1 Datentypen

Aufgabe:
Gegenstände der realen Welt so durch geeignete Strukturen zu modellieren, daß sie algorithmisch möglichst einfach und effizient zu verarbeiten sind.

1.1 Grundbegriffe

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.

Eine Möglichkeit: Nimm den "Datenbrei" als solches hin – einige Programmiersprachen tun das auch

Weitere Möglichkeit: Präge Daten eine Struktur auf.
Strukturierungskriterium: Operationen, die mit den Daten möglich sind.

Datenobjekte, auf denen die gleiche Menge von Operationen möglich ist und die sich strukturell gleich verhalten, faßt man zu einem *Datentyp* zusammen.

=> alle Werte eines Datentyps haben das gleiche operationale Verhalten.

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.

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: Persönlicher Geschmack und Sprachniveau bestimmen, was man als elementar betrachtet.

Notationen: IN, $\mathbb{Z}$ und IR
$\mathbb{A}$={'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:

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

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

Fundamentale Idee der Informatik: Baukastenprinzip

- wenige Grundbausteine, die ***elementaren Datentypen***,
- 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).

Formalisierung Baukasten:

- Δ Klasse aller Datentypen
- Δ_e Klasse der elementaren Datentypen.
- Für i=1,...,n sei

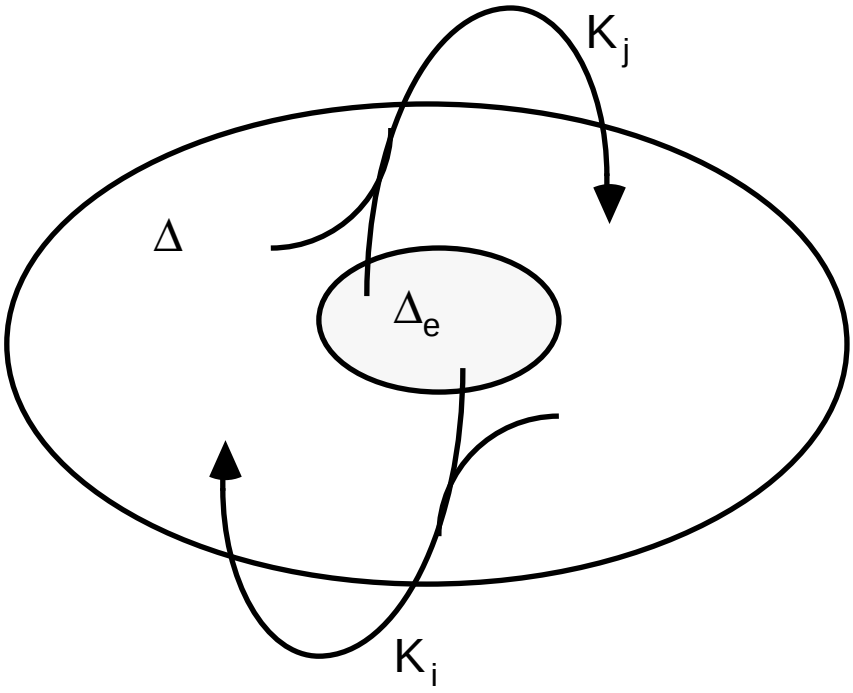
  $K_j: \Delta\times...\times\Delta\rightarrow\Delta$

  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*.



Informatik *Standardbaukasten* $B_0=(\Delta_0,K_0)$

- praktisch allen Programmiersprachen integriert
- sechs elementare Datentypen und sieben Konstruktoren:

  Δ_0={nat,int,real,char,text,bool},

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

Bedeutung:

- *Enumeration*: Bildung eines elementaren Datentyps durch Aufzählung aller seiner Elemente,
- *Restriktion*: Bildung einer Teilmenge eines Datentyps,
- *Aggregation*: gleichrangiges Zusammensetzen mehrerer Datentypen zu einem einzigen (kartesisches Produkt),
- *Generalisation*: Vereinigung von disjunkten Datentypen zu einem einzigen,
- *Rekursion*: Übergang zu einem Datentyp mit abzählbar unendlich vielen Elementen, die gleichartig aus einfacheren Elementen desselben Typs aufgebaut sind,
- *Potenzmengenbildung*: Bildung der Menge aller (endlichen) Teilmengen eines Datentyps,
- *Bildung von Funktionenräumen*: Übergang von zwei Datentypen zum Datentyp, dessen Wertemenge die Menge aller Funktionen zwischen diesen Datentypen ist.

Thema im folgenden: Δ_0 und K_0.

Selektoren/Destruktoren:
implizite Mitdefinition von Funktionen, die den Konstruktionsprozeß wieder rückgängig machen und es ermöglichen, einen in einer Datenstruktur versteckten elementaren Wert "wieder aufzuspüren".
=> *Destruktoren* oder besser *Selektoren*.

Beispiel:
Konstruktor (,...,) für Tupelbildung
Projektion auf die i-te Komponente als Destruktor/Selektor

1.2 Elementare Datentypen

Im allen Programmiersprachen: int, nat, real, char, bool und text.
(*Standarddatentypen*)

Merkmale:
- endliche Wertemengen
- lineare Ordnung <. Ist also $M=\{m_1,m_2,m_3,...,m_n\}$ die Wertemenge eines dieser Datentypen, so gilt
  $m_1<m_2<m_3<...<m_n$.

(*skalare Datentypen*)

Zubehör: *Standardoperationen* oder *Standardfunktionen*
Notationen einer Funktion f:
Infixnotation, z.B. xfy (z.B. x+y) (meist Operation genannt)
Präfixnotation, f(x,y) (z.B. sin(x)) (meist Funktion genannt)
Postfixnotation, z.B. xyf (z.B. x!, Fakultät).

Vordefinierte Standardfunktionen für alle skalaren Datentypen (außer real):
Funktionen pred und succ.
Vorgängerfunktion (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

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

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

Weitere Standardfunktion für alle skalaren Datentypen (mit Ausnahme von real, int und nat):

ord,

bildet jedes Element des Datentyps auf die natürliche Zahl abbildet, die die Position des Elements innerhalb der Ordnung angibt.

