Informatik I und II

Rudolf Berghammer

Institut für Informatik und Praktische Mathematik
Christian-Albrechts-Universität Kiel
Preusserstraße 1-9
24105 Kiel

Skriptum zu den gleichnamigen Vorlesungen, abgehalten
in den Studienjahren 1994 / 95, 1996 / 97 und 1998 / 99 an der
Christian-Albrechts-Universität zu Kiel.

Inhaltsverzeichnis

1 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, Telephone), 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 Hauptstudium, d.h. nach dem Vordiplom, werden einzelne Themen der obigen drei Punkte in Spezialvorlesungen behandelt. Beispiele sind Rechnerorganisation, Aufbau digitaler Rechenanlagen, Gomputergraphik und Computer Vision zur technischen Informatik, Übersetzerbau, Datenbanksysteme, Betriebssysteme und Datenverfeinerung zur praktischen Informatik und Formale Sprachen, Automatentheorie, Komplexitätstheorie, Programmlogiken und Semantik von Programmiersprachen zur theoretischen Informatik. Dabei kann man einige Vorlesungen auch einem anderen Bereich zuordnen, wie etwa Semantik von Programmiersprachen der praktischen Informatik oder Datenbanksysteme der theoretischen Informatik.

In einer Grundvorlesung über Informatik, d.h. einer Vorlesung vor dem Vordiplom, wird in Deutschland derzeit noch üblicherweise ein Einstieg in und Überblick über ihre Teilbereiche gegeben. Dies erleichtert einerseits den Einstieg in das in der Regel neue Studienfach, erlaubt es andererseits aber 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 und II auf dem Gebiet der Programmierung liegt. Dieses Skriptum entstand aus den Vorlesungen Informatik I und II, wie sie in Kiel in den Studienjahren 1994/95 und 1996/97 abgehalten wurden. Sehr knapp stellt sich der Inhalt wie folgt dar:

- Zuerst stellen wir im ersten Teil, den nachfolgenden zwei Kapiteln, viele Grundbegriffe und Notationen bereit, die teilweise auch in den Anfängervorlesungen der Mathematik gebracht werden, teilweise aber informatikspezifisch sind. Es sollte schon an dieser Stelle darauf hingewiesen werden, daß sich der Gebrauch des sich am Aufbau der Objekte orientierenden Prinzips der Induktion (auch in der die betrachteten Objekte „abbauenden“ Form der Rekursion) und termmäßiges Rechnen – zwei Begriffe, die in diesem Teil auf die Schulkenntnisse aufbauend eingeführt werden – wie ein roter Faden durch das Skriptum zieht.

- Dann befassen wir uns in vier Kapiteln, dem zweiten Teil der Vorlesung, mit Programmiersprachen und Programmierung, wobei wir zwei Stilrichtungen unterscheiden, nämlich funktionale und imperative Sprachen bzw. Programmierung. Bei jedem Stil betrachten wir in einem ersten Kapitel zuerst die Grundlagen, also elementare Sprachkonzepte und Programmierfertigkeiten. Diese werden dann im nächsten Kapitel jeweils vertieft, d.h. weitere Sprachkonzepte eingeführt, die dann ausgereiftere Programmiertechniken erlauben.

- Im Verlaufe des zweiten Teils lernen wir Programmiermethodik und -entwicklung an vielen kleinen Beispielen kennen. Im dritten Teil der Vorlesung, der nur aus einem Kapitel besteht, betrachten wir schließlich noch komplexere Beispiele und Algorithmen, etwa schnelle Sortierverfahren und schnelle Suchverfahren in Datenmengen mittels geeigneter baumartiger Darstellungen, und gehen auch etwas auf Effizienzfragen und die Grenzen der Algorithmisierbarkeit ein.

Das Skriptum enthält auch einen Anhang. In ihm werden, als Ergänzung des zweiten Teils, in gedrängter Weise Sprachkonstrukte vorgestellt, mit denen man große Programmsysteme auf sichere Weise inkrementell konstruieren kann. Dieses Thema wird in einer Vorlesung über Informatik im ersten Studienjahr in der Regel nur sehr knapp oder auch garnicht angesprochen, da es für eine umfassende Behandlung zu diesem Zeitpunkt viel zu umfangreich ist.

Die Vorlesungen Informatik I und II (als Teil des sogenannten A-Strangs der Informatik-Grundvorlesungen) erstrecken sich über ein Jahr und wenden sich an die Studienanfänger des Diplomstudiengangs Informatik und der Diplomstudiengänge Mathematik und Physik mit Nebenfach Informatik. Parallel zu ihnen werden jeweils die Vorlesungen Informatik I und II für Ingenieure der Fachrichtungen Elektrotechnik, Informatik und Materialwissenschaft (als Teil des B-Strangs der Grundvorlesungen) abgehalten, wo in der zweiten Vorlesung – *die im Gegensatz zur ersten Vorlesung des B-Strangs auch pflichtig ist für Studienanfänger des Diplomstudiengangs Informatik* – die Grundzüge der technischen Informatik behandelt werden. Die eben genannten Grundvorlesungen werden im zweiten Studienjahr fortgesetzt durch Vorlesungen, in denen unter anderem die Grundlagen der Programmübersetzung und Betriebssystemkonzepte sowie einige wichtige Gebiete der theoretischen Informatik besprochen werden.

Die Aufteilung der Grundvorlesungen in Informatik in mehrere Stränge ist durch die spezielle Kieler Situation bedingt – mit Studierenden sowohl der Diplom-Informatik als auch der Ingenieur-Informatik im Hauptfach Informatik. Diese beiden Studiengänge bereiten die Studierenden auf unterschiedliche Tätigkeiten im späteren Berufsleben vor. Der eher software-orientierte Diplominformatiker wird dann hauptsächlich Programme für die verschiedensten Anwendungen entwickeln, während der Ingenieurinformatiker eher hardwaregeprägt ist und in einem Umfeld arbeiten wird, das von computerbasierten technischen Systemen bestimmt ist. Es sollte an dieser Stelle noch erwähnt werden, daß die Universität Kiel noch eine dritte Anfängervorlesung in Informatik anbietet (den C-Strang), und zwar als Teil eines „studium generale“ oder für Ausbildungsgänge, wie beispielsweise Naturwissenschaften (außer Physik), Wirtschaftswissenschaften und Magisterstudiengänge, wo Informatikkenntnisse von Vorteil sind bzw. sogar verlangt werden, nicht aber in der Tiefe wie bei den Informatikern, Ingenieuren, Mathematikern und Physikern. Man versucht in Kiel also, bei der Vermittlung von Informatikwissen den Bedürfnissen sowohl der Studierenden als auch des Studienfachs möglichst gerecht zu werden.

An der Universität Kiel erfolgt die Ausbildung von Informatikern und Ingenieuren in den Kernfächern des Grundstudiums stark verzahnt. Damit wird unter anderem versucht, eine Interdisziplinarität und Flexibilität im technischen Anwendungsbereich zu erreichen, wie sie der Arbeitsmarkt für hochqualifizierte Kräfte heutzutage und in noch viel stärkerem Maße in der Zukunft verlangt. Andererseits sollen durch die starke Verzahnung von Informatikern und Ingenieuren aber auch Synergieeffekte genutzt werden.

Zum Schluß dieser Einleitung geben wir noch einige Lehrbücher an, die sich als begleitende oder vertiefende Literatur zur Vorlesung eignen. Zuerst seien genannt:

- Bauer F.L., Goos G.: Informatik – Eine einführende Übersicht. 2 Teile, Springer Verlag (1985).

Diese Bücher sind Klassiker und decken viele der oben genannten Punkte ab. Leider sind sie nicht mehr auf dem neuesten Stand. Ein sehr schönes Übersichtsbuch, insbesondere geeignet für einen Leserkreis bestehend aus Anfängern des Informatikstudiums, ist nachfolgend aufgeführt:

- Pepper P.: Grundlagen der Informatik. Oldenburg Verlag (1995).

Funktionale Programmierung erlernen wir anhand der Programmiersprache ML. Derzeit existieren zu dieser Programmiersprache noch keine deutschsprachigen Lehrbücher, obwohl sie, insbesondere im akademischen Bereich, schon sehr verbreitet ist. Als zwei sehr gute englische Lehrbücher zu dieser Sprache können jedoch die nachfolgend genannten empfohlen werden.

- Myers C., Clack C., Poon E.: Programming with Standard ML. Prentice Hall (1993).
- Ullmann J.D.: Elements of ML programming. Prentice Hall (1994).

Modula-2, die modernere Nachfolgersprache des bekannten Pascal, ist unsere Programmiersprache für imperative Programmierung. Sie wird beispielsweise sehr gut beschrieben in den folgenden Lehrbüchern, von welchen man das erste auch als allgemeine Einführung in die Informatik empfehlen kann:

- Appelrath H.-J., Ludewig J.: Skriptum Informatik – Eine konventionelle Einführung. Teubner Verlag (1992).
- Puchan J., Stucky W., Wolff von Gudenberg J.: Programmieren mit Modula-2. Teubner Verlag (1994).

Für den dritten Teil der Vorlesung, den Entwurf von komplexen Algorithmen und Datenstrukturen und die Fragen zur Effizienz, kann man die drei folgenden Lehrbücher sehr empfehlen:

- Horowitz E., Sahni S.: Fundamentals of computer algorithms. Computer Science Press (1981).
- Ottmann T., Widmayer P.: Algorithmen und Datenstrukturen. BI-Wissenschaftsverlag (1990).
- Ottmann T. (Hrsg.): Prinzipien des Algorithmenentwurfs. Spektrum Akademischer Verlag (1998).

Von dem Buch von E. Horowitz und S. Sahni existiert auch eine deutsche Übersetzung, die unter dem Namen „Algorithmen“ im Jahr 1981 beim Springer Verlag erschienen ist. Das von T. Ottmann herausgegebene Buch besteht aus einer Serie von sehr lesenswerten Einzelartikeln zu verschiedenen Entwurfsmethoden. Viele der in der Vorlesung behandelten Tatsachen über Sortieralgorithmen und Suchbäume findet man auch in dem oben genannten Buch von H.-J. Appelrath und J. Ludewig. Diese eben aufgeführte kurze Liste von Lehrbüchern ist natürlich keineswegs vollständig. Falls notwendig, wird auch noch zu speziellen Themen weitere Literatur im Laufe des Skriptums angegeben.

2 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.

2.1 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 mit“):

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

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 „$\ldots$“ 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):

$\forall$	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 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.

2.1.1 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 natürliche Zahlen (wobei das Symbol „:=“ heißt „ist als Name definiert mit“):

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

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.

2.1.2 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 **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:

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

Die Konstruktion $\langle x, y \rangle$ heißt ein **Paar**. Ist $|M| \in \mathbb{N}$, so nennt man M **endlich**. Für endliche Mengen gilt $\left|2^M\right| = 2^{|M|}$, deshalb 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.

2.1.3 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:

$$\begin{tikzcd}
& \{\mathit{tt},\ \mathit{ff}\} & \\
\{\mathit{tt}\} \arrow[ur] & & \{\mathit{ff}\} \arrow[ul] \\
& \emptyset \arrow[ul] \arrow[ur] \arrow[uu] &
\end{tikzcd}$$

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.

2.1.4 Definition (Relation)

Eine Teilmenge R von $M \times N$ heißt (binäre) **Relation** auf M und N mit **Argumentbereich** M und **Resultatbereich** 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.

2.1.5 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 Abschnitt 2.1.2.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** $RS \in 2^{M \times P}$ 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** $I \in 2^{M \times M}$ ist durch

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

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

Man beachte, daß nicht beliebige Relationen miteinander verknüpft werden dürfen. Bei der Vereiningung, 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:

$$(QR)S = Q(RS) \qquad\qquad IR = R = RI$$

Man notiere, daß in den rechten Gleichungen in IR die identische Relation auf dem Argumentbereich von R und in RI 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.

2.1.6 Spezielle homogene Relationen

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

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

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.

2.1.7 Funktionen

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

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

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

[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 Notation „$f(x) = \text{undef}$" um anzuzeigen, daß kein Element zu x in Relation steht.

2.1.8 Beispiele (für Funktionen)

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

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

Diese Relation ist eine Funktion und sieht dann 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:

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

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 $(g \circ f)(x) := g(f(x))$.

2.1.9 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 Abschnitt 2.1.2.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:

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

Für $n = 1$ ist dieses direkte Produkt $\prod_{i=1}^{1} M_i$ isomorph zur Menge M_1 vermöge der bijektiven Funktion

$$f : \prod_{i=1}^{1} M_i \longrightarrow M_1 \qquad\qquad f(\langle m_1 \rangle) = m_1$$

und wird in der Regel mit dieser Menge identifiziert; für $n = 0$ besteht das direkte Produkt nur aus dem leeren Tupel $\langle\rangle$.

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:

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

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.

Wir wollen diesen Abschnitt mit einigen Betrachtungen zur Fundierung von direkten Produkten und n-Tupeln beenden. Bei der Einführung dieser Begriffe haben wir uns an die Notationen gehalten, die (vielleicht mit runden statt spitzen Klammern bei den Tupeln) schon von der Schule her bekannt sind. In der Mathematik liefert jedoch die Mengenlehre den begrifflichen Rahmen und die Sprache, in der sie betrieben wird. Somit haben wir direkte Produkte und Tupel noch mengentheoretisch zu erklären. Dies ist möglich, indem man $\prod_{i=1}^{n} M_i$ festlegt als die Menge der Funktionen $f : \{\ 1, \ldots, n\ \} \longrightarrow \bigcup_{i=1}^{n} M_i$ mit $f(i) \in M_i$ für alle $i, 1 \leq i \leq n$, und

$$\langle f_1, f_2, \ldots, f_{n-1}, f_n \rangle \tag{1}$$

als eine besondere Schreibweise für die Funktion $f : \{\ 1, \ldots, n\ \} \longrightarrow \bigcup_{i=1}^{n} M_i$ mit $f(i) = f_i$ für alle $i, 1 \leq i \leq n$, auffaßt. Unter Verwendung der mengentheoretischen Definition einer Funktion als Relation führt dies dazu, daß (1) nur eine andere Schreibweise für die Menge

$$\{\ \langle 1, f_1 \rangle, \langle 2, f_2 \rangle, \ldots, \langle n-1, f_{n-1} \rangle, \langle n, f_n \rangle\ \} \tag{2}$$

von Paaaren ist. Nun gilt es noch die Schreibweise $\langle x, y \rangle$ von Paaren mengentheoretisch zu beschreiben. Hierzu verwendet man in der Mengenlehre $\{\, x,\ \{\, x, y\,\}\,\}$ und dies führt schließlich dazu, daß (1) nur eine andere Schreibweise für

$$\{\ \{\, 1,\ \{\, 1, f_1\,\}\,\}, \{\, 2,\ \{\, 2, f_2\,\}\,\}, \ldots, \{\, n-1,\ \{\, n-1, f_{n-1}\,\}\,\}, \{\, n,\ \{\, n, f_n\,\}\,\}\ \} \tag{3}$$

ist. Diese eben durchgeführte Reduktion macht auch klar, daß das spezielle Element $\langle\rangle$ aus dem 0-fachen direkten Produkt $\prod_{i=1}^{0} M_i$ die einzige Funktion von $\emptyset$ nach $\emptyset$, nämlich $\emptyset$, ist.

2.2 Induktion und induktive Definitionen

Induktion (oft auch in der Form von Rekursion, etwa bei rekursiven Funktionen oder Rechenvorschriften) 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:

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

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:

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

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:

2.2.1 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

größtes Element von N	$:\Longleftrightarrow$	$\forall\, y \in N \;:\; y \leq x$
kleinstes Element von N	$:\Longleftrightarrow$	$\forall\, y \in N \;:\; x \leq y$
maximales Element von N	$:\Longleftrightarrow$	$\forall\, y \in N \;:\; x \leq y \to x = y$
minimales Element von N	$:\Longleftrightarrow$	$\forall\, y \in N \;:\; y \leq x \to y = x$

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. ■

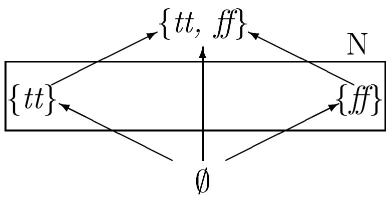
Ketten schreibt man oft auch in der Form $m_0 \leq m_1 \leq m_2 \leq \ldots$ und eine Kette wird stationär, wenn sie von der Form $m_0 \leq \ldots \leq m_j = m_{j+1} = \ldots$ ist. Die in Punkt b) definierten speziellen Elemente müssen nicht immer existieren. Gibt es ein größtes Element, so ist dies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

2.2.2 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.
- 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.

2.2.3 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 stationär wird. Letzteres heißt, daß es keine abzählbar-unendlichen echt absteigenden Ketten $\ldots < m_2 < m_1 < m_0$ 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:

2.2.4 Satz (Noethersche 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 $P(y)$. 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:

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

Damit 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.

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 $\mathit{succ} : \mathbb{N} \to \mathbb{N}$ erzeugt vorstellen, da $n = \mathit{succ}(\mathit{succ}(\ldots(\mathit{succ}(0))\ldots))$ mit n *succ*-Anwendungen gilt. Dies ist ein Spezialfall einer induktiven Definition. Allgemein hat man folgendes Verfahren:

2.2.5 Induktive Definition 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 $S \subseteq U$ heißt von B und den Konstruktoren c **induktiv erzeugt** (auch: **induktiv definiert**), falls sie die kleinste Menge ist mit

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

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:

2.2.6 Satz (Strukturelle 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:

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

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 Abschnitt 2.2.5 (1) sofort

$$B \subseteq M.$$

Nun sei ein Konstruktor $c \in C$ gegeben. Dann zeigen die zweite Prämisse des Induktionsschemas und Abschnitt 2.2.5 (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 Abschnitt 2.2.5.

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 Abschnitt 2.2.5 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 $x \in S$. $\blacksquare$

Bei Induktionsprinzipien und -beweisen haben sich 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 2.2.4, 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.

2.2.7 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 2.2.6 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:

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

   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:

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

   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 in Abschnitt 3.3 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. $x^{m+n} = x^m * x^n$ 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 2.2.6, verifiziert man wie folgt, indem man die Definition von $pow(x, 0)$ verwendet:

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

Beim **Induktionsschluß**, der zweiten Oberformel der Inferenzregel von Satz 2.2.6, haben wir nur einen Konstruktor 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.

c) Die **ganzen Zahlen** $\mathbb{Z}$ sind von $\{0\}$, der Nachfolgerfunktion $succ(x) := x + 1$ und der Vorgängerfunktion $pred(x) := x - 1$ induktiv erzeugt. Satz 2.2.6 liefert das folgende Induktionsprinzip:

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

Eine Eigenschaft auf den ganzen Zahlen gilt also, wenn sie für die Null gilt und wenn ihre Gültigkeit erhalten bleibt, falls man zum Nachfolger und zum Vorgänger übergeht.

Als Beispiele für eine induktive Definition über der Menge $\mathbb{Z}$ betrachten wir die übliche Ordnung $\leq$ und reduzieren sie auf die Funktionen $succ$ und $pred$. Dies ist beispielsweise wie folgt möglich:

$$\begin{array}{ll} (1) & \forall\, x \in \mathbb{Z} : x \leq x \\ (2) & \forall\, x \in \mathbb{Z} : \neg(x \leq x - 1) \\ (3) & \forall\, x, y \in \mathbb{Z} : x \leq y \to x \leq y + 1 \\ (4) & \forall\, x, y \in \mathbb{Z} : x \leq y \to x - 1 \leq y \\ (5) & \forall\, x, y \in \mathbb{Z} : \neg(x \leq y) \to \neg(x \leq y - 1) \\ (6) & \forall\, x, y \in \mathbb{Z} : \neg(x \leq y) \to \neg(x + 1 \leq y) \end{array}$$

Stellt man sich diese Relation als Boolesche Matrix mit Spalten- und Zeilenindizes aus $\mathbb{Z}$ vor, so wird durch (1) die Diagonale mit lauter Einsen besetzt und durch (2) die untere Nebendiagonale mit lauter Nullen. Aufbauend auf diese Belegung bestimmen die restlichen Formeln, daß die Matrix eine obere Dreiecksmatrix ist, was genau der üblichen Ordnung auf $\mathbb{Z}$ entspricht.

Dem aufmerksamen Leser ist sicher aufgefallen, daß in den Formeln (1) bis (6) zur Spezifikation der Ordnung auf $\mathbb{Z}$ die Null gar nicht vorkommt. Der wesentliche Grund hierfür ist, daß man die ganzen Zahlen mittels *succ* und *pred* und jeder beliebigen Zahl $n \in \mathbb{Z}$ induktiv erzeugen kann. Somit darf in der Festlegung der Ordnung die Null nicht vorkommen. ■

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 Abschnitt 2.1.7) 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.

3 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.

3.1 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.

3.1.1 Definition (Zeichenreihe)

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. 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:

3.1.2 Satz (Erzeugungsprinzip)

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

3.1.3 Satz (Induktion auf Zeichenreihen)

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

$$\frac{\begin{array}{c} P(\langle\rangle) \\ \forall\, a \in A, w \in A^* : \quad P(w) \rightarrow P(\mathit{prefix}_a(w)) \end{array}}{\forall\, w \in A^* : P(w)}$$

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

$$\frac{\begin{array}{c} P(\langle\rangle) \\ \forall\, a \in A, w \in A^* : P(w) \rightarrow P(\mathit{postfix}_a(w)) \end{array}}{\forall\, w \in A^* : P(w)}$$

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 3.1.3 wird im folgenden Beispiel gegeben. Dieses führt auch eine wichtige weitere Funktion auf A^* ein, die Konkatenation (Hintereinanderschreibung) von Zeichenreihen.

3.1.4 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 2.2.7,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:

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

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 $a_1 \dots a_n$ ohne Klammern und Kommata statt $\langle a_1, \dots, a_n \rangle$. Manchmal schreibt man auch "$a_1 \dots a_n$".
- Man schreibt ε statt $\langle\rangle$. Die Zeichenreihe ε heißt die **leere Zeichenreihe** oder das **leere Wort**. Für $A \setminus \{\varepsilon\}$ schreibt man A^+.
- Bei der Konkatenation sind die Schreibweisen $w_1 w_2$ oder auch $w_1 \cdot w_2$ statt der funktionalen Notation $\mathit{conc}(w_1, w_2)$ üblich.
- Bei den Anfügeoperationen schreibt man $a \,\&\, w$ statt $\mathit{prefix}_a(w)$ und $w \,\&\, a$ statt $\mathit{postfix}_a(w)$. 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 3.1.4 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.

3.1.5 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 $\mathit{top} : A^* \longrightarrow A$ und Entfernen des ersten Elements mittels $\mathit{rest} : A^* \longrightarrow A^*$:

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

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

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

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 die Längenoperation und

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

für 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.

3.1.6 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:

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

Gilt $w_1 = w_2 \vee w_1 <_{lex} w_2$, 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 3.1.6 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 3.1.6 mit der Zeichenreihe r = stein zu.

Wir werden nicht beweisen, daß die in Definition 3.1.6 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.

3.1.7 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.

3.1.8 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:

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

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

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

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 Abschnitt 3.1.8.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.

3.1.9 Beispiel (für Textersetzung)

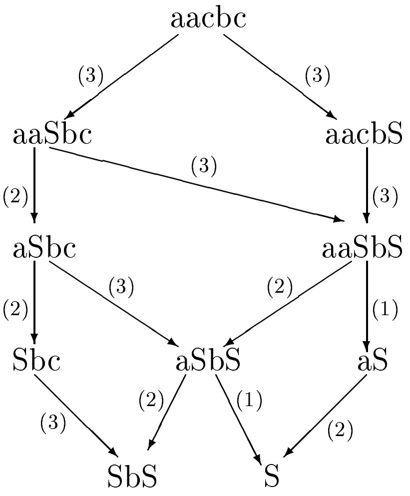
Wir betrachten ein Textersetzungssystem mit dem Zeichenvorrat

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

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:

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

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

$$v \stackrel{*}{\Rightarrow} w_1 \wedge v \stackrel{*}{\Rightarrow} w_2 \wedge w_1, w_2 \text{ sind Normalformen} \quad\Longrightarrow\quad w_1 = w_2?$$

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

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

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 $u \stackrel{*}{\Rightarrow} v$ abbildet. Eindeutigkeit und Existenz einer erreichbaren Normalform müssen aber nicht immer gegeben ein. Das System aus Beispiel 3.1.8 verletzt etwa Punkt 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, da für das Wort ca beispielsweise nur die unendliche Sequenz

$$\mathrm{ca} \stackrel{(3)}{\Rightarrow} \mathrm{Sa} \stackrel{(4)}{\Rightarrow} \mathrm{ca} \stackrel{(3)}{\Rightarrow} \mathrm{Sa} \stackrel{(4)}{\Rightarrow} \ldots$$

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

Im folgenden spezialisieren wir nun Textersetzung zur Beschreibung der **Syntax von Programmiersprachen**. Dies wird auf den Begriff der Grammatik führen, der sehr zentral ist in der Informatik, beispielsweise beim Übersetzerbau oder in der theoretischen Informatik.

3.1.10 Beispiel (zur Motivation)

In vielen Programmiersprachen kann man gewissen Dingen einen Namen geben, z.B. Funktionen oder Mengen von Daten. Solche Namen heißen **Bezeichner** oder **Identifier** und sind nach festen Regeln aufgebaute Zeichenreihen. Normalerweise hat man:

(1) Ein Bezeichner ist eine nichtleere Zeichenreihe, die nur Buchstaben, Ziffern und Sonderzeichen enthält und außerdem mit einem Buchstaben anfängt.

Nun muß man noch Buchstaben, Ziffern und Sonderzeichen erklären. Im Falle genau eines Sonderzeichens „_“ führt dies zu:

(2) Ein Buchstabe ist A oder B oder C oder ... oder Z (d.h. ein Großbuchstabe) oder a oder b oder c oder ... oder z (d.h. ein Kleinbuchstabe).

(3) Eine Ziffer ist 0 oder 1 oder 2 oder ... oder 9.

(4) Ein Sonderzeichen ist _ .

Es bietet sich nun an, die Menge der Bezeichner induktiv zu erzeugen. Dies ist beispielsweise wie folgt möglich:

a) Man definiert zuerst die drei Mengen der Buchstaben, Ziffern und des Sonderzeichens:

$$\begin{aligned} Bu &:= \{\ \mathrm{A, B, C}, \ldots, \mathrm{Z, a, b, c}, \ldots, \mathrm{z}\ \} \\ Zi &:= \{\ 0, 1, 2, \ldots, 9\ \} \\ So &:= \{\ \_\ \} \end{aligned}$$

b) Dann definiert man induktiv eine Menge Fo (mnemotechnisch für „Folgen“) mittels der Basis $\{\varepsilon\}$ und der Konstruktoren $prefix_a$, mit $a \in Bu \cup Zi \cup So$. In der üblichen mathematischen Schreibweise sieht dies wie folgt aus, wobei man den letzten Punkt oft wegläßt, wenn vorher das Wort „induktiv“ verwendet wurde:

- Es ist $\varepsilon \in Fo$.
- Für alle $w \in Fo$ und alle $a \in Bu \cup Zi \cup So$ gilt $a \,\&\, w \in Fo$.
- Es gibt keine weiteren Elemente in Fo.

c) Schließlich legt man die Menge der Bezeichner wie folgt fest:

$$Bez := \{\ a \,\&\, w\ :\ a \in Bu \wedge w \in Fo\ \}$$

Nun können wir a), b) und c) sofort in ein Textersetzungssystem überführen. Dieses Sytem arbeitet über einem Zeichenvorrat, der aus den Zeichen

$$\mathrm{A}, \mathrm{B}, \mathrm{C}, \ \ldots, \mathrm{Z}, \mathrm{a}, \mathrm{b}, \mathrm{c}, \ \ldots, \mathrm{z}, 0, \ldots, 9, \ \_$$

sowie den „neuen“ Zeichen oder „Hilfszeichen“ (für die eingeführten Mengen)

$$Bu, Zi, So, Fo, Bez$$

besteht. Dies sind auch Zeichen (also unteilbar) und keine Zeichenreihen, obwohl die Hinschreibung zwei bzw. drei Buchstaben bedingt. Zur Unterscheidung werden deshalb die Hilfszeichen in kursiver Schrift gesetzt und die Einzelzeichen in normaler Schrift. Die Regeln des Textersetzungssystems sind nachfolgend gegeben, wobei sich die Gruppen (i), (ii) und (iii) genau aus den obigen Punkten a), b) und c) ergeben.

$$\begin{array}{ll}
\text{Gruppe (i)} & Bu \rightarrow \mathrm{A} \\
 & \quad \ldots \\
 & Bu \rightarrow \mathrm{z} \\
 & Zi \rightarrow 0 \\
 & \quad \ldots \\
 & Zi \rightarrow 9 \\
 & So \rightarrow _ \\
\text{Gruppe (ii)} & Fo \rightarrow \varepsilon \\
 & Fo \rightarrow BuFo \\
 & Fo \rightarrow ZiFo \\
 & Fo \rightarrow SoFo \\
\text{Gruppe (iii)} & Bez \rightarrow BuFo
\end{array}$$

Ob eine Zeichenreihe w aus dem gegebenen Zeichenvorrat ein Bezeichner ist, sieht man nun genau daran, ob $Bez \stackrel{*}{\Rightarrow} w$ zutrifft, oder nicht. Eine kleine Beispielrechnung ist

$$\begin{array}{lll}
Bez & \Rightarrow BuFo & \text{wegen (iii)} \\
 & \Rightarrow \mathrm{A}Fo & \text{wegen (i)} \\
 & \Rightarrow \mathrm{A}ZiFo & \text{wegen (ii)} \\
 & \Rightarrow \mathrm{A1}Fo & \text{wegen (i)} \\
 & \Rightarrow \mathrm{A1} & \text{wegen (ii),}
\end{array}$$

wobei im letzten Schritt die Konkatenation 1ε zu ε vereinfacht wurde. Also ist die Zeichenreihe A1 ein Bezeichner. Hingegen ist etwa die Zeichenreihe 1A kein Bezeichner. Hätte man $Bez \stackrel{*}{\Rightarrow} 1A$, so muß (iii) verwendet werden, d.h. es folgt $Bez \Rightarrow BuFo \stackrel{*}{\Rightarrow} 1A$. Aber $BuFo \stackrel{*}{\Rightarrow} 1A$ ist nicht möglich, wie eine Analyse aller Regelanwendungen zeigt. ■

Eine Abstraktion von diesem Beispiel führt zu den beiden wichtigen Begriffen der formalen Sprachen und Grammatiken.

3.1.11 Definition (Formale Sprache)

Eine **formale Sprache** L ist eine Teilmenge einer Menge A^* von Zeichenreihen. Die grundlegenden Operationen auf formalen Sprachen sind

a) die üblichen **mengentheoretischen** Operationen $L_1 \cup L_2, L_1 \cap L_2, \overline{L} = A^* \setminus L$,

b) die **Produktbildung**, die erklärt ist durch

$$L_1L_2 := \{\ w_1w_2\ :\ w_1 \in L_1 \wedge w_2 \in L_2\ \},$$

c) die **Kleene'sche Sternoperation** oder **reflexiv-transitive Hülle**

$$L^* := \bigcup_{n\geq 0} L^n = \{\varepsilon\} \cup L \cup L^2 \cup L^3 \cup \ldots,$$

wobei die n-ten Potenzen L^n einer formalen Sprache L induktiv definiert sind mittels $L^0 := \{\varepsilon\}$ und $L^{i+1} := LL^i$,

d) die **Plusoperation** oder **transitive Hülle**

$$L^+ := \bigcup_{n\geq 1} L^n = L \cup L^2 \cup L^3 \cup \ldots = LL^*.$$ ■

Offensichtlich gilt auch die Gleichung $L^* = \{\varepsilon\} \cup L^+$ und damit sind, wegen, $L^+ = LL^*$, beide Hüllen wechselseitig aufeinander abstützend definierbar.

Nun zu den Grammatiken. Sie sind im Prinzip Textersetzungssyteme $(A, \mathfrak{R})$, in denen man erstens den Zeichenvorrat A aufteilt in die „eigentlichen“ Zeichen (aus denen die Zeichenreihen aufgebaut sind, die man verarbeiten will) und Hilfszeichen zur Verarbeitung, zweitens ein Spezialzeichen aus der Menge der Hilfszeichen als Start einer jeden Ersetzungsfolge festlegt und schließlich drittens an die Regeln noch geringfügige Einschränkungen stellt.

3.1.12 Definition (Grammatik)

Eine **Grammatik** ist ein Quadrupel $G = (N, T, P, S)$ und besteht aus

a) den **nichtterminalen** Zeichen N (auch **syntaktische Variable** oder Hilfszeichen genannt),

b) den **Terminalzeichen** T mit $N \cap T = \emptyset$,

c) einer Menge P von sogenannten **Produktionsregeln** $l \to r$ mit $l \in (N \cup T)^* \setminus \{\varepsilon\}$ und $r \in (N \cup T)^*$ so, daß in l mindestens ein Zeichen aus N vorkommt,

d) dem **Startsymbol** (auch **Axiom** genannt) $S \in N$.

Aufbauend auf die in Abschnitt 3.1.8.c festgelegte Berechnungsrelation $\stackrel{*}{\Rightarrow}$ definiert man

$$\mathcal{L}(G) := \{\ w \in T^* \ : \ S \stackrel{*}{\Rightarrow} w\ \}$$

als die **durch die Grammatik G beschriebene (formale) Sprache** oder den **Sprachschatz** von G. Eine formale Sprache $L \subseteq T^*$ wird durch G **beschrieben**, falls $L = \mathcal{L}(G)$ gilt. ■

Je nach den weiter gemachten Einschränkungen in den Formen der Produktionsregeln definiert man verschiedene Grammatikklassen. Für die praktische Informatik sind die folgenden zwei Klassen von Bedeutung:

a) **Kontextfreie Grammatiken:** Hier haben die Regeln die Form $A \to r$ mit $A \in N$ und $r \in (N \cup T)^*$.

b) **Reguläre Grammatiken:** Hiervon gibt es zwei Arten. Haben die Regeln die Form

$$A \to w \qquad\qquad A \to wB$$

mit $A, B \in N$ und $w \in T^*$, so spricht man von einer **rechtsregulären** Grammatik. Hat man hingegen Regeln die Form

$$A \to w \qquad\qquad A \to Bw$$

mit $A, B \in N$ und $w \in T^*$, so wird die Grammatik **linksregulären** genannt.

In der Literatur verwendet man auch noch Abkürzungen, wie etwa $l \to r_1 \mid \ldots \mid r_n$ für die Menge $\{\ l \to r_1, \ldots, l \to r_n\ \}$ von Produktionsregeln. Auch wir machen im folgenden Gebrauch von diesen Abkürzungen.

3.1.13 Beispiele (für Grammatiken)

a) Es seien die Mengen T und BEZ wie folgt festgelegt:

$$\begin{array}{rcl} T & := & \{\ \mathrm{A}, \ldots, \mathrm{Z}, \mathrm{a},\ \ldots, \mathrm{z}, 0, \ldots, 9, \_\} \\ BEZ & := & \{\ w \in T^* \ : \ w \neq \varepsilon \wedge top(w) \in \{\mathrm{A},\ldots,\mathrm{z}\}\ \} \end{array}$$

Damit besteht die Menge BEZ genau aus den schon früher betrachteten Bezeichnern. Es ist eine einfache Übung zu verifizieren, daß die Gleichheit $BEZ = \mathcal{L}(G)$ gilt, falls die Grammatik $G = (N, T, P, S)$ wie folgt festgelegt ist:

Die Nichtterminalzeichen N sind definiert durch:

$$N := \{ \ Bu, Zi, So, Fo, Bez \ \}$$

Terminalzeichen sind genau die Zeichen aus T. Die Menge P der Produktionsregeln besteht aus:

$$\begin{array}{lcl} Bu & \rightarrow & \mathrm{A} \mid \ldots \mid \mathrm{Z} \mid \mathrm{a} \mid \ldots \mid \mathrm{z} \\ Zi & \rightarrow & 0 \mid \ldots \mid 9 \\ So & \rightarrow & \_ \\ Fo & \rightarrow & \varepsilon \mid BuFo \mid ZiFo \mid SoFo \\ Bez & \rightarrow & BuFo \end{array}$$

Das Startsymbol S ist das Hilfszeichen Bez.

Offensichtlich ist die Grammatik G auch **kontextfrei**. Wegen der vierten Produktionsregel ist sie jedoch **nicht regulär**.

b) Nun betrachten wir Dezimalzahlen, also nichtleere und endliche Folgen von Ziffern, wobei auch führende Nullen zugelassen sind. Wir wollen diese Menge durch eine Grammatik $G = (N, T, P, S)$ beschreiben.

Dazu beginnen wir mit den folgenden zwei Mengen, wobei die zweite Menge DZ wie in a) wiederum genau die ist, die uns interessiert:

$$\begin{array}{rcl} T & := & \{ \ 0, \ldots, 9 \ \} \\ DZ & := & \{ \ w \in T^* \ : \ w \neq \varepsilon \ \} = \{0, \ldots, 9\}^+ \end{array}$$

Eine einfache Grammatik G zur Beschreibung der Dezimalzahlen $d \in DZ$ mit eventuell führenden Nullen sieht beispielsweise wie folgt aus:

Die Nichtterminalzeichen N beinhalten genau die Zeichen Dz und Zi. Terminalzeichen sind wiederum die Zeichen aus T. Als Produktionsregeln in P haben wir:

$$\begin{array}{lcl} Zi & \rightarrow & 0 \mid \ldots \mid 9 \\ Dz & \rightarrow & Zi \mid ZiDz \end{array}$$

Das Startsymbol S ist Dz.

Es ist eine einfache Übungsaufgabe, zu verifizieren, daß tatsächlich die Gleichung $DZ = \mathcal{L}(G)$ gilt. ■

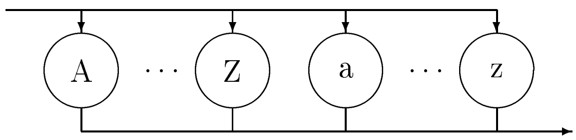
Wir hoffen, daß durch diese zwei Beispiele die enge Verbindung von Textersetzungssystemen und Grammatiken besonders klar wird, insbesondere, wenn man den Punkt a) nochmals mit Beispiel 3.1.10 vergleicht.

Früher wurden Programmiersprachen syntaktisch nur durch Grammatiken beschrieben. Dann entdeckte man, daß die Produktionen einer Grammatik G graphisch sehr einfach darstellbar sind. Das führte zu den **Syntaxdiagrammen**. Nichtterminalzeichen werden durch Rechtecke dargestellt, Terminalzeichen durch Kreise oder Ovale. Man durchläuft zu einer gegebenen Zeichenreihe das Diagramm ab dem Einstieg längs der Pfeile. Erreicht man mittels eines Pfeilweges den Ausstieg, so ist die Zeichenreihe im Sprachschatz $\mathcal{L}(G)$. Anhand eines Beispieles wird dies am schnellsten klar.

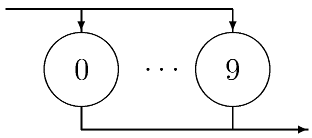
3.1.14 Beispiele (für Syntaxdiagramme)

a) Zu der Grammatik aus Beispiel 3.1.13.a zur Beschreibung der Bezeichner bekommen wir die folgenden Syntaxdiagramme:

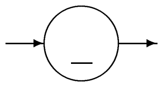
Bu:



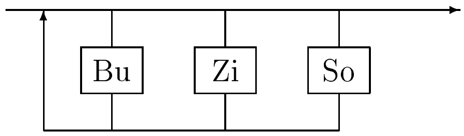
Zi:



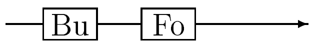
So:



Fo:

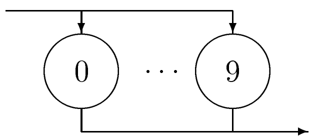


Bez:

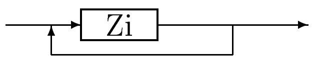


b) Im Falle der Dezimalzahlen mit zugelassenen führenden Nullen, also für die Grammatik von Beispiel 3.1.13.b, bekommen wir die nachfolgenden Syntaxdiagramme:

Zi:



Dz:



Diese Syntaxdiagramme sind natürlich sehr einfach. Wesentlich kompliziertere Syntaxdiagramme findet man beispielsweise in Anhang B des in der Einleitung erwähnten Buchs von Appelrath und Ludewig. Diese legen die Syntax der Programmiersprache Modula-2 fest. ■

3.2 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.

3.2.1 Semantik und Interpretation

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

$$\boxed{\text{Darstellung}} \xrightarrow[\text{Semantik}]{I} \boxed{\text{Information}}$$

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.

3.2.2 Beispiel (Binärdarstellung, Dualsystem)

Die Syntax der Binärzahlen (oder Zahlen des Dualsystems) ist gegeben durch die Grammatik $G = (N, T, P, S)$ mit

$$\begin{array}{lcl} N & = & \{ \ Bz \ \} \\ T & = & \{ \ \mathrm{O}, \ \mathrm{L} \ \} \\ P & = & \{ \ Bz \to \mathrm{O} \mid \mathrm{L} \mid Bz\mathrm{O} \mid Bz\mathrm{L} \ \} \\ S & = & Bz. \end{array}$$

Durch die Produktionsregeln von P sind offensichtlich auch führende Nullen wie in der Zeichenreihe OOOLL erlaubt.

Nun zur Semantik der Binärzahlen. Sie ist gegeben durch eine Funktion

$$[\![.]\!] : \mathcal{L}(G) \longrightarrow \mathbb{N},$$

deren nachfolgende induktive Definition genau dem syntaktischen Aufbau der Binärworte $\varepsilon \neq w \in \mathcal{L}(G)$ durch das Anfügen von rechts folgt.

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

Auf der rechten Seite dieser Definition verwenden wir die für Zahlen übliche Dezimaldarstellung als Bezugssystem. Die Festlegung der Semantik durch die Gleichungen (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

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

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

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

3.2.3 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 Ziffern 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 Abschnitt 3.1.5 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 Modula-2 etwas auf ihre semantischen Festlegungen eingehen, jedoch nicht in großer mathematischer Gründlichkeit.

3.3 Signaturen und Terme

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

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

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 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.

3.3.1 Definition (Signatur)

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 **Typ** oder **Art**).

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

c) Jedem $f \in F$ ist genau ein Element $w \in S^* \setminus \{\varepsilon\}$ und genau ein Element $s \in S$ zugeordnet. Das Element f heißt dann **Funktionssymbol** 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 den von uns später verwendeten Programmiersprachen ML und Modula-2 üblich.

3.3.2 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:

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

b) Nun betrachten wir Zeichenreihen aus A^* mit den in Abschnitt 3.1 eingeführten Operationen. Setzen wir für den Zeichenvorrat A genau die Zeichen c_1 bis c_n voraus, so führt dies zu der nachfolgenden Signatur $\Sigma_{A^*} = (S, K, F)$:

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

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

$$zero \qquad succ(zero) \qquad succ(succ(zero))$$

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

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

über der Signatur Σ_{A^*}. Dabei beschreiben die ersten drei Terme Zahlen, der vierte ein Zeichen und der fünfte 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, wie schon bei den Bezeichnern (induktive Definition der Menge *Fo* in Beispiel 3.1.10.b), die leichter verstehbare Schreibweise und lassen auch die letzte Bedingung „es gibt keine weiteren Terme" weg.

3.3.3 Definition (Term)

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 $\mathcal{T}^s_\Sigma(X)$ 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 $c \in \mathcal{T}^s_\Sigma(X)$.

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

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

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

Beim Aufbau von Termen nach dieser Definition müssen in Punkt c), der Anwendung eines Funktionssymbols, also sowohl Stelligkeit (Anzahl der Argumente) als auch Typisierung (Sorten der Argumente) stimmen.

3.3.4 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** etc. 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.

3.3.5 Beispiele (für Termdarstellung)

Wir betrachten, in der gewohnten mathematischen Notation, den Term

$$2^5 - 1.$$

In der imperativen Programmiersprache Algol 60, einem Vorläufer von Modula-2 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

[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.

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 die Darstellung

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

In den Sprachen ML und Modula-2, 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, 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.

3.3.6 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 Wertes 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.

3.3.7 Definition (Σ-Algebra)

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 $s_1, \ldots, s_k \to s$ 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 einer Variablen einen Wert zuordnen.

3.3.8 Definition (Belegung)

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 $\sigma : X \longrightarrow \bigcup_{s \in S} s^A$ mit der Eigenschaft, daß $\sigma(x) \in s^A$ genau dann, wenn s die Sorte von x ist, für alle $x \in X$ gilt. ■

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

3.3.9 Definition (Wert)

Es seien die Signatur Σ, die Σ-Algebra A und die Variablenmenge X wie in Definition 3.3.8 vorausgesetzt. Der **Wert** $I[t]_\sigma^A$ eines Terms t zur Belegung σ in A ist induktiv wie folgt definiert:

a) Für alle Konstantensymbole $c \in K$ gilt
$$I[c]_\sigma^A = c^A.$$

b) Für alle Variablen $x \in X$ gilt
$$I[x]_\sigma^A = \sigma(x).$$

c) Für alle zusammengesetzten Terme der Form $f(t_1, \ldots, t_k)$ gilt
$$I[f(t_1, \ldots, t_k)]_\sigma^A = f^A(I[t_1]_\sigma^A, \ldots, I[t_k]_\sigma^A).$$ ■

Ist t ein Term der Sorte s, so ist der Wert von t ein Element aus s^A oder undefiniert. In der Definition 3.3.9 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.

3.3.10 Satz (Koinzidenzlemma)

Gilt die Gleichung $\sigma_1(x) = \sigma_2(x)$ für alle Variablen x, die in dem Term $t \in \mathcal{T}_\Sigma(X)$ vorkommen, so hat man

$$I[t]^A_{\sigma_1} = I[t]^A_{\sigma_2}.$$

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

$$I[c]^A_{\sigma_1} = c^A = I[c]^A_{\sigma_2}.$$

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 $\sigma_1(x) = \sigma_2(x)$, das gewünschte Resultat

$$I[x]^A_{\sigma_1} = \sigma_1(x) = \sigma_2(x) = I[x]^A_{\sigma_2}.$$

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

$$\begin{array}{rcll} I[f(t_1, \ldots, t_k)]^A_{\sigma_1} & = & f^A(I[t_1]^A_{\sigma_1}, \ldots, I[t_k]^A_{\sigma_1}) & \text{Definition des Werts} \\ & = & f^A(I[t_1]^A_{\sigma_2}, \ldots, I[t_k]^A_{\sigma_2}) & \text{wegen der Induktionshypothese} \\ & = & I[f(t_1, \ldots, t_k)]^A_{\sigma_2} & \text{Definition des Werts,} \end{array}$$

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 $I[t]^A_\sigma$ unabhängig von σ. 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. Zuerst zur formalen Definition solcher Systeme:

3.3.11 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 Abschnitt 3.1.8 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.

3.3.12 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.

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

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.

3.3.13 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

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

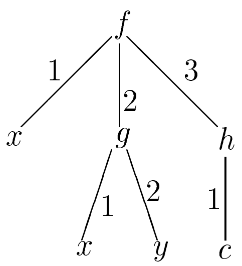
so bekommen wir als Resultat der Anwendung

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

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

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

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

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

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.

3.3.14 Termersetzung

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

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

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

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

Die Konstruktion in Punkt c) dieser Festlegung ist vollkommen analog zur Konstruktion in Abschnitt 3.1.8.c bei der Textersetzung. Dem liegt ein allgemeines Prinzip auf Relationen zugrunde.

3.3.15 Definition (Hüllen)

Ist R eine homogene Relation, so definiert man induktiv $R^0 := I$ 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. ■

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.

3.3.16 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:

$$\begin{array}{ll} (1) & fac(0) \to 1 \\ (2) & fac(succ(n)) \to succ(n) * fac(n) \\ (3) & \text{„Vereinfachungsregeln" von } \mathbb{N} \end{array}$$

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

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

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:

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

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:

$$\begin{tikzcd} & f(f(a,a),f(a,a)) \arrow[dl] \arrow[dd] \arrow[dr] & \\ f(a,f(a,a)) \arrow[dr] & & f(f(a,a),a) \arrow[dl] \\ & f(a,a) \arrow[d] & \\ & a & \end{tikzcd}$$

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

c) Im folgenden sei die Signatur die von Beispiel 3.3.2.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:

$$
\begin{tikzcd}
 & add(0, succ(0)) \arrow[dl, shift left, "(3)"] \arrow[dr, "(2)"] & \\
add(succ(0), 0) \arrow[ur, shift left, "(3)"] \arrow[dr, "(1)"'] & & succ(add(0, 0)) \arrow[dl, "(1)"] \arrow[loop right, "(3)"] \\
 & succ(0) &
\end{tikzcd}
$$

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 $l_i\sigma \stackrel{*}{\Leftrightarrow} r_i\sigma$ 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.

3.4 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), die alle nach einem festen „Strickmuster“ ablaufen. Nachfolgend geben wir einige grundsätzliche Gedanken zum Algorithmusbegriff an. Wir beginnen mit einer informellen Beschreibung, geben später aber auch eine auf Textersetzung aufbauende mathematische Präzisierung an. Auch die Grenzen der Algorithmisierbarkeit werden anhand von Beispielen aufgezeigt.

3.4.1 Informelle Beschreibung

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.

3.4.2 Darstellung von 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 Telephonieren, die man in jedem Telephonhäuschen 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. Beispiele für solche nicht berechenbare Relationen / Funktionen werden wir später noch angeben.

3.4.3 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 Modula-2 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 Abschnitt 3.4.3.f sind Algorithmen zur Lösung von linearen Gleichungssystemen. 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. Im folgenden beschreiben wir den vierten Ansatz der Markov-Algorithmen genauer, da diese eine spezielle Variante von Textersetzung darstellen („gesteuerte Textersetzung") und wir jene schon kennen.

3.4.4 Markov-Algorithmen

Ein **Markov-Algorithmus** über einem Zeichenvorrat A ist ein Paar $(A, \mathfrak{R})$ mit einer endlichen Folge $\mathfrak{R} = \langle R_1, \ldots, R_k \rangle$ von Regeln. Jede Regel hat eine der beiden folgenden Gestalten

$$(1)\quad l \to r \qquad\qquad (2)\quad l \to_{\bullet} r,$$

wobei $l, r \in A^*$ Zeichenreihen sind. Regeln der Form (2) heißen **abbrechend**. ■

Gewöhnliche Textersetzung kann eine Zeichenreihe in mehrere andere transformieren – je nach Regelauswahl und Anwendungsstelle. Bei Markov-Algorithmen wendet man jeweils die Regel mit dem kleinsten Index so weit wie möglich links in der vorgegebenen Zeichenreihe an und erreicht damit die Eindeutigkeit des Resultats. Weiterhin verschärft man die Terminierungsbedingungen durch Verwendung der abbrechenden Regeln. Zur formalen Festlegung der zwei Einschritt-Ersetzungsrelationen $\Rightarrow$ und $\Rightarrow_\bullet$ analog zur Textersetzung 3.1.8 benötigen wir den Begriff der **Teilzeichenreihe**, den wir auch schon bei der Textersetzung implizit verwendet haben: v ist eine Teilzeichenreihe von w, falls es $l, r \in A^*$ mit $lvr = w$ gibt.

3.4.5 Markov-Ersetzung

Zu einem Markov-Algorithmus $(A, \mathfrak{R})$ mit Regeln $\mathfrak{R} = \langle R_1, \ldots, R_k \rangle$ mit jeweils linken Seiten l_i und rechten Seiten r_i $(1 \leq i \leq k)$ und $u, v \in A^*$ gilt die **(nichtterminierende) Einschritt-Ersetzung** $u \Rightarrow v$, falls es $i, 1 \leq i \leq k$, und $a, e \in A^*$ gibt, so daß die folgenden Eigenschaften gelten:

a) Es ist $u = al_ie$ und $v = ar_ie$ („Regel R_i überführt die Zeichenreihe u in die Zeichenreihe v") und keine linke Seite $l_j, 1 \leq j \leq i-1$, ist Teilzeichenreihe von u („Regel R_i hat unter den anwendbaren Regeln den kleinsten Index").

b) Die linke Seite l_i kommt in der Zeichenreihe al_i genau einmal als Teilzeichenreihe vor („Regel R_i wird soweit wie möglich links angewendet").

c) R_i ist nicht abbrechend und mindestens eine linke Seite $l_j, 1 \leq j \leq k$, ist Teilzeichenreihe von v („man kann nach Anwendung von Regel R_i weiterersetzen").

Die **terminierende Einschritt-Ersetzung** $u \Rightarrow_\bullet v$ gilt für $u, v \in A^*$, falls es $i, 1 \leq i \leq k$, und $a, e \in A^*$ gibt, so daß so daß a) und b) zutreffen, c) aber nicht gilt.

Die **Berechnungsrelation** $\overset{\mathrm{M}}{\Rightarrow}_\bullet$ ist für $u, v \in A^*$ definiert als

$$
\begin{aligned}
&u \overset{\mathrm{M}}{\Rightarrow}_\bullet v \quad :\Longleftrightarrow \\
&\quad (\exists\, w \in A^* : u \overset{*}{\Rightarrow} w \wedge w \Rightarrow_\bullet v) \vee (u = v \wedge \text{ kein } l_i \text{ ist Teilzeichenreihe von } u),
\end{aligned}
$$

wobei die Relation $\overset{*}{\Rightarrow}$ die reflexiv-transitive Hülle der Einschritt-Ersetzung $\Rightarrow$ ist. ■

Jeder Markov-Algorithmus hat eine eindeutige Berechnungsrelation $\overset{\mathrm{M}}{\Rightarrow}_\bullet \subseteq A^* \times A^*$ und definiert damit eine partielle Funktion durch die Festlegung

$$
f : A^* \longrightarrow A^* \qquad\qquad f(u) = \begin{cases} v & : \ u \overset{\mathrm{M}}{\Rightarrow}_\bullet v \\ \text{undef} & : \ \text{sonst.} \end{cases}
$$

Nachfolgend geben wir einige Beispiele für Markov-Algorithmen an. Dabei stützen wir uns auf die Strichzahl-Darstellung von natürlichen Zahlen, bei der die Zahl n durch eine Zeichenreihe $|^n$ mit n Strichen notiert wird. Als Darstellung der Null dient somit die leere Zeichenreihe.

3.4.6 Beispiele (Markov-Algorithmen über Strichzahlen)

a) Eine **Realisierung der Addition** mittels eines Markov-Algorithmus entspricht der Überführung der Zeichenreihe $|^n + |^m$ in die Zeichenreihe $|^{n+m}$. Ein entsprechender Markov-Algorithmus ist:

$$\begin{array}{rcl} A & : & |, + \\ \mathfrak{R} & : & (1) + \to_{\bullet} \varepsilon \end{array}$$

Es wird also aus der Eingabe das trennede Zeichen + entfernt, dann stoppt das Verfahren sofort.

b) Der nachfolgende Markov-Algorithmus zum **Test auf Geradesein** formt die Strichzahl $|^n$ in O um, falls n ungerade ist, und in L, falls n gerade ist.

$$\begin{array}{rcll} A & : & |, \mathrm{O}, \mathrm{L} & \\ \mathfrak{R} & : & (1) & || \to \varepsilon \\ & & (2) & | \to_{\bullet} \mathrm{O} \\ & & (3) & \varepsilon \to_{\bullet} \mathrm{L} \end{array}$$

Im Gegensatz zum ersten Verfahren ist hier ein iteratives Vorgehen nötig. Bei der Abarbeitung werden durch die Regel (1) solange jeweils zwei Striche entfernt, bis nur mehr ein Strich oder kein Strich mehr übrig ist. Im ersten Fall ist dann noch Regel (2) anwendbar und liefert „ungerade"; im zweiten Fall folgt das Resultat „gerade" mittels Regel (3).

c) Ein Markov-Algorithmus zur **Überführung von der Strich- in die Binärdarstellung** sieht schließlich wie folgt aus:

$$\begin{array}{rcll} A & : & |, \mathrm{O}, \mathrm{L}, s \text{ (Hilfszeichen, „Schiffchen")} & \\ \mathfrak{R} & : & (1) & s|| \to |s \\ & & (2) & s| \to \mathrm{L} \\ & & (3) & s \to \mathrm{O} \\ & & (4) & | \to s| \end{array}$$

Auch hier wird iterativ vorgegangen, im Gegensatz zum Verfahren aus b) jedoch „geschachtelt", wie man sich leicht anhand eines kleinen Beispiels klarmacht. Das Verfahren ist nur für positive Zahlen korrekt und arbeitet analog zur Semantik der Binärzahlen von Beispiel 3.2.2. Die Strichzahl der Null kann mit ihm nicht umgeformt werden; dies erfordert eine Sonderbehandlung. ■

Es ist eine einfache Übung und auch sehr zu empfehlen, den letzten Markov-Algoritmus so zu erweitern, daß beliebige Dezimalzahlen in Binärzahlen umgewandelt werden. Im Gegensatz zur Umwandlung von Strichzahlen in Binärzahlen ist dabei keine Sonderbehandlung der Null erforderlich.

Eine Fülle von Markov-Algorithmen findet man etwa in der Doktorarbeit „Konstruktion effizienter Markov-Algorithmen durch Programm-Transformationen" von H. Partsch, die auch als Bericht 7915 des Instituts für Informatik der Technischen Universität München im Jahr 1979 publiziert wurde.

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 zwei berühmte algorithmisch nicht lösbare Probleme in der Form von Entscheidungsproblemen mit „ja“ und „nein“ als möglichen Antworten an. Da man jedem solchen Problem eine Funktion in die Wahrheitswerte $\mathbb{B}$ zuordnen kann, liefern die beiden Probleme auch Beispiele für nicht berechenbare Funktionen.

3.4.7 Beispiele (für unlösbare Probleme)

a) 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 $p(b_1, \ldots, b_n) = 0$?

b) Das folgende Problem, genannt das **Halteproblem** (für Markov-Algorithmen), ist unlösbar.

Eingabe: Ein Markov-Algorithmus $(A, \mathfrak{R})$ und eine Eingabe $v \in A^*$.

Frage: Gibt es ein $w \in A^*$ mit $v \overset{\mathrm{M}}{\Rightarrow}_{\bullet} w$? ■

Das erste 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

[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.

mit einer der Anstöße zur Präzisierung des Algorithmusbegriffs. Normalerweise wird das Beispiel 3.4.7.b, das Halteproblem, in der Literatur mittels Turingmaschinen formuliert. Seine Unlösbarkeit hat eine entscheidende Auswirkung auf die Informatik, denn es gilt: **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. Nachfolgend geben wir ein Beispiel für einen Terminierungsbeweis an. Weitere Verfahren zum Beweis von Terminierung werden wir im Laufe der Skriptums noch kennenlernen.

3.4.8 Beispiel (Terminierung)

Für den Markov-Algorithmus aus Beispiel 3.4.6.b gilt

$$|^n \overset{\mathrm{M}}{\Rightarrow}_\bullet \begin{cases} \mathrm{O} & \text{falls n ungerade} \\ \mathrm{L} & \text{falls n gerade.} \end{cases}$$

Der Beweis erfolgt durch noethersche Induktion auf der geordneten Menge $(\mathbb{N}, \leq)$:

Induktionsbeginn: Falls $n = 0$ gilt, so ist nur die Regel (3) anwendbar und bringt sofort die Beziehung

$$|^0 = \varepsilon \Rightarrow_\bullet \mathrm{L}.$$

Induktionsschluß: Es sei $n > 0$. Wir unterscheiden drei Fälle. Gilt $n = 1$, so haben wir

$$|^1 = | \Rightarrow_\bullet \mathrm{O},$$

denn es ist nur Regel (2) anwendbar. Ist n ungerade und $n \neq 1$, so impliziert dies

$$|^n = |||^{n-2} \Rightarrow |^{n-2} \overset{\mathrm{M}}{\Rightarrow}_\bullet \mathrm{O}$$

nach Regel (1) (ganz links angewendet!) und der Induktionshypothese, denn auch $n-2$ ist ungerade. Ist n hingegen gerade, so haben wir schließlich

$$|^n = |||^{n-2} \Rightarrow |^{n-2} \overset{\mathrm{M}}{\Rightarrow}_\bullet \mathrm{L}$$

ganz analog zu eben nach Regel (1) und der Induktionshypothese. ■

4 Funktionale Programmierung: Einführung

Programmiersprachen fallen 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.

4.1 Grundlegendes

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.

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 mittels Termersetzung verwendet. Wir wollen dies im folgenden an zwei einfachen Beispielen aus der Arithmetik demonstrieren.

4.1.1 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

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

[8]Von einer applikativen Sprachen 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 $\mathit{isperfect}(6) = \mathit{tt}$ wegen

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

Der Zusammenhang zwischen den Gleichungen (P_1) bis (P_4) und der Berechnung des Resultats tt des Aufrufs $\mathit{isperfect}(6)$ 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.

4.1.2 Beispiel (einer rekursiv beschriebenen Funktion)

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

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

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

$$\mathit{isqrt}(0) = 0. \tag{Q_1}$$

Da für die Null damit das Resultat bekannt ist, sei nun $n \neq 0$. Wir zielen wiederum auf eine Rekursion ab und nehmen deshalb $x = \mathit{isqrt}(n-1)$ 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 4.1.1, 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. $(x - 1) \not\mid n$ 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.

4.1.3 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 4.1.1, 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 4.1.1 in eine spezielle Notation handelt. Dabei sind `int` und `bool` die Sorten für $\mathbb{Z}$ bzw. $\mathbb{B}$ und der Teilbarkeitstest von $x-1$ durch n 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 4.1.2, 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 ist eine relativ junge Sprache. Sie entstand Ende der 70er Jahre 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. Dieses System ist normalerweise interpretierend, erlaubt jedoch durch spezielle Kommandos auch eine Kompilation zur Effizienzsteigerung. ■

Wir haben in Abschnitt 4.1.3 die beiden ML-Programme in Schreibmaschinenschrift gesetzt, ebenso Teile daraus im umgebenden Text. Diese Schreibweise werden wir auch im weiteren Verlauf des Skriptums beibehalten.

4.2 Elementare Datenstrukturen

In Abschnitt 3.3 hatten wir Signaturen $\Sigma = (S, K, F)$ eingeführt, darüber Σ-Terme definiert und 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 **Konstanten** bzw. **Operationen** genannt.

b) Es gibt eine Algebra zur Signatur von ML, welche die Sorten, Konstanten und Operationen interpretiert. Für diese Algebra A gelten 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 Festlegung 3.3.3 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 3.3.9 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 zusätzlich immer noch `=` : m`,`m `-> bool` als Gleichheits- bzw. `<>` : m`,`m `-> bool`, als Ungleichheitsoperation definiert sind.

4.2.1 Einelementiger Bereich

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

Der einelementige Bereich wird später benötigt, um aus den elementaren Datenstrukturen durch in ML fest vorgegebene Konstruktionsmechanismen neue, zusammengesetzte Datenstrukturen zu erzeugen.

4.2.2 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.

4.2.3 Ganze Zahlen

Sorte:	`int`	Ganze Zahlen $\mathbb{Z}$
Konstanten:	0, 1, ...	Symbole für 0, 1 ...
Operationen:	`~ : int -> int`	Einstelliges Minus
	`+, -, * : int,int -> int`	Arithmetik
	`div : int,int -> int`	Ganzzahlige Division
	`mod : int,int -> int`	Rest bei ganzzahliger Division
	`min, 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 `min`, `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 0 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. Auch bei der nachfolgenden Datenstruktur legen wir eine idealisierende Interpretation zugrunde, obwohl tatsächlich nur die sogenannten Maschinenzahlen mit ihren Operationen vorliegen.

4.2.4 Reelle Zahlen

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

Im Beispiel `10.83E3` von Abschnitt 4.2.4 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 `/`, `sqrt` und `ln` gelten die üblichen Undefiniertheiten. Zu den beiden Operationen `real` und `truncate` ist noch eine Bemerkung angebracht, die wir, wegen ihrer allgemeinen Bedeutung für Programmiersprachen, als eigenen Punkt notieren.

4.2.5 Sortenanpassung

In vielen Programmiersprachen sind zwischen Sorten sogenannte **Teilsortenrelationen** festgelegt. Sind m eine Sorte und n eine Teilsorte von m, so kann eine Operation $f : m \to p$ auch auf einen Term t der Sorte n angewendet werden. Für die Sprache ML gilt jedoch: **Es gibt keine Teilsortenrelationen**. Alle Sortenanpassungen müssen somit **explizit** vorgenommen werden, so von `int` nach `real` durch die Operation `real` von Punkt 4.2.4 bzw. durch spezielle Schreibweisen wie `1.0`, `2.0` usw. Die Umwandlung einer reellen Zahl in eine ganze Zahl geschieht durch die Operation `truncate`, durch die der Nachkommateil abgehackt wird. Die Anwendung dieser Operation entspricht also keiner Rundung! ■

Andere Sprachen, wie Modula-2, das wir später noch besprechen werden, haben Teilsortenrelationen, und damit geschieht, bei einem Term $f(t)$ mit einem Funktionssymbol f und einem Term t wie in Abschnitt 4.2.5 vorausgesetzt, bei der Anwendung $f(t)$ eine **implizite Sortenanpassung**.

Nun zur letzten elementaren Datenstruktur von ML, den Zeichenreihen. Im folgenden sei Z der Zeichenvorrat von ML, bestehend aus den kleinen und großen Buchstaben, den Ziffern und einigen Sonderzeichen wie &, _, %, $ und so fort.

4.2.6 Zeichenreihen

Sorte:	`string`	Zeichenreihen Z^*
Konstanten:	Zeichenreihen mit " begrenzt	z.B. `"Rudolf"`
Operationen:	`size : string -> int`	Länge
	`^ : string,string -> string`	Konkatenation
	`chr : int -> string`	Zeichenzuordnung
	`ord : string -> int`	Code des ersten Zeichens
	`substring : string,int,int -> string`	Teilzeichenreihe
	`<, <=, >, >= : string,string -> bool`	Lexikograph. Vergleiche ■

Für die Konkatenation und die lexikographischen Vergleichsoperationen ist die Infixschreibweise vorgesehen. Die drei einstelligen Operationen `size`, `ord` und `chr` werden, wie auch die dreistellige Operation `substring`, in der funktionalen Schreibweise notiert. Die Bedeutung von `ord`, `chr` und `substring` ist dabei wie folgt:

a) Durch `chr` wird einer natürlichen Zahl i mit $0 \leq i \leq 255$ eine einelementige Zeichenreihe zugeordnet. Etwa liefert `chr(65)` die Zeichenreihe `"A"` und `chr(55)` die Zeichenreihe `"7"`. Der Hintergrund dieser Zuordnung ist die sogenannte ASCII-Codierung von Zeichen.

b) Die Operation `ord` stellt eine Art Umkehrfunktion zu `chr` dar. Sie bildet eine nichtleere Zeichenreihe der Form aw, mit einer einelementigen Zeichenreihe a, auf die natürliche Zahl n ab, deren `chr`-Wert gleich a ist.

c) 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 einfacher Weise die Restbildung auf Zeichenreihen realisieren. Auch das erste Element einer Zeichenreihe kann durch sie (als einelementige Zeichenreihe) einfach berechnet werden, nämlich mittels `substring(s,0,1)`.

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

4.3 Rekursive Rechenvorschriften

Das zentrale Konzept einer funktionalen Programmiersprache ist die rekursive Rechenvorschrift, die eine mathematische Funktion beschreibt. Oft werden Rechenvorschriften auch Funktionen genannt, genauer ML-Funktionen, Modula-2-Funktionen usw. Wir unterscheiden in der Sprechweise präziser und bezeichnen als

- **(partielle) Funktionen** diejenigen mathematischen Gebilde $f : M \longrightarrow N$, die in Abschnitt 2.1 als spezielle Relationen eingeführt wurden,
- und als **ML-Rechenvorschriften** gewisse Teile von ML-Programmen, die einer speziellen Syntax genügen und deren Semantiken (Werte, Interpretationen) (partielle) Funktionen in dem eben beschriebenen mathematischen Sinne sind.

So ist etwa die Semantik der ML-Rechenvorschrift `isperfect` von Beispiel 4.1.3 zum Testen perfekter Zahlen die partielle Funktion

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

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.

4.3.1 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 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 Sorte m_i $(1 \leq i \leq k)$, Resultatsorte n und Rumpf t wie folgt:

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

Dabei ist das Semikolon als Abschlußzeichen Teil der Deklaration. 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 Bezeichnungen aus der ML-Signatur bzw. schon andersweitig eingeführten Bezeichnungen 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 4.2.

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 Modula-2 in den Kapiteln über imperative Programmierung beibehalten. Es ist dann jeweils eine einfache Übungsaufgabe, die umgangssprachliche Syntax zu formalisieren, etwa mittels einer Grammatik oder durch Syntaxdiagramme.

Bei der Festlegung von Rechenvorschriften in Abschnitt 4.3.1 handelt es sich nicht um die gesamte Sprache ML. Später werden wir beispielsweise in der Funktionalität statt Sorten auch komplexere Datenstrukturen 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 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 4.1.1 und 4.1.2 und ihre Übertragung nach ML motiviert wurde.

4.3.2 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 diese Festlegung 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 diese Rechenvorschrift für ein nichtnegatives Argument die Fakultät 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 3.1.1 eingeführt, so können wir festlegen:

$$reverse(s_1 \dots s_n) = s_n \dots s_1 \tag{R_1}$$

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

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

wählt, oder aber die Funktion *reverse* induktiv durch

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

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 von *reverse* in ML interessiert, so ist hierfür die Spezifikation (R_3) der geeignetste Ausgangspunkt. Aufbauend auf die Bemerkungen am Ende von Abschnitt 4.2 und (R_3) erhalten wir sofort das folgende Programm:

```
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);
```

Diese Rechenvorschrift 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

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

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.

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 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.

4.3.3 Systeme von Rechenvorschriften

Es gibt Situationen, in denen Rekursion **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:

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

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

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.

