9 Präzisierung der Daten

Um ein Problem zu lösen, sind die Gegenstände der realen Welt so durch geeignete Strukturen zu modellieren, daß sie algorithmisch möglichst einfach und effizient zu verarbeiten sind. Schon beim Verteilungsproblem hatten wir gesehen, daß es nicht immer leicht ist, zu realen Objekten geeignete Modellierungen zu finden. So war erst ein längerer wiederholter Analyse- und Erprobungsprozeß nötig, um die Merkmale der wartenden Besuchergruppen (Kinderzahl/von auswärts?/Ankunftszeit) in eine geeignete Datenstruktur abzubilden (die Codenummer), die zunächst vom Verteiler M und später auch vom Computer C einfach zu handhaben war. In diesem Abschnitt wollen wir uns mit den allgemeinen Prinzipien befassen, um Gegenstände durch Datentypen zu modellieren, die maschinell verarbeitet werden können.

9.1 Grundbegriffe

Die Welt, die durch ein Informatiksystem modelliert werden soll, besteht gewöhnlich aus sehr vielen Daten in zahlreichen verschiedenen Formen und Darstellungen. Mit diesen Daten operiert das Informatiksystem, bis eine Lösung gefunden ist. Man könnte zwar prinzipiell den "Datenbrei" als solches hinnehmen – und einige Programmiersprachen tun das auch –, tatsächlich kann man den Daten aber relativ leicht eine Struktur aufprägen. Das Strukturierungskriterium sind die Operationen, die mit den Daten möglich sind. Jeweils alle Datenobjekte, auf denen die gleiche Menge von Operationen möglich ist und die sich strukturell gleich verhalten, faßt man zu einem *Datentyp* zusammen. So sichert man ab, daß alle Werte eines Datentyps das gleiche operationale Verhalten aufweisen. Typisierung erleichtert die Manipulation von Daten erheblich und macht Programme sicherer und nachvollziehbarer. Zugleich werden durch die Typisierung gerade solche Informationen über die Eigenschaften von Daten und ihren Operationen erfaßt, die sich durch einen Übersetzer automatisch überprüfen lassen. Man definiert daher:

Definition A:
Ein **Datentyp**, kurz **Typ**, D ist ein Paar D=(W,R) bestehend aus einer Wertemenge W und einer Menge R von Operationen, die auf W definiert sind.
Ein **elementarer** Datentyp ist ein Datentyp, dessen Wertemenge nicht weiter in Datentypen zerlegt werden kann.

Bemerkung: Wie bereits früher erläutert, hängt es vom persönlichen Geschmack oder vom Sprachniveau, vor allem der Programmiersprache, ab, wann man einen Datentyp als elementar betrachtet.

Notationen: Sei neben IN, Z und IR

A={'a',...'z','A',...,'Z','0',...,'9',',',':','◊'...} die Menge der Zeichen (◊=Leerzeichen),

IB={true,false} die Menge der Wahrheitswerte.

Beispiele: Im Vorgriff auf die weiteren Abschnitte dieses Kapitels sind im folgenden die üblichen elementaren Datentypen aufgelistet. Wir kürzen sie in Zukunft durch die rechts stehenden Bezeichner ab:

(A,{...}) =: char
(A^*,{•,...}) =: text
(IN,{+,-,*,div,mod,=,≠,>,...}) =: nat
(Z,{+,-,*,div,mod,=,≠,...}) =: int
(IR,{+,-,*,/,=,≠,...}) =: real
(IB,{and,or,not}) =: bool

Notation: Häufig unterscheiden wir nicht zwischen einem Datentyp D=(W,R) und seiner Wertemenge W, z.B. steht int auch schon mal für Z. Und statt $x \in W$ schreiben wir auch $x \in D$.

In der Informatik hat sich zur Bildung von Datentypen und -strukturen ein Baukastenprinzip (ein Beispiel für die fundamentale Idee der Orthogonalisierung) bewährt. In diesen Baukästen gibt es einige wenige Grundbausteine, die *elementaren Datentypen*, ferner einige wenige Kombinationsregeln, die *Konstruktoren*, um die Grundbausteine zu verknüpfen und damit neue Bausteine zu schaffen, die ihrerseits wieder mittels der Regeln zu noch komplexeren Bausteinen zusammengesetzt werden können (LEGO-Prinzip: wenige Sorten von Steinen, wie Achter, Vierer, Zweier, Einer; eine Kombinationsregel "aufeinanderstecken"; und schon kann man beliebige Objekte bauen). Und so sieht ein Baukasten aus (Abb. 1): Sei Δ die Klasse aller Datentypen und Δ_e die Klasse der elementaren Datentypen. Für i=1,...,n sei

$K_i: \Delta \times ... \times \Delta \rightarrow \Delta$

ein Konstruktor, der eine gewisse Anzahl von Datentypen als Argument benötigt und einen neuen Datentyp erzeugt. Dann ist für $K=\{K_1,...,K_n\}$

$B=(\Delta_e,K)$

ein *Datentyp-Baukasten*.

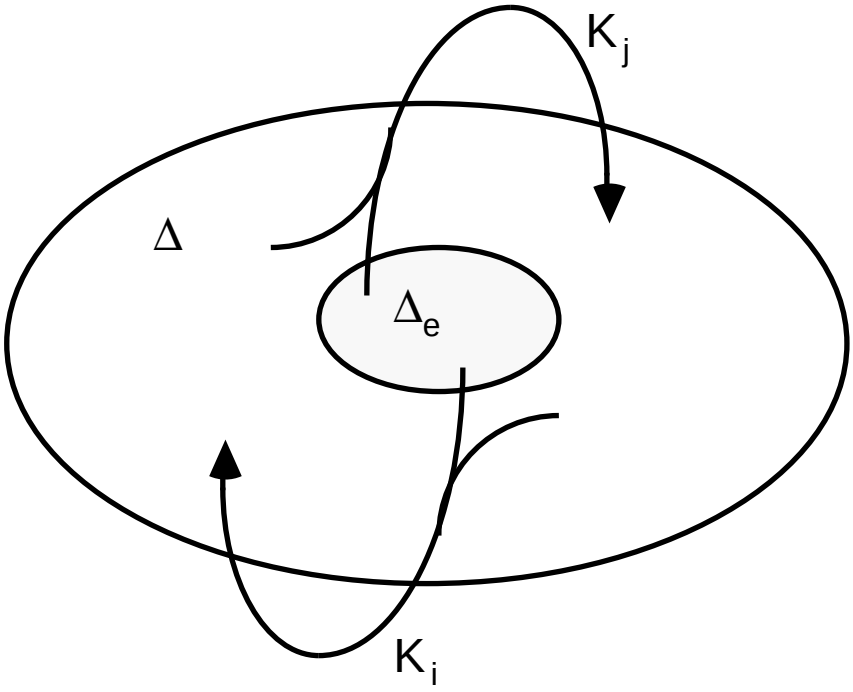


Abb. 1: Baukastenprinzip zur Definition von Datentypen

In der Informatik hat sich im Laufe der Zeit ein *Standardbaukasten* $B_0=(\Delta_0,K_0)$ etabliert, der so oder ähnlich in praktisch allen Programmiersprachen integriert ist und in dem es sechs elementare Datentypen und sieben Konstruktoren gibt:

Δ_0={nat,int,real,char,text,bool},

K_0={Enumeration, Restriktion, Aggregation, Generalisation, Rekursion, Potenzmengenbildung, Bildung von Funktionenräumen}.

Dabei bedeuten

- *Enumeration* die Bildung eines elementaren Datentyps durch Aufzählung aller seiner Elemente,
- *Restriktion* die Bildung einer Teilmenge eines Datentyps,
- *Aggregation* das gleichrangige Zusammensetzen mehrerer Datentypen zu einem einzigen (kartesisches Produkt),
- *Generalisation* die Vereinigung von disjunkten Datentypen zu einem einzigen,
- *Rekursion* den Übergang zu einem Datentyp mit abzählbar unendlich vielen Elementen, die gleichartig aus einfacheren Elementen desselben Typs aufgebaut sind,
- *Potenzmengenbildung* die Bildung der Menge aller (endlichen) Teilmengen eines Datentyps,

- *Bildung von Funktionenräumen* den Übergang von zwei Datentypen zum Datentyp, dessen Wertemenge die Menge aller Funktionen zwischen diesen Datentypen ist.

Δ_0 und K_0 werden wir unten detailliert besprechen.

Bei der Definition eines Datentyps mittels eines Konstruktors werden implizit gewisse Funktionen mitdefiniert, die den Konstruktionsprozeß wieder rückgängig machen und es ermöglichen, einen in einer Datenstruktur versteckten elementaren Wert "wieder aufzuspüren". Solche Funktionen nennt man *Destruktoren* oder besser *Selektoren*. Um im LEGO-Bild zu bleiben: Die Destruktoren/Selektoren sorgen dafür, daß man auf die einzelnen Bausteine eines LEGO-Objekts zugreifen kann, z.B. um sich ihre Farbe anzusehen, sie auszuwechseln oder zu verstärken.
Beispiel: In PRO gibt es den Konstruktor [ ], der Zahlen zu Zahlenfolgen zusammensetzt, z.B. die Zahlen 2, 7, 1 zur Folge [2,7,1]. Mittels der Operationen erstes und rest können einzelne Elemente einer Zahlenfolge wieder gewonnen werden, z.B.

7 = erstes(rest ([2,7,1])).

erstes und rest sind also Selektoren für den Konstruktor "Folgenbildung".

9.2 Elementare Datentypen

In nahezu allen Programmiersprachen sind folgende sechs elementare Datentypen vordefiniert: int, nat, real, char, bool und text. Da sie vom Programmierer ohne besondere Vorbereitung im Programm verwendet werden können, werden sie auch als *Standarddatentypen* bezeichnet.

Die Wertemengen dieser Datentypen sind üblicherweise endlich und zwischen den Werten besteht eine lineare Ordnung <. Ist also $M=\{m_1,m_2,m_3,...,m_n\}$ die Wertemenge eines dieser Datentypen, so gilt

$m_1<m_2<m_3<...<m_n$.

Datentypen, deren Wertemenge wie hier linear wie auf einer Skala angeordnet ist, nennt man auch *skalare Datentypen*.

Zu jedem Datentyp gehört (lt. Definition A) eine Reihe von Operationen, die man analog als *Standardoperationen* oder *Standardfunktionen* bezeichnet, wenn sie vordefiniert sind. Von einer *Operation* f spricht man oft dann, wenn man sie in *Infixnotation*, also z.B. als zweistellige Funktion in der Form xfy (z.B. x+y), verwenden darf. Von einer *Funktion* f spricht man, wenn man f nur in Funktionsschreibweise oder *Präfixnotation*, also z.B. als zweistellige Funktion in der Form f(x,y) (z.B. sin(x) oder erstes(leerung)), verwendet. Eine dritte weniger häufige Notation ist die *Postfixnotation*, bei der der Funktionsbezeichner

den Argumenten nachgestellt wird, also z.B. als zweistellige Funktion in der Form xyf (z.B. x!, Fakultät).

Zwei Standardfunktionen sind für alle skalaren Datentypen, mit Ausnahme des Datentyps real, vordefiniert, die Funktionen pred und succ. Sie besitzen jeweils einen Parameter und liefern als Ergebnis den bezüglich der linearen Ordnung unmittelbaren Vorgänger (engl. *pred*ecessor) bzw. Nachfolger (engl. *succ*essor). Formal: Sei $M=\{m_1,m_2,m_3,...,m_n\}$ die Wertemenge eines skalaren Datentyps (außer real). Dann sind pred: $M \to M$ und succ: $M \to M$ *partielle* Abbildungen mit

$$\text{pred}(m_i)=\begin{cases} m_{i-1}, & \text{falls } i\geq 2,\\ \perp, & \text{falls } i=1.\end{cases}$$

$$\text{succ}(m_i)=\begin{cases} m_{i+1}, & \text{falls } i<n,\\ \perp, & \text{falls } i=n.\end{cases}$$

Eine weitere Standardfunktion für alle skalaren Datentypen (mit Ausnahme von real, int und nat) ist die Funktion ord, die jedes Element des Datentyps auf die natürliche Zahl abbildet, die die Position des Elements innerhalb der Ordnung angibt. Präziser: Sei $M=\{m_1,m_2,m_3,...,m_n\}$ die Wertemenge eines skalaren Datentyps (außer real und int). Dann ist ord eine Abbildung der Form

ord: $M \to IN$ mit

$\text{ord}(m_i)=i$.

Weiterhin sind auf allen skalaren Datentypen Vergleichsoperationen definiert (siehe unten unter bool).

Nun zu den elementaren Datentypen im einzelnen.

Der Datentyp bool.

Die Wertemenge des Datentyps bool enthält zwei Elemente, die mit false (dt. falsch) und true (dt. wahr) bezeichnet werden. Die lineare Ordnung lautet

false < true.

Es gilt also z.B.

pred(true)=false, succ(false)=true, ord(false)=1 und ord(true)=2.

false und true heißen auch Boolesche Werte oder Wahrheitswerte.