Präziser: Sei $M=\{m_1,m_2,m_3,...,m_n\}$ Wertemenge eines skalaren Datentyps (außer real und int):

ord: $M \rightarrow IN$ mit
$ord(m_i)=i$.

Weitere Standardfunktionen auf allen skalaren Datentypen:

Vergleichsoperationen.

Der Datentyp bool.

Wertemenge zweielementig {false (dt. falsch),true (dt. wahr)}, sog. Boolesche Werte oder Wahrheitswerte.

lineare Ordnung

false < true.

Z.B.

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

Operationen:

zwei 2-stellige Operationen and und or, sowie eine 1-stellige Operation not
Wertetabelle:

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.

Vergleichsoperationen, deren Bildmenge der Datentyp bool ist:

<, ≤, ≥, >, =, ≠.

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.

Wertemenge von int: *endliche* Teilmenge der ganzen Zahlen $\mathbb{Z}$ in Form eines symmetrischen Intervalls um den Nullpunkt, d.h.

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

maxint computer- und programmiersprachenabhängig.
Kleincomputer z.B.: maxint = $2^{15}-1=32767$
Großcomputer: maxint = $2^{31}-1=2.147.483.647$.

Konstanten: Dezimaldarstellung als Folge von Ziffern mit oder ohne Vorzeichen + oder -.

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.

Problematik: Rechengenauigkeit

int nur *Teilmenge* der ganzen Zahlen => rechengesetze gelten nicht mehr

Beispiel: maxint=100.

(60 + 50) - 30 => *Überlauf* (engl. overflow),
60 + (50 - 30)=80

Das Assoziativgesetz der Addition verletzt.

einstellige Standardfunktionen:

abs(x) (Absolutbetrag), sqr(x) (Quadrat) und
odd(x) (Feststellung der Ungeradzahligkeit).

Der Datentyp nat.

Alle Eigenschaften von int übertragen sich sinngemäß.

Der Datentyp real.

Wertebereich: grundsätzlich alle reellen Zahlen.
Tatsächlich Beschränkung der Wertemenge auf Zahlen in sog. **Gleitpunktdarstellung**

Gleitpunktdarstellung: 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, $e \in \mathbb{Z}$, $b \geq 2$ und $1/b \leq m < 1$.

schreiben.

m **Mantisse** (Folge der gültigen Ziffern) von z
e **Exponent** (Größenordnung der Zahl)
b **Basis**.

In der Praxis:

b = 2, 10 oder 16.

Normalisierung: $1/b \leq m < 1$

Beispiele: Normalisierte Gleitpunktdarstellungen zur Basis 10:

$189{,}217 = 0.189217 \cdot 10^3$,
$-2{,}4 = -0.24 \cdot 10^1$,
$-0{,}0013 = -0.13 \cdot 10^{-2}$.

Problematik: Rechenregeln.

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

Auslöschung erzeugt **Rundungsfehler**.

Konstanten: übliche Dezimaldarstellung als Folge von Ziffern mit oder ohne Dezimalpunkt, gefolgt vom Exponententeil, eingeleitet durch Buchstaben E.
Beispiele: Konstanten vom Typ real sind:

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

Operationen:

- bekannte arithmetische Operationen +, -, * und / (Division)
- Vergleichsoperationen
- weitere 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)

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.

Wertebereich char: alle darstellbaren Zeichen des Zeichensatzes einer Programmiersprache sowie Steuerzeichen.

text = char*

(meist Längenbeschränkung auf 2^8 oder 2^{16}).

Konstanten: in Hochkommata (bei char) oder Anführungszeichen (bei text). Will man das Hochkomma (Anführungszeichen) selbst darstellen, so verdoppelt man es.

Beispiel: Konstanten vom Typ char:

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

Konstanten vom Typ text:

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

Operationen: für char

pred, succ, ord, Vergleiche

Funktion chr als Umkehrung der ord-Funkton:

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.

Resultat der ord-Funktion abhängig von interner Verschlüsselung der Zeichen, z.B. **ASCII-Code**.

Operationen: für text

Vergleichsoperationen <,=,≠,>,≤,≥, bezgl. lexikographischer Ordnung, also

"Auto"<"Autobahn" usw.

Konkatenation • zweier Texte zu einem:

"Baum"•"Haus"="BaumHaus".

1.3 Probleme der Typisierung

Beispiel:

100 ist eine reelle Zahl
100 ist eine ganze
100 ist eine natürliche Zahl
100 ist ein Gasverbrauch (mit gedachter Dimension m^3)
100 ist die Nummer eines InterCity-Zugs
100 ist Teil eines Autokennzeichens.

Sinnvolle Operationen:

- in den ersten drei Fällen alle arithmetischen Operationen
- im vierten Fall nur Operationen mit solchen anderen Werten, die zu einem im Zusammenhang mit dem Gasverbrauch sinnvollen Ergebnis führen. Z.B. Addition eines Gasverbrauchs 100 (in m^3) und eines Stromverbrauchs (in kWh) sinnlos.
- in den beiden letzten Fällen alle arithmetischen Operationen sinnlos. Z.B. sinnvolle Operation auf InterCity-Nummern „Liefere die Nummer des Gegenzugs“.

Beobachtung:
viele Objekte besitzen die gleiche Gestalt (wie die 100), gehören jedoch zu unterschiedlichen Typen.
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.

Zwei zueinander konträre Lösungsmöglichkeiten

- strenge Typisierung
- 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**.

Wertemengen aller Datentypen einer streng typisierten Sprache sind paarweise disjunkt.
Auf die Objekte eines Typs können nur diejenigen Operationen angewendet werden, die für den Typ definiert sind, und sonst keine.

***Beispiel**:* In einer streng typisierten Sprache sind folgende Konstruktionen *nicht* erlaubt, in einer schwach typisierten sind sie erlaubt:
- sin(7), da sin eine Funktion von real nach real ist, 7 jedoch ein int-Ausdruck ist;
- 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.
  Behelf durch **Typkonversion**:
  makereal: int→real
  Anschließend:12.5*makereal(17).
  Umkehrfunktion allgemein
  makeint: real→int
  Einige Programmiersprachen enthalten automatische Typkonversionen;

Vorteile der strengen Typisierung:
- Sicherheit bei Software-Entwicklung, da Typverletzungen bereits durch den Übersetzer erkannt und entsprechende Programme vom System abgewiesen werden.
- 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*.

