}$ im Hoare-Kalkül finden, wenn s partiell korrekt ist bezüglich der Vorbedingung φ und der Nachbedingung ψ?

6.4.11 Beispiel (Informelle Programmverifikation; siehe 6.3.5)

Die (partielle) Funktion $f : M \longrightarrow N$ sei mit Hilfe von drei in Java implementierten (partiellen) Funktionen $b : M \longrightarrow \mathbb{B}$, $k : M \longrightarrow M$ und $l : M \longrightarrow N$ durch

$$f(x) = \begin{cases} f(k(x)) & : \quad b(x) = tt \\ l(x) & : \quad b(x) = ff \end{cases}$$

repetitiv-rekursiv beschrieben. Wir wollen zeigen, daß das Programmstück

```
x = a;
while (b(x))
  x = k(x);
y = l(x);
```

partiell-korrekt ist bezüglich ist *true* als Vorbedingung und $y = f(a)$ als Nachbedingung. Als Schleifeninvariante $I(x)$ wählen wir $f(x) = f(a)$. Um zu zeigen, daß die Initialisierung die Schleifeninvariante etabliert, haben wir zu beweisen, daß $I(a)$ gilt. Dies ist aber trivial:

$$I(a) \iff f(a) = f(a) \iff true$$

Der zweite Schritt der Programmverifikation ist der Nachweis, daß bei jedem Schleifendurchlauf die Gültigkeit der Schleifeninvariante erhalten bleibt. Es gelte also $I(x) \wedge b(x)$. Dann folgt daraus

$$I(k(x)) \iff f(k(x)) = f(a) \iff f(x) = f(a) \iff I(x) \iff true$$

nach der Definition der Schleifeninvariante, der Definition von f im Fall $b(x)$ und der Gültigkeit von $I(x)$.

Es bleibt schließlich noch zu verifizieren, daß aus der Gültigkeit der Schleifeninvariante und der Negation der Schleifennbedingung in Kombination mit der abschließenden Zuweisung die Nachbedingung folgt. Auch dies ist nicht schwierig: Es gelte $I(x) \wedge \,!b(x)$. Dann bekommen wir, indem wir zuerst die abschließende Zuweisung verwenden, dann die Definition von f im Fall $!b(x)$ und schließlich die Gültigkeit von $I(x)$:

$$y = l(x) = f(x) = f(a)$$

Dem Leser sei zur Übung empfohlen, die eben durchgeführte Programmverifikation in eine formale Herleitung im Hoare-Kalkül zu übertragen. ■

6.5 Einschub: Zur Benutzung des Java-Systems

Zum Abschluß dieses Kapitels soll noch kurz beschrieben werden, wie das Java-System zu benutzen ist. Wie bei der Beschreibung des SML97-Systems gehen wir dabei natürlich nicht auf alle Möglichkeiten ein, sondern nur auf diejenigen, die für die Programme dieses Kapitels notwendig sind. Auch berufen wir uns auf die Installation an den Rechnern des Instituts für Informatik und Praktische Mathematik der Universität Kiel, die in der Regel von anderen Installationen abweichen kann.

6.5.1 Allgemeines

Um die derzeit aktuellste Version des Java-Systems an den Rechnern des Instituts für Informatik und Praktische Mathematik der Universität Kiel verwenden zu können, ist zuerst die Pfadvariable `PATH` des Betriebssystems Sun Solaris wie folgt zu verändern:

```
setenv PATH /home/java/jdk1.4/bin\:$PATH
```

Nach dieser Änderung der Pfadvariablen werden die einzelnen Befehle des Java-Systems ausführbar.

6.5.2 Das Arbeiten mit dem Java-System

Wir gehen zuerst davon aus, daß ein Java-Programm mit einer einzigen Klasse vorliegt, welches in einer Datei untergebracht ist. Diese Klasse enthält die Methode `main` und der Dateiname ist der Klassenname ergänzt um die Endung `.java`, falls die Klasse mittels des Modifizierers `public` als sichtbar erklärt ist. Ein Beispiel hierfür ist das Programm zum Testen von perfekten Zahlen in Beispiel 6.3.6.b). Da der Name der Hauptprogramm-Klasse hier `isperfect` ist, muß die das Java-Programm enthaltene Datei folglich `isperfect.java` heißen.