Grundsätzlich ist die Zusammenfassung von 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

```
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 gewählten 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 Gestalt (R) aus Abschnitt 4.3.1 oder Systeme von Rechenvorschriften mit der eben beschriebenen Syntax deklariert werden. Auch hier wird die Verwendungsreihenfolge durch die Aufschreibung festgelegt.

4.3.4 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 Sinne von Abschnitt 4.3.3 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.

4.3.5 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`:

$$\begin{aligned} \texttt{S(0)} &= \texttt{if 0 = 0 then 1} \\ &\qquad\qquad\quad \texttt{else S(S(0-1)-1) + 1} \\ &= \texttt{1} \end{aligned}$$

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

$$\begin{aligned} \texttt{S(n)} &= \texttt{if n = 0 then 1} \\ &\qquad\qquad\quad \texttt{else S(S(n-1)-1) + 1} \\ &= \texttt{S(S(n-1)-1) + 1} \\ &= \texttt{S(n-1) + 1} && \text{Induktionshypothese} \\ &= \texttt{n + 1} && \text{Induktionshypothese} \end{aligned}$$

b) Die Rechenvorschriften `iseven` und `isodd` von Abschnitt 4.3.3 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 der folgenden 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 4.3.2. 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 4.3.2 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 4.3.5 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 nur für diesen**.

4.4 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 4.3.1. 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.

4.4.1 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 4.3.5 haben wir Expandieren und Komprimieren fortlaufend verwendet. Nach diesen Erklärungen sind wir nun in der Lage, die erste Semantik funktionaler Sprachen anzugeben.

4.4.2 Termersetzungssemantik von Rechenvorschriften

Wir setzen voraus, daß eine einzelne Rechenvorschrift F der Form (R) von Abschnitt 4.3.1 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

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

   der Fallunterscheidung so weit wie möglich.

Ist die so konstruierte Folge endlich 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 zwei 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).

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. ■

[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.

Die in Abschnitt 4.4.2 erwähnten Regeln expandieren jeweils genau einen Aufruf und sind deshalb technisch einfach zu realisieren. Die Forderung, daß so weit wie möglich links expandiert wird, heißt insbesondere, daß von den rekursiven Aufrufen in Fallunterscheidungen ggf. die der Bedingung zuerst auszuwerten sind.

Es war lange nicht klar, daß verschiedene Parameterübergabe-Mechanismen zu verschiedenen Ergebnissen führen können. Entdeckt wurde dieser Sachverhalt erst am Ende der 60er Jahre, als man sich intensiver mit Semantik beschäftigte. Nachfolgend geben wir ein berühmtes Beispiel an, das auf J.H. Morris (1968) zurückgeht.

4.4.3 Call-by-value ist ungleich Call-by-name

Die folgende Rechenvorschrift zeigt, daß die Regeln Call-by-value und Call-by-name aus Abschnitt 4.4.2 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 von Abschnitt 4.4.2 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
		⋮	

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.

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.

4.4.4 Beispiel (Verhinderung von Mehrfachauswertung)

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 $1 + 2 + \ldots + 2^{k+1}$ Aufrufen, also, nach der Summenformel der geometrischen Reihe, zu insgesamt zu $2^{k+2} - 1$ Aufrufen. Eine Effizienzverbesserung erhält man, indem man eine Rechenvorschrift zum Quadrieren 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:

d_0	:	`power(2,16)`
d_1	:	`quad(power(2,8))`
d_2	:	`quad(quad(power(2,4)))`
d_3	:	`quad(quad(quad(power(2,2))))`
d_4	:	`quad(quad(quad(quad(power(2,1)))))`
d_5	:	`quad(quad(quad(quad(2 * quad(power(2,0))))))`
d_6	:	`quad(quad(quad(quad(2 * quad(1)))))`
d_7	:	`quad(quad(quad(quad(2))))`
d_8	:	`quad(quad(quad(4)))`
d_9	:	`quad(quad(16))`
d_{10}	:	`quad(256)`
d_{11}	:	`65536`

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. ■

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

4.4.5 Beispiel (zur Motivation)

Wir betrachten noch einmal die Rechenvorschrift `fac` von Beispiel 4.3.2.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 Funktion $f : \mathbb{Z} \longrightarrow \mathbb{Z}$ beschrieben, welche die folgenden zwei Bedingungen erfüllt:

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

Mit Hilfe eines **Funktionals**, d.h. einer Funktion, die (partielle) Funktionen auf (partielle) Funktionen abbildet, lassen sich die beiden Formeln (1) und (2) in eine Gleichung zusammenfassen. Definiert man nämlich ein Funktional

$$\tau_{\mathtt{fac}} : \mathbb{Z}^{\mathbb{Z}} \longrightarrow \mathbb{Z}^{\mathbb{Z}},$$

das einer Funktion $f : \mathbb{Z} \longrightarrow \mathbb{Z}$ eine Funktion $\tau_{\mathtt{fac}}[f] : \mathbb{Z} \longrightarrow \mathbb{Z}$ zuordnet, elementweise durch

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

so sind die beiden Formeln (1) und (2) äquivalent zur allquantifizierten 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[10], 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. Dazu betrachten wir wiederum unser 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

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

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

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

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

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

[10] Man vergleiche dies mit der Definition von (partiellen) Funktionen als spezielle Relationen, also Mengen, in Abschnitt 2.1.

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

$$\begin{array}{rcll} \tau_{\texttt{fac}}[f_1](n) & = & n * f_1(n-1) & \text{Definition von } \tau_{\texttt{fac}} \\ & = & n * \text{undef} & \text{Definition von } f_1 \\ & = & \text{undef} & \\ & = & f_1(n) & \text{Definition von } f_1 \end{array}$$

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

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:

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

Vergleicht man die eben erzielten Ergebnisse mit der Termersetzungssemantik aus Abschnitt 4.4.2, so entspricht die termmäßige Auswertung der Fakultätsrechenvorschrift offensichtlich der Funktion f_1, da für ein negatives Argument `n` die Folge

`fac(n),n*fac(n-1),n*(n-1)*fac(n-2),...`

von Termen 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 4.2 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 Termersetzungssemantik aus Abschnitt 4.4.2 verträglich ist.

4.4.6 Definition (Induziertes Funktional)

Wir gehen aus von einer ML-Rechenvorschrift der Form

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

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 ist 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:

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

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 4.4.6 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.

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 4.4.6 betrachtete Rechenvorschrift. Wie schon das motivierende Beispiel 4.4.5 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 sieht wie folgt aus: Es seien $f, g : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ zwei partielle Funktionen. Man definiert, wiederum 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

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

Betrachtet man f und g als eindeutige Relationen, so gilt $f \leq g$ genau dann, wenn $f \subseteq g$ zutrifft. Offensichtlich folgt daraus:

4.4.7 Satz (Funktionsordnung)

Die durch $(*)$ definierte Relation ist eine Ordnungsrelation auf der Menge $[\prod_{i=1}^{k} m_i^A \to n^A]$ der partiellen Funktionen. von $\prod_{i=1}^{k} m_i^A$ nach n^A. ■

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 4.4.5 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[11]. So ein kleinster Fixpunkt von τ_F existiert immer.

[11]Eine formale Rechtfertigung dieser Wahl ist im Rahmen dieser Einführungsvorlesung natürlich nicht möglich.

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.

4.4.8 Satz (Fixpunktsatz)

Für eine ML-Rechenvorschrift F wie in Definition 4.4.6 betrachtet, gelten die folgenden Aussagen:

a) Das von F induzierte Funktional τ_F besitzt bezüglich der durch $(*)$ definierten Ordnung 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 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 von τ_F auf 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.

4.4.9 Denotationelle Semantik von Rechenvorschriften

Für eine ML-Rechenvorschrift F wie in Definition 4.4.6 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 4.4.5 noch einmal auf und zeigen, daß die in ihm angegebene Rechenvorschrift tatsächlich die partielle Funktion f_1 als denotationelle Semantik besitzt.

4.4.10 Beispiel (zur denotationellen Semantik)

Für die ML-Rechenvorschrift `fac` von Beispiel 4.4.5 zur Berechnung der Fakultät einer natürlichen Zahl haben wir das induzierte Funktional $\tau_{\texttt{fac}}$ schon in diesem Beispiel angegeben. Nachfolgend berechnen wir das Funktional $\tau_{\texttt{fac}}$ mit Hilfe der in Definition 4.4.6

angegebenen formalen Festlegung:

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

Zur Bestimmung des kleinsten Fixpunktes iterieren wir nun gemäß dem Fixpunktsatz 4.4.8. 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 als 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 4.4.5 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

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

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 4.4.5.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

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

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

Es sei $n < 0$. Dann gilt die Implikation (†), 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 (†) 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:

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

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

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

b) Zu einem zweiten Beweis durch Widerspruch nehmen wir an, daß f_1 nicht der kleinste Fixpunkt von $\tau_{\texttt{fac}}$ 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 $\tau_{\texttt{fac}}[f] = f$ muß $n_0 > 0$ gelten. Wir nehmen nun zusätzlich an, daß n_0 bezüglich der üblichen Ordnung auf $\mathbb{N}$ minimal gewählt ist. Also gilt die Ungleichung $f(n_0 - 1) \neq \text{undef}$ und dies bringt

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

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.

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

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

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

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

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, haben wir $\mu(\tau_{\mathtt{fac}}) \leq fac_{\mathbb{Z}}$, also

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

für alle $n \in \mathbb{Z}$. Die Semantik der Rechenvorschrift stimmt mit der Spezifikation somit nur auf dem definierten Bereich überein. Damit jetzt, von dem eben 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.

4.4.11 Satz

Für eine ML-Rechenvorschrift F wie in Definition 4.4.6 betrachtet und eine partielle Funktion $f : \prod_{i=1}^{k} m_i^A \longrightarrow n^A$ gilt:

$$\frac{f = \tau_F[f] \qquad \forall\, \langle v_1, \ldots, v_k\rangle \in \prod_{i=1}^{k} m_i^A : f(v_1, \ldots, v_k) \neq \text{undef} \rightarrow \mu(\tau_F)(v_1, \ldots, v_k) \neq \text{undef}}{\mu(\tau_F) = f}$$

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

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

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 4.4.11 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 4.1.1, 4.1.2, 4.3.2, 4.3.5.b und 4.3.5.c waren diese Bedingungen erfüllt und somit die Übergänge korrekt. Dies muß aber nicht immer so sein. Nachfolgend geben wir ein Beispiel an.

4.4.12 Beispiel (für eine inkorrekte bzw. korrekte Entwicklung)

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

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

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

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

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. ■

Wie die Termersetzungssemantik, so kann natürlich auch die denotationelle Semantik auf mehrere Rechenvorschriften ausgedehnt werden. Handelt es sich um Deklarationen

$$\texttt{fun } F_1 \ldots \texttt{; fun } F_2 \ldots \texttt{; } \ldots \texttt{ fun } F_k \ldots \texttt{;}$$

ohne Systeme mit möglicherweise verschränkter Rekursion, wo in den Rechenvorschriften $F_i, 2 \leq i \leq k$, also nur Aufrufe von schon früher deklarierten Rechenvorschriften vorkommen dürfen, so verallgemeinert man zuerst die Definition 4.4.6 des induzierten Funktionals τ_F um einen zusätzlichen Punkt:

c') Ist der Term t von der Form $G(t_1, \ldots, t_n)$ mit einer **schon deklarierten Rechevorschrift** G, so gilt:

$$[\![G(t_1, \ldots, t_n)]\!][f](v) = \mu(\tau_G)([\![t_1]\!][f](v), \ldots, [\![t_n]\!][f](v))$$

Dann verfährt man wie bisher. Kommt jedoch ein System mit möglicherweise verschränkter Rekursion vor, so wird die Sache komplizierter. So ein System muß man als Gesamtheit auffassen und zu Funktionalen übergehen, die Tupel von (partiellen) Funktionen in ebensolche abbilden. Auch solche tupelwertigen-Funktionale besitzen einen kleinsten Fixpunkt und dieses Tupel von (partiellen) Funktionen wird als Semantik des Systems erklärt. Genaueres dazu erfährt man in einer Semantikvorlesung im Hauptstudium.

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:

4.4.13 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 aus Abschnitt 4.4.2 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[12] $e_i, 1 \leq i \leq k$, terminiert, soferne 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 4.3.2 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.

Man beachte, daß in Definition 4.4.13 das eventuell existierende letzte Folgenglied ein Grundterm ist und man somit den Wert gemäß Definition 3.3.9 festlegen kann. Im allgemeinen ist das Feststellen von Terminierung algorithmisch nicht lösbar. Das folgt aus der Church'schen These und der Unlösbarkeit des Halteproblems (siehe Beispiel 3.4.7.b). 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.

4.4.14 Verfahren zum Terminierunsbeweis

Gegeben sei eine ML-Rechenvorschrift der Form von Definition 4.4.13. Diese Rechenvorschrift terminiert unter einer Vorbedingung, falls die folgenden zwei Eigenschaften gelten:

a) Unter der Gültigkeit der Vorbedingung und der Bedingungen der Fallunterscheidungen[13] 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

[12] Der Ausdruck „definierter Term" t steht hier, wie allgemein üblich, als Kurzform für das Definiertsein des Wertes von t.

[13] 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.

Argumente bezüglich δ echt vermindert werden, falls die Vorbedingung und die Bedingungen der umgebenden Fallunterscheidungen gültig sind. ■

Nach a) und b) ist die Folge $d_0, d_1, \ldots$ der Termersetzungssemantik aus Abschnitt 4.4.2 endlich, da jede echt absteigende Kette $x_1 > x_2 > \ldots$ in noethersch geordneten Mengen $(M, \leq)$ endlich ist, und das letzte Folgenglied ist auch definiert. Auf die in Abschnitt 4.4.14.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:

d_0	:	`F(10)`
d_1	:	`F(9) + 10`
d_2	:	`F(8) + 19`
$\vdots$		$\vdots$
d_{10}	:	`F(0) + 55`
d_{11}	:	`(10 div 0) + 55`,

Der Wert des letzten Terms ist wegen der Semantik der Basisoperation `div` offensichtlich undefiniert. Hier ist der Punkt a) von Abschnitt 4.4.14 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.

4.4.15 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 Punkt a) des Verfahrens 4.4.14, die Anwendungen der Operationen: Die Addition ist total und die ganzzahlige Division durch 2 ist ebenfalls immer definiert.

Nun zu Punkt b) des Terminierungs-Verfahrens 4.4.14. Wir betrachten die noethersche Ordnung $(\mathbb{N}, \leq)$ und die Terminierungsfunktion

$$\delta : \mathbb{Z} \longrightarrow \mathbb{N} \qquad\qquad \delta(n) = \begin{cases} n & : \quad n \geq 0 \\ 0 & : \quad \text{sonst.} \end{cases}$$

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 isodd(n) then F((3*n+1) div 2)
               else n div 2;
```

(wobei der Aufruf `isodd(n)` der vorausgesetzten Operation `isodd` testet, ob `n` ungerade ist, oder nicht) terminiert, haben wir wiederum die Punkte a) und b) des Verfahrens 4.4.14 zu betrachten.

Punkt a) des Verfahrens 4.4.14 ist trivial. Zu Punkt b) von 4.4.14 betrachten wir die noethersche Ordnung $(\mathbb{N}, \leq)$ und die Terminierungsfunktion

$$\delta : \mathbb{N} \longrightarrow \mathbb{N} \qquad\qquad \delta(n) = \begin{cases} 0 & : \quad n \text{ gerade} \\ 1 + \delta(\frac{n-1}{2}) & : \quad n \text{ ungerade.} \end{cases}$$

Offensichtlich terminiert die Rekursion von δ; die Funktion ist also tatsächlich total[14]. 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

$$\delta(\frac{3 * n + 1}{2}) < \delta(n).$$

Der Beweis erfolgt durch noethersche Induktion. Zum **Induktionsbeginn** gelte $n = 1$. Dann folgt daraus

$$\begin{aligned} \delta(\tfrac{3*1+1}{2}) &= \delta(2) && \text{Arithmetik} \\ &= 0 && \text{Definition } \delta \\ &< 1 && \\ &= 1 + \delta(0) && \text{Definition } \delta \\ &= \delta(1) && \text{Definition } \delta. \end{aligned}$$

[14]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.

Für den **Induktionsschluß** setzen wir die Ungleichung $n > 1$ voraus und betrachten dann zwei Fälle.

Ist $\frac{3*n+1}{2}$ gerade, so erhalten wir ohne Verwendung der Induktionshypothese die gewünschte Abschätzung wie folgt:

$$\begin{array}{rll} \delta(\frac{3*n+1}{2}) & = 0 & \text{Definition } \delta \\ & < 1 & \\ & \leq 1 + \delta(\frac{n-1}{2}) & \\ & = \delta(n) & \text{Definition } \delta \end{array}$$

Nun sei $\frac{3*n+1}{2}$ ungerade. Hier benötigen wir zum Beweis der gewünschten Abschätzung die Induktionshypothese:

$$\begin{array}{rll} \delta(\frac{3*n+1}{2}) & = 1 + \delta(\frac{\frac{3*n+1}{2}-1}{2}) & \text{Definition } \delta \\ & = 1 + \delta(\frac{3*\frac{n-1}{2}+1}{2}) & \text{Arithmetik} \\ & < 1 + \delta(\frac{n-1}{2}) & \text{Induktionshypothese, } \frac{n-1}{2} < n \\ & = \delta(n) & \text{Definition } \delta \end{array}$$

c) Als ein letztes Beispiel zur Terminierung 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 isodd(n) 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 (wie der rekursiven Pfadordnung oder der Multiset-Ordnung) zu führen hat.

5 Funktionale Programmierung: Vertiefung

In Kapitel 4 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.

5.1 Zusammengesetzte Datenstrukturen

Bisher kennen wir von ML nur die elementaren Datenstrukturen. Diese sind gegeben durch die Sorten `unit`, `bool`, `int`, `real`, `string`, und die entsprechenden Operationen (siehe Abschnitt 4.2), 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.

5.1.1 Typen

Es seien S die Menge der Sorten von ML und A die Algebra zu deren Interpretation, wie am Anfang von Abschnitt 4.2 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 $m_1 \texttt{ * } \ldots \texttt{ * } m_k$ ein Typ, genannt **Produkttyp** mit den Komponententypen m_i, $1 \leq i \leq k$.

(3) Sind m und n Typen, so ist auch $m \texttt{ -> } n$ 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) $I[m \texttt{ -> } n] = [I[m] \to I[n]] = \{\ f\ :\ f \text{ partielle Funktion von } I[m] \text{ nach } I[n]\ \}$

Beim Typaufbau sind in ML auch die Klammern `(` und `)` erlaubt. Um solche zu sparen, haben wir in Punkt a), der syntaktischen Festlegungen der ML-Typen, schon angenommen, 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)`. ■

Im nachfolgenden Beispiel berechnen wir, aufbauend auf die Festlegungen von Abschnitt 5.1.1.b, formal die Interpretation eines Funktionstyps mit einem Produkttyp sowohl als Argument- als auch Resultattyp.

5.1.2 Beispiel (für Typinterpretation)

Für zwei Mengen M_1 und M_2 bezeichnet $M_1 \times M_2$ ihr zweifaches direktes Produkt. Damit bekommen wir beispielsweise für den Typ `int * int -> bool * real` als Interpretation

$$\begin{aligned} I[\texttt{int * int -> bool * real}] &= [I[\texttt{int * int}] \to I[\texttt{bool * real}]] \\ &= [I[\texttt{int}] \times I[\texttt{int}] \to I[\texttt{bool}] \times I[\texttt{real}]] \\ &= [\texttt{int}^A \times \texttt{int}^A \to \texttt{bool}^A \times \texttt{real}^A] \\ &= [\mathbb{Z} \times \mathbb{Z} \to \mathbb{B} \times \mathbb{R}], \end{aligned}$$

also die Menge aller partiellen Funktionen $f : \mathbb{Z} \times \mathbb{Z} \longrightarrow \mathbb{B} \times \mathbb{R}$. ■

Im letzten Kapitel 4 hatten wir ein ML-Programm erklärt als eine Folge von Deklarationen, wobei jede Deklaration mit einem Semikolon als Abschlußzeichen endete und entweder aus einer einzelnen Rechenvorschrift oder aus einem System von Rechenvorschriften bestand. Man vergleiche noch einmal mit Abschnitt 4.3. 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 beliebiger Reihenfolge** erfolgen. Wichtig ist nur, daß eine Deklaration eines Begriffs vor seiner ersten Verwendung erfolgt und daß sich keine Namenskonflikte mit den Bezeichnungen der ML-Signatur bzw. den schon deklarierten Rechenvorschriften und Typen ergeben. Auf diese allgemeinen Kontextbedingungen werden wir im weiteren aber nicht mehr eingehen.

Es gibt in ML zwei Arten von Typdeklaration. Wir beginnen mit der einfacheren der beiden.

5.1.3 Typdeklarationen 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[15]). Durch die Typdeklaration

$$\texttt{type}\ m\ \texttt{=}\ n\texttt{;} \qquad (T_A)$$

[15] In diesem Zusammenhang spricht man dann auch von „Typausdrücken“ statt von Typen.

führt man die Bezeichnung m **als Abkürzung** für den Typ n ein. Nach einer Deklaration der Gestalt (T_A) darf man im restlichen ML-Programm überall die Bezeichnung m statt des eventuell komplizierteren Typausdrucks n schreiben. ■

Wichtiger als die eben erklärte erste Art von Typdeklaration in der Sprache ML, die im Jargon wegen des verwendeten Schlüsselworts `type` auch **type-binding** genannt wird, ist die in Abschnitt 5.1.5 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.

5.1.4 Disjunkte Mengenvereinigung

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 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

$$\imath_1(x) = \langle x, 1\rangle \qquad\qquad \imath_2(x) = \langle x, 2\rangle$$

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.

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

$$\biguplus_{i=1}^{n} M_i := \bigcup_{i=1}^{n}\ (M_i \times \{\ i\ \})$$

und, für k mit $1 \leq k \leq n$, die k-te **kanonische Injektion** $\imath_k : M_k \longrightarrow \biguplus_{i=1}^{n} M_i$ als

$$\imath_k(x) = \langle x, k\rangle.$$

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.

5.1.5 Typdeklarationen 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) von Abschnitt 5.1.5 die nachfolgende Situation vorliegen, wobei die Pfeile die Funktionalitäten der Konstruktoren anzeigen:

$$\begin{tikzcd} & & m & \\ n_1 \arrow[urr, "con_1"] & n_2 \arrow[ur, "con_2"'] & \cdots & n_k \arrow[ul, "con_k"'] \end{tikzcd}$$

Nach diesen abstrakten und vielleicht auch etwas ungewohnten Begriffen behandeln wir nun mit ihrer Hilfe ein konkretes Beispiel.

5.1.6 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 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 **Ortsvektor** $\boldsymbol{x} = \langle x_1, x_2 \rangle \in \mathbb{R} \times \mathbb{R}$ und **Richtungsvektor** $\boldsymbol{y} = \langle y_1, y_2 \rangle \in \mathbb{R} \times \mathbb{R}$. Bei dieser Darstellung ist 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:

$$\begin{tikzcd}
& \texttt{line} & \\
\texttt{normform} \arrow[ur, "\texttt{nfline}"] & & \texttt{vecform} \arrow[ul, "\texttt{vfline}"'] \\
\texttt{real * real * real} \arrow[u, "\texttt{mnf}"'] & & \texttt{vector * vector} \arrow[u, "\texttt{mvf}"']
\end{tikzcd}$$

Nun betrachten wir eine konkrete Gerade, nämlich die identische Relation auf $\mathbb{R}$. Es sei also $g = \{ \ \langle x, x \rangle \ : \ x \in \mathbb{R} \ \}$. 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, liefert 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.

5.1.7 Erweiterung der Rechenvorschriften

In Abschnitt 4.3 waren in einer Rechenvorschrift, die auch Teil eines Systems sein konnte, in der Funktionalität nur Parameter $x : m$ mit einer Sorte m zugelassen worden. 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 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 4.3.1.a bis 4.3.1.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 in Abschnitt 5.1.7 angegebenen syntaktischen Erweiterungen sind natürlich auch die in Abschnitt 4.4 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.

Wir wollen zum Ende dieses Abschnittes noch kurz ein Beispiel für eine typische Verwendung eines Typs mit Konstruktoren skizzieren, nämlich, wenn das Resultat einer Berechnung von zweierlei Gestalt ist. Wir können das gestellte Problem aber noch nicht im Detail in ML lösen, da uns dazu noch die entsprechenden Sprachmittel fehlen. Diese werden erst in Abschnitt 5.6 eingeführt.

5.1.8 Beispiel (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. Mit der Menge $\mathbb{N}_\infty$ definiert durch $\mathbb{N}_\infty := \mathbb{N} \cup \{\infty\}$ ist eine sehr elegante Formalisierung des Problems möglich und gegeben durch die Funktion

$$pm : A^* \times A^+ \longrightarrow \mathbb{N}_\infty \qquad\qquad pm(s,p) = min\{\ i \in \mathbb{N}_\infty\ :\ s_i \ldots s_{i+|p|-1} = p\ \}.$$

Dabei werden die natürlichen Zahlen um ein zusätzliches größtes Element ∞ erweitert und dieses wird als das Minimum der leeren Menge definiert. Somit wird durch ∞ die negative Antwort „p ist keine Teilzeichenreihe von s" dargestellt. Weiterhin wird $s_i \ldots s_j$ für $j < i$ oder $s = \varepsilon$ als ε definiert. Erweitert man auch die Nachfolgerbildung auf $\mathbb{N}_\infty$ in der naheliegenden Weise $\infty + 1 = \infty$, so bekommt man durch eine einfache Rechnung, in der die Tatsache, daß p nicht leer ist, wesentlich eingeht, die folgende Rekursion:

$$pm(s,p) = \begin{cases} pm(rest(s),p) + 1 & : \ s \neq \varepsilon \wedge s_1 \ldots s_{|p|} \neq p \\ \infty & : \ s = \varepsilon \\ 1 & : \ \text{sonst} \end{cases}$$

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, mit `10000` als Ersatz für ∞, etwa wie folgt aus:

```
fun lp (p : string, s : string) : bool =
  if size(p) <= size(s) then substring(s,0,size(p)) = p
                        else false;

fun succ (x : int) : int =
  if x = 10000 then x + 1 else x;

fun pm (s : string, p : string) : int =
  if size(s) = 0 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 durch die Negation der Gleichung $s_1 \dots s_{|p|} = p$ 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 of unit | intres of int;
```

mit den zwei Konstruktoren `infty` und `intres` und eine Rechenvorschrift der folgenden Gestalt:

```
fun pm (s : string, p : string) : res = ... ;
```

Da uns, wie oben schon erwähnt, noch die entsprechenden Sprachmittel fehlen, können wir an dieser Stelle die Rechenvorschrift noch nicht vollständig angeben. ■

5.2 Rekursive Datenstrukturen

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.

5.2.1 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 zwei Mengen $\{\varepsilon\}$ und $A \times A^*$ einen leeren Durchschnitt haben (denn eine Zeichenreihe ist entweder leer oder nichtleer), bekommen wir für die Menge A^* der Zeichenreihen die Rekursion

$$A^* = \{\varepsilon\} \uplus A \times A^*,$$

indem wir die gewöhnliche Vereinigung und die disjunkte Vereinigung (wie in Abschnitt 5.1.4 bemerkt) identifizieren[16].

[16] Genaugenommen haben wir schon bei der Herrechnung der Rekursion von A^* eine Identifizierung vorgenommen, da wir das $n+1$-Tupel $\langle a, a_1, \dots, a_n \rangle$ im dritten Schritt gleichsetzten mit dem Paar $\langle a, \langle a_1, \dots, a_n \rangle \rangle$, dessen zweite Komponente ein n-Tupel ist.

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.

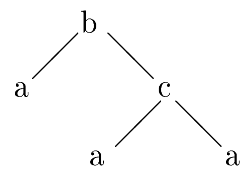
Die induktive Definition der Menge $\mathsf{Btree}(A)$ läßt sich mit Hilfe einer disjunkten Vereinigung sofort in die Rekursion

$$\mathsf{Btree}(A) = A \uplus \mathsf{Btree}(A) \times A \times \mathsf{Btree}(A)$$

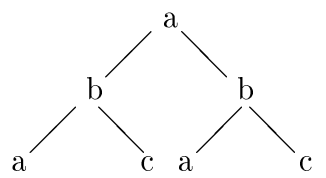
übertragen, wobei wir wiederum die gewöhnliche Vereinigung und die disjunkte Vereinigung identifizieren.

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:

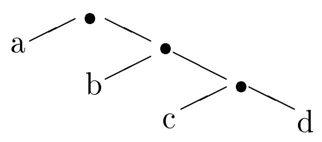
⟨a,b,⟨a,c,a⟩⟩



⟨⟨a,b,c⟩,a,⟨a,b,c⟩⟩



⟨a,•, ⟨b,•, ⟨c, • ,d⟩⟩⟩



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**[17].

[17]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.

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 $a \in \mathsf{Lisp}(A)$.

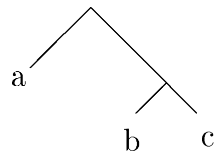
   (2) Gilt $l, r \in \mathsf{Lisp}(A)$, so auch $\langle l, r \rangle \in \mathsf{Btree}(A)$.

   Diese Definition führt sofort zur der folgenden Rekursion:

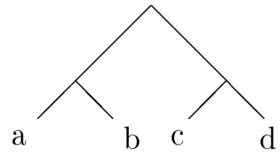
$$\mathsf{Lisp}(A) = A \uplus \mathsf{Lisp}(A) \times \mathsf{Lisp}(A)$$

   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:

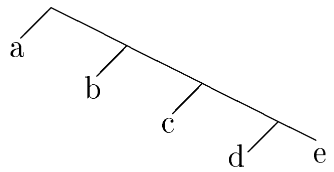
⟨a,⟨b,c⟩⟩



⟨⟨a,b⟩, ⟨c,d⟩⟩



⟨a,⟨b,⟨c,⟨d,e⟩⟩⟩⟩



   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 vorkommen. Diese Kontextbedingung wird in ML aber nur für Typdeklarationen als Abkürzungen verlangt.

Im Falle der Typdeklarationen mit Konstruktoren kann der Bezeichner auch rechts vorkommen, was zu einer weiteren Art von Typdefinition führt, die wir nun besprechen wollen.

5.2.2 Rekursive Datenstrukturen

Kommt in einer Deklaration mit Konstruktoren

$$\texttt{datatype}\ m\ \texttt{=}\ con_1\ \text{of}\ n_1\ \texttt{|}\ \dots\ \texttt{|}\ con_k\ \text{of}\ n_k\,\texttt{;}$$

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 definiert**.

Zur Bestimmung der Interpretation $I[m]$ eines wie eben rekursiv definierten Typs m geht man in einem ersten Schritt von der (syntaktischen) 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 von Mengen in Abschnitt 2.2.5, die kleinste Mengen spezifiziert, beweist man dann, daß die Gleichung $(*)$ eine kleinste Lösung D als Grenzwert der Kette

$$\emptyset \subseteq \mathcal{E}(\emptyset) \subseteq \mathcal{E}(\mathcal{E}(\emptyset)) \subseteq \dots$$

besitzt. In einem dritten Schritt definiert man schließ]lich 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 in Abschnitt 5.2.2 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 4.4.6 bei der Festlegung des von einer Rechenvorschrift induzierten Funktionals. Für den zweiten Punkt bedarf es eines Satzes ähnlich zum Fixpunktsatz 4.4.8. 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 5.2.1.

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.

5.2.3 Beispiele (Weiterführung von 5.2.1)

Im folgenden sei m ein Typ, dessen Interpretation jeweils die Menge A aus den Beispielen 5.2.1 sei. Dann übertragen sich die drei Rekursionsgleichungen unter der Einführung von Konstruktoren wie folgt in ML.

a) Wir beginnen mit den **Zeichenreihen**. Eine auf Beispiel 5.2.1.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 3.3.1, 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 $\mathcal{E}(X) = E \uplus A \times X$ gemäß Abschnitt 5.2.2 und erhalten als Resultat die Kette

$$\emptyset \subseteq E \subseteq E \cup A \subseteq E \cup A \cup A \times A \subseteq \ldots,$$

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 5.2.1.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 $\mathsf{Btree}(A)$[18]. 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 $A \times \mathsf{Btree}(A) \times A$ 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 **battmarkierten Binärbäumen** sieht die Übertragung der induktive Definition bzw. Rekursion aus Beispiel 5.2.1.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 battmarkierten Binärbäume zur Verfügung gestellt:

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 linearen Listen, gibt es so viele Anwendungen, daß sie in ML vordefiniert ist. Im folgenden gehen wir genauer auf die Datenstruktur der linearen Listen ein.

5.2.4 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

[18]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.

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

$$a_k \;\texttt{::}\; a_{k-1} \;\texttt{::}\; \ldots \;\texttt{::}\; a_1 \;\texttt{::}\; \texttt{nil}$$

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`. ■

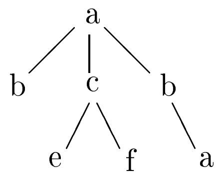
In Abschnitt 5.2.4 sind nicht alle Operationen auf Listen aufgeführt. Weitere Operationen findet man im Manual zum SML/NJ-System, das üblicherweise mit dem System ausgeliefert wird.

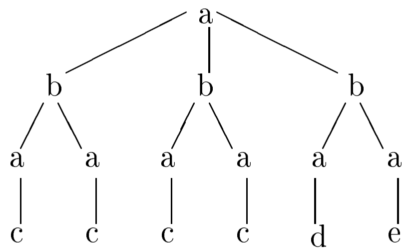
Durch obige Festlegung sind lineare Listen von Objekten beliebiger Sorte in unser bisheriges Typsystem eingeführt, und man kann nun mit den bisher vorgestellten Mitteln weiterkonstruieren. Dies führt etwa zu Listen von Listen, Bäumen von Listen, Listen von Bäumen usw., also zu einem sehr allgemeinen Konzept von Typen und Datenstrukturen.

5.2.5 Beispiel (Allgemeine Bäume)

In Beispiel 5.2.1 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 3.3.13, 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:

$$\langle \mathrm{a}, [\mathrm{b}, \langle \mathrm{c}, [\mathrm{e}, \mathrm{f}] \rangle, \langle \mathrm{b}, [\mathrm{a}] \rangle] \rangle$$

Das Beispiel oben rechts sieht in der Klammerschreibweise wie folgt aus:

$$\begin{array}{l} \langle \mathrm{a},\ [\ \langle \mathrm{b},\ [\langle \mathrm{a},\ [\mathrm{c}] \rangle, \langle \mathrm{a},\ [\mathrm{c}] \rangle]\ \rangle, \\ \qquad \langle \mathrm{b},\ [\langle \mathrm{a},\ [\mathrm{c}] \rangle, \langle \mathrm{a},\ [\mathrm{c}] \rangle]\ \rangle, \\ \qquad \langle \mathrm{b},\ [\langle \mathrm{a},\ [\mathrm{d}] \rangle, \langle \mathrm{a},\ [\mathrm{e}] \rangle]\ \rangle\ ] \rangle \end{array}$$

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 in Abschnitt 4.3.3 ein Beispiel mit verschränkter 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.

5.2.6 Systeme von rekursiven Datenstrukturen

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 Abschnitt 5.2.3 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 deshalb auch verschränkt-rekursive Typdeklarationen möglich, indem man zu Systemen übergeht und dies durch `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. ■

5.3 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 5.2.3 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.

5.3.1 Funktionale Parameter

In Abschnitt 5.1.7 hatten wir als Erweiterung des Begriffs einer Rechenvorschrift von Kapitel 4 in den Funktionalitäten auch Parameter $f : m$ `->` n zugelassen, d.h. Parameter eines Funktionstyps oder **funktionale Parameter**. Besitzt eine Rechenvorschrift F so einen Parameter, so sind beim Aufbau ihres Rumpfes auch Aufrufe von f erlaubt. ■

Durch die Verwendung von funktionalen Parametern in Rechenvorschriften kann man viele algorithmische Probleme elegant und methodisch sauber programmieren. Funktionale Parameter 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.

5.3.2 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 3.1 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

$$\mathit{map} : [A \to B] \times A^* \longrightarrow B^*,$$

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-Tupeln kann man *map* wie folgt definieren:

$$\mathit{map}(f, \langle a_1, \ldots, a_n \rangle) = \langle f(a_1), \ldots, f(a_n) \rangle$$

Offensichtlich gilt für die leere Liste ε nach der üblichen Konvention bei der Tupelschreibweise die Gleichung

$$\mathit{map}(f, \varepsilon) = \varepsilon. \qquad (M_1)$$

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

$$\begin{array}{rcll} \mathit{map}(f, w) & = & \langle f(a_1), \ldots, f(a_n) \rangle & \text{Definition von } \mathit{map} \\ & = & f(a_1) \,\&\, \langle f(a_2), \ldots, f(a_n) \rangle & \text{Definition von } \& \\ & = & f(a_1) \,\&\, \mathit{map}(f, \langle a_2, \ldots, a_n \rangle) & \text{Definition von } \mathit{map} \\ & = & f(\mathit{top}(w)) \,\&\, \mathit{map}(f, \mathit{rest}(w)) & \text{Definition von } \mathit{top}, \mathit{rest}, \end{array}$$

also die nachfolgende rekursive Gleichung:

$$\mathit{map}(f, w) = f(\mathit{top}(w)) \,\&\, \mathit{map}(f, \mathit{rest}(w)) \qquad (M_2)$$

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 *filter*(p, w), unter Beibehaltung der Reihenfolge, aus der Liste $w \in A^*$ genau die Elemente $a \in A$ mit $p(a) = \mathit{tt}$ 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:

$$\mathit{filter}(p,\varepsilon) = \varepsilon \tag{F_1}$$

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

$$\mathit{filter}(p, a \,\&\, w) = a \,\&\, \mathit{filter}(p,w). \tag{F_2}$$

Im verbleibenden Falle $p(a) = f\!f$ legen wir fest, daß

$$\mathit{filter}(p, a \,\&\, w) = \mathit{filter}(p,w) \tag{F_3}$$

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 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 betrachten wir den Funktionsbegriff in der Mathematik. Hier werden (partielle) Funktionen üblicherweise definiert durch die Notation (genannt „explizite" Definition)

$$f : M \longrightarrow N \qquad\qquad f(x) = \ldots$$

und damit ist es notwendig, ihnen einen Namen zu geben. Ein konkretes einfaches Beispiel ist die folgende Funktion:

$$\mathit{square} : \mathbb{R} \longrightarrow \mathbb{R} \qquad\qquad \mathit{square}(n) = n * n$$

Will man nun eine (partielle) Funktion **anonym** angeben, d.h. ohne einen Namen, so muß man sich beim derzeitigen Stand des Skriptums auf ihre Definition als eindeutige und totale (oder nur eindeutige) Relation rückbesinnen. Anonym schreibt sich *square* dann 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 andere Schreibweisen eingeführt, wie z.B. 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 die Funktionsabstraktion, einzuführen.

5.3.3 Funktionsabstraktion

Es seien $x_1, \ldots, x_k$ paarweise verschiedene Bezeichner, $m_1, \ldots, m_k$ Typen und t ein Term des Typs n. Dann ist ist auch

$$\texttt{fn (}x_1 \texttt{ : } m_1\texttt{, ..., } x_k \texttt{ : } m_k\texttt{) => } t \tag{FA}$$

ein Term, genannt **Funktionsabstraktion**, und sein Typ ist $m_1 \texttt{*} \ldots \texttt{*} m_k$ `->` 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, nach der Erweiterung 5.1.7, somit 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 der Funktionsabstraktion (*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 des Terms t unter einer Belegung σ mit $\sigma(x_i) = u_i$ im Sinne von Definition 3.3.9 zuordnet. Beispielsweise bekommen wir als 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

$$f(u_1, u_2) = I[\mathtt{m} * \mathtt{n}]^A_\sigma = I[\mathtt{m}]^A_\sigma * I[\mathtt{n}]^A_\sigma = \sigma(\mathtt{m}) * \sigma(\mathtt{n}) = u_1 * u_2$$

berechnet. In dieser Gleichungskette folgt die letzte Gleichung aus den Festlegungen $\sigma(\mathtt{m}) = u_1$ und $\sigma(\mathtt{n}) = u_2$.

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 4.3.2.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 4.3.2.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 4.4.6, die Festlegung des von F induzierten Funktionals, was den Rahmen dieses Skriptums überschreitet.

5.3.4 β-Konversion

Eine sehr gute operationelle Vorstellung vom 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, $1 \leq i \leq k$, gleichzeitig die Variable x_i textuell jeweils durch den Term e_i ersetzt. Beispielsweise haben wir

$$\texttt{(fn (m : real, n : real) => m * n)(3.0,4.0)} = \texttt{3.0 * 4.0} = \texttt{12.0}.$$

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.

5.3.5 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:

$$crossout(s,n)(k) = \begin{cases} \mathit{ff} & : \quad k = n \\ s(k) & : \quad k \neq n \end{cases}$$

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 5.5 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 4.4 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

$$curry(f)(a)(b) = f(a,b) \tag{$*$}$$

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`)`.

5.3.6 Punktweise Definition einer Rechenvorschrift

Eine punktweise Definition einer Rechenvorschrift in ML hat, mit Variablen x_i, Typen m_i $(1 \leq i \leq k)$ und einem Term t, die folgende Form:

$$\texttt{fun } F \ \texttt{(}x_1 \texttt{ : } m_1\texttt{)(}x_2 \texttt{ : } m_2\texttt{)}\ldots\texttt{(}x_k \texttt{ : } m_k\texttt{) = } t\texttt{;} \tag{PD}$$

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:

$$\texttt{fun}\ F\ \texttt{(}x_1\ \texttt{:}\ m_1\texttt{)}\ \texttt{:}\ m_2\ \texttt{->}\ldots\texttt{->}\ m_k\ \texttt{->}\ n\ \texttt{=}$$
$$\quad\texttt{fn (}x_2\ \texttt{:}\ m_2\texttt{) => (fn}\ \ldots\ \texttt{=> (fn (}x_k\ \texttt{:}\ m_k\texttt{) =>}\ t\texttt{)}\ldots\texttt{);}$$

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.

Nachfolgend geben wir einige Beispiele für punktweise Definitionen von Rechenvorschriften in der Programmiersprache ML an, sowie die entsprechenden semantisch gleichwertigen Rechenvorschriften mit Funktionsabstraktionen als Rümpfen.

5.3.7 Beispiele (für punktweise Definitionen)

a) Zuerst betrachten wir die oben angegebene Funktion *curry*. 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:

$$\texttt{fun curry (f :}\ m\texttt{*}n\texttt{->}p\texttt{)(x :}\ m\texttt{)(y :}\ n\texttt{) = f(x,y);}$$

Unter Verwendung einer Funktionsabstraktion statt der punktweisen Definition sieht diese Rechenvorschrift wie folgt aus:

$$\texttt{fun curry (f :}\ m\texttt{*}n\texttt{->}p\texttt{) :}\ m\texttt{->}n\texttt{->}p\ \texttt{=}$$
$$\quad\texttt{fn (x :}\ m\texttt{) => ((y :}\ n\texttt{) => f(x,y));}$$

b) Als ein zweites Beispiel betrachten wir die Komposition $(g \circ f)(x) = g(f(x))$ von Funktionen $f : A \longrightarrow B$ und $g : B \longrightarrow C$. Mit Typen m, n und p für die Mengen A, B und C kann man diese punktweise Definition in ML wie folgt durch eine Rechenvorschrift `comp` realisieren:

$$\texttt{fun comp (g :}\ n\texttt{->}p\texttt{, f :}\ m\texttt{->}n\texttt{)(x :}\ m\texttt{) = g(f(x));}$$

Als Entsprechung mit einer Funktionsabstraktion bekommt man:

$$\texttt{fun comp (g :}\ n\texttt{->}p\texttt{, f :}\ m\texttt{->}n\texttt{) :}\ m\texttt{->}p\ \texttt{=}$$
$$\quad\texttt{fn (x :}\ m\texttt{) => g(f(x));}$$

Man beachte, daß in der punktweise definierten Fassung von `comp` statt eines einzelnen Parameters, wie in (PD) angegeben, die Liste `(g :` n `->` p`, f :` m `->` n`)` von zwei Parametern verwendet wird. ■

5.4 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 $F(t_1, \ldots, t_k)$ 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.

5.4.1 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.

5.4.2 Beispiel (für eine parametrisch polymorphe Rechenvorschrift)

In der ML-Rechenvorschrift `map` von Abschnitt 5.3.2 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[19]. Dies kann man in ML dadurch ausdrücken,

[19]Wir haben bisher versucht, in ML-Programmen beliebig wählbare Teile dadurch hervorzuheben, daß wir eine kursive Schrift statt Schreibmaschinenschrift verwendeten.

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 Zeichenreihen, 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.

5.4.3 Parametrische Polymorphie in ML

a) Die Sprache ML erlaubt **parametrische Typen** bzw. Datenstrukturen: Bei der Typdeklaration mit Konstruktoren sind, als Erweiterung von Abschnitt 5.1.5, auch Typvariable `'a`, `'m` usw. erlaubt. Syntaktisch werden diese, wie schon in Beispiel 5.4.2 bei der Rechenvorschrift `map` verwendet, dem eigentlichen Typ vorangestellt. Damit erhalten wir beispielsweise als parametrischen Typ für die linearen Listen des Typs `'a`

```
datatype 'a sequ = empty |
                   append of 'a * 'a sequ;
```

als Verallgemeinerung von Beispiel 5.2.3.a.

b) In ML ist es möglich, **polymorphe Rechenvorschriften** zu deklarieren: Dies ist einerseits möglich, indem man, wie in obiger Rechenvorschrift `map`, als Erweiterung des bisherigen Begriffs, in den Funktionalitäten auch Typvariable `'a`, `'m` usw. oder parametrische Typen verwendet. Damit ist die Anwendung für jede Ersetzung der jeweiligen Typvariablen durch explizite Typen definiert. Polymorphie wird auch erreicht, indem man in den Funktionalitäten Typen wegläßt. Ein einfaches Beispiel ist die Rechenvorschrift

```
fun id (x) = x;
```

zur Realisierung der identischen Funktion auf einer beliebigen Typinterpretation. 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. ■

Neben den Typvariablen `'a`, `'m` usw., die für einen beliebigen ML-Typ stehen, gibt es noch **Typvariable mit Gleichheit** `''a`, `''m` usw., die für eine ML-Typ mit Gleichheitsoperation, also für eine Sorte der elementaren Datenstrukturen 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.

5.4.4 Beispiel (Typisierung einer polymorphen Rechenvorschrift)

Wir betrachten die folgende, ohne jede Typangabe punktweise definierte Rechenvorschrift `S`, die in der Literatur auch als S-Kombinator (hier in der ML-Version) bekannt ist:

```
fun S (f)(g)(x) = f(x)(g(x));
```

Eine Berechnung des allgemeinsten Typs von `S` startet mit der Beobachtung, daß sowohl `S(f)` als auch `S(f)(g)` und `f(x)` einen Funktionstyp besitzen. Also setzen wir mittels sieben Typvariablen so generell wie möglich

```
('a -> ('b -> 'c)) -> (('d -> 'e) -> ('f -> 'g))
```

als den Typ von `S` an. Dabei ist `'a->('b->'c)` der Typ von `f`, also `'b->'c` der Typ von `f(x)`. Weiterhin ist `'d->'e` der Typ von `g`, also `'e` der Typ von `g(x)`. Schließlich ist noch `'f` der Typ von `x` und `'g` der Typ des Rumpfes von `S`.

Da im Rumpf von `S` die Aufrufe `f(x)` und `g(x)` vorkommen, müssen `'a`, `'d` und `'f` übereinstimmen. Dies bringt als eine erste Konkretisierung des obigen Typs

```
('a -> ('b -> 'c)) -> (('a -> 'e) -> ('a -> 'g)).
```

Aus den Typen von `f(x)` und `g(x)` und dem Aufruf `f(x)(g(x))` folgt nun, daß `'b` und `'e` übereinstimmen müssen. Dies bringt uns wiederum eine Konkretisierung:

```
('a -> ('b -> 'c)) -> (('a -> 'b) -> ('a -> 'g))
```

Die eben betrachteten Aufrufe implizieren aber auch, daß `'c` der Typ von `f(x)(g(x))` sein muß. Da dieser Term exakt der Rumpf von `S` ist, bringt dies die Gleichheit von `'c` und `'g`, also schließlich

```
S : ('a -> ('b -> 'c)) -> (('a -> 'b) -> ('a -> 'c))
```

als den allgemeinsten Typ der Rechenvorschrift `S`. ■

In diesem Beispiel haben wir das Problem der Typherleitung exemplarisch behandelt. Für die praktische Verwendung ist es wichtig, einen Algorithmus zu haben, der jeweils allgemeinste Typen (sogenannte **Prinzipaltypen**) bestimmt. Für ML existiert so ein Algorithmus. Seine detaillierte Behandlung würde jedoch den Rahmen des Skriptums sprengen. Ein an Typisierungsfragen bei funktionalen Programmiersprachen interessierter Leser sei auf das Buch „Grundlagen der funktionalen Programmierung" von P. Thiemann hingewiesen, das im Jahre 1994 bei Teubner erschienen ist.

5.5 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, (iii) Typen mit Konstruktoren oder Systeme von Typen mit Konstruktoren. Als vierte Deklarationsart führen wir jetzt noch die Objektdeklarationen ein.

5.5.1 Objektdeklarationen

In der Programmiersprache ML sieht eine **Objektdeklaration** in ihrer allgemeinsten Form syntaktisch wie folgt aus:

$$\texttt{val } (x_1 \texttt{ : } m_1\texttt{,}\dots\texttt{,}x_k \texttt{ : } m_k) \texttt{ = } t\texttt{;} \qquad (OD)$$

Dabei sind die x_1 Bezeichner, die m_i 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 Objektdeklaration (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 **reine Abkürzung**. Insbesondere darf man im Fall $k = 1$, sofern der Wert von t definiert ist[20], überall im Restprogramm den Bezeichner x_1 statt des eventuell komplizierten Terms t schreiben.

Wegen der Auffassung von Objektdeklarationen als Abkürzungen ist insbesondere, wie beim type-binding, **keine Rekursion** möglich. Kommt einer der Bezeichner x_i der linken

[20]Dies ist eine Folge der Call-by-value-Semantik von ML.

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 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:

5.5.2 Beispiel (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 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 der Einleitung 4.1.3 zu ML mit der Rechenvorschrift `sqrt` ein Beispiel dafür angegeben, Dinge lokal zu deklarieren. Solche lokalen Deklarationen erweitern nochmals die Möglichkeiten, in ML Terme zu definieren.

5.5.3 Lokale Deklarationen

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 im Rahmen von Systemen) oder Objekten. Weiterhin sei t ein Term des Typs m. Dann ist

$$\texttt{let } D_1\texttt{;} \ldots D_k\texttt{; in } t \texttt{ end} \qquad (LD)$$

ebenfalls ein Term des Typs m. Ein Term der Form (LD) wird **lokale Deklaration** oder auch **Abschnitt** mit **Deklarationsteil** $D_1\,;\ \ldots\ D_k\,;$ und **Rumpf** t genannt. ■

Als Konvention dürfen in (LD) 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.

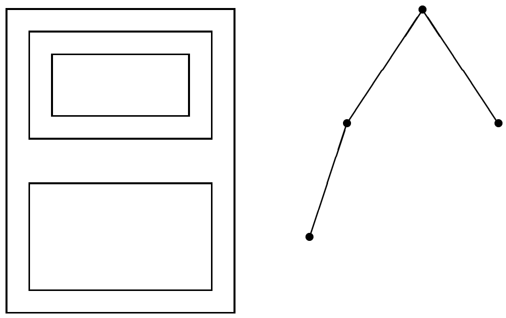
Da Abschnitte per Definition wieder Terme sind, kann man sie beliebig schachteln. Im nachfolgenden Beispiel 5.5.4 geben wir so eine Schachtelung an. Diese Beispiel dient zugleich als Motivation für Fragen, die sich im Zusammenhang mit der Semantik von Abschnitten ergeben.

5.5.4 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, die 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 globale 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 5.5.4 wie folgt darstellen:

(1) Die erste Möglichkeit wählt die Bezeichnung `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)` die Bezeichnung `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 5.5.4. Wir gehen bei der allgemeinen Klärung der Bindung / Sichtbarkeit bei Abschnitten in kleinen Schritten vor. Zuerst reduzieren wir den allgemeinen Fall auf einen Spezialfall.

5.5.5 Reduktion auf spezielle Abschnitte

In der Programmiersprache ML ist für einen allgemeinen Abschnitt

$$\texttt{let } D_1\texttt{; } \ldots D_k \texttt{ in } t \texttt{ end}$$

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.

5.5.6 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[21] 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.

5.5.7 Beispiele (für Gültigkeitsbereiche)

a) Bei den geschachtelten Abschnitten

```
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
```

[21] Man beachte, daß ein Abschnitt ein Term ist und in Abschnitt 3.3 Terme als Zeichenreihen mit einer speziellen Struktur definiert wurden.

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.

5.5.8 Semantik von Abschnitten

Es werde im Abschnitt `let` D `in` t `end` durch eine Deklaration D ein Bezeichner x eingeführt. Dann ist die **Semantik des Abschnitts** der Wert von 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 5.5.4 am Anfang dieses Abschnitts noch einmal auf.

5.5.9 Beispiel (Weiterführung von 5.5.4)

Nach der Festlegung der Semantik von Abschnitten in Abschnitt 5.5.8 und dem Gültigkeitsbereich des ersten äußeren Vorkommens von `x` ist die Semantik des Terms von Beispiel 5.5.4 gleich dem Wert des nachfolgenden Terms:

```
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
```

Dieser Term entspricht genau dem Rumpf des äußeren Abschnitts von Beispiel 5.5.4, bei dem jedes Vorkommen von `x` im Gültigkeitsbereich durch `3` ersetzt wurde.

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 zu `54` vereinfacht. ■

Die durch die Abschnitte 5.5.5 und 5.5.8 festgelegte Semantik, 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** bz. **dynamischer Sichtbarkeitsregel** spricht man hingegen, wenn von den nach dem Beispiel 5.5.4 aufgezeigten Möglichkeiten (2) gewählt wird. Wie das Beispiel 5.5.4 zeigt, müssen beide Bindungsmechanismen nicht immer das gleiche Resultat liefern. So wertet sich bei dynamischer Bindung der Abschnitt von Beispiel 5.5.4 zu `((2 + 1) + 2) * 3 * (1 + 2)`, also zu `45` aus.

Nach diesen theoretischen Untersuchungen zur Semantik von lokalen Deklarationen wollen wir im Rest dieses Abschnitts auch noch ihre praktischen Anwendungen bei der Programmierung studieren. Die Bedeutung von Abschnitten 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.

5.5.10 Beispiele (für Effizienzsteigerung durch Abschnitte)

a) In Beispiel 4.1.2 haben wir für den ganzzahligen Anteil der Quadratwurzel einer natürlichen Zahl die nachfolgende Rekursion entwickelt:

$$isqrt(n) = \begin{cases} 0 & : \; n = 0 \\ isqrt(n-1)+1 & : \; n \neq 0, (isqrt(n-1)+1)^2 \leq n \\ isqrt(n-1) & : \; \text{sonst} \end{cases}$$

Dies führte, mittels einer Objektdeklaration, in Abschnitt 4.1.3 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 positiven Arguments zieht jeder Aufruf dieser Rechenvorschrift drei weiteren rekursiven Aufrufe nach sich. Insgesamt führt die Berechnung von $isqrt(n)$ zu

$$1 + 3 + 3^2 + \ldots + 3^n = \sum_{i=0}^{n} 3^i = \frac{3^{n+1}-1}{2}$$

Aufrufen. Der Gegensatz zur programmiersprachlichen Realisierung der Funktion *isqrt* in Abschnitt 4.1.3, wo in der entsprechenden Rechenvorschrift die Mehrfachauswertung durch einen Abschnitt verhindert wird, stellt sich durch einige Zahlenbeispiele wie folgt dar:

n	obige Fassung	Fassung aus 4.1.3
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 4.1.3 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 3.4.7.a genannten 10. Hilbertschen Problems eine entscheidende Rolle spielte. Die Fibonacci-Zahlen $fib(n)$ sind rekursiv wie folgt definiert:

$$(1)\;\; fib(0) = 1 \qquad (2)\;\; fib(1) = 1 \qquad (3)\;\; fib(n+2) = fib(n+1) + fib(n)$$

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. 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

$$G(n) = \langle fib(n), fib(n+1) \rangle. \qquad (G_1)$$

Aufgrund der Gleichungen (1) und (2) gilt dann die Gleichung

$$G(0) = \langle 1, 1 \rangle. \qquad (G_2)$$

Wir wollen die Berechnung von $G(n)$ rekursiv auf die Berchnung von $G(n-1)$ zurückführen. Zu diesem Zwecke sei $\langle x, y \rangle = G(n-1)$ vorausgesetzt. Dann bekommen wir

$$\begin{array}{rcll} G(n) & = & \langle fib(n), fib(n+1) \rangle & \text{Definition von } G \\ & = & \langle fib(n), fib(n-1) + fib(n) \rangle & \text{Fibonacci-Gleichung (3)} \\ & = & \langle y, x+y \rangle & \text{Definition von } G, x, y, \end{array}$$

also die rekursive Beziehung

$$G(n) = \langle y, x+y \rangle \;\; \text{mit} \;\; \langle x, y \rangle = G(n-1). \qquad (G_3)$$

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 Berchnung von Fibonacci-Zahlen. ■

Lokale Deklarationen 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 / Funktionalität erhalten werden. Lokalen Hilfsbezeichnungen erlauben 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.

5.5.11 Beispiele (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 Sortiern 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 Liste vorliegende 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

- lo : die Liste der Elemente echt kleiner a.
- eq : die Liste der Elemente gleich a.
- up : die Liste der Elemente echt größer 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 Beispiel wird diese Vorgehensweise einem 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 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 Abschnitt 5.3.2 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 Sortierverfahrens 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 dem in Abschnitt 3.4.3 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.

5.6 Rechenvorschriften und Muster

In Abschnitt 5.2 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 2.2 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 entsprchenden 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.

5.6.1 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 5.2 vorgestellte und den Typ `unit` betreffende syntaktische Vereinfachung verwendet wird:

$$
\begin{array}{ll}
\texttt{datatype}\ m\ \texttt{=} & b_1\ \texttt{|} \\
& \ldots \\
& b_k\ \texttt{|} \\
& c_1\ \texttt{of}\ m_{11}\ \texttt{*}\ \ldots\ \texttt{*}\ m_{1s_1}\ \texttt{|} \\
& \ldots \\
& c_n\ \texttt{of}\ m_{n1}\ \texttt{*}\ \ldots\ \texttt{*}\ m_{ns_n}\texttt{;}
\end{array}
$$

So eine Deklaration ist das programmiersprachliche Gegenstück zu einer induktiven Erzeugung einer Menge im Sinne von Definition 2.2.5. 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:

$$
\begin{array}{l}
\texttt{fun}\ F\ \texttt{(}\ldots\texttt{,}b_1\ \texttt{:}\ m\texttt{,}\ldots\texttt{)}\ \texttt{:}\ p\ \texttt{=}\ e_1 \\
\quad\ldots \\
\ \texttt{|}\ F\ \texttt{(}\ldots\texttt{,}b_k\ \texttt{:}\ m\texttt{,}\ldots\texttt{)}\ \texttt{:}\ p\ \texttt{=}\ e_k \\
\ \texttt{|}\ F\ \texttt{(}\ldots\texttt{,}c_1\texttt{(}t_{11}\texttt{,}\ldots\texttt{,}t_{1s_1}\texttt{)}\ \texttt{:}\ m\texttt{,}\ldots\texttt{)}\ \texttt{:}\ p\ \texttt{=}\ e_{k+1} \\
\quad\ldots \\
\ \texttt{|}\ F\ \texttt{(}\ldots\texttt{,}c_n\texttt{(}t_{n1}\texttt{,}\ldots\texttt{,}n_{ns_n}\texttt{)}\ \texttt{:}\ m\texttt{,}\ldots\texttt{)}\ \texttt{:}\ p\ \texttt{=}\ e_{k+n}\texttt{;}
\end{array}
$$

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 $F\,\texttt{(}\ldots\texttt{,}b_i\texttt{,}\ldots\texttt{)}$ oder $F\,\texttt{(}\ldots\texttt{,}c_j\texttt{(}t_{j1}\texttt{,}\ldots\texttt{,}t_{js_j}\texttt{)}\texttt{,}\ldots\texttt{)}$ 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** $F\,\texttt{(}\ldots\texttt{,}b_i\texttt{,}\ldots\texttt{)}$ oder $F\,\texttt{(}\ldots\texttt{,}c_j\texttt{(}t_{j1}\texttt{,}\ldots\texttt{,}t_{js_j}\texttt{)}\texttt{,}\ldots\texttt{)}$.

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, abgestützt auf frühere Typdeklarationen, einige Beispiele dafür an.

5.6.2 Beispiele (Rechenvorschriften und Muster)

a) In Beispiel 5.1.6 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 5.1.6 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 =
      (a = 0.0)
  | par_x (vfline(mvf((x1,x2),(y1,y2))) : line) : bool =
      (y2 = 0.0);
```

b) Auch das Beispiel 5.1.8 von Abschnitt 5.1 können wir nun zu Ende führen. Wir haben im Prinzip nur die Nachfolgeroperation auf den durch die Typdeklaration

```
datatype res = infty of unit | intres of int;
```

beschriebenen Objekten zu realisieren und in der Rechenvorchrift `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 size(s) = 0 then infty()
                 else if lp(p,s) then intres(1)
                                 else succ(pm(rest(s),p));
```

c) Als nächstes betrachten wir Listen von ganzen Zahlen und das nachfolgende Prädikat, welches spezifiziert, daß eine Liste **aufsteigend sortiert** ist:

$$\mathit{issorted}(s) \quad :\Longleftrightarrow \quad \forall\, i \in \mathbb{N} : 1 \leq i \leq |s| - 1 \rightarrow s_i \leq s_{i+1}$$

Diese Definition impliziert für die leere Liste sofort die Eigenschaft

$$\mathit{issorted}(\varepsilon) \quad \Longleftrightarrow \quad \mathit{true}, \qquad (S_1)$$

denn in diesem Falle ist der Bereich des Allquantors leer und die Implikation somit gleichwertig zu $\mathit{false} \rightarrow s_i \leq s_{i+1}$, also wahr. Es gilt ebenso

$$\mathit{issorted}(a \,\&\, \varepsilon) \quad \Longleftrightarrow \quad \mathit{true} \qquad (S_2)$$

für jede Liste der Länge 1, denn auch hier ist der Bereich des Allquantors leer. Als dritten Fall haben wir schließlich noch die Listen der Mindestlänge 2. Für sie erhalten wir

$$\begin{array}{ll}
 & \mathit{issorted}(a \,\&\, b \,\&\, s) \\
\Longleftrightarrow & \forall\, i \in \mathbb{N} : 1 \leq i \leq |a \,\&\, b \,\&\, s| - 1 \rightarrow (a \,\&\, b \,\&\, s)_i \leq (a \,\&\, b \,\&\, s)_{i+1} \\
\Longleftrightarrow & a \leq b \wedge \forall\, i \in \mathbb{N} : 2 \leq i \leq |a \,\&\, b \,\&\, s| - 1 \rightarrow (a \,\&\, b \,\&\, s)_i \leq (a \,\&\, b \,\&\, s)_{i+1} \\
\Longleftrightarrow & a \leq b \wedge \forall\, i \in \mathbb{N} : 1 \leq i \leq |a \,\&\, b \,\&\, s| - 2 \rightarrow (a \,\&\, b \,\&\, s)_{i+1} \leq (a \,\&\, b \,\&\, s)_{i+2} \\
\Longleftrightarrow & a \leq b \wedge \forall\, i \in \mathbb{N} : 1 \leq i \leq |b \,\&\, s| - 1 \rightarrow (b \,\&\, s)_i \leq (b \,\&\, s)_{i+1} \\
\Longleftrightarrow & a \leq b \wedge \mathit{issorted}(b \,\&\, s),
\end{array}$$

was, durch das Weglassen der Zwischenglieder dieser Kette von Äquivalenzen, zu

$$\mathit{issorted}(a \,\&\, b \,\&\, s) \quad \Longleftrightarrow \quad a \leq b \wedge \mathit{issorted}(b \,\&\, s) \qquad (S_3)$$

wird. Aufbauend auf eine polymorphe rekursive Deklaration von linearen Listen analog zu Beispiel 5.2.3.a und die Eigenschaften (S_1), (S_2) und (S_3) ist somit durch

Musteranpassung sofort eine ML-Rechenvorschrift `issorted` angebbar, die testet, ob eine lineare Liste von ganzen Zahlen sortiert ist, oder nicht. Wir erhalten:

```
fun issorted (empty : int sequ) : bool =
    true
  | issorted (append(a,empty) : int sequ) : bool =
    true
  | issorted (append(a,append(b,s)) : int sequ) : bool =
    a <= b andalso issorted(append(b,s));
```

d) In Beispiel 5.2.5 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

$$\mathit{count} : \mathsf{Tree}(A) \longrightarrow \mathbb{N}$$

induktiv über den Aufbau der Bäume zu definieren: Die Induktionsanfänge sind dabei die atomaren Bäume mit

$$\mathit{count}(a) = 1 \tag{C_1}$$

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

$$\mathit{count}(\langle a, \varepsilon\rangle) = 1, \tag{C_2}$$

und im Falle $w \neq \varepsilon$, d.h. $w = b \,\&\, s$, definieren wir

$$\mathit{count}(\langle a, b \,\&\, s\rangle) = \mathit{count}(b) + \mathit{count}(\langle a, s\rangle). \tag{C_3}$$

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 Deklaration des Systems von Abschnitt 5.2.6 durch eine Typvariable `'m` sofort ergibt. 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 in Abschnitt 5.2.4 behandelten vordefinierten linearen Listen verwendet werden, da hier die Standarddarstellungen wie `[]` oder `[3,5,6]` als Abkürzungen für bestimmte Terme aufzufassen sind. In dem nachfolgenden Beispiel 5.6.3 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.

5.6.3 Beispiel (Sortieren durch Mischen)

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:

$$\texttt{merge([1,3,5,7],[1,3,5,7,9,11])} = \texttt{[1,1,3,3,5,5,7,7,9,11]}$$

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 also 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 5.5.11.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. ■

In den Beispielen 5.6.2 und 5.6.3 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.

5.6.4 Beispiel (Auswertung mittels Musteranpassung)

Wir setzen die Rechenvorschrift `issorted` von Beispiel 5.6.2.b voraus. Ihre Definition führt durch das Richten der drei Fälle zu dem Termersetzungssystem

$$\begin{array}{rcl}
\texttt{issorted(empty)} & \to & \texttt{true} \\
\texttt{issorted(append(a,empty))} & \to & \texttt{true} \\
\texttt{issorted(append(a,append(b,s)))} & \to & \texttt{a <= b andalso issorted(append(b,s))}
\end{array}$$

mit Variablenmenge $X = \{\, \mathtt{a}, \mathtt{b}, \mathtt{s} \,\}$. Als ein konkretes Beispiel für eine Berechnung betrachten wir die Auswertung

```
   issorted(append(1,append(2,append(3,empty))))
⇒ 1<=2 andalso issorted(append(2,append(3,empty)))
⇒ 1<=2 andalso 2<=3 andalso issorted(append(3,empty))
⇒ 1<=2 andalso 2<=3 andalso true
⇒ true,
```

wobei die ersten drei Schritte gemäß dem eben angegebenen Termersetzungssystem durchgeführt werden und der letzte Schritt eine Vereinfachung mittels der Eigenschaften der elementaren Datenstrukturen von ML darstellt. Auch diese Vereinfachung kann formal als Termersetzung angesehen werden. ■

Nicht sinnvoll, wenn auch in der Sprache 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 funktionalen Programmiersprachen mit Musterkonzept die Muster gesteuert angepaßt. Die nachstehende Regelung gilt ohne Ausnahme für alle diese Sprachen.

5.6.5 Regel zur 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 so anzuordnen, daß sie von oben nach unten immer allgemeiner werden.

5.6.6 Musteranpassung und Wild-cards

Beim Entwurf von Rechenvorschriften mittels Musteranpassung kommt es manchmal vor, daß gewisse Parameter in den rechten Seiten nach dem Gleichheitssymbol nimals verwendet werden. Ein Beispiel ist etwa die nachfolgende Realisierung der strikten Disjunktion, deren Korrektheit aus der Regel zur Musteranpassung von oben nach unten folgt:

```
fun or (false : bool, false : bool) : bool = false
  | or (x : bool, y : bool) : bool = true;
```

Damit man sich in solchen Fällen nicht unnötig mit Parameternamen „herumplagen“ muß, die dann doch nie sinnvoll verwendet werden, erlaubt die Sprache ML die Verwendung des Zeichens „_“ als sogenanntes **Wild-card-Symbol**, welches für einen beliebigen Parameter und sogar für Muster stehen darf. Unter Verwendung des Wild-Card-Symbols für die Parameter x und y erhält man

```
fun or (false : bool,false : bool) : bool = false
  | or (_ : bool,_ : bool) : bool = true;
```

für die obige Rechenvorschrift. ■

Wir haben bisher Muster nur in Verbindung mit der Deklaration von Rchenvorschriften 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.

5.6.7 Fallunterscheidung durch Musteranpassung

Wir betrachten wiederum eine Deklaration eines Typs m mit den Konstruktoren b_i und c_j in der schematischen Form von Abschnitt 5.6.1:

$$\begin{array}{lll}\texttt{datatype}\ m\ \texttt{=} & b_1 & \texttt{|} \\ & \dots & \\ & b_k & \texttt{|} \\ & c_1\ \texttt{of}\ m_{11}\ \texttt{*}\ \dots\ \texttt{*}\ m_{1s_1} & \texttt{|} \\ & \dots & \\ & c_n\ \texttt{of}\ m_{n1}\ \texttt{*}\ \dots\ \texttt{*}\ m_{ns_n}\texttt{;} & \end{array}$$

Es sei a ein Term des Typs m. Dann hat die allgemeine **Fallunterscheidung durch Musteranpassung** die syntaktische Form

$$\begin{array}{ll}\texttt{case}\ a\ \texttt{of} & b_1\ \texttt{=>}\ e_1 \\ \texttt{|} & \dots \\ \texttt{|} & b_k\ \texttt{=>}\ e_k \\ \texttt{|} & c_1\texttt{(}t_{11}\texttt{,}\dots\texttt{,}t_{1s_1}\texttt{)}\ \texttt{=>}\ e_{k+1} \\ & \dots \\ \texttt{|} & c_n\texttt{(}t_{n1}\texttt{,}\dots\texttt{,}t_{ns_n}\texttt{)}\ \texttt{=>}\ e_{k+n} \end{array}$$

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 definiert als der Typ der Fallunterscheidung durch Musteranpassung.

Auch dieses Sprachkonstrukt erlaubt die Verwendung des Wild-card-Symbols zu Vereinfachungszwecken.