LISP klassische Programmiersprache mit dynamischer Typisierung

Vorteile statische Typpisierung:
- Lesbarkeit und Übersichtlichkeit der Programme,
- Steigerung der Effizienz, da in das übersetzte Programm kein Code für die Typprüfungen eingebunden werden muß,
- leistungsfähige *Typinferenzsysteme* bereit, die aus den meisten Ausdrücken den zugehörigen Datentyp ableiten können, so daß Typdeklarationen in vielen Fällen unterbleiben können.

In diesem Kapitel immer:
- strenge Typisierung.

1.4 Konstruktoren

1.4.1 Enumeration

Einführung endlicher linear geordneter Mengen als neue elementare Datentypen durch Aufzählung aller zugehörigen Elemente, sog. *Aufzählungstyp*.
Schematisch:

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

Hierbei ist jedes Element d_i explizit anzugeben.

Strenge Typisierung => D ist ein *neuer* Typ ist, dessen Wertemenge disjunkt von der Wertemenge aller übrigen Typen ist.

Beispiel: Das Element -7 vom Typ $\mathbb{Z}$ 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.

Reihenfolge der Elemente $d_1,...,d_n$ definiert *lineare Ordnung* < auf D mit:

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

Aufzählungstypen sind also skalare Datentypen.

Beispiel: Für den Aufzählungstyp $\underline{\text{typ}}\ D \equiv$ (rot, grün, blau) gilt rot<grün<blau.

Operationen:

- pred und succ.
- Vergleichsoperatoren entsprechend der linearen Ordnung.

1.4.2 Restriktion

Übergang zu einer (endlichen oder unendlichen) Teilmenge eines bereits definierten Datentyps.

Allgemeinster Fall: Restriktion durch Angabe eines Prädikats, also einer Abbildung

P: M→{true, false}.

Schematisch:

typ D' ≡ D {x | P(x)}.

P ist ein Prädikat.

Wertemenge von D' = Menge aller x vom Typ D, die das Prädikat P erfüllen.

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

Beispiel: typ Zahl ≡ int {x | true}

Trotz strenger Typisierung: D' ***kein*** neuer Typ, d.h., Operationen auf D können auf Operanden aus D' angewendet werden. D und D' sind also nicht disjunkt.

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

Wertemenge alle geraden ganzen Zahlen.

Sehr allgemeines Konzept meist nicht realisiert:

Spezialfälle:

a) *Restriktion durch Intervallbildung:* Einschränkung des Grundtyps auf ein zusammenhängendes Intervall:

typ D' ≡ D [a ..b].

D skalarer Typ. Wertemenge von D' = die Menge aller $x \in D$ mit $a \leq x \leq b$.
Lineare Ordnung für D überträgt sich auf D'.
Beispiel: typ Lottozahlen ≡ int [1..49].

b) *Restriktion durch Enumeration:* Aufzählung der gewünschten Elemente der Teilmenge:

typ D' ≡ D $\{d_1,d_2,...,d_n\}$,

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

1.4.3 Potenzmengenbildung

Übergang zur Menge aller Teilmengen von D'. Schema:

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

Beispiele:

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

Wertemenge von zahlen = Menge aller Teilmengen von int.

2) typ $grundfarbe \equiv \{rot,gelb,blau\}$
typ $farben \equiv 2^{grundfarbe}$.

Objekte von Typ farben:

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

Interpretation: Mischen der drei Grundfarben gebildet werden können.

Universelle Konstante: leere Menge $\emptyset$.

Operationen: übliche Mengenoperationen:

$\cup$: $D \times D \to D$ (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),
$\subseteq$: $D \times D \to \{true, false\}$ (Teilmenge),
$\in$: $D' \times D \to \{true, false\}$ (Element-Beziehung).

In Programmiersprachen:
meist Beschränkung auf endliche Datentypen als Grundtyp D'
=> jedes Objekt des Typs D ist endliche Menge von Objekten des Datentyps D'.

1.4.4 Aggregation

Zusammensetzen von mehreren (möglicherweise verschiedenen) Datentypen $D_1,D_2,...,D_n$ zu einem n-Tupel. Schematisch

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

Mathematisch ist D das kartesische Produkt der Typen $D_1,...,D_n$.

Bezeichnungen:

$D_1,...,D_n$: **Komponenten** von D.

Elemente von D sind n-Tupel $d=(d_1,d_2,...,d_n)$ mit $d_i \in D_i$.

$D_1=D_2=...=D_n$ => D **homogen**, anderenfalls **inhomogen**.

Beispiel: $\underline{typ}$ Datum ≡ (int [1..31], int [1..12], int [1900..1999]).

Operationen:

Selektoren

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

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

$\pi_{i,n}$ **Projektion** von d auf die i-te Komponente.

Zusammenhang zwischen Konstruktor und Selektor:

Konstruktionsaxiom

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

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.

Iteration der Selektion bei geschachtelten aggregierten Typen:

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

Beispiel:

$\underline{typ}$ Schulzeit ≡ (Datum,Datum).

Tag des Abgangsdatums:

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

Für $d=((17,8,1960),(22,4,1976)) \in$ Schulzeit ist

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

parallele Selektion: gleichzeitige Anwendung mehrerer Selektoren auf das gleiche Element:

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

Anwendung in der Programmierung:

Durchlaufen aller Komponenten eines Objektes $d \in D$ in der Reihenfolge $d_1, d_2, d_3, ...$.