Die Übersetzung des Java-Programms in der Datei `isperfect.java` erfolgt durch den folgenden Befehl:

```
javac isperfect.java
```

Durch den Übersetzungsvorgang wird eine Datei `isperfect.class` erzeugt. Weiterhin werden die zwei folgenden Meldungen ausgegeben:

```
Note: isperfect.java uses or overrides a deprecated API.
Note: Recompile with -deprecation for details.
```

Die erste Meldung besagt, daß unser Programm eine sogenannte API (eine Anwenderprogramm-Schnittstelle) verwendet, welche in der nächsten Version des Java-Systems nicht mehr vorhanden sein wird. Durch die zweite Meldung wird darauf hingewiesen, wie man mit der zusätzlichen Option `deprecation` beim Aufruf von `javac` genauere Hinweise bekommen kann[29].

Ist ein Java-Programm syntaktisch nicht korrekt, so werden beim Übersetzen die Fehler durch das Java-System zusammen mit der Angabe der entsprechenden Zeilen und Spalten aufgeführt. Weil wir dies von ML her schon kennen, verzichten wir auf die Angabe eines

[29] Es handelt sich übrigens um die Methode `readLine` der Klasse `java.io.DataInputStream`, die in `IntRead` verwendet wird. Eine Version, die zu keiner Warnung führt, ist beispielsweise möglich, indem man in `IntRead` die Anweisung `s = new java.io.DataInputStream(System.in).readLine();` durch `s = new java.io.BufferedReader(new java.io.InputStreamReader(System.in)).readLine();` ersetzt.

Beispiels. Das Programm ist dann entsprechend zu korrigieren und der Übersetzungsvorgang zu wiederholen.

Nach dem erfolgreichen Übersetzen des Programms erfolgt nun der Aufruf durch die Eingabe des folgenden Befehls:

```
java isperfect
```

Man gibt dann eine Zahl ein und enthält die entsprechende Antwort, womit die Programmausführung beendet ist. Weitere Ausführungen sind auf die gleiche Weise ohne nochmaliges Übersetzen möglich. Erst wenn die Datei `isperfect.class` gelöscht wird, ist ein neuer Übersetzungsvorgang nötig.

Man beachte, daß beim Befehl zur Programmausführung die Datei-Endung `.java` und auch die Datei-Endung `.class` nicht mehr erwähnt werden darf im Gegensatz zum Befehl `javac` für die Übersetzung, wo `.java` explizit erwähnt werden muß.

Nun gehen wir davon aus, daß ein Java-Programm in mehreren Dateien untergebracht sei, wobei eine Datei auch mehrere Deklarationen von Klassen enthalten darf. Wie schon erwähnt, darf in jeder dieser Dateien höchstens eine Klasse mit dem Modifizierer `public` versehen sein. Die Namen dieser Klassen müssen dann den jeweiligen Dateinamen ohne der `java`-Endung entsprechen. Sinnvoll ist zudem, daß in genau einer Klasse des gesamten Programms die Methode `main` vorkommt (obwohl Java mehrere Vorkommen von `main` erlaubt) und diese Hauptprogramm-Klasse mittels `public` als sichtbar deklariert ist.

Es ist möglich, die Programme der einzelnen Dateien analog zur obigen Vorgehensweise zu übersetzen. Dabei werden bei der Übersetzung eines Programms alle Programme mitübersetzt, d.h. entsprechende `class`-Dateien erzeugt, von denen – auch mittelbar – Klassen verwendet werden.

Wir nehmen nun an, daß die Datei `MainClass.java` die sichtbare Hauptprogramm-Klasse `MainClass` mit der Methode `main` enthält. Dann genügt es bei einer vernünftigen Architektur des Gesamtprogramms mittels

```
javac MainClass.java
```

den Programmteil aus `MainClass.java` zu übersetzen und genau diesen dann auch mit Hilfe des Befehls

```
java MainClass
```

auszuführen.

7 Einige abschließende Bemerkungen

Wir haben uns in dem vorliegenden Skriptum zur Vorlesung Informatik I hauptsächlich mit der Einführung in die Programmierung beschäftigt. Um diese wohl wichtigste Disziplin der Informatik dem Leser näherbringen zu können, benötigten wir Programmiersprachen. Wir haben ML und Java ausgewählt.