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 $c_n(t_{n1},\ldots,t_{ns_n})$ 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

$$
\begin{array}{l}
\texttt{fun}\ F\ (\ldots,b_1 : m,\ldots) : p = e_1 \\
\quad \ldots \\
\ \mid F\ (\ldots,b_k : m,\ldots) : p = e_k \\
\ \mid F\ (\ldots,c_1(t_{11},\ldots,t_{1s_1}) : m,\ldots) : p = e_{k+1} \\
\quad \ldots \\
\ \mid F\ (\ldots,c_n(t_{n1},\ldots,n_{ns_n}) : m,\ldots) : p = e_{k+n};
\end{array}
$$

von Abschnitt 5.6.1 mit Hilfe einer Fallunterscheidung durch Musteranpassung und unter Verwendung eines Parameters x an Stelle des obigen allgemeinen Terms a gleichwertig schematisch wie folgt formulieren:

$$
\begin{array}{l}
\texttt{fun}\ F\ (\ldots,x : m,\ldots) : p = \\
\quad \texttt{case}\ x\ \texttt{of}\ b_1 \Rightarrow e_1 \\
\qquad\qquad\quad \mid \ldots \\
\qquad\qquad\quad \mid b_k \Rightarrow e_k \\
\qquad\qquad\quad \mid c_1(t_{11},\ldots,t_{1s_1}) \Rightarrow e_{k+1} \\
\qquad\qquad\quad\ \ \ldots \\
\qquad\qquad\quad \mid c_n(t_{n1},\ldots,t_{ns_n}) \Rightarrow e_{k+n};
\end{array}
$$

Diese Form ist manchmal etwas übersichtlicher als die Originalform.

Im konkreten Fall der Rechenvorschrift `issorted` aus Beispiel 5.6.2, die feststellt, ob eine lineare Liste von ganzen Zahlen sortiert ist, oder nicht, ergibt sich

```
fun issorted (w : int sequ) : bool =
  case w of empty => true
          | append(a,empty) => true
          | append(a,append(b,s)) => a <= b
                                     andalso issorted(append(b,s));
```

als Form, die eine Fallunterscheidung durch Musteranpassung verwendet. ■

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

$$\texttt{if}\ b\ \texttt{then}\ e_1\ \texttt{else}\ e_2$$

und die durch Musteranpassung

```
case b of true => e1
        | false => e2
```

definierte Fallunterscheidung mit zwei Zweigen äquivalent sind.

5.7 Ausnahmen und Ausnahmenbehandlung

Bei Rechenvorschriften, die in ML durch Musteranpassung deklariert werden, sollte man in der Regel darauf achten, daß bezüglich der entsprechenden Typdeklarationen mit Konstruktoren vollständige Fallunterscheidungen vorgenommen werden. Manchmal ist es aber durchaus sinnvoll oder aber auch gar nicht möglich, alle Fälle der zugrundeliegenden Typdeklarationen abzudecken, beispielsweise wenn partielle Funktionen wie *top* oder *rest* auf den Zeichenreihen durch entsprechende Rechenvorschriften wie

```
fun top (append(x,s) : 'm sequ) : 'm = x;
```

für *top* oder

```
fun rest (append(x,s) : 'm sequ) : 'm sequ = s;
```

für *rest* auf den linearen Listen des (schon früher behandelten) rekursiven Typs `'m sequ` realisiert werden. In diesen Fällen, die nicht alle Möglichkeiten von vorkommenden Mustern ausschöpfen, bekommt man durch das ML-System die Warnung

```
Warning: match not exhaustive
```

mitgeteilt. Damit ist der Benutzer dafür verantwortlich, daß die Rechenvorschrift nur auf passende Muster (Argumente) angewendet wird.

Eine sinnvolle vollständige Fallunterscheidung bei allen durch Musteranpassung deklarierten Rechenvorschriften ist in ML durch das nachfolgend eingeführte Konzept der Ausnahmen möglich, mit dem wir dieses Kapitel über die Vertiefung der funktionalen Programmierung beenden.

5.7.1 Ausnahmen und Ausnahmenbehandlung

Zur Behandlung von Partialitäten in Rechenvorschriften gibt es in der Sprache ML die **Ausnahmen**, in der englischsprachlichen ML-Literatur **exceptions** genannt. So eine Ausnahme ist als **Konstante** des ausgezeichneten ML-Tys `exn` beschrieben durch einen Namen e und wird, als Erweiterung der bisher zugelassenen Deklarationsarten, wie folgt eingeführt:

$$\texttt{exception}\ e; \qquad (A_1)$$

Neben den Ausnahmenkonstanten gibt es in ML noch die **parametrisierten Ausnahmen**. Diese werden syntaktisch mittels

$$\texttt{exception}\ e\ \texttt{of}\ m; \qquad (A_2)$$

deklariert, wobei e wiederum ein Name und m ein Typ ist. Durch eine Ausnahmedeklaration der Form (A_2) wird e formal als eine ausnahmenkonstruierende Operation des Typs m `->` `exn` erklärt, deren Anwendung auf einen Term t des Typs m die Ausnahme e`(`t`)` erzeugt.

Nach einer Deklaration einer Ausnahme 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

`raise` e

im Fall einer Ausnahmenkonstante wie in (A_1) deklariert und mittels

`raise` e`(`t`)`,

wobei t ein Term des Typs m ist, im Fall einer parametrisierten Ausnahme wie in (A_2) deklariert. ■

Für die partiellen Operationen der elementaren Datenstrukturen stellt ML eine Reihe von vordefinierten Ausnahmen bereit. Ein Beispiel hierfür ist `Div`, welche bei einem Aufruf der Basisoperation `div` mit Null als zweitem Argument geliefert wird. Die weiteren vordefinierten Ausnahmen findet man im (schon bei den linearen Listen erwähnten) Manual zum SML/NJ-System.

Im nächsten Beispiel demonstrieren wir die Verwendung einer Ausnahme anhand der uns schon sehr bekannten Rechenvorschrift `fac` zur Bestimmung der Fakultät einer natürlichen Zahl.

5.7.2 Beispiel (für eine Ausnahmenbehandlung

Das in diesem Skriptum verwendete ML-System SML/NJ kann die Fakultät von n nur bis zum Wert $n = 12$ berechnen. Ansonsten werden die Zahlen zu groß. Eine ML-Rechenvorschrift, in der alle möglichen Fehlerfälle bei der Berechnung der Fakultät aufgefangen werden, ergibt sich aus der Rechenvorschrift von Beispiel 4.3.2.b durch eine geeignete Ergänzung um Ausnahmen. Dies sieht dann wie folgt aus:

```
fun fac (n : int) : int =
  let exception Negativ_input;
      exception Too_large_input
  in  if n < 0
        then raise Negativ_input
        else if n = 0
               then 1
               else if n < 13
                       then n * fac(n - 1)
                       else raise Too_large_input
  end;
```

Wird die Rechenvorschrift `fac` mit einem negativen Argument oder einem Argument echt größer als 12 aufgerufen, so liefert das ML-System die Meldung

`uncaught exception`

(unbehandelte Annahme), gefolgt vom Namen der Ausnahme, die den Fehler beschreibt. Etwa erhalten wir das Resultat

`uncaught exception Negativ_input`,

falls das Argument eines Aufrufs der Rechenvorschrift negativ ist. ■

Es ist klar, daß die **lokale Deklaration von Ausnahmen ebenfalls der statischen Sichtbarkeitsregel gehorcht**, die schon bei der Behandlung von Abschnitten erklärt wurde.

Normalerweise stoppt das ML-System die Auswertung eines ML-Terms oder ML-Programmes beim Erreichen der ersten (vordefinierten oder vom Benutzer deklarierten) Ausnahme. Ausnahmen können jedoch auch durch Terme und Rechenvorschriften weitergereicht werden, so daß beim Auftreten des ersten Ausnahmefalls das Programm nicht abbricht, sondern eine andere, geeignete Aktion eingeleitet wird. Diese kann dann natürlich später wiederum auf Ausnahmen stoßen, welche dann ggf. ebenso behandelt werden können. Man spricht bei diesem Vorgehen von Ausnahmenpropagierung.

5.7.3 Ausnahmenpropagierung

Das der Ausnahmenpropagierung entsprechende Sprachkonstrukt in ML hat die nachfolgende Syntax:

$$\begin{array}{rl} e \;\texttt{handle} & a_1 \;\texttt{=>}\; t_1 \\ & \texttt{|}\; a_2 \;\texttt{=>}\; t_2 \\ \dots & \quad\dots \\ & \texttt{|}\; a_n \;\texttt{=>}\; t_n \end{array}$$

Dabei ist e ein Term, die a_i, $1 \leq i \leq n$, sind Ausnahmen und die t_i, $1 \leq i \leq n$, sind die diesen Ausnahmen zugeordneten Terme. Die Typen dieser Terme und der Typ von e müssen übereinstimmen und bilden den Typ der Ausnahmenpropagierung.

Die operationelle Semantik einer solchen Ausnahmenpropagierung ist hauptsächlich durch die zwei folgenden Regeln gegeben:

(1) Ist der Wert des Terms e keine Ausnahme, so ist der Wert der Ausnahmenpropagierung der Wert von e.

(2) Ist hingegen der Wert von e eine Ausnahme a_i der aufgeführten Liste der Ausnahmen, so ist der Wert der Ausnahmenpropagierung der Wert des zugeordneten Terms t_i.

Ist keine dieser Regeln anwendbar, so ist der Wert der Ausnahmenpropagierung entweder durch eine übergeordnete Ausnahmendeklaration, -behandlung und -propagierung definiert, auf die dann diese Regeln angewendet werden, oder undefiniert, falls es eine solche nicht gibt. ■

Dieses Transportieren von Ausnahmen durch mehrstufige Schachtelungen von Rechenvorschriften nach oben zeigt die gesamte Stärke des Ansatzes von ML. Es ist damit nämlich in kontrollierter Weise wirklich das beliebige Durchreichen von Fehlermeldungen möglich, welches in imperativen Programmiersprachen üblicherweise durch unkontrollierte Rücksprünge realisiert wird. Letztere mögen zwar effizienter sein, sind aber eine große Fehlerquelle und darüberhinaus einer mathematischen Korrektheitsanalyse in der Regel nur äußerst schwer zugänglich. Bei ML ist dies anders. Hier muß man sich bei allen Rechenvorschriften, durch derer Aufrufe eine Annahme nach oben gereicht wird, nur Definitions- und Wertebereich um entsprechende Annahmenkonstruktoren erweitert denken.

Auf weitere Einzelheiten und Beispiele, welche Ausnahmen betreffen, wollen wir an dieser Stelle aus Platzgründen nicht mehr eingehen. Einiges zu diesem Thema findet man noch im Kapitel 20 „More about Exceptions“ des in der Einleitung erwähnten Buchs „Elements of ML-Programming“ von J.D. Ullmann.

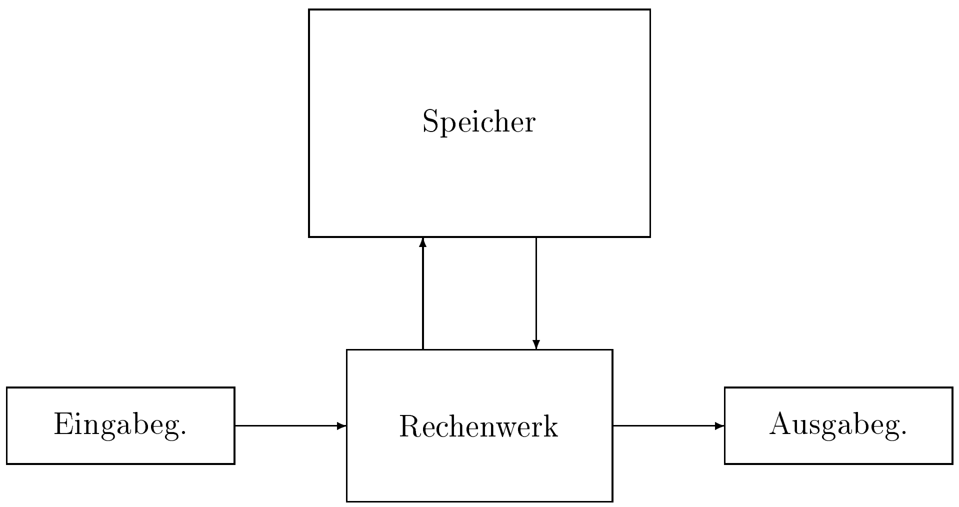
Damit beenden wir vorerst die funktionale Programmierung und wenden uns in den nächsten beiden Kapiteln prozeduraler/imperativer Programmierung zu. Später, etwa bei den Datenstrukturen, werden wir immer wieder auf die Prinzipien funktionaler Programmierung zurückkommen, da dadurch vieles einsichtiger und klarer wird. Die ML-Version des Quicksort-Algorithmus von Beispiel 5.5.10 ist ein typisches Beispiel. Es sei an dieser Stelle aber auch noch einmal ausdrücklich darauf hingewiesen, daß wir nur einen Teil von ML vorgestellt und viele wichtige Themen entweder nur knapp (wie Polymorphie) oder gar nicht (wie Datenabstraktion und das Modulkonzept) angesprochen haben. Aber auch die vorgestellten Themen wurden aus Platzgründen oft nicht bis in das letzte Detail aufgegliedert. Wer sich wirklich intensiv mit funktionaler Programmierung (z.B. anhand von ML) befassen will, kommt an weiterführender Literatur, wie den erwähnten Büchern von Ullman und Thiemann oder auch Originalliteratur, nicht vorbei.

6 Imperative Programmierung: Einführung

Wie schon in der Einleitung von Kapitel 4 erwähnt, teilen sich die Programmiersprachen auf in die imperativen und die deklarative Sprachen. Bisher haben wir uns mit der wichtigsten Teilklasse der deklarativen Sprachen beschäftigt, den funktionalen Programmiersprachen[22]. Wir behandelten dieses Thema anhand unserer Modellsprache ML. Die nächsten zwei Kapitel des Skriptums sind nun der imperativen Programmierung gewidmet, die in der Praxis derzeit der funktionalen Programmierung – hauptsächlich aus Effizienzgründen – immer noch vorgezogen wird. Dabei unterscheiden wir wiederum in Einführung (dieses Kapitel) und Vertiefung (nächstes Kapitel). Als Modellsprache für imperative Programmierung wählen wir die Programmiersprache Modula-2, eine Weiterentwicklung der bekannten Programmiersprache Pascal.

6.1 Grundlegendes

Programmiersprachen sind Sprachen 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 Konsole) mit Daten versorgt, verarbeitet diese mit Hilfe eines Speichers und übermittelt die Resultate an ein Ausgabegerät (z.B. einen Drucker). Dabei kann der Speicher sowohl Daten als auch Programme aufnehmen.

Eine simple, aber dennoch sehr brauchbare Vorstellung von einem **Speicher** ist die eines

[22]Es sollte an dieser Stelle erwähnt werden, daß, neben den funktionalen Programmiersprachen, noch weitere deklarativen Sprachklassen praktisch relevant sind. Es sei an dieser Stelle insbesondere die weitverbreitete logische Programmiersprache Prolog erwähnt.

linearen Speicherbandes mit numerierten Feldern, wie nachfolgend bildlich dargestellt:

0	1	2	3	4	5	...

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 ein **früheres Datum einfach ersetzt** werden kann. Höhere und modernere imperative Sprachen verwenden auch keine expliziten Adressen mehr, sondern symbolische Bezeichner. Ein solcher Bezeichner wird gewöhnlich **Programmvariable** genannt. Der Mechanismus zur Ersetzung eines Datums in der x zugeordneten Speicherzelle durch ein neues Datum t ist die **Zuweisung** x `:=` t.

Eine Zuweisung ist ein Spezialfall einer **Anweisung**. Dies sind die Bausteine, aus denen imperative Programme in der einfachsten Art aufgebaut sind. Sie verändern durch ihre Abarbeitung bestimmte Zellen des Speichers. Spezielle Anweisungen dienen zum Lesen und/oder Drucken von Daten. Drucken verändert dabei den Speicher nicht.

6.1.1 Beispiel (Kegelstumpf)

Wir wollen, analog zu Beispiel 4.3.2.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, welche jeweils einer Anweisung oder einer Folge von Anweisungen entsprechen. Diese sind nachfolgend angegeben:

(1) Einlesen der Daten r, R, h.

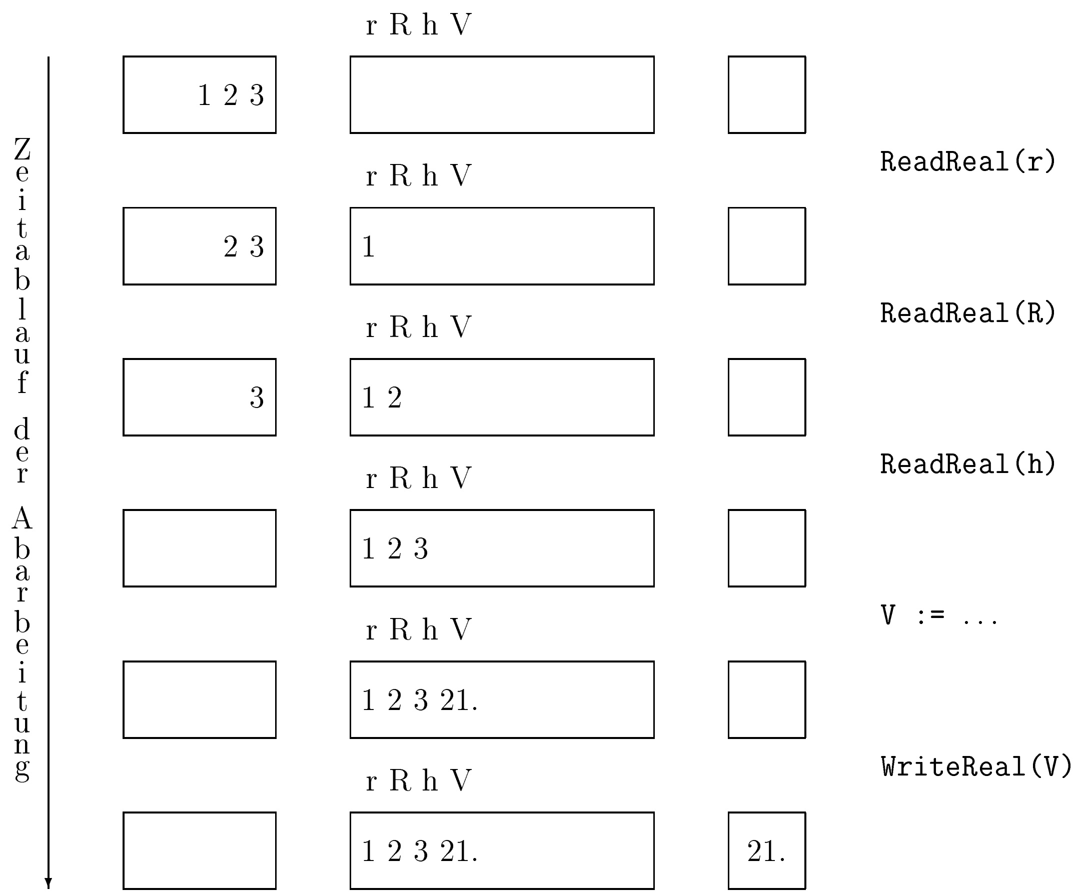
(2) Berechnung von V nach obiger Formel.

(3) Ausdrucken von V.

Wir benötigen also vier Speicherzellen, die wir durch die Programmvariablen `r`, `R`, `h` und `V` bezeichnen. Dann kann man die Schritte (1), (2) und (3) wie folgt formalisieren:

```
ReadReal(r);
ReadReal(R);
ReadReal(h);
V := 1.0472 * h * (r*r + r*R + R*R);
WriteReal(V);
```

Diese Anweisungsfolge stellt einen Teil eines Programms dar. Die ersten drei Anweisungen lesen die Daten ein und speichern sie in den mit `r`, `R` bzw. `h` bezeichneten Speicherzellen ab, die vierte Anweisung berechnet das Volumen und speichert es in der Speicherzelle mit dem Namen `V` ab und die letzte 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. statt dem genauen Wert von $21.9912 = 1.0472 * 3 * (1 + 2 + 4)$ für V geschrieben. ■

Bisher haben wir idealisierterweise angenommen, daß eine Programmvariable 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 Programmvariable ein Menge von Speicherzellen. 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 Programmvariablen x abgespeichert ist. Synonym dazu ist auch die Sprechweise „D ist der **Wert** oder der **Inhalt** von x“.

6.1.2 Die imperative Programmiersprache Modula-2

Wie schon in der Einleitung zu diesem Kapitel erwähnt, werden wir in diesem Skriptum Modula-2 als imperative Programmiersprache verwenden. Die Anweisungsfolge aus Beispiel 6.1.1 stellt einen Teil eines Modula-2-Programms dar. Vervollständigen wir sie, so erhalten wir das folgende Modula-2-Programm:

```
MODULE Cone;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadReal, WriteReal;
VAR r, R, h, V : REAL;
BEGIN
  WriteString('Gib r, R und h ein.');
  WriteLn;
  ReadReal(r);
  ReadReal(R);
  ReadReal(h);
  V := 1.0472 * h * (r*r + r*R + R*R);
  WriteReal(V,7,4);
  WriteLn;
END Cone .
```

Diesem Programm wird in der ersten Zeil der Name `Cone` gegeben. Durch die zweite und dritte Zeile werden die Anweisungen für Eingabe und Ausgabe bereitgestellt. Dann erfolgt die Erklärung der verwendeten Programmvariablen, welche jeweils die Sorte `REAL` für die reellen Zahlen $\mathbb{R}$ besitzen. Den größten Teil des Rests kennen wir bereits.

Die Geschichte von Modula-2 beginnt mit **Algol 60**, der Antwort europäischer und amerikanischer Hochschulen auf den IBM-Entwurf **Fortran**. Bei der Weiterentwicklung dieser Sprache in den 60er Jahren schieden sich die Geister. Ein IFIP-Kommittee entwickelte die theoretisch höchst interessante, aber praktisch nicht realisierbare Universalsprache **Algol 68**. Die Antwort auf diese „Monstersprache" war 1970 von N. Wirth die einfache und wahrscheinlich auch deshalb erfolgreiche Sprache **Pascal**. Trotz aller Erfolge zeigte auch Pascal Schwächen, insbesondere beim „Programmieren im Großen", d.h. dem Modularisieren von großen Programmteilen in einzelne, getrennt behandelbare Teile. Deshalb entwickelte N. Wirth Pascal weiter und kam über die Zwischenversion **Modula** zu **Modula-2**, das 1979 vorgestellt wurde.

Speziell verwenden wir in diesem Skriptum das MOCKA System der GMD („Gesellschaft für Mathematik und Datenverarbeitung") Karlsruhe. Die Sprache des Systems ist eine geringe Erweiterung des von N. Wirth publizierten Standards von Modula-2; das System selbst enthält neben Kommandos zum Übersetzen und Binden noch viele andere nützliche Kommandos und Hilfen zur Fehlersuche etc. ■

Es sollte an dieser Stelle auch erwähnt werden, daß Modula-2 keinesfalls der Endpunkt einer Entwicklung ist, sondern mittlerweile Erweiterungen bzw. Nachfolgesprachen existieren, die den modernen Entwicklungen der Informatik seit 1979 Rechnung tragen. Insbesondere wurden in die Erweiterungen und Nachfolger Konzepte zur wichtigen objektorientierten Programmierung integriert. Auf diese Dinge können wir, obwohl sie von großer Bedeutung sind, in einer Anfängervorlesung aber noch nicht eingehen.

6.2 Elementare Datenstrukturen

Wie im Fall von ML gibt es auch für Modula-2 eine Signatur $\Sigma = (S, K, F)$, die vorimplementierte Sorten, Konstantensymbole (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 durch die Bezeichnung `REAL` die Zuordnung $\mathtt{REAL}^A = \mathbb{R}$ in Abschnitt 6.1.2 gemeint.

Im folgenden geben wir die elementaren Datenstrukturen von Modula-2 (ausschnittsweise) an. Man beachte, daß, wie im Fall von ML, für jede elementare Datenstruktur **Gleichheit** `=` und **Ungleichheit** `#` zusätzlich zu den nachfolgenden Operationen existieren.

6.2.1 Wahrheitswerte

Sorte:	`BOOLEAN`	Wahrheitswerte $\mathbb{B}$
Konstanten:	`TRUE : BOOLEAN`	Symbol für *tt*
	`FALSE : BOOLEAN`	Symbol für *ff*
Operationen:	`NOT : BOOLEAN -> BOOLEAN`	Negation
	`AND : BOOLEAN,BOOLEAN -> BOOLEAN`	Konjunktion
	`OR : BOOLEAN,BOOLEAN -> BOOLEAN`	Disjunktion ■

Die Schreibweise für die Negation `NOT` ist die funktionale. Für Konjunktion `AND` und Disjunktion `OR` sieht Modula-2 eine Infix-Schreibweise vor. Man beachte, daß in Modula-2, im Gegensatz zu ML, sowohl `AND` als auch `OR` nicht von links nach rechts ausgewertet werden. In Modula-2 sind **alle Operationen der elementaren Datenstrukturen strikt**.

Im Gegensatz zur Programmiersprache ML besitzt Modula-2 eine elementare Datenstruktur für die natürlichen Zahlen.

6.2.2 Natürliche Zahlen

Sorte:	`CARDINAL`	Nat. Zahlen $\mathbb{N}$
Konstanten:	0, 1, ...	Symb. f. 0, 1 ...
Operationen:	`- : CARDINAL -> INTEGER`	Unäres Minus
	`+, -, * : CARDINAL,CARDINAL -> CARDINAL`	Arithmetik
	`DIV : CARDINAL,CARDINAL -> CARDINAL`	Ganzzahl. Div.
	`MOD : CARDINAL,CARDINAL -> CARDINAL`	Rest ganz. Div.
	`<, <=, >, >= : CARDINAL,CARDINAL -> BOOLEAN`	Vergleichsop. ■

Zu den Schreibweisen ist folgendes zu bemerken: Bis auf das einstellige Minus werden alle Operationen wie üblich in Infixnotation geschrieben. Das einstellige Minus wird in Präfixform notiert. Mit Ausnahme der binären Subtraktion, `DIV` und `MOD` sind alle Operationen total. Damit die binäre Subtraktion definiert ist, muß das erste Argument mindestens so groß wie das zweite sein. Eine Anwendung der Operationen `DIV` und `MOD` ist nur dann definiert, falls der Wert des zweiten Arguments nicht 0 ist.

6.2.3 Ganze Zahlen

Sorte:	`INTEGER`	Ganze Zahlen $\mathbb{Z}$
Konstanten:	0, 1, ...	Symbole für 0, 1 ...
Operationen:	`- : INTEGER -> INTEGER`	Unäres Minus
	`+, -, * : INTEGER,INTEGER -> INTEGER`	Arithmetik
	`DIV : INTEGER,INTEGER -> INTEGER`	Ganzzahlige Div.
	`MOD : INTEGER,INTEGER -> INTEGER`	Rest bei ganz. Div.
	`<, <=, >, >= : INTEGER,INTEGER -> BOOLEAN`	Vergleichsoperat. ■

Im Vergleich zur elementaren Datenstruktur der natürlichen Zahlen ändert sich bei der Datenstruktur der ganzen Zahlen fast nichts. Die Schreibweisen bleiben gleich und die Semantik der Operationen ist die wie im Fall der natürlichen Zahlen $\mathbb{N}$, nur auf die ganzen Zahlen $\mathbb{Z}$ ausgeweitet. Insbesondere wird die binäre Subtraktion total.

Wir haben, wie schon bei ML, in den Abschnitten 6.2.2 und 6.2.3, die Mengen $\mathbb{N}$ und $\mathbb{Z}$ idealisierenderweise als Interpretationen der Sorten `CARDINAL` und `INTEGER` angenommen. Tatsächlich handelt es sich natürlich nur um endliche Intervalle. Im Gegensatz zu ML kann man diese in Modula-2 durch entsprechende Operationen erfragen. Für MOCKA liefern diese folgende Werte:

$$\begin{aligned} \texttt{MAX(CARDINAL)} &= 4294967295 \\ \texttt{MAX(INTEGER)} &= 2147483647 \\ \texttt{MIN(INTEGER)} &= 2147483647 \end{aligned}$$

Man beachte, daß bei `MIN(INTEGER)` nicht die kleinste (negative) im System darstellbare ganze Zahl geliefert wird, sondern die positive Zahl, die aus ihr durch Betragsbildung entsteht.

6.2.4 Reelle Zahlen

Sorte:	`REAL`	Reelle Zahlen $\mathbb{R}$
Konstanten:	Alle Gleitkommadarstellungen	Beispielsw. `1`, `2.0`, `2E-3`
Operationen:	`- : REAL -> REAL`	Unäres Minus
	`+, -, *, / : REAL,REAL -> REAL`	Arithmetik
	`<, <=, >, >= : REAL,REAL -> BOOLEAN`	Vergleichsoperationen ■

Wie `2E-3` andeutet, erlaubt die Gleitkommadarstellung von Modula-2 im Gegensatz zu der von ML, auch, daß die Mantisse eine ganze Zahl (ohne Nachkommateil) ist.

Im Vergleich zur elementaren Datenstruktur der ganzen Zahlen wird bei der Datenstruktur für die reellen 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 einstellige Minus werden alle Operationen in Infix-Schreibweise notiert.

Im Vergleich zur Sprache ML fällt auf, daß die Funktionen der Analysis für $\sqrt{x}, sin(x)$ etc. nicht Teil der Signatur von Modula-2 sind. Dem liegt die Philosophie zugrunde, die Sprache klein und schlank zu halten und kontrollierte, individuelle Erweiterungen nur im Rahmen des Modulkonzeptes zuzulassen. Für MOCKA existiert ein solcher vorimplementierter Modul („eine sogenannte Bibliothek“) mit den entsprechenden Operationen. Schreibt man in einem Modula-2-Programm die Zeile

```
FROM MathLib IMPORT sqrt, exp, ln, sin, cos, arctan, real, entier;
```

so hat man die Operationen zur Verfügung für Quadratwurzel, e-Funktion, natürlichen Logarithmus, Sinus, Cosinus, Arcustangens, sowie `real : INTEGER -> REAL` zur Einbettung der ganzen in die reellen Zahlen und `entier : REAL -> INTEGER` zur Abhackung. Damit hat `real` die gleiche Wirkung wie in ML und `entier` entspricht der ML-Operation `truncate`. Siehe Abschnitt 4.2.4. An welche Stelle obige `FROM-IMPORT`-Zeile im Modula-2-Programm zu schreiben ist, wird später erklärt.

6.2.5 Zeichen

Sorte:	`CHAR`	Zeichenvorrat Modula-2	
Konstanten:	Zeichen, mit ' begrenzt	z.B. `'R'`	
Operationen:	`<, <=, >, >= : CHAR,CHAR -> BOOLEAN`	Vergleichsoperationen	■

Dabei besitzt Modula-2 etwa den gleiche Zeichenvorrat wie ML, der aus den kleinen und großen Buchstaben, Ziffern und einigen Sonderzeichen wie _, %, $ besteht. Die Definition der Vergleichsoperationen entspricht der ASCII-Zuordnung.

In Abschnitt 4.2.5 wurde mittels Teilsortenrelationen erklärt, was Sortenanpassung ist, und auch erwähnt, daß in Modula-2 implizite Sortenanpassung vorgenommen wird. Dies geschieht aber nur bei Sorten der Signatur, die wir nicht besprochen haben (z.B. `SHORTINT` und `LONGINT` als weitere Sorten für ganze Zahlen) und bei Typen, die wir noch besprechen werden (den Teilbereichstypen). **Zwischen den bisher aufgeführten Sorten `BOOLEAN`, `CARDINAL`, `INTEGER`, `REAL` und `CHAR` gibt es keine Teilsortenrelationen.** Anpassungen müssen, wie in ML, durch Operationen oder Schreibweisen explizit vorgenommen werden. Dabei sind die Transfer-Operationen teils implementierungsabhängig!

6.3 Die Sprache der while-Programme

In dem folgenden Abschnitt stellen wir die elementarsten Anweisungen der Sprache Modula-2 vor. Diese unterteilen sich in die Eingabe- und Ausgabeanweisungen für die elementaren Datentypen, die eigentlich nicht „direkt fundamental“ sind und über `FROM-IMPORT` zur Verfügung gestellt werden, und die restlichen Anweisungen, welche in der Literatur die Sprache der while-Programme genannt werden.

6.3.1 Ein- und Ausgabe elementarer Datentypen

Ist x eine Programmvariable, so ist das Eingeben eines Datums von der Konsole und das Abspeichern in der x zugeordneten Speicherzelle möglich durch die **Lese-Anweisungen**

- `Read(`x`)` falls x von der Sorte `CHAR` ist,
- `ReadCard(`x`)` falls x von der Sorte `CARDINAL` ist,
- `ReadInt(`x`)` falls x von der Sorte `INTEGER` ist,
- `ReadReal(`x`)` falls x von der Sorte `REAL` ist.

Ist t ein Term, aufgebaut über den elementaren Datenstrukturen von Modula-2, so ist das Ausdrucken des Wertes von t auf dem Bildschirm möglich durch

- `Write(`t`)` falls t von der Sorte `CHAR` ist,
- `WriteCard(`t`,`n`)` falls t von der Sorte `CARDINAL` ist,
- `WriteInt(`t`,`n`)` falls t von der Sorte `INTEGER` ist,
- `WriteReal(`t`,`n`,`k`)` falls t von der Sorte `REAL` ist.

In `WriteCard(`t`,`n`)` und `WriteInt(`t`,`n`)` ist n eine natürliche Zahl, die die zu druckenden Stellen angibt. In `WriteReal(`t`,`n`,`k`)` hat n die gleiche Bedeutung. Durch eine weitere ganze Zahl k kann man noch die Nachkommastellen und die Darstellungsart des Ausdrucks wie folgt steuern:

- $k > 0$: Übliche Darstellung mit k Nachkommastellen (mit Rundung)
- $k = 0$: Drucke als ganze Zahl (mit Rundung)
- $k < 0$: Wie $k > 0$, nur Gleitkommadarstellung.

Schließlich gibt es noch eine Drucke-Anweisung der Form

- `WriteString(`w`)`

zur Ausgabe von Zeichenreihen (vergleiche mit Abschnitt 6.1.2), etwa um Eingaben vom Programmbenutzer anzufordern oder ausgegebene Daten zum Verständnis mit Erklärungen zu versehen. In dieser Anweisung steht w für eine in Hochkommata eingeschlossene Zeichenreihe. Durch die bisherigen **Drucke-Anweisungen** werden die Werte in MOCKA nur in einen Puffer geschrieben. Um den Puffer auf dem Bildschirm sichtbar zu machen, ist die spezielle Drucke-Anweisung

- `WriteLn`

notwendig, die, nach der Leerung des Ausgabepuffers, noch für eine neue Zeile auf dem Bildschirm sorgt. Man nennt die Drucke-Anweisung `WriteLn` auch die Anweisung des Zeilenvorschubs. ■

Dem Leser ist vielleicht aufgefallen, daß es in der Programmiersprache Modula-2 keine Lese- und Drucke-Anweisungen für die elementare Datenstruktur der Wahrheitswerte gibt. Hier hilft man sich in der Regel, indem man Wahrheitswerte durch Zahlen oder einzelne Zeichen codiert einliest und beim Drucken an Stelle der Konstanten `TRUE` und `FALSE` entsprechende Zeichenreihen zur Erklärung ausgibt.

Es sollte an dieser Stelle auch noch bemerkt werden, daß das Lesen und Drucken von Daten trotz aller Bemühungen der Systemhersteller immer noch etwas implementierungsabhängig ist. So wird bei anderen Modula-2-Systemen im Gegensatz zu MOCKA-System oft der Puffer zum Bildschirm ohne die Anweisung `WriteLn` geleert.

Nun betrachten wir die grundlegendsten Anweisungen von Modula-2. Vom theoretischen Standpunkt aus gesehen, reichen diese zur Programmierung von Algorithmen völlig aus. Aus praktischen und methodischen Gründen werden wir später jedoch bei der Vertiefung der imperativen Programmierung weitere Anweisungsformen hinzunehmen.

6.3.2 Anweisungen

Die Menge der **Anweisungen** ist (beim derzeitigen Stand des Skriptums) induktiv wie folgt definiert.

a) Jede Lese-Anweisung von Abschnitt 6.3.1 ist eine Anweisung.

b) Jede Drucke-Anweisung von Abschnitt 6.3.1 ist eine Anweisung.

c) Ist x eine Programmvariable der Sorte m und t ein Term (über der Signatur von Modula-2) der Sorte m, so ist die **Zuweisung** x `:=` t eine Anweisung.

d) Die **leere Anweisung** `SKIP` ist eine Anweisung (wobei man in Modula-2 statt `SKIP` die leere Zeichenreihe schreibt).

e) Sind s_1, s_2 Anweisungen und ist b ein Term (über der Signatur von Modula-2) der Sorte `BOOLEAN`, so sind auch die **zweiseitige bewachte Anweisung**

$$\texttt{IF } b \texttt{ THEN } s_1 \texttt{ ELSE } s_2 \texttt{ END}$$

und die **einseitige bewachte Anweisung**

$$\texttt{IF } b \texttt{ THEN } s_1 \texttt{ END}$$

mit der **Bedingung** b, dem **then-Fall** s_1 und dem **else-Fall** s_2 ebenfalls Anweisungen.

f) Ist s eine Anweisung und b ein Term (über der Signatur von Modula-2) der Sorte `BOOLEAN`, so ist auch die **while-Schleife**

$$\texttt{WHILE } b \texttt{ DO } s \texttt{ END}$$

mit der **Schleifenbedingung** b und dem **Schleifenrumpf** s ebenfalls eine Anweisung.

g) Sind s_1 und s_2 Anweisungen, so ist auch deren **sequentielle Komposition** s_1 ; s_2 eine Anweisung.

Eine Anweisung, in der keine Lese- und keine Drucke-Anweisung vorkommt, heißt ein Programm der **Sprache der while-Programme**, kurz ein **while-Programm**. ■

Die informelle Bedeutung der einzelnen Anweisungen ergibt sich direkt aus den Bezeichnungen. Was die Lese- und Druckanweisungen und die Zuweisungen bewirken wurde schon erklärt. Die leere Anweisung `SKIP` bewirkt keine Speicheränderung. Bei der zweiseitigen bewachten Anweisung wird der then-Fall oder der else-Fall ausgeführt, je nachdem, ob der Wert der Bedingung gleich tt oder ff ist. Die einseitige bewachte Anweisung kann man durch die zweiseitige wie folgt definieren:

$$\texttt{IF } b \texttt{ THEN } s_1 \texttt{ END} = \texttt{IF } b \texttt{ THEN } s_1 \texttt{ ELSE SKIP END}$$

Bei einer while-Schleife wird der Schleifenrumpf solange ausgeführt, bis der Wert der Schleifenbedingung gleich ff ist. Diese informellen Erklärungen betreffen nur Abarbeitungen ohne Fehlerfälle. Fehler treten auf, wenn

- ein Term undefiniert ist (bei einer Zuweisung die rechte Seite, bei einer bedingten Anweisung bzw. while-Schleife die Bedingung und bei einer Drucke-Anweisung der zu druckende Term)
- eine Typisierung nicht paßt (bei der Zuweisung und beim Einlesen eines Datums von der Konsole)
- eine while-Schleife nicht terminiert (d.h. die Schleifenbedingung niemals den Wert tt annimmt).

Ein Modula-2-Programm der einfachen Bauart, wie wir sie in diesem Kapitel betrachten, besteht nun im Prinzip aus einer Anweisung, der die Deklaration der verwendeten Programmvariablen mit jeweiliger Sortenangabe vorangestellt ist. Nach der Sprachphilosophie von Modula-2 werden die Lese- und Drucke-Anweisungen und die mathematischen Funktionen durch Importation zur Verfügung gestellt.

Im nachfolgenden Abschnitt geben wir genau an, wie die bisher betrachteten Modula-2--Programme aufgebaut sind.

6.3.3 Einfache Modula-2-Programme

Ein imperatives Programm der Programmiersprache Modula-2, wie wir es eingeschränkterweise in diesem Kapitel betrachten, besteht aus fünf Teilen:

a) Es beginnt mit der **Kopfzeile** der Gestalt

$$\texttt{MODULE}\ P\,;$$

mit einem frei wählbaren Bezeichner P, dem Programmnamen.

b) Dann folgt der **Importteil**. Dieser hat die folgende Form:

```
FROM InOut IMPORT Read,...,WriteLn;
FROM MathLib IMPORT sqrt,...,entier;
```

Dabei müssen nur die Operationen aus `MathLib` bzw. Namen aus `InOut` in den Listen nach `IMPORT` aufgeführt sein, die im Rumpf des Programms (siehe Teil d)) verwendet werden. Gegebenenfalls kann also eine Zeile des Importteils fehlen oder sogar der gesamte Importteil. Der letzte Fall führt jedoch zu sinnlosen Programmen, die nicht mit der Umwelt zu kommunizieren in der Lage sind.

c) Auf den Importteil folgt der **Deklarationsteil**, in dem die im Rumpf des Programms verwendeten Programmvariablen zusammen mit ihren Sorten deklariert werden. Dieser hat die folgende allgemeine Form

$$\begin{array}{ll} \texttt{VAR} & x^1_1,\ldots,x^1_{k_1}\ :\ m_1\,; \\ & x^2_1,\ldots,x^2_{k_2}\ :\ m_2\,; \\ & \quad\ldots \\ & x^n_1,\ldots,x^n_{k_n}\ :\ m_n\,; \end{array}$$

mit den x^i_j als Namen der Programmvariablen der Sorten m_i, $1 \leq i \leq n$.

d) Anschließend folgt der **Rumpf** der Gestalt

$$\texttt{BEGIN}\ s\ \texttt{END}$$

mit einer Anweisung s, aufgebaut wie in Abschnitt 6.3.2 erklärt.

e) Am Ende des Programms steht noch einmal der Programmname P, gefolgt von einem Punkt. ■

Bevor wir konkrete Beispiele für Modula-2-Programme angeben, soll noch das Verhältnis von repetitiver Rekursion - wie wir es von funktionaler Programmierung kennen - und der while-Schleife geklärt werden, da dies der Hintergrund für viele imperative Programme ist.

6.3.4 Repetitive Rekursion und 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 folgt durch eine repetitive Rekursion beschrieben ist:

$$f(x) = \begin{cases} f(k(x)) & : \ b(x) = tt \\ l(x) & : \ b(x) = \mathit{ff} \end{cases}$$

Nun sei $a \in M$ ein beliebiges Element. Wenn wir die repetitive Rekursion termmäßig durch $f(a) = f(k(a)) = f(k^2(a)) = \ldots$ „abwickeln“, so erhalten wir offensichtlich für $f(a)$ die Gleichung

$$f(a) = \begin{cases} l(k^n(a)) & : \ n = min\{i : b(k^i(a)) = \mathit{ff}\} \\ \text{undef} & : \ \forall\, i : b(k^i(a)) = tt. \end{cases}$$

Wenn wir also weiterhin annehmen, daß die (partiellen) Funktionen f, b, k und l als Operationen der Modula-2-Signatur vorliegen und die Mengen M und N durch Sorten m und n interpretiert werden, so ist, mit Programmvariablen x und y der Sorten m bzw. n, die Zuweisung

$$y \ := \ f(a) \tag{1}$$

mit einem Term a der Sorte m gleichwertig zum folgenden Programmstück, bestehend aus einer Initialisierung, einer while-Schleife und einer „Nachberechnung“:

```
x := a;
WHILE b(x) DO
  x := k(x)                    (2)
  END;
y := l(x)
```

Auf diese Weise kann man sich durch den Schritt von (1) nach (2) von der Annahme befreien, daß f in der Signatur vorkommt, d.h. vorimplementiert ist. Eine Wiederholung der Schritte erlaubt schließlich oft, ein Modula-2-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 Programmvariablen vorausgesetzt, als Erweiterung der obigen Regel „(1) $\Rightarrow$ (2)“ die Gleichwertigkeit der Zuweisung (mit Termen a_i, $1 \leq i \leq n$)

$$y \ := \ f(a_1, \ldots, a_n) \tag{3}$$

mit dem folgenden Programmstück, in dem das Sprachkonstrukt `(...) := (...)`, die sogenannte **kollaterale Zuweisung**, den Programmvariablen links des Zuweisungssymbols `:=` **gleichzeitig** die Werte der rechten Terme zuweist:

$$
\begin{array}{l}
\texttt{(}x_1\texttt{,...,}x_n\texttt{) := (}a_1\texttt{,...,}a_n\texttt{);}\\
\texttt{WHILE b(}x_1\texttt{,...,}x_n\texttt{) DO}\\
\quad \texttt{(}x_1\texttt{,...,}x_n\texttt{) := (}k_1\texttt{(}x_1\texttt{,...,}x_n\texttt{),...,}k_n\texttt{(}x_1\texttt{,...,}x_n\texttt{))}\\
\quad \texttt{END;}\\
y \texttt{ := } l\texttt{(}x_1\texttt{,...,}x_n\texttt{)}
\end{array}
\tag{4}
$$

Die kollaterale Zuweisung existiert in Modula-2 nicht, man muß also auch diese **Pseudo-Anweisung** mittels Sequentialisierung durch Modula-2-Zuweisungen (ev. mit Hilfsvariablen) realisieren. ■

Im folgenden zeigen wir anhand von zwei Beispielen, wie man mit Hilfe der Techniken von Abschnitt 6.3.4 ein Pseudoprogramm durch Entfernung aller Pseudooperationen und der kollateralen Zuweisungen in ein Modula-2-Programm überführt.

6.3.5 Beispiele (für Programmentwicklung)

a) Wir betrachten zuerst die Zuweisung

```
y := fac(a)
```

mit einer Programmvariablen `y` der Sorte `CARDINAL` und einer Pseudooperation `fac` zur Berechnung der Fakultät einer natürlichen Zahl. Aus Beispiel 4.3.5 wissen wir, daß die Gleichung `fac(a)` = `facr(a,1)` gilt, mit `facr` definiert durch die Repetition

$$\texttt{facr(n,res)} = \begin{cases} \texttt{facr(n-1,n*res)} & : \ \texttt{n} \neq \texttt{0} \\ \texttt{res} & : \ \texttt{n} = \texttt{0}. \end{cases}$$

Wir können also von der obigen Zuweisung `y := fac(a)` übergehen zur Zuweisung

```
y := facr(a,1)
```

mit einer Pseudooperation `facr` und mit Hilfe der zweiten Regel „(3) ⇒ (4)“ von Abschnitt 6.3.4 ist diese gleichwertig zu dem folgenden Pseudo-Programmstück:

```
(n,res) := (a,1);
WHILE n # 0 DO
  (n,res) := (n-1,n*res)
  END;
y := res
```

Es bleiben noch die zwei kollateralen Zuweisungen, die es zu sequentialisieren gibt. 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 Programmstück

```
res := 1;
n := a;
WHILE n # 0 DO
  res := n*res;
  n := n-1
  END;
y := res.
```

Dessen Vervollständigung durch die Kopfzeile, den Import- bzw. Deklarationsteil sowie Eingabe- und Ausgabeanweisungen liefert schließlich das nachstehende Modula-2-Programm zur Berechnung der Fakultät einer natürlichen Zahl:

```
MODULE Fac;
FROM InOut IMPORT WriteLn, ReadCard, WriteCard;
VAR a, y, n, res : CARDINAL;
BEGIN
  ReadCard(a);
  res := 1;
  n := a;
  WHILE n # 0 DO
    res := n*res;
    n := n-1
    END;
  y := res;
  WriteCard(y,10);
  WriteLn
END Fac.
```

b) Für dieses Beispiel setzen wir eine Pseudooperation `isperfect` voraus, die testet, ob eine natürliche Zahl Zahl perfekt ist, oder nicht. Wir betrachten die Zuweisung

```
res := isperfect(a)
```

mit einer Programmvariablen `res` der Sorte `BOOLEAN`. Nun verwenden wir die Ergebnisse von Beispiel 4.1.1 mit einer Hilfsoperation `H`, einer weiteren Pseudooperation, um die zweifache Fallunterscheidung von Beispiel 4.1.1 auf eine einfache zurückzuführen, und transformieren die obige Zuweisung in

```
res := isp(a,a,a)
```

mit der Pseudooperation `isp` definiert durch die repetitiven Rekursion

$$\mathtt{isp(n,x,y)} = \begin{cases} \mathtt{isp(n,x\text{-}1,H(n,x\text{-}1,y))} & : \mathtt{x} \neq \mathtt{0} \\ \mathtt{y} = \mathtt{0} & : \mathtt{x} = \mathtt{0} \end{cases}$$

und der Hilfsoperation H definiert durch die Fallunterscheidung

$$\mathtt{H(n,x,y)} = \begin{cases} \mathtt{y-x} & : \ \mathtt{x} \text{ teilt } \mathtt{n} \\ \mathtt{y} & : \ \mathtt{x} \text{ teilt nicht } \mathtt{n}\,. \end{cases}$$

Damit können wir wiederum den zweiten der in Abschnitt 6.3.4 vorgestellten Übergänge verwenden und bekommen als gleichwertig zu `res := isp(a,a,a)`, also auch zur originalen Zuweisung, das Programmstück

```
(n,x,y) := (a,a,a);
WHILE x # 0 DO
  (n,x,y) := (n,x-1,H(n,x-1,y))
  END;
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 DO
  x := x-1;
  y := H(n,x,y);
  n := n
  END;
res := (y=0)
```

Es verbleibt noch die Aufgabe, in der Zuweisung `y := H(n,x,y)` die Pseudooperation H durch Operationen der Modula-2-Signatur zu realisieren. Dazu beachten wir, daß n von x dann und nur dann geteilt wird, wenn $\mathtt{x} = 0$ impliziert $\mathtt{n} = 0$ und aus $\mathtt{x} \neq 0$ folgt, daß der Rest der ganzzahligen Division von n durch x gleich Null ist. Dies führt mit Hilfe der Operation MOD zur Darstellung

$$\mathtt{H(n,x,y)} = \begin{cases} \mathtt{y-x} & : \ \mathtt{x} \neq 0 \ \wedge \ \mathtt{n\,MOD\,x} = 0 \\ \mathtt{y} & : \ \text{sonst}\,, \end{cases}$$

wobei die Konjunktion sequentiell aufzufassen ist. Der Beweis dieser Gleichung erfolgt durch einige Fallunterscheidungen; wir wollen aber darauf nicht näher eingehen. Aufgrund der obigen Gleichung kann man die Zuweisung `y := H(n,x,y)` realisieren durch

```
IF x # 0 THEN IF n MOD x = 0 THEN y := y-x END END.
```

Setzen wir diese geschachtelten einseitigen bewachten Anweisung in das obige Programmstück ein und streichen wir noch die zur leeren Anweisung SKIP offensichtlich gleichwertige Zuweisung `n := n`, so erhalten wir daraus schließlich durch eine Vervollständigung um die Kopfzeile, den Import- bzw. Deklarationsteil, sowie Eingabe- und Ausgabeanweisungen das nachstehende Modula-2-Programm:

```
MODULE Perfect;
FROM InOut IMPORT WriteLn, WriteString, ReadCard;
VAR a, n, x, y : CARDINAL;
    res : BOOLEAN;
BEGIN
  ReadCard(a);
  x := a;
  y := a;
  n := a;
  WHILE x # 0 DO
    x := x - 1;
    IF x # 0 THEN IF n MOD x = 0 THEN y := y - x END END
    END;
  res := (y = 0);
  IF res THEN WriteString('Perfekte Zahl')
         ELSE WriteString('Nicht perfekte Zahl') END;
  WriteLn
END Perfect .
```

Da in Modula-2 Wahrheitswerte nicht gedruckt werden können, gibt dieses Programm entsprechende Zeichenreihen als Meldungen aus. ■

Es ist klar, daß man in den beiden entwickelten Programmen noch Programmvariable einsparen kann. So kann man im Programm `Fac` etwa `a` sparen, wenn man mit der Zuweisung `y := fac(y)` statt `y := fac(a)` die Programmherleitung beginnt. Auch die Programmvariable `res` kann beispielsweise im Programm `Perfect` eingespart werden, wenn man die entsprechenden Zeichenreihen 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 Regeln von Abschnitt 6.3.4 klarer wird.

6.4 Betrachtungen zu Semantik und Verifikation

In Abschnitt 4.4 hatten wir bei den semantischen Betrachtungen zu ML zwei Arten zur Definition der Semantik kennengelernt, die operationelle von Abschnitt 4.4.2 und die denotationelle von Abschnitt 4.4.6. Auch für imperative Sprachen sind beide Arten von Semantikfestlegung möglich. Wir besprechen im folgenden neben der operationellen noch eine weitere 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 zu funktionaler Programmierung -

wie man Semantik bei Programmentwicklung einsetzen kann. Der Einfachheit halber beschränken wir uns im weiteren bei den semantischen Betrachtungen auf die Sprache der while-Programme mit den Anweisungsarten c) bis g) aus Abschnitt 6.3.2.

Wir beginnen mit der operationellen Semantik. Dazu benötigen wir einige Begriffe, die wir nachfolgend zusammenstellen.

6.4.1 Speicherzustände und Konfigurationen

Es bezeichne X die Menge der verwendeten Programmvariablen, welche bei einem while-Programm als einem Teil eines Modula-2-Programms normalerweise übereinstimmt mit den im Deklarationsteil eingeführten Programmvariablen. Weiterhin sei $V := \bigcup_{m \in S} m^A$, mit m^A als Interpretation der Modula-2-Sorte $m \in S$, die Menge der sogenannten **semantischen Objekte**. Einen **Speicherzustand** kann man mathematisch modellieren als eine partielle Funktion

$$\sigma : X \longrightarrow V,$$

die einer Programmvariablen x der Sorte $m \in S$ das unter x abgespeicherte Datum $\sigma(x)$ aus der Menge m^A zuordnet, oder für x undefiniert ist, falls die durch x bezeichnete Speicherzelle keinen Inhalt besitzt. Ist σ ein Speicherzustand, x eine Programmvariable der Sorte $m \in S$ und $u \in m^A$ ein semantisches Objekt, so definieren wir die **Abänderung** $\sigma[x \leftarrow u]$ von σ an der Stelle x zu u als Speicherzustand

$$\sigma[x \leftarrow u](y) = \begin{cases} \sigma(y) & : \quad y \neq x \\ u & : \quad y = x. \end{cases}$$

Mit dieser Festlegung gilt offensichtlich die Gleichheit $\sigma[x \leftarrow u][x \leftarrow v] = \sigma[x \leftarrow v]$.

Da Speicherzustände nichts anderes als (partielle) Belegungen im Sinne von Definition 3.3.8 sind, überträgt sich auch die Definition 3.3.9 des Wertes eines Terms von Belegungen auf Speicherzustände. Im folgenden bezeichne $I[t]_\sigma$ den Wert des Modula-2-Terms t zum Speicherzustand σ. Im Vergleich zu Definition 3.3.9 haben wir die Algebra A unterdrückt, da sie für Modula-2 einfürallemal gewählt (also kein Parameter bei der Wertbestimmung) ist.

Neben den Speicherzuständen benötigen wir zur Definition der operationellen Semantik noch **Konfigurationen**. Eine Konfiguration ist entweder ein Speicherzustand oder ein Paar $\langle s, \sigma \rangle$ mit einem while-Programm s und einem Speicherzustand σ. ■

Nach diesen Vorbereitungen können wir nun die operationelle Semantik von while-Programmen formal definieren. Die Analogie der nachfolgenden Festlegung zur Text- oder Termersetzung ist offensichtlich. Sie wird in der Literatur auch als strukturelle operationelle Semantik bezeichnet, kurz SOS, um sie von anderen Ansätzen zur operationellen Semantik[23] zu unterscheiden.

[23]Ein weiterer Ansatz ist etwa die sogenannte natürliche operationelle Semantik. Diese legt nicht die individuellen Einzelschritte fest, sondern, wie man von einer Konfiguration $\langle s, \sigma \rangle$ zu einem Speicherzustand kommt.

6.4.2 Operationelle Semantik von while-Programmen

Die operationelle Semantik der while-Programme ist definiert mittels einer Einschritt-Ersetzungsrelation $\Rightarrow$ auf Konfigurationen, welche die individuellen Berechnungsschritte wie folgt festlegt:

a) Zuweisung:

$$\frac{I[t]_\sigma \text{ ist definiert}}{\langle x \ \texttt{:=}\ t, \sigma\rangle \Rightarrow \sigma[x \leftarrow I[t]_\sigma]}$$

b) Leere Anweisung:

$$\langle \texttt{SKIP}, \sigma\rangle \Rightarrow \sigma$$

c) Zweiseitige bewachte Anweisung:

$$\frac{I[b]_\sigma = tt}{\langle \texttt{IF}\ b\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2\ \texttt{END}, \sigma\rangle \Rightarrow \langle s_1, \sigma\rangle}$$

$$\frac{I[b]_\sigma = \mathit{ff}}{\langle \texttt{IF}\ b\ \texttt{THEN}\ s_1\ \texttt{ELSE}\ s_2\ \texttt{END}, \sigma\rangle \Rightarrow \langle s_2, \sigma\rangle}$$

d) Einseitige bewachte Anweisung:

$$\frac{I[b]_\sigma = tt}{\langle \texttt{IF}\ b\ \texttt{THEN}\ s\ \texttt{END}, \sigma\rangle \Rightarrow \langle s, \sigma\rangle} \qquad \frac{I[b]_\sigma = \mathit{ff}}{\langle \texttt{IF}\ b\ \texttt{THEN}\ s\ \texttt{END}, \sigma\rangle \Rightarrow \sigma}$$

e) While-Schleife:

$$\frac{I[b]_\sigma = tt}{\langle \texttt{WHILE}\ b\ \texttt{DO}\ s\ \texttt{END}, \sigma\rangle \Rightarrow \langle s;\ \texttt{WHILE}\ b\ \texttt{DO}\ s\ \texttt{END}, \sigma\rangle}$$

$$\frac{I[b]_\sigma = \mathit{ff}}{\langle \texttt{WHILE}\ b\ \texttt{DO}\ s\ \texttt{END}, \sigma\rangle \Rightarrow \sigma}$$

f) Sequentielle Komposition:

$$\frac{\langle s_1, \sigma_1\rangle \Rightarrow \langle s_1', \sigma_2\rangle}{\langle s_1\,; s_2, \sigma_1\rangle \Rightarrow \langle s_1'\,; s_2, \sigma_2\rangle} \qquad \frac{\langle s_1, \sigma_1\rangle \Rightarrow \sigma_2}{\langle s_1\,; s_2, \sigma_1\rangle \Rightarrow \langle s_2, \sigma_2\rangle}$$

Wir sagen, daß das while-Programm s den Speicherzustand σ_1 in den Speicherzustand σ_2 überführt, falls die Beziehung $\langle s, \sigma_1\rangle \overset{*}{\Rightarrow} \sigma_2$ gilt, mit $\overset{*}{\Rightarrow}$ als der reflexiv-transitiven Hülle der Einschritt-Ersetzungsrelation. ■

Die eben angegebenen Regeln sind eigentlich intuitiv klar. Erwähnenswert sind vielleicht die Regeln zur sequentiellen Komposition. Die linke Regel impliziert insbesondere, daß die Abarbeitung von s_1 beendet sein muß, bevor wir s_2 ausführen können. Die rechte Regel besagt dann, daß die Abarbeitung von s_2 mit dem Speicherzustand beginnt, den s_1 aus dem originalen erzeugt hat.

6.4.3 Beispiel (für operationelle Semantik)

Wir betrachten drei Programmvariable `x`, `y` und `z` der Sorte `CARDINAL`. Weiterhin sei der Speicherzustand $\sigma = (\mathtt{x} \mapsto 5, \mathtt{y} \mapsto 7)$ vorgegeben, wobei die Schreibweise für die partielle Funktion σ selbsterklärend ist. Dann bekommen wir beispielsweise, nach der Regel a) und der rechten Regel f) von Abschnitt 6.4.2, daß

$$\begin{aligned}\langle \mathtt{z\ :=\ x;\, x\ :=\ y;\, y\ :=\ z}\,, \sigma\rangle &\Rightarrow \langle \mathtt{x\ :=\ y;\, y\ :=\ z}\,, \sigma[\mathtt{z} \leftarrow 5]\rangle \\ &\Rightarrow \langle \mathtt{y\ :=\ z}\,, \sigma[\mathtt{z} \leftarrow 5][\mathtt{x} \leftarrow 7]\rangle \\ &\Rightarrow \sigma[\mathtt{z} \leftarrow 5][\mathtt{x} \leftarrow 7][\mathtt{y} \leftarrow 5],\end{aligned}$$

da im letzten Schritt gilt $I[\mathbf{z}]_{\sigma[\mathtt{z}\leftarrow 5][\mathtt{x}\leftarrow 7]} = \sigma[\mathtt{z} \leftarrow 5][\mathtt{x} \leftarrow 7](\mathbf{z}) = 5$. Startet man die Berechnung des selben Programs hingegen im Speicherzustand $\sigma' = (\mathtt{x} \mapsto 5, \mathtt{z} \mapsto 7)$, so bekommt man

$$\langle \mathtt{z\ :=\ x;\, x\ :=\ y;\, y\ :=\ z}\,, \sigma'\rangle \Rightarrow \langle \mathtt{x\ :=\ y;\, y\ :=\ z}\,, \sigma'[\mathtt{z} \leftarrow 5]\rangle.$$

Da der Wert von `y` bezüglich des Speichers $\sigma'[\mathtt{z} \leftarrow 5]$ nicht definiert ist, kann keine weitere Regel angewendet und somit auch kein Speicherzustand erreicht werden. ■

Nach diesem kleinen Beispiel wenden wir uns nun der zweiten Art von Semantik zu. Im Gegensatz zur operationellen Semantik von Abschnitt 6.4.2 orientiert sich diese nicht an der konkreten Abarbeitung einer Anweisung s, sonderen charakterisiert die durch s bewirkte Speicheränderung durch ein Paar (φ, ψ) von Aussagen, wobei φ den Zustand vor und ψ den Zustand nach der Ausführung von s beschreibt. Formal werden Aussagen durch Formeln ausgedrückt.

6.4.4 Formeln

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 Modula-2-Signatur von Abschnitt 6.2 enthält, und einer (auch nicht näher spezifizierten Menge) von Variablen, die mindestens alle Programmvariablen enthält.

Ausgehend von den Termen der Sorte `BOOLEAN`, werden Formeln aufgebaut mit Hilfe der bekannten logischen Symbole (Junktoren) $\neg, \wedge, \vee, \rightarrow, \leftrightarrow$ und der beiden Quantoren $\forall$ und $\exists$. D.h. 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 auch $(\neg\varphi)$.

(3) Sind φ und ψ Formeln, so auch $(\varphi \wedge \psi), (\varphi \vee \psi), (\varphi \rightarrow \psi)$ und $(\varphi \leftrightarrow \psi)$.

(4) Ist x eine Variable, **aber keine Programmvariable**, und φ eine Formel, so sind auch $(\forall x : \varphi)$ und $(\exists x : \varphi)$ Formeln.

Durch diese induktive Definition 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. ■

Die Unterscheidung in die Programmvariablen (das sind die Variablen, die in einem while-Programm als symbolische Bezeichnungen für eine Speicherzelle auftreten) und die restlichen Variablen, genannt **logische Variablen**, ist ganz natürlich.

6.4.5 Beispiel (Programmvariable vs. logische Variable)

Wir betrachten das folgende while-Programm s, wobei `x` eine Programmvariable der Sorte `CARDINAL` ist:

$$\mathtt{x} := \mathtt{x} + \mathtt{x}$$

Dann könnte eine mögliche Aussage sein: Nach Ausführung von s besitzt `x` einen geradzahligen Wert. Geradzahligkeit von `x` schreibt sich beispielsweise als Formel

$$\exists\, n : \mathtt{x} = 2 * n,$$

wobei n eine logische Variable und damit $\mathtt{x} = 2 * n$ ein Term der Sorte `BOOLEAN` der erweiterten Signatur ist. Es ist klar, daß n damit keine Programmvariable sein darf, deren Wert ja im Laufe der Programmausführung variieren kann, sondern eine Variable im übliche Sinn der Logik, also eine logische Variable, sein muß. ■

Für das weitere benötigen wir einen **Gültigkeitsbegriff** für Formeln (bezüglich eines vorgegebenen Speicherzustands). 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 Bedeutungen zu transformieren. Weiterhin benötigen wir einen **Terminierungsbegriff** für while-Programme s, ebenfalls bezüglich eines vorgegebenen Speicherzustands σ_1. Letzterer besagt, daß es einen Speicherzustand σ_2 gibt, so daß mittels der operationellen Semantik von Abschnitt 6.4.2 die Relation $\langle s, \sigma_1 \rangle \overset{*}{\Rightarrow} \sigma_2$ etabliert werden kann. Die folgenden Begriffe sind wesentlich für das weitere Vorgehen.

6.4.6 Korrektheitsbegriffe für imperative Programme

Es sei s ein while-Programm. Weiterhin seien φ und ψ Formeln, genannt **Vor-** und **Nachbedingung**.

a) Die Anweisung s ist **partiell-korrekt** bezüglich φ und ψ, falls aus der Gültigkeit von φ vor der Ausführung von s und aus der Terminierung von s folgt, daß ψ nach der Ausführung von s gilt.

b) Die Anweisung s ist **total-korrekt** bezüglich φ und ψ, falls aus der Gültigkeit von φ vor der Ausführung von s die Terminierung von s folgt, sowie die Gültigkeit von ψ 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. ■

Obwohl die Begriffe in Abschnitt 6.4.6 doch recht informell eingeführt wurden, kann man mit Hilfe logischer Umformungen den folgenden, sehr wichtigen Sachverhalt zeigen:

6.4.7 Satz

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 die Terminierung von s aus der Gültigkeit von φ folgt.

Beweis: Wir betrachten die nachstehenden drei Abkürzungen:

V	:	Vor der Ausführung von s gilt φ.
N	:	Nach der Ausführung von s gilt ψ.
T	:	Das while-Programm s terminiert.

Dann schreibt sich die Behauptung des Satzes als Äquivalenz wie folgt:

$$V \Longrightarrow (T \wedge N) \quad \Longleftrightarrow \quad ((V \wedge T) \Longrightarrow N) \wedge (V \Longrightarrow T) \qquad (*)$$

Nun führen wir zum Beweis von $(*)$ 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 6.4.7 bietet es sich an, statt der totalen Korrektheit die partielle Korrektheit und die Terminierung (bei einer gegebenen Vorbedingung) zu formalisieren. Dies ist beispielsweise mit Hilfe von denotationeller Semantik möglich. Für die praktische Programmentwicklung und -verifikation aber wesentlich besser geeignet ist das folgende Vorgehen:

(1) Man behandelt **Terminierung von while-Programmen informell**, indem man fordert, daß Schleifen nur endlich oft durchlaufen werden[24] und die Anwendungen der Operationen der Signatur unter den Bedingungen der Schleifen bzw. bewachten Anweisungen definiert sind.

(2) Man definiert **partielle Korrektheit formal**, indem man ein Regelsystem zur Herleitung von Ausdrücken der Art $\{\varphi\}\, s\, \{\psi\}$ angibt. Die Angabe eines solchen Systems entspricht der axiomatischen Definition der Semantik von while-Programmen bei partieller Korrektheit als Korrektheitsbegriff.

Das Regelsystem, welches wir im folgenden für (2) angeben werden, geht auf Untersuchungen von R. Floyd und insbesondere von C.A.R. Hoare aus dem Jahr 1969 zurück und wird

[24] Dazu sind wie in ML noethersch geordnete Mengen hilfreich

in der Literatur **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 ist eine Formel wie in Abschnitt 6.4.4 definiert (man sagt: eine **prädikatenlogische Formel** oder **Zusicherung**) oder ein Ausdruck der Gestalt $\{\varphi\}\, s\, \{\psi\}$ mit Vorbedingung φ, while-Programm s und Nachbedingung ψ (man sagt: 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**.

6.4.8 Der 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 **gültig** sind, ist die folgende Regel ein Axiom:

$$\frac{}{\varphi}$$

Für alle Formeln φ ist die folgende Regel ein Axiom:

$$\frac{}{\{\varphi\} \ \texttt{SKIP} \ \{\varphi\}}$$

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 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

$$\frac{\{\varphi \wedge b\}\, s_1\, \{\psi\} \quad \{\varphi \wedge \texttt{NOT}(b)\}\, s_2\, \{\psi\}}{\{\varphi\} \ \texttt{IF} \ b \ \texttt{THEN} \ s_1 \ \texttt{ELSE} \ s_2 \ \texttt{END} \ \{\psi\}}$$

für die zweiseitige bewachte Anweisung, die Regel

$$\frac{\{\varphi \wedge b\}\, s\, \{\psi\} \quad \varphi \wedge \texttt{NOT}(b) \rightarrow \psi}{\{\varphi\} \ \texttt{IF} \ b \ \texttt{THEN} \ s \ \texttt{END} \ \{\psi\}}$$

für die einseitige bewachte Anweisung, die Regel

$$\frac{\{I \wedge b\}\, s\, \{I\}}{\{I\} \ \texttt{WHILE} \ b \ \texttt{DO} \ s \ \texttt{END} \ \{I \wedge \texttt{NOT}(b)\}}$$

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) **Sprachunabhängigen Nicht-Axiome (Konsequenzregeln genannt):** Deren gibt es zwei, nämlich die Regel

$$\frac{\varphi \to \psi \qquad \{\psi\}\, s\, \{\xi\}}{\{\varphi\}\, s\, \{\xi\}}$$

zur Abschwächung der Vorbedingung φ der Unterformel zur Vorbedingung ψ der Oberformel und die Regel

$$\frac{\{\varphi\}\, s\, \{\psi\} \qquad \psi \to \xi}{\{\varphi\}\, s\, \{\xi\}}$$

zur Verstärkung der Nachbedingung ξ der Unterformel zur Nachbedingung ψ der Oberformel. Dabei sind die in den Regeln auftauchenden Formeln und Anweisungen wiederum beliebig. ■

Wie die Regeln der operationellen Semantik von Abschnitt 6.4.2, so sind auch die eben aufgeführten Regeln intuitiv sofort einsichtig. In Worten besagt etwa die Regel für die leere Anweisung, daß diese den Speicher nicht verändert. Die erste Konsequenzregel liest sich wie folgt: Impliziert φ die Formel ψ und folgt 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 Rumpfes einer Schleife die Formel I invariant (d.h. erhält ihre Gültigkeit), so gilt I nach Ausführung der Schleife (und natürlich auch die Negation der Schleifenbedingung), falls I vor 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.

