3 Implementierung von Datentypen

Bisher haben wir noch nie darüber nachgedacht, wie man Datenstrukturen auf die Speicherstruktur einer realen Maschine abbildet, so daß sie von der Maschine effizient ausgeführt bzw. verwaltet werden können. Dazu gehört u.a. die Frage, wie man die Formularmaschine realisiert oder wie man Records, Listen oder Bäume in Speicherzellen unterbringt. Dies behandeln wir in den folgenden Abschnitten.
Wie unsere Basismaschine ungefähr aussieht, wissen wir bereits aus Abschnitt 4 der Vorlesung "Algorithmen, Daten, Programme I". Auch wie sie programmiert wird (mit ASS), haben wir dort behandelt. In ASS zu programmieren, ist aber überaus mühsam, wie wir gesehen haben. Daher werden wir uns auf ein etwas höheres Sprachniveau begeben, auf dem aber noch die typischen maschinenbezogenen Aspekte sichtbar bleiben. Das ist das Niveau von Standard-PASCAL. Auf diesem Niveau werden wir zeigen, wie die gängigen Datentypen implementiert werden.

3.1 Die PASCAL-Maschine

PASCAL wurde um 1970 von Prof. Dr. Niklaus Wirth an der Eidgenössischen Technischen Hochschule in Zürich entwickelt und hat – aus damaliger Sicht – wegen ihrer Einfachheit und Mächtigkeit eine große Verbreitung erlangt.

Manche Sprachelemente von PASCAL werden Ihnen bereits von PRO her bekannt vorkommen. Das ist kein Zufall, denn PASCAL ist wie PRO eine *imperative Programmiersprache* (s. Abschnitt 1). Die Syntax von (Standard-)PASCAL finden Sie in Form von Syntaxdiagrammen zusammengefaßt im Duden Informatik.

3.1.1 Datentypen

PASCAL verfügt über folgende elementare Datentypen, die bis auf real und string skalar sind, mit den üblichen Operationen:

Typ	Werte	wichtige Operationen
boolean	false, true	and, or, not
integer	7, -10, +3276	+, -, *, div, mod, sqr, abs
real	-1.0, 2.5, 3.7E4, -0.2E-2, 0.072E+2	+, -, *, /, abs, sqr, sin, sqrt, ...
char	'a', 'B', '"' (einzelnes Hochkomma), ' ' (Leerzeichen)	chr (Umwandlung einer Zahl in ein Zeichen), ord (Umkehrfunktion dazu)
string	'Text', '-275', 'Peter''s Fahrrad'	keine Operationen, nur Konstanten

Auf skalaren Datentypen gibt es die gewohnten Standardfunktionen pred und succ.

Es folgen nun vorab einige Konstruktoren.

Enumeration.
Bildung eines skalaren Typs durch Aufzählung der Elemente in der Form

$(d_1,d_2,...,d_n)$

mit der impliziten linearen Ordnung $d_1<d_2<...<d_n$, z.B.

(rot, gruen, blau).

Restriktion.
In PASCAL existiert nur die Restriktion durch Intervallbildung, die von allen skalaren Datentypen in folgender Form gebildet werden kann:

a..b,
z.B. 1800..2000
oder 'a'..'z'
oder rot..blau.

Deklaration von Typen, Konstanten und Variablen.
PASCAL ist eine streng typisierte Programmiersprache: Alle Objekte besitzen genau einen Datentyp.
Typdeklarationen besitzen die allgemeine Form

type $t_1=T_1$;
...;
$t_n=T_n$

Hierbei sind $t_1,...,t_n$ Bezeichner und $T_1,...,T_n$ Typen. Das Semikolon verbindet einzelne Typdeklarationen.

Beispiele:
```
type Wochentag = (Mo, Di, Mi, Do, Fr, Sa, So);
     Arbeitstag = Mo..Fr;
     Jahrhundert = 1900..1999;
     Ganzzahl = integer
```

Die letzte Definition bewirkt, daß der Datentyp integer auch unter dem Synonym Ganzzahl verfügbar ist.

Konstanten deklariert man durch Gleichsetzung eines Bezeichners mit einer Konstanten eines Datentyps. Das reservierte Wort const wird vorangestellt.

Beispiele:
```
const pi = 3.1415926;
      DiesesJahr = 1997;
      Initial = 'A';
      Ausgabetext = 'Paul''s Schwester'
```

Der Typ einer Konstanten ergibt sich aus der Darstellung der Konstanten. Ist etwa die Konstante eine Zahl mit Dezimalpunkt oder dem Exponentenzeichen E, so wird ihr der Typ real zugeordnet.

Variablen deklariert man in PASCAL etwa so wie in PRO. Lediglich das Wort def muß durch das reservierte Wort var ersetzt werden. Mehrere Variablendekarationen müssen durch ein Semikolon (wie bei Typ- und Konstantendeklarationen) miteinander verbunden werden. var muß man dann nicht wiederholen.

Beispiele:
```
var  Tag: Wochentag;
     Jahr: Jahrhundert;
     x,y: integer;
     a,b,c: boolean
```

Anstelle des Typbezeichners darf in einer Variablendeklaration auch der Typ direkt hingeschrieben werden. Man kann also eine Deklaration der Form

```
type t=T;
var  x: t
```

durch

```
var  x: T
```

abkürzen.

Bevor wir weitere Konstruktoren behandeln, schieben wir zunächst die entsprechenden Sprachelemente im Bereich der Anweisungen ein.

3.1.2 Anweisungen

Elementare Anweisungen.