Der Datentyp bool sieht zwei 2-stellige Operationen and und or, sowie eine 1-stellige Operation not vor. Die Operationen sind durch folgende Wertetabelle festgelegt:

x	y	x and y	x or y	not x
false	false	false	false	true
false	true	false	true	true
true	false	false	true	false
true	true	true	true	false

x and y liefert den Wert true dann und nur dann, wenn x *und* y den Wert true besitzen.
x or y liefert den Wert true dann und nur dann, wenn x *oder* y den Wert true besitzen.
not x liefert den Wert true dann und nur dann, wenn x *nicht* den Wert true besitzt.

Für alle skalaren Datentypen existieren sechs Vergleichsoperationen, deren Bildmenge der Datentyp bool ist. Sie lauten:

$<, \leq, \geq, >, =, \neq$.

Eine Vergleichsoperation liefert den Wahrheitswert false, falls der Vergleich falsch ist, und sie liefert den Wert true, falls der Vergleich zutrifft.

Beispiel: Der Vergleich

12<>3 liefert den Wahrheitswert true,
12<>12 liefert den Wahrheitswert false,
3<=2 liefert den Wahrheitswert false,
false<true liefert den Wahrheitswert true,
true<false liefert den Wahrheitswert false.

Der Datentyp int.

Der Datentyp int entspricht im wesentlichen unserem in PRO verwendeten Typ Zahl. Die Wertemenge M von int ist eine *endliche* Teilmenge der ganzen Zahlen Z in Form eines symmetrischen Intervalls um den Nullpunkt, d.h.

$M=\{x \in Z \mid -maxint \leq x \leq maxint\}$.

Der Wert von maxint hängt vom Computer und von der Programmiersprache ab. Auf Kleincomputern besitzt maxint meist den Wert $2^{15}-1=32767$, auf Großcomputern $2^{31}-1=2.147.483.647$.

Konstanten vom Typ int notiert man in der üblichen Dezimaldarstellung als Folge von Ziffern mit oder ohne Vorzeichen + oder -.

Der Datentyp int verfügt über folgende 2-stellige *Standardoperationen:*

+ (Addition), - (Subtraktion), * (Multiplikation), div (ganzzahlige Division ohne Rest), mod (Restbildung bei ganzzahliger Division).

Addition, Subtraktion und Multiplikation sind in der üblichen Weise definiert, solange man den Wertebereich von -maxint bis +maxint nicht verläßt.

Beispiele: Für mod und div gilt:

7 div 2=3, 7 mod 2=1, 14 div 3=4, 14 mod 3=2,

-14 div 3=-4, -14 mod 3=-2.

Man beachte, daß die arithmetischen Gesetze beim Datentyp int nicht uneingeschränkt gelten, da int ja nur eine *Teilmenge* der ganzen Zahlen repräsentiert.

Beispiel: Angenommen, maxint=100. Die folgende Rechnung zeigt, daß Rechenergebnisse von der Reihenfolge abhängen, in der Ausdrücke ausgewertet werden. So kommt es bei der Rechnung

(60 + 50) - 30

zu einem *Überlauf* (engl. overflow), da die Addition 60+50 aus dem zur Verfügung stehenden Zahlenbereich herausführt. Die Rechnung

60 + (50 - 30)=80

kann problemlos durchgeführt werden. Das Assoziativgesetz der Addition gilt im Bereich int also nicht mehr.

Außer obigen Standardoperationen gibt es in int noch folgende einstellige Standardfunktionen:

abs(x) (Absolutbetrag), sqr(x) (Quadrat) und

odd(x) (Feststellung der Ungeradzahligkeit).

Der Datentyp nat.

Auf diesen Datentyp übertragen sich alle Eigenschaften von int sinngemäß.

Der Datentyp real.

Der Wertebereich des Datentyps real sind grundsätzlich alle reellen Zahlen. Da ein Computer jedoch ein beschränktes Gerät ist, das keine unendlichen Dezimalzahlen, wie z.B. π=3,1415926..., darstellen kann, wird die Wertemenge auf Zahlen in sog. **Gleitpunktdarstellung** beschränkt. Die Gleitpunktdarstellung ist eine Methode zur *näherungsweisen* Darstellung von reellen Zahlen in halblogarithmischer Form. Sei $z \in IR$ eine positive reelle Zahl. z kann man stets in der Form

$z = m \cdot b^e$ mit $b \in IN$, $e \in Z$, $b \geq 2$ und $^1/_b \leq m < 1$.

schreiben. m heißt **Mantisse** und gibt den Zahlenwert (also die Folge der gültigen Ziffern) von z an, e ist der **Exponent** und charakterisiert die Größenordnung der Zahl. b heißt **Basis**. In der Praxis wählt man für b meist eine der Zahlen 2, 10 oder 16. Durch die Bedingung $^1/_b \le m < 1$ (**Normalisierung**) wird die Darstellung eindeutig. Für z=0 setze man m=0, und für negative reelle Zahlen verwende man das Minuszeichen. Somit kann man alle reellen Zahlen auf diese Weise darstellen

Beispiele: Normalisierte Gleitpunktdarstellungen zur Basis 10:

$189{,}217 = 0.189217 \cdot 10^3$,

$-2{,}4 = -0.24 \cdot 10^1$,

$-0{,}0013 = -0.13 \cdot 10^{-2}$.

Wegen der Endlichkeit von Computern steht sowohl für die Mantisse als auch für den Exponenten nur eine endliche Anzahl von Stellen zur Verfügung. Daraus folgt, daß

a) nur eine endliche Menge von reellen Zahlen durch den Datentyp real darstellbar ist,
b) jede darstellbare reelle Zahl "nur" eine rationale Zahl, sogar eine endliche Dezimalzahl (falls b=10) ist.

Die Endlichkeit der Darstellung hat einige merkwürdige Effekte: Die gewohnten Rechenregeln gelten nicht mehr. Alles wird nur noch näherungsweise berechnet, und im Laufe von Rechnungen können sich kleine Fehler zu total falschen Ergebnissen aufsummieren.

Beispiel: Angenommen, es stehen für die Mantisse 4 und für den Exponenten 2 Stellen zur Verfügung; die Basis b sei 10. Wir berechnen 2000+0.7-2000 auf zwei verschiedene Arten:

a) $(2000+0.7)-2000 = (0.2000 \cdot 10^4 + 0.7000 \cdot 10^0) - 0.2000 \cdot 10^4$
$= (0.2000 \cdot 10^4 + 0.0000 \cdot 10^4) - 0.2000 \cdot 10^4 = 0$

b) $(2000-2000)+0.7 = (0.2000 \cdot 10^4 - 0.2000 \cdot 10^4) + 0.7000 \cdot 10^0$
$= 0.0000 \cdot 10^0 + 0.7000 \cdot 10^0 = 0.7000 \cdot 10^0 = 0.7$.

Die Addition und Subtraktion zweier real-Zahlen kann nur erfolgen, wenn beide den gleichen Exponenten haben. Daher muß zuvor eine Angleichung an den größeren Exponenten erfolgen. Die Zahl 0.7 wird hierbei in Fall a) ausgelöscht. Wenn anschließend das Ergebnis mit 1000 multipliziert wird, so beträgt der Unterschied zwischen den Rechnungen bereits den Wert 700 usw. Dieser Effekt entsteht durch *Auslöschung* kleinerer Zahlen und wird als **Rundungsfehler** bezeichnet.

Die Verwendung von real-Zahlen ist also nicht unproblematisch. Speziell bei der Programmierung von numerischen Problemen (z.B. Näherungsverfahren) muß vorher ge-

nau analysiert werden, wie die Endergebnisse durch Rechenungenauigkeiten verfälscht werden können.

Konstanten vom Typ real schreibt man in der üblichen Dezimaldarstellung als Folge von Ziffern mit oder ohne Dezimalpunkt, gefolgt vom Exponententeil, der durch den Buchstaben E eingeleitet wird.
Beispiele: Konstanten vom Typ real sind:

-1, 2.5, 3.7E4, -0.2E-2, 0.0072E+2.

In unserer geläufigen Darstellung würden wir für die drei letzten Zahlen schreiben

$3.7 \cdot 10^4$, $-0.2 \cdot 10^{-2}$, $0.0072 \cdot 10^2$ bzw.

37000, -0.002, 0.72.

Der Datentyp real verfügt neben den bekannten arithmetischen Operationen +, -, * und / (Division) und den Vergleichsoperationen (siehe unter bool) im allgemeinen noch über folgende Standardfunktionen:

abs(x)	(Absolutbetrag, analog wie bei int definiert)
sqr(x)	(Quadrat, analog wie bei int definiert)
sin(x)	(Sinus)
cos(x)	(Cosinus)
arctan(x)	(arcustangens)
exp(x)	(Exponentialfunktion e^x)
ln(x)	(natürlicher Logarithmus für x>0)
sqrt(x)	(Quadratwurzel für x≥0)

Man beachte, daß abs und sqr int-Werte liefern, wenn die Argumente vom Typ int sind.

Möchte man in einer Rechnung zugleich Dezimalzahlen und ganze Zahlen verwenden, so muß man Funktionen zur *Typkonversion* einsetzen (s. Abschnitt 9.3).

Die Datentypen char und text.
Der Wertebereich des Datentyps char (Abk. von engl. character = dt. Zeichen) umfaßt alle darstellbaren Zeichen des Zeichensatzes einer Programmiersprache sowie Steuerzeichen, die nicht ausgegeben werden können, aber das Ausgabegerät beeinflussen können. Der Typ text läßt sich als $char^*$ beschreiben, also als Datentyp, dessen Wertemenge prinzipiell alle endlichen Folgen von Zeichen aus char enthält. Tatsächlich dürfen die Texte $w \in text$ meist eine vorgegebene feste Länge (meist 2^8 oder 2^{16}) nicht überschreiten.

Konstanten der Datentypen char oder text schließt man oft in Hochkommata (bei char) oder Anführungszeichen (bei text) ein. Will man das Hochkomma (Anführungszeichen) selbst darstellen, so verdoppelt man es.
Beispiel: Konstanten vom Typ char sind:

'a', '9', 'B', '?', ' ' (Leerzeichen), '''' (einzelnes Hochkomma).

Konstanten vom Typ text sind:

"Auto", "9", "Wer ist da?", "Er sagte: ""Hier bin ich""".

Neben pred, succ, ord und den Vergleichen gibt es für char noch die Funktion chr, die die Umkehrung der ord-Funkton bildet:

chr: int→char mit chr(i)=c falls ord(c)=i gilt.

Beispiel: Angenommen, es gilt: ord('A')=55. Dann gilt: chr(55)='A'. Es gilt also stets chr(ord(c))=c und ord(chr(i))=i.

Welches Resultat die ord-Funktion für die einzelnen Zeichen genau liefert, hängt davon ab, wie der Computer die Zeichen intern verschlüsselt. Oft wird der **ASCII-Code** (sprich: aski, Abk. für **a**merican **s**tandard **c**ode for **i**nformation **i**nterchange) verwendet, bei dem jedes Zeichen durch eine Zahl zwischen 0 und 127 dargestellt wird.

Die üblichen Operationen auf Texten sind die Vergleichsoperationen <,=,≠,>,≤,≥, wobei die Vergleiche jeweils die lexikographische Ordnung zugrundelegen, also

"Auto"<"Autobahn" usw.

Weieter Standardoperation ist die Konkatenation • zweier Texte zu einem:

"Baum"•"Haus"="BaumHaus".

9.3 Probleme der Typisierung

In den folgenden Abschnitten werden wir eine Reihe von Konstruktoren kennenlernen, die es uns erlauben, eine Vielzahl von Datentypen zu definieren. Zuvor müssen wir jedoch einige Probleme klären, die sich aus den einzelnen Datentypen und ihrem Verhalten zueinander ergeben. Als Motivation betrachte man folgende Situation:
Beispiel: Die Zahl 100 ist eine reelle Zahl und zugleich eine ganze und eine natürliche Zahl. Sie kann aber auch einen Gasverbrauch (mit gedachter Dimension m^3), die Nummer eines InterCity-Zugs oder den Teil eines Autokennzeichens bezeichnen.
In den ersten drei Fällen erscheint es vernünftig, wenn man die 100 allen arithmetischen Operationen unterwerfen kann.
Im vierten Fall sollte die 100 nur mit solchen anderen Werten verknüpft werden, die zu einem im Zusammenhang mit dem Gasverbrauch sinnvollen Ergebnis führen. Die

Addition eines Gasverbrauchs 100 (in m^3) und eines Stromverbrauchs (in kWh) erscheint sinnlos.
In den beiden letzten Fällen sind alle arithmetischen Operationen sinnlos. Eine sinnvolle Operation auf InterCity-Nummern wäre aber z.B. „Liefere die Nummer des Gegenzugs“.