6.4.9 Herleitungen im Hoare-Kalkül

Die Menge der **Herleitungen** im Hoare-Kalkül mit jeweiliger **Wurzel** ist induktiv wie folgt definiert:

a) Jedes Axiom $\overline{B}$ ist eine Herleitung, und die Wurzel ist B.

b) Hat man eine Regel $\dfrac{A_1 \dots A_n}{B}$ 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:

$$\frac{\mathcal{H}_1 \dots \mathcal{H}_n}{B}$$

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, mit Formeln oder Hoare-Formeln als Knotenmarkierungen. Diese „Herleitungsbäume" besitzen als Blattmarkierungen nur gültige Formeln. Weiterhin gilt für jeden anderen Knoten, daß man von seiner Markierung die Markierungen seiner unmittelbaren Baumnachfolger bekommt, indem man eine Regel des Hoare-Kalküls von unten nach oben „liest". Die Wurzel eines Herleitungsbaums ist mit der hergeleiteten Formel markiert.

In Abschnitt 6.4.6 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.**

Bevor wir nun Beispiele für Herleitungen von Hoare-Formeln im Hoare-Kalkül angeben und die Verbindung zur Programmiermethodik herstellen, ist noch eine Bemerkung zur Zuweisung angebracht.

6.4.10 Kollaterale Zuweisung

In Abschnitt 6.3.4 hatten wir die kollaterale Zuweisung $(x_1, \dots, x_n)$:= $(t_1, \dots, t_n)$ als Pseudo-Anweisung eingeführt, die Programmherleitungen aus Repetitionen erleichtert, aber bei einem Übergang zu „echtem" Modula-2 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 Abschnitt 6.4.8 definieren wir für kollaterale Zuweisungen das Axiom

$$\overline{\{\varphi^{t_1,\dots,t_n}_{x_1,\dots,x_n}\}\ (x_1, \dots, x_n)\ :=\ (t_1, \dots, t_n)\ \{\varphi\}}$$

für alle Formeln φ. In diesem Axiom ist $\varphi^{t_1,\dots,t_n}_{x_1,\dots,x_n}$ diejenige Formel, die aus der Formel φ entsteht, indem man gleichzeitig alle Programmvariablen 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 kollaterale Zuweisungen verwendet werden.

6.4.11 Beispiel (Verifikation eines gegebenen Programms)

Wir betrachten das nachfolgende while-Programm mit drei Programmvariablen `n`, `a` und `res` jeweils von der Sorte `CARDINAL`.

```
(n,res) := (a,1);
WHILE n # 0 DO
  (n,res) := (n-1,n*res)
  END
```

Nach der Herleitung von Beispiel 6.3.5 wissen wir schon, daß der Wert von `res` nach der Abarbeitung gleich der Fakultät des Wertes von `a` ist. Wir wollen nun mit den neu vorgestellten Mitteln zeigen, daß das Programm total-korrekt ist bezüglich `TRUE` als Vorbedingung und $\mathtt{res} = \mathtt{a}!$ als Nachbedingung. Dazu verwenden wir Satz 6.4.7 und spalten die gestellte Aufgabe in zwei Teilaufgaben auf.

Die **Terminierung** des Programms ist offensichtlich, da der Wert von `n` in jedem Schleifendurchlauf echt vermindert wird und die Schleifenbedingung zusichert, daß `n - 1` definiert ist.

Zum Beweis der **partiellen Korrektheit** verfolgen wir die Werte der Programmvariablen 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.

Nach i-tem Durchlauf	`n`	`res`
0	$\mathtt{a}$	1
1	$\mathtt{a} - 1$	$\mathtt{a}$
2	$\mathtt{a} - 2$	$(\mathtt{a} - 1) * \mathtt{a}$
...	...	...
i	$\mathtt{a} - i$	$\prod_{j=\mathtt{a}-i+1}^{\mathtt{a}} j$
...	...	...

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

$$\{\mathtt{TRUE}\} \quad \begin{array}{l} \mathtt{(n,res)\ :=\ (a,1);} \\ \mathtt{WHILE\ n\ \#\ 0\ DO} \\ \quad \mathtt{(n,res)\ :=\ (n-1,n*res)} \\ \quad \mathtt{END} \end{array} \quad \{\mathtt{res} = \prod_{j=1}^{\mathtt{a}} j\}$$

im Hoare-Kalkül. Zur Vereinfachung bezeichne im folgenden *init* die Initialisierung von `n` und `res` und *loop* die while-Schleife. Dann haben wir

$$\frac{\{\mathtt{TRUE}\}\ \mathit{init}\ ;\ \mathit{loop}\ \{\mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j \wedge \mathtt{n} = 0\} \qquad \mathtt{res} = \prod_{j=\mathtt{n}+1}^{\mathtt{a}} j \wedge \mathtt{n} = 0 \rightarrow \mathtt{res} = \prod_{j=1}^{\mathtt{a}} j}{\{\mathtt{TRUE}\}\ \mathit{init}\ ;\ \mathit{loop}\ \{\mathtt{res} = \prod_{j=1}^{\mathtt{a}} j\}}$$

nach der Konsequenzregel zur Verstärkung der Nachbedingung der Unterformel. Zur Gewinnung einer Gesamtherleitung brauchen wir noch jeweils Herleitungen für die Hoare-Formel bzw. die Formel über dem Strich:

Der Fall der rechten Oberformel ist einfach. Diese stellt offensichtlich eine gültige Formel dar und eine Herleitung ist somit das folgende Axiom:

$$\frac{}{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \ \wedge \ \texttt{n = 0} \rightarrow \texttt{res} = \prod_{j=1}^{\texttt{a}} j}$$

Im Fall der linken Oberformel bekommen wir durch die Regeln für die sequentielle Komposition und die while-Schleife die nachstehende Figur, da `NOT(n # 0)` gleichwertig ist zu `n = 0`:

$$\frac{\begin{array}{l}\{\texttt{TRUE}\}\\ \mathit{init}\\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}\end{array} \qquad \dfrac{\begin{array}{l}\{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n # 0}\}\\ \texttt{(n,res) := (n - 1,n * res)}\\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}\end{array}}{\begin{array}{l}\{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}\\ \mathit{loop}\\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n = 0}\}\end{array}}}{\{\texttt{TRUE}\}\ \mathit{init}\,;\mathit{loop}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n = 0}\}}$$

Nach dieser Reduktion stehen noch zwei Herleitungen aus. Die linke Hoare-Formel wird hergeleitet durch

$$\frac{\dfrac{}{\texttt{TRUE} \rightarrow \texttt{1} = \prod_{j=\texttt{a}+1}^{\texttt{a}} j} \qquad \dfrac{}{\{\texttt{1} = \prod_{j=\texttt{a}+1}^{\texttt{a}} j\}\ \mathit{init}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}}{\{\ \texttt{TRUE}\}\ \mathit{init}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}$$

denn die beiden Oberformeln sind nach Definition Axiome und die Unterformel folgt nach der ersten Konsequenzregel. Auch die obige rechte Hoare-Formel ist mittels dieser Konsequenzregel wie nachfolgend angegeben herleitbar:

$$\frac{\dfrac{}{\begin{array}{l}\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n # 0}\\ \quad \rightarrow \texttt{n * res} = \prod_{j=\texttt{n}}^{\texttt{a}} j\end{array}} \qquad \dfrac{}{\begin{array}{l}\{\texttt{n * res} = \prod_{j=\texttt{n}}^{\texttt{a}} j\}\\ \texttt{(n,res) := (n - 1,n * res)}\\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}\end{array}}}{\{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j \wedge \texttt{n # 0}\}\ \texttt{(n,res) := (n - 1,n * res)}\ \{\texttt{res} = \prod_{j=\texttt{n}+1}^{\texttt{a}} j\}}$$

Insgesamt haben wir also die folgende Herleitungsstruktur zur partiellen Korrektheit des oben angegebenen while-Programms zur Berechnung der Fakultät:

$$\dfrac{\dfrac{\dfrac{}{A_5} \quad \dfrac{}{A_6}}{A_3} \qquad \dfrac{\dfrac{\dfrac{}{A_8} \quad \dfrac{}{A_9}}{A_7}}{A_4}}{\dfrac{A_2}{A_1}}$$

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 **Problemspezifikation** vorliegen als Paar

φ : Vorbedingung (was sind die Voraussetzungen an die Eingabe)

ψ : Nachbedingung (welche Werte sollen berechnet und in welchen Programmvariablen abgespeichert werden)

und sucht ein while-Programm s mit $\{\varphi\}\, s\, \{\psi\}$. 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 Hand in Hand mit seinem Korrektheitsbeweis, wobei die Richtung, wie es weitergeht, normalerweise durch den Beweis bestimmt wird. Um das Vorgehen 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 ist nachfolgend angegeben.

6.4.12 Satz (Entwurf von Programmen)

Eine Hoare-Formel der Form

$$\{\varphi\}\ init\ ;\ \texttt{WHILE}\ b\ \texttt{DO}\ body\ \texttt{END}\ \{\psi\}$$

mit Anweisungen *init* (der Initialisierung) und *body* (dem Schleifenrumpf) ist herleitbar, falls, mit einer Formel (der Invariante) I die folgenden drei Beweisverpflichtungen gelten:

(1) Die Formel $I \wedge \texttt{NOT}(b) \to \psi$ ist gültig.

(2) Die Hoare-Formel $\{\varphi\}\ init\ \{I\}$ ist herleitbar.

(3) Die Hoare-Formel $\{I \wedge b\}\ body\ \{I\}$ ist herleitbar.

Beweis: Man ergänzt die nach den Beweisverpflichtungen (2) und (3) existierenden Herleitungen zu einer Gesamtherleitung wie folgt:

$$\dfrac{\dfrac{\dfrac{\dots^{(2)}}{\{\varphi\}\ init\ \{I\}} \quad \dfrac{\dfrac{\dots^{(3)}}{\{I \wedge b\}\ body\ \{I\}}}{\{I\}\ \texttt{WHILE}\ b\ \texttt{DO}\ body\ \texttt{END}\ \{I \wedge \texttt{NOT}(b)\}}}{\{\varphi\}\ init;\ \texttt{WHILE}\ b\ \texttt{DO}\ body\ \texttt{END}\ \{I \wedge \texttt{NOT}(b)\}} \qquad \dfrac{}{I \wedge \texttt{NOT}(b) \to \psi}}{\{\varphi\}\ init;\ \texttt{WHILE}\ b\ \texttt{DO}\ body\ \texttt{END}\ \{\psi\}}$$

Man beachte dabei, daß, nach der Beweisverpflichtung (1), die rechte Oberformel des ersten Schrittes 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 suche man zuerst eine Verstärkung der Nachbedingung, welche die Invariante und die Schleifenbedingung der zu konstruierenden while-Schleife liefert. Anschließend suche man eine Initialisierung, welche die Invariante unter der Vorbedingung etabliert, und einen Schleifenrumpf, dessen Ausführung die Gültigkeit der Invariante aufrechterhält. Dieses Verfahren kann natürlich auch bei der Konstruktion der Initialisierung und des Schleifenrumpfs wiederum angewendet werden, so daß beispielsweise auch Programme mit geschachtelten 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 entwickeln kann.

6.4.13 Beispiel (Entwicklung eines Programms)

Gegeben sei die Aufgabe, ein while-Programm s zu entwickeln, so daß die nachfolgende Hoare-Formel herleitbar ist:

$$\{\texttt{TRUE}\}\ s\ \{\texttt{x} = \mathit{fib}(\texttt{a})\}$$

Dabei sind `x` und `a` zwei Programmvariablen der Sorte `CARDINAL` und *fib* eine Operation der erweiterten Signatur zur Berechnung der Folge der Fibonacci-Zahlen, d.h. definiert durch die 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 5.5.10.b, wo wir ein ML-Programm zur Berechnung von Fibonacci-Zahlen entwickelt haben. Weiterhin beachte man, daß $\mathit{fib}(\texttt{a})$ kein Modula-2-Term ist – was wir dadurch anzeigen, daß die Klammern bei der Anwendung von *fib* nicht in Schreibmaschinenschrift gesetzt sind.

Zur Berechnung von *fib* orientieren wir uns an der oben angegebenen Rekursion und verwenden neben `x` noch zwei weitere Programmvariablen zur Tabellierung der Funktionswerte von *fib* für zwei aufeinanderfolgende Argumente.

Formal setzen wir die Programmentwicklung an mit der offensichtlich gültigen Formel

$$\texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1) \wedge \texttt{u = a} \ \rightarrow\ \texttt{x} = \mathit{fib}(\texttt{a}),$$

wobei `u` und `y` die neuen Programmvariablen der Sorte `CARDINAL` sind. Mit Blick auf diese gültige Formel und auf die Beweisverpflichtung (1) von Satz 6.4.12 können wir nun als Invariante I und als Schleifenbedingung b versuchen

$$I \ \stackrel{\wedge}{=}\ \texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1) \qquad\qquad b \ \stackrel{\wedge}{=}\ \texttt{u \# a},$$

da der Term (die Ungleichung) $\texttt{NOT(u # a)}$ äquivalent ist zur Gleichung $\texttt{u = a}$. Damit gilt die Beweisverpflichtung (1) von Satz 6.4.12 mit $\texttt{TRUE}$ als Vorbedingung φ und $\texttt{x} = \mathit{fib}(\texttt{a})$ als Nachbedingung ψ, und unser zukünftiges while-Programm hat die Form

$$\begin{array}{l} \mathit{init}\,; \\ \texttt{WHILE u # a DO } \mathit{body} \texttt{ END} \end{array}$$

Es bleiben noch die Initialisierung *init* und der Schleifenrumpf *body* zu finden, so daß auch die restlichen Beweisverpflichtungen (2) und (3) von Satz 6.4.12 gelten. Wir setzen für die Initialisierung *init* die nachfolgende Zuweisung an:

$$\texttt{(x,y,u) := (1,1,0)}$$

Dieser Ansatz ist motiviert durch die Bedingung $\texttt{u # a}$ der Schleife. D.h. man wird sich der Eingabe $\texttt{a}$ von „unten“ her nähern. Mit $\texttt{u} = \texttt{0}$ als Start ergibt sich die Initialisierung der Programmvariablen $\texttt{x}$ und $\texttt{y}$ mittels $\texttt{1}$ aus der Rekursion der Fibonacci-Zahlen. Der formale Beweis der Beweisverpflichtung (2) von Satz 6.4.12 verwendet die erste Konsequenzregel und sieht wie folgt aus:

$$\dfrac{\dfrac{}{\texttt{TRUE} \to \texttt{1} = \mathit{fib}(\texttt{0}) \wedge \texttt{1} = \mathit{fib}(\texttt{1})} \qquad \dfrac{}{\begin{array}{l} \{\texttt{1} = \mathit{fib}(\texttt{0}) \wedge \texttt{1} = \mathit{fib}(\texttt{1})\} \\ \texttt{(x,y,u) := (1,1,0)} \\ \{\texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1)\} \end{array}}}{\{\texttt{TRUE}\}\ \texttt{(x,y,u) := (1,1,0)}\ \{\texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1)\}}$$

Es bleiben also noch die Entwicklung des Schleifenrumpf *body* und die Verifikation der Beweisverpflichtung (3) von Satz 6.4.12. Da Null der Initialwert der Programmvariablen $\texttt{u}$ ist, müssen wir von $\texttt{u}$ nach $\texttt{u + 1}$ übergehen, um irgendwann schließlich $\texttt{a}$ zu treffen. Offensichtlich gilt für diese Abänderung von $\texttt{u}$ die Implikation

$$\texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1) \implies \texttt{y} = \mathit{fib}(\texttt{u + 1}) \wedge \texttt{x + y} = \mathit{fib}(\texttt{u + 1}+1),$$

da die Gleichung

$$\texttt{x} + \texttt{y} = \mathit{fib}(\texttt{u}) + \mathit{fib}(\texttt{u}+1) = \mathit{fib}(\texttt{u + 1}+1)$$

nach der oben angegebenen Fibonacci-Rekursion zutrifft. Also haben wir als Ansatz für den Schleifenrumpf die kollaterale Zuweisung

$$\texttt{(x,y,u) := (y,x + y,u + 1).}$$

Ein formaler Beweis der Beweisverpflichtung (3) von Satz 6.4.12 verwendet wiederum die erste Konsequenzregel und ist nachstehend angegeben:

$$\dfrac{\dfrac{}{\begin{array}{l} \texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1) \\ \wedge\ \texttt{u # a} \to \\ \texttt{y} = \mathit{fib}(\texttt{u + 1}) \wedge \texttt{x + y} = \mathit{fib}(\texttt{u + 1}+1) \end{array}} \qquad \dfrac{}{\begin{array}{l} \{\texttt{y} = \mathit{fib}(\texttt{u + 1}) \wedge \texttt{x + y} = \mathit{fib}(\texttt{u + 1}+1)\} \\ \texttt{(x,y,u) := (y,x + y,u + 1)} \\ \{\texttt{x} = \mathit{fib}(\texttt{u}) \wedge \texttt{y} = \mathit{fib}(\texttt{u}+1)\} \end{array}}}{\left\{\begin{array}{l} \texttt{x} = \mathit{fib}(\texttt{u}) \wedge \\ \texttt{y} = \mathit{fib}(\texttt{u}+1)\ \wedge \texttt{u # a} \end{array}\right\}\ \texttt{(x,y,u) := (y,x + y,u + 1)}\ \left\{\begin{array}{l} \texttt{x} = \mathit{fib}(\texttt{u}) \wedge \\ \texttt{y} = \mathit{fib}(\texttt{u}+1) \end{array}\right\}}$$

Insgesamt haben wir für s das folgende while-Programm entwickelt, wobei alle verwendeten Programmvariablen `x`, `y`, `u` und `a` die Sorte `CARDINAL` besitzen:

```
(x,y,u) := (1,1,0);
WHILE u # a DO
  (x,y,u) := (y,x+y,u+1)
  END
```

Eine Sequentialisierung der Initialisierung und des Schleifenrumpfs dieses Programms gefolgt von einer Vervollständigung zu einem Modula-2-Programm liefert schließlich:

```
MODULE Fibonacci;
FROM InOut IMPORT WriteLn, WriteCard, ReadCard;
VAR a, x, y, u, h : CARDINAL;
BEGIN
  ReadCard(a);
  u := 0;
  y := 1;
  x := 1;
  WHILE u # a DO
    u := u + 1;
    h := y;
    y := x + y;
    x := h
    END;
  WriteCard(x,5);
  WriteLn
END Fibonacci .
```

Man beachte, daß in `Fibonacci` im Vergleich zu obigem while-Programm bei der Sequentialisierung der kollateralen Zuweisung in der Schleife eine Hilfsvariable `h` verwendet wird. Theoretisch ginge es auch ohne diese, indem man

```
(x,y,u) := (y,x+y,u+1)
```

sequentialisiert mittels der Zuweisungsfolge

```
u := u+1; y := x+y; x := y-x.
```

Solche Tricks machen Programme aber nur undurchsichtig, schwer verständlich und deshalb auch schwer verifizierbar. Sie haben somit in der Informatik nichts verloren. ■

7 Imperative Programmierung: Vertiefung

Nachdem im letzten Kapitel die Grundelemente imperativer Programmierung anhand der elementaren Datenstrukturen und Anweisungen von Modula-2 behandelt wurden, befassen wir uns nun vertieft mit diesem Programmierparadigma. Auf der Seite der Datenstrukturen bedeutet dies die Verallgemeinerung der bisherigen Sorten zu Typen; bei den Kontrollstrukturen beinhaltet dies insbesondere, daß wir uns mit dem Abstraktionsmechanismus der Prozedur beschäftigen. Wie bei der Programmiersprache ML in Kapitel 5, so beginnen wir auch dieses Kapitel wiederum mit dem Ausbau der Datenstrukturen.

7.1 Aufzählungs- und Bereichstypen

Bisher kennen wir die fünf vordefinierten Sorten `BOOLEAN`, `CARDINAL`, `INTEGER`, `REAL` und `CHAR` von Modula-2, die in Abschnitt 6.2 bei der Vorstellung der elementaren Datenstrukturen angegeben wurden. Darauf aufbauend werden nun Typen eingeführt. Wir behandeln in diesem Abschnitt zuerst die zwei einfachsten Fälle.

7.1.1 Aufzählungstypen

Ein **Aufzählungstyp** in Modula-2 mit $n > 0$ paarweise verschiedenen Bezeichnern id_1 bis id_n (auch **Konstanten** genannt) ist syntaktisch gegeben durch die Tupelbildung $(id_1, \ldots, id_n)$. ■

Der Sinn eines Aufzählungstyps ist, die Daten im Programm möglichst so darzustellen, wie sie in der Wirklichkeit auftreten. Zwei einfache Beispiele sind etwa

`(pik,herz,karo,kreuz)`

mit vier Konstanten zur Darstellung der Kartenfarben und

`(sunday,monday,tuesday,wednesday,thursday,friday,saturday)`

mit sieben Konstanten zur Darstellung der Wochentage. Diese Darstellungen vermeiden Codierungen, etwa durch die Zahlen von 1 bis 4 bzw. von 1 bis 7, und tragen somit zum besseren Verständnis der sie enthaltenen Programme bei.

7.1.2 Bereichstypen

Es seien a und e zwei Konstanten einer Sorte m, die nicht die Sorte `REAL` ist. Dann ist der durch a und e bestimmte **Bereichstyp** (auch **Ausschnittstyp** genannt) über dem **Grundtyp** m syntaktisch gegeben durch `[`a`..`e`]`. ■

Den eckigen Klammern bei der Definition eines Bereichstyps kann noch der Grundtyp vorangestellt sein. Dies ist nur interessant im Fall von `INTEGER`, wenn in `INTEGER[`a`..`e`]`

die Konstanten a und e natürliche Zahlen darstellen. Dann wird INTEGER[a..e] als Teil von INTEGER aufgefaßt, während [a..e] als Teil von CARDINAL aufgefaßt wird.

An dieser Stelle ist eine wichtige Bemerkung angebracht, die sich auf unser weiteres Vorgehen bezieht. Bei der Teilsprache von Modula-2, die im letzten Kapitel eingeführt wurde, werden im Importteil eines Programms neben den Lese- und Drucke-Anweisungen ggf. nur noch mathematische Operationen importiert. Die Einführung von Typen kann aber dazu führen, daß weitere Dinge importiert werden müssen. Wir kommen darauf bei den Zeigern zurück. Der Deklarationsteil eines Programms besteht bisher nur aus der Deklaration von Programmvariablen. Auch hier ergibt sich durch die Typen eine Erweiterung um Deklarationen von Typen in der nachfolgenden schematischen Art:

$$\begin{array}{r}
\texttt{TYPE}\ m_1 = \ldots; \\
\vdots \\
m_k = \ldots;
\end{array}$$

Dabei sind alle m_i, $1 \leq i \leq k$, frei wählbare Bezeichner für Typen und rechts der Gleichheitszeichen stehen jeweils Typen (genauer wiederum, wie bei ML, Typausdrücke). Bisher kennen wir davon nur die in den Abschnitten 7.1.1 und 7.1.2 eingeführten; andere Typkonstruktionen werden wir im Laufe des Skriptums noch kennenlernen. Eine weitere Erweiterung des Deklarationsteils betrifft die Variablendeklarationen. Bisher war es nur erlaubt, Programmvariablen jeweils einer Sorte zu deklarieren. Nun lassen wir verallgemeinernd auch zu, daß die deklarierten Programmvariablen einen beliebigen Typ statt nur eine Sorte besitzen. Dabei dürfen die Deklarationen von Typen und Programmvariablen vermischt werden, wie das folgende Beispiel zeigt[25]:

```
VAR  x : CARDINAL;
TYPE colour = (red,yellow,green);
     day = (sunday,monday,tuesday,wednesday,
             thursday,friday,saturday);
     month = [1..12];
     year = [1900..1999];
     PosCard = [1..MAX(CARDINAL)];
VAR  a : colour;
     y : day;
     z : year;
```

Wesentlich ist nur, daß die Deklaration eines Typs m vor der Deklaration der Programmvariablen des Typs m erfolgt. Die letzte Erweiterung der Programme, welche die Einführung der Typen nach sich zieht, betrifft den Programmrumpf. Durch die Typen werden neue Konstanten und Operationen zur Verfügung gestellt, so daß der bisherige Termbegriff erweitert wird. Typen führen aber auch zu einer Erweiterung der Menge der Anweisungen, wie wir noch sehen werden.

Nachfolgend geben wir zuerst die durch die Aufzählungs- und Bereichstypen bewirkte Erweiterung des Termbegriffs an.

[25] Es sei schon an dieser Stelle erwähnt, daß sich diese Vermischung fortsetzt bei den Deklarationen von Prozeduren und Rechenvorschriften, denen wir uns später zuwenden werden.

7.1.3 Erweiterung des Termbegriffs

Durch einem Aufzählungstyp m werden zum Termaufbau, neben den Programmvariablen des Typs m aus dem Deklarationsteil, zusätzlich zur Verfügung gestellt:

a) Die durch ihn eingeführten Konstanten.

b) Eine Gleichheitsoperation `=`, eine Ungleichheitsoperation `#`, und die Vergleichsrelationen `<`, `<=`, `>` und `>=`, die alle in der gewohnten Infix-Schreibweise notiert werden. Die Gleichheitsoperation und Ungleichheitsoperation entsprechen der textuellen Gleichheit bzw. Ungleichheit der Bezeichner, die Operation `<` ist durch die Aufschreibung von links nach rechts festgelegt, also gelten beispielsweise

$$\texttt{red}\,\texttt{<}\,\texttt{yellow} = tt \qquad \texttt{yellow}\,\texttt{<}\,\texttt{green} = tt \qquad \texttt{green}\,\texttt{<}\,\texttt{yellow} = f\!f$$

im Fall der oben eingeführten Ampelfarben, und die restlichen Vergleichsoperationen werden schließlich wie üblich auf `<` und `=` zurückgeführt.

Auf einem Bereichstyp m hat man alle Operationen des Grundtyps zur Verfügung und es geschieht **implizite Typanpassung**. Somit tragen nur die deklarierten Programmvariablen des Typs m zur Erweiterung des Termbegriffs bei. ■

Nun geben wir die durch die Aufzählungs- und Bereichstypen verursachten zusätzlichen Anweisungen an.

7.1.4 Erweiterung des Anweisungsbegriffs

Auf allen Sorten, Aufzählungs- und Bereichstypen von Modula-2 sind die Vergleichsrelationen `<`, `<=`, `>` und `>=` festgelegt. Dies gilt auch für die Wahrheitswerte `BOOLEAN`, die in Modula-2 als Aufzählungstyp `(FALSE,TRUE)` aufgefaßt werden. Mit Ausnahme der Sorte `REAL` erfüllt die Vergleichsoperation `<=` jeweils die Axiome einer sogenannten **linearen diskreten Ordnung** $\leq$, bei der man alle Elemente in der Form einer endlichen oder unendlichen Kette

$$\ldots \leq x_{i-2} \leq x_{i-1} \leq x_i \leq x_{i+1} \leq x_{i+2} \leq \ldots$$

anordnen und damit von nächstkleineren und nächstgrößeren Elementen reden kann.

In Modula-2 wird dieser Übergang durch zwei Anweisungen bewerkstelligt. Ist x eine Programmvariable von einer Sorte ungleich `REAL`, einem Aufzählungs- oder Bereichstyp, so wird durch die Anweisung

- `DEC(`x`)` der Wert von x zum Wert der nächstkleineren Konstante verändert (sofern eine solche vorhanden ist) und durch
- `INC(`x`)` der Wert von x zum Wert der nächstgrößeren Konstante verändert (sofern eine solche vorhanden ist).

Als ein einfaches Beispiel betrachten wir die oben eingeführte Programmvariable `a` vom Aufzählungstyp `colour`. Hat `a` den Wert `red`, so besitzt `a` nach der Ausführung von `INC(a)` den Wert `yellow`.

Neben diesen beiden Anweisungen verallgemeinert sich durch die Aufzählungs- und Bereichstypen auch der Zuweisungsbegriff. Wir lassen nun in x `:=` t verallgemeinernd zu, daß x eine Programmvariable eines Aufzählungs- oder Bereichstyps m und t ein Term des Typs m ist. ■

Für weitere Operationen auf Sorten, Aufzählungs- und Bereichstypen verweisen wir auf das in der Einleitung zitierte Buch „Skriptum Informatik: Eine konventionelle Einführung" von H.J. Appelrath und J. Ludewig.

7.2 Zusammengesetzte Datentypen

Wie bei ML, so werden auch in der Programmiersprache Modula-2 Typen (Typausdrücke) induktiv auf den Sorten aufbauend definiert. Bei ML hatten wir die Konstruktoren `*` und `->` zur Produkt- bzw. Funktionstypenbildung; siehe Abschnitt 5.1.1. Bedingt durch das imperative Programmierparadigma liegen in Modula-2 die Dinge etwas anders. Insbesondere ist die syntaktische Definition von Typen und deren (formale) Semantik im Vergleich zu ML wesentlich komplizierter. Deshalb fassen wir die induktive Definition auch nicht wie in Abschnitt 5.1.1 in einem Punkt zusammen, sondern behandeln die einzelnen Fälle getrennt. Wir beginnen mit der Basis der Definition und rechtfertigen damit im Grunde die Sprechweise des letzten Abschnitts 7.1.

7.2.1 Typen in Modula-2: Basis

a) Jede Sorte der Signatur von Modula-2 ist ein Typ.

b) Jeder Aufzählungstyp und jeder Bereichstyp ist ein Typ. Diese speziellen Typen werden **einfache Typen** genannt. ■

Nach dieser Basis der induktiven Definition der Typen in Modula-2 wenden wir uns nun dem Induktionsschritt zu. Hier behandeln wir drei Arten von Typkonstruktionen, nämlich **Felder** (auch Reihungen oder Arrays genannt), **Verbunde** (auch Records genannt) und **Zeiger** (auch Verweise oder Pointer genannt). Wir beginnen mit den Feldern.

7.2.2 Felder

Es seien *index* ein **einfacher Typ** (in diesem Zusammenhang **Indextyp** genannt) und m ein beliebiger Typ (in diesem Zusammenhang **Komponententyp** genannt). Dann ist auch

`ARRAY` *index* `OF` m

ein Typ, genannt **Feldtyp** oder **Arraytyp**. Ein Objekt eines solchen Typs (ein **Feld** oder **Array**) ist eine mit *index* indizierte Reihung von Programmvariablen des Typs m. Die einzelnen Programmvariablen eines Felds nennt man auch **indizierte Programmvariablen**. ■

Man beachte, daß ein Feld eine **homogene** Datenstruktur ist in dem Sinne, daß die Werte aller in ihm abgespeicherten Daten vom selben Typ sind. Beispielsweise sind nach den zwei Deklarationen

```
VAR a : ARRAY [1..100] OF REAL;
    b : ARRAY colour OF REAL;
```

im Restprogramm verfügbar eine Programmvariable `a` für eine Reihung von 100 indizierten Programmvariablen des Typs `REAL`, die mit den Zahlen von 1 bis 100 indiziert sind, und eine Programmvariable `b` für eine Reihung von 3 indizierten Programmvariablen des Typs `REAL`, die mit den Konstanten `red`, `yellow` und `green` indiziert sind.

Nachfolgend geben wir die durch die Einführung der Felder bewirkten Erweiterungen der bisherigen Sprache an.

7.2.3 Erweiterungen des Term- und Anweisungsbegriffs

Es seien a ein Feld und i ein Term des entsprechenden Indextyps. Dann ist die mit i indizierte Programmvariable von a bezeichnet durch die Notation a`[`i`]`. Man sagt auch, daß man mittels a`[`i`]` auf die i-te Programmvariable von a zugreift. Feldzugriffe erweitern die Möglichkeiten, Terme in Modula-2 zu konstruieren. Ist a ein Feld mit Komponententyp m und i ein Index-Term, so ist a`[`i`]` ein Term des Typs m. Falls m eine Sorte ungleich `BOOLEAN` ist, so kann insbesondere a`[`i`]` durch eine der Drucke-Anweisungen von Abschnitt 6.3.1 ausgegeben werden.

Analog zum letzten Abschnitt 7.1 lassen wir auch bei Feldern zu, daß in einer Zuweisung x `:=` t auf der linken Seite eine Programmvariable eines Feldtyps steht und rechts ein Term des gleichen Typs. Da es aber keine Operation in Modula-2 gibt, die einen Feldtyp als Resultattyp besitzt[26], ist die rechte Seite einer solchen Zuweisung ebenfalls immer eine Programmvariable.

Eine **Änderung** des Wertes der indizierten Programmvariablen a`[`i`]` eines Felds a ist möglich, indem man bei Zuweisungen auf der linken Seite auch die indizierte Programmvariable a`[`i`]` zuläßt, d.h., Typgleichheit beider Seiten vorausgesetzt,

a`[`i`] := t`

schreibt. Falls der Komponententyp eine Sorte ungleich `BOOLEAN` ist, so kann das Eingeben eines Wertes und das Abspeichern unter a`[`i`]` durch eine der Lese-Anweisungen von Abschnitt 6.3.1 geschehen. ■

[26]Eine Ausnahme bildet die Darstellung von Zeichenreihen durch Konstanten für Felder mit Komponententyp `CHAR`, auf die wir aber nicht eingehen.

Für das obige Feld `a` des Typs `ARRAY [1..100] OF REAL` ist somit `a[52] + 28.33` ein Term der Sorte `REAL` und `a[52] := a[52] + 28.33` eine Zuweisung.

7.2.4 Mehrdimensionale Felder

Bei einem Feldtyp kann der Komponententyp wiederum ein Feldtyp sein, also etwa von der Form wie nachfolgend angegeben:

$$\texttt{ARRAY}\ index_1\ \texttt{OF ARRAY}\ index_2\ \texttt{OF}\ m \tag{1}$$

In diesem Fall erfolgt der Zugriff durch die Konstruktion $a\texttt{[}i\texttt{][}j\texttt{]}$, denn $a\texttt{[}i\texttt{]}$ ist eine Programmvariable des Typs $\texttt{ARRAY}\ index_2\ \texttt{OF}\ m$. Gleiches gilt für die Änderung des Wertes dieser zweifach indizierten Programmvariablen. Hier hat man $a\texttt{[}i\texttt{][}j\texttt{]} := t$ als Zuweisung. Für geschachtelte Deklarationen und Zugriffe sind Abkürzungen möglich, nämlich

$$\texttt{ARRAY}\ index_1\texttt{,}index_2\ \texttt{OF}\ m \tag{2}$$

statt $\texttt{ARRAY}\ index_1\ \texttt{OF ARRAY}\ index_2\ \texttt{OF}\ m$ und auf dies aufbauend $a\texttt{[}i\texttt{,}j\texttt{]}$ statt $a\texttt{[}i\texttt{][}j\texttt{]}$ und die Zuweisung $a\texttt{[}i\texttt{,}j\texttt{]} := t$ statt $a\texttt{[}i\texttt{][}j\texttt{]} := t$. Objekte des Feldtyps (1) oder (2) nennt man zwei-dimensionale Felder. Die Verallgemeinerung auf **n-dimensionale Felder** ist offensichtlich. ■

Im folgenden zeigen wir anhand eines einfachen Beispiels, wie die Programmentwicklungstechnik von Abschnitt 6.3.4 auch auf Programme mit Feldern angewendet werden kann.

7.2.5 Beispiel (Maximumssuche)

Gegeben sei ein Feld des Typs `ARRAY [1..N] OF CARDINAL`, genannt `a`. Wir betrachten, mit einer Pseudooperation `Max` zur Maximumsbestimmung, die nachstehende Zuweisung, die einer Programmvariablen `res` ein maximales Feldelement von `a` zuweist:

$$\texttt{res := Max}_{1 \leq i \leq \texttt{N}}\ \texttt{a[}i\texttt{]}$$

Zur Überführung dieser Zuweisung in ein Modula-2-Programmstück betten wir `Max` in eine allgemeinere Pseudooperation

$$\texttt{Maxim(k)} = \mathrm{Max}_{\texttt{k} \leq i \leq \texttt{N}}\ \texttt{a[}i\texttt{]}$$

ein, die sich durch Parametrisierung der unteren Grenze 1 zu `k` ergibt. Wir erhalten damit zur originalen Zuweisung gleichwertig die Zuweisung

$$\texttt{res := Maxim(1).}$$

Eine rekursive Version der Pseudooperation `Maxim` ist sehr einfach zu entwickeln. Wir bekommen nach einer kurzen Rechnung die Gleichung

$$\texttt{Maxim(k)} = \begin{cases} \texttt{a[k]} & : \ \texttt{k} = \texttt{N} \\ \texttt{max(a[k],Maxim(k+1))} & : \ \text{sonst} \end{cases}$$

mit einer weiteren Pseudooperation max, welche das Maximum von zwei natürlichen Zahlen bestimmt. Die Terminierung dieser Rekursion ist offensichtlich. Der nächste Schritt besteht in der Einbettung der Pseudooperation Maxim in eine weitere, aber nun repetitiv-rekursiv beschriebene Pseudooperation. Die Operation max zur Maximumsbestimmung ist assoziativ mit der Null als neutralem Element. Wir folgen also dem schon bei ML gewählten Ansatz mittels der Gleichung

$$\texttt{Maximum(k,m)} = \texttt{max(m,Maxim(k))}$$

und erhalten damit res := Maxim(1) und somit auch res := $\texttt{Max}_{1\leq i\leq \texttt{N}}\texttt{a[i]}$ als gleichwertig zur Zuweisung

$$\texttt{res := Maximum(1,0).}$$

Weiterhin gilt für die Pseudooperation Maximum, daß

$$\begin{aligned}
\texttt{Maximum(k,m)} &= \texttt{max(m,Maxim(k))} \\
&= \begin{cases} \texttt{max(m,a[k])} & : \ \texttt{k} = \texttt{N} \\ \texttt{max(m,max(a[k],Maxim(k+1)))} & : \ \text{sonst} \end{cases} \\
&= \begin{cases} \texttt{max(m,a[k])} & : \ \texttt{k} = \texttt{N} \\ \texttt{max(max(m,a[k]),Maxim(k+1))} & : \ \text{sonst} \end{cases} \\
&= \begin{cases} \texttt{max(m,a[k])} & : \ \texttt{k} = \texttt{N} \\ \texttt{Maximum(k+1,max(m,a[k]))} & : \ \text{sonst.} \end{cases}
\end{aligned}$$

Auch die aus dieser Rechnung sich ergebende repetitive Rekursion

$$\texttt{Maximum(k,m)} = \begin{cases} \texttt{max(m,a[k])} & : \ \texttt{k} = \texttt{N} \\ \texttt{Maximum(k+1,max(m,a[k]))} & : \ \text{sonst} \end{cases}$$

für die Pseudooperation Maximum terminiert. Nun können wir die Regel „(3) ⇒ (4)“ aus Abschnitt 6.3.4 anwenden und erhalten als gleichwertig zu res := Maximum(1,0) das nachstehende Programmstück:

```
(k,m) := (1,0);
WHILE k # N DO
  (k,m) := (k + 1, max(m,a[k]))
  END;
res := max(m,a[k])
```

Als nächste Tätigkeiten stehen an die Sequentialisierung der kollateralen Zuweisungen und die Entfernung der Pseudooperation max. Ersteres ist trivial. Man hat die Reihenfolge „erst m, dann k“. Letzteres ist durch eine bewachte Anweisung möglich. Das Resultat ist:

```
m := 0;
k := 1;
WHILE k # N DO
  IF m < a[k] THEN m := a[k] END;
  k := k + 1
  END;
IF m < a[k] THEN res := a[k]
            ELSE res := m END
```

Man beachte, daß die Programmvariable `k` hier wirklich vom Indextyp `[1..N]` des Felds gewählt werden kann. Die formale Herleitung dieser Anweisung aus der initial gegebenen Zuweisung erfordert keinen Indexüberlauf wie oft üblich. Dieser wird durch die abschließende bewachte Anweisung verhindert.

Es verbleibt noch die Vervollständigung der hergeleiteten Anweisung zu einem vollständigen Modula-2-Programm, indem man unter anderem den Feldtyp und die verwendeten Programmvariablen `a`, `k` und `m` deklariert. So ergibt sich schließlich das Programm

```
MODULE Maximum;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadCard, WriteCard;
VAR a : ARRAY [1..20] OF CARDINAL;
    k : [1..20];
    m, res : CARDINAL;
BEGIN
  WriteString('Gib 20 natuerliche Zahlen ein.');
  WriteLn;
  k := 1;
  WHILE k # 20 DO
    ReadCard(a[k]);
    k := k + 1
    END;
  k := 1;
  m := 0;
  WHILE k # 20 DO
    IF m < a[k] THEN m := a[k] END;
    k := k + 1
    END;
  IF m < a[k] THEN res := a[k]
              ELSE res := m END;
  WriteString('Maximum: ');
  WriteCard(res,10);
  WriteLn
END Maximum .
```

wenn man beispielsweise die Konstante `N` als natürliche Zahl 20 wählt. Man beachte, daß es in Modula-2 keine dynamischen Felder gibt, deren Indextypen von Programmvariablenwerten abhängen können. Dies bedeutet einen Rückschritt gegenüber der Vorgängersprache Algol 60, wo solches möglich war, und ist in der Praxis oft hinderlich. ■

Nach den Feldern behandeln wir nun die Verbunde als zusammengesetzte Datentypen. Bei Feldern haben wir indizierte Programmvariable, die alle den gleichen Komponententyp besitzen. Der Indextyp ist geordnet, und der Zugriff erfolgt durch den Index. Typisch für Verbunde ist nun, daß die Komponenten von verschiedenen Typen sein dürfen und die Zugriffe durch sogenannte Selektoren erfolgen.

7.2.6 Verbunde

Zu $k > 0$ seien $m_1, \dots, m_k$ Typen (in diesem Zusammenhang **Komponententypen** genannt) und $sel_1, \dots, sel_k$ Bezeichner (in diesem Zusammenhang **Selektoren** genannt). Dann ist auch

$$\texttt{RECORD}\ sel_1\ \texttt{:}\ m_1\texttt{;}\dots\texttt{;}\ sel_k\ \texttt{:}\ m_k\ \texttt{END}$$

ein Typ, genannt **Verbundtyp** oder **Recordtyp**. Ein Objekt eines solchen Typs (ein **Verbund** oder **Record**) ist ein k-Tupel von Programmvariablen, jeweils vom Typ m_j und indiziert mit dem Selektor sel_j, $1 \leq j \leq k$. Falls einige der durch die Selektoren indizierten Programmvariablen den gleichen Typ haben, dann darf man die entsprechenden Selektoren vor der Typangabe zu einer Liste $sel_{j_1}, \dots, sel_{j_k} : m$ zusammenziehen. ■

Als einfaches Beispiel betrachten wir die Darstellung von Personendaten durch Verbunde, wobei einer Person die Anrede, der Name, der Wohnort und das Alter zugeordnet ist:

```
TYPE salutationtyp = (Mrs,Mr);
     Person = RECORD salutation : salutationtyp;
                     name, city : ARRAY [1..20] OF CHAR;
                     age : CARDINAL
              END;
VAR  p : person;
```

Nach diesen Deklarationen stehen im Restprogramm der Aufzählungstyp `salutationtyp`, der Verbundtyp `person` mit seinen vier Selektoren und die Programmvariable `p` vom Typ `person` zur Verfügung, wobei `p` ein Quadrupel von Programmvariablen mit Typen

`salutationtyp`   `ARRAY [1..20] OF CHAR`   `ARRAY [1..20] OF CHAR`   `CARDINAL`

ist, jeweils indiziert mit den Bezeichnern

`salutation`   `name`   `city`   `age`.

Nachfolgend geben wir die durch die Einführung der Verbunde bewirkten Erweiterungen der bisherigen Sprache an.

7.2.7 Erweiterungen des Term- und Anweisungsbegriffs

Jede einzelne Programmvariable eines Verbunds ist ansprechbar durch den **Selektorzugriff** $v\texttt{.}sel_j$, wobei v der Verbund und sel_j der j-te Selektor ist. Selektorzugriffe erweitern wiederum die Möglichkeiten, in Modula-2 Terme zu konstruieren und Daten einzulesen oder auszugeben. Beim Lesen und Drucken muß der Typ des Terms $v\texttt{.}sel_j$ natürlich eine Sorte ungleich `BOOLEAN` sein, damit die Lese- und Drucke-Anweisungen von Abschnitt 6.3.1 angewendet werden können.

Durch die Typfestlegung des Verbundtyps wird $v\texttt{.}sel_j$ als eine Programmvariable des Typs m_j definiert. Die **Änderung** des Wertes dieser Variablen zum Wert eines Terms t

des selben Typs ist möglich durch die Zuweisung

$$v\,.\,sel_j \ \texttt{:=} \ t.$$

Wir lassen auch bei Verbunden zu, daß in einer Zuweisung $x \texttt{:=} t$ auf der linken Seite eine Programmvariable eines Verbundtyps steht und rechts ein Term des gleichen Typs. Analog zu den Feldern gibt es jedoch auch in Modula-2 keine Operation, die einen Verbundtyp als Resultattyp besitzt, so daß die rechte Seite einer solchen Zuweisung immer eine Programmvariable ist. ■

Für den oben angegebenen Verbund `p` ist `3 * p.age + 15` ein Term der Sorte `CARDINAL` und `p.salutation` ein Term des Aufzählungstyps `salutationtyp`.

Durch Verbunde werden Tupel von Programmvariablen eingeführt. Der Verbundtyp von Modula-2 ist somit das „imperative Gegenstück" zu dem in ML vorhandenen „funktionalen" Produkttyp. Auch die in ML vorhandene „funktionale" disjunkte Vereinigung hat eine Entsprechung in Modula-2. Dies ist der nun zu besprechende variante Teil eines Verbundes.

7.2.8 Variantenbildung bei Verbunden

Bisher wird ein Verbundtyp interpretiert als Tupel von Programmvariablen. Dabei ist die Anzahl und die Typisierung fix. Gerade dies möchte man aber oft nicht haben, sondern wünscht etwa, daß der Verbundtyp interpretiert werde als Paar von Programmvariablen

$$\langle v\,.\,sel_1, v\,.\,sel_2\rangle,$$

jeweils vom Typ m_1 bzw. m_2, oder als Tripel von Programmvariablen

$$\langle v\,.\,sel'_1, v\,.\,sel'_2, v\,.\,sel'_3\rangle,$$

jeweils vom Typ n_1, n_2 bzw. n_3. Dies kann man nun dadurch ausdrücken, daß man den Programmvariablentupeln noch als zusätzliche Komponente eine **Diskriminator-Programmvariable** $v\,.\,disk$ eines **Diskriminatortyps** hinzufügt. Beim obigen Fall von zwei Varianten bietet sich etwa die Sorte `BOOLEAN` als Diskriminatortyp an, d.h. man hat als Objekte des Verbundtyps dann Tripel

$$\langle v\,.\,disk, v\,.\,sel_1, v\,.\,sel_2\rangle,$$

wobei die Programmvariable $v\,.\,disk$ den Wert `TRUE` hat, oder Quadrupel

$$\langle v\,.\,disk, v\,.\,sel'_1, v\,.\,sel'_2, v\,.\,sel'_3\rangle,$$

wobei nun die Programmvariable $v.disk$ den verbleibenden Wert `FALSE` besitzt. Die von Modula-2 hierfür gewählte programmiersprachliche Notation ist:

```
RECORD CASE disk  : BOOLEAN
       OF   TRUE  : sel₁ : m₁; sel₂ : m₂ |
            FALSE : sel'₁ : n₁; sel'₂ : n₂; sel'₃ : n₃
       END
END
```

Dieser Typ ist ein Verbundtyp, der nur aus einem varianten Teil besteht. Modula-2 läßt auch Mischungen zu, d.h. Verbunde, die sowohl fixe (wie in Abschnitt 7.2.6 eingeführt) und variante (wie im obigen Beispiel) Teile haben. Auch ist Variantenbildung nicht auf den Fall von zwei Varianten beschränkt. ■

Wir gehen bezüglich der Syntax von Verbunden mit Variantenbildung nicht ins Detail, sondern versuchen, eine typische Anwendung anhand des schon in Kapitel 5 behandelten Beispiels der Geradendarstellung zu geben (siehe auch Abschnitte 5.1.6 und 5.6.2).

7.2.9 Beispiel (Geradendarstellungen)

In ML hatten wir zur Darstellung von Geraden in Vektorform bzw. Normalform drei Typen `normform`, `vector` und `vecform` eingeführt. Den entsprechenden Deklarationen von Beispiel 5.1.6 entsprechen in Modula-2:

```
TYPE normform = RECORD a, b, c : REAL END;
     vector   = RECORD abs, ord : REAL END;
     vecform  = RECORD place, dir : vector END;
```

Dabei ist für eine Programmvariable `v` vom Verbundtyp `vector` durch `v.abs` die x-Koordinate und durch `v.ord` die y-Koordinate gegeben. Ist `vl` eine Programmvariable vom Verbundtyp `vecform`, so ist `vl.place` ein Term des Verbundtys `vector` und steht für den Ortsvektor und `vl.dir` ebenfalls ein Term des Verbundtyps `vector` und steht für den Richtungsvektor. Die Bedeutung der drei Selektoren `a`, `b` und `c` im Verbundtyp `normform` ist klar.

Um Geraden nun sowohl in Vektor- als auch in Normalform darstellen zu können, braucht man einen Diskriminatortyp mit zwei Konstanten, etwa

```
form = (vform,nform),
```

und einen Verbund, der nur aus einem varianten Teil besteht und drei Selektoren besitzt, nämlich einen Diskriminator-Selektor zum Zugriff auf den Wert der Diskriminator-Variable, einen Selektor für die Vektorform, falls die Diskriminator-Programmvariable den Wert `vform` hat, und einen Selektor für die Normalform, falls die Diskriminator-Programmvariable den Wert `nform` hat. Dies führt zu der folgenden Typdeklaration:

```
TYPE form     = (vform,nform);
     normform = RECORD a, b, c : REAL END;
     vector   = RECORD abs, ord : REAL END;
     vecform  = RECORD place, dir : vector END;
     line     = RECORD CASE gf    : form
                       OF   vform : vf : vecform |
                            nform : nf : normform
                       END
                END;
```

Will man nun einer Programmvariablen `g` des Verbundtyps `line` die identische Relation $g = \{ \langle x, x \rangle : x \in \mathbb{R} \}$ als Wert zuordnen, so hat man dies durch entsprechende Zuweisungen zu bewerkstelligen. Wir beginnen mit der Normalform. Nehmen wir, wie in Beispiel 5.1.6, die Darstellung

$$g = \{ \ \langle x, y \rangle \in \mathbb{R} \times \mathbb{R} \ : \ 1 * x + (-1) * y + 0 = 0 \ \},$$

so führt dies zu den folgenden vier Zuweisungen, wobei die erste Zuweisung die Diskriminator-Programmvariable mit dem richtigen Wert besetzt und die restlichen drei Zuweisungen die Koeffizienten belegen:

```
g.gf := nform;
g.nf.a := 1.0;
g.nf.b := -1.0;
g.nf.c := 0.0;
```

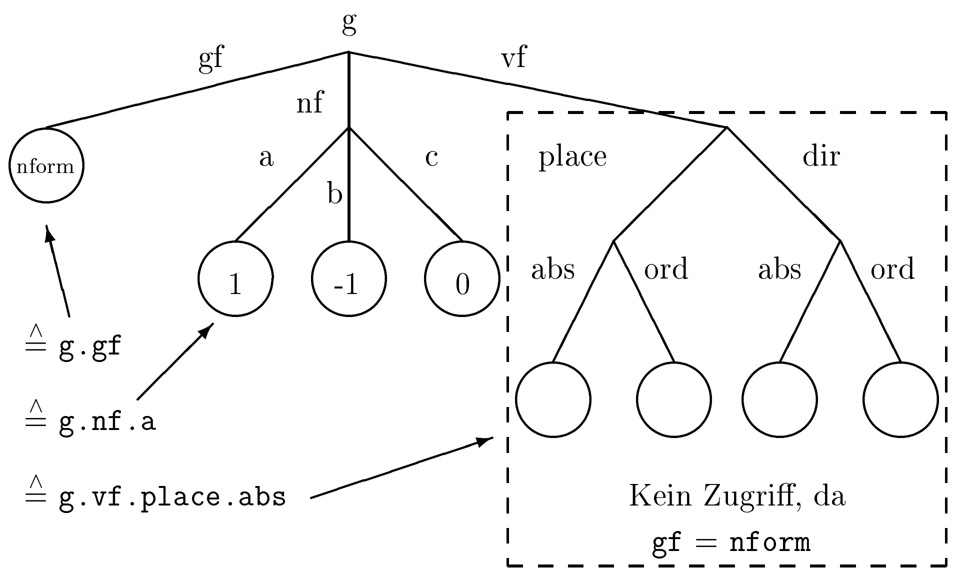
Um `g` den Wert der identischen Relation in der Vektorform

$$g = \{ \ \langle 0, 0 \rangle + \lambda * \langle 1, 1 \rangle \ : \ \lambda \in \mathbb{R} \ \}$$

mit dem Ortsvektor $\langle 0, 0 \rangle$ und dem Richtungsvektor $\langle 1, 1 \rangle$ zuzuordnen, ist die folgende Zuweisungssequenz notwendig:

```
g.gf := vform;
g.vf.place.abs := 0.0;
g.vf.place.ord := 0.0;
g.vf.dir.abs := 1.0;
g.vf.dir.ord := 1.0;
```

Bildlich kann man sich diese Darstellungen bzw. die entsprechenden Zugriffe sehr gut durch Bäume verdeutlichen. Im ersten Fall der Normalformdarstellung haben wir etwa den folgenden Selektorbaum:



Das folgende Programm zeigt eine Vervollständigung der obigen Typdeklarationen zu einem Modula-2-Programm, das eine Gerade einliest und in Abhängigkeit von der Darstellungsform feststellt, ob sie parallel zur x-Achse ist.

```
MODULE Line;
FROM InOut IMPORT WriteLn, WriteString, Read,
                  ReadReal, WriteReal;
TYPE form     = (vform,nform);
     normform = RECORD a, b, c : REAL END;
     vector   = RECORD abs, ord : REAL END;
     vecform  = RECORD place, dir : vector END;
     line     = RECORD CASE gf    : form
                       OF   vform : vf : vecform |
                            nform : nf : normform
                       END
                END;
VAR f : CHAR;
    g : line;
    res : BOOLEAN;
BEGIN
  WriteString('Gib Geradenform ein (V/N).');
  WriteLn;
  Read(f);
  IF f = 'V' THEN g.gf := vform;
                  WriteString('Gib Ortsvektor ein.');
                  WriteLn;
                  ReadReal(g.vf.place.abs);
                  ReadReal(g.vf.place.ord);
                  WriteString('Gib Richtungsvektor ein.');
                  WriteLn;
                  ReadReal(g.vf.dir.abs);
                  ReadReal(g.vf.dir.ord) END;
  IF f = 'N' THEN g.gf := nform;
                  WriteString('Gib a, b, c ein.');
                  WriteLn;
                  ReadReal(g.nf.a);
                  ReadReal(g.nf.b);
                  ReadReal(g.nf.c) END;
  IF g.gf = vform THEN res := g.vf.dir.ord = 0.0
                  ELSE res := g.nf.a = 0.0 END;
  IF res THEN WriteString('Parallel zur Abszisse')
         ELSE WriteString('Nicht parallel zur Abszisse') END;
  WriteLn
END Line .
```

Da Aufzählungstypen in Modula-2 nicht eingelesen und Wahrheitswerte nicht gedruckt werden können, liest dieses Programm entsprechende Zeichen `V` oder `N` ein und druckt einen das Resultat beschreibenden Text aus. ■

Nach den Feldern und Verbunden kommen wir nun zum letzten zusammengesetzten Typ von Modula-2, welcher auch der am schwierigsten zu verstehende ist. Allen Datentypen, die wir bisher kennengelernt haben, ist gemeinsam, daß für Objekte des Typs die Größe (d.h. der Umfang, den eine Darstellung im Speicher benötigt) aus dem Typ ersichtlich war. In vielen Fällen der Praxis ist aber nicht von vornherein bekannt, wieviele Daten zu verarbeiten sind. Damit muß man zu **dynamischen Datenstrukturen** wie Listen oder Bäumen übergehen. In der Sprache ML ist dies durch rekursive Typdeklarationen möglich. Im Prinzip muß man selbiges auch bei Modula-2 tun. Die Realisierung ist jedoch von der ML-Realisierung grundlegend verschieden, denn sie stützt sich wesentlich auf das in Abschnitt 6.1 als Grundlage skizzierte Maschinenmodell. Auf technische Einzelheiten, wie den Unterschied zwischen einer Speicherung im Kellerspeicher und auf der Halde werden wir bei der nachfolgenden Darstellung aber nicht eingehen, sondern versuchen, die Dinge möglichst abstrakt und maschinenunabhängig zu betrachten.

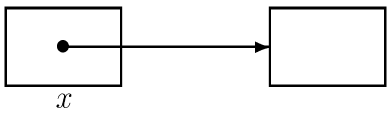
7.2.10 Zeiger

Ist m ein beliebiger Typ (in diesem Zusammenhang **Bezugstyp** genannt), so ist auch

`POINTER TO` m

ein Typ, genannt **Zeigertyp**. Ein Objekt eines solchen Typs (ein **Zeiger** oder **Pointer**) ist ein Verweis auf eine Programmvariable des Typs m, welche selbst keinen Namen besitzt (eine sogenannte **anonyme** Programmvariable ist). ■

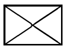
Im konkreten Speicher der von Neumann Architektur, wie er am Anfang dieses Kapitels bildlich skizziert wurde, ist der Verweis auf eine anonyme Programmvariable dadurch realisiert, daß der Wert des Zeigers die Adresse der der anonymen Programmvariablen entsprechenden Speicherzelle ist. Für das Programmieren mit Zeigern ist es jedoch sehr vorteilhaft, sich von den konkreten als Zahlen dargestellten Adressen freizumachen und stattdessen in graphischen Darstellungen wie



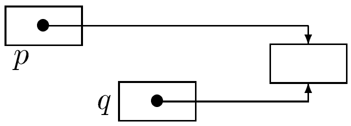
zu denken, wobei x eine Programmvariable für einen Zeiger ist und der Pfeil den Verweis (Inhalt) von x zu einer anderen Programmvariablen darstellt. Man spricht im Zusammenhang mit diesen Bildern auch von **Zeigergeflechten**. Programmieren mit Zeigern wird bei dieser abstrakten graphischen Auffassung zur Erstellung und Manipulation von Zeigergeflechten und Modula-2 bietet dazu geeignete Sprachkonstrukte an. Nachfolgend geben wir die durch die Einführung der Zeiger bewirkten Erweiterungen der Terme und Anweisungen der bisherigen Sprache an.

7.2.11 Erweiterungen des Term- und Anweisungsbegriffs

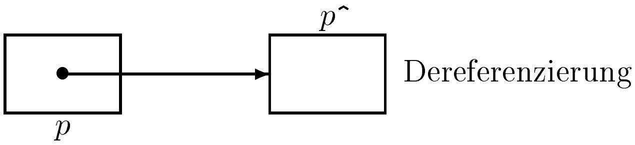
Im Gegensatz zu den Feldern und Verbunden gibt es bei den Zeigern eine Konstante, nämlich einen Zeiger, der auf keine Programmvariable verweist. Dieser Zeiger wird für alle Zeigertypen einheitlich mit dem Symbol `NIL` bezeichnet. Seine graphische Darstellung sieht üblicherweise wie folgt aus:



Als Operationen auf einem Zeigertyp `POINTER TO` m gibt es die **Gleichheit** `=` und die **Ungleichheit** `#`, beide wie üblich in Infix-Schreibweise notiert. Dabei heißen zwei Zeiger p und q gleich, wenn sie entweder beide `NIL` sind oder auf die gleiche Programmvariable zeigen, wie das nachstehende Bild andeutet:



Ist p ein Zeiger des Typs `POINTER TO` m, so ist p`^` die Programmvariable des Typs m, auf die p zeigt. Der Übergang von p zu p`^` heißt **Dereferenzierung** von p und sieht graphisch wie folgt aus:



Die Dereferenzierung von Zeigern stellt eine Operation auf Zeigern dar und kann somit beim Termaufbau verwendet werden. Ein Beispiel ist p`^ + 3`, falls der Bezugstyp gleich der Sorte `CARDINAL` ist. Falls der Bezugstyp eine Sorte ungleich `BOOLEAN` ist, so kann insbesondere p`^` durch eine der Lese- oder Drucke-Anweisungen von Abschnitt 6.3.1 eingelesen oder ausgegeben werden.

Bei Programmvariablen p eines Zeigertyps gibt es zwei Arten von Zuweisungen, nämlich p `:=` t_1, mit einem Term t_1 des Typs von p, und p`^ :=` t_2, mit einem Term t_2, dessen Typ der Bezugstyp des Zeigertyps ist. Man beachte, daß nach einer Zuweisung p `:=` q die Gleichheit p`^` $=$ q`^` gilt.

Ist p ein Term des Zeigertyps `POINTER TO` m, so ist `NEW(`p`)` eine Anweisung. Sie legt eine anonyme Programmvariable an und weist p als Wert einen Verweis auf diese anonyme Programmvariable zu (d.h. ihre Adresse). Wichtig für die praktische Programmierung ist die folgende Tatsache: In der Modula-2-Realisierung stützt sich `NEW(`p`)` auf eine weitere Anweisung `ALLOCATE(`p`,SIZE(`m`))` (die auf der Halde Speicher belegt). Deshalb ist in den Importteil des Programms zusätzlich die Zeile

```
FROM Storage IMPORT ALLOCATE;
```

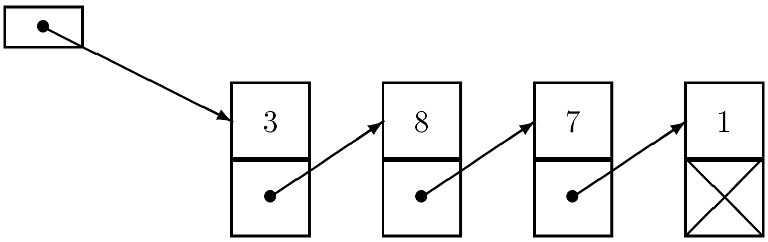
aufzunehmen. Streng genommen handelt es sich nämlich bei `NEW` und `ALLOCATE` um sogenannte Standardprozeduren aus dem vordefinierten Modul `Storage` und bei den Anweisungen `NEW(`p`)` und `ALLOCATE(`p`,SIZE(`m`))` um Prozeduraufrufe[27]. Das Prozedurkonzept wird aber erst später behandelt und deshalb haben wir bis zum jetzigen Stand des Skriptums die Prozeduraufrufe als (atomare) Anweisungen angesehen. ■

Auf das Gegenstück `DEALLOCATE` von `ALLOCATE` zum Freigeben von Speicherplatz auf der Halde gehen wir nicht näher ein. Wer sich intensiver mit der expliziten Speicherbelegung durch den Programmierer mittels `ALLOCATE` von `DEALLOCATE` beschäftigen will, muß auf Spezialliteratur zu Modula-2 zurückgreifen.

Die Mächtigkeit des Zeigerkonzepts zeigt sich darin, daß ein Zeiger auf eine Verbundvariable verweisen kann, die als Komponente(n) wiederum Zeiger enthalten kann. Damit ist es möglich, dynamische Datenstrukturen aufzubauen. Nachfolgend geben wir ein bekanntes Beispiel an.

7.2.12 Beispiel (Listen in Modula-2)

Im folgenden betrachten wir Listen natürlicher Zahlen. So eine Liste, etwa $\langle 3, 8, 7, 1\rangle$, kann man wie folgt graphisch als durch Zeiger dargestellt angeben:



In diesem Zusammenhang spricht man auch von **einfach verketteten Listen**, da von jedem nichtterminalen Listenelement genau ein Verweis zum nächsten Listenelement führt.

Eine einfach verkettete Liste ist also ein Zeiger auf eine Sequenz von Paaren, wobei von diesen jeweils eine Komponente die eigentliche Information trägt und die andere Komponente zum nächsten Paar verweist. Als Typdeklarationen zur Beschreibung solcher Listen darstellende Zeigergeflechte bieten sich die nachfolgend gegebenen an, in der Paare mittels eines Verbundtyps `listel` implementiert werden:

```
TYPE cardlist = POINTER TO listel;
     listel   = RECORD top: CARDINAL;
                       rest: cardlist
                END;
```

Durch diese Deklarationen ist der Zeigertyp `cardlist` rekursiv deklariert. Unserer bisherigen Philosophie nach dürfte zwischen den Deklarationen eigentlich kein Semikolon

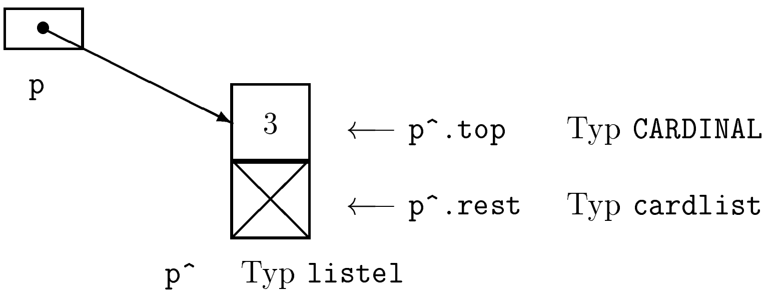
[27] Dies gilt auch für die in Abschnitt 6.3.1 aufgeführten Lese- und Drucke-Anweisungen. Auch bei ihnen handelt es sich eigentlich um Aufrufe von entsprechenden und durch das vorimplementierte Modul `InOut` zur Verfügung gestellten Prozeduren. Schließlich sind auch in den Anweisungen `DEC(`x`)` und `INC(`x`)` von Abschnitt 7.1.4 `DEC` und `INC` vorimplementierte Prozeduren

stehen, da in der ersten Zeile `listel` noch nicht deklariert ist. Da aber Zeiger auf Adressen zeigen, ist der für einen Zeiger auf `listel` benötigte Speicherplatzbedarf an dieser Stelle schon exakt bekannt, denn er ist für alle Zeiger gleich. Somit kann man entsprechend Speicherplatz reservieren (was in Modula-2 der eigentliche Zweck einer Deklaration ist), ohne den Bezugstyp `listel` zu kennen.

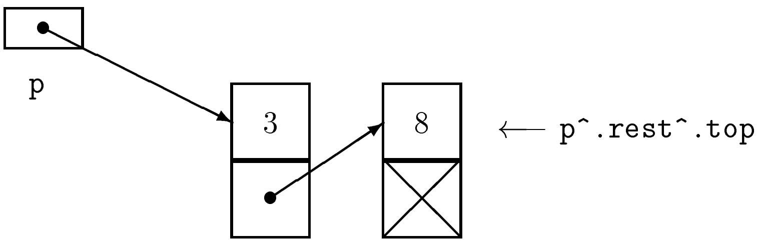
Es sei nun `p` eine Programmvariable vom Zeigertyp `cardlist`. Dann erzeugen die drei Anweisungen

```
NEW(p);
p^.top := 3;
p^.rest := NIL;
```

ein Zeigergeflecht, das die einelementige Liste $\langle 3 \rangle$ beschreibt und in der graphischen Darstellung wie nachfolgend angegeben aussieht:



Nun fügen wir an die bisher erstellte Liste $\langle 3 \rangle$ das Element 8 von rechts an, betrachten also die zweielementige Liste $\langle 3, 8 \rangle$. Das dieser Liste entsprechende Zeigergeflecht sieht wie folgt aus:



Um dieses Zeigergeflecht zu erstellen, haben wir die obige Anweisungssequenz wie folgt um drei Anweisungen zu ergänzen, die ein neues Paar schaffen und dann dessen zwei Komponenten entsprechend mit 8 bzw. `NIL` besetzen:

```
NEW(p);
p^.top := 3;
p^.rest := NIL;
NEW(p^.rest);
p^.rest^.top := 8;
p^.rest^.rest := NIL;
```

Für den Aufbau einer beliebigen Liste braucht man einen Hilfszeiger zum Anfügen, der z.B. immer auf das letzte Element oder das erste Element zeigt. Beispielsweise kann man mit Hilfe eines Hilfszeigers `h` des Typs `cardlist` an eine durch den Zeiger `p` beschriebene Liste ein Element mit Inhalt `x` von rechts wie folgt anfügen:

Programm:

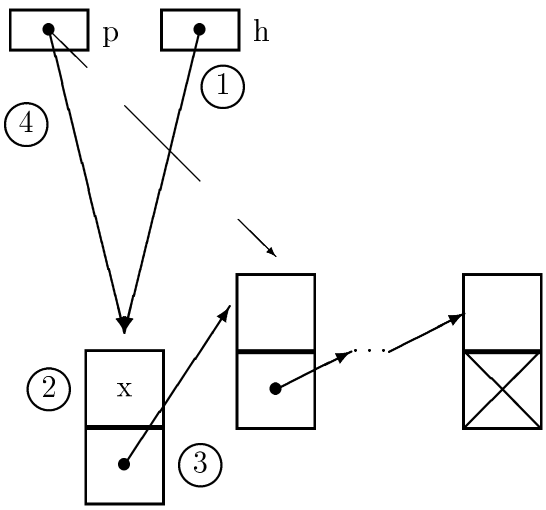
```
NEW(h);              1

h^.top := x;         2

h^.rest := p;        3

p := h               4
```

Graphische Darstellung:



In dieser bildlichen Darstellung der Programmabarbeitung entsprechen die auf der linken Seite angegebenen Marken auf der rechten Seite den Stellen, an denen das Zeigergeflecht geändert wird. Das rechte Bild zeigt das Zeigergeflecht nach Abarbeitung der vier Anweisungen, der alte Verweis von `p` im Zeigergeflecht ist mittels einer durchbrochenen Linie dargestellt. ■

Die Verwendung von Zeigern bei der Programmierung in Modula-2 hat viele Vorteile. Diese beinhalten insbesondere die nachfolgenden:

(1) Mittels der Zeigergeflechte hat man die Möglichkeit, Implementierungen für dynamische Datenstrukturen angeben zu können.

(2) Man spart Speicherplatz durch das Kopieren von Zeigern statt von Werten.

(3) Es ist von Vorteil, daß Objektgröße und -struktur bei der Zeigerweitergabe nicht bekannt zu sein brauchen.

(4) Es ist möglich, bei Rechenvorschriften beliebige Ergebnisse abzuliefern.