Zuweisung: Wichtigste elementare Anweisung aller imperativer Programmiersprachen; allgemeine Form in PASCAL

```
x:=E          (in PRO: ← statt :=)
```

x ist eine Variable, E ein Ausdruck; x und E müssen den gleichen Typ besitzen. Nur wenn x vom Typ integer und E vom Typ real ist oder umgekehrt, wird eine automatische *Typanpassung* durchgeführt.

Eingabe: Einlesen einer Folge von n durch Leerzeichen getrennten Werten von der Tastatur und Zuweisung an n Variablen $x_1,...,x_n$:

```
read(x1,...,xn).
```

Oder bei Eingabeschluß mit der Return-Taste:

```
readln(x1,...,xn).
```

Ausgabe: Ausgeben einer Folge von n durch Ausdrücke $E_1,...,E_n$ bestimmten Werten auf den Bildschirm:

```
write(E1,...,En).
```

Oder Ausgabe mit anschließendem Zeilenvorschub:

writeln(E_1,...,E_n).

Einzelner Zeilenvorschub:

writeln.

Konstruktoren.

Sequenz: Verbindung von Anweisungen A_1,...,A_n durch den Konstruktor ";" zu einer neuen Anweisung

A_1;...;A_n,

z.B.: x:=7; x:=x+y.

Verbundanweisung: Verbindung von Anweisungen A_1,...,A_n zu einer Sequenz mit Klammerung:

begin A_1;...;A_n end.

Traditionell schreibt man die Verbundanweisung zur besseren Übersichtlichkeit in eingerückter Form auf:

```
begin
    A1;
    ...;
    An
end.
```

Bedingte Anweisung: Hierfür stehen drei Versionen zur Verfügung.

Die einseitige Alternative

if B then A

und die zweiseitige Alternative

if B then A else A'.

Hierbei sind B ein Boolescher Ausdruck und A, A' Anweisungen, entweder elementare oder Verbundanweisungen, aber keine Sequenzen.

Ferner gibt es die Fallunterscheidung. Sind E ein Ausdruck mit Wert innerhalb eines skalaren Datentyps (außer real) und A_1,...,A_m elementare oder Verbundanweisungen, so ist die allgemeine Fallunterscheidung definiert durch:

```
case E of
    a11,...,a1n1: A1;
    ...
    am1,...,amnm: Am
end.
```

Für das erste a_{ij}, für das $E=a_{ij}$ ist, wird A_i ausgeführt.

Schleifen: In PASCAL gibt es drei verschiedene Konstruktoren für Schleifen:

Die *Zählschleife* notiert man in der Form

for i:=E to E' do A

oder for i:=E downto E' do A.

Hierbei ist i eine beliebige Variable eines skalaren Datentyps T (außer real), E und E' sind Ausdrücke vom Typ T, die einmalig zu Beginn der Schleife ausgewertet werden. Bedeutung: Falls E>E' tue nichts, sonst führe A nacheinander aus für

i=E, succ(E),...,$succ^k$(E) mit k=max{j | $succ^j$(E)≤E'}.

Analog wird auch die downto-Schleife ausgewertet, wobei man statt succ die Funktion pred setzt.

Die beiden bedingten Schleifen notiert man durch

while B do A

oder durch

repeat A' until B.

Hierbei sind B eine Boolescher Ausdruck, A eine elementare oder eine Verbundanweisung und A' eine elementare, einer Verbundanweisung *oder eine Sequenz*.

Die Semantik der while-Schleife entspricht der bekannten solange-Schleife. Die repeat-Schleife wird solange ausgeführt, bis B wahr wird. Dies bedeutet, daß A' in jedem Falle mindestens einmal ausgeführt wird.

Damit sind die Sprachelemente im Bereich der Anweisungen abgeschlossen.

Nun können wir uns den prinzipiellen Aufbau eines Programms ansehen. Ein PASCAL-Programm besitzt folgenden schematischen Aufbau:

program <Bezeichner> (input,output,<durch Komma getrennte Liste von
Filevariablen, die im Programm verwendet werden);
<Deklarationen mit Konstanten, Typen, Variablen, Funktionen,
Prozeduren (dies ist die empfohlene Reihenfolge) durch Semikolon getrennt>
<Verbundanweisung> .

Beispiel: Man betrachte die PASCAL-Version von Mischen in Abschnitt 1 unter dem Unterabschnitt "imperative Programmierung".

3.1.3 Weitere Datentypkonstruktoren

Homogene Aggregation.

PASCAL verfügt nur über die Bildung eines n-dimensionalen Arrays in der allgemeinen Form

array [I_1,...,I_n] of T.

Hierbei sind I_1,...,I_n Indextypen, für die jeder beliebige endliche skalare Datentyp (meist eine Restriktion von integer) verwendet werden kann. T ist der Grundtyp.

Beispiele: array [4..9] of integer

array [(rot,gruen,blau)] of 1800..2000

array [1..100,1..50] of real.

Der Zugriff zu einzelnen Komponenten eines Arrays a (die Selektion) erfolgt in der allgemeinen Form

a [E_1,...,E_n],

wenn a wie folgt deklariert wurde:

var a: array [I_1,...,I_n] of T.

Dabei sind E_1,...,E_n Ausdrücke, deren Werte jeweils in den Wertemengen I_1,...,I_n liegen.

Inhomogene Aggregation.

Die Zusammenfassung von Daten unterschiedlichen Datentyps unter einem Bezeichner nennt man in PASCAL *Verbund* oder *Record*. Jede Komponente eines Verbundes erhält einen Bezeichner. Auf die einzelnen Komponenten wird mit der dot-Notation zugegriffen.