Man erkennt in obigem Beispiel folgendes: In einem Programm kommen möglicherweise viele Objekte vor, die alle die gleiche Gestalt besitzen (wie die 100), jedoch zu unterschiedlichen Typen gehören. Die Maschine sollte bei der Ausführung des Programms jedoch für jedes Objekt entscheiden, ob die jeweils angewendeten Operationen erlaubt sind oder nicht und ggf. mit einer Fehlermeldung abbrechen. Zur Lösung dieser Aufgabe kennt man zwei zueinander konträre Verfahren, die strenge Typisierung und die schwache Typisierung.

Definition B:
Eine Programmiersprache heißt **streng typisiert**, wenn in jedem Programm der Sprache alle Größen zu genau einem Datentyp gehören, andernfalls **schwach typisiert**.

Die Wertemengen aller Datentypen einer streng typisierten Sprache sind also paarweise disjunkt. In streng typisierten Sprachen kann man jedem Objekt einen eindeutigen Typ zuordnen, und auf die Objekte eines Typs können nur diejenigen Operationen angewendet werden, die für den Typ definiert sind, und sonst keine. Eine Ausnahme bilden lediglich Typen, die durch gewisse Konstruktoren (vor allem durch Restriktion, s.u.) aus einem anderen Typ hervorgegangen sind.

Beispiel: In einer streng typisierten Sprache sind folgende Konstruktionen *nicht* erlaubt, in einer schwach typisierten sind sie erlaubt:
- der Ausdruck sin(7), da sin eine Funktion von real nach real ist, 7 jedoch ein int-Ausdruck ist;
- der Ausdruck x+true, da die Addition die Funktionalität int×int→int oder real×real→real besitzt, zumindest der zweite Parameter true jedoch weder zu int noch zu real gehört. In einer schwach typisierten Sprache stellt sich erst während der Laufzeit des Programms heraus, ob die rechnerinternen Darstellungen von x und true Zahlen entsprechen, die addiert werden können;
- der Ausdruck 17*12.5, da die Multiplikation die Funktionalität int×int→int oder real×real→real besitzt, jedoch in beiden Fällen einer der beiden Parameter den falschen Typ besitzt.

Dieser Ausdruck läßt sich korrigieren, indem man auf 17 eine Funktion zur *Typkonversion* der allgemeinen Form

makereal: int→real

anwendet. Anschließend kann man 12.5 und makereal(17) multiplizieren.

Die Umkehrfunktion hat die allgemeine Form

makeint: real→int

und wandelt real-Werte in int-Werte um.

Einige Programmiersprachen enthalten automatische Typkonversionen; in diesem Fall kann man beide Zahlen unmittelbar verknüpfen. Man muß sich jedoch vergewissern, welchen Typ das Ergebnis besitzt.

Strenge Typisierung erhöht bei der Software-Entwicklung i.a. die Sicherheit, da Typverletzungen bereits durch den Übersetzer erkannt und entsprechende Programme vom System abgewiesen werden. Zudem besitzen die Programme eine größere Portabilität, da sie keinen Bezug auf rechnerinterne Darstellungen nehmen können.

Definition C:
Eine Programmiersprache heißt *statisch typisiert*, falls alle oder die meisten Typüberprüfungen zur Übersetzungszeit durchgeführt werden können. Werden die Typprüfungen während der Laufzeit des Programms durchgeführt, so spricht man von *dynamischer Typisierung*.

Während LISP eine klassische Programmiersprache mit dynamischer Typisierung ist, ist man in den letzten Jahren mehr und mehr zu Programmiersprachen mit statischer Typisierung übergegangen. Denn einerseits erhöht das die Lesbarkeit und Übersichtlichkeit der Programme, und es steigert die Effizienz, da in das übersetzte Programm kein Code für die Typprüfungen eingebunden werden muß, andererseits stellen aktuelle Programmiersprachen oft leistungsfähige *Typinferenzsysteme* bereit, die aus den meisten Ausdrücken den zugehörigen Datentyp ableiten können, so daß Typdeklarationen dennoch in vielen Fällen unterbleiben können.

Wir gehen in diesem Kapitel immer von strenger Typisierung der zugrundeliegenden fiktiven Programmiersprache aus und beschreiben im Einzelfall die zugehörigen Phänomene und ihre Lösung.

9.4 Konstruktoren

Nach Erarbeiten der üblichen in Programmiersprachen vorkommenden elementaren Datentypen behandeln wir in diesem Abschnitt die Konstruktoren, mit denen man die elementaren Typen zu beliebig komplexen Strukturen zusammensetzen kann.

9.4.1 Enumeration

Die Enumeration ist ein Konstruktor, mit dem man durch Einführung endlicher linear geordneter Mengen neue elementare Datentypen definieren kann, indem man alle zugehörigen Elemente aufzählt. Man erhält dann einen sog. *Aufzählungstyp*. Schematisch definiert man einen Aufzählungstyp D durch

$$\underline{\text{typ}}\ D \equiv \{d_1,d_2,...,d_n\}.$$

Hierbei ist jedes Element d_i explizit anzugeben.

Unter Berücksichtigung der Annahme strenger Typisierung beachte man, daß D ein *neuer* Typ ist, dessen Wertemenge disjunkt von der Wertemenge aller übrigen Typen ist. Kommt etwa ein Element d_i noch in einem anderen Datentyp vor, so sind beide Elemente dennoch verschieden voneinander. Das Element -7 vom Typ Z ist verschieden von dem Element -7 des Datentyps D mit

$$\underline{\text{typ}}\ D \equiv \{12,3,-7,25\}.$$

Da man einem Datenelement nicht immer ansehen kann, zu welchem Typ es gehört, fordert man in vielen Programmiersprachen, daß die Elemente eines Aufzählungstyps verschieden von allen übrigen Datentypen gewählt werden. So darf man D meist nicht wie oben angebenen definieren, da D nicht disjunkt zum Typ int ist. Ebenso ist der Datentyp {'x', 2.0, Otto, false} meist nicht zulässig, da er real-, bool- und char-Elemente besitzt.

Die Reihenfolge, in der die Elemente $d_1,...,d_n$ angegeben werden, legt in vielen Programmiersprachen zugleich eine *lineare Ordnung* < auf D fest, für die gilt:

$$d_1<d_2<...<d_{n-1}<d_n.$$

Die Aufzählungstypen sind dann ebenfalls skalare Datentypen.
Beispiel: Für den Aufzählungstyp typ D ≡ (rot, grün, blau) gilt rot<grün<blau.

Zur Manipulation von Aufzählungstypen gibt es – wenn sie skalar sind – die bereits oben genannten zwei Funktionen pred und succ. Weiterhin sind die Vergleichsoperatoren entsprechend der linearen Ordnung erlaubt. Es gilt also z.B. succ(rot)=grün, succ(succ(rot))=blau, pred(grün)=rot und grün<blau.

9.4.2 Restriktion

Dieser Konstruktor modelliert den Übergang zu einer (endlichen oder unendlichen) Teilmenge eines bereits definierten Datentyps. Der allgemeinste Fall der Restriktion erfolgt durch Angabe eines Prädikats. Ein **Prädikat** über einer Menge M ist eine Aussage, die wahr oder falsch sein kann, also eine Abbildung

P: M→{true, false}.

Schematisch definiert man eine Restriktion D' des Datentyps D dann durch

typ D' ≡ D {x | P(x)}.

P ist ein Prädikat. Die Wertemenge von D' ist die Menge aller x vom Typ D, die das Prädikat P erfüllen, für die die Aussage P also wahr ist.

Sonderfall: Verwendet man das Prädikat true, so realisiert die Restriktion die Umbenennung eines Typs.

Beispiel: Mit

typ Zahl ≡ int {x | true}

erhalten wir den aus PRO geläufigen Typ Zahl.

Trotz strenger Typisierung ist D' hier *kein* neuer Typ, d.h., Operationen, die auf D definiert sind, können auch auf Operanden aus D' angewendet werden (sofern D' durch die Operation nicht verlassen wird). D und D' sind also nicht disjunkt.

Beispiel: typ D' ≡ int {x | x mod 2=0}.

D' besitzt als Wertemenge alle geraden ganzen Zahlen.

In dieser Allgemeinheit ist die Restriktion in den meisten Programmiersprachen nicht zugelassen. Als Spezialfälle erhält man folgende Konstruktoren:

a) *Restriktion durch Intervallbildung:* Man schränkt den Grundtyp auf ein zusammenhängendes Intervall ein und definiert schematisch

   typ D' ≡ D [a ..b].

   D muß hier ein skalarer Typ sein. Die Wertemenge von D' ist die Menge aller x∈ D mit a≤x≤b. Die für D geltende lineare Ordnung überträgt sich auf D'.

   Beispiel: typ Lottozahlen ≡ int [1..49].

b) *Restriktion durch Enumeration:* Man zählt die gewünschten Elemente der Teilmenge auf, d.h., man definiert schematisch

   typ D' ≡ D $\{d_1,d_2,...,d_n\}$,

   wobei alle d_i explizit anzugeben sind und aus dem Datentyp D stammen müssen.

   Beispiel: typ HeutigeLottozahlen ≡ Lottozahlen {2,7,13,17,22,49}.

9.4.3 Potenzmengenbildung

Um die Objekte eines Datentyps D' zu Mengen zusammenfassen zu können, geht man zur Potenzmenge, d.h. zur Menge aller Teilmengen von D' über. Man definiert schematisch:

$$\underline{\text{typ}}\ D \equiv 2^{D'}.$$

Beispiele:

1) $\underline{\text{typ}}\ \text{zahlen} \equiv 2^{\text{int}}$.

Die Wertemenge von zahlen ist die Menge aller Teilmengen der ganzen Zahlen int.

2) $\underline{\text{typ}}\ \text{grundfarbe} \equiv \{\text{rot,gelb,blau}\}$

$\underline{\text{typ}}\ \text{farben} \equiv 2^{\text{grundfarbe}}$.

Objekte von Typ farben sind z.B.

{rot,gelb}, {gelb}, {rot,blau,gelb} oder die leere Menge $\varnothing$.

Die Wertemenge von farben läßt sich interpretieren als die Menge der Farben, die durch Mischen der drei Grundfarben gebildet werden können.

Universelle Konstante in jedem Potenzmengentyp ist die leere Menge $\varnothing$.

Auf Objekten vom Typ D sind die üblichen Mengenoperationen vorgesehen, also

$\cup$: $D\times D\to D$ (Vereinigung),

$\cap$: $D\times D\to D$ (Durchschnitt),

\: $D\times D\to D$ (Differenz: $D_1 \backslash D_2 = \{d \in D_1 \mid d \notin D_2\}$),

$|\cdot|$: $D\to \mathbb{N}$ (Anzahl der Elemente, Kardinalität),

$\subseteq$: $D\times D\to\{\underline{\text{true}}, \underline{\text{false}}\}$ (Teilmenge),

$\in$: $D'\times D\to\{\underline{\text{true}}, \underline{\text{false}}\}$ (Element-Beziehung).

In Programmiersprachen sind nur endliche Datentypen als Grundtyp D' zugelassen, und somit ist jedes Objekt des Typs D eine endliche Menge von Objekten des Datentyps D'. Einige Programmiersprachen bieten einen Potenzmengentyp an, viele nicht. Häufig wird verlangt, daß die Wertemenge des Grundtyps D' eine bestimmte Maximalzahl von Elementen nicht überschreiten darf.

9.4.4 Aggregation

Aggregation nennt man das Zusammensetzen von mehreren (möglicherweise verschiedenen) Datentypen $D_1,D_2,...,D_n$ zu einem n-Tupel. Man definiert schematisch

$$\underline{\text{typ}}\ D \equiv (D_1,D_2,...,D_n).$$

Mathematisch ist $D=D_1\times D_2\times...\times D_n$ das kartesische Produkt der Typen $D_1,...,D_n$. $D_1,...,D_n$ heißen **Komponenten** von D. Die Elemente von D sind n-Tupel der Form

$d=(d_1,d_2,...,d_n)$ mit $d_i \in D_i$. Gilt $D_1=D_2=...=D_n$, so nennt man D **homogen**, anderenfalls **inhomogen**.

Beispiel: typ Datum ≡ (int [1..31], int [1..12], int [1900..1999]).

Die drei Datentypen, aus denen Datum zusammengesetzt ist, entstehen durch Restriktion mittels Intervallbildung aus dem Datentyp int. Jedes Objekt d vom Typ Datum besteht aus drei Komponenten, z.B. $d=(17,10,1989)$.

Zu jedem aggregierten Typ $D \equiv (D_1,D_2,...,D_n)$ gehören die **Selektoren**

$\pi_{i,n}: D \rightarrow D_i$ mit

$\pi_{i,n}(d) = d_i$.

Im mathematischen Sinne ist $\pi_{i,n}$ die **Projektion** von d auf die i-te Komponente. $\pi_{i,n}$ liefert also zu jedem Element $d=(d_1,d_2,...,d_n)$ das i-te Element d_i. Konstruktor und Selektor der Aggregation sind durch folgende beiden Gesetze miteinander verknüpft: Durch das *Konstruktionsaxiom*

$(\pi_{1,n}(d),\pi_{2,n}(d),...,\pi_{n,n}(d)) = d$ für alle $d \in D$

und durch das *Selektionsaxiom*

$\pi_{i,n}(d_1,d_2,...,d_n) = d_i$ für alle $d_1 \in D_1$, $d_2 \in D_2$,..., $d_n \in D_n$, i=1,...,n.