Der letzte Punkt wird später an der entsprechenden Stelle des Skriptums noch im Detail behandelt werden. Obigen Vorteilen stehen jedoch eine Reihe schwerwiegender Nachteile gegenüber, insbesondere die folgenden:

(1) Programme mit Zeigern sind schwer zu durchschauen und noch schwerer semantisch formal zu beschreiben und zu verifizieren.

(2) Bei Programmen mit Zeigern hat der Programmierer selbst dafür zu sorgen, daß in den aufgebauten Zeigergeflechten alle Programmvariablen tatsächlich erreicht werden und ggf. Speicherplatz wieder freigegeben wird.

(3) Beim Kopieren von Werten, auf die gezeigt wird, braucht man spezielle und häufig aufwendige Algorithmen.

Man kann die Auswirkungen der Nachteile mildern, indem man Zeiger durch geeignete Typdeklarationen und die Verwendung von Rechenvorschriften und Prozeduren (siehe später) möglichst verbirgt bzw. ganz lokal einsetzt. Wie dies geschieht, wird später bei den Algorithmen und Datenstrukturen anhand von Beispielen gezeigt. Vor einer allzu freien Verwendung von Zeigern bei der Programmierung ist dringend abzuraten!

7.3 Weitere Formen von Anweisungen

Im folgenden stellen wir drei weitere Formen von Anweisungen vor, die die bisherige Sprache erweitern. Dabei erklären wir ihre Wirkung durch die Reduktion auf schon besprochene Sprachkonstrukte. In der Literatur wird dieses Vorgehen zur Semantikdefinition manchmal auch **transformationelle Semantik** genannt.

7.3.1 Repeat-Anweisung (repeat-Schleife)

Sind s eine Anweisung und b ein Term der Sorte `BOOLEAN`, so ist auch die **repeat-Schleife**

$$\texttt{REPEAT } s \texttt{ UNTIL } b$$

mit **Schleifenbedingung** b und **Schleifenrumpf** s wiederum eine Anweisung. Die Semantik dieser Anweisung ist per Definition gleich der Semantik des Programmstücks

$$s\texttt{; WHILE NOT(}b\texttt{) DO } s \texttt{ END.}$$ ■

Die repeat-Schleife führt also eine Anweisung s solange aus, bis eine Bedingung b gilt. Man beachte, daß s mindestens einmal ausgeführt wird und die Abbruchbedingung erst durch die Abarbeitung des Schleifenrumpfs entstehen kann. Dies macht die Verwendung des Sprachkonstrukts `REPEAT` vorteilhaft für die Eingabe von Daten, birgt aber mitunter auch gewisse Gefahren. Bei der repeat-Schleife spricht man auch von einer nichtabweisenden Wiederholung, im Gegensatz zur while-Schleife, die man auch abweisende Wiederholung nennt.

7.3.2 Laufanweisung (for-Schleife)

Es sei m ein Typ, der einfach ist, oder eine Sorte ungleich `REAL` darstellt. Weiterhin seien a, e zwei Terme des Typs m und i eine Programmvariable des Typs m. Ist s eine Anweisung, **deren Ausführung den Wert von i nicht verändert**, so ist auch die **for-Schleife**

$$\texttt{FOR } i \texttt{ := } a \texttt{ TO } e \texttt{ DO } s \texttt{ END}$$

mit **unterer Grenze** a, **oberer Grenze** e, **Laufvariable** i und **Schleifenrumpf** s wiederum eine Anweisung. Ihre Semantik ist per Definition gleich der des Programmstücks

$$\begin{array}{l}
i \texttt{ := } a\texttt{;} \\
E \texttt{ := } e\texttt{;} \\
\texttt{WHILE } i\texttt{<}E \texttt{ DO } s\texttt{; INC(}i\texttt{) END;} \\
\texttt{IF } i\texttt{=}E \texttt{ THEN } s \texttt{ END},
\end{array}$$

wobei E eine neue Programmvariable des Typs m ist. ■

Innerhalb der Anweisung s wird die Laufvariable i nach der gemachten Einschränkung wie eine Konstante behandelt. Nach dem Verlassen der Schleife hat i den Wert von E, **was aber nicht ausgenutzt werden darf**, da dies nicht bei allen Modula-2-Übersetzern so ist. Der „offizielle" Modula-2-Sprachbericht legt sich hier nicht fest. Eine for-Schleife wird auch eine gezählte Wiederholung genannt, da die Anzahl der Schleifendurchläufe vor dem Schleifeneintritt schon feststeht. Sie ist ideal zum Arbeiten mit Feldern. Neben der obigen Form gibt es für m gleich `CARDINAL` oder `INTEGER` noch eine Erweiterung der for-Schleife zu

$$\texttt{FOR } i \texttt{ := } a \texttt{ TO } e \texttt{ BY } d \texttt{ DO } s \texttt{ END},$$

wobei der zusätzliche Term d die **Schrittweite** (Inkrement/Dekrement) ist. Auch diese allgemeinere Form kann mit Hilfe der while-Schleife erklärt werden, was hier aber nicht geschehen soll.

7.3.3 Fallunterscheidung (case-Anweisung)

Es sei m ein einfacher Typ oder eine Sorte ungleich `REAL`. Weiterhin seien t ein Term des Typs m, $c_i, 1 \leq i \leq k$, paarweise verschiedene Konstanten des Typs m und $s_i, 1 \leq i \leq k$, Anweisungen. Dann ist auch die **case-Anweisung**

$$\texttt{CASE } t \texttt{ OF } c_1 \texttt{ : } s_1 \texttt{ | } \ldots \texttt{ | } c_k \texttt{ : } s_k \texttt{ END}$$

eine Anweisung. Ihre Semantik ist per Definition gleich der der geschachtelten bewachten Anweisung

$$\begin{array}{rl}
\texttt{IF } t \texttt{ = } c_1 & \texttt{THEN } s_1 \\
\texttt{ELSE IF } t \texttt{ = } c_2 & \texttt{THEN } s_2 \\
\vdots & \quad\vdots \\
\texttt{ELSE IF } t \texttt{ = } c_k & \texttt{THEN } s_k \texttt{ END} \ldots \texttt{END}.
\end{array}$$

mit den k Wächtern $t = c_i$, $1 \leq i \leq k$, und den k Zweigen s_i, $1 \leq i \leq k$. ■

Mit Hilfe einer case-Anweisung können Verzweigungen, die nur vom Wert eines Terms t abhängen, oft kürzer und klarer formuliert werden. Es gibt noch Erweiterungen obiger Anweisung, etwa mit einem abschließenden `ELSE` s_{k+1} `END`, mit der Bedeutung, daß s_{k+1} ausgeführt wird, wenn $t \neq c_i$ für alle $1 \leq i \leq k$ gilt. An Stelle von Konstanten sind auch allgemeinere Konstruktionen möglich, dies soll hier aber nicht vertieft werden.

Neben der repeat- und der for-Schleife existiert in Modula-2 noch eine wesentlich allgemeinere loop-Schleife, auf die wir aber nicht eingehen. Auch die with-Anweisung, die vorteilhaft bei der Programmierung mit Verbunden ist, besprechen wir nicht. Bezüglich dieser Sprachkonstrukte sei auf die Modula-2 Literatur verwiesen.

Wir beenden diesen Abschnitt mit einem Programmbeispiel, in dem die nichtabweisende Schleife zur Eingabe verwendet wird.

7.3.4 Beispiel (Maximumssuche)

In Beispiel 7.2.5 entwickelten wir ein Modula-2-Programm, welches eine fixe Anzahl von natürlichen Zahlen einliest und deren Maximum berechnet. Mit Hilfe von Zeigern kann man sich nun von der Beschränkung freimachen, daß der Programmtext die fixe Anzahl der einzulesenden Zahlen, wie 20 in Beispiel 7.2.5, festlegt. Nachfolgend ist ein entsprechendes Modula-2-Programm angegeben:

```
MODULE List;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadCard, WriteCard;
FROM Storage IMPORT ALLOCATE;
TYPE cardlist = POINTER TO listel;
     listel   = RECORD top : CARDINAL; rest : cardlist END;
VAR s, t : cardlist;
    k, res : CARDINAL;
BEGIN
  WriteString('Gib Liste positiver natuerlicher Zahlen ein');
  WriteString('(mit 0 als Abschlusszeichen).');
  WriteLn;
  s := NIL;
  REPEAT ReadCard(k);
         IF k # 0 THEN NEW(t);
                       t^.rest := s;
                       t^.top := k;
                       s := t
                       END
  UNTIL k = 0;
  res := 0;
  WHILE s # NIL DO
    IF res < s^.top THEN res := s^.top END;
    s := s^.rest
    END;
  WriteString('Maximum: ');
  WriteCard(res,10);
  WriteLn
END List .
```

Dieses Programm verwendet eine repeat-Schleife zur Eingabe. Der Einfachheit halber liest es nur positive Zahlen in die Liste `s` ein, deren Maximum nach Abarbeitung der Eingabeschleife durch die Initialisierung `res := 0` und die nachfolgende while-Schleife bestimmt wird. Die Zahl `0` wird als Abschlußzeichen der Eingabe aufgefaßt. ■

7.4 Prozeduren

Funktionale Sprachen wie ML basieren primär auf einem Termbegriff und beinhalten das Sprachkonstrukt der Rechenvorschrift zur **Parametrisierung von Termen**. Imperative Sprachen wie Modula-2 bauen hingegen auf dem Begriff der Anweisung auf. Auch dafür steht ein Sprachmittel zur **Parametrisierung** zur Verfügung, die **Prozedur**. Wegen der Wichtigkeit dieses Sprachkonstrukts spricht man statt von imperativen auch von **prozeduralen Sprachen**. Wir beginnen die Einführung des Prozedurbegriffs mit zwei motivierenden Beispielen.

7.4.1 Beispiel (zur Motivation)

Das Skalarprodukt von zwei Vektoren $a, b \in \mathbb{R}^n$ ist definiert als die Summe der Produkte der Komponenten:

$$a * b = a_1 * b_1 + \ldots + a_n * b_n = \sum_{i=1}^{n} a_i * b_i \qquad (*)$$

Es werde nun die Aufgabe gestellt, ein Modula-2-Programm zur Berechnung des Skalarprodukts zu schreiben. Dazu sei die Länge der beiden Vektoren als Modula-2-Konstante `n` vorgegeben.

Für die beiden bei der Skalarproduktbildung beteiligten Vektoren bietet sich jeweils eine Darstellung in Form eines eindimensionalen Felds der Länge `n` an. Dies führt zu den nachstehenden Deklarationen im Deklarationsteil des zu erstellenden Programms:

```
TYPE vector = ARRAY [1..n] OF REAL;
VAR a, b : vector;
```

Nun behandeln wir den Programmrumpf. Hier sind zuerst die Werte der Vektoren in die entsprechenden Felder `a` und `b` einzulesen. Dazu kann man beispielsweise zwei for-Schleifen mit einer gemeinsamen (im Deklarationsteil noch einzuführenden) Laufvariablen `i` des Bereichstyps `[1..n]` wie folgt verwenden:

```
FOR i := 1 TO n DO
  ReadReal(a[i]) END;
FOR i := 1 TO n DO
  ReadReal(b[i]) END;
```

Schließlich ist, etwa ebenfalls mittels einer for-Schleife mit der Laufvariablen `i`, das Skalarprodukt $(*)$ in einer Programmvariablen zu berechnen und auszugeben. Wählen wir `res` als Namen für die das Resultat beinhaltende Programmvariable, so führt dies zu dem folgenden Programmstück:

```
res := 0;
FOR i := 1 TO n DO
  res := res + a[i] * b[i] END;
WriteReal(res,10,4);
```

Offensichtlich ist $\texttt{res} = \sum_{j=1}^{\texttt{i}-1} \texttt{a[}j\texttt{]} * \texttt{b[}j\texttt{]}$ die Invariante derjenigen while-Schleife zur Berechnung des Skalarproduktes, welche durch die definierende Transformation von 7.3.2 aus der for-Schleife entsteht.

Betrachtet man nun die zwei for-Schleifen zur Eingabe, so stellt man fest, daß sie bis auf die Namen der Felder identisch sind. Es bietet sich also an, sie nach den Feldern zu parametrisieren, sowie für dieses Konstrukt – **Prozedur** genannt – einen frei wählbaren Bezeichner als Namen zu vergeben. In der Syntax von Modula-2 erhalten wir beispielsweise zum Einlesen von Feldern des Typs `vector` die folgende Prozedur:

```
PROCEDURE ReadVector (VAR v : vector);
  VAR i : CARDINAL;
  BEGIN
  FOR i := 1 TO n DO
    ReadReal(v[i]) END
  END ReadVector;
```

Mit Hilfe von `ReadVector` kann man nun die zwei Schleifen zum Einlesen der Felder `a` und `b` durch die zwei Prozeduraufrufe

```
ReadVector(a);
ReadVector(b);
```

ersetzen. Dieses Beispiel demonstriert auch, daß Aufrufe von Prozeduren den Anweisungsbegriff betreffen, was wir an früherer Stelle schon einmal bemerkt haben. ■

Die Prozedur `ReadVector` aus Beispiel 7.4.1 besitzt einen Parameter `v`, der für eine Programmvariable des Typs `vector` steht. Dies wird durch das Voranstellen des Wortsymbols `VAR` angezeigt. Solche Parameter nennt man **Variablenparameter**. Oft ist es aber auch sinnvoll, neben Variablenparametern in Prozeduren auch Parameter für Terme/Objekte analog zu den funktionalen Sprachen einzuführen. Wir motivieren dies nachfolgend ebenfalls am Beispiel des Skalarprodukts.

7.4.2 Beispiel (Objektparameter bei Prozeduren)

In Modula-2 sind Felder statisch durch Konstanten im Programmtext festgelegt. Durch die Programmstücke von Beispiel 7.4.1 wird genau das Skalarprodukt von zwei durch Felder implementierte Vektoren der Länge `n` berechnet; sie können nicht für andere Dimensionen verwendet werden. Eine gewisse Abhilfe ist möglich, indem man Speicherplatz opfert und im Programm die Konstante `n` symbolisch einführt[28] und dabei groß genug wählt. Dann kann man für zwei Vektoren $a, b \in \mathbb{R}^k$ mit $k \leq$ `n` das Skalarprodukt berechnen, indem man k als zusätzlichen Parameter mitführt. Die neue Eingabeprozedur für das Feld `v` und die Vektorlänge `k` ist nachfolgend angegeben:

```
PROCEDURE ReadVector (VAR v : vector; k : CARDINAL);
  VAR i : CARDINAL;
  BEGIN
  FOR i := 1 TO k DO
    ReadReal(v[i]) END
  END ReadVector;
```

Als Prozedur sieht die Berechnung des Skalarprodukts von zwei Vektoren, die durch die ersten `k` Elemente der Felder `v` und `w` des Feldtyps `vector` dargestellt werden, und die Abspeicherung des Resultats in der Programmvariablen `res` wie folgt aus:

```
PROCEDURE sp (v, w : vector; k : CARDINAL; VAR res : REAL);
  VAR i : CARDINAL;
  BEGIN
  res := 0;
  FOR i := 1 TO k DO
    res := res + v[i] * w[i] END
  END sp;
```

Offensichtlich entstehen die beiden Prozeduren durch geeignete Parametrisierungen der Programmstücke aus Beispiel 7.4.1. In ihnen wird die Tatsache, daß `k` ein Parameter für einen Term (hier der Sorte `CARDINAL`) ist, durch das Fehlen des Wortsymbols `VAR` angezeigt. Solche Parameter heißen **Objektparameter**, **Wertparameter** oder **gewöhnliche Parameter**.

In der Prozedur `sp` sind auch die Felder `v` und `w` als Objektparameter vereinbart. Dies ist möglich, da die Abarbeitung der for-Schleife weder `v` noch `w` verändert. Hingegen ist eine Vereinbarung des Parameters `v` in der Lese-Prozedur `ReadVector` als Objektparameter nicht erlaubt, da hier die Abarbeitung der entsprechenden for-Schleife den Inhalt verändert. Ebenso ist es nicht erlaubt, in `sp` den Parameter `res` als Objektparameter zu deklarieren. Man beachte schließlich noch, daß Aufrufe von `ReadVector` und `sp` nur mit einem Wert für `k` erfolgen dürfen, der kleiner oder gleich dem Wert von `n` ist. ■

Nach diesen einführenden Beispielen beschäftigen wir uns nun allgemein mit dem Prozedurbegriff. Damit **erweitert sich natürlich der Deklarationsteil** unserer bisherigen Sprache, indem wir auch Prozedurdeklarationen zulassen.

[28]Die symbolische Einführung von Konstanten einer Sorte erfolgt im Deklarationsteil nach dem Wortsymbol `CONST`. Ein Beispiel ist `CONST n = 25` für die Konstante `n` der Sorte `CARDINAL`. Symbolische Konstanten sind spezielle Terme, die insbesondere das Korrigieren und Ändern von Programmen erleichtern. Etwa bietet es sich für das Programm aus Beispiel 7.2.5 an, am Anfang für die obere Feldgrenze 20 eine symbolische Konstante `N` mittels `CONST N = 20` einzuführen und diese an Stelle der Zahl 20 im Restprogramm zu verwenden. Dann ist eine Abänderung für andere Feldlängen sehr einfach möglich.

7.4.3 Prozedurdeklarationen und -aufrufe

Eine **Prozedurdeklaration** in der Programmiersprache Modula-2 besteht im wesentlichen aus vier Teilen, nämlich

a) dem **Namen**, der frei wählbar ist,

b) der **Liste der (formalen) Parameter** mit den Typangaben und der Kennzeichnung der Variablenparameter mittels des Wortsymbols `VAR`,

c) dem **Deklarationsteil** zur Deklaration der lokalen Typen, Programmvariablen und wiederum Prozeduren, sowie

d) dem **Rumpf**, auch **Anweisungsteil** oder **Block** genannt, wiederum gefolgt von dem Namen und dem Semikolon als Abschlußzeichen.

Die etwas detailliertere Syntax einer Prozedurdeklaration ist

$$\begin{array}{l}\texttt{PROCEDURE} \gg \mathit{Name} \ll \texttt{(} \gg \mathit{Liste\ der\ Parameter} \ll \texttt{);} \\ \quad \gg \mathit{Deklaration\ der\ lokalen\ Dinge} \ll \texttt{;} \\ \texttt{BEGIN}\ s\ \texttt{END} \gg \mathit{Name} \ll \texttt{;}\end{array}$$

mit Programmvariablen- und Typdeklarationen wie bisher eingeführt und s als Anweisung. Als Programmvariablen in s dürfen verwendet werden die lokal deklarierten Programmvariablen, die Variablenparameter, die Programmvariablen und Variablenparameter der umgebenden Prozeduren und die Programmvariablen des gesamten Programms.

Durch die Möglichkeit, Prozeduren zu deklarieren, verändert sich nicht nur der Deklarationsteil der Sprache. Da Prozeduren Anweisungen parametrisieren, legen wir fest, daß **Aufrufe**

$$P\texttt{(} \gg \mathit{Liste\ der\ Argumente} \ll \texttt{)}$$

von Prozeduren P mit – von der Typisierung her passenden – Programmvariablen auf den Variablenparameterpositionen und Termen auf den Objektparameterpositionen ebenfalls Anweisungen sind. Man beachte, daß es durch diese Festlegung möglich ist, daß im Rumpf einer Prozedur P ein Aufruf von P vorkommt. Man nennt P dann eine **rekursive Prozedur**. ■

Auf die Semantik der Prozeduraufrufe, d.h. die Mechanismen zur Parameterübergabe, wird nun genauer eingegangen. Die Vorgehensweise im Fall einer einzelnen Prozedur P ist dabei wie folgt: Wir reduzieren ein Programm mit Aufrufen von P durch eine definierende Transformation schrittweise auf ein Programm ohne Aufrufe von P, womit wir dann auch die Deklaration von P aus dem Deklarationsteil entfernen dürfen. Wiederholte Anwendungen dieser Vorgehensweise erlauben somit die Reduktion eines beliebigen Programms mit (terminierenden) Prozeduren auf ein prozedurloses Programm, dessen Semantik wir bereits (im Fall der while-Programme sogar formal) festgelegt haben. Die definierende Transformation der Reduktion, Kopierregel genannt, wird nachfolgend beschrieben.

7.4.4 Die Kopierregel

Es sei die folgende Prozedur P im Deklarationsteil eines Modula-2-Programms oder einer anderen Prozedur Q vereinbart:

$$\begin{array}{ll} \texttt{PROCEDURE}\ P\ \texttt{(VAR}\ x_1\ \texttt{:}\ m_1\texttt{;}\dots\texttt{;}\ \texttt{VAR}\ x_k\ \texttt{:}\ m_k\texttt{;}\ y_1\ \texttt{:}\ n_1\texttt{;}\dots\texttt{;}\ y_l\ \texttt{:}\ n_l\texttt{);} & \\ \quad \gg \textit{Deklaration der lokalen Dinge} \ll\texttt{;} & (P) \\ \texttt{BEGIN}\ s\ \texttt{END}\ P\texttt{;} & \end{array}$$

Wir haben dabei zur Vereinfachung der nachfolgenden Ersetzung die Variablenparameter x_i, $1 \leq i \leq k$, und Objektparameter y_j, $1 \leq j \leq l$, von P zu zwei Listen zusammengefaßt; normalerweise dürfen sie auch gemischt auftreten.

Nun betrachten wir im Rumpf des Programms oder der Prozedur Q einen Prozeduraufruf der Form

$$P(z_1,\dots,z_k,t_1,\dots,t_l) \qquad (*)$$

mit (deklarierten) Programmvariablen z_i des Typs m_i, $1 \leq i \leq k$, und Termen t_j des Typs n_j, $1 \leq j \leq l$. Eine **Anwendung der Kopierregel** verändert das vorliegende Modula-2-Programm bzw. die die Deklaration von P enthaltende Prozedur Q wie folgt:

a) Sie führt zuerst im Deklarationsteil von Q neue Programmvariablen y'_j des Typs n_j ein, $1 \leq j \leq l$.

b) Sie deklariert dann die lokal in P deklarierten Typen, Programmvariablen und Prozeduren ebenfalls im Deklarationsteil von Q. Sollten sich dabei Bezeichnungskonflikte ergeben, so sind diese durch eine vorherige **konsistente Umbenennung** innerhalb von P aufzulösen.

c) Sie ersetzt schließlich den Aufruf $(*)$ in Q durch die Anweisungen

$$y'_1\ \texttt{:=}\ t_1\texttt{;}\dots\texttt{;}\ y'_l\ \texttt{:=}\ t_l\texttt{;}\ \hat{s},$$

wobei die Anweisung $\hat{s}$ aus dem Rumpf s von P entsteht, indem alle x_i zu z_i und alle y_j zu y'_j umbenannt werden.

Kommen im Aufruf $(*)$ indizierte Programmvariablen $a\texttt{[}i\texttt{]}$ auf Variablenparameterposition vor, so sind vor der Anwendung von a) bis c) zuerst die Werte der Indexterme i mittels ebenfalls neu in Q deklarierten Programmvariablen i' und sich unmittelbar vor $(*)$ befindlichen neuen Zuweisungen $i'\ \texttt{:=}\ i$ auszuwerten und dann noch die indizierten Programmvariablen $a\texttt{[}i\texttt{]}$ in $(*)$ durch $a\texttt{[}i'\texttt{]}$ zu ersetzen. ■

Nach der Festlegung der Kopierregel-Ersetzung in Abschnitt 7.4.4 hat man also zwei Mechanismen zur Parameterübergabe bei Prozeduren in Modula-2: Bei Objektparametern wird beim Aufruf erst der Term ausgewertet und der Wert abgespeichert. Dann wird mit diesem Wert der Rumpf ausgeführt. Dies ist genau der **Call-by-value**-Mechanismus, wie wir ihn schon von der funktionalen Programmierung her kennen. Bei Variablenparametern wird der Name der Programmvariablen des Aufrufs direkt im Rumpf verwendet, d.h. der formale Parameter an die Speicherzelle der Aufrufvariablen gebunden. Man nennt dies in der Literatur den **Call-by-reference**-Mechanismus.

Man beachte, daß die eben getroffene Festlegung der Parameterübergabe per Kopierregel-Ersetzung nicht für alle Programmiersprachen gilt. Zwei weitere Aufruf-Mechanismen bei Prozeduren sind beispielsweise Call-by-name für Objektparameter und Call-by-value-result für Variablenparameter.

Im nachfolgenden Beispiel greifen wir die Berechnung des Skalarprodukts noch einmal auf und demonstrieren damit die Anwendung der Kopierregel.

7.4.5 Beispiel (Nichtrekursive Prozeduraufrufe)

Wir betrachten das folgende Modula-2-Programm `Scalarproduct` zur Berechnung des Skalarprodukts von zwei durch Felder `a` und `b` implementierte Vektoren, wobei wir sowohl für die Eingabe der Felder als auch zur Berechnung des Skalarprodukts Prozeduren und Prozeduraufrufe verwenden:

```
MODULE Scalarproduct;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
CONST n = 20;
TYPE vector = ARRAY [1..n] OF REAL;
VAR a, b : vector;
    res : REAL;
PROCEDURE ReadVector (VAR v : vector; k : CARDINAL);
  VAR i : CARDINAL;
  BEGIN
  FOR i := 1 TO k DO
    ReadReal(v[i]) END
  END ReadVector;
PROCEDURE sp (v, w : vector; k : CARDINAL; VAR res : REAL);
  VAR i : CARDINAL;
  BEGIN
  res := 0;
  FOR i := 1 TO k DO
    res := res + v[i] * w[i] END
 END sp;
BEGIN
  ReadVector(a,n);
  ReadVector(b,n);
  sp(a,b,n,res);
  WriteReal(res,10,4);
  WriteLn
END Scalarproduct .
```

Nun wenden wir die Kopierregel auf alle drei Prozeduraufrufe in diesem Programm an und erhalten als Resultat das nachfolgende Programm, in dem kein Aufruf der deklarierten Prozeduren mehr im Rumpf vorkommt:

```
MODULE Scalarproduct;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
CONST n = 20;
TYPE vector = ARRAY [1..n] OF REAL;
VAR a, b, vv, ww : vector;
    res : REAL;
    i, kk : CARDINAL;
PROCEDURE ReadVector (VAR v : vector; k : CARDINAL);
  VAR i : CARDINAL;
  BEGIN
  FOR i := 1 TO k DO
    ReadReal(v[i]) END
  END ReadVector;
PROCEDURE sp (v, w : vector; k : CARDINAL; VAR res : REAL);
  VAR i : CARDINAL;
  BEGIN
  res := 0;
  FOR i := 1 TO k DO
    res := res + v[i] * w[i] END
  END sp;
BEGIN
  kk := n;
  FOR i := 1 TO kk DO
    ReadReal(a[i]) END;
  kk := n;
  FOR i := 1 TO kk DO
    ReadReal(b[i]) END;
  vv := a;
  ww := b;
  kk := n;
  res := 0;
  FOR i := 1 TO kk DO
    res := res + vv[i] * ww[i] END;
  WriteReal(res,10,4);
  WriteLn
END Scalarproduct .
```

In diesem Modula-2-Programm sind die neuen, den Objektparametern `v`, `w` und `k` der Prozeduren `ReadVector` und `sp` entsprechenden, Programmvariablen im Hauptprogramm jeweils durch Doppelbuchstaben bezeichnet, wobei wir die Programmvariable `kk` nur einmal einführen. Auch die lokale Programmvariable `i` von sowohl `ReadVector` als auch `sp` wird im Deklarationsteil von `Scalarproduct` nur einmal aufgeführt.

Der Rumpf des Programms `Scalarproduct` enthält nun keinen Prozeduraufruf mehr und damit ist es erlaubt, die Deklarationen von `ReadVector` und `sp` aus dem Deklarationsteil zu entfernen. Das Resultat sieht wie folgt aus, wenn wir auch die beiden überflüssigen Vorkommen von `kk := n` entfernen:

```
MODULE Scalarproduct;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
CONST n = 20;
TYPE vector = ARRAY [1..n] OF REAL;
VAR a, b, vv, ww : vector;
    res : REAL;
    i, kk : CARDINAL;
BEGIN
  kk := n;
  FOR i := 1 TO kk DO
    ReadReal(a[i]) END;
  FOR i := 1 TO kk DO
    ReadReal(b[i]) END;
  vv := a;
  ww := b;
  res := 0;
  FOR i := 1 TO kk DO
    res := res + vv[i] * ww[i] END;
  WriteReal(res,10,4);
  WriteLn
END Scalarproduct .
```

Es ist klar, daß die Effizienz dieses Programms noch gesteigert werden kann, indem man auf die „Zwischenfelder“ `vv` und `ww` verzichtet und direkt mit `a` und `b` arbeitet. Dies ist der Fall, wenn man `v` und `w` in der Prozedur `sp` als Variablenparameter festlegt. Allgemein sollten die Programmvariablen für zusammengesetzte Datenstrukturen bei Prozeduren immer als Variablenparameter deklariert sein, auch wenn sie durch die Prozedurrümpfe nicht verändert werden. Dadurch vermeidet man ihr Kopieren beim Aufruf. Wegen möglicher Seiteneffekte erfordert ein solches Vorgehen natürlich eine besondere Sorgfalt. ■

Die oben gezeigte Berechnung entspricht in ihrer Erweiterung auf rekursive Prozeduren genau dem Vorgehen, wie es bei der funktionalen Programmiersprache ML bei den rekursiven Rechenvorschriften durch die Termersetzungssemantik eingeführt wurde. Im nächsten Abschnitt geben wir an, wie man ein Programm mit einer eventuell rekursiven Prozedur durch die Kopierregel schrittweise auf ein Programm ohne diese Prozedur reduzieren kann.

7.4.6 Kopierregel-Semantik für (auch rekursive) Prozeduren

Wir setzen voraus, daß eine Prozedur P der Gestalt (P), Abschnitt 7.4.4, in einem Programm oder in einer Prozedur Q deklariert ist. Ausgehend von Q wird eine Folge $Q_0, Q_1, \ldots$ von Programmen bzw. Prozeduren wie folgt konstruiert:

a) Starte mit Q_0 als Q.

b) Bilde das Programm bzw. die Prozedur Q_{i+1} aus dem Vorgänger Q_i in zwei Schritten:

   (1) Ersetze in Q_i jeden Aufruf von P gemäß der Kopierregel 7.4.4 und modifiziere das Programm bzw. die Prozedur entsprechend.

   (2) Vereinfache das / die durch (1) entstehende Programm / Prozedur soweit wie möglich.

Ist die so konstruierte Folge endlich und Q_N das letzte Folgenglied, so ist die Semantik des Programms bzw. der Prozedur Q gleich der Semantik von Q_N, wobei P aus dem Deklarationsteil entfernt ist. Ansonsten ist die Semantik von Q undefiniert. ■

Aufbauend auf die Kopierregel-Semantik kann man jedes Modula-2-Programm, in dem jeder Aufruf einer rekursiven Prozedur terminiert, auf ein Programm ohne Prozeduren reduzieren, falls man statt der Lese-Anweisungen entsprechende Initialisierungen im Programmtext verwendet. Im nächsten Beispiel demonstrieren wir dies.

7.4.7 Beispiel (Rekursive Prozeduraufrufe)

Wir betrachten ausschnittsweise ein Modula-2-Programm:

```
    ...
VAR a, b : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN
  IF n = 0 THEN SKIP
           ELSE P(x,n - 1); x := x + a END
  END P;
    ...
P(b,2);
    ...
```

Nun berechnen wir die Bedeutung des Aufrufs `P(b,2)`, indem wir abwechselnd die Kopierregel-Semantik verwenden und anschließend vereinfachen. Die neuen Programmvariablen bezeichnen wir dabei durch nachgestellte Zahlen. Wir starten mit dem Aufruf `P(b,2)` im Rumpf und bekommen durch die Anwendung der Kopierregel:

```
    ...
VAR a, b, y1 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
IF y1 = 0 THEN SKIP
          ELSE P(b,y1 - 1); b := b + a END;
    ...
```

Nun vereinfachen wir die bewachte Anweisung im Programmrumpf zum else-Fall und den zweiten Parameter `y1 - 1` des Prozeduraufrufs `P(b,y1 - 1)` zu `1`. Diese Vereinfachung verwendet (wie auch alle nachfolgenden) nur offensichtliche Eigenschaften der bisher vorhandenen Sprachkonstrukte, wie etwa Termvereinfachung oder Einsetzen von Programmvariablenwerten, welche auch mittels der bisherigen Semantik beweisbar sind. Wesentlich ist, daß sie nicht auf Prozeduren Bezug nimmt. Das Resultat der Vereinfachung sieht wie folgt aus:

```
    ...
VAR a, b, y1 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
P(b,1);
b := b + a;
    ...
```

Der nächste Schritt besteht wiederum in der Anwendung der Kopierregel, nun auf den Prozeduraufruf `P(b,1)` im Rumpf des Programms. Wir erhalten dadurch das nachstehende Resultat:

```
    ...
VAR a, b, y1, y2 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
y2 := 1;
IF y2 = 0 THEN SKIP
          ELSE P(b,y2 - 1); b := b + a END;
b := b + a;
    ...
```

Wie nach der ersten Anwendung der Kopierregel, so vereinfachen wir nun wiederum die bewachte Anweisung des Programmrumpfs zum else-Fall und den zweiten Parameter `y2 - 1` des Prozeduraufrufs `P(b,y2 - 1)` zu `0`. Dies bringt:

```
    ...
VAR a, b, y1, y2 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
y2 := 1;
P(b,0);
b := b + a;
b := b + a;
    ...
```

Nun sind wir fast fertig. Eine letztmalige Anwendung der Kopierregel auf den Prozeduraufruf `P(b,0)` liefert schließlich

```
    ...
VAR a, b, y1, y2, y3 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
y2 := 1;
y3 := 0;
IF y3 = 0 THEN SKIP
          ELSE P(b,y3 - 1); b := b + a END;
b := b + a;
b := b + a;
    ...
```

was sich zu dem Programmausschnitt

```
    ...
VAR a, b, y1, y2, y3 : CARDINAL;
PROCEDURE P (VAR x : CARDINAL; n : CARDINAL);
  BEGIN ... END P;
    ...
y1 := 2;
y2 := 1;
y3 := 0;
b := b + a;
b := b + a;
    ...
```

vereinfacht. Somit entspricht der Originalaufruf `P(b,2)` der Zuweisung `b := b + a + a`, wobei zusätzlich drei Hilfsspeicherplätze verwendet werden, die durch `y1`, `y2` und `y3` bezeichnet sind. Offensichtlich wirkt für ein beliebiges `n` der Prozeduraufruf `P(b,n)` wie die Zuweisung `b := b +`$\sum_{i=1}^{n}$ `a`. ■

Da in Modula-2-Prozeduren wiederum Prozeduren deklariert werden dürfen, haben wir damit, analog zur funktionalen Programmiersprache ML, die Frage zu klären, was mit Bezeichnungen zu geschehen hat, die in einer Prozedur auftreten, dort aber weder lokal deklariert noch formale Parameter sind. Es stellt sich also wiederum das Problem der Bindung globaler Bezeichnungen. Um dies zu lösen, gehen wir analog zu ML vor; man vergleiche mit Abschnitt 5.5.

7.4.8 Blockstruktur

Der **Blockstrukturbaum** eines Modula-2-Programms ist wie folgt definiert:

a) Für das gesamte Programm und für jede deklarierte Prozedur gibt es einen Knoten, der mit dem Namen des Programms bzw. der Prozedur markiert ist.

b) Es wird eine Kante von einem P-Knoten zu einem Q-Knoten gezogen, falls Q im Deklarationsteil von P deklariert wird.

P heißt **statischer Vorgänger** von Q, falls es einen Kantenzug von P zu Q im Blockstrukturbaum gibt. ■

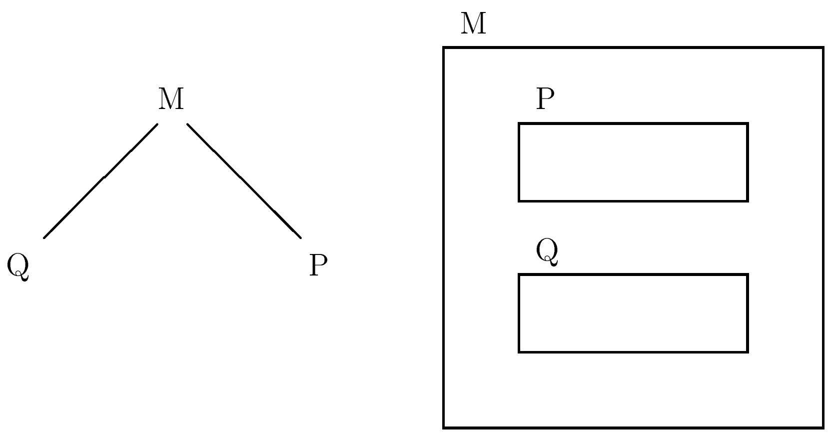
Offensichtlich ist das gesamte Programm statischer Vorgänger aller in ihm deklarierten Prozeduren. Der Blockstrukturbaum ist das imperative Gegenstück zum applikativen Abschnittsstrukturbaum; siehe Beispiel 5.5.4. Wie dort verwendet man zur Darstellung der Blockstruktur auch geschlossene, ineinandergeschachtelte Kästchen.

7.4.9 Beispiel (zur Blockstruktur)

Wir betrachten das folgende einfache Modula-2-Programm mit zwei parameterlosen Prozeduren, in dem die Prozedur `P` eine globale Programmvariable `x` enthält:

```
MODULE M;
FROM InOut IMPORT WriteLn, WriteInt;
VAR x : INTEGER;
PROCEDURE P;
  BEGIN x := x + 1 END P;
PROCEDURE Q;
  VAR x : INTEGER;
  BEGIN x := 1; P; WriteInt(x,5); WriteLn END Q;
BEGIN x := 3; Q END M .
```

Nachfolgend findet man links den Blockstrukturbaum und rechts die Darstellung der Blockstruktur durch ineinandergeschachtelte Rechtecke:



Der statische Vorgänger der Prozedur `P` ist das gesamte Programm `M`. Gleiches gilt für die Prozedur `Q`. Das gesamte Programm `M` hat keinen statischen Vorgänger. Dabei haben wir, wie bisher immer im Skriptum, bei der zeichnerischen Darstellung von Bäumen die Kantenrichtung von oben nach unten angenommen. ■

Analog zu ML haben wir als Konsequenz der Kopierregel-Semantik die folgende Festlegung der Bindung: Die Programmiersprache **Modula-2 hat statische** Bindung. Ist x eine globale Bezeichnung einer Prozedur, so nimmt man bei der Ausführung den Wert desjenigen x, welches im ersten statischen Vorgänger (lokal deklariert oder als Parameter) eingeführt wird, den man in Richtung gesamtes Programm im Blockstrukturbaum erreicht.

Im Beispiel 7.4.9 erhöht der Aufruf von `P` also den Wert der ersten Deklaration von `x` um 1, und das Programm `M` druckt somit die Zahl 1. Dieses Resultat bekommt man formal auch durch die Kopierregel-Semantik.

7.4.10 Beispiel (Weiterführung von 7.4.9)

Die Anwendungen der Kopierregel erfolgen von innen nach außen. Im ersten Schritt haben wir somit den Aufruf von `Q` im Rumpf von `M` zu ersetzen, wobei wir die lokale Programmvariable `x` von `Q` auch im Deklarationsteil von `M` zu vereinbaren haben. Da aber `x` dort schon deklariert ist, müssen wir vorher konsistent in `Q` umbenennen, etwa wie folgt:

```
PROCEDURE Q;
  VAR xQ : INTEGER;
  BEGIN xQ := 1; P; WriteInt(xQ,5); WriteLn END Q;
```

Nach Anwendung der Kopierregel und Entfernung von `Q` erhalten wir dann das folgende Programm:

```
MODULE M;
FROM InOut IMPORT WriteLn, WriteInt;
VAR x, xQ : INTEGER;
PROCEDURE P;
  BEGIN x := x + 1 END P;
BEGIN
  x := 3;
  xQ := 1; P; WriteInt(xQ,5); WriteLn
END M .
```

Es verbleibt noch die Anwendung der Kopierregel auf den Aufruf von `P`. Sie ersetzt `P` durch `x := x + 1` und wir bekommen schließlich nach dem Entfernen von `P`:

```
MODULE M;
FROM InOut IMPORT WriteLn, WriteInt;
VAR x, xQ : INTEGER;
BEGIN
  x := 3;
  xQ := 1; x := x + 1; WriteInt(xQ,5); WriteLn
END M .
```

Offensichtlich druckt dieses Modula-2-Programm die Zahl 1. ■

Statt den Blockstrukturbaum zu verwenden, kann man statische Bindung, wie bei ML, auch durch Gültigkeitsbereiche beschreiben. Man findet dies etwa in dem in der Einleitung erwähnten Buch von H.J. Appelrath und J. Ludewig in Abschnitt 2.3.2.

Bei der Variante „dynamische Bindung“, die wir auch bei ML erwähnten, würde in dem obigen Program die Prozedur `P` den Wert der Deklaration der Programmvariablen `x` in `Q` nehmen, da `P` in `Q` aufgerufen wird[29]. Also würde `P` das `Q`-lokale `x` um 1 erhöhen, und das Resultat wäre somit das Ausdrucken der Zahl 2.

Wir beenden diesen Abschnitt mit einigen Betrachtungen zur Programmentwicklung unter Einbeziehung von rekursiven Prozeduren.

Die Technik des Expandierens und Komprimierens wurde in diesem Skriptum in den Kapiteln über funktionale Programmierung schon häufig verwendet, beispielsweise um Funktionen in rekursive Rechenvorschriften zu überführen oder rekursive Rechenvorschriften durch andere gleichwertige, aber effizientere zu ersetzen. Dieses Verfahren kann auch auf Prozeduren übertragen werden. Hier ist das Expandieren eines Prozeduraufrufs – in Analogie zur funktionalen Programmierung – die Ersetzung durch den Prozedurrumpf, wobei gleichzeitig die Parameter durch die Argumente des Aufrufs textuell substituiert werden, und Komprimieren ist – ebenfalls in Analogie zur funktionalen Programmierung – die Umkehrung dieser Tätigkeit, d.h. die Ersetzung einer Anweisung durch den Aufruf einer Prozedur, deren Rumpf nach der textuellen Substitution der Parameter durch die Aufrufargumente mit ihm übereinstimmt.

Nachfolgend demonstrieren wir Expandieren und Komprimieren bei Prozeduren anhand von zwei Beispielen.

7.4.11 Beispiel (Prozeduren und Schleifen)

Wir betrachten die folgende parameterlose Prozedur P, wobei in der Anweisung s der while-Schleife kein Aufruf von P vorkomme:

```
PROCEDURE P;
  BEGIN
  WHILE b DO s END
  END P;
```

Nun transformieren wir einen Aufruf P (im Rest des vorliegenden Modula-2-Programms) analog zu Beispiel 7.4.7. Wir beginnen mit dem Aufruf

$$P$$

und erhalten durch Expansion (die Kopierregel-Semantik) die gleichwertige while-Schleife

```
WHILE b DO s END,
```

[29]Man sagt, daß `P` dynamischer Vorgänger von `Q` ist. Bei dynamischer Bindung orientiert man sich an der Aufruffolge zur Laufzeit.

woraus sich durch ein einmaliges Abrollen der while-Schleife die – wiederum gleichwertige – zweiseitige bewachte Anweisung

```
IF b THEN s; WHILE b DO s END
     ELSE SKIP END
```

ergibt. Nun verwenden wir noch einmal die Definition der Prozedur P, was dem Komprimieren entspricht, und erhalten schließlich

```
IF b THEN s; P
     ELSE SKIP END
```

als gleichwertig zum Originalaufruf. Damit haben wir, nach einer entsprechenden Ergänzung, eine rekursive Fassung von P erreicht, die genau dann terminiert, wenn die while-Schleife terminiert, und somit zur while-Schleife äquivalent ist. Wir halten das Ergebnis unserer Entwicklung noch einmal fest: Eine while-Schleife

```
WHILE b DO s END
```

ist gleichwertig zu dem Aufruf P der parameterlosen rekursiven Prozedur

```
PROCEDURE P;
  BEGIN
  IF b THEN s; P
       ELSE SKIP END
  END P;
```

sofern in dem Schleifenrumpf keine Aufrufe von P vorkommen. ■

Auch die anderen vorgestellten Schleifenformen von Modula-2 können durch Aufrufe von entsprechenden parameterlosen rekursiven Prozeduren äquivalenterweise beschrieben werden.

In Kapitel 6 hatten wir gezeigt, wie man mit Hilfe von repetitiv-rekursiven Pseudooperationen imperative Programme entwickeln kann, indem man durch die angegebenen Regeln schrittweise die Pseudooperationen durch Modula-2-Sprachkonstrukte, insbesondere while-Schleifen, ersetzt. In Kombination mit der Methode des Expandierens und Komprimierens kann man rekursive Pseudooperationen auch zur Entwicklung von rekursiven Prozeduren verwenden. Wir nehmen uns dazu Beispiel 6.3.5.a noch einmal vor und entwickeln ein weiteres, auf eine rekursive Prozedur aufbauendes, Fakultätsprogramm.

7.4.12 Beispiel (für Programmentwicklung)

Wir betrachten die folgende Zuweisung, mit Programmvariablen `res` und `x` der Sorte `CARDINAL` und `fac` als einer Pseudooperation zur Berechnung der Fakultät einer natürlichen Zahl:

```
res := fac(x)
```

Aus der funktionalen Programmierung, Beispiel 4.3.5, kennen wir schon die Einbettung `fac(x)` = `facr(x,1)`, mit der repetitiv-rekursiven Pseudooperation `facr`, wie in Beispiel 6.3.5 im Rahmen von Modula-2 eingeführt:

$$\mathtt{facr(y,r)} = \begin{cases} \mathtt{facr(y-1, y*r)} & : \quad \mathtt{y} \neq 0 \\ \mathtt{r} & : \quad \mathtt{y} = 0 \end{cases}$$

Dies führt zur zu `res := fac(x)` gleichwertigen Zuweisung

```
res := facr(x,1)
```

und mit Hilfe einer geeigneten Zuweisung schließlich zum gleichwertigen Programmstück

```
res := 1;
res := facr(x,res).
```

Nun abstrahieren wir die zweite Zuweisung dieses Programmstücks zu einer Prozedur mit dem Namen `Facr`, d.h. also zu:

```
PROCEDURE Facr (VAR y, r : CARDINAL);
  BEGIN r := facr(y,r) END Facr;
```

Wir erhalten als äquivalent zum obigen Programmstück und damit auch zur originalen Zuweisung das Programmstück

```
res := 1;
Facr(x,res).
```

Es bleibt noch die Pseudooperation `facr` im Rumpf der Prozedur `Facr` zu eliminieren. Dazu transformieren wir den Rumpf geeignet, bis eine terminierende Rekursion erreicht ist. Wir starten mit

```
r := facr(y,r).
```

Nun verwenden wir die repetitiv-rekursive Beschreibung der Pseudooperation `facr` und bekommen die zweiseitige bewachte Anweisung

```
IF y = 0 THEN r := r
          ELSE r := facr(y-1,y*r) END.
```

Durch das Herausziehen der Termberechnungen aus dem Aufruf von `facr` in Zuweisungen und die anschließende Ersetzung von `r := r` durch die leere Anweisung `SKIP` erhalten wir daraus

```
IF y = 0 THEN SKIP
          ELSE r := y*r;
               y := y-1;
               r := facr(y,r) END.
```

Nun sind wir in einer Position, wo wir die Definition der Prozedur `Facr` anwenden können. Dieses Komprimieren bringt

```
IF y = 0 THEN SKIP
          ELSE r := y*r;
               y := y-1;
               Facr(y,r) END.
```

Damit haben wir für die Prozedur `Facr` eine terminierende rekursive Fassung ohne Pseudooperation `facr` erreicht. Ein Übergang nach echtem Modula-2 (mit einer einseitigen bewachten Anweisung an Stelle der zweiseitigen bewachten Anweisung mit der leeren Anweisung) und die Ergänzung durch Eingabe und Ausgabe liefert schließlich

```
MODULE FacProc;
FROM InOut IMPORT WriteLn, ReadCard, WriteCard;
VAR x, res : CARDINAL;
PROCEDURE Facr (VAR y, r : CARDINAL);
  BEGIN
  IF y # 0 THEN r := y*r;
                y := y-1;
                Facr(y,r)
                END
  END Facr;
BEGIN
  ReadCard(x);
  res := 1;
  Facr(x,res);
  WriteCard(res,10);
  WriteLn
END FacProc .
```

als Programm zur Berechnung der Fakultät einer natürlichen Zahl. Die Terminierung der Prozedur `Facr` ist offensichtlich, da unter der Gültigkeit von `y # 0` die Subtraktion `y-1` definiert ist und der Wert des ersten Arguments bei jedem Aufruf echt abnimmt. ■

7.5 Rechenvorschriften

Bis zu diesem Stand des Skriptums haben wir Prozeduren nur als Parametrisierungen von Anweisungen besprochen. Dies hat zur Konsequenz, daß ein Aufruf einer Prozedur eine Anweisung darstellt. Das Prozedurkonzept von Modula-2 ist jedoch wesentlich allgemeiner und betrifft auch den Termbegriff. In diesem Abschnitt gehen wir kurz darauf ein. Wir behandeln den zu besprechenden Begriff jedoch nicht in der Allgemeinheit, wie er unter Modula-2-Benutzern als sogenannte Funktionsprozedur bekannt ist, sondern schränken ihn auf das ein, was wir – in Analogie zu ML – eine (Modula-2)-Rechenvorschrift nennen. Diese Einschränkung ist hauptsächlich durch methodische Gesichtspunkte begründet.

7.5.1 Funktionsprozeduren

Eine **Funktionsprozedur** in Modula-2 ist syntaktisch dadurch gegeben, daß man, im Vergleich zu dem Prozedurbegriff des letzten Abschnitts 7.4, die folgenden zwei zusätzlichen Dinge hat:

a) Der Parameterliste der Prozedur ist noch ein sogenannter **Ergebnistyp** nachgestellt, abgetrennt durch einen Doppelpunkt. Wir haben also, im Vergleich zu Abschnitt 7.4.3, als Syntax:

$$\begin{array}{l} \texttt{PROCEDURE} \gg \textit{Name} \ll (\gg \textit{Liste der Parameter} \ll) \ : \ m; \\ \quad \gg \textit{Deklaration der lokalen Dinge} \ll; \\ \texttt{BEGIN} \ s \ \texttt{END} \gg \textit{Name} \ll; \end{array}$$

Dabei darf m **kein Feldtyp und kein Verbundtyp** sein.

b) Im Rumpf kommt mindestens eine **return-Anweisung**

$$\texttt{RETURN } t$$

vor, wobei t ein Term des Ergebnistyps m der Prozedur ist. ■

Informell gesprochen, bewirkt eine Funktionsprozedur eine Speicheränderung wie eine „normale“ Prozedur im Sinne des letzten Abschnitts, bis bei der sequentiellen Ausführung der Anweisungen des Rumpfs die erste return-Anweisung erreicht ist. Dann stoppt die Ausführung, der Term t wird bezüglich der vorliegenden Programmvariableninhalte ausgewertet und dann als Wert der Prozedur definiert. Aufrufe von Funktionsprozeduren bewirken also zweifaches:

(1) Sie verändern den vorliegenden Speicher wie Anweisungen.

(2) Sie berechnen ein Objekt wie Terme.

Wegen Eigenschaft (2) stellt ein Aufruf einer Funktionsprozedur **keine Anweisung dar, sondern einen Term**! Der anweisungsartige Seiteneffekt geschieht während der Auswertung des Terms. Solches ist aber nicht nur **sehr schwierig** durch eine formale mathematische Semantik korrekt zu beschreiben, sondern auch noch zusätzlich **sehr gefährlich** für die Programmierung und führt zu fehlerträchtigen und schwer zu durchschauenden Programmen. Letzteres zeigt schon das folgende kleine Beispiel.

7.5.2 Beispiel (Gefährliche Funktionsprozeduren)

Wir betrachten die folgende Funktionsprozedur `P`, wobei `a` eine globale Programmvariable der Sorte `CARDINAL` ist:

```
PROCEDURE P (x : CARDINAL) : CARDINAL;
  BEGIN a := a + x; RETURN a END P;
```

Weiterhin nehmen wir an, daß irgendwo im Programmrumpf der Term

```
P(2) + P(2)
```

ausgewertet werde. Was ist sein Wert an dieser Stelle? Im Gegensatz zu der in der Mathematik üblichen Anschauung, hängt dies entscheidend von der Auswertung der Argumente der Addition ab. Wir geben zur Verdeutlichung dieser Problematik zwei mögliche Fälle an:

- Eine sequentielle Auswertung der Argumente von links nach rechts wirkt wie die Zuweisung `a := a + 4` und liefert somit den Wert von `2 * a + 6` als Wert des Prozeduraufrufs `P(2) + P(2)`.

- Eine parallele Auswertung der Argumente der Addition wirkt hingegen wie die Zuweisung `a := a + 2` und liefert somit den Wert von `2 * a + 4` als Wert von `P(2) + P(2)`.

Verschlimmert wird die Sache noch dadurch, daß die Beschreibung von Modula-2 sich in der Auswertung von Argumenten nicht festlegt, also Übersetzer verschiedenen Code erzeugen dürfen, was eine Portierung solcher Programme sehr schwierig macht. ■

Genauso einfach kann man Beispiele für Funktionsprozeduren P und Q so angeben, daß die elementarsten Regeln der Arithmetik, wie beispielsweise das Kommutativgesetz, nicht mehr gelten, da $P(x) + Q(x)$ ungleich $Q(x) + P(x)$ für gewisse Argumente x sein kann. Dies ist natürlich fatal! Schließlich sollte noch erwähnt werden, daß auch der im letzten Kapitel vorgestellte und für die Programmierung sehr bedeutende Hoare-Kalkül davon ausgeht, daß bei der Berechnung eines Terms der Speicherzustand nich verändert wird. Es ist also nicht möglich, ihn auf Funktionsprozeduren zu erweitern, im Gegensatz zu Erweiterungen auf die „normalen" Prozeduren im Sinne von Abschnitt 7.4, die sehr wohl existieren.

Vom methodischen Gesichtspunkt aus betrachtet, sollte man **möglichst ohne globale Programmvariablen programmieren**. Es empfiehlt sich, stattdessen dafür zusätzliche Variablenparameter in die Prozeduren mit aufzunehmen. Wäre bei einer Funktionsprozedur ein Seiteneffekt nötig, so tut man gut daran, statt der Funktionsprozedur eine „normale" Prozedur zu verwenden und das Resultat in einer Programmvariablen abzulegen („Resultatvariable"). Diese letzte Bemerkung trifft insbesondere auf Zeiger zu. Liefert man Zeigergeflechte als Resultate von Funktionsprozeduren ab, ohne vorher zu kopieren, so kann eine andere Funktionsprozedur einen nicht beabsichtigten Effekt erzielen, der schwer zu durchschauen ist.

Wegen der Gefährlichkeit der Funktionsprozeduren mit Seiteneffekten betrachten wir im folgenden nur mehr solche, die ohne Seiteneffekte arbeiten. Diese verhalten sich wie die Rechenvorschriften von ML, beschreiben also Funktionen. Die Einschränkung wird auch durch die Namensgebung „Rechenvorschriften" an Stelle von Funktionsprozeduren hervorgehoben. Wir legen fest:

7.5.3 Rechenvorschriften

Eine Funktionsprozedur P heißt eine **Rechenvorschrift**, falls die drei folgenden Einschränkungen gelten:

a) Im Rumpf von P kommen weder Lese- noch Drucke-Anweisungen vor.

b) Kommt im Rumpf von P eine Programmvariable auf der linken Seite einer Zuweisung vor, so muß sie in P lokal deklariert sein.

c) Wird eine Prozedur Q ungleich P im Rumpf von P aufgerufen, so ist auch Q eine Rechenvorschrift. ■

Die Punkte 7.5.3.a bis 7.5.3.c sind syntaktische Eigenschaften, die man aus der Aufschreibung von P ablesen kann.

Wir haben auf die Forderung verzichtet, daß alle Parameter von Rechenvorschriften Objektparameter sein müssen, damit zusammengesetzte Datenstrukturen ohne Kopieraufwand mittels Call-by-reference übergeben werden können. Übergibt man in Modula-2 bei Prozeduren Verbunde oder Felder mittels Call-by-value, so wird kopiert und damit steht nach der Beendigung eines Aufrufs das ursprüngliche Objekt unter dem gleichen Namen wieder zur Verfügung. Auch Zeiger werden kopiert, wenn man sie bei Prozeduren mittels Call-by-value übergibt, verweisen jedoch auf das gleiche Objekt wie das Original. Ohne die zweite Forderung wäre es somit dennoch durch eine Dereferenzierung möglich, daß bei einem Zeiger-Objektparameter p durch einen Aufruf das Zeigergeflecht manipuliert wird, für welches p als Einstiegspunkt wirkt.

Unter Verwendung von Rechenvorschriften erlaubt auch die Programmiersprache Modula-2 in bescheidenem Maße funktionales Programmieren, wenn man von den zum Verkehr mit der Außenwelt notwendigen Lese- und Drucke-Anweisungen[30] absieht. Dabei erweist sich jedoch das Fehlen von allgemeinen Fallunterscheidung auf Termebene, die beliebige Schachtelungen erlauben, oft sehr hinderlich. In Modula-2 gibt es nur den Spezialfall

$$\texttt{IF}\ b\ \texttt{THEN RETURN}\ t_1 \\ \texttt{ELSE RETURN}\ t_2\ \texttt{END},$$

der aber eigentlich eine zweiseitige bewachte Anweisung ist, da `RETURN` t eine Anweisung darstellt. In Rechenvorschriften darf man den obigen Spezialfall jedoch mit der von der funktionalen Sprache ML her bekannten Fallunterscheidung

$$\texttt{if}\ b\ \texttt{then}\ t_1\ \texttt{else}\ t_2$$

gleichsetzen. Fallunterscheidungen mit Fallunterscheidungen in Bedingungen sind in Modula-2 syntaktisch nicht möglich. Sie können jedoch einfach durch Aufrufe von Rechenvorschriften realisiert werden, wie das nachfolgende Beispiel zeigt. In ihm übertragen wir den Test auf Perfektsein einer natürlichen Zahl von ML (siehe Abschnitt 4.1.3) nach Modula-2. Damit erhalten wir, neben dem Programm von Beispiel 6.3.5.b, welches durch Programmentwicklung mit Hilfe der Regel „(3) $\Rightarrow$ (4)“ von Abschnitt 6.3.4 entstand, ein weiteres Modula-2-Programm für diese Aufgabe.

7.5.4 Beispiel (Perfekte Zahlen in Modula-2)

Das folgende Modula-2-Programm testet, ob eine natürliche Zahl perfekt ist, oder nicht. Die in ihm verwendeten Funktionsprozeduren `isp` und `isperfect` ergeben sich direkt aus

[30] In Abschnitt 6.3.1 wurden diese eingeführt. Dabei handelt es sich, wie schon einmal erwähnt, genaugenommen um Aufrufe von vordefinierten Prozeduren, die durch Import vom vordefinierten Modul `InOut` zur Verfügung gestellt werden.

den entsprechenden ML-Rechenvorschriften von Abschnitt 4.1.3. Bezüglich des Divisionstests, realisiert durch die Rechenvorschrift `divides`, vergleiche man mit der in Beispiel 6.3.5.b gemachten Bemerkung.

```
MODULE PerfectProc;
FROM InOut IMPORT WriteLn, WriteString, ReadCard;
VAR a : CARDINAL;
PROCEDURE divides (x, y : CARDINAL) : BOOLEAN;
  BEGIN
  IF y = 0 THEN RETURN (x = 0)
           ELSE RETURN (x MOD y = 0) END
  END divides;
PROCEDURE isp (n, x, y : CARDINAL) : BOOLEAN;
  BEGIN
  IF x = 0
     THEN RETURN (y = 0)
     ELSE IF divides(n,x - 1)
             THEN RETURN isp(n,x - 1,y - x + 1)
             ELSE RETURN isp(n,x - 1,y) END END
  END isp;
PROCEDURE isperfect (n : CARDINAL) : BOOLEAN;
  BEGIN RETURN isp(n,n,n) END isperfect;
BEGIN
  ReadCard(a);
  IF isperfect(a)
     THEN WriteString('Perfekte Zahl')
     ELSE WriteString('Nicht perfekte Zahl') END;
  WriteLn
END PerfectProc .
```

Da in Modula-2 Wahrheitswerte nicht gedruckt werden können, gibt dieses Programm, wie jenes von Beispiel 6.3.5.b, einen entsprechenden Text aus. ■

Rechenvorschriften dürfen keine Felder und Verbunde als Resultate abliefern. Dies ist sehr ärgerlich, wenn beispielsweise ein Paar von Resultaten zu berechnen ist, und stellt, im Vergleich zur funktionalen Programmierung, ebenfalls eine Einschränkung dar. Wie wir jedoch schon in Abschnitt 7.2 bei den Zeigern erwähnt haben, kann man diese Einschränkung mit Hilfe von Zeigern umgehen. Wir demonstrieren dies im letzten Beispiel dieses Abschnitts.

7.5.5 Beispiel (Fibonacci-Zahlen in Modula-2)

Wir greifen die schnelle Berechnung der Fibonacci-Zahlen $fib(n)$ durch die Einbettung in eine paarwertige Funktion

$$G : \mathbb{N} \longrightarrow \mathbb{N} \times \mathbb{N} \qquad G(n) = \langle fib(n), fib(n+) \rangle$$

noch einmal auf, wie sie in Beispiel 5.5.10.b durch zwei ML-Rechenvorschriften `G` und `fib` angegeben wurde. Zur ihrer Übertragung in Modula-2-Rechenvorschriften benötigen wir einen, dem Resultaityp `int * int` der ML-Rechenvorschrift `G` entsprechenden, Modula-2-Typ `cardpair` für Paare von natürlichen Zahlen, der auch Ergebnistyp einer Rechenvorschrift sein darf. Wir wählen hierzu den Umweg über Zeiger, deklarieren also:

```
TYPE cardpair = POINTER TO pairrec;
     pairrec  = RECORD first, second : CARDINAL END;
```

Um mit Objekten dieses Typs auf funktionale Weise arbeiten zu können, benötigen wir mindestens drei Rechenvorschriften, nämlich

```
PROCEDURE make (x, y : CARDINAL) : cardpair;
  VAR p : cardpair;
  BEGIN
  NEW(p);
  p^.first := x;
  p^.second := y;
  RETURN p
  END make;
```

zum Aufbau von Objekten des Typs `cardpair` aus Paaren natürlicher Zahlen und

```
PROCEDURE p1 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.first END p1;
```

zur Auswahl der ersten bzw.

```
PROCEDURE p2 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.second END p2;
```

zur Auswahl der zweiten Komponente eines Paars. Mit Hilfe dieser drei Rechenvorschriften ist es nun offensichtlich, wie man die ML-Rechenvorschrift `G` in eine entsprechende Modula-2-Rechenvorschrift zu übertragen hat:

```
PROCEDURE G (n : CARDINAL) : cardpair;
  VAR p : cardpair;
  BEGIN
  IF n = 0 ELSE RETURN make(1,1)
           ELSE p := G(n-1);
                RETURN make(p2(p),p1(p) + p2(p)) END
  END G;
```

Auch die Übertragung der ML-Rechenvorschrift `fib` in eine entsprechende Modula-2-Rechenvorschrift ist offensichtlich:

```
PROCEDURE fib (n : CARDINAL) : CARDINAL;
  BEGIN RETURN p1(G(n)) END fib;
```

Die Ergänzung dieser Programmstücke zu einem vollständigen Modula-2-Programm

```
MODULE FibonacciProc;
FROM InOut IMPORT WriteLn, WriteCard, ReadCard;
FROM Storage IMPORT ALLOCATE;
VAR n : CARDINAL;
TYPE cardpair = POINTER TO pairrec;
     pairrec  = RECORD first, second : CARDINAL END;
PROCEDURE make (x, y : CARDINAL) : cardpair;
  VAR p : cardpair;
  BEGIN
  NEW(p);
  p^.first := x;
  p^.second := y;
  RETURN p
  END make;
PROCEDURE p1 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.first END p1;
PROCEDURE p2 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.second END p2;
PROCEDURE G (n : CARDINAL) : cardpair;
  VAR p : cardpair;
  BEGIN
  IF n = 0 ELSE RETURN make(1,1)
           ELSE p := G(n-1);
                RETURN make(p2(p),p1(p) + p2(p)) END
  END G;
PROCEDURE fib (n : CARDINAL) : CARDINAL;
  BEGIN RETURN p1(G(n)) END fib;
BEGIN
  ReadCard(n);
  WriteCard(fib(n),10);
  WriteLn
END FibonacciProc .
```

das eine natürliche Zahl n einliest und die n-te Fibonacci-Zahl $\mathit{fib}(n)$ ausdruckt, ist nun trivial. ■

Damit beenden wir die Vorstellung der imperativen Programmierung. Wie am Ende der Kapitel über funktionale Programmierung, so sei auch an dieser Stelle ausdrücklich darauf hingewiesen, daß wir nur einen Teil der grundlegenden Prinzipien des imperativen Paradigmas und der Sprachmittel der von uns verwendeten Sprache Modula-2 behandeln konnten. Insbesondere haben wir den gesamten Bereich „Prozeduren als Parameter“ ausgespart. Wer sich intensiver mit imperativer Programmierung befassen will, kommt an weiterführender Literatur und auch Originalliteratur nicht vorbei.

8 Algorithmen und Datenstrukturen

In den letzten vier Kapiteln 4 bis 7 wurde versucht, eine Einführung in funktionale und prozedurale Programmierung zu geben, wobei Sprachkonzepte und qualitative Aspekte wie Programmentwurf und -korrektheit im Vordergrund standen. Die Effizienz der gezeigten Programme wurde nur manchmal angesprochen. In diesem Kapitel geht es nun aber gerade um die Effizienz der Programme und auch darum, wie diese mit einer geeigneten Darstellung der Daten zusammenhängt. Dazu greifen wir zuerst in Abschnitt 8.1 das Thema „Sortieren“ noch einmal auf, das wir bei der Vertiefung von funktionaler Programmierung schon einmal behandelten. Dies ist dadurch motiviert, daß Sortierverfahren einerseits zu den wichtigsten Verfahren überhaupt gehören und wir durch sie andererseits auch einen guten Einstieg in den nächsten Abschnitt 8.2 über Komplexität und Effizienz bekommen. Nach diesem theoretischen Abschnitt konzentrieren wir uns im Rest dieses Kapitels wieder auf praktische Probleme bei Binärbäumen und Graphen.

8.1 Sortieralgorithmen

Sortieren ist eine sehr wichtige elementare Aufgabe, welche auch oft innerhalb komplexerer Algorithmen auftaucht. In der funktionalen Programmiersprache ML haben wir bereits drei Sortierverfahren auf Listen ganzer Zahlen kennengelernt, nämlich

(1) Sortieren durch das Einsortieren eines Elements in eine schon sortierte Liste (Rechenvorschrift `sort` in Beispiel 5.5.11.b),

(2) Sortieren nach der Methode des Aufteilens (Rechenvorschrift `quicksort` in Beispiel 5.5.11.c) und

(3) Sortieren durch das Mischen von sortieren Listen (Rechenvorschrift `mergesort` in Beispiel 5.6.3.c).

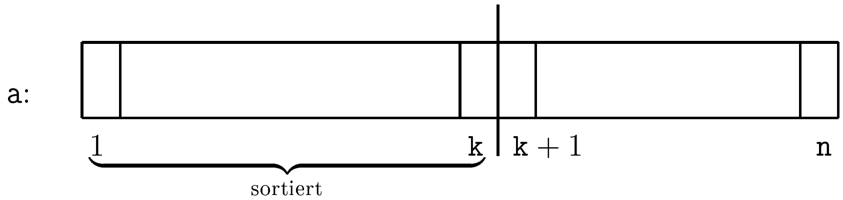
Bei imperativen Programmiersprachen, wie Modula-2, ist die Standardaufgabe beim Sortieren, ein gegebenes Feld zu sortieren. Wie dies geschehen kann, soll in diesem Abschnitt anhand einiger Programme demonstriert werden. Der Einfachheit halber betrachten wir nur Felder von ganzen Zahlen. Eine Übertragung auf Felder mit anderen Komponententypen ist offensichtlich. Wir beginnen mit der Übertragung des auf das Einsortieren eines Elements basierenden Sortierverfahrens (1) von ML nach Modula-2.

8.1.1 Sortieren durch Einsortieren

Es sei `a` das zu sortierende Feld mit Komponententyp `INTEGER` und Indextyp `[1..n]`. Überträgt man die Sortieridee des Verfahrens „Sortieren durch Einsortieren“ von der Sprache ML nach Modula-2, so hat man die in dem folgenden Bild dargestellte allgemeine Situation zu betrachten, welche genau der **Invariante**

$$\forall\, i : 1 \le i \le \mathtt{k} - 1 \rightarrow \mathtt{a}[i] \le \mathtt{a}[i+1]$$

des zu erstellenden Programms (mit einer weiteren Programmvariablen k) entspricht:



Der Fall $\mathtt{k} = 0$, bei dem das gesamte Feld noch unsortiert ist, ist der Startpunkt des Sortierverfahrens, bei $\mathtt{k} = \mathtt{n}$ ist das gesamte Feld sortiert, das Sortierverfahren also beendet. Man durchläuft im Laufe des Verfahrens somit die Zahlen von Null bis $\mathtt{n} - 1$. Es gilt nun, für k ungleich n zwei Aufgaben zur Aufrechterhaltung der Invariante zu lösen, nämlich

(1) die neue Position des Feldelements $\mathtt{a[k+1]}$ zu bestimmen, d.h. einen Index j im sortierten Teil des Felds, so daß $\mathtt{a[j]} \leq \mathtt{a[k+1]} < \mathtt{a[j+1]}$ gilt,

(2) die kollaterale Zuweisung (Pseudooperation)

$$\mathtt{(a[j+1],\ldots,a[k],a[k+1])} \; := \; \mathtt{(a[k+1],a[j+1],\ldots,a[k])}$$

zu realisieren, die das Einfügen von $\mathtt{a[k+1]}$ an der richtigen Stelle $\mathtt{j}{+}1$ im sortierten Teilfeld bei der gleichzeitigen Verschiebung der Feldelemente $\mathtt{a[j+1]}$ bis a[k] um eine Stelle „nach rechts“ beschreibt.