Die Deklaration eines Datentyps "Verbund" lautet allgemein

```
type t = record
           t1 : T1;
           t2 : T2;
           ...
           tn : Tn
         end.
```

Dabei sind t,t_1,...,t_n verschiedene (!) Bezeichner und T_1,...,T_n beliebige Datentypen (auch Verbundtypen sind möglich). Die Wertemenge von t ist also die Menge aller möglichen n-Tupel der Form (a_1,a_2,...,a_n), wobei jedes a_i vom Datentyp T_i ist.

Beispiel: Ein Verbund mit Daten über einen Mitarbeiter:

```
type tt = record
            name : array [1..10] of char;
            gebdat : array [1..3] of integer;
            gehalt : real;
            geschlecht: (m,w)
          end;
var   angestellter : tt
```

oder mit dem Datum seinerseits als Verbund:

```
type tt1 = record
             name : array [1..10] of char;
             gebdat : record
                        tag : 1..31;
                        monat : 1..12;
                        jahr : integer
                      end;
             gehalt : real;
             geschlecht: (m,w)
           end;
var   angest : tt1
```

Die Selektion erfolgt mit der dot-Notation: Für die Variable v vom Verbundtyp t bezeichnet der Ausdruck

$$v.t_i$$

die i-te Komponente des Verbunds.

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

with $v_1,v_2,...,v_n$ do A,

wobei $v_1,v_2,...,v_n$ Variablen eines Recordtyps und A eine elementare oder eine Verbundanweisung ist.

Generalisation.

In PASCAL wird die Generalisation durch den *varianten Verbund* (*varianter Record*) realisiert, eine Erweiterung eines einfachen Verbunds um einen varianten Anteil. Allgemeine Form:

```
type t = record
           t1 : T1;
           ...                          fester Anteil
           tn : Tn;
           case i : I of
                i1 : (s11 : S11; ... ; s1m1 : S1m1);
                ...                                          varianter Anteil
                ip : (sp1 : Sp1; ... ; spmp : Spmp)
         end
```

Der feste Anteil des varianten Verbundes hat die bekannte Form eines einfachen Verbundes. Im varianten Anteil ist I der Bezeichner eines endlichen skalaren Datentyps und i ein beliebiger Bezeichner, der sog. Typdiskriminator. $i_1,...,i_p$ sind alle möglichen Werte des Datentyps I. Eine Zeile der Form

$$s_{i1} : S_{i1}; ... ; s_{im_i} : S_{im_i}$$

hat die gleiche syntaktische Form wie der feste Anteil eines Verbundes, d.h. die s_{ik} sind Bezeichner (Selektoren), die S_{ik} Datentypbezeichner.

Die case-Klausel hat folgende Wirkung: Es wird ein Bezeichner i vom Typ I deklariert. Falls der Bezeichner i im Programm z.B. den Wert $i_j \in I$ besitzt, dann besteht der Verbund zu diesem Zeitpunkt aus seinem festen Anteil und aus den Deklarationen, die in Klammern hinter i_j stehen. Die übrigen Bezeichner innerhalb des varianten Anteils sind dann nicht zugreifbar; der Versuch, dies doch zu tun, wird vom Computer bemerkt und mit einer Fehlermeldung quittiert. Der Bezeichner i diskriminiert also durch seinen Wert andere Deklarationen (daher der Name *Typdiskriminator*). Für $i=i_j$ ist der obige variante Verbund t identisch mit dem einfachen Verbund

```
record
    t1 : T1;
    ...
```

$t_n : T_n$;
$i : I$; {wobei i den Wert i_j besitzt}
$s_{i_j}1 : S_{i_j1}$;
...
$s_{i_jm_j} : S_{i_jm_j}$
end

Mathematisch ist die Wertememge von t die Menge

$$T_1\times...\times T_n\times I\times S_{11}\times...\times S_{1m_1}\cup T_1\times...\times T_n\times I\times S_{21}\times...\times S_{2m_2}\cup$$
$$...\cup T_1\times...\times T_n\times I\times S_{p1}\times...\times S_{pm_p}.$$

Beispiel: Wir deklarieren eine Variable fahrzeug als varianten Verbund:

```
type art = (fahrrad, lkw, bus);
var fahrzeug :
    record
        hersteller : array [1..20] of char;
        neupreis : real;
        case aktuell : art of
            fahrrad: (nabenschaltung : boolean);
            lkw: (ladefläche : real);
            bus: (stehplätze : integer)
    end;
```

Der Datentyp art ist notwendig, weil im varianten Anteil nur Datentypbezeichner zugelassen sind. Der Typdiskriminator ist die Variable aktuell. Wird aktuell auf den Wert bus gesetzt, so enthält der Verbund neben dem festen Anteil nur noch die Deklaration stehplätze: integer analog für die Fälle fahrrad und lkw.

Der Selektion auf Komponenten eines varianten Verbundes erfolgt wie bei einfachen Verbunden.

Man beachte noch folgende Nebenbedingungen:

- Alle Bezeichner innerhalb des Verbundrumpfes müssen verschieden sein, auch wenn sie in unterschiedlichen Varianten vorkommen.
- Innerhalb eines Verbundrumpfes darf nur ein varianter Anteil enthalten sein, der stets auf den festen Anteil folgt. Innerhalb einer Variante sind jedoch weitere Varianten möglich. Varianten können also geschachtelt werden.
- Falls der variante Anteil für ein oder mehrere Werteelemente w des Typdiskriminators leer sein soll, so lautet die Variante:

```
case ...
    ...
    w : ();
    ...
```

Potenzmengenbildung.