Definiere *Nachfolgerfunktion* N auf der Menge der Selektoren:

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

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

Beispiel: Ausdruck aller Komponenten des Objektes d vom Typ D:

```
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.

Erweiterung: **Verbund** oder **Record**.

Verwende Bezeichner statt Projektionen.

Schema:

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

$t_1,...,t_n$ paarweise verschiedene (!) Bezeichner (*Selektoren*)

$T_1,...,T_n$ beliebige Datentypen.

Wertemenge: Menge aller möglichen n-Tupel der Form $(a_1,a_2,...,a_n)$, wobei jedes a_i vom Datentyp T_i ist und über den Bezeichner t_i angesprochen werden kann.

Beispiel: Personaldaten:

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

Oder besser:

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

Einfügen dieses Typs:

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

Selektion: *dot-Notation* (engl. dot = Punkt).

$v.t_i$ i-te Komponente des Verbunds

v meist nur ein einzelner Bezeichner aber *kein* Ausdruck sein.

Beispiel:

```
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.
```

Beachte: innerhalb eines Verbundes verwendete Bezeichner *untereinander* verschieden:

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

Benutzung:

```
x
y.x
y
y.y.
```

Abkürzung bei mehrfachem Zugriff die Komponenten der gleichen Verbundvariablen durch

inspect ... tue ...

ständiges Voranstellen der Verbundvariablen vor die Komponentenbezeichner (mit Punkt dazwischen) entfällt.

Beispiel:

```
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 V_1, V_2, ..., V_n tue ...

Beispiel:

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

1.4.4.1 Homogene Aggregation in imperativen Sprachen

Konstruktor array (Feld)
Zusammenfassung einer beliebigen vorher festzulegenden Anzahl von Daten gleichen Datentyps unter einem Bezeichner

Zugriff (*Selektion*) erfolgt über **Index**

Voraussetzung:

Indextyp I

Grundtyp T.

Indextypen sind oft Restriktionen von int.

Definitionsschema:

typ A ≡ array I of T.

Wertemenge:

$\{i_1,i_2,...,i_n\}$ Wertemenge von I mit der Ordnung $i_1<i_2<...<i_n$.

Wertemenge von A = Menge aller n-Tupel

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

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.

Selektion in der Form a_i.

Schreibweise:

a(E)

E Ausdruck vom Indextyp I und Wert i.

a(E): das zu i gehörende Element des Tupels, also a_i, **i-te Komponente** von a.

Oft eckige Klammern a[E] statt a(E).

Beispiele:

1) Komponentenweise Addition zwei Felder a und b mit je fünf Elementen vom Typ int:

```
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) Einlesen eines Textes mit 100 Buchstaben
   Ausgabe der Häufigkeit für jeden der Buchstaben a, b, c, d oder e, mit der er im eingelesenen Text auftritt:

```
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.
```

Verallgemeinerung:

$\underline{\text{typ}}$ 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

Abkürzung:

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

Analog

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

durch

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

abkürzen.

Dimension: Anzahl der in einer Felddeklaration aufgeführten Indextypen.
Felder der Dimension 1: **1-dimensional** oder **linear** (Vektoren)
Felder der Dimension 2: **2-dimensional** (Matrizen) usw.

1.4.5 Generalisation

Vereinigung disjunkter Datentypen zu einem *neuen* Datentyp. Schema:

typ $D \equiv D_1 \mid D_2 \mid ... \mid D_n$

wobei $D_i \cap D_j = \emptyset$ für alle $1 \leq i < j \leq n$ gilt.

$D_1, D_2, ..., D_n$ heißen *Varianten* von D.

n=1 -> Umbenennung eines Datentyps.

Beispiele:

1) Darstellung einer Geraden im Koordinatensystem:
   - Normalform ax+by+c=0,
   - Punkt-Steigungsform $y=m(x-x_0)+y_0$,
   - Zwei-Punkte-Form $(y-y_0)/(x-x_0)=(y_1-y_0)/(x_1-x_0)$.

   Definiere:

   typ punkt ≡ (real,real);
   typ steigung ≡ real;
   typ gerade ≡ (punkt,punkt) | (punkt,steigung) | (real,real,real).
2) Ergänzung von Datentypen um Fehlerelemente.
   Definiere ⊥ (undefiniert):

   typ intplus ≡ int | {undef};

Beachte: strenge Typisierung => D ist nicht mengentheoretische Vereinigung der disjunkten Typen $D_1, ..., D_n$.

Stattdessen: Kopiere Typen $D_1, ..., D_n$, vereinige Kopien, identifiziere Vereinigung mit D.

Wertemenge von D **formal verschieden** von der Vereinigung $D_1 \cup D_2 \cup ... \cup D_n$.

Konsequenz für die Operationen: 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.

Wunschvorstellung dabei: Operationen, die man für D oder für $D_1, ..., D_n$ definiert hat, seien genau auf die jeweiligen Typen beschränkt (meist jedoch automatische *Typkonversion*).

Beispiel: einfachster Fall für n=1:

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.

Standardoperation: in: zu welcher Variante gehört ein Objekt x. Es gilt:

$$x \text{ in } D = \begin{cases} \text{true, falls } x \text{ vom Datentyp } D \text{ ist,} \\ \text{false, sonst.} \end{cases}$$

Beispiel: Sei zählerstand vom Typ gasverbrauch. Dann:

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

Beispiele:

1)

```
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) Personendaten eines Standesamtes:

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));
```

1.4.6 Rekursion

Bisher: nur endliche Strukturen mit möglicherweise großen, aber *endlichen* Wertebereichen.

Neu: Rekursion (von lat. recurrere=zurückführen) bewirkt Übergang zu *abzählbar unendlichen* Wertebereichen, deren Elemente gleichartig aus einfacheren Elementen eines Datentyps aufgebaut sind.
Formulierung **unendlicher** Sachverhalte durch **endliche** Beschreibung.

Schema:

typ D ≡ T | D'.

- T und D' Datentypen.
- T ist der sog. *terminale* (*atomare*) Datentyp, auf den sich die Definition (schließlich) abstützt,
- D' irgendeinen Typ, in dessen Definition wieder D vorkommt.
- "|" bekanntes Zeichen zur Generalisation.

Zur Bedeutung von D
Annahme: D' hat also die Form

... D ... D ...

Wertemenge von D ist dann disjunkte Vereinigung

D=T∪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∪D'=T∪(... D ... D ...)
=T∪(... T ... T ...)∪(... T ... D' ...)∪(... D' ... T ...)∪(... D' ... D' ...)
=T∪(... T ... T ...)∪(... T ... (... D ... D ...) ...)∪(... (... D ... D ...) ... T ...)
∪(... (... D ... D ...) ... (... D ... D ...) ...).

Wertebereich von D 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

typ D ≡ T | D'

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

typ D ≡ T | D_1;
typ D_1 ≡ T_1 | D_2;
…
typ D_{n-1} ≡ T_n | 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) Texte:
   nichtleerer Text ist entweder ein einzelnes Zeichen, oder er besteht aus einem einzelnen Zeichen, dem ein Text folgt. Daher:

   typ X ≡ char | (char,X).

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

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

   X beschreibt Vereinigung aller folgenden Datentypen:

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

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

   Wertemenge von X = Menge aller nichtleeren Zeichenfolgen über char (ignoriere Klammern und Kommata).
   Hinzunahme des leeren Textes:

   typ X' ≡ {eps} | (char,X').

   Der leere Text wir durch das einzige Element des enumerierten Typs {eps} erfaßt.

Wertemenge von X' isomorph zur Wertemenge des Standarddatentyps text. Beispiele für Objekte vom Typ X' sind:

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

Beachte: {eps} terminaler Typ.

2) Allgemein **Linkssequenz**:

typ L ≡ {eps} | (D,L)

„**links“**: an eine gegebene Sequenz kann links ein neues Element angefügt werden.

eps *leere Sequenz*, *leeres Wort*.

L Vereinigung aller Typen der Form

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

X aus Beispiel 1: Linkssequenz über char.

Analog **Rechtssequenz**:

typ R ≡ {eps} | (R,D).

R Vereinigung aller Typen der Form

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

3) Anwendung für die indirekte Rekursion: einfache arithmetische Ausdrücke 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:

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

Beachte: Klammern "(" und ")" sowohl für die Aggregation, als auch als Zeichen in Termen.

Spezielle rekursive Datentypen: Files und Bäume.