Die Positionssuche und die Einfügung bzw. Verschiebung kann man elegant kombinieren, indem man vom Index k ab rückwärts läuft. Ein entsprechendes Programmstück sieht wie folgt aus, wobei im then-Fall der bewachten Anweisung mittels einer Hilfsprogrammvariablen h die Vertauschung von a[j] und a[j+1] bewerkstelligt wird:

```
j := k;
WHILE j >= 1 DO
  IF a[j] > a[j+1] THEN h := a[j];
                        a[j] := a[j+1];
                        a[j+1] := h END;
  j := j - 1
  END
```

Der Leser mache sich die Invariante dieser Schleife klar! Nach der oben gemachten Bemerkung ist dieses Programmstück nun noch in eine Schleife einzubetten, die für k vom Startwert 0 bis zum Endwert n - 1 läuft. Hier bietet sich eine for-Schleife an, die in

```
FOR k := 0 TO n - 1 DO
  j := k;
  WHILE j >= 1 DO
    IF a[j] > a[j+1] THEN h := a[j];
                          a[j] := a[j+1];
                          a[j+1] := h END;
    j := j - 1
    END
  END
```

resultiert. Eine **Indextransformation** `k` $\mapsto$ `k` $+ 1$ ermöglicht es, daß die Laufvariable `k` vom gleichen Typ als der Indextyp des Felds `a` ist, wir also die Schleifenstruktur

```
FOR k := 1 TO n DO
  j := k - 1;
  WHILE ... END
  END
```

erhalten. Dies ist vorteilhaft, wenn man statt des Bereichstyps `[1..n]` einen Aufzählungstyp als Indextyp des zu sortierenden Felds wählen würde. Als eine weitere Effizienzverbesserung bietet sich an, die innere Schleife **vorzeitig terminieren** zu lassen, falls keine Vertauschung mehr stattzufinden hat, da dann die neue Position erreicht ist. Dies ist durch eine zusätzliche Programmvariable der Sorte `BOOLEAN` möglich, deren Negation man konjunktiv der bisherigen Bedingung der while-Schleife hinzufügt. Sie wird mit `FALSE` initialisiert und zu `TRUE` umbesetzt, falls die Vertauschungen beendet sind. Insgesamt erhalten wir für den Fall von `[1..20]` als Indextyp

```
MODULE Insort;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadInt, WriteInt;
VAR a : ARRAY [1..20] OF INTEGER;
    j, k : [1..20];
    h : INTEGER;
    posfound : BOOLEAN;
BEGIN
  WriteString('Gib 20 ganze Zahlen ein.');
  WriteLn;
  FOR k := 1 TO 20 DO
    ReadInt(a[k]) END;
  FOR k := 1 TO 20 DO
    j := k-1;
    posfound := FALSE;
    WHILE (j >= 1) AND NOT posfound DO
      IF a[j] > a[j+1] THEN h := a[j];
                            a[j] := a[j+1];
                            a[j+1] := h
                       ELSE posfound := TRUE END;
      j := j - 1
      END
    END;
  WriteString('Sortiertes Feld:');
  WriteLn;
  FOR k := 1 TO 20 DO
    WriteInt(a[k],5) END;
  WriteLn
END Insort .
```

als komplettes Modula-2-Programm zum Einlesen der 20 Zahlen in ein Feld, Sortieren dieses Felds in aufsteigender Ordnung nach der Methode des Einsortierens und zum schließlichen Ausdrucken des sortierten Felds. ■

Bevor wir die Übertragung des Quicksort-Verfahrens von ML nach Modula-2 betrachten, soll noch ein weiterer einfacherer Sortieralgorithmus in Modula-2 behandelt werden, dessen Güte bezüglich der Effizienz der Güte des Algorithmus von Abschnitt 8.1.1 gleich ist. Dieser basiert anschaulich auf der Idee der Minimumsuche und formal auf einer Verstärkung der Invariante von `Insort`.

8.1.2 Sortieren durch Minimumsuche

Wir betrachten noch einmal die am Beginn von Beispiel 5.5.11 spezifizierte Funktion $sort : \mathbb{Z}^* \longrightarrow \mathbb{Z}^*$ zum aufsteigenden Sortieren einer Liste von ganzen Zahlen. Für diese Funktion gilt offensichtlich die Gleichung

$$sort(\varepsilon) = \varepsilon \tag{1}$$

und, wie man ebenso leicht aus der prädikatenlogischen Spezifikation von *sort* in Beispiel 5.5.11 beweist, die Gleichung ($a, b \in \mathbb{Z}^*$, $x \in \mathbb{Z}$)

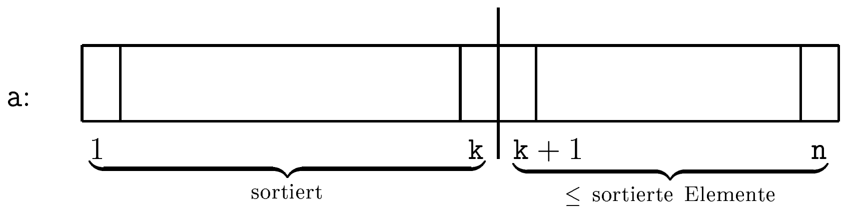
$$sort(a\,x\,b) = x\,sort(a\,b) \tag{2}$$

falls $x \leq a_i$ und $x \leq b_j$ für alle i, j mit $1 \leq i \leq |a|$ und $1 \leq j \leq |b|$ gilt, d.h. x ein minimales Element der Eingabeliste ist. Eine Implementierung der durch die Formeln (1) und (2) gegebenen Rekursion als ein ML-Programm ist offensichtlich. Man benötigt zwei Hilfsrechenvorschriften zur Bestimmung eines minimalen Listenelementes bzw. zur Entfernung eines Listenelementes. Auf die exakte Formulierung des ML-Sortierprogramms sei an dieser Stelle verzichtet, diese sei dem Leser als einfache Programmierübung überlassen.

Es habe das zu sortierende Feld `a` wiederum den Komponententyp `INTEGER` und den Indextyp `[1..n]` mit einer natürlichen Zahl `n`. Wir übertragen nun die obige Algorithmenidee des Sortierens durch Minimumsuche in die Programmiersprache Modula-2 zum Sortieren eines Felds. Dies führt zur Betrachtung der im folgenden Bild dargestellten allgemeinen Situation, in der, wie schon in Abschnitt 8.1.1 beim Sortieren durch Einsortieren, wiederum die **Invariante**

$$\begin{array}{l} \forall\, i : 1 \leq i \leq \mathtt{k} - 1 \rightarrow \mathtt{a}[i] \leq \mathtt{a}[i+1] \\ \wedge\, \forall\, i, j : 1 \leq i \leq \mathtt{k} \wedge \mathtt{k} + 1 \leq j \leq \mathtt{n} \rightarrow \mathtt{a}[i] \leq \mathtt{a}[j] \end{array}$$

des Verfahrens deutlich aufgezeigt wird:



Wie in Abschnitt 8.1.1, so ist auch hier $\texttt{k} = 0$ der Startpunkt des Verfahrens, da dies offensichtlich die Invariante etabliert. Ebenso endet das Verfahren bei $\texttt{k} = \texttt{n}$, da dies gleichwertig dazu ist, daß das gesamte Feld sortiert ist. Man durchläuft somit wiederum mit dem Index k den Bereich von Null bis $\texttt{n}-1$. Um den sortierten Bereich mit der Zusatzbedingung $\texttt{a[}i\texttt{]} \leq \texttt{a[}j\texttt{]}$ für i aus dem sortierten und j aus dem nichtsortierten Bereich um einen Index zu vergrößern, haben wir für k ungleich n zwei Aufgaben zu bewerkstelligen, nämlich

(1) die Position j mit $\texttt{k}+1 \leq \texttt{j} \leq \texttt{n}$, eines minimalen Elements a[j] im nichtsortierten Teil von a zu bestimmen und

(2) anschließend die Elemente $\texttt{a[k}+1\texttt{]}$ und a[j] zu vertauschen.

Dies sichert zu, daß die Gültigkeit der Invariante bei jedem Schleifendurchlauf erhalten bleibt. Das Programmstück für die Suche der Position j zwischen $\texttt{k}+1$ und n ist spezifiziert durch die Nachbedingung

$$\texttt{a[j]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{k+1}\ \leq i \leq \texttt{n}\ \},$$

welche impliziert wird von der Konjunktion

$$\texttt{a[j]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{r} \leq i \leq \texttt{n}\ \} \wedge \texttt{r} = \texttt{k+1},$$

mit einer weiteren (den Suchbereich durchlaufenden) Programmvariablen r. Nimmt man den ersten Teil dieser Konjunktion als Invariante $Inv(\texttt{a},\texttt{j},\texttt{r})$ und ihren zweiten Teil als Negation der Schleifenbedingung r # k + 1, so führt die Initialisierung

```
r := n;
j := n;
```

zur Gültigkeit der Invariante $Inv(\texttt{a},\texttt{j},\texttt{r})$, da $\texttt{a[n]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{n} \leq i \leq \texttt{n}\ \}$ offensichtlich wahr ist. In der Schleife ist nun aufgrund dieser Initialisierung und der r zugedachten Rolle der Wert von r um 1 zu erniedrigen, und j dabei so zu verändern, daß die Invariante erhalten bleibt. Offensichtlich gilt die Implikation

$$\texttt{a[j]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{r} \leq i \leq \texttt{n}\ \} \to \texttt{a[r-1]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{r-1} \leq i \leq \texttt{n}\ \},$$

also $Inv(\texttt{a},\texttt{j},\texttt{r}) \to Inv(\texttt{a},\texttt{r-1},\texttt{r-1})$, falls die Bedingung a[r-1] < a[j] gleich TRUE ist. Ist sie hingegen FALSE, also TRUE der Wert ihres Negats a[r-1] >= a[j], so gilt

$$\texttt{a[j]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{r} \leq i \leq \texttt{n}\ \} \to \texttt{a[j]} = min\{\ \texttt{a[}i\texttt{]}\ :\ \texttt{r-1} \leq i \leq \texttt{n}\ \},$$

also $Inv(\texttt{a},\texttt{j},\texttt{r}) \to Inv(\texttt{a},\texttt{j},\texttt{r-1})$. Insgesamt bekommen wir, analog zum Vorgehen in Abschnitt 6.4, die Position j durch das nachstehende Programmstück:

```
r := n;
j := n;
WHILE r # k+1 DO
  IF a[r-1] < a[j] THEN j := r-1 END;
  r := r-1
  END
```

Dieses Programmstück ist jetzt noch in eine Schleife einzubauen, die für `k` von `0` bis `n - 1` führt und nach der Bestimmung von `j` in der `k`-Schleife noch die Feldelemente `a[j]` und `a[k+1]` vertauscht. Wir haben also die folgende Schleifenstruktur:

```
FOR k := 0 TO n - 1 DO
  r := n;
  j := n;
  WHILE ... END
  h := a[j];
  a[j] := a[k+1];
  a[k+1] := h
  END
```

Auch hier bietet sich, wie beim Programm `Insort` des letzten Abschnitts 8.1.1, eine Indextransformation $\mathtt{k} \mapsto \mathtt{k} + 1$ an, und wir erhalten für den speziellen Indextyp `[1..20]` schließlich

```
MODULE Minsort;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadInt, WriteInt;
VAR a : ARRAY [1..20] OF INTEGER;
    j, k, r : [1..20];
    h : INTEGER;
BEGIN
  WriteString('Gib 20 ganze Zahlen ein.');
  WriteLn;
  FOR k := 1 TO 20 DO
    ReadInt(a[k]) END;
  FOR k := 1 TO 20 DO
    r := 20;
    j := 20;
    WHILE r # k DO
      IF a[r-1] < a[j] THEN j := r - 1 END;
      r := r - 1
      END;
    h := a[j];
    a[j] := a[k];
    a[k] := h
    END;
  WriteString('Sortiertes Feld:');
  WriteLn;
  FOR k := 1 TO 20 DO
    WriteInt(a[k],5) END;
  WriteLn
END Minsort .
```

als Sortierprogramm, welches eine Folge von 20 Zahlen in ein Feld einliest, dieses sortiert und die 20 Zahlen aufsteigend sortiert wieder ausdruckt. ■

Als nächsten Sortieralgorithmus behandeln wir nun das Quicksort-Verfahren und geben eine Implementierung in Modula-2 an.

8.1.3 Quicksort

Es sei $sort : \mathbb{Z}^* \longrightarrow \mathbb{Z}^*$ die Sortierfunktion auf Listen ganzer Zahlen. Die Idee des Quicksort-Verfahrens in der Sprache ML von Beispiel 5.5.11.b war, für eine nichtleere Liste s von ganzen Zahlen und ein Element x aus s die Gleichung

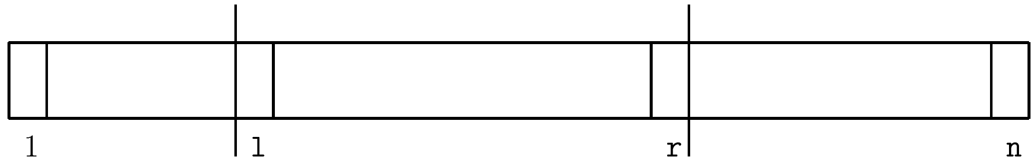
$$sort(s) = sort(lo)\, eq\, sort(up) \tag{1}$$

zu verwenden, wobei die drei Teillisten lo, eq und up die Elemente y von s mit $y < x$, $y = x$ bzw. $y > x$ enthalten. Bei einer Übertragung dieser Idee nach Modula-2 ist die Dreiteilung von Gleichung (1) etwas hinderlich. Man betrachtet stattdessen für $s \neq \varepsilon$ und einem Element x aus s die Gleichung

$$sort(s) = sort(leq)\, sort(geq), \tag{2}$$

wobei nun die zwei Teillisten leq und geq von s Elemente y von s mit $y \leq x$ bzw. $y \geq x$ enthalten. Man beachte, daß die Zerlegung von s gemäß Gleichung (2) nicht eindeutig sein muß, im Gegensatz zur Zerlegung (1), die immer eindeutig ist.

Nun sei `a` das zu sortierende Feld mit dem Indextyp `[1..n]` und dem Komponententyp `INTEGER`. Mit Blick auf Gleichung (2) betrachten wir die in dem nachfolgenden Bild beschriebene allgemeine Situation, wobei für die inneren Feldindizes die Abschätzung $1 \leq$ `l` $\leq$ `r` $\leq$ `n` gelte:

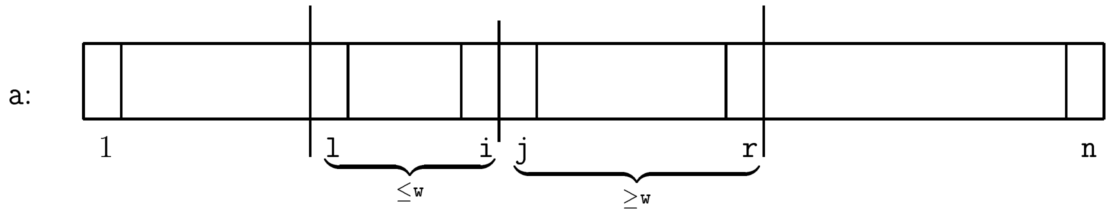
a: 

Dabei seien die äußeren Teilfelder schon sortiert. Weiterhin seien alle Elemente des linken Teilfelds kleiner oder gleich als alle Elemente des mittleren Teilfelds und diese wiederum kleiner oder gleich als alle Elemente des rechten Teilfelds. Dies ist die **Invariante** des Verfahrens, welches wir als Prozedur

```
PROCEDURE quicksort (VAR l, r : CARDINAL); ...
```

mit dem globalen Feld `a` zu formulieren vorhaben. Am Anfang ist das gesamte Feld unsortiert, es gilt also `l` $= 1$ und `r` $=$ `n` bzw. `L` $= 1$ und `R` $=$ `n` für die Programmvariablen des Initialaufrufs `quicksort(L,R)`. Das Verfahren terminiert, falls `l` $\geq$ `r` gilt.

Man wählt nun bei der oben vorliegenden allgemeinen Situation aus dem mittleren Teilfeld von `a` mit der linken Grenze `l` und der rechten Grenze `r` irgendeinen Komponentenwert `w` aus, etwa `w` $=$ `a[(l+r) DIV 2]`. Dann teilt man das Teilfeld nach dem Wert `w` in zwei weitere Teilfelder auf, nämlich in die Elemente `a[l]` bis `a[i]`, die alle kleiner oder gleich `w` sind, und die Elemente `a[j]` bis `a[r]`, mit (normalerweise) `i` $+ 1 =$ `j`, die alle größer oder gleich `w` sind. Bildlich sieht dies wie folgt aus:



Und schließlich ruft man das Verfahren rekursiv für die Teilfelder von `l` bis `i` als auch von `j` bis `r` auf, falls diese Teilfelder mindestens die Länge 2 haben. Wird die Invariante durch die Aufrufe `quicksort(l,i)` und `quicksort(j,r)` aufrechterhalten und sortieren die beiden Aufrufe die entsprechenden Teilfelder, so ist nach ihren Abarbeitungen auch das gesamte Feld sortiert. Bisher haben wir, unter Verwendung von zwei Pseudooperationen `min` und `max`, somit die folgende Struktur der zu erstellenden Sortierprozedur:

```
PROCEDURE quicksort (VAR l, r : CARDINAL);
  VAR w : INTEGER;
      i, j : CARDINAL;
  BEGIN
  j := l;
  i := r;
  w := a[(l + r) DIV 2];
  REPEAT j := min { k : l ≤ k ≤ r ∧ a[k] ≥ w };
         i := max { k : l ≤ k ≤ r ∧ a[k] ≤ w };
         ≫ Ggf. Vertauschen von a[i] und a[j] und Weiterzählen ≪
  UNTIL  j > i;
  IF l < i THEN quicksort(l,i) END;
  IF j < r THEN quicksort(j,r) END
  END quicksort;
```

Es bleibt noch der Rumpf der repeat-Schleife von `quicksort` zu realisieren, d.h. die Zuweisungen an die Programmvariablen `j` und `i`, das Vertauschen der Feldelemente `a[i]` und `a[j]` im Fall `j` < `i` und das Weiterzählen zu formulieren. Wir gehen wie folgt vor:

(1) Der Index `j` startet bei `l` und läuft **vorwärts**, bis man ein Element `a[j]` findet, daß größer oder gleich `w` ist. Dies schreibt sich in Modula-2 als Schleife

```
WHILE a[j] < w DO j := j + 1 END.
```

(2) Der Index `i` startet bei `r` und läuft **rückwärts**, bis man ein Element `a[i]` findet, daß kleiner oder gleich `w` ist. In Modula-2 erhalten wir hierfür

```
WHILE a[i] > w DO i := i - 1 END.
```

(3) Gilt nach Beendigung von (1) und (2) die Beziehung `j` ≤ `i`, so vertauscht man beide Indizes und zählt `j` um 1 hoch sowie `i` um 1 runter. Im Fall `j` = `i` zählt man hingegen nur `j` um 1 hoch bzw. `i` um 1 runter, ohne zu vertauschen. Bei `j` > `i` ist man schon fertig, tut also nichts. In Modula-2 wird dies insgesamt zu

```
IF j <= i THEN IF j < i THEN ≫ Vertauschen ≪ END;
               j := j + 1;
               i := i - 1 END.
```

Das noch verbleibende Programmstück zum Vertauschen von `a[i]` und `a[j]` haben wir dabei schon öfter realisiert.

Wie schon oben erwähnt, erfolgt der Initialaufruf von `quicksort` mit zwei Programmvariablen `L` und `R`, deren Inhalte anfangs die Feldgrenzen `1` und `n` von `a` sind. Für `n` = 20 erhalten wir das nachfolgend angegebene Sortierprogramm in Modula-2:

```
MODULE Quicksort;
FROM InOut IMPORT WriteLn, WriteString,
                  ReadInt, WriteInt;
VAR a : ARRAY [1..20] OF INTEGER;
    k, L, R : CARDINAL;
PROCEDURE quicksort (VAR l, r: CARDINAL);
  VAR w : INTEGER;
      i, j : CARDINAL;
      h : INTEGER;
  BEGIN
  j := l;
  i := r;
  w := a[(l+r) DIV 2];
  REPEAT WHILE a[j] < w DO j := j + 1 END;
         WHILE a[i] > w DO i := i - 1 END;
         IF j <= i THEN IF j < i THEN h := a[i];
                                      a[i] := a[j];
                                      a[j] := h END;
                        j := j + 1;
                        i := i - 1 END
  UNTIL j > i;
  IF l < i THEN quicksort(l,i) END;
  IF j < r THEN quicksort(j,r) END
  END quicksort;
BEGIN
  WriteString('Gib 20 ganze Zahlen ein.');
  WriteLn;
  FOR k := 1 TO 20 DO
    ReadInt(a[k]) END;
  L := 1;
  R := 20;
  quicksort(L,R);
  WriteString('Sortiertes Feld:');
  WriteLn;
  FOR k := 1 TO 20 DO
    WriteInt(a[k],5) END;
  WriteLn
END Quicksort .
```

An dieser Stelle soll auch noch kurz auf die Typisierung der Programmvariablen `k`, `L` und `R` des Programms `Quicksort` und der Variablenparameter `l` und `r` der Prozedur `quicksort` eingegangen werden. Offensichtlich liegen während der gesamten Abarbeitung des Verfahrens die Werte von `k`, `L`, `R`, `l` und `r` im Intervall von `1` bis `20`. Es liegt somit nahe, statt der Sorte `CARDINAL` den Bereichstyp `[1..20]` als Typ zu verwenden. Auf direkte Weise ist dies leider in Modula-2 nicht möglich, denn die Sprache erlaubt **keine Bereichstypen als Typen von Prozedurparametern**. Eine indirekte Abhilfe ist, für den Typ `[1..20]` einen Namen einzuführen. ■

Offensichtlich kann die Prozedur `quicksort` statt mit einer repeat-Schleife auch mit einer while-Schleife realisiert werden. Da man so eine Version oft in der Literatur findet, sei dies an dieser Stelle extra erwähnt.

Wie wir später sehen werden, sind die Verfahren von Abschnitt 8.1.1 und Abschnitt 8.1.2 gleich effizient, während das Quicksort-Verfahren 8.1.3 normalerweise wesentlich schneller arbeitet. Es gibt noch weitere „schnelle" Sortierverfahren, etwa das Sortieren durch Mischen (welches wir schon bei ML kennengelernt haben) oder das Sortieren mittels eines sogenannten Heaps, auf die aber hier nicht eingegangen werden kann. Es sei deshalb auf weiterführende Literatur verwiesen.

8.2 Effizienz und Komplexität

Verschiedene Algorithmen oder Programme zur Lösung desselben Problems können verschieden effizient sein, wobei die Effizienzkriterien, wie schon in Abschnitt 3.4.3 gezeigt, sehr verschieden sein können. Im folgenden befassen wir uns der Einfachheit halber nur mit der **Laufzeiteffizienz** von Algorithmen, welche in der Praxis auch das wichtigste Effizienzmaß darstellt.

8.2.1 Methoden zur Effizienzbeurteilung

a) Ist ein Algorithmus in einer fest gewählten Programmiersprache (mit dem zugehörigen System) implementiert und wird er auf einem bestimmten Rechner zu bestimmten Eingabedaten ausgeführt, so kann man die Effizienz durch **Zeitmessung** bestimmen, d.h. durch eine **mitlaufende Uhr**. Sowohl ML als auch Modula-2 bieten Sprachkonstrukte zur Zeitmessung an.

b) Normalerweise **abstrahiert** man aber von der Programmiersprache, dem Rechner und den Eingabedaten. Dies führt zu der Auffassung von einem Effizienzmaß als **Aufwandsfunktion**

$$\xi : \mathit{Problemgr\"o\ss e} \;\mapsto\; \mathit{Anzahl\ der\ Rechenschritte},$$

eines Algorithmus, wobei sowohl die Problemgröße als auch die Anzahl der benötigten (als elementar angesehenen) Rechenschritte normalerweise natürliche Zahlen sind.

c) Durch den Ansatz b) wird zu einer Problemgröße $n \in \mathbb{N}$ die Zahl $\xi(n)$ der von einem Algorithmus auszuführenden Rechenschritte exakt angegeben. Dies auszurechnen ist für komplexe Algorithmen aber oft viel zu schwierig. Man gibt sich deshalb mit einer **Betrachtung von oberen Schranken** zufrieden und spricht dann von **Worst-case-analysis**. Da man normalerweise auch über die Problemgröße nichts genaueres aussagen kann, läßt man n gegen unendlich gehen. Dies bezeichnet man als **asymptotische Abschätzung**. ■

Worst-case-analysis mit asymptotischer Abschätzung ist die derzeit gängige Methode der Laufzeiteffizienzbeurteilung. Wir befassen uns im folgenden nur mit ihr. Die nachfolgenden Bezeichnungen, welche auf den Zahlentheoretiker E. Landau zurückgehen, sind wesentlich für den Ansatz. In ihnen bezeichnet $\mathbb{R}_{\geq 0}$ die Menge der nichtnegativen reellen Zahlen.

8.2.2 Definition (Asymptotische Größenordnung)

a) Zu einer Funktion $f : \mathbb{N} \longrightarrow \mathbb{R}_{\geq 0}$ heißt die Funktionsmenge

$$\mathcal{O}(f) = \{\ g \in \mathbb{R}_{\geq 0}^{\mathbb{N}}\ :\ \exists\, c > 0, n_0 \in \mathbb{N} : \forall\, n \geq n_0 : g(n) \leq c * f(n)\ \}$$

(wobei c eine reelle Zahl ist) die **asymptotische Größenordnung** von f.

b) Ist $\xi : \mathbb{N} \longrightarrow \mathbb{N}$ die Aufwandsfunktion eines Algorithmus $\mathcal{A}$, so heißt ξ von f **asymptotisch dominiert** (oder $\mathcal{O}(f)$ die **asymptotische Größenordnung**, die **Komplexität**, der **Aufwand**) von ξ bzw. $\mathcal{A}$), falls $\xi \in \mathcal{O}(f)$ gilt. ■

Ist f eine Funktion auf den natürlichen Zahlen, die niemals Null als Funktionswert annehmen kann, so hat man die Darstellung

$$\mathcal{O}(f) = \{\ g \in \mathbb{R}_{\geq 0}^{\mathbb{N}}\ :\ \exists\, c > 0 : \lim_{n \to \infty} \frac{g(n)}{f(n)} \leq c\ \},$$

falls der Grenzwert des Bruchs $\frac{g(n)}{f(n)}$ existiert. Im Unendlichen unterscheiden sich also die Funktionswerte von f und g zu jedem Argument $n \in \mathbb{N}$ höchstens um die Konstante c.

8.2.3 Beispiele (für Größenordnungen)

Sind die Funktionen f, g auf den natürlichen Zahlen definiert durch die Gleichungen $f(n) = t_1$ und $g(n) = t_2$, mit t_1 und t_2 als Ausdrücke in der Veränderlichen n, so schreibt man in der Literatur üblicherweise $t_2 \in \mathcal{O}(t_1)$ statt $g \in \mathcal{O}(f)$.

Mit diesen Abkürzungen gelten beispielsweise die drei Beziehungen

$$5 \in \mathcal{O}(1) \qquad 4 * n + 3 \in \mathcal{O}(n) \qquad 3 * n^2 + 2 \in \mathcal{O}(n^2).$$

Zum Beweis von, sagen wir, $4*n+3 \in \mathcal{O}(n)$ betrachten wir die Ungleichung $4*n+3 \leq 5*n$, die für alle $n \geq 3$ zutrifft. Nun wählen wir in Definition 8.2.2.a die Konstante $c > 0$ als 5 und $n_0 \in \mathbb{N}$ als 3.

Beispielsweise gilt auch die Beziehung $3*n^2+2 \in \mathcal{O}(n^k)$ für alle natürlichen Zahlen $k \geq 2$, aber andererseits auch $3*n^2+2 \notin \mathcal{O}(n)$. ■

Bisher kennen wir nur zwei Klassen von Problemen, nämlich diejenigen, welche algorithmisch lösbar sind, und diejenigen, welche algorithmisch nicht lösbar sind. Wegen Definition 8.2.2 können wir nun die Klasse der lösbaren Probleme verfeinern, indem wir auch den Aufwand für die Lösung betrachten.

8.2.4 Definition (Komplexitätsklassen)

Es sei $\xi : \mathbb{N} \longrightarrow \mathbb{N}$ die Aufwandsfunktion eines Algorithmus $\mathcal{A}$. Man legt fest:

a) $\mathcal{A}$ heißt **polynomial-zeitbeschränkt**, falls es ein Polynom $p(n) = n^k$ mit $k \in \mathbb{N}$ und $\xi \in \mathcal{O}(p) = \mathcal{O}(n^k)$ gibt.

b) $\mathcal{A}$ heißt **exponentiell-zeitbeschränkt**, falls es eine Exponentialfunktion $e(n) = k^n$ mit $k \in \mathbb{N}$ und $\xi \in \mathcal{O}(e) = \mathcal{O}(k^n)$ gibt. ■

Warum macht man diesen Unterschied? Bei einer naiven Betrachtung der verwendeten Symbole n und k scheint zwischen $p(n) = n^k$ und $e(n) = k^n$ kein großer Unterschied zu bestehen. Es spielt jedoch für das Wachstumsverhalten einer Funktion eine entscheidende Rolle, ob die Veränderliche n als Basis oder als Exponent einer Potenzierung auftritt. Dies wird im folgenden Beispiel klar dargestellt.

8.2.5 Beispiel (Vergleich $\mathcal{O}(n^k)$ und $\mathcal{O}(k^n)$)

Wir betrachten drei Algorithmen $\mathcal{A}_1$, $\mathcal{A}_2$ und $\mathcal{A}_3$ und die vier verschiedenen Problemgrößen $n = 10, 30, 60, 100$, die etwa vier verschiedene Feldlängen im Fall von Sortieralgorithmus sein können. Weiterhin betrachten wir die folgenden drei Aufwandsfunktionen (als Terme in n geschrieben):

$$\text{Algorithmus } \mathcal{A}_1 : n^2 \qquad \text{Algorithmus } \mathcal{A}_2 : n^3 \qquad \text{Algorithmus } \mathcal{A}_3 : 2^n$$

Nun nehmen wir an, daß ein einzelner elementarer Schritt eines jeden der vorliegenden Algorithmen genau 10^{-6} Sekunden Rechenzeit benötigt. Dann ergeben sich die in der nachfolgenden Tabelle zusammengestellten Gesamtrechenzeiten:

Algorithmus	$n = 10$	$n = 30$	$n = 60$	$n = 100$
$\mathcal{A}_1$	10^{-4} Sek.	10^{-3} Sek.	0.004 Sek.	0.01 Sek.
$\mathcal{A}_2$	10^{-3} Sek.	0.03 Sek.	0.2 Sek.	1 Sek.
$\mathcal{A}_3$	10^{-3} Sek.	17 Min.	36000 Jahre	$4*10^{15}$ Jahre

Dabei stehen die Abkürzungen „Sek“ bzw. „Min“ für Sekunden bzw. Minuten. Zwischen den Begriffen „polynomial“ und „exponentiell“ liegt also ein Qualitätssprung. Diese Diskrepanz kann auch durch Technologiesprünge nicht behoben werden. Eine Verbesserung der Schnelligkeit eines Rechners um den Faktor 1000 bewirkt beispielsweise bei einer Aufwandsfunktion aus $\mathcal{O}(n^2)$ viel, bei einer Aufwandsfunktion aus $\mathcal{O}(2^n)$ hingegen fast gar nichts. ■

Eine heutzutage weitgehend akzeptierte Arbeitshypothese der Informatik, die von J. Edmonds im Jahre 1964 aufgestellt wurde, besagt: **Ein Algorithmus ist praktikabel, falls er polynomial-zeitbeschränkt ist.** Im folgenden untersuchen wir unsere bisherigen Sortierprogramme auf Praktikabilität.

8.2.6 Komplexität von Sortieralgorithmen

Wir betrachten die Sortierprogramme aus den Abschnitten 8.1.1 bis 8.1.3. Als Problemgröße n definieren wir die Länge eines beliebigen zu sortierenden Feldes – also nicht die in den Programmen jeweils angegebene Zahl 20 – und als elementare Rechenschritte definieren wir Lese- und Druckanweisungen, Zuweisungen und Termauswertungen.

a) **Eingaben:** Bei allen drei Sortierprogrammen besteht die Eingabe aus zwei Drucke-Anweisungen und n Lese-Anweisungen, also insgesamt $n + 2$ elementaren Rechenschritten. Die Aufwandsfunktion für die Eingabe ist somit in allen drei Fällen aus $\mathcal{O}(n)$; man sagt, die Eingabe erfordert **linearer Aufwand**.

b) **Ausgaben:** Die Aufwandsfunktion für die Ausgabe ist bei allen drei Sortierprogrammen von der gleichen asymptotischen Größenordnung wie die Aufwandsfunktion für die jeweilige Eingabe, also auch aus $\mathcal{O}(n)$, denn es werden jeweils $n + 3$ Drucke-Anweisungen ausgeführt.

c) **Sortierverfahren:** Der Sortiervorgang des Programms `Insort` besteht aus zwei geschachtelten Schleifen. Dabei durchläuft die äußere for-Schleife für `k` die Zahlen von 1 bis n. Bei der inneren while-Schleife sind im ungünstigsten Fall jeweils `k` $- 1$ Vertauschungen auszuführen, wenn nämlich das Feld absteigend sortiert ist und das Feldelement `a[k]` bis zum Feldanfang transportiert werden muß. Ohne die genaue Anzahl der elementaren Rechenschritte bestimmen exakt zu müssen, bekommen wir, daß die Aufwandsfunktion für die innere Schleife aus $\mathcal{O}(n)$ ist. Dies impliziert, daß die Aufwandsfunktion für das Sortieren durch Einsortieren aus $\mathcal{O}(n^2)$ ist. Man spricht hier von einem **quadratischen Aufwand**.

   Das Sortieren durch Minimumsuche hat eine Aufwandsfunktion, die ebenfalls von n^2 asymptotisch dominiert wird. Der Sortiervorgang des Programms `Minsort` besteht nämlich auch aus zwei geschachtelten Schleifen, wobei die äußere for-Schleife für `k` die Zahlen von 1 bis n und die innere while-Schleife für `r` die Zahlen von `k` bis n durchläuft[31].

[31] Man beachte, daß beim Sortieren durch Minimumsuche der exakte Aufwand unabhängig ist vom Inhalt des Feldes, im Gegensatz zum Sortieren durch Einsortieren.

Beim Qicksort-Verfahren, formuliert im Programm `Quicksort`, liegt der ungünstigste Fall vor, wenn das gewählte Element `w = a[(l+r) DIV 2]` bei jeder im Ablauf vorkommenden Partitionierung von `a` gerade der minimale oder maximale im unsortierten Bereich von `a[l]` bis `a[r]` vorkommende Wert ist. Dann wird pro Rekursionsstufe der Prozedur `quicksort` nur dieser Wert nach links oder nach rechts transportiert. Die erforderliche Aufwandsfunktion dafür ist aus $\mathcal{O}(n)$. Weiterhin sind in diesem Fall $\mathcal{O}(n)$ rekursive Aufrufe erforderlich. Die Aufwandsfunktion für das Quicksort-Verfahren ist somit im ungünstigsten Fall aus $\mathcal{O}(n^2)$. Diese spezielle Feldbelegung ist aber relativ unwahrscheinlich. Wie man durch eine wahrscheinlichkeitstheoretische Analyse zeigen kann, benötigt das Verfahren im Mittel $\mathcal{O}(n*log_2 n)$ Rechenschritte zum Sortieren. ■

Hat man polynomial-zeitbeschränkte Algorithmen als praktikabel definiert, so ist es demnach ein Kernanliegen der Informatik, die Probleme, für welche praktikable Algorithmen existieren, von jenen zu trennen, für die das nicht zutrifft. Leider ist diese Kernfrage noch offen. Im folgenden wollen wir sie etwas erläutern und Teilergebnisse angeben, die man erzielt hat. Dabei bleiben wir aber informell. In voller mathematischer Strenge wird das Thema in einer Vorlesung über theoretische Informatik oder Komplexitätstheorie behandelt.

8.2.7 Die Klassen $\mathcal{P}$ und $\mathcal{NP}$ (informell)

Gegeben sei eine Funktion $f : M \longrightarrow N$, die ein Problem beschreibt. Weiterhin sei zu jedem Argument $x \in M$ von f eine Problemgröße $n_x \in \mathbb{N}$ definiert.

a) f heißt **polynomial-zeitberechenbar**, falls es (bezüglich der Festlegung der Problemgröße) einen polynomial-zeitbeschränkten Algorithmus gibt, der zu jedem $x \in M$ den Funktionswert $f(x) \in N$ berechnet.

b) f heißt **nichtdeterministisch polynomial-zeitberechenbar**, wenn zu jedem $x \in M$ eine Zahl $k \in \mathbb{N}$ existiert, so daß die beiden folgenden Bedingungen gelten:

   (1) Der Funktionswert $f(x) \in N$ kann durch Betrachtung von $e(n_x)$ Fällen algorithmisch berechnet werden, wobei $e \in \mathcal{O}(k^{n_x})$ mit $k \in \mathbb{N}$ gilt[32].

   (2) Die problembeschreibenden Funktionen $f_1, f_2, \ldots, f_{e(n_x)}$ eines jeden einzelnen Falls aus (1) sind polynomial-zeitberechenbar.

Mit $\mathcal{P}$ bezeichnet man die Klasse der polynomial-zeitberechenbaren Funktionen; mit $\mathcal{NP}$ die Klasse der nichtdeterministisch polynomial-zeitberechenbaren Funktionen. ■

Offensichtlich gilt $\mathcal{P} \subseteq \mathcal{NP}$, denn trifft $f \in \mathcal{P}$ zu, so hat man einen einzelnen Fall, $e \in \mathcal{O}(1^{n_x})$, den man polynomial zeitberechnen kann.

[32] Dies heißt, daß man exponentiell viele Fälle zu betrachten hat.

Durch die bisher angegebenen Programme haben wir schon viele Beispiele für polynomial-zeitberechenbare Funktionen kennengelernt, etwa die Sortierfunktion $\mathit{sort} : \mathbb{Z}^* \longrightarrow \mathbb{Z}^*$ auf Listen, für die beispielsweise Quicksort ein polynomial-zeitbeschränkter Algorithmus ist. Nachfolgend geben wir ein Problem an, dessen beschreibende Funktion nichtdeterministisch polynomial-zeitberechenbar ist, von der man aber noch nicht weiß, ob sie auch polynomial-zeitberechenbar ist.

8.2.8 Beispiel (Tautologietest)

Gegeben sei eine Formel φ, wie beim Hoare-Kalkül eingeführt, die nun aber einschränkenderweise aufgebaut ist nur mittels der Booleschen Variablen $x_1, \ldots, x_k$, der Verknüpfungen $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$, sowie Klammerung, also ohne echte Boolesche Terme und Quantoren. Die Formel φ heißt eine **Tautologie**, falls sie für alle Werte der Variablen aus $\mathbb{B} = \{\mathit{tt}, \mathit{ff}\}$ immer wahr ist.

Beispielsweise ist, für $k = 1$, die Formel $x_1 \vee \neg x_1$ eine Tautologie, wie man an der nachfolgenden Tabelle sofort sieht[33]:

Wert von x_1	Wert von $x_1 \vee \neg x_1$
ff	*tt*
tt	*tt*

Hingegen ist, nun für $k = 2$, die Formel $x_1 \vee \neg x_2$ keine Tautologie. Auch dies macht man sich am schnellsten klar, indem man alle ihre Werte für alle vier möglichen Kombinationen der Variablenwerte ausrechnet:

Wert von x_1	Wert von x_2	Wert von $x_1 \vee \neg x_2$
ff	*ff*	*tt*
ff	*tt*	*ff*
tt	*ff*	*tt*
tt	*tt*	*tt*

Wegen der zweiten Kombination ist die Formel $x_1 \vee \neg x_2$ keine Tautologie.

Es bezeichne nun $\mathcal{F}$ die Menge aller oben eingeführten Formeln. Das Problem des **Tautologietests**, ausgedrückt durch die problembeschreibende Funktion

$$f_{\mathrm{taut}} : \mathcal{F} \longrightarrow \mathbb{B} \qquad\qquad f_{\mathrm{taut}}(\varphi) = \mathit{tt} \iff \varphi \text{ ist eine Tautologie},$$

ist in der Klasse $\mathcal{NP}$ enthalten, wenn man eine Auswertung einer der Verknüpfungen $\neg$, $\wedge$, $\vee$, $\rightarrow$ und $\leftrightarrow$ jeweils als einen elementaren Rechenschritt zählt und die Problemgröße n_φ durch die Anzahl der Verknüpfungen der Formel φ definiert. Man beachte, daß die Beziehung $k \in \mathcal{O}(n_\varphi)$ gilt.

[33] Wir identifizieren hier Formeln mit Booleschen Termen und bezeichnen den Wert einer Formel genau dann mit dem Wahrheitswert *tt*, falls die Formel gültig ist.

Zum Beweis von $f_{\text{taut}} \in \mathcal{NP}$ betrachten wir zu einem Argument $\varphi \in \mathcal{F}$ der Funktion die nachfolgend skizzierte Tabelle, in der alle 2^k möglichen Kombinationen der Werte der Variablen x_1 bis x_k in den Zeilen unter dem Querstrich aufgeführt sind:

Wert von x_1	Wert von x_2	...	Wert von x_{k-1}	Wert von x_k	Wert von φ
ff	*ff*	...	*ff*	*ff*	
ff	*ff*	...	*ff*	*tt*	
ff	*ff*	...	*tt*	*ff*	
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	
tt	*tt*	...	*tt*	*tt*	

Die Aufgabe, den Funktionswert $f(\varphi)$ zu bestimmen, kann also in die $2^k \in \mathcal{O}(2^{n_\varphi})$ Fälle aufgeteilt werden, jeweils den Wert der Formel φ für die durch Zeile i der Tabelle gegebenen Werte der Variablen x_1 bis x_k zu berechnen ($1 \leq i \leq 2^k$). Offensichtlich ist jeder einzelne Fall polynomial-zeitberechenbar. ■

Es gibt viele praktisch sehr relevante Probleme, von denen man in den letzten drei Jahrzehnten zeigen konnte, daß die problembeschreibenden Funktionen in der Klasse $\mathcal{NP}$ enthalten sind. Dies heißt, daß man für all diese Probleme zumindest einen Algorithmus kennt, der in exponentiell vielen Schritten zu jeder Eingabe eine Ausgabe berechnet. Der Nachweis, daß alle diese Probleme auch in $\mathcal{P}$ liegen, glückte aber bisher nicht immer. Für die noch ausstehenden Fälle – und einer davon ist, wie oben schon erwähnt, der Tautologietest – kennt man also bisher noch keine praktikablen Lösungsalgorithmen.

Die Frage, ob jedes algorithmisch lösbare Problem auch durch einen praktikablen Algorithmus gelöst werden kann, ist somit äquivalent dazu, ob die Gleichung $\mathcal{P} = \mathcal{NP}$ gilt. Aufgrund der ungezählten Bemühungen ungezählter Informatikforscher in den letzten drei Jahrzehnten **spricht vieles dafür, daß $\mathcal{P} = \mathcal{NP}$ nicht gilt**. Wahrscheinlich existieren also Probleme, die zwar algorithmisch lösbar sind, nicht aber durch praktikable Algorithmen. Der Tautologietest scheint eines dieser Probleme zu sein, denn es wurde für seine problembeschreibende Funktion f_{taut} gezeigt

$$\mathcal{P} = \mathcal{NP} \iff f_{\text{taut}} \in \mathcal{P}.$$

Eine genaue mathematische Darstellung der eben nur informell aufgezeigten Resultate findet man etwa in dem in der Einleitung zitierten Buch „Fundamentals of computer algorithms“ von E. Horowitz und S. Sahni. Wie oben schon erwähnt, wird auch auf Vorlesungen über theoretische Informatik und Komplexitätstheorie hingewiesen.

8.3 Binäre Suchbäume

Nach den doch mehr theoretischen Gedanken zu Effizienz und Komplexität im letzten Abschnitt kehren wir nun wieder zu konkreten und für die Praxis wichtigen Problemstellungen und deren algorithmischen Lösungen zurück. Eine in der Praxis häufig auftretende Problemstellung ist die nachfolgend geschilderte. Man hat als Voraussetzungen gegeben:

(1) Eine **Schlüsselmenge** $\mathcal{S}$, die **linear geordnet** ist, d.h. für $s_1, s_2 \in \mathcal{S}$ gilt $s_1 \leq s_2$ oder $s_2 \leq s_1$.

(2) Eine nicht näher spezifizierte Menge $\mathcal{I}$ von **Informationsobjekten**.

Man betrachtet nun Listen von Paaren $\langle s_1, I_1\rangle, \langle s_2, I_2\rangle, \ldots, \langle s_k, I_k\rangle$ mit $s_i \in \mathcal{S}$ und $I_i \in \mathcal{I}$ für alle i mit $1 \leq i \leq k$. Auf solchen Listen möchte man nun gerne (mindestens) die folgenden drei Operationen durchführen:

1. **Vergrößern** einer Liste um ein Paar $\langle s, I\rangle$.
2. **Entfernen** eines vorgegebenen Paares $\langle s, I\rangle$ aus einer Liste.
3. **Test**, ob ein vorgegebenes Paar $\langle s, I\rangle$ in einer Liste enthalten ist.

Eine Realisierung der Listen durch Felder scheidet in der Praxis aus, da man die Maximallänge nicht kennt und Felder eine feste Obergrenze haben. Als nächstes bietet sich eine direkte Listenimplementierung an, was in ML als eingebaute Datenstruktur gegeben ist und in Modula-2 mittels Zeigern bewerkstelligt werden kann. Damit hat man aber lange Suchzeiten beim Enthaltensein-Test, im Durchschnitt $\frac{k}{2}$ Rechenschritte, was sich natürlich auch negativ auf die Effizienz des Entfernens auswirkt, da das zu entfernende Element erst gesucht werden muß.

Die Grundidee einer noch schnelleren Suche ist die gleiche wie beim Suchen im Telephonbuch. Man schlägt hier normalerweise ungefähr in der Mitte auf und vergleicht bzw. sucht dann durch fortwährende Halbierung. Eine Suche durch fortwährende Halbierung des Suchraums läßt sich sehr effizient durch **Binärbäume** realisieren. Diese Datenstruktur kennen wir bereits von der funktionalen Programmierung her, siehe Abschnitt 5.2. Damals begannen wir die Induktion beim Aufbau der Binärbäume mit den atomaren („einknotigen“) Bäumen. Da wir jedoch oben auch den Fall $k = 0$ der leeren Liste von Paaren zu berücksichtigen haben, müssen wir nun die Induktion mit dem leeren Baum starten. Aus diesem Grund verwenden wir auch die Bezeichnung $\mathsf{Btree}(A)$ aus Abschnitt 5.2 im weiteren nicht, sondern führen die neue Bezeichnung $\mathsf{BT}(\mathcal{S})$ ein. Der Einfachheit halber unterdrücken wir auch die Informationsobjekte. Entscheidend für das weitere Vorgehen ist der in Teil c) der nachfolgenden Definition formulierte Begriff.

8.3.1 Definition (Binärer Suchbaum)

Es sei $(\mathcal{S}, \leq)$ eine linear geordnete Menge von Schlüsseln.

a) Die Menge $\mathsf{BT}(\mathcal{S})$ der **knotenmarkierten Binärbäume** über der Schlüsselmenge $\mathcal{S}$ ist zusammen mit der **Knotenmarkierung** $nodes : \mathsf{BT}(\mathcal{S}) \longrightarrow 2^{\mathcal{S}}$ und der **Teilbaumfunktion** $st : \mathsf{BT}(\mathcal{S}) \longrightarrow 2^{\mathsf{BT}(\mathcal{S})}$ induktiv wie folgt definiert:

   (1) Es gelten $etree \in \mathsf{BT}(\mathcal{S})$, $nodes(etree) = \emptyset$ und $st(etree) = \{etree\}$.

(2) Falls $l, r \in \mathsf{BT}(\mathcal{S})$ und $s \in \mathcal{S}$, so gilt auch $\langle l, s, r\rangle \in \mathsf{BT}(\mathcal{S})$ und es ist definiert

$$\begin{array}{rrcl} \text{i)} & nodes(\langle l, r, s\rangle) & = & nodes(l) \cup \{s\} \cup nodes(r) \\ \text{ii)} & st(\langle l, r, s\rangle) & = & st(l) \cup \{\langle l, s, r\rangle\} \cup st(r). \end{array}$$

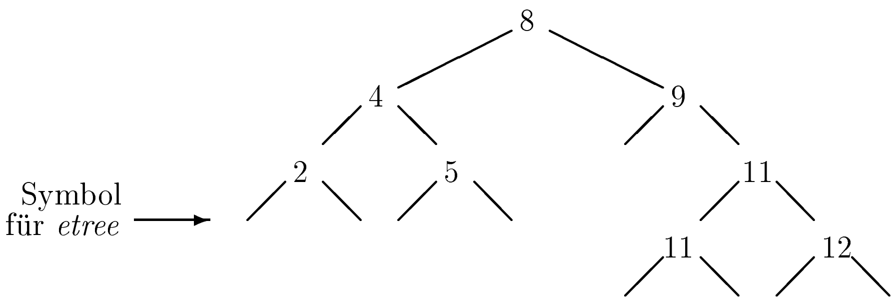
b) Zu $t \in \mathsf{BT}(\mathcal{S})$ heißt $b \in st(t)$ ein **Teilbaum** und zu $\langle l, s, r\rangle \in \mathsf{BT}(\mathcal{S})$ heißt s die **Wurzelmarkierung**, l der **direkte linke Teilbaum** und r der **direkte rechte Teilbaum**. Weiterhin nennt man *etree* den **leeren Baum**.

c) Ein Element $t \in \mathsf{BT}(\mathcal{S})$ heißt ein **binärer Suchbaum** über der Schlüsselmenge $\mathcal{S}$, falls für jeden Teilbaum $\langle l, s, r\rangle \in st(t)$ und für alle Schlüssel $s_1 \in nodes(l)$ und $s_2 \in nodes(r)$ gilt $s_1 \leq s < s_2$. Es ist $\mathsf{ST}(\mathcal{S})$ die Menge der binären Suchbäume über der Schlüsselmenge $\mathcal{S}$. ■

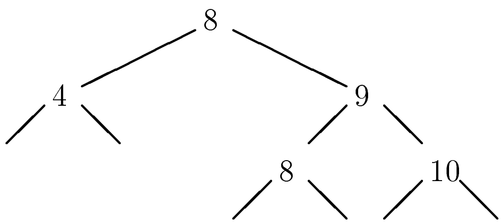
Im folgenden stellen wir, wie schon in Abschnitt 5.2, Binärbäume und binäre Suchbäume als Diagramme mit Linien (Kanten) und Knoten (Markierungen) dar.

8.3.2 Beispiele (für Bäume)

Es sei die Schlüsselmenge $\mathcal{S}$ die Menge $\mathbb{N}$ der natürlichen Zahlen, versehen mit mit der üblichen Ordnung. Dann ist



ein binärer Suchbaum, wie man leicht durch das Überprüfen aller möglichen Teilbäume erkennt. Hingegen ist der knotenmarkierte Binärbaum



kein binärer Suchbaum Für die Wurzelmarkierung 8 des gesamten Baums und die Knotenmarkierung 8 im direkten rechten Teilbaum gilt nämlich nicht die in Definition 8.3.1 geforderte Abschätzung $8 < 8$. ■

Wir kehren nun zur Implementierung von Listen zurück. Da wir die Informationsobjekte unterdrücken, haben wir Listen $s_1, s_2, \ldots, s_k$, mit $s_i \in \mathcal{S}$ für alle i mit $1 \leq i \leq k$, zu betrachten sowie die obigen drei Operationen auf der Menge all dieser Listen. Im folgenden entwickeln wir entsprechende Programmteile in Modula-2. Es sei an dieser Stelle darauf hingewiesen, daß eine ML-Implementierung wesentlich einfacher wäre. In Abschnitt 8.4 werden wir jedoch den Ansatz dieses Abschnitts noch verfeinern, und dieses kann dann in ML nicht mehr effizient bewerkstelligt werden - im Gegensatz zu Modula-2 mit seinen Zeigern. Deshalb beginnen wir gleich mit der Implementierung der binären Suchbäume in der Programmiersprache Modula-2.

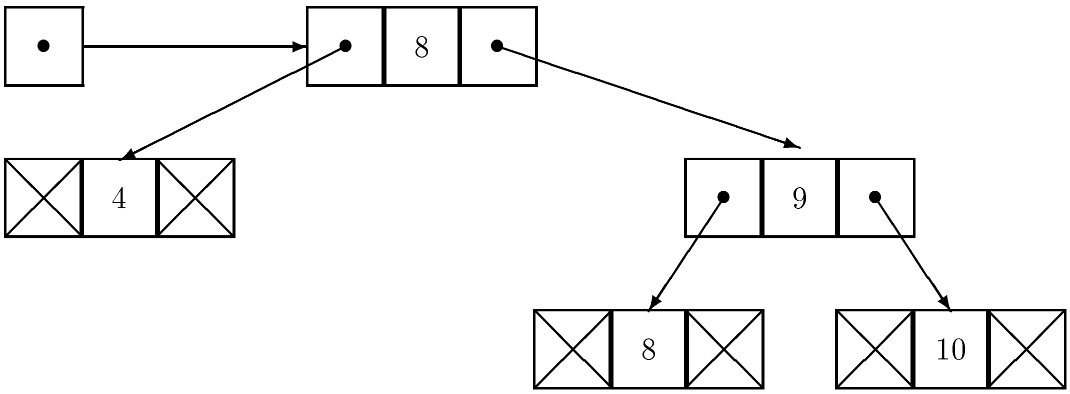
8.3.3 Ein Typ für Binärbäume

Mit Hilfe von Zeigern ist es einfach, Binärbäume darzustellen. Im folgenden nehmen wir, der Einfachheit halber, $\mathbb{N}$ als Schlüsselmenge an. Dann erhalten wir durch die Deklarationen

```
TYPE key = CARDINAL;
     searchtree = POINTER TO node;
     node = RECORD left : searchtree;
                   root : key;
                   right : searchtree
               END;
```

einen Typ `searchtree` zur Darstellung von knotenmarkierten Binärbäumen. Dem leeren Baum entspricht dabei der Zeiger `NIL`, die direkten Teilbäume bekommt man durch die Selektoren. `left` und `right` und die Wurzelmarkierung durch den Selektor `root`. ■

Beispielsweise stellt sich der zweite knotenmarkierte Binärbaum von Beispiel 8.3.2, der kein binärer Suchbaum ist, als Zeigergeflecht wie folgt dar:



Die in Abschnitt 8.3.3 deklarierten zwei Typen `node` und `searchtree` implementieren beliebige knotenmarkierte Binärbäume, mit `key` als Typ für die einzelnen Knotenmarkierungen. Es ist offensichtlich, daß mit Hilfe einer vierten Komponente im Verbundtyp `node` auch noch die Informationsobjekte integriert werden könnten.

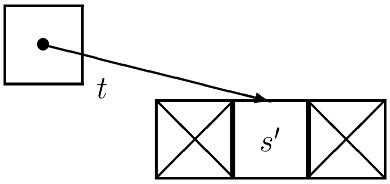
Wir kommen nun zu den Operationen „Vergrößern“, „Entfernen“ und „Test“. Die ersten beiden Operationen manipulieren knotenbewertete Binärbäume. Wir werden sie so entwerfen, daß **die Eigenschaft, ein binärer Suchbaum zu sein, bei der Anwendung jeweils erhalten bleibt**. Fängt man dann im Laufe von Baummanipulationen den Baumaufbau mit dem leeren Suchbaum *etree* an, so bleibt man innerhalb der Menge der binären Suchbäume. Dies erlaubt schließlich, den Test auf Enthaltensein durch fortwährendes Halbieren effizient zu realisieren.

8.3.4 Einfügen in binäre Suchbäume (Vergrößern)

Es seien $\langle l, s, r\rangle \in \mathsf{ST}(\mathcal{S})$ ein binärer Suchbaum und $s' \in \mathcal{S}$ ein Schlüssel. Wir wollen einen neuen mit s' markierten Knoten einfügen. Das Vorgehen dabei ist wie folgt:

- Gilt $s' \leq s$, so haben wir den s'-Knoten im direkten linken Teilbaum l einzufügen.
- Gilt $s' > s$, so haben wir den s'-Knoten im direkten rechten Teilbaum r einzufügen.

Es bleibt der Fall, einen neuen mit s' markierten Knoten in den leeren Baum einzufügen. Dies heißt, einen neuen Baum der Form $\langle \mathit{etree}, s', \mathit{etree}\rangle$ zu erzeugen. Da der Zeiger `NIL` den leeren Baum darstellt, sieht das Zeigergeflecht für $\langle \mathit{etree}, s', \mathit{etree}\rangle$ wie folgt aus:



Die eben aufgezeigte rekursive Vorgehensweise überträgt sich sofort nach Modula-2 und führt zu der folgenden rekursiven Prozedur `insert` zum Einfügen von `s` in `t`:

```
PROCEDURE insert (VAR t : searchtree; s : key);
  BEGIN
  IF t = NIL
    THEN NEW(t);
         t^.left := NIL;
         t^.root := s;
         t^.right := NIL
    ELSE IF s <= t^.root
           THEN insert(t^.left,s)
           ELSE insert(t^.right,s)
         END
  END
END insert;
```

Die Anweisungsfolge im äußeren then-Zweig dieser Prozedur erzeugt dabei genau das oben angegebene einfache Zeigergeflecht zur Darstellung von $\langle \mathit{etree}, s, \mathit{etree}\rangle$. ■

Nachdem wir das Einfügen eines Schlüssels in eines binären Suchbaum durch eine rekursive Prozedur beschrieben haben, wenden wir uns nun der inversen Operation zu, dem Entfernen eines Schlüssels aus einem binären Suchbaum. Dies ist, wie sich nachfolgend zeigen wird, von allen drei Operationen, die wir betrachten, die am schwierigste zu implementierende.

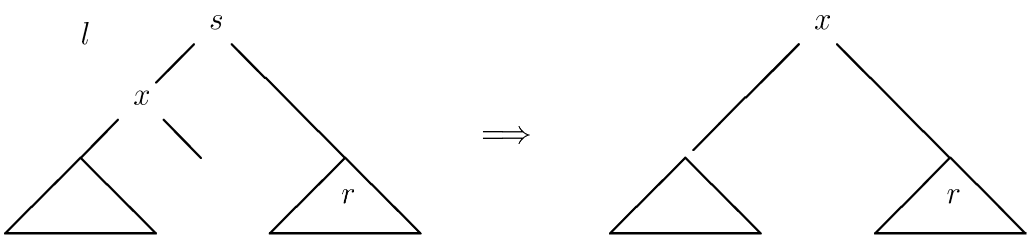
8.3.5 Entfernen aus binären Suchbäumen (ein Vorkommen)

Beim Entfernen eines Schlüssels s aus einem binären Suchbaum – genauer, beim Entfernen eines Knoten mit Marke s – gibt es die folgenden zwei Varianten. Entweder wird **genau ein Vorkommen** des spezifizierten Schlüssels entfernt oder **alle Vorkommen** des spezifizierten Schlüssels werden entfernt. Wir verwenden binäre Suchbäume zur Implementierung von Listen. Deshalb ist es naheliegend, sich auf die erste Variante zu beschränken. Die Realisierung eines Verfahrens zum einmaligen Entfernen eines Vorkommens eines Schlüssels geschieht wiederum rekursiv und erfordert eine Reihe von Fallunterscheidungen.

1. Ist der gegebene binäre Suchbaum leer, so bewirkt das Entfernen eines Schlüssels nichts.
2. Nun sei der gegebene binäre Suchbaum nicht leer und habe die Form $\langle l, s, r\rangle$. Wir wollen einen mit s' markierten Knoten aus $\langle l, s, r\rangle$ entfernen und gehen dabei wie folgt rekursiv vor:
   - 2.1 Gilt $s' < s$, so liegt s' im direkten linken Teilbaum l und muß dort einmal entfernt werden.
   - 2.2 Gilt $s' > s$, so liegt s' im direkten rechten Teilbaum r und muß dort einmal entfernt werden.
   - 2.3 Gilt $s' = s$, so ist die Wurzel so zu entfernen, daß die Suchbaumeigenschaft erhalten bleibt. Hier gibt es wiederum zwei Fälle:
     - $*$ Ist der direkte linke Teilbaum leer, so entfernt man offensichtlich s, indem man zum direkten rechten Teilbaum r übergeht.
     - $*$ Ist der direkte linke Teilbaum nicht leer, so bestimmt man ein maximales Element $x \in \mathit{nodes}(l)$. Dann entfernt man die Wurzel von $\langle l, s, r\rangle$, indem man s durch x ersetzt und x in l einmal entfernt.

Wo findet man nun einen maximalen Schlüssel $x \in \mathit{nodes}(l)$ als Kandidaten für eine neue Wurzelmarkierung, wie im zweiten Fall von 2.3 gefordert? Doch offensichtlich im direkten rechten Teilbaum von l und davon wiederum im direkten rechten Teilbaum und so weiter. Im Prinzip ist diese Überlegung korrekt, im Detail erfordert die Suche nach x, die Ersetzung von s durch x und die Entfernung von x auf die eben geschilderte Weise jedoch noch eine weitere Fallunterscheidung.

Hat der direkte linke Teilbaum l keinen echten direkten rechten Teilbaum, dann ist offensichtlich das gesuchte x gerade seine Wurzelmarkierung. Man „schließt also kurz" und ersetzt s durch x wie in dem nachfolgenden Bild angedeutet:



Existiert hingegen ein echter direkter rechter Teilbaum des direkten linken Teilbaums l, so läuft man von der l-Wurzel ausgehend – bildlich gesprochen – solange nach rechts, bis man einen Knoten erreicht, dessen rechter Nachfolger (das gesuchte x) keinen rechten Nachfolger mehr hat. Dann wird x die neue Wurzelmarkierung und der rechte Nachfolger des Vorgängers von x der linke Nachfolger von x. Der Leser mache sich diese Baummanipulation an einem Bildchen klar! Die Suche nach dem Knoten mit der Markierung x erfordert in der Modula-2-Zeigerimplementierung von binären Suchbäumen einen Hilfszeiger.

Formulieren wir den eben skizzierten Algorithmus in der Programmiersprache Modula-2, so führt dies zur rekursiven Prozedur

```
PROCEDURE delete (VAR t : searchtree; s : key);
  VAR h : searchtree;
  BEGIN
  IF t = NIL
    THEN (* SKIP *)
    ELSE IF s <= t^.root
           THEN delete(t^.left,s) END;
         IF s > t^.root
           THEN delete(t^.right,s) END;
         IF s = t^.root
           THEN IF t^.left = NIL
                  THEN t := t^.right
                  ELSE h := t^.left;
                       IF h^.right = NIL
                         THEN t^.root := h^.root;
                              t^.left := h^.left
                         ELSE WHILE h^.right^.right # NIL DO
                                h := h^.right END;
                              t^.root := h^.right^.root;
                              h^.right := h^.right^.left
                       END
              END END
  END
END delete;
```

zum Entfernen eines Vorkommens von `s` in `t`. Eine Modifikation dieser Prozedur `delete`, so daß durch die nene Prozedur alle Vorkommen von `s` in `t` entfernt werden, ist offensichtlich und sei dem Leser zur Übung empfohlen. ■

Nachdem wir die Operationen, die binäre Suchbäume verändern, durch rekursive Prozeduren in Modula-2 implementiert haben, wenden wir uns nun der verbleibenden Operation zu, die zu einem binäre Suchbaum einen Wahrheitswert berechnet. In der Modula-2-Implementierung wird dies zu einer rekursiven Rechenvorschrift führen.

8.3.6 Test auf Enthaltensein bei binären Suchbäumen

Es sei $isin : \mathsf{ST}(\mathcal{S}) \times \mathcal{S} \mapsto \mathbb{B}$ eine Funktion, so daß $isin(t, s') = tt$ genau dann gilt, wenn der Schlüssel s' in t vorkommt. Eine formale induktive Definition von $isin$ ist

$$\begin{array}{lrcl} (1) & isin(etree, s') & = & \mathit{ff} \\ (2) & isin(\langle l, s, r\rangle, s') & = & isin(l, s') \vee (s = s') \vee isin(r, s'). \end{array}$$

Diese Gleichungen beschreiben auch einen Algorithmus. Wir wollen jedoch einen effizienteren gewinnen. Dazu formen wir die zweite Gleichung um, indem wir – geleitet durch die Suchbaum-Eigenschaft – drei passende Fallunterscheidungen einführen: Gilt $s' < s$, so treffen die beiden rechten Bedingungen $s = s'$ und $isin(r, s')$ von (2) nicht zu, letztere nicht wegen der Suchbaum-Eigenschaft von $\langle l, s, r\rangle$. Also gilt hier

$$isin(\langle l, s, r\rangle, s') = isin(l, s'). \tag{E_1}$$

Analog zeigt man im Fall $s' > s$ die Gleichung

$$isin(\langle l, s, r\rangle, s') = isin(r, s'). \tag{E_2}$$

Gilt schließlich die Gleichheit $s' = s$ für die Schlüssel, so folgt aus (2) sofort die Gleichung

$$isin(\langle l, s, r\rangle, s') = tt. \tag{E_3}$$

Verwenden wir nun die Gleichungen (E_1) bis (E_3) statt der originalen Gleichung (2), so erhalten wir in Verbindung mit (1) eine terminierende Rekursion für die Funktion $isin$, welche kaskadenartige Aufrufe vermeidet und deren Abarbeitung damit effizienter bewerkstelligt werden kann. In Modula-2 stellt sich diese Rekursion durch

```
PROCEDURE isin (t : searchtree; s : key): BOOLEAN;
  BEGIN
  IF t = NIL
    THEN RETURN FALSE
    ELSE IF t^.root > s THEN RETURN isin(t^.left,s) END;
         IF t^.root = s THEN RETURN TRUE END;
         IF t^.root < s THEN RETURN isin(t^.right,s) END
  END
END isin;
```

in Form einer rekursiven Rechenvorschrift dar. ■

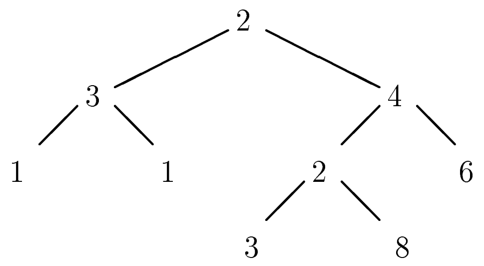
Binäre Suchbäume werden normalerweise zur Speicherung und Verwaltung von Daten wie Telephonbücher, Personallisten, Konten etc. verwendet. Mit Suchbäumen kann man aber auch sortieren, wie wir nun demonstrieren. Dazu betrachten wir die zuerst **Linearisierung** von knotenmarkierten Binärbäumen. Davon gibt es drei spezielle Arten. Wir beschreiben diese durch drei Funktionen.

$$preorder, postorder, inorder : \mathsf{BT}(\mathcal{S}) \mapsto \mathcal{S}^*,$$

die wie folgt induktiv definiert sind:

$$\begin{array}{lrcl}
(1) & preorder(etree) & = & \varepsilon \\
(2) & preorder(\langle l, s, r \rangle) & = & s\, preorder(l)\, preorder(r) \\
(3) & postorder(etree) & = & \varepsilon \\
(4) & postorder(\langle l, s, r \rangle) & = & postorder(l)\, postorder(r)\, s \\
(5) & inorder(etree) & = & \varepsilon \\
(6) & inorder(\langle l, s, r \rangle) & = & inorder(l)\, s\, inorder(r)
\end{array}$$

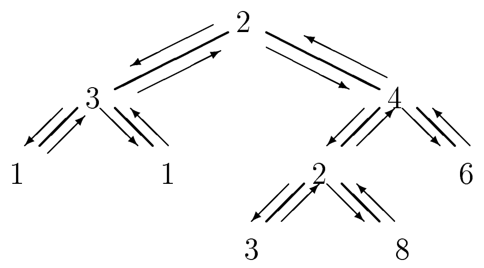
Diese Liniarisierungen unterscheiden sich also nur dadurch, wie die Wurzelmarkierung mit den Linearisierungen der direkten Teilbäume kombiniert wird. Zur Verdeutlichung dieser drei Funktionen betrachten wir den nachfolgend bildlich angegebenen knotenmarkierten Binärbaum t, wobei wir die 10 Striche für den leeren Binärbaum aus Gründen der Lesbarkeit weggelassen haben.



Wie man leicht nachrechnet, gelten die drei Gleichungen

$$\begin{array}{rcl}
preorder(t) & = & \langle 2, 3, 1, 1, 4, 2, 3, 8, 6 \rangle \\
postorder(t) & = & \langle 1, 1, 3, 3, 8, 2, 6, 4, 2 \rangle \\
inorder(t) & = & \langle 1, 3, 1, 2, 3, 2, 8, 4, 6 \rangle.
\end{array}$$

Man kann sich insbesondere die Wirkungsweise der beiden Funktionen *preorder* und *postorder* gut verdeutlichen, indem man den gegebenen knotenmarkierten Binärbaum durchläuft. Nachfolgend ist ein Durchlauf für den obigen Baum t angegeben:



Schreibt man, ausgehend von der Wurzel, die besuchten Knoten in der Reihenfolge auf, in der man sie während des Durchlaufs zum ersten Male erreicht, so ergibt sich die Liste $\langle 2, 3, 1, 1, 4, 2, 3, 8, 6 \rangle$, also hat man den preorder-Durchlauf.