Zur Deklaration eines Datentyps "Potenzmenge von ..." (Wortsymbol in PASCAL: set of) benötigt man einen skalaren Datentyp T (außer real). Man deklariert dann einen Potenzmengentyp M in PASCAL allgemein wie folgt:

type M = set of T.

Eine Variable m vom Typ M besitzt eine der Mengen {X | X⊆T, |X| endlich} als Wert.

Beispiel: Eine Variable m, die Mengen von ganzen Zahlen als Werte annehmen kann, wird deklariert durch

var m : set of integer.

m kann Werte aus der Menge {X | X⊆integer, |X| endlich} annehmen.

Bei den meisten PASCAL-Systemen darf die Anzahl der Elemente in einer Menge eine bestimmte maximale Größe nicht überschreiten (z.B. 256 oder 4096 Elemente). Daher ist in der Praxis der Typ set of integer in der Regel verboten.

Konstante Mengen notiert man durch Auflistung aller ihrer Elemente in eckigen Klammern, also

$[a_1,a_2,...,a_n]$ für die Menge $\{a_1,a_2,...,a_n\}$,

wobei die a_i Elemente des Datentyps T sind. Allgemeiner können anstelle der a_i auch stehen:

1. Ausdrücke, die Werte aus T liefern.
2. Intervalle von T. Allgemein entspricht eine Mengenkonstante der Form

   $[E_{11}..E_{12},E_{21}..E_{22},...,E_{n1}..E_{n2}]$

   der Menge

   $\{x \in T \mid$ es gibt ein i, $1 \le i \le n$, mit $E_{i1} \le x \le E_{i2}\}$.

   Dabei sind alle E_{ij} Ausdrücke mit Werten in T.

Die leere Menge ∅ wird durch das Symbol [ ] dargestellt.

Beispiele:

a)
```
type monate = set of 1..12;
var  sommer, winter : monate;
```
Der Wertebereich von sommer und winter ist die Menge aller Teilmengen der Zahlen 1 bis 12.

b)
```
type arbeitszeit = 8..17;
var sprechzeiten : set of arbeitszeit
```
Durch folgende Zuweisung wird sprechzeiten auf eine Menge von Zeiten zwischen 8 und 16 gesetzt:

```
sprechzeit := [8..10, 12, 15..17].
```

Standardoperationen auf Elementen vom Datentyp set:

- \+ (Vereinigung von Mengen, mathem. Symbol: ∪)
- * (Durchschnitt von Mengen, mathem. Symbol: ∩)
- \- (Differenz von Mengen, mathem. Symbol: \)

Zusätzlich gibt es vier Vergleichsoperationen, die Werte vom Typ boolean liefern:

= (Gleichheit von Mengen)

<> (Ungleichheit von Mengen)

<= (Teilmengeneigenschaft, $\subseteq$)

>= (Obermengeneigenschaft, $\supseteq$).

Die Infix-Operation in hat die gleiche Bedeutung wie das Elementsymbol $\in$. Sei x ein Element des Grundtyps T und m eine Menge vom Typ set of T. Es gilt:

$$\text{in} : T \times 2^T \rightarrow \text{boolean}$$

$$x \text{ in } m = \begin{cases} \text{true, falls } x \in m, \\ \text{false, falls } x \notin m. \end{cases}$$

Files.

Zur Deklaration eines Datentyps "File" (Wortsymbol in PASCAL: file of) benötigt man einen beliebigen Grundtyp T. Man deklariert dann ein File f allgemein wie folgt:

```
type filetyp = file of T;
var  f : filetyp.
```

Die Wertemenge von f ist die Menge aller Folgen

$t_1 t_2 ... t_n$ mit $n \geq 0$ und $t_i \in T$.

Für $n=0$ sprechen wir von einem *leeren* File.

Beispiel:

a) Ein File f von ganzen Zahlen wird deklariert durch

```
var f : file of integer
```

b) Ein File, in dem der Vor- und Nachname, sowie das Geburtsdatum von Personen gespeichert werden sollen, wird deklariert durch:

```
var  personen: file of record
          vorname, nachname: array [1..10] of char;
          datum:    record
                         tag: 1..31;
                         monat: 1..12;
                         jahr: integer
                    end
          end
```

Zur Selektion: Bei der Deklaration einer Variablen f vom Typ file of T wird automatisch eine *Puffervariable* vom Datentyp T mit vereinbart. Diese kann über den Bezeichner f↑ angesprochen werden. f↑ besitzt immer den Wert der Filekomponenten, die sich gerade unter dem Sichtfenster (vgl. Abschnitt 9 aus "Algorithmen, Daten, Programme I") befindet. Zur Bearbeitung eines Files stehen folgende Standardoperationen zur Verfügung (f sei stets deklariert als var f: file of T):