Handelt es sich bei einem der Komponententypen wiederum um einen aggregierten Typ, so muß man die Selektion iterieren, um auf ein elementares Datenelement zugreifen zu können. Allgemein berechnet man die Komposition von Selektionen

$\pi_{i_k,n_k} \circ ... \circ \pi_{i_2,n_2} \circ \pi_{i_1,n_1}$,

wobei man mittels π_{i_1,n_1} hierbei zunächst auf die i_1-te Komponente eines aus n_1 Komponenten bestehenden Datentyps zugreift, der aus n_2 Komponenten besteht, dessen i_2-te Komponente man mittels π_{i_2,n_2} selektiert, usw.

Beispiel: Wir definieren unter Verwendung des Typs Datum von oben einen Typ Schulzeit mit dem Einschulungs- und dem Abgangsdatum

typ Schulzeit ≡ (Datum,Datum).

Den Tag des Abgangsdatums erhält man dann durch die doppelte Selektion

$\pi_{1,3} \circ \pi_{2,2}$.

Ist $d=((17,8,1960),(22,4,1976))$ ein Element von Schulzeit, so ist

$\pi_{1,3}(\pi_{2,2}(d))=\pi_{1,3}(22,4,1976)=22$.

Als *parallele Selektion* bezeichnet man die gleichzeitige Anwendung mehrerer Selektoren auf das gleiche Element. Als Resultat erhält man ein Tupel

$(\pi_{i_1,n}, \pi_{i_2,n}, ..., \pi_{i_k,n})(d)= (d_{i_1}, d_{i_2}, ..., d_{i_k})$.

Eine wichtige Anwendung in der Programmierung besteht im Durchlaufen aller Komponenten eines Objektes $d \in D$ in der Reihenfolge $d_1,d_2,d_3,\ldots$. Hierzu benötigt man eine *Nachfolgerfunktion* N auf der Menge der Selektoren, also

$$N: \{\pi_{i,n} \mid n \in IN,\ 1 \le i \le n\} \rightarrow \{\pi_{i,n} \mid n \in IN,\ 1 \le i \le n\} \text{ mit}$$
$$N(\pi_{i,n}) = \pi_{i+1,n} \text{ für } 1 \le i < n.$$

N liefert also zu jedem Selektor $\pi_{i,n}$, $i<n$, den nächsten Selektor $\pi_{i+1,n}$.

Beispiel: Um alle Komponenten des Objektes d vom Typ D auszudrucken, kann man unter Verwendung der Schleife von PRO folgendermaßen programmieren:

```
π:= π1,n;
solange π definiert tue
    write(π(d));
    π:=N(π)
ende
```

In den üblichen Programmiersprachen steht solch eine Nachfolgerfunktion N allerdings nicht zur Verfügung.

Eine erweiterte Form der Aggregation lassen einige Programmiersprachen zu, indem man die einzelnen Komponenten statt über Projektionen über einen Bezeichner anspricht. Eine solche Zusammenfassung nennt man **Verbund** oder **Record**. Jede Komponente eines Verbundes erhält einen Bezeichner. Auf die einzelnen Komponenten (die *Selektion*) wird durch den Bezeichner des Verbundes, gefolgt (durch einen Punkt getrennt) vom Bezeichner der Komponente oder in ähnlicher Form zugegriffen.

Die Definition eines Record lautet allgemein:

$$\underline{typ}\ T \equiv (t_1 : T_1,\ t_2 : T_2,\ \ldots,\ t_n : T_n).$$

Dabei sind $t_1,\ldots,t_n$ paarweise verschiedene (!) Bezeichner, die Selektoren, und $T_1,\ldots,T_n$ beliebige Datentypen. Die Wertemenge von T ist die Menge aller möglichen n-Tupel der Form $(a_1,a_2,\ldots,a_n)$, wobei jedes a_i vom Datentyp T_i ist und über den Bezeichner t_i angesprochen werden kann.

Beispiel: Ein Verbund, der Personaldaten beschreiben soll, läßt sich wie folgt definieren:

```
typ personal ≡ (name : text, gebdat : (nat[1..31], nat[1..12], nat),
                gehalt : real, geschlecht: {m,w}).
```

Wie gesagt können die Typen innerhalb eines Verbundes selbst wieder Verbundtypen sein. Wir können z.B. den Selektor gebdat auch als Verbundtyp deklarieren. Ein Datum ist strukturiert in eine Tages-, eine Monats- und eine Jahresangabe. Wir definieren also:

```
gebdat : (tag: nat[1..31], monat: nat[1..12], jahr: nat)
```

und fügen diesen Typ ein. Wir erhalten

```
typ personal ≡ (name : text, gebdat : (tag: nat[1..31], monat: nat[1..12], jahr: nat),
                gehalt : real; geschlecht: {m,w}).
```

Die **Selektion** erfolgt oft mit der *dot-Notation* (engl. dot = Punkt). Seien die Variable v und der Verbundtyp T wie oben deklariert. Dann bezeichnet der Ausdruck

$$v.t_i$$

die i-te Komponente des Verbunds. v darf meist nur ein einzelner Bezeichner aber *kein* Ausdruck sein. t_i kann wiederum ein dot-Ausdruck sein.

Beispiel: Wir betrachten den oben definierten Verbundtyp personal. In der folgenden Sequenz werden die Komponenten von angest sukzessive mit Werten belegt, anschließend wird das Gehalt um 200 DM erhöht:

```
def angest: personal;
angest.name ← "MEIER";
angest.gebdat.tag ← 14;
angest.gebdat.monat ← 3;
angest.gebdat.jahr ← 1936;
angest.gehalt ← 2783.24;
angest.geschlecht ← w;
angest.gehalt ← angest.gehalt + 200.
```

Man beachte noch, daß die innerhalb eines Verbundes verwendeten Bezeichner *untereinander* verschieden sein müssen, aber mit außerhalb des Verbundes vorkommenden Bezeichnern übereinstimmen dürfen. In der folgenden Deklaration

```
def  x : int;
def  y : ( x : real, y : bool).
```

kann man die Variable x eindeutig von der Variablen y.x unterscheiden, ebenso die Variable y von der Variablen y.y. Zum besseren Verständnis kann man sich die Zeichenfolge y.y als *einen* Bezeichner vorstellen.

Im obigen Beispiel empfindet man es als lästig, in der dot-Notation ständig die Bezeichner angest bzw. gebdat wiederholen zu müssen. Zur Abkürzung bieten Programmiersprachen eine Anweisung der allgemeinen Form

```
inspect ... tue ...
```

Wird mehrfach hintereinander auf die Komponenten der gleichen Verbundvariablen zugegriffen, so kann durch Verwendung der inspect-Anweisung das ständige Voranstellen der Verbundvariablen vor die Komponentenbezeichner (mit Punkt dazwischen) unterbleiben.

Beispiel: Im obigen Beispiel können wir die Sequenzen wie folgt abkürzen:

```
inspect angest tue
    name←"MEIER";
    inspect gebdat tue
        tag←14;
        monat←3;
        jahr←1936
    ende;
    gehalt←2783.24;
    geschlecht←w;
    gehalt←gehalt+200
ende.
```

Mehrfach ineinander geschachtelte inspect-Anweisungen der Form

```
inspect V1 tue ...
    inspect V2 tue ...
        ...
        inspect Vn tue ...
```

können oft verkürzt werden zu

```
inspect V1, V2, ..., Vn tue ...
```

Beispiel: Obiges Beispiel verkürzen wir weiter auf die Sequenz

```
inspect angestellter,gebdat tue
    name←"MEIER";
    tag←14;
    monat←3;
    jahr←1936;
    gehalt←2783.24;
    geschlecht←w;
    gehalt←gehalt+200
end.
```

9.4.4.1 Homogene Aggregation in imperativen Sprachen

In imperativen Programmiersprachen gibt es eine besondere Erscheinungsform der Aggregation vermöge des Sprachelements array (Feld).

Der Konstruktor array faßt eine beliebige vorher festzulegende Anzahl von Daten gleichen Datentyps unter einem Bezeichner zu einem **Feld** (**array**) zusammen. Der Zugriff zu einzelnen Datenelementen eines Feldes (die *Selektion*) erfolgt über einen **Index**, der mit dem Bezeichner verbunden wird. Zur Deklaration eines Datentyps "Feld" benötigt man einen *Indextyp* I und einen *Grundtyp* T. Der Indextyp ist ein endlicher skalarer Datentyp, meist eine Restriktion von int. Man deklariert einen Feldtyp A oft wie folgt:

typ A ≡ array I of T.

Sei $\{i_1,i_2,...,i_n\}$ die Wertemenge von I mit der Ordnung $i_1<i_2<...<i_n$. Dann ist die Wertemenge von A die Menge aller n-Tupel

$(a_{i_1}, a_{i_2}, ..., a_{i_n})$ mit $a_{i_j} \in T$.

Ein Objekt vom Typ A besitzt eines dieser Tupel als Wert (oder ist undefiniert).

Beispiel: typ A ≡ array nat[4..9] of int .

Jedes Objekt a des Typs A kann Werte aus der Menge aller 6-Tupel

$(a_4, a_5, ..., a_9)$

mit ganzen Zahlen $a_4,a_5,...,a_9$ annehmen.

Soweit die Konstruktion. Die **Selektion** lehnt sich in ihrer Schreibweise an die Form a_i an, wobei jedoch statt der mit der Tastatur schwierig darzustellenden Indizierung runde Klammern verwendet werden. Sei E ein Ausdruck, dessen Wert i in der Wertemenge des Indextyps I liegt. Dann bezeichnet der Ausdruck

a(E)

das zu i gehörende Element des Tupels, also a_i. Ist i das k-te Element in der Aufzählung der Wertemenge von I, dann heißt a(E) (=a(i)) die **k-te Komponente** des Feldes a. In obigem Beispiel bezeichnet a(3*3-2) das zu 7 gehörende Element a_7; a(7) ist das vierte Element des Feldes a. Statt der runden Klammern a(E) verwendet man in vielen Programmiersprachen eckige Klammern a[E].

Beispiele:

1) In folgendem Programm werden zwei Felder a und b mit je fünf Elementen vom Typ int deklariert. Die beiden Felder werden anschließend komponentenweise addiert und ausgegeben. Das Programm addiert also fünfstellige Vektoren. Die Spezifikation:

   spec vektoradd: $Z^5 \times Z^5 \rightarrow Z^5$ with
   vektoradd(a,b)=c where
   pre a=$(a_1,...,a_5)$ und b=$(b_1,...,b_5)$ mit $a_i,b_i \in Z$ für $1 \le i \le 5$
   post c=$(c_1,...,c_5)$ mit $c_i=a_i+b_i$ für $1 \le i \le 5$.

   Das Programm:

```
typ feld ≡ array nat[1..5] of int;
def a,b: feld;
def i: int;
i←1;
solange i≤5 tue
      lies(a(i));
      lies(b(i));
      zeige(a(i)+b(i))
```

ende.

2) In folgendem Programm wird ein Text mit 100 Buchstaben eingelesen. Ausgegeben wird für jeden der Buchstaben a, b, c, d oder e die Häufigkeit, mit der er im eingelesenen Text auftritt. Die Spezifikation:

spec such: $A^* \rightarrow (A \times IN_0)^5$ with
such(w)=z where
pre |w|=100
post $z=((c_1,n_1),...,(c_5,n_5))$ mit c_1='a', c_2='b', c_3='c', c_4='d', c_5='e'
und $n_i=\#_{c_i}(w)$ für $1 \leq i \leq 5$, wobei

$$\#_c(w) = \begin{cases} 0, & \text{falls } w=\varepsilon, \\ 1+\#_c(w'), & \text{falls } w=cw', \\ \#_c(w'), & \text{falls } w=xw' \text{ und } x \neq c. \end{cases}$$

Im Deklarationsteil wird ein Feld anzahl vom Grundtyp int deklariert. Der Indextyp besteht aus der Menge der möglichen Buchstaben, er ist eine Restriktion durch Intervallbildung des Typs char:

char['a'..'e'].