8.3.7 Sortieren durch Suchbäume

Ist $t \in \mathsf{ST}(\mathcal{S})$ ein Suchbaum, so ist die Liste $inorder(t)$ offensichtlich aufsteigend sortiert. Damit ergibt sich sofort ein Sortierverfahren, indem man zuerst aus den gegebenen Elementen einen binären Suchbaum aufbaut und diesen dann im inorder-Verfahren linearisiert. Eine Modula-2-Realisierung dieser Vorgehensweise ist nachfolgend angegeben.

```
MODULE SearchtreeSort;
FROM InOut IMPORT WriteLn, WriteString, ReadCard, WriteCard;
FROM Storage IMPORT ALLOCATE;
TYPE key = CARDINAL;
     searchtree = POINTER TO node;
     node = RECORD .......... END;
VAR k : key;
    t : searchtree;
PROCEDURE insert (VAR t : searchtree; s: key);
  BEGIN ............ END insert;
PROCEDURE sort (VAR t : searchtree);
  BEGIN
  IF t # NIL THEN sort(t^.left);
                  WriteCard(t^.root,5);
                  sort(t^.right)
             ELSE (* SKIP *) END
  END sort;
BEGIN
  WriteString('Gib Liste positiver natuerlicher Zah-');
  WriteString('len ein (mit 0 als Abschlusszeichen).');
  WriteLn;
  t := NIL;
  REPEAT ReadCard(k);
         IF k # 0 THEN insert(t,k) END
  UNTIL k = 0;
  sort(t);
  WriteLn
END SearchtreeSort .
```

Durch dieses Programm werden die zu sortierenden Zahlen der Reihe nach vom Benutzer eingegeben, wobei der Aufbau des Suchbaums erfolgt. Anschließend wird dieser Suchbaum linearisiert, wobei aber kein Objekt irgendeines Modula-2-Typs für Listen erzeugt wird, sondern die sortierte Liste der Zahlen direkt auf dem Bildschirm erscheint. ■

8.4 Balancierte binäre Suchbäume

Das Verhalten der binären Suchbäume ist nur dann befriedigend, wenn jeder vorliegende Baum annähernd ausgeglichen ist, d.h. beispielsweise die Entfernungen aller Blätter von der Wurzel (als Längen der Kantenzüge definiert) annähernd gleich sind oder die Anzahl der Knoten seines direkten linken Teilbaums in etwa der Anzahl der Knoten seines direkten rechten Teilbaums entspricht und diese Eigenschaft auch für alle inneren, d.h. nicht direkten, Teilbäume gilt. Erst dann ist der Aufwand der im letzten Abschnitt entwickelten Modula-2-Implementierungen der drei Operationen „Vergrößern", „Entfernen" und „Test" auf Suchbäumen kleiner als $\mathcal{O}(n)$, mit n als der Knotenzahl, etwa $\mathcal{O}(log_2 n)$ beim vollständigen Binärbaum. Entartet hingegen ein binärer Suchbaum zu einer linearen Liste

$$\langle etree, s_1, \langle etree, s_2, \langle \ldots \langle etree, s_n, etree \rangle \ldots \rangle \rangle \rangle,$$

was etwa geschieht, indem man die Elemente der Kette $s_1 \leq s_2 \leq \ldots \leq s_n$ von Schlüsseln in aufsteigender Reihenfolge in den leeren Baum *etree* einfügt, so erfordern die eben erwähnten Operationen linearen Aufwand. Als eine Konsequenz ist der Aufwand des Sortierverfahrens des letzten Abschnitts quadratisch.

In der Praxis arbeitet man deshalb mit ausgeglichenen (im Jargon: **balancierten**) Suchbäumen. Es gibt, wie aus den oben erwähnten zwei Kriterien ersichtlich ist, mehrere Möglichkeiten zum Balancieren. Die im folgenden vorgestellte Möglichkeit der Höhenbalance geht auf die zwei Russen **A**delson-**V**elsky und **L**andis (1962) zurück und nimmt an, daß **alle in einem knotenmarkierten Binärbaum vorkommenden Schlüssel paarweise verschieden sind**. Damit ist es bei diesem Ansatz erlaubt, **Knoten und deren Markierung zu identifizieren**. Im folgenden bezeichne $nn(t)$ die **Knotenanzahl** eines Baums $t \in \mathsf{BT}(\mathcal{S})$; die formale induktive Definition sei dem Leser überlassen.

8.4.1 Definition (AVL-Baum)

a) Als **Höhe** eines Baums bezeichnet man die Länge eines längsten Kantenzugs von der Wurzel zu einem Blatt (oder umgekehrt). Für knotenmarkierte Binärbäume läßt sie sich formal induktiv definieren durch

$$\begin{array}{ll} (1) & height(etree) = 0 \\ (2) & height(\langle l, s, r \rangle) = 1 + max(height(l), height(r)). \end{array}$$

b) Ein binärer Suchbaum $t \in \mathsf{ST}(\mathcal{S})$ heißt ein **AVL-Baum**, falls $nn(t) = |nodes(t)|$ und für alle Teilbäume $\langle l, s, r \rangle \in st(t)$ gilt $|height(l) - height(r)| \leq 1$. Mit $\mathsf{AVL}(\mathcal{S})$ bezeichnen wir die Menge aller AVL-Bäume über $\mathcal{S}$. ■

Der große Vorteil von AVL-Bäumen ergibt sich aus dem nachfolgenden Satz 8.4.2. Insbesondere folgt aus diesem Satz nämlich, daß bei einem AVL-Baum t der Test auf Enthaltensein in der Modula-2-Implementierung des Abschnitts 8.3.6 die Laufzeitschranke $\mathcal{O}(log_2 m)$ besitzt, mit m als der Knotenzahl von t.

8.4.2 Satz (Höhe eines AVL-Baums)

Ist $t \in \mathsf{AVL}(\mathcal{S})$ ein AVL-Baum mit $m = |nodes(t)|$ und $m \neq 0$, so gilt für seine Höhe die Eigenschaft $height(t) \in \mathcal{O}(log_2 m)$.

Beweis: Es bezeichne $\mathit{fib}(n)$ die n-te Fibonacci-Zahl. Wir zeigen zuerst durch Induktion nach $height(t)$ die Abschätzung

$$|nodes(t)| \geq \mathit{fib}(height(t)). \tag{1}$$

Der Induktionsbeginn $height(t) = 1$ ist trivial. Hier ist t ein Baum mit genau einem Knoten und die Abschätzung wird, wegen $\mathit{fib}(1) = 1$, zu $1 \geq 1$.

Zum Induktionsschluß $height(t) > 1$ habe t die Gestalt $\langle l, s, r\rangle$. Wir unterscheiden zwei Fälle. Zuerst sei die Differenz von $height(l)$ und $height(r)$ gleich 1 und es gelte o.B.d.A. die Gleichung $height(l) = height(r) + 1$. Hier bekommen wir

$$\begin{aligned}
|nodes(t)| &\geq |nodes(l)| + |nodes(r)| && \text{Schlüssel paarweise verschieden}\\
&\geq \mathit{fib}(height(l)) + \mathit{fib}(height(r)) && \text{Induktionshypothese}\\
&= \mathit{fib}(height(t) - 1) + \mathit{fib}(height(t) - 2) && \text{Annahme}\\
&= \mathit{fib}(height(t)) && \text{Fibonacci-Rekursion.}
\end{aligned}$$

Gilt hingegen der zweite noch mögliche Fall $height(l) = height(r)$, so bekommen wir

$$\begin{aligned}
|nodes(t)| &\geq \mathit{fib}(height(t) - 1) + \mathit{fib}(height(t) - 1) && \text{wie eben}\\
&\geq \mathit{fib}(height(t) - 1) + \mathit{fib}(height(t) - 2) && \text{Monotonie Fibonacci-Folge}\\
&= \mathit{fib}(height(t)) && \text{Fibonacci-Rekursion.}
\end{aligned}$$

Als zweiten Beweisteil zeigen wir nun für alle natürlichen Zahlen n die Abschätzung

$$\mathit{fib}(n) \geq 2^{n \; div \; 2}. \tag{2}$$

Es gilt $\mathit{fib}(n) \geq 2 * \mathit{fib}(n-2) \geq 2^2 * \mathit{fib}(n-4) \geq \ldots$ wegen der Fibonacci-Rekursion und der Monotonie der Fibonacci-Folge und wir erhalten (formal natürlich durch Induktion)

$$\mathit{fib}(n) \geq 2^i * \mathit{fib}(n - 2 * i)$$

für alle natürlichen Zahlen $i \leq \frac{n}{2}$. Ist n gerade, so folgt hieraus

$$\mathit{fib}(n) \geq 2^{\frac{n}{2}} * \mathit{fib}(0) = 2^{n \; div \; 2},$$

indem wir i als $\frac{n}{2}$ wählen und $\mathit{fib}(0) = 1$ verwenden. Ist n jedoch ungerade, so nutzen wir $\mathit{fib}(n) \geq \mathit{fib}(n-1)$ und erhalten daraus, nach der eben durchgeführten Abschätzung, daß

$$\mathit{fib}(n) \geq \mathit{fib}(n-1) \geq 2^{(n-1) \; div \; 2} = 2^{n \; div \; 2}.$$

Kombinieren wir nun die beiden Abschätzungen (1) und (2), so führt dies zu

$$2^{height(t) \; div \; 2} \leq |nodes(t)|$$

bzw., durch Logarithmisierung der beiden Seiten und der Definition von m, zu

$$height(t) \; div \; 2 \leq log_2 |nodes(t)| = log_2 m.$$

Diese letzte Abschätzung impliziert aber $height(t) \leq 2 * log_2 m$ für $height(t)$ gerade und $height(t) \leq 2 * log_2 m + 1$ sonst, also offensichtlich die Behauptung. ■

AVL-Bäume sind spezielle binäre Suchbäume und damit kann man von der Modula-2-Implementierung des Abschnitts 8.3 die beiden Typen `nodes` und `searchtree` sowie die Rechenvorschrift `isin` (fast) direkt übernehmen. Die Prozeduren `insert` und `delete` des Vergrößerns und Entfernens der Abschnitte 8.3.4 bzw. 8.3.5 kann man hingegen nicht weiterverwenden, da sie die Eigenschaft, ein AVL-Baum zu sein, nicht erhalten. Im Vergleich zu den bisherigen Versionen sind die entsprechenden Prozeduren für AVL-Bäume wesentlich komplizierter, obwohl sie asymptotisch effizienter sind. Sehr hilfreich bei der Entwicklung von Implementierungen der baummanipulierenden Operationen auf AVL-Bäumen ist der folgende Begriff.

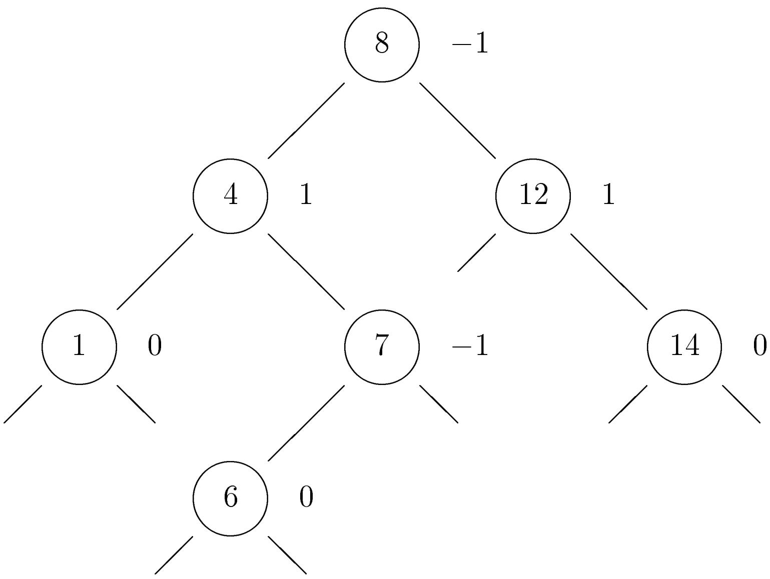
8.4.3 Definition (Balancefaktor)

Ist $\langle l, s, r\rangle$ ein Teilbaum eines knotenmarkierten Binärbaums, so heißt $height(r) - height(l)$ der **Balancefaktor** von $\langle l, s, r\rangle$ (kurz auch: von s). ■

Bei einer zeichnerischen Darstellung von knotenmarkierten Binärbäumen – insbesondere also von ALV-Bäumen – mit Balancefaktoren schreibt man üblicherweise den Balancefaktor an die Wurzel des jeweiligen Teilbaums. Um den Balancefaktor von der Markierung eines Knotens unterscheiden zu können, wird letztere in der Regel eingekreist. Auch wir verwenden diese Konvention in diesem Abschnitt.

8.4.4 Beispiel (AVL-Baum mit Balancefaktoren)

Nachfolgend ist ein binärer Suchbaum mit eingekreisten Knotenmarkierungen und den Balancefaktoren angegeben.



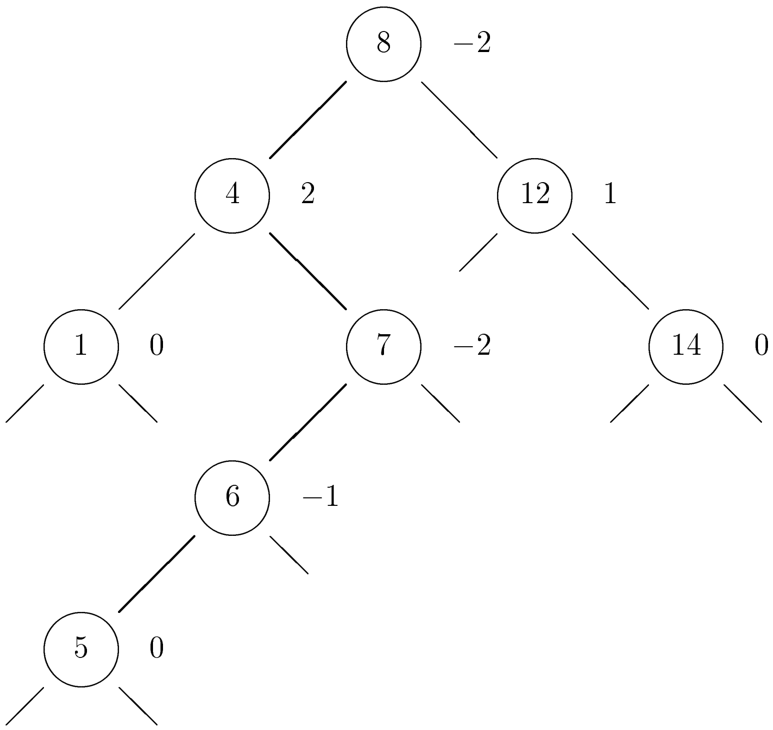
Da alle Balancefaktoren der vorkommenden Teilbäume / Knoten, kurz: die des Baums, aus der Menge $\{-1, 0, 1\}$ sind, handelt es sich um einen AVL-Baum. ■

Wir behandeln im Rest dieses Abschnitts die Modifikation sowohl der Vergrößerungs- als auch der Entfernungs-Operation auf so eine Weise, daß die AVL-Eigenschaft erhalten bleibt. Dabei geben wir aber nur die algorithmischen Ideen an und verzichten auf die den modifizierten Operationen entsprechenden ausformulierten Modula-2-Prozeduren. Solche Prozeduren findet der Leser beispielsweise in dem schon öfter erwähnten Buch von H.-J. Appelrath und J. Ludewig. Man beachte weiterhin, daß per Annahme im folgenden alle Schlüssel eines Baums verschieden sind und wir deshalb die Knoten und die Schlüssel eines AVL-Baums identifizieren dürfen.

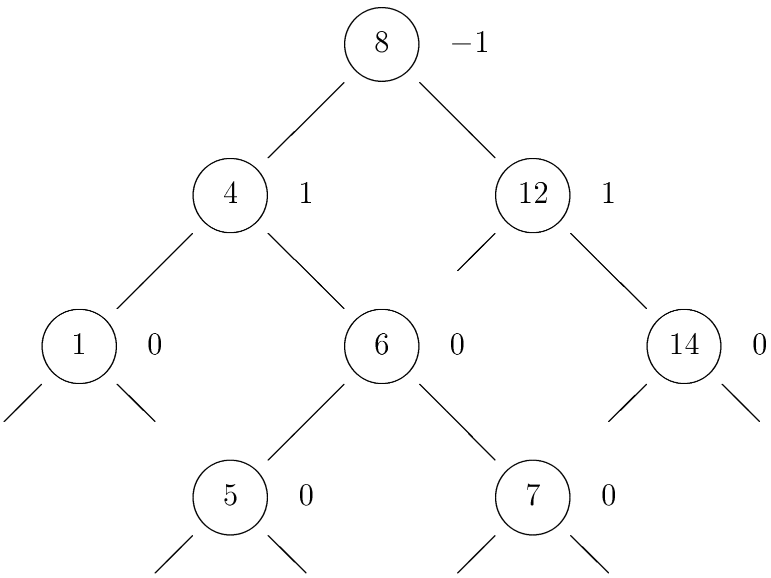
Grundlegend für das Vergrößern eines AVL-Baums um einen Knoten bzw. das Entfernen eines Knotens aus einem AVL-Baum sind zwei lokale Baummanipulationen, die in dem nachfolgenden Beispiel motiviert werden.

8.4.5 Beispiel (zur Motivation der Rotationen)

Wir betrachten den AVL-Baum von Beispiel 8.4.4 und fügen in diesen Baum einen neuen Knoten mit der Markierung 5 ein. Dadurch entsteht der nachfolgend dargestellte Baum, in dem auch die jeweiligen Balancefaktoren angegeben sind und der „Einfügeweg“ $8, 4, 7, 6, 5$ von der Wurzel zum Knoten 5 fett eingezeichnet ist:



Dieser binäre Suchbaum besitzt Balancefaktoren, die nicht aus der Menge $\{-1, 0, 1\}$ sind. Er ist also kein AVL-Baum. Indem man den Teilbaum mit der Wurzel 7 um diesen Knoten nach rechts „rotiert“, kann er jedoch wiederum zu einem AVL-Baum mit derselben Knotenmenge als der Originalbaum gemacht werden. Bildlich sieht dieser AVL-Baum wie folgt aus:



Wie man durch einen Vergleich der beiden Bilder sofort sieht, dreht bei der Transformation des alten Suchbaums in den neuen AVL-Baum die Wurzel 7 des im Uhrzeigersinn „rotierenden“ Teilbaums nach rechts unten ab und zieht dabei den direkten linken Teilbaum mit der Wurzel 6 nach rechts oben hoch. ■

In der Klammerdarstellung von Binärbäumen handelt es sich bei der Rotation des letzten Beispiels um den Übergang von $\langle\langle \ldots, 6, \mathit{etree}\rangle, 7, \mathit{etree}\rangle$ nach $\langle \ldots, 6, \langle \mathit{etree}, 7, \mathit{etree}\rangle\rangle$. Allgemein spricht man bei einem Baumübergang

$$\langle\langle \alpha, a, \beta\rangle, x, \gamma\rangle \implies \langle \alpha, a, \langle \beta, x, \gamma\rangle\rangle \qquad (\mathit{RRot})$$

von einer **Rechtsrotation** von a nach x und bei dem inversen Baumübergang

$$\langle \alpha, a, \langle \beta, x, \gamma\rangle\rangle \implies \langle\langle \alpha, a, \beta\rangle, x, \gamma\rangle \qquad (\mathit{LRot})$$

von einer **Linksrotation** von x nach a. In der Klammerdarstellung entsprechen Rotationen also Umklammerungen. Nach diesen Vorbereitungen befassen wir uns nun mit der Modifikationen der Vergrößerungs-Operation auf so eine Weise, daß die AVL-Eigenschaft erhalten bleibt. Im Prinzip generalisieren wir dabei nur das eben gebrachte Beispiel durch eine vollständige Fallunterscheidung.

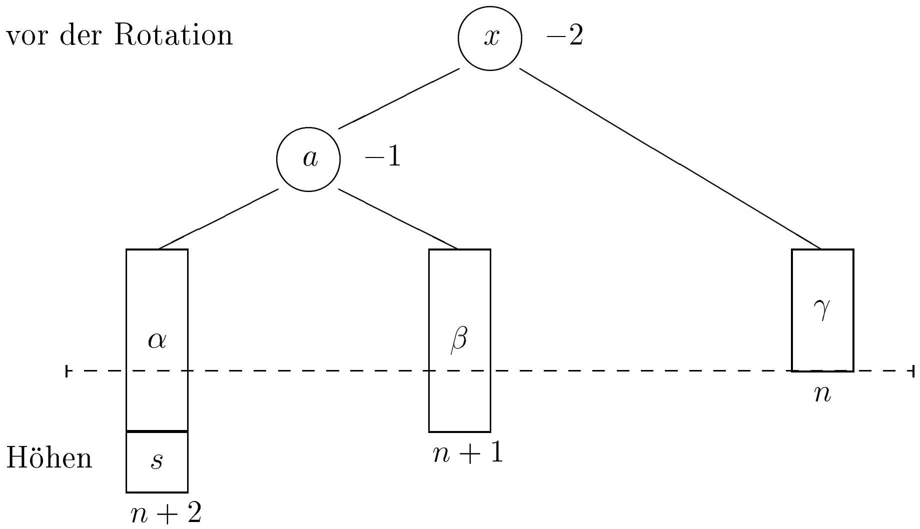
8.4.6 Einfügen in AVL-Bäume

Gegeben seien ein AVL-Baum $t \in \mathsf{AVL}(\mathcal{S})$ und ein Schlüssel $s \in \mathcal{S}$ mit $s \notin \mathit{nodes}(t)$. Zum Einfügen von s in t geht man algorithmisch wie folgt vor:

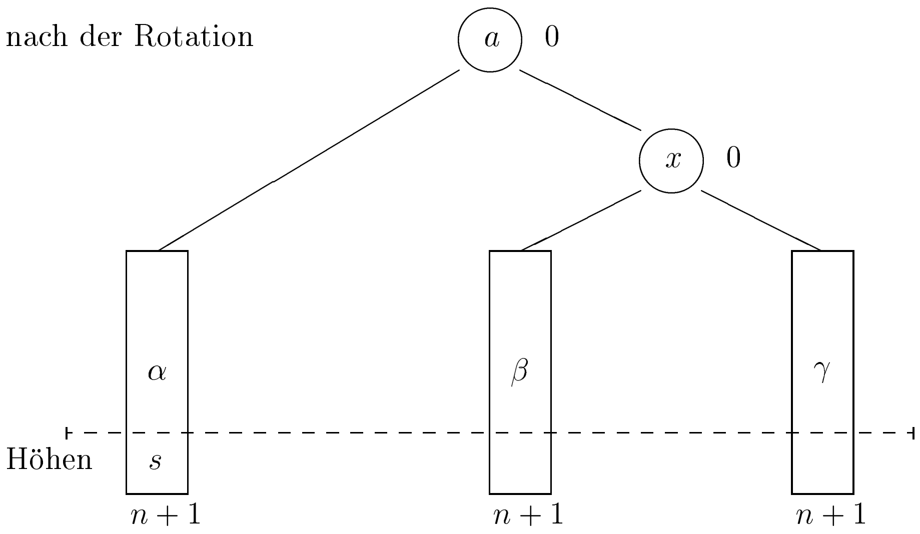
a) Zuerst wird der Schlüssel s wie in Abschnitt 8.3.4 beschrieben, also wie bei den binären Suchbäumen, in t als neues Blatt eingefügt.

b) Dann geht man im neuen Baum den Einfügeweg in Richtung Wurzel des Gesamtbaums zurück und berechnet für seine Knoten die neuen Balancefaktoren. Sind alle -1, 0 oder 1, so wurde, wie einfache Überlegungen zeigen, die AVL-Eigenschaft durch das Einfügen von s nicht zerstört und man ist fertig, Andernfalls nimmt man beim Gehen in Richtung der Gesamtwurzel den ersten Knoten x mit einem Balancefaktor -2 oder 2 den man erreicht, also den ersten Knoten, der zur Verletzung der AVL-Eigenschaft beiträgt. Nun können vier Fälle vorkommen:

(1) Es tritt im neuen Baum ein Überhang im **direkten linken Teilbaum des direkten linken Teilbaums** von x auf, der sich wie folgt darstellt, wobei die angegebenen Höhen die Längen von längsten Kantenzügen zu x bezeichnen.

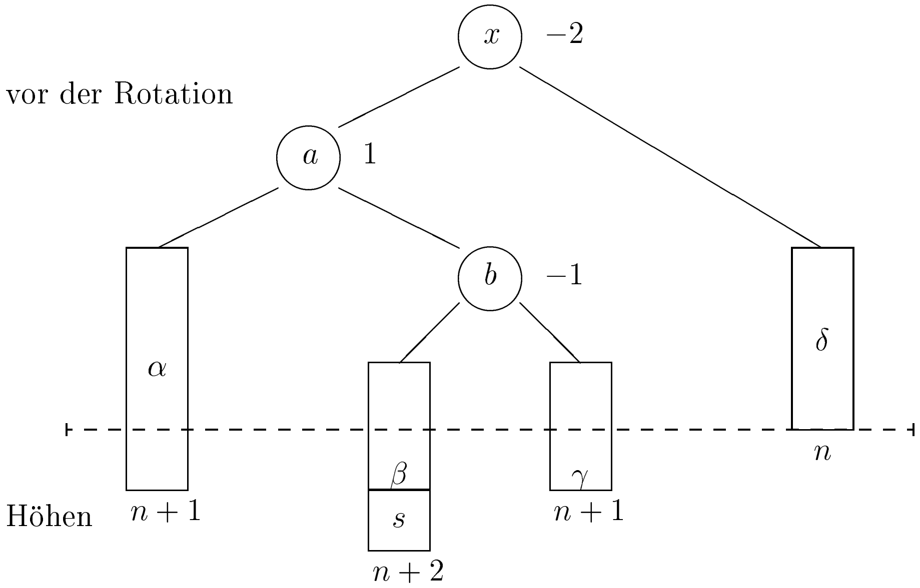


Der Balancefaktor -1 des Knotens a in diesem Bild ist dabei eine Folge der Annahme und des daraus sich ergebenden Balancefaktors -2 von x. Nun transformiert man den obigen Teilbaum des neuen Baums durch eine **Rechtsrotation** von a nach x, ersetzt ihn also durch den nachfolgenden Baum:

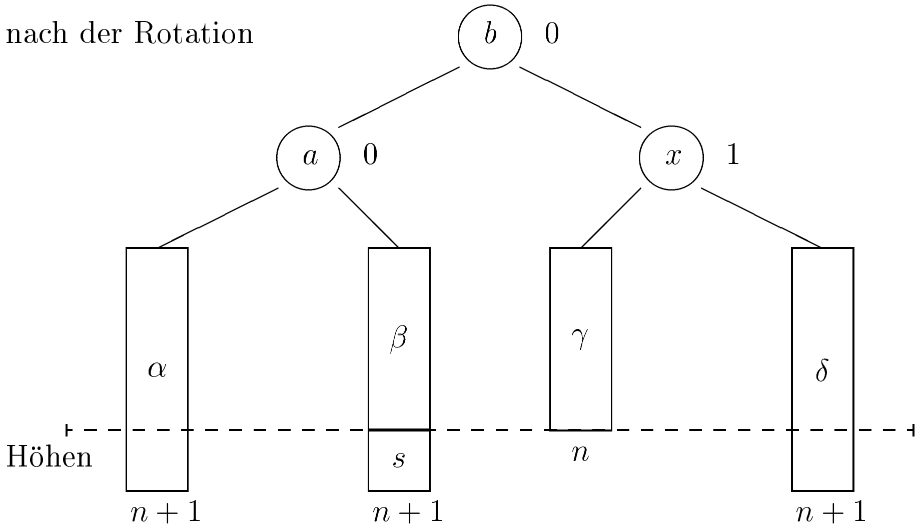


Es sind β, γ und, wegen der Annahme an den Knoten x, auch α (mit dem neuen Knoten s) AVL-Bäume und somit erfüllt auch der dargestellte Baum die AVL-Eigenschaft. Weiterhin bekommen durch die Rotation die Balancefaktoren der restlichen Knoten offensichtlich wieder die Werte, die sie vor dem Einfügen von s hatten. Also erfüllt der neue Gesamtbaum die AVL-Eigenschaft.

(2) Es tritt im neuen Baum ein Überhang im **direkten rechten Teilbaum des direkten linken Teilbaums** von x auf. Hier gibt es zwei Situationen, nämlich



(mit dem Knoten s im Teilbaum β) und die dazu symmetrische, bei der, grob gesagt, „β kürzer und γ länger“ ist. Zur Balancierung des gezeichneten Teilbaums transformieren wir ihn zuerst durch eine **Linksrotation** von b nach a und dann das Resultat durch eine **Rechtsrotation** von b nach x. Dies bringt:



Offensichtlich ist dies ein AVL-Baum. Also erfüllt auch der neue Gesamtbaum diese Eigenschaft. Den symmetrischen Fall behandelt man in analoger Weise.

(3) Es tritt im neuen Baum ein Überhang im **direkten rechten Teilbaum des direkten rechten Teilbaums** von x auf. Hier balanciert man, dual zu Fall (1), den neuen Baum durch eine **Linksrotation** zu einem AVL-Baum aus.

(4) Es tritt im neuen Baum ein Überhang im **direkten linken Teilbaum des direkten rechten Teilbaums** von x auf. Hier balanciert man schließlich den neuen Baum zu einem AVL-Baum aus, indem man, dual zu (2), eine **Rechtsrotation** gefolgt von einer **Linksrotation** anwendet. ■

Man beachte, daß in den ersten beiden Fällen beim Einfügen die Ungleichungen $a < x$ bzw. $a < b < x$ ausgenutzt wurden, um nach den Rotationen die Forderung c) von Definition 8.3.1 wieder zu erfüllen. Auch bei den restlichen Fällen geht diese paarweise Verschiedenheit der Schlüssel natürlich ein. Weiterhin beachte man, daß nach dem Ausbalancieren die originale Höhe wieder erreicht ist; dies ist der Grund dafür, daß dann das Verfahren schon beendet werden kann.

Bevor wir uns mit der algorithmischen Realisierung der zweiten baummanipulierenden Operation beschäftigen, ist eine kurze Bemerkung zum Sortieren durch Suchbäume angebracht. Wie schon erwähnt wurde, ist bei einem binären Suchbaum die von uns angegebene Einfügeoperation von der Laufzeitkomplexität $\mathcal{O}(n)$, falls n die Knotenzahl des Baums ist, und dies bedingt insbesondere, daß der in Abschnitt 8.3.7 vorgestellte Sortieralgorithmus quadratischen Aufwand erfordert. Bei AVL-Bäumen bekommt man wegen Satz 8.4.2 für das **Einfügen einen $\mathcal{O}(log_2 n)$-Algorithmus**, wenn man nach der in Abschnitt 8.4.6 beschriebenen Methode vorgeht. Dies impliziert, daß mit dieser neuen Einfügeoperation das Verfahren von Abschnitt 8.3.7 in $\mathcal{O}(n * log_2 n)$ Zeit sortiert: Man hat nämlich n Einfügungen auszuführen, wobei jede Einfügung $\mathcal{O}(log_2 n)$ Zeit benötigt, und dann einen inorder-Durchlauf zu bewerkstelligen, welcher $\mathcal{O}(n)$ Zeit kostet.

Nach der Übertragung der Operation zum Einfügen von Schlüsseln von binären Suchbäumen auf AVL-Bäume kommen wir nun zur Übertragung der Operation zum Entfernen von Schlüsseln. Dies wird noch komplizierter als der eben betrachtete Fall.

8.4.7 Entfernen aus AVL-Bäumen

Wir betrachten $t \in \mathsf{AVL}(\mathcal{S})$ und $s \in \mathcal{S}$ und wollen einen mit s markierten Knoten aus t entfernen, soferne vorhanden. Dazu geht man wie folgt vor:

a) Zuerst wird (unter Modifikation der Balancefaktoren auf dem Weg zum entsprechenden Knoten) wie bei den normalen Suchbäumen gelöscht, d.h. s durch den Vorgänger im inorder-Verfahren (dem Maximum im direkten linken Teilbaum) ersetzt.

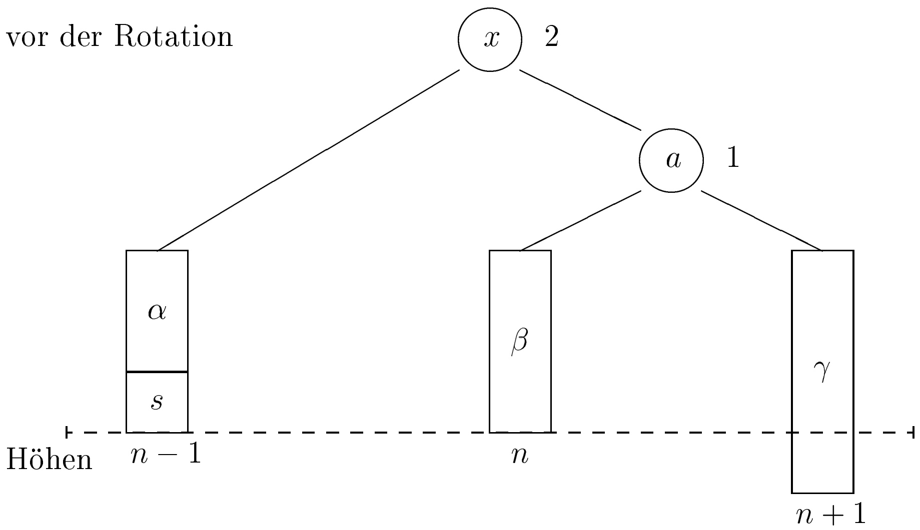
b) Dann geht man im neuen Baum von der Löschstelle, also der Stelle, an der sich früher s befand, in Richtung Wurzel des Gesamtbaums zurück und balanciert den Baum durch Rotationen wiederum zu einem AVL-Baum aus. Wie beim Einfügen, so gibt es auch hier mehrere Fälle. Zuerst haben wir zu unterscheiden in:

(i) Der Balancefaktor eines Teilbaums mit Wurzel x wird 2, da s in seinem **direkten linken Teilbaum entfernt** wurde.

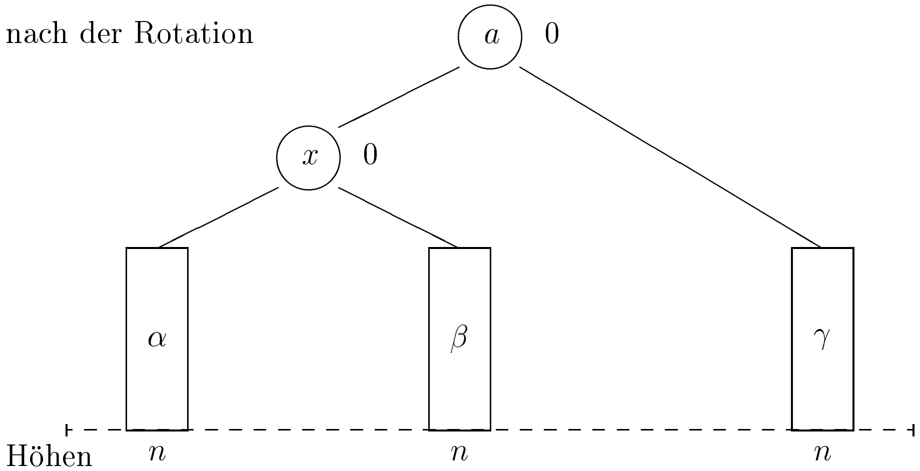
(ii) Der Balancefaktor eines Teilbaums mit Wurzel x wird -2, da s in seinem **direkten rechten Teilbaum entfernt** wurde.

Diese beiden Fälle sind vollkommen symmetrisch, deshalb behandeln wir im weiteren das Balancieren nur für den Fall (i) mit dem Balancefaktor 2 für x nach dem Entfernen von s. Hier haben wir drei Unterfälle zu betrachten:

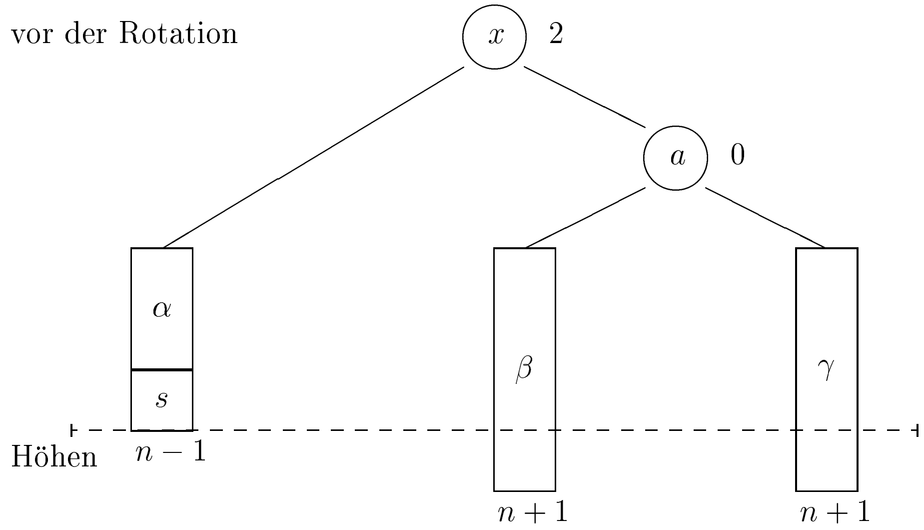
(1) Der direkte rechte Teilbaum des betrachteten Teilbaums hat 1 als Balancefaktor. Bezeichnen wir die Wurzel dieses Teilbaums mit a und seine direkten Teilbäume mit β und γ, so heißt dies $height(\beta) + 1 = height(\gamma)$. Bildlich stellt sich diese Situation wie folgt dar:



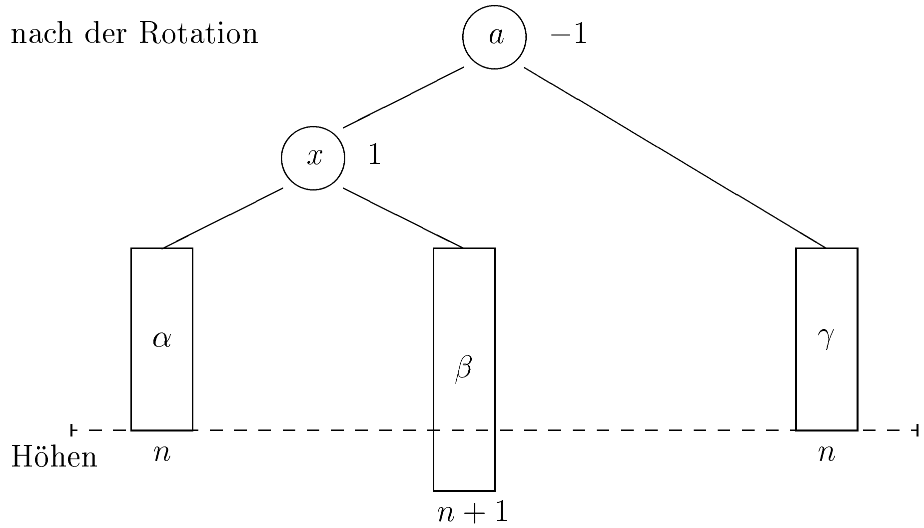
Dabei ist s nicht mehr als Teil von α angenommen, was durch $n-1$ als Höhenangabe angedeutet ist. Eine einfache Möglichkeit, diesen binären Suchbaum zu einen AVL-Baum auszubalancieren, ist eine **Linksrotation** von a nach x. Das Resultat dieser Transformation sieht wie folgt aus:



(2) Der direkte rechte Teilbaum des betrachteten Teilbaums besitzt den Balancefaktor 0. Bezeichnen wir die Wurzel dieses Teilbaums bzw. seine direkten Teilbäume wiederum mit a, β und γ, so haben β und γ die gleiche Höhe und wir bekommen die nachfolgende bildliche Darstellung (mit s wiederum nicht als Teil von α):



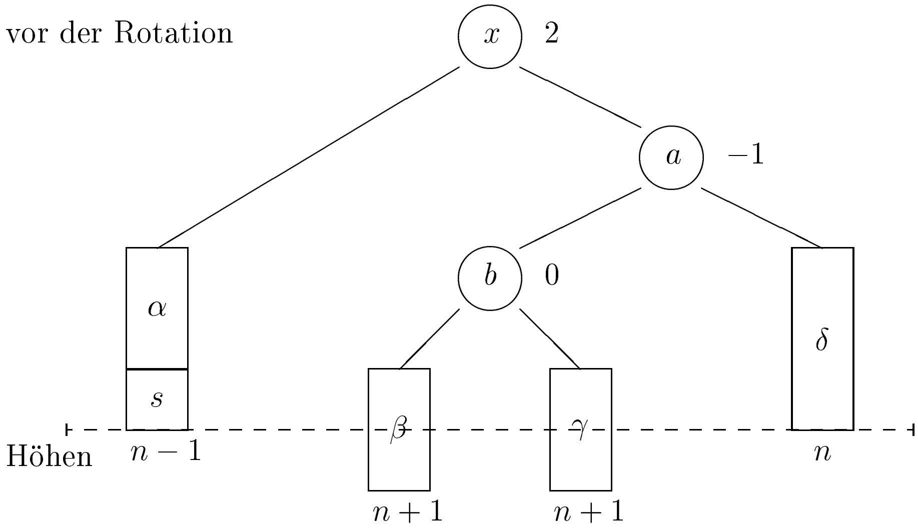
Auch hier balanciert man den binären Suchbaum zu einem AVL-Baum mit Hilfe einer **Linksrotation** von a nach x aus und erhält das folgende Resultat:



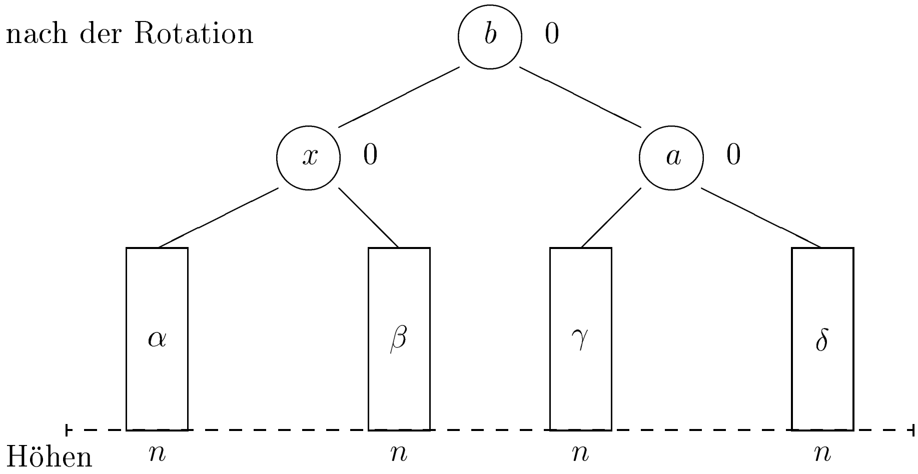
Man beachte, daß die alten und neuen „Bilder“ der bisher behandelten Fälle (1) und (2) bis auf die Höhe des Teilbaums β (die im zweiten Fall um 1 größer ist als im ersten Fall) identisch sind.

(3) Der direkte rechte Teilbaum des betrachteten Teilbaums besitzt den Balancefaktor -1. Bezeichnen wir die Wurzel dieses Teilbaums wiederum mit a, so muß man jetzt noch nach dem Balancefaktor des „linken“ Nachfolgers von a unterscheiden. Es sei b dieser Knoten.

Ist der Balancefaktor von b gleich 0, so sieht der betrachtete Baum wie folgt aus (mit s wiederum nicht als Teil von α):



Hier balanciert man durch eine **Rechtsrotation** von b nach a gefolgt von einer **Linksrotation** von b nach x zum nachstehenden AVL-Baum:



Ist der Balancefaktor von b gleich 1, so ist die Höhe des Teilbaums β im Vergleich zum letzten Fall um 1 kleiner und die eben verwendeten Rotationen balancieren den betrachteten Teilbaum ebenfalls aus.

Der letzte Fall ist schließlich, daß der Balancefaktor von b gleich -1 ist. Nun ist die Höhe des Teilbaums γ im Vergleich zum ersten Fall um 1 kleiner und eine Rechts-Links-Rotation führt auch hier zum Ziel.

c) Durch die Transformation b) kann sich – im Gegensatz zum Einfügen – die gesamte Baumhöhe ändern, wie man leicht durch einen Vergleich der graphischen Darstellungen feststellt. Dann ist zwar der betrachtete Teilbaum ausbalanciert, durch eine Höhenveränderung kann die AVL-Eigenschaft jedoch für den Gesamtbaum verlorengegangen sein. Bei einer Höhenveränderung durch Schritt b) ist deshalb der (neue)

Balancefaktor für den nächsten Knoten in Richtung Wurzel zu überprüfen und gegebenenfalls das Verfahren b) an dieser Stelle erneut durchzuführen.

Wenn man wie eben beschrieben vorgeht, so bekommt man ein $\mathcal{O}(log_2 n)$ Löschverfahren, wobei n die Anzahl der Knoten des vorliegenden AVL-Baums ist. Man beachte, daß auch beim Entfernen die paarweise Verschiedenheit der Schlüssel zur Aufrechterhaltung der Suchbaum-Eigenschaft verwendet wird. ■

Wie schon erwähnt, ist der AVL-Ansatz nicht der einzige Ansatz zum Balancieren. Hat man es etwa mit sehr großen Datenmengen zu tun, die nicht mehr im sogenannten Arbeitsspeicher eines Rechners untergebracht werden können, sondern beispielsweise auf einer Platte liegen, so scheidet eine Balancierung nach dem AVL-Kriterium wegen des viel zu großen Aufwands aus. Man schwächt deshalb die Anforderungen in verschiedenen Weisen ab, was zu verschiedensten Baumformen, wie B-Bäumen, HB-Bäumen oder gewichtsbalancierten Bäumen, für effizientes Suchen führt.

8.5 Graphen und graphentheoretische Aufgabenstellungen

Im Laufe des Skriptums lernten wir schon einige Typen von Bäumen kennen, so in Kapitel 5 und in den letzten beiden Abschnitten 8.3 und 8.4. Bäume sind spezielle Graphen und jene nichts anderes als Relationen auf Knoten. Graphen werden oft zur anschaulichen Visualisierung abstrakter Konzepte verwendet, aus der sich dann Einsichten und Ideen zu Problemlösungen gewinnen lassen. An graphentheoretischen Problemen kann man auch sehr schön die verschiedenen Methoden zum Entwurf von effizienten Algorithmen demonstrieren. In diesem Abschnitt behandeln wir die grundlegendsten Begriffe der Graphentheorie und demonstrieren, wie Probleme der Praxis auf abstrakte graphentheoretische Probleme reduziert werden können. Graphentheoretischen Algorithmen werden im nächsten Abschnitt 8.6 behandelt.

Wir beginnen mit einer Reihe von Definitionen, welche die grundlegenden Begriffe der Graphentheorie einführen.

8.5.1 Definition (Graphen)

Ein **gerichteter Graph**, kurz Graph, ist ein Paar $g = (V, R)$, wobei V eine nichtleere Menge ist, deren Elemente **Knoten** genannt werden, und $R \subseteq V \times V$ eine Relation auf V ist, deren Elemente **Pfeile** genannt werden. Ist $\langle x, y\rangle \in R$ ein Pfeil, so heißt x **Anfangsknoten** und y **Endknoten**. Man sagt auch, y ist **Nachfolger** von x und x **Vorgänger** von y. ■

Ein Graph $g = (V, R)$ mit endlicher Knotenmenge heißt **endlich**. Ist man an graphentheoretischen Algorithmen interessiert, so nimmt man üblicherweise Endlichkeit an. Endliche Graphen kann man auch bildlich darstellen. Die Knoten werden Punkte in der euklidischen Zeichenebene und die Pfeile werden Kurven (oft Geraden), deren Richtung von x

nach y durch eine Pfeilspitze bei y angegeben wird. Diese Darstellung haben wir auch schon bei den Bäumen verwendet, wobei auf die Pfeilspitze verzichtet werden konnte, da wir immer „Abwärtsrichtung“ angenommen haben.

8.5.2 Definition (Grundbegriffe)

Es sei $g = (V, R)$ ein Graph. Zu $x \in V$ definiert man die **Vorgänger-** und **Nachfolgermengen** durch

$$pred(g, x) = \{\ y \in V \ :\ \langle y, x\rangle \in R\ \} \qquad succ(g, x) = \{\ y \in V \ :\ \langle x, y\rangle \in R\ \}$$

und nennt $|pred(g, x)|$ den **Eingangsgrad** von x und $|succ(g, x)|$ den **Ausgangsgrad** von x. Knoten mit Eingangsgrad 0 heißen **Quellen**, solche mit Ausgangsgrad 0 **Senken** und solche ohne Vorgänger und Nachfolger **isoliert**. ■

In Zeichnungen erkennt man Quellen daran, daß in ihnen keine Pfeile enden, und Senken daran, daß in ihnen keine Pfeile beginnen. Die nachfolgenden Begriffe betreffen Erreichbarkeitsprobleme in Graphen und sind sehr wichtig für Algorithmen und praktische Aufgaben, die graphentheoretisch gelöst werden.

8.5.3 Definition (Wege und Kreise)

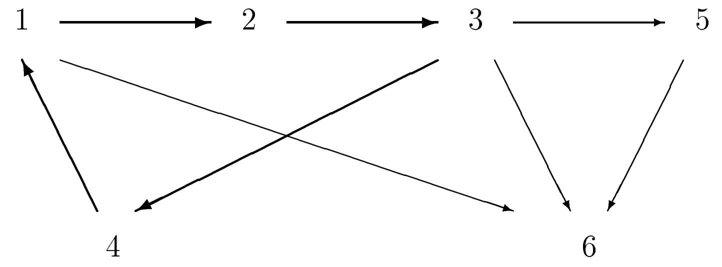
Gegeben sei ein Graph $g = (V, R)$.

a) Eine Folge $\langle x_0, \ldots, x_n\rangle$ von Knoten heißt ein **Weg** von x_0 nach x_n mit **Länge** (auch: **Pfeilanzahl**) n, falls $\langle x_i, x_{i+1}\rangle \in R$ für alle i mit $0 \leq i \leq n-1$ zutrifft.

b) Ein Weg mit $n = 0$ heißt **leer** oder **einknotig**.

c) Ein nichtleerer Weg $\langle x_0, \ldots, x_n\rangle$ mit $x_0 = x_n$ heißt **geschlossen** oder ein **Kreis**. Existiert in g kein Kreis, so nennt man g **kreisfrei**. ■

Man beachte, daß in zeichnerischen Darstellungen von Graphen identische Kreise als Folgen von Knoten durchaus verschieden sein können, da man der Zeichnung ja in der Regel nicht den Anfangsknoten entnehmen kann. Als ein einfaches Beispiel betrachten wir den nachfolgend zeichnerisch angegebenen Graphen:



In diesem Graphen ist ein zeichnerischer Kreis durch fett markierte Pfeile hervorgehoben. Die folgenden drei Sequenzen sind Kreise im Sinne der obigen Definition 8.5.3.c und zeichnerisch mit diesem Kreis identisch:

$$\begin{array}{rl} \langle 1,2,3,4,1\rangle & \text{Start 1, einmal durchlaufen} \\ \langle 2,3,4,1,2\rangle & \text{Start 2, einmal durchlaufen} \\ \langle 3,4,1,2,3,4,1,2,3\rangle & \text{Start 3, zweimal durchlaufen} \end{array}$$

In Definition 3.3.15 hatten wir die reflexiv-transitive Hülle R^* und die transitive Hülle R^+ einer Relation R eingeführt durch die Gleichungen

$$R^* = \bigcup_{n \geq 0} R^n \qquad\qquad R^+ = \bigcup_{n \geq 1} R^n.$$

Diese zwei relationalen Konstruktionen stehen in einer engen Beziehung zu Wegen und Kreisen in Graphen, wie in dem nächsten Satz aufgezeigt wird. In ihm bezeichnet I die identische Relation.

8.5.4 Satz (Wege, Kreise, Hüllen)

Für einen Graphen $g = (V, R)$ gelten die folgenden Aussagen:

a) Es gibt einen Weg von x nach y der Länge n genau dann, wenn $\langle x, y\rangle \in R^n$.

b) Es gibt einen Weg von x nach y genau dann, wenn $\langle x, y\rangle \in R^*$.

c) Der Graph g ist kreisfrei genau dann, wenn $R^+ \subseteq \overline{I}$.

Beweis: Eigenschaft a) zeigt man durch vollständige Induktion nach n.

Induktionsbeginn $n = 0$: Es gibt einen Weg der Länge 0 von x nach y genau dann, wenn $x = y$ gilt. Letzteres ist aber äquivalent zu $\langle x, y\rangle \in I = R^0$.

Induktionsschluß $n \mapsto n + 1$: Es gibt einen Weg $\langle x_0, \ldots, x_{n+1}\rangle$ von $x = x_0$ nach $y = x_{n+1}$ genau dann, wenn es einen Weg der Länge n gibt, dieser sei $\langle y_0, \ldots, y_n\rangle$ mit $x = y_0$, und einen Pfeil $\langle y_n, y\rangle \in R$. Nach Induktionsvoraussetzung ist dies äquivalent zu

$$\exists\, y_n \in V \;:\; \langle x, y_n\rangle \in R^n \wedge \langle y_n, y\rangle \in R$$

bzw. zu $\langle x, y\rangle \in R^n R = R^{n+1}$ nach der Definition des Produkts von Relationen.

Die zweite Eigenschaft b) folgt direkt aus a): Es gibt einen Weg von x nach y offensichtlich genau dann, wenn $\langle x, y\rangle \in \bigcup_{n\geq 0} R^n = R^*$.

Zum Beweis von c): Mit Hilfe von a) bekommen wir, daß es einen nichtleeren Weg von x nach y genau dann gibt, wenn $\langle x, y\rangle \in \bigcup_{n\geq 1} R^n = R^+$. Daraus folgt nun

$$\begin{array}{rcl} \text{Es gibt keinen Kreis} & \Longleftrightarrow & \neg(\exists\, x, y \in V \;:\; \langle x, y\rangle \in R^+ \wedge x = y) \\ & \Longleftrightarrow & \forall\, x, y \in V \;:\; \langle x, y\rangle \notin R^+ \vee x \neq y \\ & \Longleftrightarrow & \forall\, x, y \in V \;:\; \langle x, y\rangle \in R^+ \to \langle x, y\rangle \in \overline{I} \\ & \Longleftrightarrow & R^+ \subseteq \overline{I} \end{array}$$

unter Verwendung von grundlegenden Regeln der logischen Symbole und der Definition der Relationeninklusion. ■

Nach dieser knappen Einführung in die Graphentheorie demonstrieren wir nun anhand einiger Beispiele, wie in der Praxis auftretende Probleme auf abstrakte Probleme der Graphentheorie reduziert werden können. Wir verbinden dies mit einer Einführung von weiteren graphentheoretischen Begriffen.

8.5.5 Reihenfolgenprobleme und Projektplanung

Die im folgenden beschriebene Situation ist typisch für das Gebiet der Unternehmungsplanung (engl. Operations Research). Gegeben sei ein Gesamtprojekt, welches in einzelne Tätigkeiten zerlegt sei. Weiterhin sei eine Menge von Beziehungen zwischen den Tätigkeiten in der Form von Regeln

$$\boxed{\text{wenn } T_{i_1}, \ldots, T_{i_k} \text{ beendet sind, dann kann } T \text{ starten}}$$

angegeben. Besteht das Projekt etwa in der Ersetzung einer Maschine durch eine neuere, so kann man das Fundament für die neue Maschine z.B. erst bauen, wenn das der alten Maschine abgerissen ist. In dieser Situation stellen sich nun viele Fragen, etwa die folgenden drei:

a) Ist die Projektplanung **konsistent**, d.h. gibt es keine zyklischen Abhängigkeiten?

b) Angenommen, die Projektplanung sei konsistent. Man bestimme eine mögliche Abarbeitung der Tätigkeiten, wenn man nur ein Team / einen Prozessor zur Verfügung hat („Reihenfolge einer **sequentiellen Abarbeitung**“).

c) Wie viele Teams / Prozessoren braucht man maximal, wenn man bei konsistenter Planung soviele Tätigkeiten wie möglich **parallel ausführen** will?

Alle drei Probleme führen zu graphentheoretischen Fragestellungen, wenn man die Tätigkeiten als Knoten definiert und einen Pfeil $\langle T_i, T_j \rangle$ auf dieser Knotenmenge einführt genau dann, wenn T_j begonnen werden kann, falls T_i beendet ist. Es sei $g = (V, R)$ dieser endliche Graph. Dann haben wir:

a) Die Projektplanung ist genau dann konsistent, falls es in g keinen Kreis gibt.

b) Eine Funktion $ts : V \mapsto \{1, \ldots, |V|\} \subseteq \mathbb{N}$ heißt eine **topologische Sortierung** des kreisfreien Graphen $g = (V, R)$, falls

$$\forall\, x, y \in V \;:\; \langle x, y \rangle \in R \rightarrow ts(x) < ts(y).$$

Man kann also die Knoten so durchnumerieren, daß nur Pfeile von Knoten mit kleineren zu Knoten mit größeren Nummern führen. Eine sequentielle Abarbeitungsreihenfolge der Tätigkeiten ergibt sich somit direkt aus einer topologischen Sortierung von g.

c) Eine Folge $\langle V_1, \ldots, V_k \rangle$ von Mengen heißt eine **k-stufige Sortierung** des kreisfreien Graphen $g = (V, R)$, falls

   (1) die Mengen $V_1, \ldots, V_k$ eine Partition von V bilden (d.h. sie sind alle nicht leer, paarweise disjunkt und ihre Vereinigung liefert V),

   (2) für alle $\langle x, y \rangle \in R$ aus $x \in V_i$ und $y \in V_j$ folgt $i < j$.

   Eine Lösung des obigen Problems c) ist graphentheoretisch eine k-stufige Sortierung $\langle V_1, \ldots, V_k \rangle$, so daß k minimal ist (d.h. es gibt für $j < k$ keine j-stufige Sortierung) und für jede andere k-stufige Sortierung $\langle V'_1, \ldots, V'_k \rangle$ gilt

$$max_{1 \leq i \leq k} |V_i| \leq max_{1 \leq i \leq k} |V'_i| \, .$$

   Man braucht dann bei möglichst paralleler Ausführung der Projekttätigkeiten genau k „Arbeitstakte" und kommt mit $max_{1 \leq i \leq k} |V_i|$ Teams bzw. Prozessoren aus. ■

Zum Testen von Kreisfreiheit und zur Berechnung einer topologischen Sortierung gibt es praktikable Algorithmen. In Abschnitt 8.6 werden wir solche entwickeln. Von der Funktion, die die k-stufige Sortierung eines kreisfreien Graphen spezifiziert, wurde gezeigt, daß sie in der Funktionenklasse $\mathcal{NP}$ liegt. Einen praktikablen Algorithmus zu ihrer Berechnung kennt man jedoch noch nicht. Hätte man einen solchen, so wäre, wie im Fall des Tests auf Tautologien, die Gleichung $\mathcal{P} = \mathcal{NP}$ gezeigt.

Es gibt noch eine Erweiterung es oben genannten Ansatzes, bei dem jeder der Tätigkeiten noch eine Zeitdauer in Form einer Zahl (beispielsweise von Tagen) zugeordnet wird. Man ist dann an einer Abarbeitung interessiert, bei der die Projektdauer minimiert ist. Die entsprechende graphentheoretische Theorie nennt man **Netzplantechnik**; genaueres findet man Lehrbüchern zum Gebiet Operations Research.

8.6 Einige graphentheoretische Algorithmen

In diesem Abschnitt zeigen wir anhand von einigen Beispielen, wie man graphentheoretische Probleme algorithmisch lösen kann. Wir beginnen mit einem der wichtigsten Probleme überhaupt, dem Erreichbarkeitsproblem. Dieses Problem taucht bei vielen anderen Fragestellungen der Graphentheorie als Teilproblem auf.

8.6.1 Ein erster Erreichbarkeits-Algorithmus

Gegeben sei ein **endlicher** Graph $g = (V, R)$ mit N als der Knotenzahl. Wir betrachten zu einem festen Knoten $x \in V$ die **Erreichbarkeitsfunktion**

$$reach(g, x) = \{ \; y \in V \; : \; \langle x, y \rangle \in R^* \; \},$$

die x die Menge der von ihm aus erreichbaren Knoten zuordnet. Im folgenden entwickeln wir einen Algorithmus für die Berechnung von $reach(g, x)$. Dabei legen wir das Augenmerk auf funktionale Programmierung, verwenden also die dort vorgestellten Techniken.

Beim Ansatz zur Algorithmengewinnung orientieren wir uns an der **Länge der Wege** von x nach y. Offensichtlich haben wir, siehe Satz 8.5.4.b, die Äquivalenz von $\langle x, y\rangle \in R^*$ und $\langle x, y\rangle \in \bigcup_{i=0}^{N-1} R^i$, denn existiert ein Weg von x nach y, so gibt es auch einen Weg von x nach y, in dem jeder Knoten nur einmal vorkommt[34], und ein solcher Weg hat maximal die Länge $N-1$. Also bekommen wir die Einbettung

$$reach(g, x) = lreach(g, x, N-1), \tag{R_1}$$

mit der neuen dreistelligen Funktion *lreach*, definiert durch die Gleichung

$$lreach(g, x, k) = \{\ y \in V\ :\ \langle x, y\rangle \in \bigcup_{i=0}^{k} R^i\ \}.$$

Diese Funktion beschreibt ein, im Vergleich zum Originalproblem allgemeineres, Problem. Sie spezifiziert nämlich zu einem Graphen g, einem Knoten x und einer natürlichen Zahl k die Menge der Knoten y, für die es in g einen Weg von x nach y mit Maximallänge k gibt. Geleitet durch den dritten (im Vergleich zu *reach* neuen) Parameter berechnen wir nun eine terminierende Rekursion für die Funktion *lreach*.

Der Fall $k = 0$ ist einfach. Wir starten mit der Äquivalenz

$$\langle x, y\rangle \in \bigcup_{i=0}^{0} R^i \iff \langle x, y\rangle \in I \iff x = y$$

und bekommen daraus mittels der Definition von *lreach* die Gleichheit

$$lreach(g, x, 0) = \{\ y \in V\ :\ x = y\ \} = \{\, x\,\}. \tag{R_2}$$

Im Fall $k \neq 0$ betrachten wir die endliche Vereinigung $\bigcup_{i=0}^{k} R^i$ und erhalten durch ihre induktive Beschreibung, die Definition von R^0 und die Distributivität des Relationenprodukts über die Vereinigung sofort die Rekursion $\bigcup_{i=0}^{k} R^i = I \cup (\bigcup_{i=0}^{k-1} R^i)R$. Dies bringt nun mit Hilfe der Nachfolger-Funktion *succ*, etwas Mengen- bzw. Relationenalgebra und der Definition von *lreach*, daß

$$\begin{aligned}
lreach(g, x, k) &= \{\ y \in V\ :\ \langle x, y\rangle \in \textstyle\bigcup_{i=0}^{k} R^i\ \}\\
&= \{\ y \in V\ :\ \langle x, y\rangle \in I \cup (\textstyle\bigcup_{i=0}^{k-1} R^i)R\ \}\\
&= \{\, x\,\} \cup \{\ y \in V\ :\ \langle x, y\rangle \in (\textstyle\bigcup_{i=0}^{k-1} R^i)R\ \}\\
&= \{\, x\,\} \cup \{\ y \in V\ :\ \exists\, z \in V : \langle x, z\rangle \in \textstyle\bigcup_{i=0}^{k-1} R^i \wedge \langle z, y\rangle \in R\ \}\\
&= \{\, x\,\} \cup \{\ y \in V\ :\ \exists\, z \in lreach(g, x, k-1) : y \in succ(g, z)\ \}\\
&= \{\, x\,\} \cup \textstyle\bigcup_{z \in lreach(g,x,k-1)} succ(g, z).
\end{aligned}$$

Nun verwenden wir zur Vereinfachung der letzten Mengenkonstruktion eine weiteren Hilfsfunktion *succs* mit der Definition

$$succs(g, s) = \bigcup_{x \in s} succ(g, x)$$

[34]Anfangs- und Endknoten sind hier natürlich ausgenommen, falls $x = y$ zutrifft. Man nennt diese Wege auch „einfach“ oder „doppelpunktfrei“.

zur Berechnung der Nachfolger einer Menge s von Knoten in g. Wir erhalten dann für $k \neq 0$ die rekursive Gleichung

$$lreach(g, x, k) = \{x\} \cup succs(g, lreach(g, x, k-1)). \tag{R_3}$$

Eine rekursive Fassung der Hilfsfunktion *succs*, die sich an der Form des zweiten Arguments s orientiert, ist offensichtlich. Es gilt nämlich

$$succs(g, \emptyset) = \emptyset \tag{R_4}$$

und, für $s \neq \emptyset$ und einen Knoten $z \in s$ beliebig gewählt,

$$succs(g, s) = succ(g, z) \cup succs(g, s \setminus \{z\}). \tag{R_5}$$

Mit den Gleichungen (R_1) bis (R_5) haben wir eine terminierende rekursive Berechnung für die Funktion *reach* erreicht, die sich – nach einer Realisierung eines Typs für Knoten und eines Typs für Knotenmengen, sowie einer Implementierung der Nachfolger-Funktion *succ* – auch so direkt in die Programmiersprache ML übertragen läßt. Genaueres folgt in Abschnitt 8.6.3. ■

Bei der eben präsentierten Herleitung wurden nirgends explizit Wege verwendet. Dies ist anders im nächsten Fall, wo in der Herleitung konkret von Wegen Gebrauch gemacht wird. Zur Vereinfachung der Manipulationen sei deshalb im folgenden zu einem Graphen $g = (V, R)$, einer Knotenmenge $s \in 2^V$ und einem Knoten $y \in V$ mit

$$\mathcal{W}(g, s, y) = \{ \ p \ : \ p \text{ ist Weg in } g \text{ mit } p_1 \in s \text{ und } p_{|p|} = y \ \}$$

die Menge der Wege in g bezeichnet, die in s beginnen und in y enden. Weiterhin stehe für eine Knotenmenge $t \in 2^V$ und einen Weg p die Bedingung $p \cap t = \emptyset$ als Abkürzung dafür, daß kein Knoten von p in t enthalten ist.

8.6.2 Ein zweiter Erreichbarkeits-Algorithmus

Es seien wiederum ein **endlicher** Graph $g = (V, R)$ sowie ein fester Knoten $x \in V$ vorgegeben. Wir betrachten nun die **Erreichbarkeitsfunktion** in der Form

$$reach(g, x) = \{ \ y \in V \ : \ \mathcal{W}(g, \{x\}, y) \neq \emptyset \ \},$$

was nach Satz 8.5.4.b erlaubt ist. Bei der nun – wiederum dem funktionalen Paradigma folgenden – Programmentwicklung sammeln wir, algorithmisch betrachtet, in einer Menge t **die Knoten auf, die wir schon erreicht haben**, und führen während des Ablaufs auch die Menge s der echten Nachfolger von t mit.

Der formale Startpunkt der Deduktion ist die offensichtliche Gleichwertigkeit der obigen Problemspezifikation zur Gleichung

$$reach(g, x) = \emptyset \cup \{ \ y \in V \ : \ \exists\, p \in \mathcal{W}(g, \{x\}, y) : p \cap \emptyset = \emptyset \ \}.$$

Auf diese Weise erhalten wir durch eine Verallgemeinerung der Menge $\{x\}$ zu einem Parameter s in einer Hilfsfunktion *sreach* und von zwei Vorkommen der leeren Menge zu einem Parameter t der gleichen Hilfsfunktion die Einbettung

$$reach(g,x) = sreach(g,\{x\},\emptyset), \tag{R_6}$$

mit der dreistelligen Funktion *sreach*, definiert durch die Gleichung

$$sreach(g,s,t) = t \cup \{\ y \in V\ :\ \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset\ \}.$$

Diese Funktion spezifiziert die Erweiterung von t um die Menge der Knoten, die von s aus erreichbar sind auf Wegen, die nicht durch die Menge t führen. Algorithmisch haben s und t genau die am Anfang angegebene Bedeutung.

Beim Aufruf von *sreach* in (R_6) sind die Argumente $\{x\}$ und $\emptyset$ disjunkt. Wir betrachten deshalb im folgenden nur Aufrufe $sreach(g,s,t)$, für die $s \cap t = \emptyset$ gilt. Bei der Entwicklung einer terminierenden Rekursion für *sreach* ist diese Bedingung an das zweite und dritte Argument dann natürlich auch bei allen rekursiven Folgeaufrufen einzuhalten.

Es sei also $s \cap t = \emptyset$. Geleitet durch die Definition von *sreach* starten wir die Herleitung einer Rekursion mit dem Fall $s = \emptyset$. Hier haben wir $\mathcal{W}(g,s,y) = \emptyset$, also

$$sreach(g,\emptyset,t) = t \cup \{\ y \in V\ :\ \mathit{false}\ \} = t. \tag{R_7}$$

Im verbleibenden Fall $s \neq \emptyset$ wählen wir ein beliebiges Element z aus der Knotenmenge s. Es gilt dann $z \notin t$ wegen der Voraussetzung $s \cap t = \emptyset$ und somit auch die Beziehung $z \in \{\, y \in V : \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset\,\}$, denn der einknotige Weg $\langle z \rangle$ beginnt in s, endet im Knoten z und hat mit der Knotenmenge t einen leeren Schnitt. Also haben wir

$$sreach(g,s,t) = t \cup \{\, z\,\} \cup \{\ y \in V\ :\ \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset\ \}.$$

Nun betrachten wir den Existenzquantor in dieser deskriptiven Mengenbeschreibung und formen ihn so um, daß die Menge s im Laufe der Abarbeitung der durch die Umformung entstehenden Rekursion schließlich irgendwann leer wird. Es gilt

$$\begin{array}{rl} & \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset \\ \Longleftrightarrow & \exists\, q \in \mathcal{W}(g,(s \cup succ(g,z)) \setminus (t \cup \{z\}), y) : q \cap (t \cup \{z\}) = \emptyset. \end{array}$$

Beweis: „$\Longleftarrow$“: Es sei vorausgesetzt q mit

$$q \in \mathcal{W}(g,(s \cup succ(g,z)) \setminus (t \cup \{z\}), y) \qquad\qquad q \cap (t \cup \{z\}) = \emptyset.$$

Falls der Weg q in der Menge s startet, so gilt $q \in \mathcal{W}(g,s,y)$ und auch $q \cap t = \emptyset$ und damit erfüllt q die Forderungen, die auf der linken Seite der Äquivalenz an p gestellt werden. Liegt hingegen der erste Knoten von q in der Nachfolgermenge $succ(g,z)$, so ist es offensichtlich erlaubt, die Vergrößerung des Weges q um den neuen Anfangsknoten z, also $z\,q$, als p der linken Seite zu wählen.

„$\Longrightarrow$“: Hier geht man analog vor. Es sei also p gegeben, so daß

$$p \in \mathcal{W}(g,s,y) \qquad\qquad p \cap t = \emptyset$$

gelten. Startet $p \in \mathcal{W}(g,s,y)$ in z, so dürfen wir p als einfach voraussetzen und die Verkürzung $q := rest(p)$ erfüllt $q \in \mathcal{W}(g,(s \cup succ(g,z)) \setminus (t \cup \{z\}),y)$ und $q \cap (t \cup \{z\}) = \emptyset$. Andernfalls, also wenn $p_1 \neq z$, ist es erlaubt, den Knoten z als nicht in p vorkommend anzunehmen. Dann gilt offensichtlich $p \in \mathcal{W}(g,(s \cup succ(g,z)) \setminus (t \cup \{z\}),y)$ und $q \cap (t \cup \{z\}) = \emptyset$ und die Wahl von p als q zeigt die rechte Seite.