Im Grunde genommen spielte die Wahl der Programmiersprachen eine eher untergeordnete Rolle, obwohl es natürlich Sprachen unterschiedlicher Ausprägung und Qualität gibt. Statt ML hätten wir beispielsweise auch Haskell oder Scheme und statt Java beispielsweise auch Eiffel oder Sather nehmen können. Dem aufmerksamen Leser ist nämlich sicherlich nicht entgangen, daß Konzepte und Methoden des Programmierens und ihre mathematische Fundierung im Zentrum der Betrachtungen standen und die verwendeten Programmiersprachen nur als Hilfsmittel zur Verdeutlichung und Formalisierung dienten. Wer die Konzepte und Methoden des Programmierens beherrscht, kann sich im allgemeinen sehr schnell in jede neue Sprache einarbeiten. Hat man hingegen das Programmieren nur orientiert an einer bestimmten Sprache gelernt, so wird man später, wie die Erfahrung zeigt, in der Regel in deren Schemata denken[30].

Der von uns gewählte Zugang hatte zwei Konsequenzen. Beide Programmiersprachen wurden nicht in jedem Detail vorgestellt, sondern nur soweit, wie es der Sache dienlich war. Aus diesem Grund haben wir etwa bei ML auf die sogenannten Records und auch das Sprachkonstrukt `local` und bei Java auf die so beliebten Inkrement- und Dekrementoperatoren ++ und −− verzichtet. Weiterhin wurde in Kauf genommen, daß gewisse Sprachkonstrukte im Skriptum eine Bedeutung haben, die in dem jeweiligen Kontext zwar gerechtfertigt und auch vernünftig ist, aber nicht der eigentlichen Intention der Sprache entspricht. Ein Beispiel hierzu ist die Angabe des Resultattyps bei ML-Rechenvorschriften in der von uns gewählten Weise. In ML hat ein der Parameterliste nachgestellter Typ eigentlich eine andere Bedeutung. Ein weiteres Beispiel ist die Zuweisung in Java, die eigentlich einen Term darstellt, dessen Auswertung als Seiteneffekt den Speicherzustand ändert.

Im Vordergrund des Skriptums stand das sogenannte Programmieren-im-Kleinen, also der Entwurf von Programmen, bei denen die sogenannte sequentielle Algorithmik entscheidend ist und die üblicherweise ohne Schwierigkeiten in einer Datei von noch überschaubarer Größe untergebracht werden können. Das Programmieren-im-Großen, bei dem algorithmische Ideen eher zweitrangig sind und stattdessen Modellierung, Strukturierung, Modularisierung und Abstraktion eine große Rolle spielen, wurde zwar in den beiden Kapiteln 5 und 6 in den Grundzügen vorgestellt, eine hinreichende Vertiefung war jedoch aus zeitlichen Gründen nicht möglich. Wegen der notwendigerweise kleinen oder höchstens mittelgroßen Beispiele scheint Programmieren-im-Großen im Rahmen einer Anfängervorlesung sowieso eher schlecht zu unterrichten zu sein, insbesondere, wenn dabei auch noch fensterorientierte Oberflächen usw. ins Spiel kommen. Hier ist ein auf gewisse Grundlagen

[30] P. Pepper drückt dies in seinem, gerade für den Anfänger sehr zu empfehlenden, Buch „Grundlagen der Informatik“ (Oldenburg-Verlag, 2. Auflage, 1995) wie folgt aus: Wer nur einen Hammer hat, für den sieht die ganze Welt wie ein Nagel aus.

(wie sie etwa in dieser Vorlesung Informatik I gelegt wurden) aufbauendes Programmierpraktikum mit einer geeigneten zentralen Unterrichtseinheit – die keine Vorlesung im üblichen Sinne ist – sicher wesentlich geeigneter und solch eine Veranstaltung folgt in der Tat im 3. Semester als „Softwarepraktikum“.

Ebenfalls aus zeitlichen Gründen kamen die Datenstrukturen viel zu kurz, welche bei der Programmierung natürlich mit eine entscheidende Rolle spielen. Dies gilt insbesondere für die imperative Programmierung, wo wir nicht einmal auf die äußerst wichtigen Felder eingehen konnten. Viel zu kurz kamen auch allgemeine Betrachtungen zur Analyse von Algorithmen im Hinblick auf die Laufzeit und den Speicherplatz (z.B. schlechtester Fall bzw. durchschnittlicher Fall). Neben der fehlenden Zeit war dies durch die in der Regel noch nicht genügende mathematische Erfahrung und Ausbildung bedingt. Die eben aufgezählten Punkte stellen aber kein Manko von Informatik I dar, da gerade effiziente Datenstrukturen, ihre Verwendung im Rahmen von effizienten Algorithmen und die Analyse von Algorithmen den hauptsächlichen Stoff der Vorlesung Informatik II bilden und dann auch die entsprechenden mathematische Grundlagen vorhanden sind.