Der Anweisungsteil beginnt mit der *Initialisierung* des Feldes: Alle Komponenten werden auf Null gesetzt. Anschließend wird 100-mal ein Zeichen eingelesen und, sofern es sich um einen der betreffenden Buchstaben handelt, die zu ihm gehörende Feldkomponente jeweils um Eins erhöht. Ausgegeben wird schließlich jeder Buchstabe zusammen mit seiner Häufigkeit:

```
typ  buchstaben ≡ char['a'..'e'];
def  anzahl : array buchstaben of int;
def  ch : char;
def  i : int;
ch ← 'a';
solange ch≠'⊥' tue
    anzahl(ch) ← 0;
    ch ← succ(ch)
ende;
i :=1;
solange i≤100 tue
    lies (ch);
    wenn ch∈ {'a','b','c','d','e'} dann
        anzahl(ch)← anzahl(ch)+1
    sonst
    ende;
    i← i+1
ende;
ch ← 'a';
solange ch≠'⊥' tue
```

```
        zeige(ch);
        zeige(anzahl(ch));
        ch ← succ(ch)
ende.
```

Eine verallgemeinerte Situation erhält man, wenn man in einer Felddeklaration der Form

$\underline{\text{typ}}$ A ≡ $\underline{\text{array}}$ I $\underline{\text{of}}$ T

für T wiederum einen $\underline{\text{array}}$-Typ einsetzt. Man definiert so einen Typ der Form $\underline{\text{array}}$ I $\underline{\text{of}}$ $\underline{\text{array}}$ J $\underline{\text{of}}$ T. Seien $I_1,I_2,...,I_n$ Indextypen. Allgemein kann man also folgende Deklaration bilden:

$\underline{\text{typ}}$ A ≡ $\underline{\text{array}}$ I_1 $\underline{\text{of}}$
$\underline{\text{array}}$ I_2 $\underline{\text{of}}$
...
$\underline{\text{array}}$ I_n $\underline{\text{of}}$ T

Zur Deklaration und Verwendung dieser "Mehrfachfelder" kann folgende Abkürzung verwenden, indem man die Indextypen aggregiert:

$\underline{\text{array}}$ $(I_1,I_2,...,I_n)$ $\underline{\text{of}}$ T.

Auf die gleiche Weise kürzt man Selektorausdrücke der Form

$a(E_1)(E_2)$ ... (E_n)

durch

$a(E_1,E_2,...,E_n)$

ab.

Die Anzahl der in einer Felddeklaration aufgeführten Indextypen bezeichnet man als **Dimension** des Feldes. Felder der Dimension 1 heißen **1-dimensional** oder **linear**, Felder der Dimension 2 **2-dimensional** usw. Mit eindimensionalen Feldern beschreibt man Vektoren, mit zweidimensionalen Feldern Matrizen.

9.4.5 Generalisation

Die Generalisation vereinigt disjunkte Datentypen zu einem *neuen* Datentyp. Man schreibt

$\underline{\text{typ}}$ D ≡ D_1 | D_2 | ... | D_n

wobei $D_i \cap D_j = \emptyset$ für alle $1 \le i < j \le n$ gilt. $D_1,D_2,...,D_n$ nennt man auch *Varianten* von D. Für n=1 ergibt sich eine Umbenennung eines Datentyps.

Beispiele:

1) Für die Darstellung einer Geraden im Koordinatensystem kennt man mindestens drei verschiedene Formen:
   - die Normalform ax+by+c=0,

- die Punkt-Steigungsform $y=m(x-x_0)+y_0$,
- die Zwei-Punkte-Form $(y-y_0)/(x-x_0)=(y_1-y_0)/(x_1-x_0)$.

Um alle Formen in einem Datentyp zu erfassen, definiert man mit der Generalisation:

typ punkt ≡ (real,real);
typ steigung ≡ real;
typ gerade ≡ (punkt,punkt) | (punkt,steigung) | (real,real,real).

2) Häufig muß man Datentypen um Fehlerelemente ergänzen. Wir haben dies bereits bei Funktionen getan, und die Wertebereiche um das Element ⊥ (undefiniert) erweitert, z.B.:

typ intplus ≡ int | {undef};

Wegen der strengen Typisierung ist D nicht einfach die mengentheoretische Vereinigung der disjunkten Typen $D_1,...,D_n$, vielmehr werden durch die Generalisation anschaulich die Typen $D_1,...,D_n$ kopiert, vereinigt und anschließend mit D identifiziert. Die Wertemenge von D ist also formal verschieden von der Vereinigung $D_1 \cup D_2 \cup ... \cup D_n$. Dies hat bei strenger Typisierung Konsequenzen für die Operationen, die auf D bzw. auf den Varianten definiert sind. Operationen, die auf D_i definiert sind, sind nicht automatisch auch für D definiert, auch dann nicht, wenn die beteiligten Objekte aus D aus der Variante D_i stammen. Diese Vorschrift hängt mit dem Wunsch zusammen, Operationen, die man für D oder für $D_1,...,D_n$ definiert hat, genau auf die jeweiligen Typen zu beschränken. In vielen Programmiersprachen sorgt jedoch eine automatische *Typkonversion* dafür, daß sich Operationen von D_i auf D übertragen.

Beispiel: Der einfachste Fall der Generalisation liegt für n=1 vor. Wir definieren:

typ wasserverbrauch ≡ int;
typ gasverbrauch ≡ int.

Die Operationen von int übertragen sich nicht auf wasserverbrauch und gasverbrauch. Das erscheint sinnvoll, denn viele Gaswerke rechnen den Gasverbrauch von m^3 zunächst in kWh um, bevor sie den Gesamtpreis ermitteln. Solch eine Umrechnungsfunktion darf nicht auf Wasserverbrauchszahlen angewendet werden, auch wenn der gleiche Grundtyp int dies prinzipiell nahelegt. Definieren wir also unterschiedliche Operationen für die Typen, so möchten wir sicher gehen, daß die Operationen auch nur auf Objekte der zugehörigen Typen angewendet werden.

Wichtigste Standardoperation auf generalisierten Typen ist der Operator in, mit dem man feststellen kann, zu welcher Variante ein Objekt x gehört. Es gilt:

x in D= true, falls x vom Datentyp D ist,

false, sonst.

Beispiel: Ein Objekt zählerstand vom Typ gasverbrauch können wir gemäß folgender Anweisung behandeln:

```
wenn zählerstand in gasverbrauch
    dann "Rechne zählerstand von m^3 in kWh um" ende
```

In Programmiersprachen mit automatischer Typkonversion ist das Konzept der Generalisation etwas aufgeweicht. So sind zwar die beiden Typen A und B definiert durch

```
typ  A ≡ int;
     B ≡ int;
```

verschieden. Die automatische Typkonversion sorgt jedoch dafür, daß man Objekte beider Typen in Ausdrücken mischen darf, z.B. ist

```
a←b
```

zugelassen, wenn a vom Typ A und b vom Typ B ist.

Beispiele:

1) Wir möchten einen Typ fahrzeug definieren, dessen Merkmale der Hersteller des Fahrzeugs, sein Neupreis, ob er Nabenschaltung besitzt oder nicht, die Ladefläche und die Zahl der Stehplätze ist. Offensichtlich sind die letzten drei Merkmale nicht gleichzeitig sinnvoll mit Werten zu belegen. Bei einem LKW wird das Merkmal ladefläche belegt und die Merkmale nabenschaltung und stehfläche sind unsinnig. Ist andererseits das Merkmal nabenschaltung=true, so handelt es sich um ein Fahrrad und das Merkmal stehplätze ist unsinnig. Bei einem Bus ist dagegen gerade dieses Merkmal gefragt. Fazit: Man möchte abhängig von dem Wert einzelner Verbundkomponenten bestimmte andere aus- oder einblenden. Dies leistet die Generalisation. Sie sorgt hier außerdem für größere Programmsicherheit. Hat z.B. stehplätze einen Wert, so läßt sich automatisch erkennen, daß der Zugriff auf die Komponente nabenschaltung unsinnig ist und mit einer Fehlermeldung beantwortet werden muß. Wir führen eine zusätzliche Komponente sorte ein, die je nach Wert (fahrrad, lkw oder bus) die relevanten Komponenten sichtbar werden läßt:

```
typ fahrzeug ≡ (hersteller: text, neupreis: real,
                    (sorte: {fahrrad},nabenschaltung: bool) |
                    (sorte: {lkw}, ladefläche: real) |
                    (sorte: {bus}, stehplätze: nat)).
```

   sorte ist hier ein sog. *Typdiskriminator*.

2) Wir definieren einen Datentyp, der typische Personendaten eines Standesamtes enthalten soll. Im einzelnen sollen gespeichert werden:

Nachname
Vornamen (bis zu 3)
Geschlecht (w, m)
Familienstand (led, verh, verw, gesch)
falls verheiratet oder verwitwet:
- Heiratsdatum (Tag, Monat, Jahr)
falls geschieden:
- Scheidungsdatum (Tag, Monat, Jahr)
- erste Scheidung? (bool)

Wir definieren wie folgt:

```
typ geschlecht ≡ {m,w};
typ datum ≡ (jahr: int, monat: nat[1..12], tag: nat[1..31]);
typ namen ≡ (vorname1: text, vorname2: text, vorname3: text, name: text);
typ person ≡ (name: namen, gesch: geschlecht,
                    (famstand: {ledig} |
                    (famstand: {verh,verw}, heiratdatum: datum) |
                    (famstand: {gesch}, scheidatum: datum, erste: bool));
```

9.4.6 Rekursion

Bisher führten alle Datentypkonstruktoren auf endliche Strukturen mit insgesamt zwar möglicherweise großen, aber *endlichen* Wertebereichen. Mit der Rekursion (von lat. recurrere=zurückführen) überspringen wir die Grenze von endlichen hin zu unendlichen Strukturen, denn die Rekursion bewirkt den Übergang zu *abzählbar unendlichen* Wertebereichen, deren Elemente gleichartig aus einfacheren Elementen eines Datentyps aufgebaut sind. Mit der Rekursion kann man mit einer endlichen Beschreibung unendliche Sachverhalte formulieren.

Eine rekursiver Datentyp D hat prinzipiell folgende Form:

typ D ≡ T | D'.

Hierbei sind T und D' Datentypen. T ist der sog. *terminale* (*atomare*) Datentyp, auf den sich die Definition (schließlich) abstützt, D' bezeichnet irgendeinen Typ, in dessen Definition wieder D vorkommt. Das Zeichen "|" ist das bekannte Zeichen zur Generalisation. Welche Bedeutung besitzt D, was wird durch D definiert? Nehmen wir an D kommt in D' zweimal vor; D' hat also die Form

... D ... D ...

Die Wertemenge von D ist dann offenbar die disjunkte Vereinigung

$D=T \cup D'$.

Da in D' wiederum D vorkommt, sind diese D selbst per Definition disjunkte Vereinigungen von T und D', so daß in nächster Näherung gilt:

$$D=T\cup D'=T\cup(\ldots D \ldots D \ldots)$$
$$=T\cup(\ldots T \ldots T \ldots)\cup(\ldots T \ldots D' \ldots)\cup(\ldots D' \ldots T \ldots)\cup(\ldots D' \ldots D' \ldots)$$
$$=T\cup(\ldots T \ldots T \ldots)\cup(\ldots T \ldots (\ldots D \ldots D \ldots) \ldots)\cup(\ldots (\ldots D \ldots D \ldots) \ldots T \ldots)$$
$$\cup(\ldots (\ldots D \ldots D \ldots) \ldots (\ldots D \ldots D \ldots) \ldots).$$

Die im letzten Ausdruck vorkommenden D können nun fortwährend weiter ersetzt werden. Der Wertebereich von D ist folglich die unendliche Vereinigung gewisser gleichartig aufgebauter anderer Wertebereiche.

Definition D:
Die Definition eines Problems, eines Datentyps, eines Verfahrens oder einer Funktion durch sich selbst bezeichnet man als **Rekursion**.
Erscheint auf der rechten Seite einer Datentypdefinition der Form

$$\underline{typ}\ D \equiv T \mid D'$$

innerhalb von der D' Datentyp D selbst, so spricht man von **direkter Rekursion**.
Gibt es eine Folge von Definitionen

$$\underline{typ}\ D \equiv T \mid D_1;$$
$$\underline{typ}\ D_1 \equiv T_1 \mid D_2;$$
$$\ldots$$
$$\underline{typ}\ D_{n-1} \equiv T_n \mid D_n;$$

mit $n\geq 2$ der Art, daß D für $1\leq i\leq n-1$ nicht in D_i, wohl aber in D_n vorkommt, so spricht man von **indirekter Rekursion**.

Beispiele:

1) Ein nichtleerer Text ist entweder ein einzelnes Zeichen, oder er besteht aus einem einzelnen Zeichen, dem ein Text folgt. Wir definieren daher:

   $$\underline{typ}\ X \equiv char \mid (char,X).$$

   char ist der terminale Typ. Beispiele für Objekte vom Typ X sind:

   'a' oder ('A',('u',('t','o'))).

   X beschreibt die Vereinigung aller folgenden Datentypen:

   char, (char, char), (char, (char, char)),...,
   (char, (char, (char,... (char, char)...))),... ,

   also $\bigcup_{i=1}^{\infty}$ (char, (char, (char,... (char, char)...)))
   i-mal

Die Wertemenge von X entspricht der Menge aller nichtleeren Zeichenfolgen über char (man ignoriere die Klammern und Kommata).

Nimmt man noch den leeren Text hinzu, so definiert man

typ X' ≡ {eps} | (char,X').