Mit der Abkürzung $tz = t \cup \{z\}$, wobei z ein beliebiges Element aus der Menge s ist, führt die eben gezeigte Äquivalenz in Verbindung mit den obigen Berechnungen und der Definition von *sreach* zu

$$\begin{aligned} sreach(g,s,t) &= t \cup \{\ y \in V \ : \ \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset\ \} \\ &= tz \cup \{\ y \in V \ : \ \exists\, p \in \mathcal{W}(g,s,y) : p \cap t = \emptyset\ \} \\ &= tz \cup \{\ y \in V \ : \ \exists\, p \in \mathcal{W}(g,(s \cup succ(g,z)) \setminus tz,y) : p \cap tz = \emptyset\ \} \\ &= sreach(g,(s \cup succ(g,z)) \setminus tz, tz). \end{aligned}$$

Ist die Parameterrestriktion für s und t erfüllt, so bleibt sie auch für $(s \cup succ(g,z)) \setminus tz$ und tz gültig. Zusammenfassend haben wir also in dem betrachteten Fall mit den eingeführten Abkürzungen die Rekursion

$$sreach(g,s,t) = sreach(g,(s \cup succ(g,z)) \setminus tz, tz). \qquad (R_8)$$

Auch von den Gleichungen (R_6) bis (R_8) nach einem funktionalen ML-Programm ist der Weg nicht weit; siehe wiederum Abschnitt 8.6.3. Die Terminierung der durch (R_7) bis (R_8) gegebenen Rekursion folgt aus der Tatsache, daß, wegen der Parameterrestriktion, der zweite Parameter von *sreach* im Komplement des dritten Parameters enthalten ist und dieses Komplement bei jedem Aufruf echt (um den Knoten z) verkleinert wird. ■

Mit den Berechnungen der Abschnitte 8.6.1 und 8.6.2 ist die Hauptarbeit bei der Algorithmisierung der Erreichbarkeitsfunktion *reach* getan. Man beachte, daß es sich beim zweiten Erreichbarkeitsalgorithmus um einen nichtdeterministischen Algorithmus handelt, da der Knoten z aus s beliebig wählbar ist. Im folgenden geben wir die den Gleichungen (R_1) bis (R_5) bzw. (R_6) bis (R_8) entsprechenden ML-Rechenvorschriften und die zur Implementierung von Knotenmengen und Graphen notwendigen ML-Programmstücke an. Da ML keinen Nichtdeterminismus beinhaltet, haben wir in der ML-Version des zweiten Erreichbarkeitsalgorithmus die beliebige Auswahl von z aus s in einer deterministischen Weise vorzunehmen.

8.6.3 ML-Implementierungen der Erreichbarkeits-Algorithmen

Wir nehmen der Einfachheit halber die Knotenmenge V als ein Intervall der natürlichen Zahlen von der Eins bis `N` an. Beispielsweise haben wir für eine Knotenmenge mit 13 Knoten dann die nachstehenden ML-Deklarationen:

```
val N : int = 13;
type node = int;
```

Als nächstes stellt sich nach der Implementierung der Knoten die Aufgabe, Knotenmengen in der Sprache ML mit den wichtigsten Konstanten und Operationen zu realisieren. Hierzu gibt es viele Möglichkeiten. Die von uns nachfolgend gewählte Mengenimplementierung stellt eine Menge $s \in 2^V$ als Prädikat von V nach $\mathbb{B}$ dar. In ML liest sich dies wie folgt:

```
type nodeset = node -> bool;
```

Natürlich gibt es effizientere Implementierungen; wir haben diese nur gewählt, um die Konstante und Operationen, wie nachfolgend angegeben, möglichst einfach und für den Leser nachvollziehbar realisieren zu können. Wir verzichten auf eine weitere Kommentierung des folgenden ML-Codes, da sich die Bedeutung der deklarierten Konstante bzw. Rechenvorschriften jeweils direkt aus der Namensgebung ergibt.

```
val empty : nodeset = fn (u: node) => false;

fun isempty (s : nodeset) : bool =
  let fun ise (k : int) : bool =
        if k = 0 then true
                 else not(s(k)) andalso ise(k-1)
  in  ise(N) end;

fun insert (s : nodeset, x : node) : nodeset =
  fn (u : node) => if u = x then true else s(u);

fun delete (s : nodeset, x : node) : nodeset =
  fn (u : node) => if u = x then false else s(u);

fun union (s : nodeset, t : nodeset) : nodeset =
  fn (u : node) => s(u) orelse t(u);

fun diff (s : nodeset, t : nodeset) : nodeset =
  fn (u : node) => s(u) andalso not(t(u));

fun elem (s : nodeset) : node =
  let fun ele (k : int) : node =
        if s(k) then k else ele(k+1)
  in  ele(1) end;
```

Die Konstante `N` wird nur an einer Stelle verwendet, nämlich in der Rechenvorschrift `isempty`, die testet, ob eine Menge leer ist. Mit Hilfe von `elem` wird ein Element (hier das kleinste) einer nichtleeren Menge geliefert.

Im dritten Schritt haben wir nun Graphen zu implementieren. Wir stellen einen Graphen $g = (V, R)$ dar durch eine Funktion, die jedem Knoten die Menge seiner Nachfolger zuordnet. Dies führt in ML zur nachfolgenden Deklaration:

```
type graph = node -> nodeset;
```

Aufbauend auf den Typ `graph` der partiellen Funktionen von den Knoten in die Knotenmengen erhalten wir auch eine äußerst einfache Realisierung der Nachfolgerfunktion *succ* wie folgt:

```
fun succ (g : graph, x : node) : nodeset = g(x);
```

Nach diesen Vorbereitungen können wir nun die Rekursionen der Abschnitte 8.6.1 und 8.6.2 direkt nach ML übertragen. Im ersten Fall erhalten wir:

```
fun reach (g : graph, x : node) : nodeset =
  let fun succs (g : graph, s : nodeset) : nodeset =
        if isempty(s)
        then s
        else let val y : node = elem(s);
             in  union(succ(g,y),succs(g,delete(s,y))) end;
      fun lreach (k : int) : nodeset =
        if k = 0 then insert(empty,x)
             else insert(succs(g,lreach(k-1)),x)
  in  lreach(N-1) end;
```

Im Vergleich zu den Funktionen *reach* und *lreach* des ersten Erreichbarkeitsalgorithmus ist in der Rechenvorschrift `reach` die Rechenvorschrift `lreach` lokal deklariert. Durch diese Unterordnung ist es möglich, in `lreach` mit einem Parameter `k` auszukommen.

Auch in der folgenden ML-Realisierung des zweiten Erreichbarkeitsalgorithmus haben wir durch eine Unterordnung einen Parameter unterdrückt:

```
fun reach (g : graph, x : node) : nodeset =
  let fun sreach (s : nodeset, t : nodeset) : nodeset =
        if isempty(s)
        then t
        else let val z : node = elem(s);
                 val tz : nodeset = insert(t,z)
             in  sreach(diff(union(s,succ(g,z)),tz),tz) end
  in  sreach(insert(empty,x),empty) end;
```

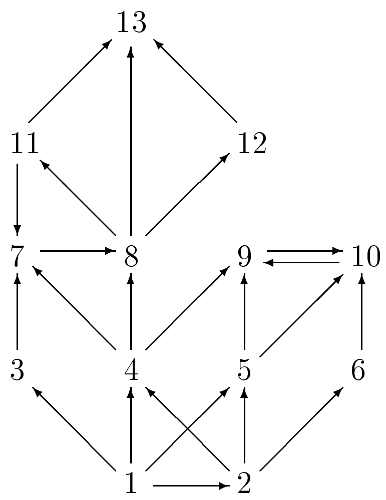
Die Rechenvorschrift `sreach` ist repetitiv, somit ist auch eine unmittelbare Formulierung in Modula-2 durch eine while-Schleife möglich. Im Fall von `lreach` ist hingegen vor einem Übergang zu Modula-2 erst noch eine Entrekursivierung zu bewerkstelligen. ■

Bei der oben angegebenen einfachen Implementierung von Mengen und Graphen durch Funktionstypen stellt sich natürlich das Problem, einen konkreten Graphen aufzubauen und das Resultat einer Algorithmenanwendung zu visualisieren. Das erste Problem kann etwa gelöst werden durch die zwei nachfolgenden Rechenvorschriften:

```
fun arcin (g : graph, x : node, y : node) : graph =
  fn (u : node) => if u = x then insert(succ(g,u),y)
                            else succ(g,u);

fun create (w : node list) : graph =
  if null(w) then fn (u : node) => empty
             else arcin(create(tl(tl(w))),hd(w),hd(tl(w)));
```

Dabei realisiert `arcin` das Einfügen eines einzelnen Pfeils in einen Graphen und `create` baut mittels `arcin` aus linearen Listen Graphen auf. Beispielsweise wird der Graph



wie folgt in ML als Objekt `g` des Typs `graph` erzeugt:

```
val g : graph = create([1,2,1,3,1,4,1,5,2,4,2,5,2,6,3,7,4,7,
                        4,8,4,9,5,9,5,10,6,10,7,8, 8,11,8,12,
                        8,13,9,10,10,9,11,7,11,13,12,13]);
```

Um dann die von einem Knoten aus erreichbaren Knoten beispielsweise als lineare Liste darzustellen, kann man etwa die Rechenvorschrift `filter` von Abschnitt 5.3.2.b verwenden, und die erreichbaren Knoten aus der linearen Liste aller Knoten herauszufiltern.

Nach dem Erreichbarkeitsproblem, seiner zweifachen algorithmischen Lösung in ML und der Diskussion des Graphenaufbaus bzw. der Visualisierung der Resultate kommen wir nun zu einem weiteren wichtigen graphentheoretischen Problem.

8.6.4 Test auf Kreisfreiheit

Vorausgesetzt sei ein **endlicher** Graph $g = (V, R)$. Wir wollen bestimmen, ob g **kreisfrei** ist. Dazu verwenden wir die folgende Äquivalenz:

$$g \text{ ist kreisfrei } \iff \forall\, t \in 2^V \;:\; (\forall\, x \in t\, \exists\, y \in t : \langle y, x\rangle \in R) \rightarrow t = \emptyset$$

Beweis: „$\Longrightarrow$“: Gäbe es eine nichtleere Menge t, in der jeder Knoten einen Vorgänger besitzt, so kann man eine unendliche Folge $\langle x_1, x_2, \ldots\rangle$ konstruieren, mit $\langle x_{i+1}, x_i\rangle \in R$ für alle $i \in \mathbb{N}$. Da g endlich ist, gibt es verschiedene Indizes i und j mit $i > j$ und $x_i = x_j$. Damit hat man in g einen Kreis $\langle x_i, \ldots, x_j\rangle$ gefunden, was ein Widerspruch ist.

„$\Longleftarrow$“: Auch diese Richtung beweisen wir durch Kontraposition. Wäre g nicht kreisfrei, so bekommt man durch die Knoten eines Kreises eine nichtleere Menge t, in der jeder Knoten einen Vorgänger besitzt.

Aufgrund der eben bewisenen Eigenschaft erhalten wir die folgende Spezifikation eines Prädikats *cyclefree*, das testet, ob ein Graph kreisfrei ist:

$$\mathit{cyclefree}(g) = \mathit{tt} \iff \forall\, t \in 2^V \;:\; (\forall\, x \in t\, \exists\, y \in t : \langle y, x\rangle \in R) \rightarrow t = \emptyset$$

Ein möglicher Ansatz zu einem Algorithmus zur Berechnung der Werte dieses Prädikats besteht darin, eine obere Schranke s für die in ihrer Spezifikation verwendete Knotenmenge t einzuführen. Dies führt zur Einbettung

$$\mathit{cyclefree}(g) = \mathit{cf}(g, V), \qquad (C_1)$$

mit einem zweistelligen Prädikat cf, welches festgelegt ist durch die Äquivalenz

$$\mathit{cf}(g,s) = \mathit{tt} \iff \forall\, t \in 2^s \;:\; (\forall\, x \in t \,\exists\, y \in t : \langle y, x\rangle \in R) \to t = \emptyset.$$

Um eine terminierende Rekursion für cf zu entwickeln, betrachten wir nun die Quellen von s, d.h. diejenigen Knoten aus s, die keine Vorgänger in s besitzen:

$$\mathit{sources}(g,s) = \{\; x \in s \;:\; \mathit{pred}(g,x) \cap s = \emptyset \;\}$$

Als Terminierungsfall der Rekursion bietet sich $\mathit{sources}(g,s) = \emptyset$ an. Dann gilt nämlich die Gleichung

$$\mathit{cf}(g,s) = (s = \emptyset). \qquad (C_2)$$

Zum Beweis von „$\Longrightarrow$" beachten wir, daß, wegen $\mathit{sources}(g,s) = \emptyset$, die Menge $s \in 2^s$ die Formel $\forall\, x \in s \,\exists\, y \in s : \langle y, x\rangle \in R$ erfüllt. Also impliziert $\mathit{cf}(g,s) = \mathit{tt}$, nach der obigen definierenden Äquivalenz, die Gleichung $s = \emptyset$. Die verbleibende Richtung „$\Longleftarrow$" ist klar, da die Formel

$$\forall\, t \in 2^{\emptyset} \;:\; (\forall\, x \in t \,\exists\, y \in t : \langle y, x\rangle \in R) \to t = \emptyset$$

wahr ist, also, wiederum nach der obigen definierenden Äquivalenz, $\mathit{cf}(g,s) = \mathit{tt}$ gilt.

Der verbleibende Fall ist $\mathit{sources}(g,s) \neq \emptyset$. Sei q eine beliebige Quelle in s. Ist dann $t \in 2^s$ eine Menge, in der jeder Knoten einen Vorgänger hat, so gilt $q \notin t$, also $t \in 2^{s\setminus\{q\}}$. Dies zeigt

$$\begin{array}{rl} & \forall\, t \in 2^s \;:\; (\forall\, x \in t \,\exists\, y \in t : \langle y, x\rangle \in R) \to t = \emptyset \\ \iff & \forall\, t \in 2^{s\setminus\{q\}} \;:\; (\forall\, x \in t \,\exists\, y \in t : \langle y, x\rangle \in R) \to t = \emptyset \end{array}$$

und die Definition des Prädikats cf bringt schließlich

$$\mathit{cf}(g,s) = \mathit{cf}(g, s \setminus \{q\}). \qquad (C_3)$$

Die drei Gleichungen (C_1) bis (C_3) stellen einen – wiederum nichtdeterministischen – terminierenden Algorithmus für das Prädikat *cyclefree* dar. ■

Mit der Deduktion des letzten Abschnittes ist wiederum die Hauptarbeit bei der Algorithmisierung des Tests auf Kreisfreiheit getan und eine Übertragung nach ML imgrunde trivial.

8.6.5 ML-Implementierung des Tests auf Kreisfreiheit

Wir setzen für die Knoten V und die Knotenmengen die Implementierungen von Abschnitt 8.6.3 voraus. Wegen der Gleichung (C_1) benötigen wir im folgenden noch eine Konstante des Typs `nodeset` für die gesamte Knotenmenge V. Bei der gewählten Implementierung von Knotenmengen durch den Funktionstyp `node -> bool` erhalten wir diese in einfachster Weise wie folgt:

```
val nodes : nodeset = fn (u : node) => true;
```

Setzen wir nun eine ML-Rechenvorschrift `sources` zur Implementierung der gleichnamigen Funktion *sources* voraus, so erhalten wir aus den drei Gleichungen (C_1) bis (C_3) von Abschnitt 8.6.4 bei einer Unterdrückung des Graphen in der Hilfsrechenvorschrift `cf` die Rechenvorschrift

```
fun cyclefree (g : graph) : bool =
  let fun cf (s : nodeset) : bool =
        let val t : nodeset = sources(g,s)
        in if isempty(t) then isempty(s)
                         else cf(delete(s,elem(t))) end
  in  cf(nodes) end;
```

zum Testen von Kreisfreiheit. Es verbleibt noch die Programmierung der Rechenvorschrift `sources`. Eine offensichtliche Realisierung mit Hilfe einer Rechenvorschrift

```
fun meet (s : nodeset, t : nodeset) : nodeset =
  fn (u : node) => s(u) andalso t(u);
```

zur Durchschnittsbildung ist:

```
fun sources (g : graph, s : nodeset) : nodeset =
  let fun so (t : nodeset) : nodeset =
        if isempty(t)
        then empty
        else let val x : node = elem(t)
             in  if isempty(meet(pred(g,x),s))
                 then insert(so(delete(t,x)),x)
                 else so(delete(t,x)) end
  in  so(s) end;
```

Diese Realisierung testet Knoten für Knoten daraufhin ab, ob die die Menge $sources(g, s)$ beschreibende Gleichung $pred(g, x) \cap s = \emptyset$ erfüllt ist, oder nicht. Sie setzt aber voraus, daß, im Gegensatz zu Abschnitt 8.6.3, nun ein Graph nicht durch die Nachfolgerfunktion *succ*, sondern durch die **Vorgängerfunktion** *pred* **dargestellt** wird. ■

Es mag verwundern, warum in Abschnitt 8.6.5 eine vorgängerorientierte Graphendarstellung verwendet wurde, in Abschnitt 8.6.4 jedoch eine nachfolgerorientierte, denn es ist offensichtlich, daß man Kreisfreiheit auch feststellen kann, indem man fortwährend Senken entfernt. Der Grund für die Vorgängermengen-Darstellung in Abschnitt 8.6.5 ist, daß eine leichte Modifikation der Rechenvorschrift `cyclefree` sofort zu einer Rechenvorschrift führt, der einen kreisfreien Graphen topologisch sortiert. Dies wird nachfolgend demonstriert.

8.6.6 Topologisches Sortieren von Graphen in ML

Eine topologische Sortierung ts eines **kreisfreien** Graphen $g = (V, R)$ ordnet den Knoten $x \in V$ Nummern $ts(x)$ so zu, daß Pfeile nur von kleineren zu größeren Nummern führen. Statt Nummern zu verwenden, kann man bei einer topologischen Sortierung von g die Knotenmenge V auch in eine lineare Liste überführen, so daß sich die Numerierung durch die Reihenfolge der Listenelemente von links nach rechts ergibt. Das folgende Beispiel soll dies verdeutlichen:

$$\begin{tikzcd} a \arrow[r] \arrow[d] & b \arrow[d] \\ c \arrow[ur] & d \end{tikzcd}$$

ts		
a	$\longrightarrow$	1
b	$\longrightarrow$	3
c	$\longrightarrow$	2
d	$\longrightarrow$	4

Liste
$\langle a, c, b, d \rangle$

Die Berechnung einer linearen Liste, welche eine topologische Sortierung eines kreisfreien Graphen darstellt, ist möglich durch die nachfolgend angegebene leichte Modifikation der Rechenvorschrift `cyclefree` des letzten Abschnitts:

```
fun topsort (g : graph) : node list =
  let fun ts (s : nodeset) : node list =
        let val t : nodeset = sources(g,s)
        in  if isempty(t) then nil
                          else let val x : node = elem(t)
                               in  x :: ts(delete(s,x)) end
  in  ts(nodes) end;
```

Dabei ist wiederum angenommen, daß ein Graph durch die **Vorgängerfunktion** *pred* **dargestellt** wird. Algorithmisch betrachtet, sucht die Hilfsrechenvorschrift `ts` irgendeine Quelle im gesamten Graphen und weist ihr die Nummer 1 (bzw. die erste Listenposition) zu. Wegen der Kreisfreiheit existiert eine Quelle. Dann wird diese entfernt. Eine nächste existierende Quelle im kreisfreien Restgraphen bekommt die Nummer 2 (bzw. die zweite Listenposition) und so weiter, bis schließlich alle Knoten entfernt sind. Auf eine formale Herleitung der `topsort` entsprechenden Rekursion aus einer mathematischen Problemspezifikation verzichten wir. ■

Der große Mangel der in den Abschnitten 8.6.5 und 8.6.6 vorgestellten ML-Programme ist die Quellensuche. Sie erfordert, sogar bei einer optimalen Implementierung der Knotenmengen, bei einem Graphen mit N Knoten $\mathcal{O}(N^2)$ Aufwand, so daß der Gesamtaufwand sowohl des Tests auf Kreisfreiheit als auch des topologischen Sortierens $\mathcal{O}(N^3)$ beträgt. Führt man jedoch einen Vektor der „aktuellen" Eingangsgrade als zusätzlichen Parameter ein, so kann man Quellen offensichtlich mit linearem Aufwand bestimmen, indem man jeden Knoten auf den Eingangsgrad 0 testet. Dies führt dann zu Versionen von `cyclefree` und `topsort` mit quadratischer Laufzeit. Sammelt man schließlich im Laufe des Graphenabbaus auch noch alle Quellen mit Hilfe eines weiteren Parameters in einer Liste auf, so erhält man sogar $\mathcal{O}(max(M, N))$ als Komplexität, mit $M = |R|$, also als Anzahl der

Pfeile. Man nennt die Technik, komplexe Ausdrücke mittels weiterer Parameter (bzw. im imperativen Fall mittels weiterer Programmvariablen) inkrementell zu berechnen, **formales Differenzieren**; wir müssen an dieser Stelle jedoch auf entsprechende weiterführende Literatur verweisen.

Als nächstes behandeln wir nun das Problem der Berechnung von Hüllen. Wegen der Gleichungen $R^* = I \cup R^+$ und $R^+ = RR^* = R^*R$ genügt es, einen Algorithmus für eine der beiden Hüllen zu haben, da man aus diesem sofort einen für die andere konstruieren kann. Im folgenden konzentrieren wir uns auf die transitive Hülle R^+. Die nachstehenden Resultate stehen in einer sehr engen Beziehung zu Satz 4.4.8. Der allgemeine Hintergrund ist die Fixpunkttheorie auf vollständigen Verbänden bzw. auf vollständig geordneten Mengen, den sogenannten Cpos. Genaueres findet der Leser in Büchern über Verbandstheorie oder die Semantik von Programmiersprachen.

8.6.7 Fixpunkt-Algorithmen zur Berechnung der transitiven Hülle

Gegeben sei ein **endlicher** Graph $g = (V, R)$ mit N als Knotenanzahl und einer Relation R auf V. Dann gilt für die transitive Hülle $R^+ = R^*R = (I \cup R^+)R = R \cup R^+R$, also, mit der Funktion

$$\tau \ : \ 2^{V\times V} \longrightarrow 2^{V\times V} \qquad\qquad \tau(X) = R \cup XR$$

auf Relationen, die Gleichung $R^+ = \tau(R^+)$. Ein Element X mit $X = \tau(X)$ nennt man, wie bei der denotationellen Semantik von ML schon eingeführt, einen **Fixpunkt** von τ.

Es bezeichne $O \in 2^{V\times V}$ die Nullrelation. Weiterhin sei $X \in 2^{V\times V}$ ein Fixpunkt von τ. Die Funktion τ ist **monoton** in dem Sinne, daß für alle Relationen $R_1, R_2 \in 2^{V\times V}$ aus $R_1 \subseteq R_2$ folgt $\tau(R_1) \subseteq \tau(R_2)$. Eine unmittelbar durch Induktion beweisbare Folgerung der Monotonie-Eigenschaft ist die Inklusion $\tau^n(O) \subseteq X$ für alle $n \in \mathbb{N}$. Der Induktionsbeginn ist $\tau^0(O) = O \subseteq X$ und beim Induktionsschluß folgt $\tau^{n+1}(O) = \tau(\tau^n(O)) \subseteq \tau(X) = X$ aus $\tau^n(O) \subseteq X$ in Verbindung mit der Monotonie und der Fixpunkt-Eigenschaft. Alles zusammen zeigt $R^+ = \bigcup_{n\geq 1} R^n \subseteq X$ und R^+ ist somit der **kleinste Fixpunkt** $\mu(\tau)$ von τ bezüglich der Relationeninklusion.

Kleinste Fixpunkte von monotonen Funktionen berechnet man vom kleinsten Element ausgehend dadurch, daß man die Funktion immer wieder anwendet. Man vergleiche auch mit Satz 4.4.8. In unserem Fall führt diese Iteration zu einer Kette

$$O \subseteq \tau(O) \subseteq \tau^2(O) \subseteq \tau^3(O) \subseteq \dots \qquad (K_1)$$

in der geordneten Menge $(2^{V\times V}, \subseteq)$, welche, wegen der Endlichkeit von $2^{V\times V}$, irgendwann stationär wird. Für die einzelnen Kettenglieder von (K_1) gilt $\tau^n(O) = \bigcup_{i=1}^{n} R^i$, was sich leicht durch Induktion beweisen läßt, und als Grenzwert der Kette ergibt sich somit

$$\bigcup_{n\geq 0} \tau^n(O) = \bigcup_{n\geq 0} \bigcup_{i=1}^{n} R^i = \bigcup_{n\geq 1} R^n = R^+ = \mu(\tau).$$

Wegen $R^+ = \bigcup_{n=1}^{N-1} R^n$ wird die Kette (K_1) spätestens beim $N-1$-ten Element stationär.

Formulieren wir nun die Berechnung der einzelnen Kettenglieder von (K_1) in Modula-2-artiger Schreibweise, so erhalten wir das while-Programm

X `:=` O;
`WHILE` $X \neq \tau(X)$ `DO` X `:=` $\tau(X)$ `END`

mit Schleifeninvariante $X \subseteq \mu(\tau)$. Die Etablierung der Invariante durch die Initialisierung und die Erhaltung ihrer Gültigkait durch den Schleifenrumpf sind offensichtlich. Nach Beendigung der Schleife gilt folglich $X \subseteq \mu(\tau)$ und $X = \tau(X)$, woraus die gewünschte Nachbedingung $X = \mu(\tau) = R^+$ folgt.

Der eben hergeleitete Hüllenalgorithmen kann offensichtlich auch in „echtem" Modula-2 ohne größeren Aufwand realisiert werden. Man benötigt dazu nur eine geeignete Darstellung von Relationen. Hier bieten sich zweidimensionale Felder an, mit dem Knotentyp als beiden Indextypen und `BOOLEAN` als Komponententyp. Die Nullrelation wird dann zum Feld, welches nur `FALSE`-Einträge besitzt; die Programmierung von Rechenvorschriften zum Testen der Gleichheit von Relationen und von Prozeduren zur Vereinigung und zur Produktbildung ist offensichtlich. Bei N als Knotenzahl erfordert in dieser Implementierung die Berechnung von $\tau(X)$ größenordnungsmäßig $\mathcal{O}(N^3)$ Rechenschritte. Die while-Schleife wird $\mathcal{O}(N)$-mal durchlaufen. Insgesamt hat der obige Fixpunkt-Algorithmus zur Berechnung von R^+ also eine Komplexität von $\mathcal{O}(N^4)$.

Die transitive Hülle R^+ kann jedoch auch schneller berechnet werden. Dazu betrachten wir die, im Vergleich zu τ nur geringfügig modifizierte, monotone Funktion

$$\rho \;:\; 2^{V\times V} \longrightarrow 2^{V\times V} \qquad\qquad \rho(X) = R \cup XX$$

und berechnen analog zu τ den kleinsten Fixpunkt $\mu(\rho)$ von ρ ebenfalls als Grenzwert der Kette

$$O \subseteq \rho(O) \subseteq \rho^2(O) \subseteq \rho^3(O) \subseteq \dots \tag{K_2}$$

der iterierten Funktionsanwendungen auf die Nullrelation. Es kann allgemein durch eine Induktion für alle natürlichen Zahlen $n \geq 1$ die Gleichung $\rho^n(O) = \bigcup_{i=1}^{2^{n-1}} R^i$ gezeigt werden. Damit gilt auch

$$\bigcup_{n\geq 0} \rho^n(O) = O \cup \bigcup_{n\geq 1} \rho^n(O) = O \cup \bigcup_{n\geq 1} \bigcup_{i=1}^{2^{n-1}} R^i = \bigcup_{n\geq 1} R^n = R^+.$$

Im Gegensatz zur Kette (K_1) wird die Kette (K_2) jedoch schon spätestens nach $\mathcal{O}(log_2 N)$ Gliedern stationär.

Fassen wir das vorhergehende zusammen und bedenken wir, daß die Berechnung von $\rho(X)$ in der Feldimplementierung von Relationen auch $\mathcal{O}(N^3)$ Rechenschritte erfordert, so bekommen wir den Algorithmus

X `:=` O;
`WHILE` $X \neq \rho(X)$ `DO` X `:=` $\rho(X)$ `END`

mit einem Aufwand $\mathcal{O}(N^3 * log_2 N)$. Um die partielle Korrektheit dieses Pseudo-Modula-2-Programms bezüglich der Nachbedingung $X = R^+$ zu zeigen, verwendet man $X \subseteq \mu(\rho)$

als Schleifeninvariante. Nach Beendigung der Schleife gilt dann $X = \mu(\rho)$. Die, analog zum ersten Ansatz einfach zu zeigende, Fixpunkteigenschaft $R^+ = \mu(\rho)$ liefert schließlich das Gewünschte. ■

Nach diesen beiden Fixpunkt-Algorithmen kommen wir nun zum Abschluß noch zu einem dritten und noch schnelleren Algorithmus zur Berechnung von R^+, der S. Warshall (1962) zugeschrieben wird, aber vor ihm schon von B. Roy im Jahr 1959 publiziert wurde. Zu seiner Herleitung benötigen wir den Begriff der **inneren Knoten eines nichtleeren Wegs**, definiert durch

$$inner(\langle y_0, \dots, y_n \rangle) = \{\, y_1, \dots, y_{n-1} \,\}.$$

Man beachte, daß im Fall $n = 1$, also eines Wegs der Länge 1, die Menge der inneren Knoten leer ist.

8.6.8 Der Algorithmus von Roy und Warshall

Wir betrachten, analog zu Abschnitt 8.6.7, einen **endlichen** Graphen $g = (V, R)$ mit $N = |V|$. Weiterhin nehmen wir an, daß die Knotenmenge durch $V = \{x_1, \dots, x_N\}$ gegeben sei. Nun definieren wir, bei einer Unterdrückung des gegebenen Graphen g, ein dreistelliges Prädikat W auf $V \times V \times \mathbb{N}$ durch

$$W(x, y, n) = tt \iff \begin{array}{l}\text{Es gibt einen nichtleeren Weg } p \text{ von} \\ x \text{ nach } y \text{ mit } inner(p) \subseteq \{x_1, \dots, x_n\}.\end{array}$$

Nach Satz 8.5.4.a existiert ein nichtleerer Weg p von x nach y genau dann, wenn die Beziehung $\langle x, y \rangle \in R^+$ gilt. Da $inner(p) \subseteq \{x_1, \dots, x_N\} = V$ immer wahr ist, folgt daraus unmittelbar die Äquivalenz

$$W(x, y, N) = tt \iff \langle x, y \rangle \in R^+, \qquad (W_1)$$

welche die Beziehung zwischen der eigentlich zu berechnenden Relation R^+ und dem Hilfsprädikat W herstellt. Nun beweisen wir eine terminierende Rekursion für das Prädikat W, die durch das dritte Argument gesteuert wird. Wegen der Einbettung (W_1) bietet sich $n = 0$ als Terminierungsfall an. Hier haben wir die Äquivalenz

$$W(x, y, 0) = tt \iff \langle x, y \rangle \in R, \qquad (W_2)$$

da die Menge der nichtleeren Wege p mit $inner(p) \subseteq \emptyset$ genau die Menge R der Pfeile ist und somit solch ein Weg von x nach y genau dann existiert, wenn es einen Pfeil $\langle x, y \rangle \in R$ gibt.

Es verbleibt noch der Fall $n \neq 0$. Hier gilt

$$W(x, y, n) = W(x, y, n-1) \vee (W(x, x_n, n-1) \wedge W(x_n, y, n-1)). \qquad (W_3)$$

Beweis: „$\Longrightarrow$“: Es gelte $W(x, y, n) = tt$ und es sei p ein Weg von x nach y mit der Eigenschaft $inner(p) \subseteq \{x_1, \dots, x_n\}$. Offensichtlich dürfen wir auch annehmen, daß alle inneren Knoten von p verschieden sind. Nun gibt es zwei Fälle. Ist x_n kein Knoten von p, dann gilt sogar $inner(p) \subseteq \{x_1, \ldots, x_{n-1}\}$ und somit $W(x, y, n-1) = tt$ nach der Definition von W. Kommt x_n hingegen als ein Knoten von p vor, so kann man p schreiben als Konkatenation $p = q_1\, x_n\, q_2$ mit Teilwegen $q_1\, x_n$ von x nach x_n und $x_n\, q_2$ von x_n nach y, so daß

$$inner(q_1\, x_n) \subseteq \{x_1, \ldots, x_{n-1}\} \qquad\qquad inner(x_n\, q_2) \subseteq \{x_1, \ldots, x_{n-1}\}.$$

Nach Definition von W zeigt dies $W(x, x_n, n-1) = tt$ und $W(x_n, y, n-1) = tt$.

„$\Longleftarrow$“: Auch hier gibt es zwei Fälle. Gilt $W(x, y, n-1) = tt$, so offensichtlich auch $W(x, y, n) = tt$, denn jeder nichtleere Weg mit $inner(p) \subseteq \{x_1, \ldots, x_{n-1}\}$ erfüllt auch $inner(p) \subseteq \{x_1, \ldots, x_n\}$. Hat man hingegen $W(x, x_n, n-1) = tt$ und $W(x_n, y, n-1) = tt$, d.h. die Existenz von nichtleeren Wegen $q_1\, x_n$ von x nach x_n und $x_n\, q_2$ von x_n nach y, die beide nur Knoten aus $\{x_1, \ldots, x_{n-1}\}$ enthalten, so ist $p := q_1\, x_n\, q_2$ ein Weg von x nach y mit $inner(p) \subseteq \{x_1, \ldots, x_n\}$. Dies zeigt $W(x, y, n) = tt$. ∎

Durch die drei Eigenschaften (W_1), (W_2) und (W_3) ist ein rekursiver Algorithmus beschrieben, der die transitive Hülle R^+ einer Relation R „paarweise“ berechnet. Nachfolgend betrachten wir nun programmiersprachliche Implementierungen dieses Algorithmus sowohl in ML als auch in Modula-2.

8.6.9 Implementierungen des Algorithmus von Roy und Warshall

Es ist offensichtlich, daß sich der durch (W_1), (W_2) und (W_3) beschriebene rekursive Algorithmus unmittelbar in die Programmiersprache ML übertragen läßt. Man benötigt dazu zum vorliegenden endlichen Graphen $g = (V, R)$ mit $N = |V|$ als Knotenanzahl nur ML-Implementierungen der Knotenmenge V, der Konstanten N und der Relation R.

Bei den Knoten und der Knotenanzahl gehen wir, der Einfachheit halber, im folgenden analog zu den Implementierungen der beiden Erreichbarkeitsalgorithmen in Abschnitt 8.6.3 vor, deklarieren sie also beispielsweise mittels

```
val N : int = 30;
type node = int;
```

im Fall eines Graphen mit 30 Knoten. Auch die Relationenimplementierung könnten wir direkt von den Erreichbarkeitsalgorithmen übernehmen. Die Äquivalenz (W_2) suggeriert jedoch eine noch einfachere Darstellung von R durch ein Prädikat vom direkten Produkt $V \times V$ in die Wahrheitswerte $\mathbb{B}$. Dieses Prädikat bildet ein Knotenpaar $\langle x, y\rangle$ genau dann nach tt ab, falls es in der Relation R enthalten ist. In der Sprache ML führt diese Implementierung zur nachfolgenden Typdeklaration:

```
type graph = node * node -> bool;
```

Aufbauend auf die bisherigen Deklarationen von `N`, `node` und `graph` bekommen wir die Rekursionsgleichungen und die einbettende Beziehung des Algorithmus von Roy und Warshall dann in der folgenden Form:

```
fun warshall (g : graph) : graph =
  let fun W (x : node, y : node, n : node) : bool =
        if n = 0 then g(x,y)
                 else W(x,y,n-1) orelse
                      (W(x,n,n-1) andalso W(n,y,n-1))
  in  fn (x : node, y : node) => W(x,y,N) end;
```

Jeder Aufruf der Rechenvorschrift `W` führt schlimmstenfalls zu 3 Folgeaufrufen. Eine einfache Rechnung zeigt, daß der Aufruf `W(g)` insgesamt $\mathcal{O}(3^{\mathtt{N}})$ rekursive Aufrufe von `W` nach sich zieht. Damit ist der durch die Rechenvorschrift `warshall` gegebene Algorithmus nicht praktikabel.

Um die Anzahl der rekursiven Aufrufe der Rechenvorschrift `W` während der Abarbeitung in vernünftigen (d.h. polynominalen) Grenzen zu halten, bietet sich eine Tabellierung der Werte von `W(x,y,n)` in Abhängigkeit von `n` an. Formal wird diese Tabellierung beschreiben durch eine neue Rechenvorschrift der nachstehenden Form:

```
fun wt (n : node) : graph =
  fn (x : node, y : node) => W(x,y,n);
```

Unter Verwendung der Definitionen der Rechenvorschriften `warshall` und `wt` bekommen wir dann sofort die Gleichung

```
warshall(g) = fn (x : node, y : node) => W(x,y,N)
            = wt(N),
```

welche angibt, wie sich `warshall` auf `wt` abstützt. Um nun `wt` statt `W` in `warshall` verwenden zu können, brauchen wir jedoch eine direkt-rekursive Version von `wt`, welche wir nun herleiten.

Den Terminierungsfall `n` $=$ `0` erhält man unter Verwendung der Definition von `wt`, der Eigenschaft (W_2) und der extensionalen Gleichheit von Funktionen wie folgt:

```
wt(0) = fn (x : node, y : node) => W(x,y,0)
      = fn (x : node, y : node) => g(x,y)
      = g
```

Im anderen Fall `n` $\neq$ `0` setzen wir die Abkürzung `h` $=$ `wt(n-1)` voraus und erhalten mit ihrer Hilfe und wegen `wt(n)(x,y)` $=$ `W(x,y,n)` die Gleichheit

```
  wt(n)
= fn (x : node, y : node) => W(x,y,n)
= fn (x : node, y : node) => W(x,y,n-1) orelse
                             (W(x,n,n-1) andalso W(n,y,n-1))
= fn (x : node, y : node) => wt(n-1)(x,y) orelse
                             (wt(n-1)(x,n) andalso wt(n-1)(n.y))
= fn (x : node, y : node) => h(x,y) orelse (h(x,n) andalso h(n.y)).
```

Wählen wir nun an Stelle der kaskadenartig-rekursiven Rechenvorschrift `W` in der ersten Version von `warshall` die Rechenvorschrift `wt` und bezeichnen diese ebenfalls mit dem Symbol `W`, so erhalten wir aus den obigen Rechnungen die folgende tabellierende Version von `warshall` mit einer linear-rekursiven Hilfsrechenvorschrift:

```
fun warshall (g : graph) : graph =
  let fun W (n : node) : graph =
        if n = 0 then g
                 else let val h : graph = W(n-1)
                      in  fn (x : node, y : node) =>
                          h(x,y) orelse (h(x,n) andalso h(n,y)) end
  in  W(N) end;
```

Im Gegensatz zur Originalversion zieht hier ein Aufruf von `W` nur mehr einen Folgeaufruf nach sich, d.h. man hat $\mathcal{O}(\mathtt{N})$ rekursive `W`-Aufrufe. Praktische Versuche zeigen jedoch, daß auch diese Rechenvorschrift nicht sehr effizient die transitive Hülle berechnet. Dies liegt an der von uns gewählten Darstellung der Relationen / Graphen, genauer am Funktionstyp `node * node -> node`. Der Bezeichnung `h` wird `W(n-1)` nämlich nicht als Funktionsobjekt zugeordnet, in dem Sinne, daß eine ausgerechnete Tabelle der Funktionswerte mit konstantem Zugriff vorliegt, sondern als Term. Diese interne Termdarstellung von Funktionsobjekten in ML bewirkt, daß die drei Aufrufe von `h` im Rumpf der Funktionsabstraktion indirekt wieder zu den drei Folgeaufrufen von `W` führen.

Die Rechenvorschrift `warshall` in ihrer tabellierenden Forn ist also noch nicht praktikabel, aber nichtsdestoweniger auch nicht ganz unnütz. Sie führt nämlich zu einer effizienten ML-Implementierung des Algorithmus von Roy und Warshall, wenn man Graphen nicht durch Funktionstypen implementiert, sondern etwa durch lineare Listen. Natürlich verkompliziert sich dadurch die Berechnung des neuen Graphen aus dem alten `h`, was durch die Funktionsabstraktion

```
fn (x : node, y : node) =>
  h(x,y) orelse (h(x,n) andalso h(n,y))
```

in dem obigen Programm so einfach war.

Es scheint deshalb ein vernünftiger Weg zu sein, von der tabellierenden ML-Version des Algorithmus von Roy und Warshall zu einer tabellierenden imperativen Version überzugene, in der zweidimensionale Boolesche Felder zur Darstellung von Graphen verwendet werden. Dieser Weg wäre auch in ML gehbar, da die Sprache ML, neben den in diesem Skriptum behandelten funktionalen Sprachkonstruktionen, auch noch einige imperative Konzepte und darunter insbesondere zweidimensionale Felder beinhaltet. Da wir jedoch ML nur als funktionale Modellsprache des Skriptums gewählt haben, geben wir nachfolgend die imperative Version des Algorithmus von Roy und Warshall in der imperativen Modellsprache Modula-2 dieses Skriptums an.

Im nachfolgenden Modula-2-Programm `Warshall` wird der eigentliche Algorithmus von Roy und Warshall im Programmrumpf durch die folgenden drei geschachtelten Schleifen implementiert; die `i`-Schleife entspricht dabei der Rekursion über `n` in der tabellierenden ML-Rechenvorschrift `W` und die `j`- und `k`-Schleife entsprechen der Berechnung der

Funktionsabstraktion über x und y:

```
FOR i := 1 TO N DO
  FOR j := 1 TO N DO
    FOR k := 1 TO N DO
      g[j,k] := g[j,k] OR (g[j,i] AND g[i,k])
  END END END
```

Der Rest des Programmrumpfs dient nur zum Einlesen des Graphen in ein Boolesches Feld bzw. zur Ausgabe der transitiven Hülle. Hier ist das gesamte Programm:

```
MODULE Warshall;
FROM Inout IMPORT WriteLn, WriteString, ReadCard;
CONST N = 30;
TYPE node = [1..N];
     graph = ARRAY node,node OF BOOLEAN;
VAR  g : graph;
     i, j, k, p : CARDINAL;
BEGIN
  FOR i := 1 TO N DO
    FOR j := 1 TO N DO
      g[i,j] := FALSE END END;
  WriteString('Gib Anzahl der Pfeile ein.');
  WriteLn;
  ReadCard(p);
  FOR k := 1 TO p DO
    WriteString('Gibt Pfeil als Paar ein.');
    WriteLn;
    ReadCard(i);
    ReadCard(j);
    g[i,j] := TRUE END;
  FOR i := 1 TO N DO
    FOR j := 1 TO N DO
      FOR k := 1 TO N DO
        g[j,k] := g[j,k] OR (g[j,i] AND g[i,k])
    END END END;
  FOR i := 1 TO N DO
    FOR j := 1 TO N DO
      IF g[i,j] THEN WriteString('L ')
                ELSE WriteString('O ') END END;
    WriteLn
    END
END Warshall .
```

Offensichtlich ist die Laufzeit des obigen Modula-2-Programms von kubischer Ordnung, d.h. wir haben einen $\mathcal{O}(\mathtt{N}^3)$-Algorithmus. ■

Es ist klar, daß geringfügige Modifikationen von Warshall erlauben, verschiedenste graphentheoretische Probleme zu lösen. Beispielsweise ist ein Knoten `y` von einem anderen Knoten `x` aus genau dann erreichbar, wenn nach der Änderung von `g` zur transitiven Hülle gilt

```
g[x,y] = TRUE.
```

Auch Kreisfreiheit kann sehr einfach durch eine Modifikation des Programms `Warshall` getestet werden, siehe Satz 8.5.4.c. Gilt nach Änderung von `g` zur transitiven Hülle nämlich

```
g[x,x] = FALSE.
```

für alle Knoten `x`, so ist der eingegebene Originalgraph kreisfrei. Schließlich berechnet das Programm `Warshall` die reflexiv-transitive Hülle der Eingabe, falls man nach den drei geschachtelten Schleifen die for-Schleife

```
FOR i := 1 TO N DO
  g[i,i] := TRUE END
```

einfügt, die alle Diagonalelemente des den Graphen darstellenden zwei-dimensionalen Felds zu `TRUE` setzt.

A Datenabstraktion in ML und Modula-2

Der Mechanismus der Datenabstraktion ist ein wesentliches Hilfsmittel bei der inkrementellen Konstruktion großer Programmsysteme. In diesem Anhang geben wir zuerst eine knappe Einführung in das Gebiet der Datenabstraktion – zusammen mit einigen historischen Betrachtungen – und beschreiben dann etwas genauer, welche Möglichkeiten unsere beiden Modellsprachen ML und Modula-2 für das funktionale bzw. imperative Programmieren bieten. Das Thema Datenabstraktion ist natürlich viel zu umfassend, als daß wir es hier erschöpfend behandeln können.

A.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.

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 Idee 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 **Daten- und Operationenkapselung** 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 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 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 und Modula-2 zeigen.

A.1.1 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) 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.

A.1.2 Importation und Exportation

Eine geschlossene Einheit eines großen Programmsystems repräsentiert eine Sammlung von Datentypen, Programmvariablen (im imperativen Fall), Konstanten und Operationen (Prozeduren und Rechenvorschriften). Exportation ist die Kennzeichnung der Programmteile, die eine Einheit nach außen zur Verwendung in anderen Einheiten zur Verfügung stellt. Diese Programmteile werden auch **sichtbare Konstituenten** genannt. 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 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.

A.2 Datenabstraktion in ML

Nach der Einführung in die Datenabstraktion skizzieren wir nun im folgenden die Sprachmittel zur Datenabstraktion der beiden Sprachen dieser Vorlesung. Dabei halten wir uns an die Reihenfolge ihrer Einführung, beginnen also mit 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 mit den abstrakten Datentypen.

A.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 $\gg$ *Typausdruck* $\ll$ 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.

A.2.2 Beispiel (Abstrakter Datentyp für 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 dar. Wir beginnen mit den Strukturen.

A.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 zur Vereinfachung von Selektionen bei Verbunden existiert in ML eine Abkürzung für Strukturzugriffe. Beispielsweise entspricht der normalen Schreibweise $h(S.f(S.c,A))$ die abkürzende Schreibweise `let open` S `in` $h(f(c,A))$ `end`, 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 und Modula-2 behandelten Deklarationen, auf die üblichen Fragen nach Gültigkeit usw, welche es bei den Zugriffen zu beachten gilt. Darauf soll jedoch im Rahmen dieses Anhangs 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 A.2.2 als Struktur `NAT` an.

A.2.4 Beispiel (Struktur für natürliche Zahlen)

Das folgende ML-Programm zeigt die Übersetzung des abstrakten Datentyps von Beispiel A.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;
```

Man beachte, daß bei einer Verwendung von `NAT` in anderen Programmteilen wegen der Nichtkapselung auch Zugriffe der Formen `NAT.mkzero()` und `NAT.mknext(`x`)` möglich sind. ■

Weitere Beispiele für Strukturen geben wir später an. Zuerst behandeln wir jedoch die noch ausstehenden Sprachmittel der ML-Modulsprache.

A.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. Definition 3.3.1). In ML kann nun die **Signatur** A **passend zu einer Struktur** S eigens deklariert werden. Die Syntax einer Signatur sieht schematisch wie folgt aus:

`signature` A `=`
  `sig` $\gg$ *Liste der Typen der Konstituenten von* $S \ll$ `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. ■

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.

A.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 in S deklarierten Typen, Konstanten und Rechenvorschriften enthält. Ändert man nun die (in der originalen Syntax gegebene) Deklaration von S zu

```
structure S : A =
  structure ≫Liste von Deklarationen≪ end;
```

ab, d.h. stellt man dem Namen durch ein Semikolon abgetrennt den Signaturnamen nach, so heißt nun S eine durch A beschränkte Signatur. Von den in S deklarierten Typen, Konstanten und Rechenvorschriften sind dann nur die in A aufgeführten Konstituenten nach außen sichtbar.

Als Beispiel betrachten wir noch einmal die natürlichen Zahlen. Beschreibt man sie nicht durch eine unbeschränkte Signatur wie in Beispiel A.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 A.2.4, so werden nun die Konstituenten `mkzero` und `mknext` verborgen. ■

Ist eine Struktur S in der eben beschriebenen Form 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. Erlaubt sind also beispielsweise eine Struktur S und eine Signatur A wie nachfolgend skizziert:

```
structure S : A =            signature A =
    ...                          ...
  fun f(x) = x;                val f : 'a list -> 'a list;
    ...                          ...
  end;                         end;
```

Der Grund ist, daß in der Struktur S die Rechenvorschrift f den polymorphen Typ $'b$`->`$'b$ mit einer Typvariablen $'b$ besitzt und in der Signatur A der Bezeichnung f der Typ $'a$ `list` `->` $'a$ `list` zugeordnet ist, mit einer Typvariablen $'a$. Den Signatur-Typ von f erhält man aber aus dem Struktur-Typ durch Ersetzung von $'b$ durch $'a$ `list`.

A.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

`functor` F `(`s `:` A_1`)` `:` A_2 `=` $\gg$*Rumpf*$\ll$`;`

mit zwei Signaturen A_1 und A_2 und einem formalen Parameter s für eine Struktur passend zu A_1. Im Rumpf von F wird mit Hilfe von s eine Struktur definiert, die zur Signatur A_2 paßt. ∎

Das folgende einfache Beispiel aus der Algebra soll die Benutzung von Funktoren zur Parametrisierung von Strukturen in der Programmiersprache ML verdeutlichen.

A.2.8 Beispiel (Funktoren in ML)

Ein Monoid ist eine algebraische Struktur bestehend aus einer nichtleeren Menge, einer assoziativen Verknüpfung auf dieser Menge und einem neutralen Element aus dieser Menge. Benutzt man die übliche mathematische Schreibweise, so entspricht einem Monoid $(M, \circ, e)$ die folgende ML-Signatur `MONOID`:

```
signature MONOID =
  sig type m;                    Elemente
      val comp : m*m -> m;       innere Operation
      val e : m;                 neutrales Element
  end;
```

In Monoiden kann man potenzieren, wobei die Potenzfunktion induktiv definiert wird durch $x^0 = e$ und $x^{n+1} = x^n \circ x$. Damit hat man also eine erweiterte algebraische Struktur $(M, \circ, e, pow)$, welcher, unter Verwendung von ganzen statt natürlichen Zahlen, die folgende ML-Signatur `POWER` entspricht:

```
signature POWER =
  sig type m;                       Elemente
      val comp : m*m -> m;          innere Operation
      val e : m;                    neutrales Element
      val pow : m*int -> m;         Potenzierung
   end;
```

Die Abbildung einer Struktur passend zur Signatur `MONOID` zu einer Struktur passend zur Signatur `POWER` wird in ML nun durch den folgenden Funktor `F` beschrieben, in dem eine Rechenvorschrift zur Potenzierung rekursiv erklärt wird:

```
functor F (mon : MONOID) : POWER =
  struct type m = mon.m;
         val comp : m*m -> m = mon.comp;
         val e : m = mon.e;
         fun pow (x : m, n : int) : m =
           if n = 0 then e
                    else comp(pow(x,n-1),x);
  end;
```

Nun wollen wir den Funktor `F` konkret anwenden. Dazu nehmen wir die folgende Struktur `RealAdd` passend zur Signatur `MONOID`:

```
structure RealAdd =
  struct type m = real;
         fun comp (r : real, s : real) : real = r+s;
         val e : real = 0.0;
  end
```

Diese ML-Struktur entspricht dem Monoid $(\mathbb{R}, +, 0)$. Der Funktor `F` ist anwendbar und liefert eine ML-Struktur, die der algebraischen Struktur $(\mathbb{R}, +, 0, pow)$ entspricht, mit $pow(x, 0) = 0$, $pow(x, n+1) = pow(x, n) + x$ für alle $n \in \mathbb{N}$ und $pow(x, n) = \text{undef}$ sonst. Insbesondere gilt also $pow(x, n) = n * x$ für $n \in \mathbb{N}$. Formal haben wir somit: Deklariert man durch die Funktoranwendung

```
structure RealMult : POWER = F(RealAdd);
```

eine Struktur `RealMult` zur Signatur `POWER`, so entspricht die ML-Struktur `RealMult` der eben betrachteten algebraischen Struktur $(\mathbb{R}, +, 0, pow)$. Etwa wertet sich der Ausdruck `RealMult.pow(2.0,7)` im SML-System zu `14.0` mit der Typangabe `RealMult.m`, also `real`, aus. ■

Bei der Verwendung von ML zur Programmierung von großen Programmsystemen zeigt sich oft, daß dabei die eigentlich zu bewältigende Aufgabe der Entwurf von geeigneten Funktoren ist, und nicht der von einzelnen Strukturen.

A.3 Das Modulkonzept von Modula-2

Das hauptsächliche Ziel bei der Weiterentwicklung von Pascal zu Modula-2 war es, ein standardisiertes Modulkonzept zu konzipieren, welches einerseits Datenabstraktion mit möglichst unabhängigen Modulen erlaubt, durch die Verwendung von vorimplementierten Modulen andererseits auch ermöglicht, die Sprache klein und überschaubar zu halten. Das Modulkonzept ist uns in den Modula-2-Teilen des Skriptums schon ausschnittsweise in der Gestalt des Importteils am Anfang eines jeden Programms begegnet. Nun soll etwas genauer erklärt werden, was hinter ihm steckt.

Vom syntaktischen Standpunkt aus betrachtet, gibt es in Modula-2 drei Kategorien von Modulen, die für Datenabstraktion von Bedeutung[35] sind. Diese wollen wir nun der Reihe nach vorstellen.

A.3.1 Programm-Module

Ein Programm-Modul entspricht dem Hauptprogramm eines in einzelne Module aufgeteilten Modula-2-Programmsystems. Insbesondere gibt es in den Programm-Modulen keine Exportation. Die Syntax eines Programm-Moduls mit Namen P ist:

`MODULE` P;
$\gg$ *Importlisten* $\ll$;
$\gg$ *Deklarationen wie vorgestellt* $\ll$;
`BEGIN`
$\gg$ *Anweisungen wie vorgestellt* $\ll$;
`END` P .

Wir haben es bei einem Programm-Modul somit insbesondere mit einem Modula-2-Programm im bisherigen Sinne des Skriptums zu tun. ■

Ein Programm-Modul wird von einem Modula-2-Übersetzer als Übersetzungseinheit akzeptiert.

A.3.2 Externe programmierte Module

Externe programmierte Module sind diejenigen Module, welche im Laufe der Programmierung eines in Module unterteilten Programmsystems erstellt und von anderen Modulen in Anspruch genommen werden. Um eine Schnittstellenüberprüfung zu erreichen, wird so ein Modul M in zwei Teile gegliedert:

(1) Der erste Teil ist der **Definitions-Modul** von M. Hier wird im Prinzip die Signatur (d.h. die Schnittstelle) von M beschrieben. Es werden, nach den Listen der

[35] Die vierte Kategorie von Modulen in Modula-2 sind die sogenannten lokalen oder internen Module, welche auch ihre Berechtigung haben, bezüglich Datenabstraktion aber nicht relevant sind.

importierten Konstituenten (welche auch fehlen können), genau diejenigen symbolischen Konstanten, Typen, Programmvariablen und sogenannten Kopfzeilen der Rechenvorschriften und Prozeduren aufgeführt, die vom Modul exportiert werden. Die Syntax des Definitions-Modul ist schematisch nachstehend gegeben:

`DEFINITION MODULE` M;
≫ *Imporlisten* ≪;
≫ *Angabe der Konstanten, Typen, Programmvariablen, Kopfzeilen* ≪;
`END` M .

Man beachte, daß bei den symbolischen Konstanten und Typen im Definitions-Modul die vollständigen Deklarationen vorhanden sein können, im Gegensatz zu den Rechenvorschriften und Prozeduren. Dies widerspricht aber dann dem Prinzip der Datenkapselung und sollte deshalb wenn möglich unterlassen werden. Genaueres zu dieser Thematik folgt noch nach dem nächsten Beispiel.

(2) Der zweite Teil ist der **Implementierungs-Modul** von M. Hier werden die Konstanten und Typen deklariert, falls dies nicht schon im Definitions-Modul geschehen ist. Weiterhin werden zu den Kopfzeilen der Rechenvorschriften und Prozeduren des Definitions-Moduls von M nun die vollständigen Deklarationen formuliert. Schließlich können auch noch lokale, nicht sichtbare Hilfsprogrammvariablen, -typen usw. verwendet werden. Die Syntax des Implementierungs-Modul ist gleich der eines Programm-Moduls, sieht also schematisch wie folgt aus:

`IMPLEMENTATION MODULE` M;
≫ *Importlisten* ≪;
≫ *Deklarationen wie vorgestellt* ≪;
`BEGIN`
≫ *Anweisungen wie vorgestellt* ≪;
`END` M .

In vielen Fällen ist der Anweisungsteil leer; bei der Verwendung von lokalen Programmvariablen erfolgt hier normalerweise nur eine Initialisierung. Bei einem leeren Anweisungsteil ist es erlaubt, das Wort `BEGIN` wegzulassen. Man hat dann

`IMPLEMENTATION MODULE` M;
≫ *Importlisten* ≪;
≫ *Deklarationen wie vorgestellt* ≪;
`END` M .

als syntaktische Form.

Wegen dieser Aufteilung kann die Schnittstellenüberprüfung ohne Kenntniss der Implementierung erfolgen. ■

Sowohl der Definitions-Modul als auch der Implementierungs-Modul eines programmierten externen Moduls werden von einem Modula-2-Übersetzer ebenfalls als Übersetzungseinheit akzeptiert. Beide sind die imperativen Gegenstücke zu den funktionalen ML-Sprachmitteln Signatur und Struktur. Im Gegensatz zu ML erlaubt Modula-2 jedoch keine generischen programmierten Module, was ihre Wiederverwendbarkeit doch sehr einschränkt.

A.3.3 Beispiel (Paare von natürlichen Zahlen)

Wir betrachten die Implementierung von Paaren von natürlichen Zahlen in Modula-2, wie wir sie in Beispiel 7.5.5 für die Berechnung der Fibonacci-Zahlen benötigten. Bezeichnen wir den entsprechenden externen programmierten Modul mit `CardPair`, so ergeben sich aus dem Programm von Beispiel 7.5.5 sofort

```
DEFINITION MODULE CardPair;
TYPE cardpair;
  (* Paare natuerlicher Zahlen *)
PROCEDURE make (x, y : CARDINAL) : cardpair;
  (* Zusammenfassung zu Paaren *)
PROCEDURE p1 (p : cardpair) : CARDINAL;
  (* Erste Komponente eines Paars *)
PROCEDURE p2 (p : cardpair) : CARDINAL;
  (* Zweite Komponente eines Paars *)
END CardPair .
```

als der Definitions-Modul von `CardPair` und

```
IMPLEMENTATION MODULE CardPair;
FROM Storage IMPORT ALLOCATE;
TYPE cardpair = POINTER TO pairrec;
     pairrec  = RECORD first, second : CARDINAL END;
PROCEDURE make (x, y : CARDINAL) : cardpair;
  VAR p : cardpair;
  BEGIN
  NEW(p);
  p^.first := x;
  p^.second := y;
  RETURN p
  END make;
PROCEDURE p1 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.first END p1;
PROCEDURE p2 (p : cardpair) : CARDINAL;
  BEGIN RETURN p^.second END p2;
END CardPair .
```

als sein Implementierungs-Modul. Man beachte, daß der Anweisungsteil dieses Implementierungs-Moduls leer ist. Im Implementierungs-Modul des programmierten externen Moduls `CardPair` stellt der Verbundtyp `pairrec` eine verborgene Konstituente dar. ■

Typen wie `cardpair`, bei denen im Definitions-Modul eines programmierten externen Moduls nur der Name und nicht die gesamte Deklaration aufgeführt ist, heißen im Sprachgebrauch von Modula-2 auch private, undurchsichtige oder **opake Typen**. Aus Effizienzgründen gilt nun für Modula-2, daß **ein opaker Typ ein Zeigertyp sein muß**. Sichtbare Typen in programmierten externen Modulen, die nicht Zeigertypen sind, müssen im entsprechenden Definitions-Modul vollständig deklariert sein; man erhält also Einsicht in ihren inneren Aufbau.

Leider ist es in der Programmiersprache Modula-2, ebenfalls aus Effizienzgründen, nicht möglich, opake Typen transitiv zu verwenden. Wird also ein Typ m_1 in einem programmierten externen Modul M_1 als opak festgelegt, so ist es nicht erlaubt, in einem anderen programmierten externen Modul M_2 einen opaken Typ m_2 zu definieren, der sich auf m_1 stützt. In der Praxis heißt dies, daß bei großen und hierarchisch komplexer gestalteten Systemen in der Regel in den Definitions-Modulen die vollständigen Typdeklarationen stehen, das Prinzip der Kapselung also verletzt ist.

Nach diesen Bemerkungen zur Sichtbarkeit von Typen kommen wir nun zur letzten Art von Modulen in der Programmiersprache Modula-2.

A.3.4 Bibliotheks-Module

Dies sind Module, die vorübersetzt vorliegen und in anderen Modulen beliebig oft verwendet werden können. Drei solche Module haben wir im Laufe des Skriptums schon kennengelernt, nämlich `InOut` zur Ein- und Ausgabe, `MathLib` für die Funktionen der Analysis und `Storage`, wodurch vorimplementierte Prozeduren für die Speicherverwaltung zur Verfügung gestellt werden. Neben diesen Modulen gibt es in Modula-2 noch eine Reihe weiterer Bibliotheks-Module, etwa `SysLib` für Systemprogrammierung und Parallelität. ■

Man beachte, daß die Bibliotheks-Module von Modula-2 system- und implementierungsabhängig sind. Man spricht in diesem Zusammenhang auch von der **Bibliothek eines Modula-2-Systems**. Solche Bibliotheken erlauben auch das Verbergen von maschinenabhängigen Implementierungsdetails, beispielsweise der Eingabe und Ausgabe oder des Dateisystems, und erhöhen dadurch die Portabilität der Programme. Genaueres zu den Bibliotheks-Modulen des von uns verwendeten MOCKA-Systems findet man im Benutzer-Manual, das normalerweise mit dem System ausgeliefert wird.

Die logische Verknüpfung von Modulen erfolgt durch das Importieren der Bezeichner aus den Schnittstellen mittels Importlisten. Darauf wollen wir nun etwas genauer eingehen. Wir beschreiben jedoch nur die grundlegendsten Dinge und gehen, wie bei ML, auf Sichtbarkeitsregeln usw. nicht genauer ein.

A.3.5 Importation und Exportation in Modula-2

Gegeben sei ein Modula-2-Programmsystem, mit einem Programm-Modul P und zwei programmierten externen Modulen M und N, letztere jeweils bestehend aus dem Definitions-Modul und dem Implementierungs-Modul. Schematisch liegt also die folgende Situation vor:

```
MODULE P;
   ...
BEGIN ... END P.

DEFINITION MODULE M;
   ...
END M.
IMPLEMENTATION MODULE M;
   ...
BEGIN ... END M.
DEFINITION MODULE N;
   ...
END N.
IMPLEMENTATION MODULE N;
   ...
BEGIN ... END N.
```

Weiterhin seien k_1 und k_2 die sichtbaren Konstituenten von M, also aufgeführt im Definitions-Modul und deklariert im Implementierungs-Modul, und k_1 und k_2' diejenigen von N.

Falls die Importliste des Hauptprogramms P aus `FROM` M `IMPORT` k_1, k_2; besteht, so können die sichtbaren Konstitienten von M in P in der bisher üblichen Weise verwendet werden. Hingegen ist bei der Importliste `FROM` M `IMPORT` k_1; nur die Verwendung von k_1 in P in der bisher üblichen Weise zulässig. Allgemein darf man bei einer Importation in einem Modul M_1 der Art

`FROM` M_2 `IMPORT` $k_1, k_2, \ldots, k_n$;

jede sichtbare Konstituente von M_2, die in der Liste nach dem Wort `IMPORT` vorkommt, nur unter der dort angegebenen Bezeichnung in M_1 verwenden. Man nennt dies **normalen Import**.

Schwierigkeiten ergeben sich im obigen Beispielfall jedoch, wenn man sowohl die Konstituente k_1 aus dem Modul M als auch aus dem Modul N in P verwenden will. Nicht zulässig ist `FROM` M `IMPORT` k_1; `FROM` N `IMPORT` k_1; weil nach der ersten Import-Liste der Name k_1 schon für die Konstituente aus M vergeben ist. Abhilfe bietet hier der sogenannte **qualifizierte Import** mittels `IMPORT` M; `IMPORT` N; an. Diese Art von Importation erlaubt nämlich die Verwendung der Konstituenten nur in den qualifizierten Formen $M.k_1$, $M.k_2$, $N.k_1$ und $N.k_2'$. Allgemein darf man bei einer qualifizierten Importation in einem Modul M_1 der Art

`IMPORT` M_2;

jede sichtbare Konstituente k von M_2 nur unter der qualifizierten Bezeichnung $M_2.k$ in M_1 verwenden. Qualifizierter Import erlaubt die Unterscheidung von Programmteilen, die oft gleich bezeichnet werden, wie beispielsweise das Einfügen in Datenstrukturen. ■

Zum Abschluß dieses Anhangs geben wir nun noch ein Beispiel für die Benutzung von Modulen in der Programmiersprache Modula-2 an.

A.3.6 Beispiel (Berechnung von $\mathbf{R}^*$ in Modula-2)

Wir greifen die Berechnung der transitiven Hülle mittels Fixpunktbildungen von Abschnitt 8.6.7 noch einmal auf und präsentieren nachfolgend ein Modula-2-Programm für eine abgewandelte Form des effizienteren Fixpunkt-Algorithmus. Bei diesem Verfahren wird die reflexiv-transitive Hülle R^* als Grenzwert der Kette $S_0 \subseteq S_1 \subseteq \ldots$ von Relationen berechnet, die durch

$$(1)\;\; S_0 := I \cup R \qquad\qquad (2)\;\; S_{i+1} := S_i S_i$$

definiert ist. Wie man leicht durch eine Induktion nach $n \in \mathbb{N}$ zeigt, gilt nämlich die Gleichung $S_n = \bigcup_{i=0}^{2^n} R^i$, folglich also

$$R^* = \bigcup_{n \geq 0} R^n = \bigcup_{n \geq 0} \bigcup_{i=0}^{2^n} R^i = \bigcup_{n \geq 0} S_n.$$

Für das weitere setzen wir einen programmierten externen Modul `Relationen` voraus, der homogene Relationen auf der Menge der natürlichen Zahlen von 1 bis n darstellt, wobei n kleiner als eine vorgegebene Konstante N ist.

Der Definitions-Modul von `Relationen` ist, mit entsprechenden Kommentaren versehen, nachstehend angegeben:

```
DEFINITION MODULE Relationen;
TYPE rel;
  (* Typ der homogenen Relationen *)
PROCEDURE N : CARDINAL;
  (* Liefert maximale Kardinalitaet der Grundmenge *)
PROCEDURE readrel (VAR r : rel; n : CARDINAL (* n <= N *));
  (* Liest eine Relation ueber 1,...,n ein *)
PROCEDURE writerel (VAR r : rel; n : CARDINAL (* n <= N *));
  (* Druckt eine Relation ueber 1,...,n *)
PROCEDURE id (VAR r : rel; n : CARDINAL (* n <= N *));
  (* Speichert in r identische Relation ueber 1,...,n *)
PROCEDURE union (VAR r, s : rel; n : CARDINAL (* n <= N *));
  (* Speichert in r die Vereinigung von r, s ueber 1,...,n *)
PROCEDURE quad (VAR r, s : rel; n : CARDINAL (* n <= N *));
  (* Speichert in r das Quadrat von s ueber 1,...,n *)
PROCEDURE eq (VAR r, s : rel; n : CARDINAL (* n <= N *)) : BOOLEAN;
  (* Testet r, s ueber 1,...,n auf Gleichheit *)
END Relationen.
```

Die sichtbaren Konstituenten des Moduls `Relationen` sind also der Typ `rel` der Relationen, eine parameterlose Rechenvorschrift `N` für die vorgegebene Konstante N, zwei Prozeduren `readrel` und `writerel` zum Einlesen und Drucken von Relationen, eine Prozedur `id`, die eine identische Relation erzeugt, zwei Prozeduren `union` und `quad` zum Vereinigen und Quadrieren von Relationen und eine Rechenvorschrift `eq`, die zwei Relationen auf Gleichheit testet. Dabei besitzt jede Prozedur bzw. Rechenvorschrift noch einen Parameter `n` der Sorte `CARDINAL`, der die Grundmenge der relationalen Argumente in Form ihres größten Elements angibt.

Auf Prozeduren für die weiteren Konstanten und Operationen auf Relationen verzichten wir, da diese zur Berechnung der durch (1) und (2) definierten Kette $S_0 \subseteq S_1 \subseteq \ldots$ nicht verwendet werden. Auf die Angabe des Implementierungs-Moduls verzichten wir ebenfalls, empfehlen dem Leser aber, diesen auszuprogrammieren.

Aufbauend auf den programmierten externen Modul `Relationen` können wir nun, ohne die Implementierung der Relationen zu kennen, den folgenden Programm-Modul formulieren:

```
MODULE Closure;
FROM InOut IMPORT ReadCard;
FROM Relationen IMPORT rel, readrel, writerel, id, union quad;
VAR r, s, t : rel;
    n : CARDINAL;
BEGIN
ReadCard(n);
readrel(r,n);
id(s,n);
union(s,r,n);
quad(t,s,n);
WHILE NOT(eq(s,t,n)) DO
  s := t; quad(t,s,n) END;
writerel(s,n)
END Closure .
```

In diesem Modul wird die Kette $S_0 \subseteq S_1 \subseteq \ldots$ von Relationen bis zu dem Index i_0 berechnet, wo sie stationär wird. Wegen $S_{i_0} = \bigcup_{n \geq 0} S_n = R^*$ bekommen wir auf diese Weise die reflexiv-transitive Hülle der Eingabe. ∎