1) Fileende prüfen: Es gilt: eof(f)=true, falls das Sichtfenster hinter dem letzten Element des Files f steht, und eof(f)=false in den übrigen Fällen.
2) Öffnen eines Files zum Lesen: reset(f)
   setzt das Sichtfenster auf das erste Element des Files f. Die Puffervariable f↑ erhält den Wert des ersten Fileelements, sofern f nicht leer ist. Ist f leer, so liefert der Aufruf von eof(f) den Wert true, und die Puffervariable ist undefiniert.
3) Lesen einer Zelle: get(f)
   verschiebt das Sichtfenster um eine Zelle nach rechts und weist der Puffervariablen f↑ den Inhalt der neuen Zelle zu. Falls das Fileende von f erreicht ist, hat f↑ einen undefinierten Wert. Der Aufruf eof(f) liefert dann den Wert true. get(f) darf nur verwendet werden, wenn eof(f)=false gilt.
4) Schreiben einer Zelle: put(f)
   fügt den Wert der Puffervariablen f↑ am Ende des Files f an und setzt das Sichtfenster um eine Position weiter. f↑ hat danach einen undefinierten Wert. put darf nur verwendet werden, wenn eof(f)=true gilt. Nach Ausführung von put gilt weiterhin eof(f)=true.
5) Löschen und Öffnen zum Schreiben: rewrite(f)
   löscht den gesamten Fileinhalt. f↑ ist anschließend undefiniert, und eof(f) liefert den Wert true.

Im Programm müssen die zu verwendenden Files auf besondere Weise kenntlich gemacht werden: Die Bezeichner der Files sind wie üblich zu deklarieren und zusätzlich im Programmkopf aufzuführen. Der Computer wird dadurch angewiesen, das jeweilige File auf einem externen Speichermedium zu suchen bzw. anzulegen, falls es noch nicht vorhanden ist.

3.1.4 Funktionen und Prozeduren

PASCAL besitzt ein relativ eingeschränktes Konzept für die Definition von Funktionen, das nur wenig mit der Leistungsfähigkeit des Konzepts in funktionalen Programmiersprachen gemeinsam hat. Ferner gibt es in PASCAL ein Konzept für Prozeduren.

Funktionen.

Die allgemeine Definition lautet:

function f (<Parameterliste>): T;
<Deklarationsteil wie bei Programmen>
<Verbundanweisung>

Die Parameterliste besteht aus einer Folge von formalen Parametern mit ihrem Typ und einer Festlegung der Parameterübergabeart. Sie hat die allgemeine Form

<Parameterliste>::=[var | function | procedure] <Folge von Bezeichnern>:<Typbezeichner> {;<Parameterliste>}.
<Parameterliste>::=[var | function] <Folge von Bezeichnern>:<Typbezeichner> {;<Parameterliste>} |

procedure <Folge von Bezeichnern>] {;<Parameterliste>}.

Die Parameterspezifikation ist ziemlich hakelig: Zum einen kann man Funktionen und Prozeduren (s.u.) als Parameter übergeben und damit sehr eingeschränkt Funktionale definieren – in diesem Fall verwendet man die Spezifikation

function <Bezeichner>:<Typbezeichner>

oder procedure <Bezeichner>.

Zum anderen kann man Variablen als Parameter übergeben. Dazu stehen zwei Parameterübergabearten zur Verfügung:

- call by value: In diesem Fall steht vor dem formalen Parameter nichts. Der Parameter ist innerhalb der Funktion eine (lokale) Variable, deren Initialwert der aktuelle Parameter ist.
- call by reference: Vor dem Bezeichner steht das Wort var. Wirkung: Der formale Parameter wird innerhalb der Funktion als Synonym für den aktuellen Parameter verwendet. Alle Modifikationen des formalen Parameters wirken sich auf den aktuellen Parameter aus.

Vorsicht: Funktionen können nun *Seiteneffekte* besitzen, also neben der erwarteten Berechnung des Funktionswertes weitere nicht unmittelbare ersichtliche Aktivitäten auslösen.

Innerhalb der Verbundanweisung hat irgendwo eine Zuweisung der Form

f:=w

zu erfolgen. Die Zuweisung assoziiert einen Wert w vom Typ T mit dem Bezeichner f. w ist der Funktionswert von f und tritt nach Beendigung der Funktion an die Stelle des Aufrufs.

Im Deklarationsteil der Funktion können sog. *lokale Objekte* definiert werden, die nur innerhalb der Funktion sichtbar sind. Wie der Zugriff auf eine Variable innerhalb einer Funktion genau erfolgt, wenn der Bezeichner sowohl innerhalb der Funktion als auch außerhalb definiert ist (*Namenskonflikt*), wird durch *Sichtbarkeitsregeln* festgelegt. Dazu werden folgende Bezeichnungen benötigt:

Seien P und Q zwei Funktionen (hierbei rechnet auch das Programm selbst als Funktion):

a) P ist in Q **direkt geschachtelt** (oder Q **umfaßt** P **direkt**) (im Zeichen: $P \ll Q$), falls P im Deklarationsteil von Q definiert wird.
b) P ist in Q **geschachtelt** (oder Q **umfaßt** P) (im Zeichen: $P << Q$), falls es eine endliche Folge von Funktionen $R_1, R_2, ..., R_n$, $n \geq 2$, gibt, so daß $P = R_1 \ll R_2 \ll ... \ll R_n = Q$ gilt.
c) Gibt es eine Funktion R, so daß P und Q direkt in R geschachtelt sind, also $P \ll R$ und $Q \ll R$, so sind P und Q **parallel**.
d) Ein Bezeichner x ist **lokal** bezüglich P, wenn x in der Parameterliste oder im Deklarationsteil von P vereinbart ist.
e) Ein Bezeichner x ist **global** bezüglich P, wenn es eine Funktion Q gibt mit $P << Q$ und x in der Parameterliste oder im Deklarationsteil von Q, aber nicht in der Parameterliste oder im Deklarationsteil von P vereinbart ist.