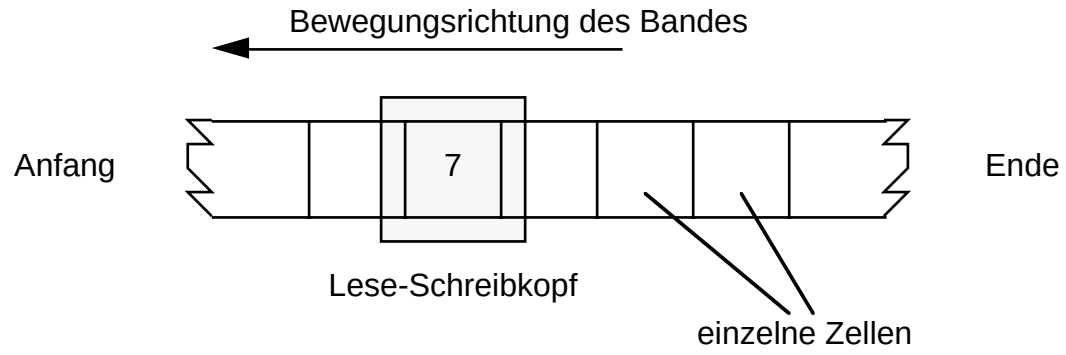
1.4.6.1 Files

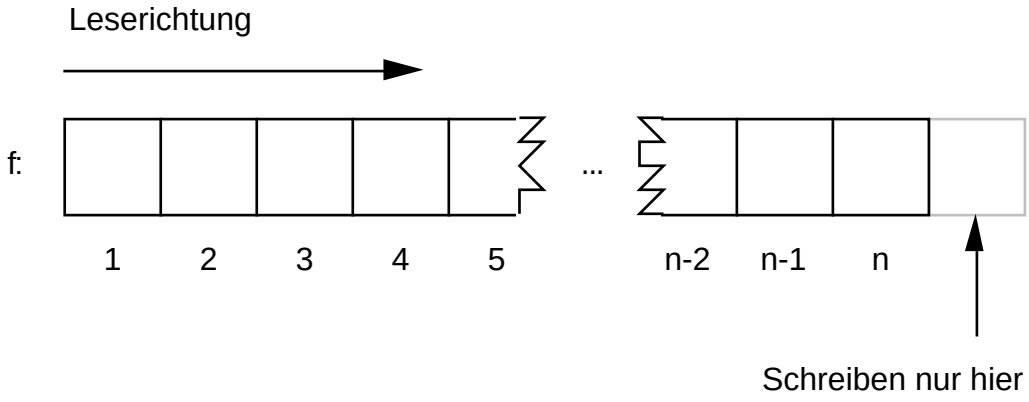
"File" = Kartei, Datei.

File -> Kombination von Links- und Rechtssequenz über einem Datentyp D mit Operationen get, move, put, eof, reset und rewrite.

Veranschaulichung:

- Magnetband (Tonband), einzelne Zellen unterteilt.
- Je Zelle genau ein Element des zugrundeliegenden Datentyps D
- Bearbeitung durch Lese-Schreibkopf
- Lesen und Schreiben einer Zelle, wenn sich die Zelle unter dem Lese-Schreibkopf befindet.
- Kopf als Sichtfenster, das zu jedem Zeitpunkt nur immer genau eine Zelle zeigt.
- *Lesender* Zugriff (die *Selektion*) beginnt grundsätzlich beim ersten Element.
- Lesen des n-ten Elements, wenn zuvor alle n-1 vorangehenden Elemente gelesen wurden, sog. **sequentieller Zugriff**.
- *Schreiben* nur am Schluß des Files.





Definition eines Files über einem beliebigen Datentyp D:

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

Hierbei:

- R beschreibt gelesene Teile des Files (links vom Sichtfenster)
- L beschreibt noch nicht gelesene Teile (rechts vom Sichtfenster)
- D Typ des Objekts, das sich gerade unter dem Sichtfenster befindet und bearbeitet werden kann.

Zu file gehören alle Objekte der Form
$f=((...(((eps,d_1),d_2),d_3),...,d_{k1}),d_k, (d_{k+1},(d_{k+2},(d_{k+3},...(d_n,eps)...))))$ mit $d_i \in D$.

Rechtssequenz ↑ Linkssequenz
Sichtfenster

Beispiel: Definition eines Files über dem Datentyp int:

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

Standardoperationen:

Funktion eof (end of file).

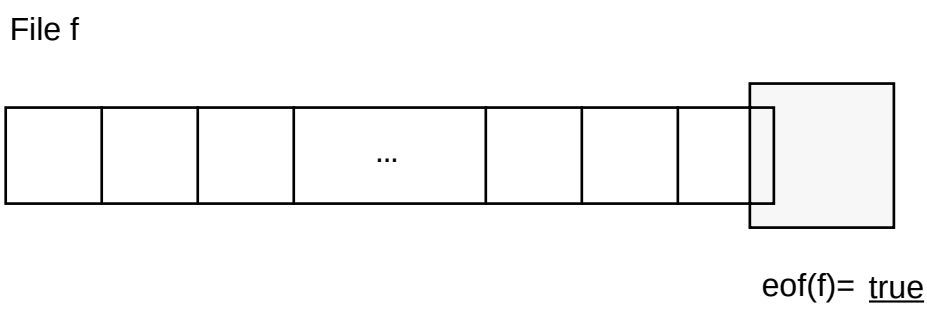
Erreichen des Fileendes: Sichtfenster hinter dem letzten Element des Files f – in diesem Falle ist das Sichtfenster undefiniert ⊥:

eof: file→bool mit

true, falls f=((...((eps,d_1),d_2),...,d_n),⊥,eps) für n≥0,

eof(f)=

false, sonst.



Funktion reset.

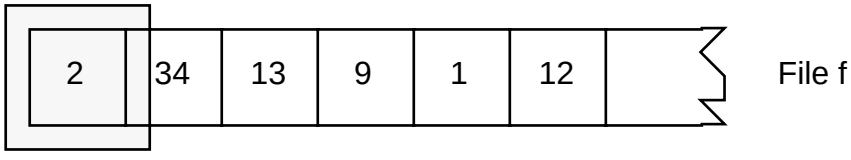
Vorbereiten eines Files zum Lesen (sog. *Öffnen*):

reset: file→file mit

(eps,d_1,(d_2,(d_3,,...(d_n,eps)...))), falls f≠(eps,⊥,eps)

reset(f)=

f, sonst.



Funktion get

liefert das unter dem Sichtfenster stehende Fileelement:

$$\text{get: file} \rightarrow D \text{ mit}$$

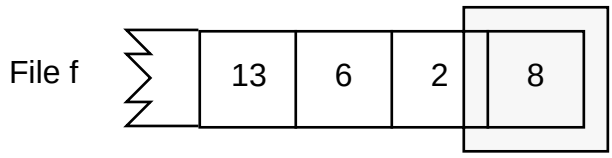
$$\text{get}(f) = \begin{cases} d_k, & \text{falls } 1 \leq k \leq n \\ \perp, & \text{sonst.} \end{cases}$$

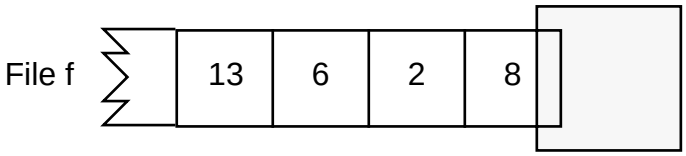
Funktion move

bewegt das Sichtfenster um eine Position nach rechts weiter:

$$\text{move: file} \rightarrow \text{file mit}$$

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



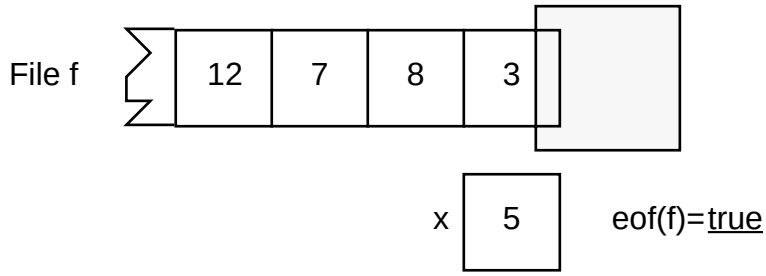


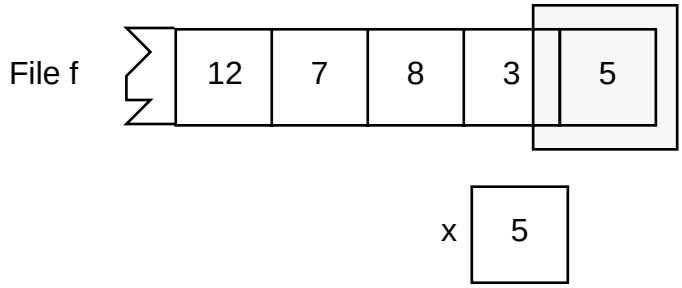
Funktion put

fügt einen Wert am Ende des Files f an:

put: D×file→file mit

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





Funktion rewrite

löscht den gesamten Fileinhalt und eof(f) liefert den Wert true:

rewrite: file→file mit

rewrite(f)=(eps,⊥,eps).

Beispiele:

1) Gegeben zwei Files f und g vom Grundtyp int.
   Aufgabe: Verkettung von f und g (**Konkatenation**) und Abspeicherung im File h:

```
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) Mischen zweier Files f und g zum File h.

Grundtyp von f, g und h:

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

f und g seien nach Nachnamen alphabetisch sortiert. Diese Sortierung soll nach dem Mischen auch für h gelten:

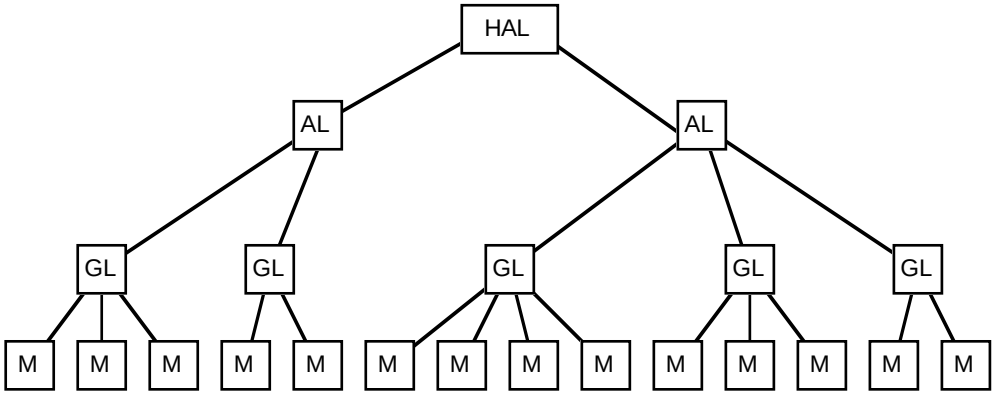
```
typ Namen ≡ (vorname: text, nachname: text);
typ L ≡ {eps} | (Namen,L);
typ R ≡ {eps} | (R,Namen);
typ Namenfile ≡ (R,Namen,L);
def f,g,h : Namenfile;
def x,y : Namen;
{An dieser Stelle seien f und g irgendwie belegt worden}
f ← reset (f);
g ← reset (g);
h ← rewrite (h);
solange nicht (eof(f) oder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

1.4.6.2 Bäume

Bisher: Sequenzen: jedes Element einer Sequenz (bis auf das letzte) hat genau einen Nachfolger.

Ab jetzt: Elemente können auch mehrere Nachfolger besitzen.

Beispiel: Personalhierarchie.
Personalgruppen: Hauptabteilungsleiter (HAL), Abteilungsleiter (AL), Gruppenleiter (GL), Mitarbeiter (M).

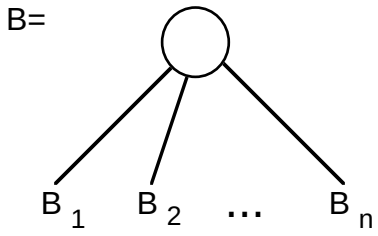


Baum.

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

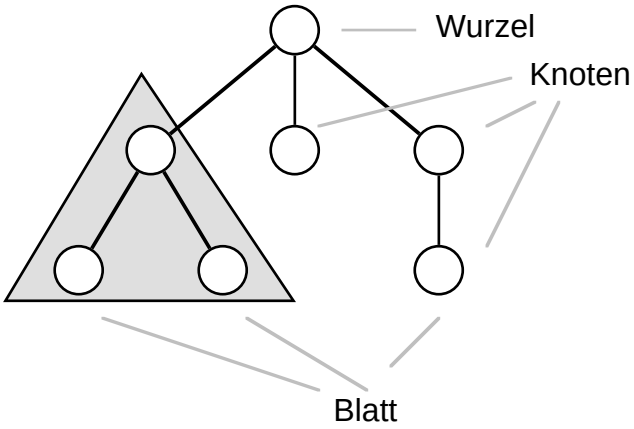


ein Baum.

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:

$$h(B) = \begin{cases} 0, & \text{falls B der leere Baum ist,} \\ 1+\max \{h(B_i) \mid 1 \leq i \leq n\}, & \text{falls B nichtleer ist.} \end{cases}$$

Beispiel:



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: Oben Markierungsmenge M={HAL,AL,GL,M}.

Besonders wichtige Rolle in der Informatik: **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**.

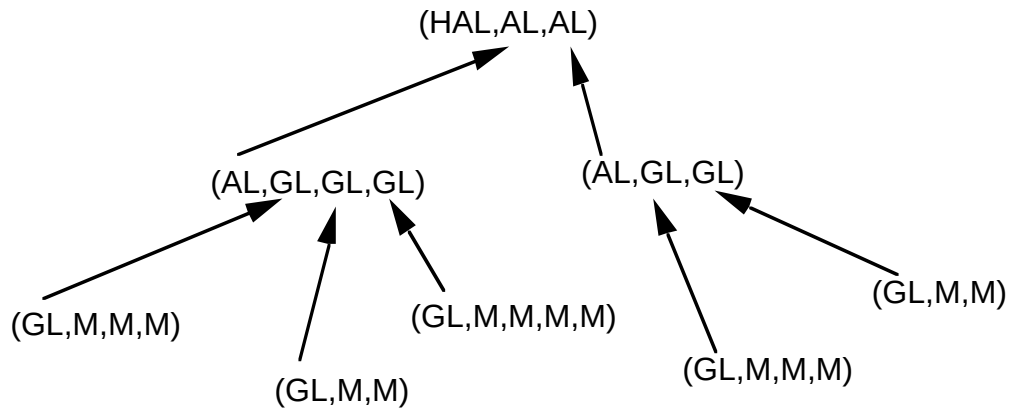
Definition als Datentyp?
Schema:
- Definiere leeren Baum
- Definiere, wie man aus zwei vorhandenen Bäumen einen größeren durch Hinzunahme eines Knotens erhält.

Problem:
zweidimensionale Darstellung von Bäumen <-> lineare Folge von Zeichen in der Typdefinition.
Trick:
Wähle lineare Darstellung, die *isomorph* zu den darzustellenden Bäumen ist.
Beispiel: (HAL,AL,AL) bedeutet
„HAL und seine ihm untergebenen AL“
oder genauso zulässig, aber weniger logisch:
(AL,AL,HAL)
Analog: (AL,GL,GL) bzw. (AL,GL,GL,GL).

Schließlich insgesamt:

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

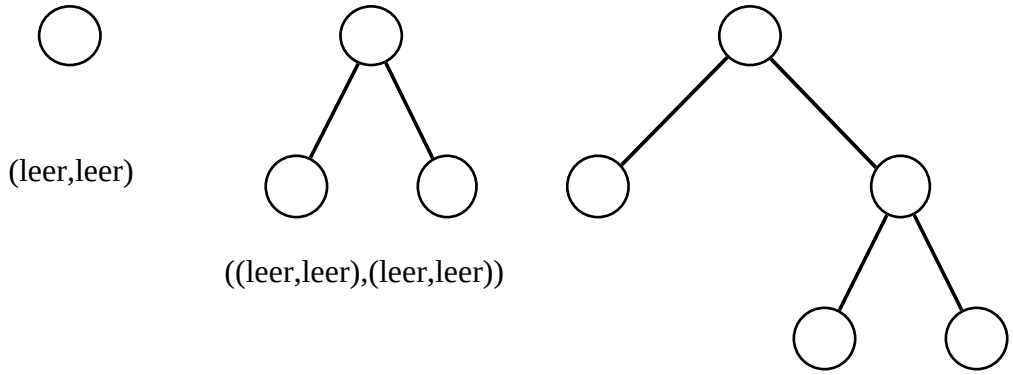
Eindeutige Darstellung.



Analog: **Typdefinition** eines *unmarkierten* binären Baumes:

typ Baum ≡ {leer} | (Baum,Baum).

Beispiel:.



Typdefinition eines **markierten** binären Baumes:
D beliebiger Datentyp für Markierungen:

typ DBaum ≡ {leer} | (DBaum,D,DBaum).

Beispiel: Markierter binärer Baum zum Entschlüsseln von Morsezeichen
Codiere Morsecode so im Baum, daß man beim Durchlaufen des Baumes von der Wurzel aus zu einem Knoten aus dem zurückgelegten Weg den Morsecode ablesen kann.
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 .-..

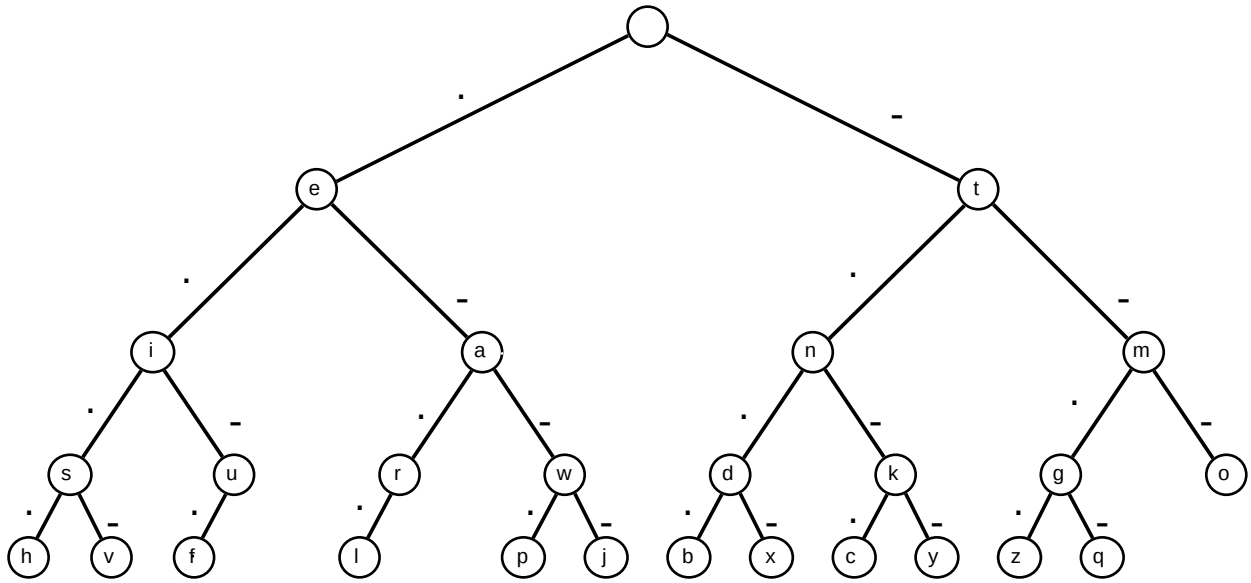


Abb. 13: Morsebaum

Sonderfall hier: Wurzel ohne Markierung, Morsebaum nicht leer.

Definition:

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.

Erweiterung: unterschiedliche Markierungsmengen für Blätter und innere Knoten.

Voraussetzung:

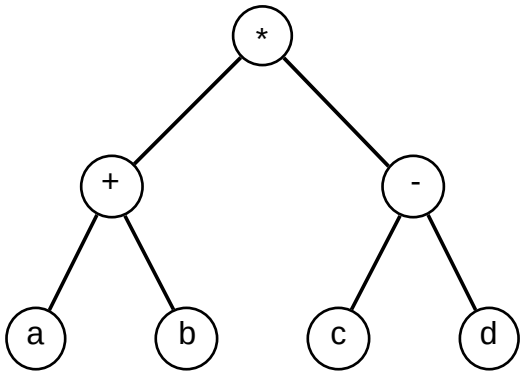
- Typ D für Markierungen der inneren Knoten
- Typ E für die Markierungen der Blätter:

Definition:

typ DEBaum ≡ E | (DEBaum,D,DEBaum).

Beispiel: arithmetische Ausdrücke.
Beispiel:

(a+b)*(c-d)



Merkmale:

- Blätter <-> Operanden
- innere Knoten <-> Operationen
- Klammern entfallen, dafür Hierarchiebildung
- Hierarchie bestimmt Reihenfolge der Auswertun: von den Blättern zur Wurzel.

Datentypdefinition:

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

Darstellung des obigen Baums:

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

1.4.7 Bildung von Funktionenräumen

Übergang von zwei Datentypen D' und D'' zum **Funktionenraum** D, d.h. zur Menge aller Abbildungen f: D'→D''. Allgemein:

typ D ≡ [D'→D''].

Beispiele:

1) typ boolFunkt ≡ [(int, int)→bool].
   Wertebereich: 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].
   Wertebereich: 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.

In der Praxis:

- Eingeschränkte Verfügbarkeit von Funktionenräumen in Programmiersprachen
- Realisierung stößt auf große Probleme
- in vielen Programmiersprachen nur einzelne feste Funktionen definierbar.
- In der Programmiersprache ML Funktionenraumkonzept weitgehend realisiert.

Funktionenräume **sehr allgemeines Konzept**:

jedes Objekt ist Funktion mit geeigneten Parametern.

Konstante 3 -> nullstellige Abbildung mit der Bezeichnung `3` von einer einelementigen Menge {()}, die nur das leere Tupel enthält, in die Menge der ganzen Zahlen int:

`3`: {()}→int mit

`3`()=3.

Bezeichne Typ {()} als unit => jeder Datentyp D ist Funktionenraum [unit→D].

Für jedes d∈ D gibt es in [unit→D] eine Funktion `d` mit `d`()=d.

Konsequenz:

- alle Datentypkonzepte auf Funktionen zurückgespielt
- Funktionenraumbildung ist universellster Konstruktor.