Der leere Text wir durch das einzige Element des enumerierten Typs {eps} erfaßt. Die Wertemenge von X' ist isomorph zur Wertemenge des Standarddatentyps text. Beispiele für Objekte vom Typ X' sind:

eps, ('a',eps) oder ('A',('u',('t',('o',eps)))).

Man beachte, daß nun {eps} der terminale Typ ist und sich folglich jede rekursive Auflösung auf {eps} abstützen muß. Daher endet jedes Objekt auf eps.

2) Allgemein bezeichnet man einen Datentyp L, der definiert ist durch

typ L ≡ {eps} | (D,L)

als **Linkssequenz** über dem Datentyp D; links deswegen, weil an eine gegebene Sequenz links ein neues Element angefügt werden kann. eps bezeichnet hier die *leere Sequenz*, das *leere Wort*.

L ist die Vereinigung aller Typen der Form

{eps}, (D,{eps}), (D,(D,{eps})), (D,(D,(D,{eps}))), (D,(D,(D,(D,{eps})))) usw.

Der Datentyp X aus Beispiel 1 ist eine Linkssequenz über char. Der aus PRO bekannte Typ Zahlenfolge ist isomorph zu einer Linkssequenz, die wie folgt definiert werden kann:

typ Zahlenfolge ≡ {eps} | (int,Zahlenfolge).

Analog spricht man bei einem Datentyp von **Rechtssequenz** über D, wenn er wie folgt definiert ist:

typ R ≡ {eps} | (R,D).

R ist die Vereinigung aller Typen der Form

{eps}, ({eps},D), (({eps},D),D), ((({eps},D),D),D), ((((eps},D),D),D),D) usw.

In einigen Programmiersprachen gehören Linkssequenzen zusammen mit Operationen wie erstes oder rest zu den Standarddatentypen.

3) Eine Anwendung für die indirekte Rekursion ist die Bildung von einfachen arithmetischen Ausdrücken (s. auch Abschnitt 9.4.6.2) mit Klammern (hier nur über einbuchstabigen Bezeichnern):

```
typ Bezeichner ≡ char['a'..'z'];
typ op ≡ {+,-,*,/};
typ Term ≡ Bezeichner | Ausdruck | ( {(}, Ausdruck, {)} );
typ Ausdruck ≡ (Term, op, Term).
```

Ausdruck verwendet Term, Term verwendet Ausdruck, und beide stützen sich schließlich auf den terminalen Datentyp Bezeichner ab. Beispiel für ein Objekt vom Typ Ausdruck ist

$$(u,+,((,(b,*,h),)))$$.

Man beachte, daß hier die Klammern "(" und ")" sowohl für die Aggregation gebraucht werden, als auch als Zeichen in Termen auftreten. Dadurch erscheinen diese Definitionen auf den ersten Blick verwirrend. Wir haben zur Unterscheidung die Aggregationsklammern vergrößert dargestellt.

Einige der Datentypen, die man mit dem Rekursionskonstruktor definieren kann, besitzen eine herausragende Stellung in der Informatik. Es sind fundamentale Strukturen, die Sie ein Leben begleiten werden. Dazu gehören Files und Bäume.

9.4.6.1 Files

"File" bedeutet soviel wie Kartei, Datei. Als File bezeichnet man eine bestimmte Kombination von Links- und Rechtssequenz über einem Datentyp D, auf der die Operationen get, move, put, eof, reset und rewrite definiert sind.

Ein File kann man sich als Magnetband (Tonband) vorstellen, das in einzelne Zellen unterteilt ist. Jede Zelle kann genau ein Element des zugrundeliegenden Datentyps D aufnehmen. Das Magnetband wird bearbeitet, indem es an einem Lese-Schreibkopf vorbeigezogen wird. Eine Zelle des Magnetbandes kann nur dann gelesen oder beschrieben werden, wenn sich die Zelle unter dem Lese-Schreibkopf befindet. Der Kopf wirkt wie ein Sichtfenster, das zu jedem Zeitpunkt nur immer genau eine Zelle zeigt (Abb. 2). Der *lesende* Zugriff zu den einzelnen Datenelementen eines Files (die *Selektion*) beginnt grundsätzlich beim ersten Element. Erst, wenn dieses gelesen wurde, kann das zweite Element gelesen werden, und erst danach das dritte usw. Das n-te Element kann also erst gelesen werden, wenn zuvor alle n-1 vorangehenden Elemente gelesen wurden. Fachmännisch spricht man vom **sequentiellen Zugriff**. Wollen wir ein File um Daten ergänzen (Schreiben), so darf dies nur am Schluß des Files geschehen (Abb. 3).

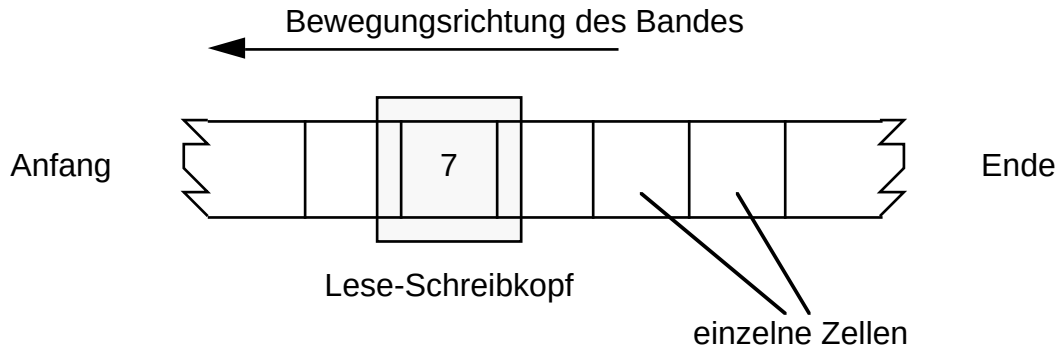


Abb. 2: Lesen und Schreiben beim File

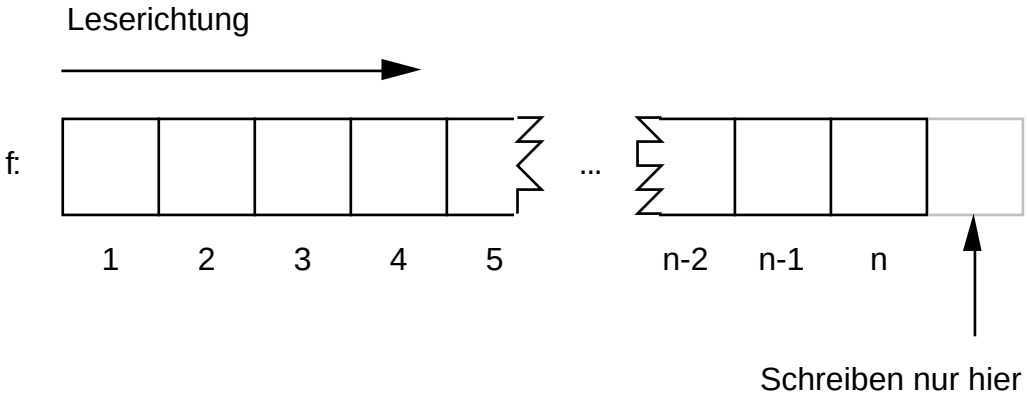


Abb. 3: File mit Lese-Schreibkopf

Die Definition eines Files über einem beliebigen Datentyp D lautet:

typ L ≡ {eps} | (D,L);
typ R ≡ {eps} | (R,D);
typ file ≡ (R,D,L).

In dieser Typdefinition file beschreibt R den Aufbau bereits gelesener Teile des Files (also links vom Sichtfenster), L den Aufbau noch nicht gelesener Teile (also rechts vom Sichtfenster) und D den Typ des Objekts, das sich gerade unter dem Sichtfenster befindet und bearbeitet werden kann. Zu file gehören alle Objekte der Form

(*) $f=\big((...(((eps,d_1),d_2),d_3),...,d_{k-1}),d_k,(d_{k+1},(d_{k+2},(d_{k+3},...(d_n,eps)...)))\big)$ mit $d_i \in D$.

Rechtssequenz ↑ Linkssequenz

Sichtfenster

Beispiel: Die Definition eines Files über dem Datentyp int lautet:

typ L ≡ {eps} | (int,L);
typ R ≡ {eps} | (R,int);
typ intfile ≡ (R,int,L).

Zur Bearbeitung eines Objekts f vom Typ file stehen üblicherweise folgende Standardoperationen zur Verfügung. f sei dabei im folgenden – wenn nicht anders angegeben – von der obigen allgemeinen Form (*).

Funktion eof (end of file).

Das Erreichen des Fileendes kann mit Hilfe der Funktion eof abgeprüft werden. Es gilt eof(f)=true, falls das Sichtfenster hinter dem letzten Element des Files f steht (Abb. 4) – in diesem Falle ist das Sichtfenster undefiniert ⊥ –, und eof(f)=false in den übrigen Fällen. Genauer:

eof: file→bool mit

true, falls $f=((...((eps,d_1),d_2),...,d_n),\perp,eps)$ für $n\geq 0$,

eof(f)=

false, sonst.

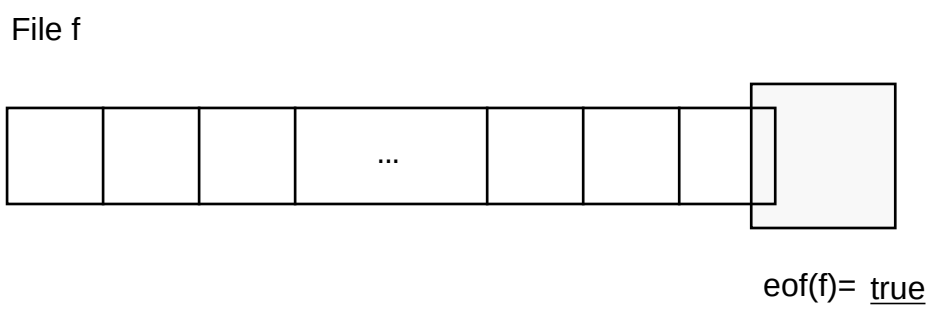


Abb. 4: Situation bei eof(f)=true

Funktion reset.

Die Operation

reset(f)

bereitet ein File zum Lesen vor (sog. *Öffnen*), setzt das Sichtfenster auf das erste Element des Files f, sofern f nicht leer ist (Abb. 5). Ist f leer, so ist das Sichtfenster undefiniert, und der Aufruf eof(f) liefert den Wert true. Genauer:

reset: file→file mit

$(eps,d_1,(d_2,(d_3,,...(d_n,eps)...)))$, falls $f\neq(eps,\perp,eps)$

reset(f)=

f, sonst.

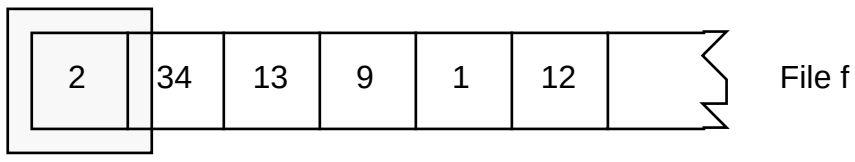


Abb. 5: Wirkung von reset

Funktion get.

Die Operation

get(f)

liefert das unter dem Sichtfenster stehende Fileelement. Falls das Fileende von f erreicht ist, also eof(f)=true, hat get(f) einen undefinierten Wert. Genauer:

get: file→D mit

$$\text{get}(f) = \begin{cases} d_k, & \text{falls } 1 \le k \le n \\ \perp, & \text{sonst.} \end{cases}$$

Funktion move.

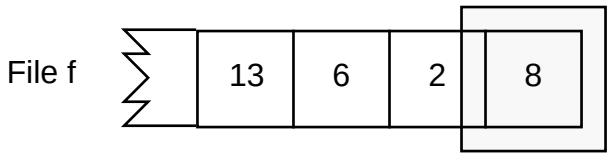
Die Operation

move(f)

bewegt das Sichtfenster um eine Position nach rechts weiter (Abb. 6). Genauer:

move: file→file mit

$$\text{move}(f) = \begin{cases} \big((\ldots((\text{eps},d_1),d_2),\ldots,d_k),d_{k+1},(d_{k+2},\ldots(d_n,\text{eps})\ldots)\big), & \text{falls } 1 \le k < n \\ \big((\ldots((\text{eps},d_1),d_2),\ldots,d_n),\perp,\text{eps}\big), & \text{sonst.} \end{cases}$$



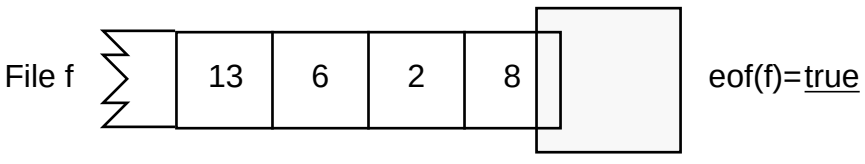


Abb. 6: Wirkung von move am Ende des Files (oben vor, unten nach Aufruf)

Funktion put.