Beispiel: In folgendem Programmfragment

```
program P (...);
function Q ...;
begin ... end;
```

```
function R ...;
    function S ... ;
    begin ... end;
    function T ...;
    begin ... end;
begin ... end;
begin
    ...
end.
```

sind die Funktionen S und T direkt in R, und R und Q sind direkt in P geschachtelt. S und T sind parallel, ebenso Q und R. S und T sind in P geschachtelt, aber nicht direkt. Man beachte, daß Q und T nicht parallel sind. Es gilt also:

$$S \underline{<<} R,\ T \underline{<<} R,\ R \underline{<<} P,\ Q \underline{<<} P,\ S << P,\ T << P.$$

Mit dieser Bezeichnungsweise können wir nun die **Sichtbarkeitsregeln** definieren. Sei P eine Funktion und x ein Bezeichner, der im Anweisungsteil von P verwendet wird.

Fall 1: x ist lokal deklariert.
Dann bezeichnet x das Objekt, das in P definiert wird.

Fall 2: x ist global deklariert.
Dann gibt es eine Funktion Q (oder das Hauptprogramm), bezüglich der x lokal ist. x bezeichnet das Objekt, das in der innersten P umfassenden Funktion deklariert ist.
Formal: Gemeint ist das Objekt x, für das gilt:
- x ist lokal bezüglich der Funktion Q.
- Für alle Funktionen R mit P<<R<<Q gilt: x ist global bezüglich R.

Fall 3: x ist weder lokal noch global definiert.
Dann ist x entweder ein Standardbezeichner (z.B. read), oder es liegt ein Fehler vor, weil x überhaupt nicht oder irgendwo in einer zu P parallelen Funktion deklariert wurde. Deklarationen in parallelen Funktionen sind für P aber nicht "sichtbar".

Anschaulich findet man folgendermaßen heraus, auf welches Objekt mit dem Bezeichner x zugegriffen wird: Zunächst schaut man im Deklarationsteil der Prozedur P nach, in der x verwendet wird. Ist x dort nicht definiert, so betrachtet man den Deklarationsteil der Prozedur, die P direkt umfaßt usw. Die erste Deklaration von x, die man auf diesem Wege erreicht, gibt Auskunft darüber, welches x an der ursprünglichen Stelle gemeint ist (Abb. 1).

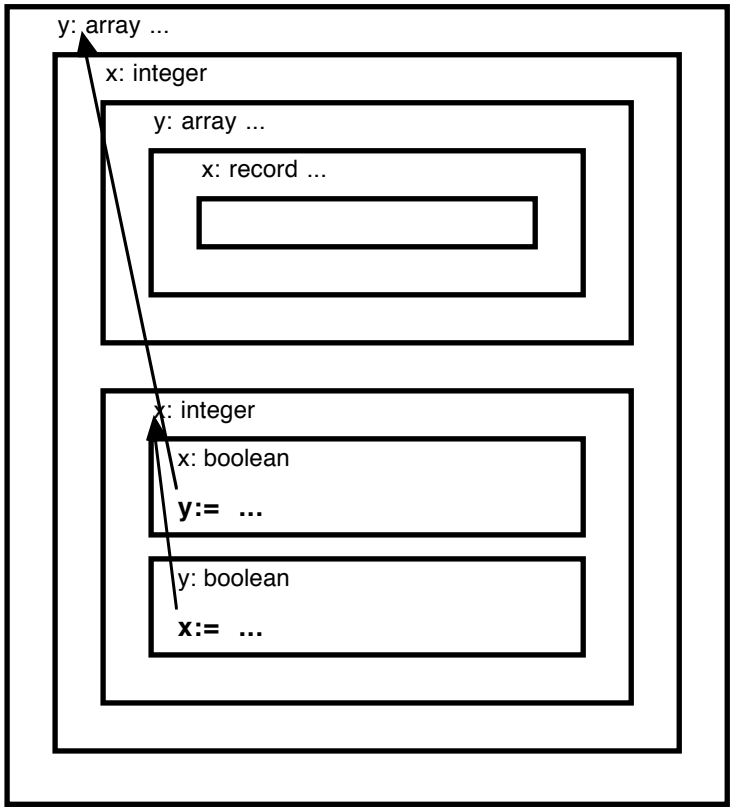


Abb. 1: Sichtbarkeitsregeln (Prozeduren/Funktionen sind hier durch Kästen dargestellt)

Beispiel:

a)

```
program ggt(input,output);
var x,y: integer;
function rest (x,y: integer):integer;
begin
      if x>=0 and y>0 then
            while x>=y do x:=x-y;
      rest:=x
end;
function ggt (x,y: integer): integer;
var r: integer;
begin
      if x>=0 and y>0 then
            while y<>0 do
            begin
                  r:=rest(x,y);
                  x:=y;
                  y:=r
            end;
      ggt:=x
end;
begin
      readln(x,y);
      writeln(ggt(x,y))
end.
```

b) Auch Rekursion ist möglich:

```
program ggt(input,output);
var x,y: integer;
```

```
begin
      if y=0 then ggt:=x else
            if y<=x then ggt:=ggt(y,x) else ggt:=ggt(x,y-x)
end;
begin
      readln(x,y);
      writeln(ggt(x,y))
end.
```

Prozeduren.

Prozeduren sind eine ggf. parametrisierte Zusammenfassung von Deklarationen und Anweisungen zu einer mit einem Bezeichner versehenen aufrufbaren Einheit. Prozeduren berechnen anders als Funktionen keine Werte, sondern sie ändern den Programmzustand (dargestellt durch die Werte aller Variablen).