Die Operation

put(x,f)

fügt einen Wert am Ende des Files f an, indem es den undefinierten Wert unter dem Sichtfenster durch x ersetzt (Abb. 7). put darf nur verwendet werden, wenn eof(f)=true gilt. Genauer:

put: D×file→file mit

$$\text{put}(z,f) = \begin{cases} ((\ldots((\text{eps},d_1),d_2),\ldots,d_k),x,\text{eps}), & \text{falls } k>n, \\ \perp, & \text{sonst.} \end{cases}$$

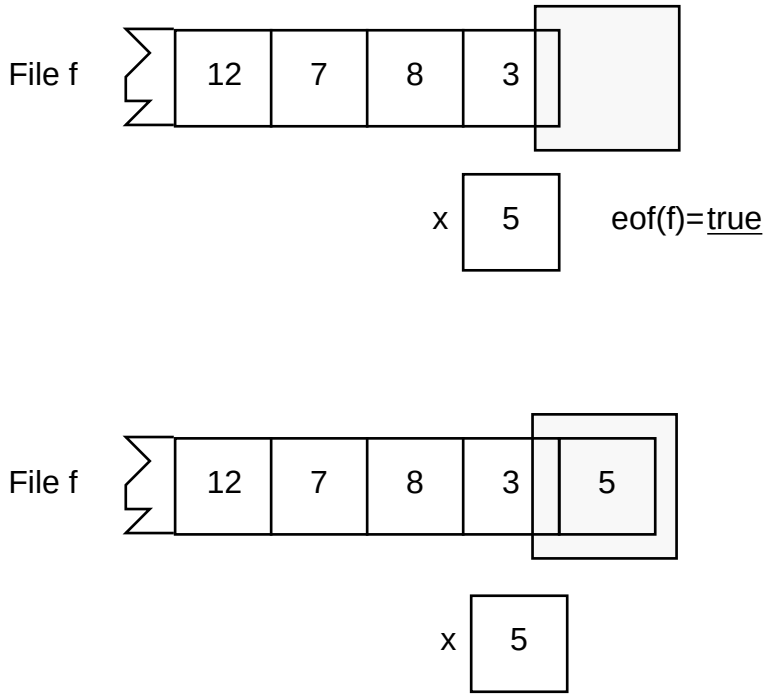


Abb. 7: Wirkung von put (oben vor, unten nach Aufruf)

Funktion rewrite.

Die Operation

rewrite(f)

löscht den gesamten Fileinhalt (Abb. 8), und eof(f) liefert den Wert true. Anschließend kann man das File neu beschreiben. Genauer:

rewrite: file→file mit

rewrite(f)=(eps,⊥,eps).

Abb. 8: Wirkung von rewrite

Beispiele:

1) Gegeben sind zwei Files f und g vom Grundtyp int. f und g sollen verkettet (konkateniert) und in dem File h abgespeichert werden. Anschaulich soll also die bekannte PRO-Anweisung für Objekte vom PRO-Typ Zahlenfolge programmiert werden:

   h ← f • g.

   Die Spezifikation:

   spec concat: $Z^* \times Z^* \rightarrow Z^*$ with
   concat(f,g)=h where
   pre f=[f_1,...,f_n], g=[g_1,...,g_m] mit $f_i, g_i \in Z$
   post h=[f_1,...,f_n,g_1,...,g_m].

   Zunächst öffnen wir f zum Lesen (reset) und h zum Schreiben (rewrite). Anschließend lesen wir fortlaufend nacheinander alle Elemente von f und schreiben Sie jeweils auf h. Danach öffnen wir g zum Lesen und schreiben dessen Elemente gleichfalls nach h, also hinter die Elemente von f:

```
typ L ≡ {eps} | (int,L);
typ R ≡ {eps} | (R,int);
typ intfile ≡ (R,int,L);
def f, g, h : intfile;
f ← reset (f);
h ← rewrite (h);
solange nicht eof (f) tue
    h ← put (get(f),h);
    f ← move(f);
    h ← move(h)
ende;
g ← reset (g);
solange nicht eof (g) tue
    h ← put (get(g),h);
    g ← move(g);
    h ← move(h)
ende.
```

2) Zwei Files f und g sollen zum File h gemischt werden (s. Kapitel 2). Der Grundtyp von f, g und h ist der Verbundtyp

typ Namen ≡ (vorname: text, nachname: text).

f und g sind nach Nachnamen alphabetisch sortiert. Diese Sortierung soll nach dem Mischen auch für h gelten. Die Spezifikation ist eine Übungsaufgabe. Das Programm:

```
typ Namen ≡ (vorname: text, nachname: text);
typ L ≡ {eps} | (Namen,L);
typ R ≡ {eps} | (R,Namen);
typ Namenfile ≡ (R,Namen,L);
def f,g,h : Namenfile;
def x,y : Namen;
{An dieser Stelle seien f und g irgendwie belegt worden}
f ← reset (f);
g ← reset (g);
h ← rewrite (h);
solange nicht (eof(f) oder eof (g)) tue
    x ← get(f);
    y ← get(g);
    wenn x.nachname < y.nachname dann
        h ← put(x,h);
        f ← move (f)
    sonst
        h ← put(y,h);
        g ← move (g)
    ende;
    h ← move (h)
ende;
solange nicht eof (f) tue
    h ← put (get (f));
    h ← move (h);
    f ← move (f)
ende;
solange nicht eof (g) tue
    h ← put (get (g));
    h ← move (h);
    g ← move (g)
ende.
```

9.4.6.2 Bäume

Sequenzen sind besonders einfach strukturierte Datenobjekte. Jedes Element der Sequenz (bis auf das letzte) hat genau einen Nachfolger. In der Praxis gibt es jedoch sehr

viele Fälle, in denen eine Struktur mit Verzweigungen vorliegt, eine Struktur also, bei der ein Element auch mehrere Nachfolger besitzt.
Beispiel: Wir betrachten folgende Personalhierarchie in einer Firma. Es gibt einen Hauptabteilungsleiter (HAL), der zwei Abteilungsleiter (AL) unter sich hat. Jeder Abteilungsleiter kontrolliert mehrere Gruppenleiter (GL), und jede Gruppe hat mehrere Mitglieder (M). Graphisch läßt sich diese Struktur wie in Abb. 9 darstellen.

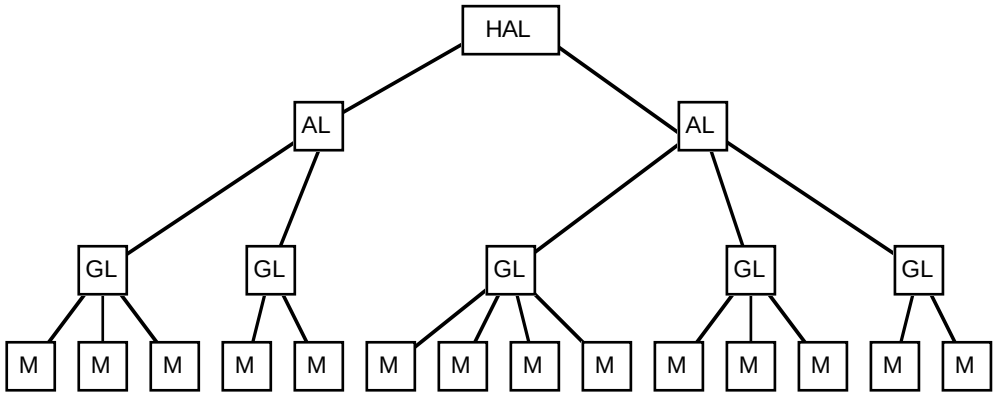
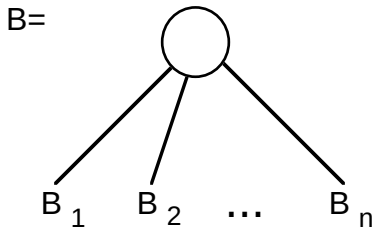


Abb. 9: Darstellung der Personalhierarchie als Baum

Eine Struktur dieser Form bezeichnet man als Baum.

Definition E:
Ein **geordneter Baum** B ist rekursiv wie folgt definiert:
1) Die leere Struktur ist ein Baum.
2) Wenn $B_1,B_2,...,B_n$ für $n\geq 1$ Bäume sind, dann ist auch



ein Baum.

Die Verbindungslinien heißen **Kanten**. Die Bäume $B_1,B_2, ...,B_n$ nennt man **Teilbäume**. Die Schnittpunkte sind die **Knoten**. Der "oberste" Knoten von B heißt **Wurzel** von B. Die Wurzeln der Teilbäume nennt man **Söhne**. Ein Knoten, der keine Söhne hat, ist ein **Blatt**. Knoten, die nicht Blätter sind, heißen **innere** Knoten. Die **Höhe** h von B ist rekursiv definiert durch:

0, falls B der leere Baum ist,

$$h(B)= \begin{cases} 1+\max\{h(B_i) \mid 1 \le i \le n\}, & \text{falls B nichtleer ist.} \end{cases}$$

Beispiel: Abb. 10 zeigt einen Baum. Ein Teilbaum der Wurzel ist schraffiert.

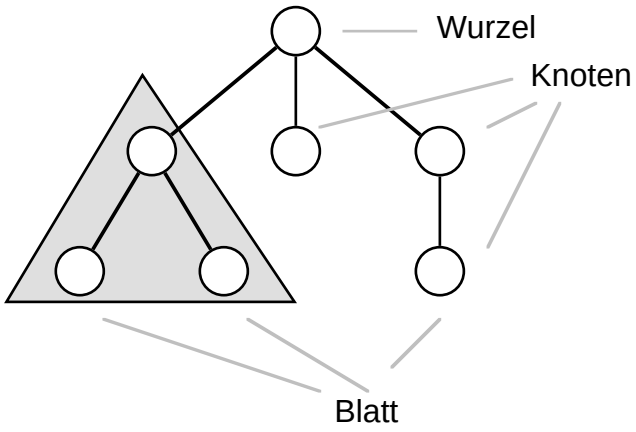


Abb. 10: Beispiel für einen Baum

Bäume verdeutlichen zunächst allgemeine hierarchische Strukturen. Zur Darstellung konkreter Strukturen versieht man die Knoten mit Markierungen. Dies können beliebige Informationen aus einem Datentyp sein.

Definition F:
Sei M eine beliebige Menge. Ein **markierter** geordneter Baum über M ist ein geordneter Baum, dessen Knoten jeweils mit einem Element aus M markiert sind.

Beispiel: Abb. 9 zeigt einen markierten geordneten Baum. Die Markierungsmenge ist hier M={HAL,AL,GL,M}.

Eine besonders wichtige Rolle in der Informatik spielen die sog. **binären Bäume**.

Definition G:
Ein **binärer Baum** ist ein markierter geordneter Baum, in dem jeder Knoten höchstens zwei Söhne hat. Man spricht dann vom **linken Sohn** und vom **rechten Sohn**.

Wie definieren wir Bäume als Datentypen? Die rekursive Definition D gibt uns einen unmittelbaren Hinweis: Wir haben in der Datentypdefinition festzulegen, wie ein leerer Baum aussieht, und wir müssen definieren, wie man aus zwei vorhandenen Bäumen ei-

nen größeren durch Hinzunahme eines Knotens erhält. Problematisch erscheint es zunächst, die zweidimensionale Darstellung von Bäumen in eine lineare Folge von Zeichen in der Typdefinition abzubilden. Wir bedienen uns hier wieder eines schon häufig verwendeten "Tricks", indem wir zwar zu einer linearen Darstellung übergehen, diese aber so geschickt wählen, daß sie *isomorph* zu den darzustellenden Bäumen ist. Zur Veranschaulichung betrachte man Abb. 9. Der Hauptabteilungsleiter ist den beiden Abteilungsleitern übergeordnet. Diese Situation kann auch durch

(HAL,AL,AL)

oder – genauso zulässig, aber weniger logisch – durch

(AL,AL,HAL)

dargestellt werden. Wir verfolgen die erste Notation. Nun ist weiter der eine Abteilungsleiter Chef zweier, der andere Abteilungsleiter Chef dreier Gruppenleiter. Also:

(AL,GL,GL) bzw. (AL,GL,GL,GL).

Schließlich zwischen Gruppenleiter und Mitarbeiter der Reihe nach:

(GL,M,M,M) bzw. (GL,M,M) bzw. (GL,M,M,M,M)
bzw. (GL,M,M,M) bzw. (GL,M,M).

Nun haben wir bereits Teilbäume linear dargestellt. Eine Gesamtdarstellung des Baumes erhalten wir, indem wir die einzelnen Teilbeschreibungen ineinandereinsetzen, wobei wir die entsprechenden Personen gem. Abb. 11 identifizieren. Ergebnis:

(HAL,(AL,(GL,M,M,M),(GL,M,M)),(AL,(GL,M,M,M,M),(GL,M,M,M),(GL,M,M))).

Diese Zeichenfolge stellt den Baum aus Abb. 9 eindeutig dar. Man kann Abb. 9 aus dieser Zeichenfolge zurückgewinnen.

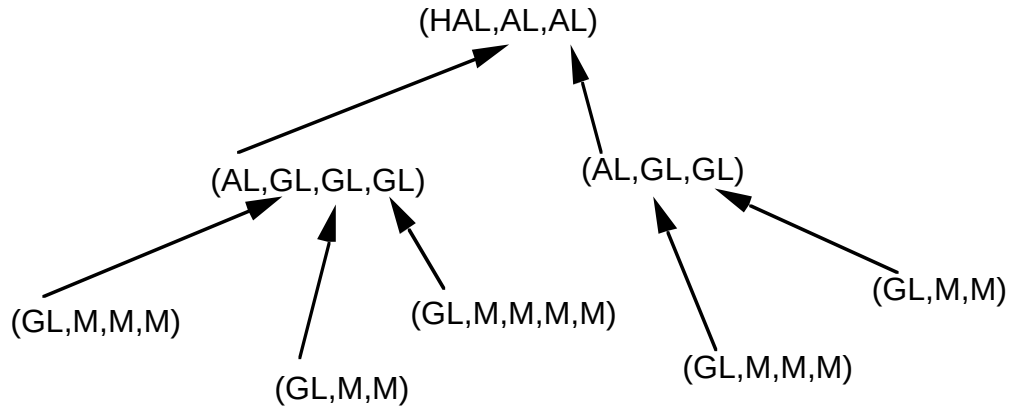


Abb. 11: Übergang zu einer linearen Darstellung eines Baumes

Den gleichen Ansatz verfolgen wir bei der Typdefinition eines zunächst unmarkierten binären Baumes und setzen:

typ Baum ≡ {leer} | (Baum,Baum).

In dieser Definition wird die Struktur des Baumes allein durch Klammerstruktur realisiert.

Beispiel: Abb. 12 zeigt drei Bäume und ihre zugehörige Darstellung als Objekt des Typs Baum.

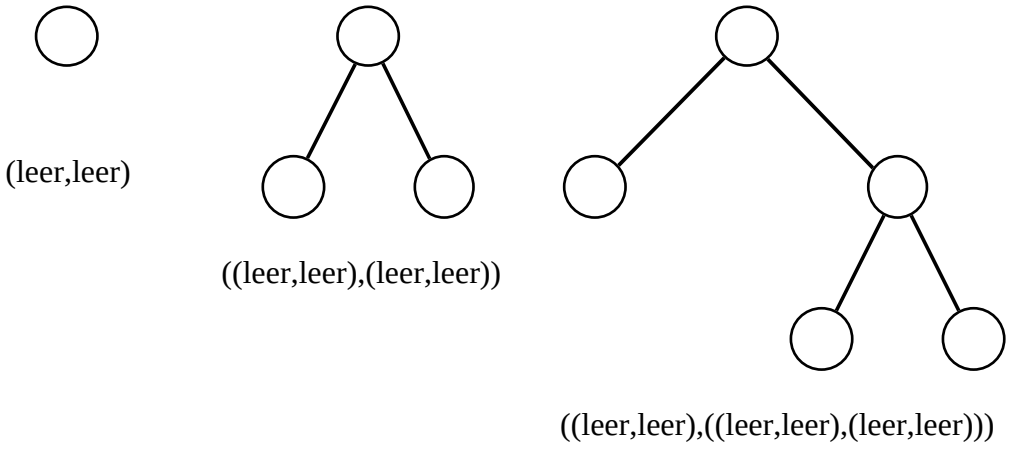


Abb. 12: Bäume und ihre lineare Darstellung

Übersichtlicher wird das Prinzip der linearen Darstellung von Bäumen, wenn man markierte binäre Bäume definiert. Sei D ein beliebiger Datentyp, dessen Werte als mögliche Markierungen verwendet werden sollen. Wir definieren:

typ DBaum ≡ {leer} | (DBaum,D,DBaum).

Die rechte Seite (DBaum,D,DBaum) symbolisiert hier einen Knoten, der mit einem Objekt aus D markiert ist und dessen linker und rechter Sohn jeweils Bäume über D sind.

Beispiel: Wir definieren einen markierten binären Baum, der zum Entschlüsseln von Morsezeichen verwendet werden kann (Abb. 13). Dazu wird der Morsecode so im Baum codiert, daß man beim Durchlaufen des Baumes von der Wurzel aus zu einem Knoten aus dem zurückgelegten Weg den Morsecode ablesen kann. Der Baum besitzt dazu folgende Eigenschaften:

1) Jeder Buchstabe kommt als Markierung eines Knotens im Baum vor.
2) Jede Kante zu einem linken Sohn ist als Morsezeichen "Punkt" zu interpretieren.
3) Jede Kante zu einem rechten Sohn ist als Morsezeichen "Strich" zu interpretieren.

So führt z.B. der Durchlauf von der Wurzel zum Buchstaben r über den Weg links/rechts/links und damit zum Morsezeichen **.-.**.

Bei der Datentypdefinition ist gegenüber den bisherigen Definitionen zu berücksichtigen, daß die Wurzel als Sonderfall keine Markierung trägt und daß ein Morsebaum nicht leer sein kann:

typ Zeichen ≡ char['a'..'z'];
typ MorseBaum ≡ (MB,{wurzel},MB);
typ MB ≡ {leer} | (MB,Zeichen,MB).

Der gesuchte Morsebaum ist nun ein Objekt vom Typ MorseBaum, aber nicht das einzige.

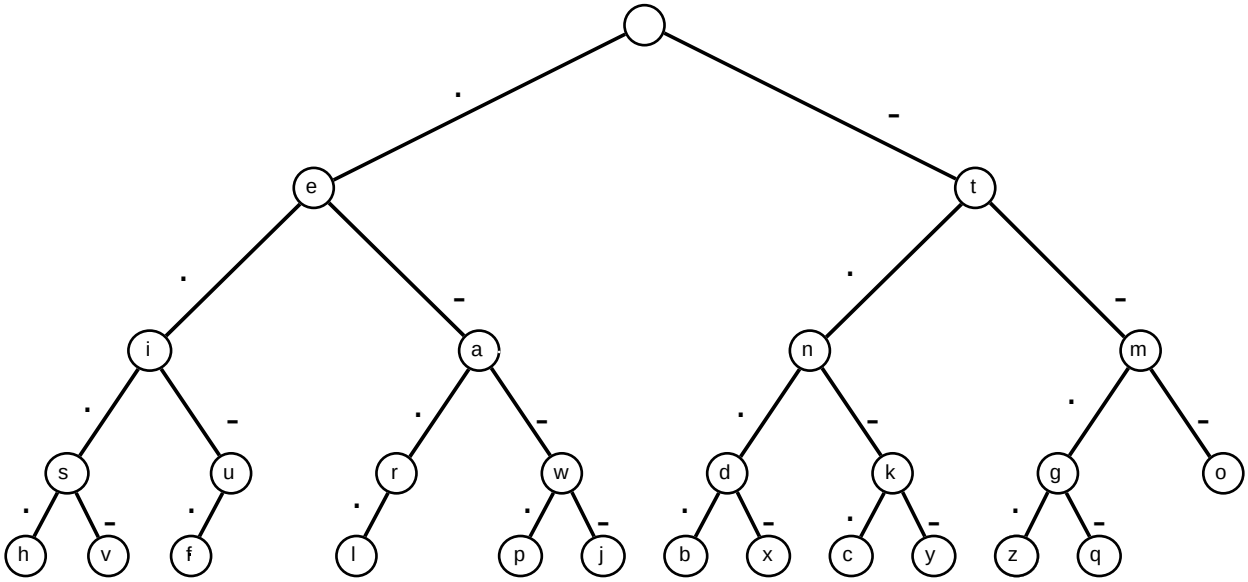


Abb. 13: Morsebaum

Eine nochmals erweiterte Form markierter Bäume erhält man, wenn man für die inneren Knoten und für die Blätter unterschiedliche Markierungsmengen zuläßt. Wir definieren einen Baum mit dem Typ D für die Markierungen der inneren Knoten und dem Typ E für die Markierungen der Blätter:

typ DEBaum ≡ E | (DEBaum,D,DEBaum).

Beispiel: Diese Form markierter Bäume benutzt man u.a. im Zusammenhang mit arithmetischen Ausdrücken. Einen arithmetischen Ausdruck kann man als Baum darstellen, indem man wie oben entsprechend der Klammerstrukturen und der Prioritäten der arithmetischen Operationen eine hierarchische Anordnung der Teilausdrücke vornimmt. Zum Beispiel wird der Ausdruck

(a+b)*(c-d)

durch den Baum in Abb. 14 dargestellt. Man erkennt dort insbesondere folgende Merkmale:

- Blätter sind mit Operanden, innere Knoten mit Operationen markiert,

- Klammern entfallen; ihre Funktion wird durch die unterschiedliche hierarchische Anordnung der Operanden und Teilausdrücke realisiert,
- durch die hierarchische Anordnung ist zugleich die Reihenfolge der Auswertung vorgegeben, von den Blättern zur Wurzel.

Den zugehörigen Datentyp der arithmetischen Ausdrücke über den vier Grundrechenarten, wobei als Operanden nur einbuchstabige Bezeichner verwendet werden, definiert man durch:

typ Bezeichner ≡ char['a'..'z'];
typ Operation ≡ {+,-,*,/};
typ Ausdruck ≡ Bezeichner | (Ausdruck,Operation,Ausdruck).

Der Baum aus Abb. 14 ist ein Objekt vom Typ Ausdruck und besitzt die Darstellung:

((a,+,b),*,(a,-,b)).

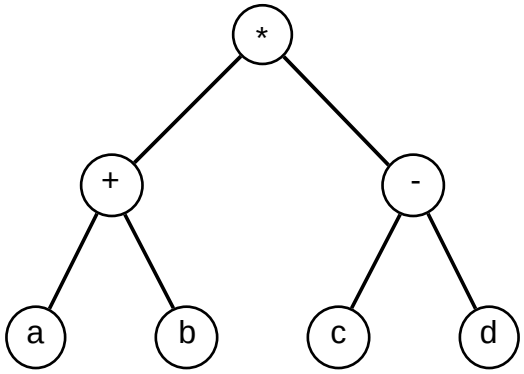


Abb. 14: Arithmetischer Ausdruck als Baum

9.4.7 Bildung von Funktionenräumen

Dieser Datentypkonstruktor bewirkt den Übergang von zwei Datentypen D' und D'' zum **Funktionenraum** D, d.h. zur Menge aller Abbildungen f: D'→D''. Allgemein definiert man

typ D ≡ [D'→D''].

Beispiele:

1) typ boolFunkt ≡ [(int, int)→bool].

   Der Wertebereich von boolFunkt ist die Menge aller Funktionen, die als Argument ein Paar von ganzen Zahlen besitzen und als Ergebnis einen Wahrheitswert liefern. Zu boolFunkt gehören z.B. die Vergleichsoperationen =, ≠, ≤ usw.

2) typ intFunkt ≡ [(int, int)→int].

Der Wertebereich von intFunkt ist die Menge aller Funktionen, die als Argument ein Paar von ganzen Zahlen besitzen und als Ergebnis eine ganze Zahl liefern. Zu intFunkt gehören z.B. die Rechenoperationen +, -, * usw.

Funktionenräume sind in Programmiersprachen oft nur sehr eingeschränkt verwirklicht, da ihre Realisierung auf große Probleme stößt. In vielen Programmiersprachen, z.B. PASCAL, ist der Funktionenraum überhaupt nicht zugänglich, man kann nur einzelne feste Funktionen definieren. In der Programmiersprache ML, die wir im Laufe der Vorlesung kennenlernen, ist das Funktionenraumkonzept jedoch weitgehend in der hier beschriebenen Form realisiert.

Funktionenräume bilden ein sehr allgemeines Konzept, dem sich alle anderen Datentypen unterordnen. Denn prinzipiell läßt sich jedes Objekt als Funktion mit geeigneten Parametern auffassen. Zum Beispiel ist die Konstante 3 zu interpretieren als nullstellige Abbildung mit der Bezeichnung $\mathbf{3}$ von einer einelementigen Menge {()}, die nur das leere Tupel enthält, in die Menge der ganzen Zahlen int, also

$$\mathbf{3}: \{()\} \rightarrow \text{int} \quad \text{mit}$$
$$\mathbf{3}() = 3.$$

Bezeichnen wir den Typ {()} mit unit, so läßt sich also jeder Datentyp D als Funktionenraum

$$[\text{unit} \rightarrow D]$$

auffassen, und für jedes $d \in D$ gibt es in $[\text{unit} \rightarrow D]$ eine Funktion $\mathbf{d}$ mit $\mathbf{d}() = d$. Auf diese Weise hat man alle Datentypkonzepte auf Funktionen zurückgespielt und die Funktionenraumbildung als den universellsten Konstruktor identifiziert.

Wegen dieser universellen Eigenschaft von Funktionen, mit der wir zugleich den Bogen von Daten hin zu den Operationen geschlagen haben, und der vielen Phänomene, die mit der Definition und Benutzung von Funktionen verbunden sind, widmen wir uns dem Gebiet intensiv im folgenden Kapitel.