Man definiert schematisch:

```
procedure p (<Parameterliste wie bei Funktionen>);
<Deklarationsteil wie bei Funktionen>
<Verbundanweisung>
```

Die Sichtbarkeitsregeln übertragen sich sinngemäß.

Beispiel: Im folgenden Programm wird ein Feld sortiert. Die beiden Prozeduren ändern den Programmzustand, indem sie Werte von Variablen ändern:

```
program sort (input, output);
type feld = array [1..10] of integer;
var    a: feld;
       k: 1..10;
procedure sortiere(var a: feld);
var   i, j: 1..10;
      procedure tausche(var x,y: integer);
      hilf: integer;
      begin
            hilf:= x; x:= y; y:= x
      end;
begin
      for i:= 1 to 9 do
            for j:= i+1 to 10 do
                  if a[i] > a[j] then tausche(a[i],a[j])
end;
begin
      for k:= 1 to 10 do read (a[k]);
      sortiere(a);
      for k:= 1 to 10 do write (a[k])
end.
```

Bisher haben wir uns weitgehend auf unsere Intuition verlassen, um herauszufinden, wie eine Funktion oder Prozedur abgearbeitet wird. Die genaue Wirkung eines Aufrufs wird durch die sog. **Kopierregel** beschrieben. Sie legt fest, wie der Prozedurrumpf, der gemäß den Sichtbarkeitsregeln und Parameterübergabearten modifiziert wurde, an die Stelle des Aufrufs kopiert und ausgeführt wird. Diese Kopierregel, die bei imperativen Sprachen wesentlich

komplizierter ist als bei funktionalen Sprachen und zu vielen Unübersichtlichkeiten führen kann, wird im folgenden präzisiert. (Man beachte aber, daß unterschiedliche PASCAL-Systeme meist unterschiedliche Kopierregeln verfolgen.)

Kopierregel.

Gegeben sei der Aufruf einer Funktion oder Prozedur in der Form

$$P\ (a_1, a_2, ..., a_n),$$

wobei $a_1,...,a_n$ die aktuellen Parameter sind. Die Kopierregel wird durch folgende Vorschrift festgelegt:

1. Der Bezeichner P muß in einer Prozedur- oder Funktionsdeklaration deklariert worden und an der Stelle des Aufrufs sichtbar sein. P muß folgendermaßen deklariert sein:

   $\underline{\text{procedure}}\ P\ (D_1; D_2; ...; D_n); R$

   oder

   $\underline{\text{function}}\ P\ (D_1; D_2; ...; D_n): T; R.$

   $D_1,...,D_n$ sind die Spezifikationen für die formalen Parameter. Jedes D_i hat eine der Formen

   $x_i: T_i$ oder
   $\underline{\text{var}}\ x_i: T_i$ oder
   $\underline{\text{procedure}}\ x_i(...)$ oder
   $\underline{\text{function}}\ x_i(...): T_i.$

   R ist der Rumpf der Prozedur bestehend aus dem Deklarationsteil und der Verbundanweisung.

   Es gibt also genau n formale Parameter (soviel wie aktuelle). Der Typ eines jeden a_i ist
   - entweder identisch zum Typ T_i von x_i, und a_i ist kein Ausdruck, falls für x_i die Referenzübergabe vereinbart ist,
   - oder zuweisungskompatibel (d.h. a_i kann x_i ggf. unter automatischer Typanpassung zugewiesen werden) zum Typ T_i von x_i, falls für x_i die Wertübergabe vereinbart ist.

   a_i ist eine Prozedur, falls x_i als Prozedur deklariert ist, und a_i ist eine Funktion mit Ergebnistyp T_i, falls x_i so deklariert ist.
2. Wenn diese Überprüfungen positiv verliefen, so wird zunächst eine Kopie des Prozedurrumpfes (einschließlich der lokal deklarierten Variablen) erzeugt und darin jedes Vorkommen des formalen Parameters x_i gemäß der Sichtbarkeitsregeln durch a_i ersetzt, falls es sich bei x_i um eine Funktion oder eine Prozedur handelt. Eventuelle Namenskonflikte werden durch Umbenennungen beseitigt.
3. Anschließend wird jedes Vorkommen von x_i durch a_i ersetzt, wenn x_i ein Parameter mit Referenzübergabe und a_i eine einzelne Variable ist. Ist aber a_i eine Feldkomponente der Form $b[E]$, so wird E zunächst ausgewertet. E habe den Wert e. Danach wird x_i überall durch $b[e]$ ersetzt.
4. Nun wird der Deklarationsteil des Prozedurrumpfs um die übrigen Parameterdeklarationen (also die formalen Parameter mit Wertübergabe) ergänzt. Ist x_i ein formaler Parameter mit

Wertübergabe, so wird zu Beginn des Deklarationsteils die Deklaration einer lokalen Variable

$\underline{\text{var}}\ x_i: T_i$

und zu Beginn des Anweisungsteils im Rumpf die Wertzuweisung

$x_i := a_i$

eingefügt.

5. Der so modifizierte Prozedurrumpf wird nun anstelle des Aufrufs eingesetzt und ausgeführt. Dabei werden mögliche Übergabefehler erkannt (z.B. unterschiedliche Typen von aktuellen und formalen Parametern).
6. Nach seiner Abarbeitung wird dieser Prozedurrumpf wieder durch den ursprünglichen Aufruf ersetzt; anschließend wird mit der Verarbeitung bei der Anweisung fortgefahren, die unmittelbar auf den Aufruf folgt.
   (Bei Funktionen tritt der berechnete Wert des Ergebnistyps nach Abarbeitung an die Stelle des Aufrufs und der Ausdruck, in dem die Funktion aufgerufen wurde, wird weiter ausgewertet.)

Soweit die Beschreibung der Kopierregel für Prozeduren und Funktionen in PASCAL. Die Kopierregel bleibt korrekt, wenn Prozeduren oder Funktionen sich selbst im Inneren ihres Rumpfes aufrufen.

Bezeichnung: Den Prozedur- oder Funktionsrumpf, den man nach Ausführung der ersten vier Schritte der Kopierregel erhält, bezeichnet man als **Inkarnation** der Prozedur bzw. Funktion.

Beispiel: Gegeben sei folgendes Programm:

```
program beispiel (input, output);
var   a: array [1..3] of integer;
      n: integer;
procedure p (var x: integer; y: real; function f: integer);
var hilf: integer;
begin
      hilf:= x+f(y);
      x:= 2*hilf
end;
begin
      n:= 1; a[1]:= 1; a[2]:= 20; a[3]:= -17;
      p (a[2+n], 2*n, sqr)
end.
```

Wir gehen nun die einzelnen Schritte der Kopierregel durch und geben ggf. den modifizierten Prozedurrumpf von p an. Die jeweils geänderten Stellen sind fett gedruckt.

1. Schritt: Die Überprüfungen liefern keine sichtbaren Fehler: a[2+n] ist vom Typ integer, 2*n ist zwar vom Typ integer, aber zuweisungskompatibel zu y, sqr ist eine integer-Funktion.

2. Schritt: Ersetzen der formalen Prozedur- und Funktionsparameter durch die aktuellen. Ergebnis:

```
var hilf: integer;
begin
      hilf:= x+sqr(y);
```

```
        x:= 2*hilf
end;
```

3. Schritt: Ersetzen der formalen Parameter mit Referenzübergabe durch die aktuellen, unter Beachtung der Ausnahmen bei Feldern. Es wird der Ausdruck 2+n=3 ausgewertet. x wird dann durch a[3] ersetzt. Ergebnis:

```
var hilf: integer;
begin
        hilf:= a[3]+sqr(y);
        a[3]:= 2*hilf
end;
```

4. Schritt: Deklaration der formalen Parameter mit Wertübergabe. Ergebnis:

```
var   y: real;
        hilf: integer;
begin
        y:= 2*n;
        hilf:= a[3]+sqr(y);
        a[3]:= 2*hilf
end;
```

Die Schritte 5 und 6 sind klar.

3.2 Begriff der Implementierung von Datentypen

Wir wollen nun beschreiben, wie man höhere Datentypen durch die PASCAL-Maschine implementiert.

Unter der Implementierung D' eines Datentyps D verstehen wir anschaulich eine Darstellung der Werte von D durch die Werte von D' und eine Simulation der Operationen von D durch Operationen oder Operationsfolgen von D', so daß sich D' nach außen genauso verhält wie D. Genauer:

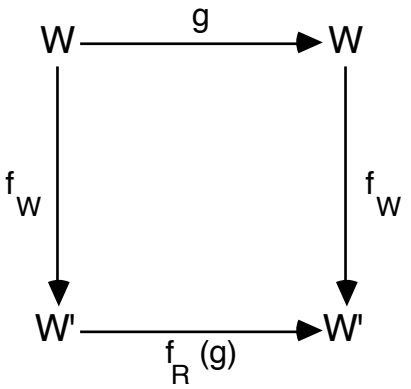
Definition A:

Für eine Menge M von Funktionen sei $M^\circ=\{f_1 \circ f_2 \circ ... \circ f_r \mid f_i \in M, r \geq 1\}$ die Menge aller Kompositionen von Funktionen aus M.

Die **Implementierung** eines Datentyps D=(W,R) durch einen Datentyp D'=(W',R') ist eine Abbildung f: D→D' mit $f=(f_W,f_R)$, wobei f_W: W→W', f_R: R→R'°, so daß für alle x∈W, g∈R gilt:

$$f_W(g(x))=f_R(g)(f_W(x)).$$

Notiert man die Funktionalität der in Definition A beteiligten Abbildungen graphisch, so erhält man ein sog. *kommutierendes Diagramm:*



Zunächst ist nun zu klären, wie die Speicherstruktur eines realen Rechners auf dem Sprachniveau von PASCAL aussieht, denn in diese Struktur ist ja schließlich jeder Datentyp abzubilden.

Der Speicher unserer Basismaschine besteht aus einer Folge von Speicherzellen gleicher Größe, die wir in PASCAL als homogene Aggregation wie folgt definieren können:

type Speicher=array Adressen of Wort.

Hierbei sind die Typen Adressen und Wort maschinenabhängig; in der Regel ist Adressen ein Restriktionstyp von nat, z.B

type Adressen=0..1048575.

Wort ist ein Typ, der die kleinste adressierbare Einheit des Speichers beschreibt.

Wir haben nun für alle bekannten Datentypen D Implementierungen durch den Datentyp Speicher der Form $f=(f_W,f_R)$: D→Speicher anzugeben. Hierzu die folgenden Abschnitte.

3.3 Implementierung von Arrays

1-dimensionale Arrays.

Gegeben sei die allgemeine Form eines 1-dimensionalen Arrays über dem Grundtyp T

typ A≡array nat [0..n] of T.

Nehmen wir zunächst der Einfachheit halber an, die Werte von T passen jeweils in ein Wort des Speichers. Dann ist eine Implementierung $f=(f_W,f_R)$: A→Speicher zu bestimmen, die folgendes leistet: Für a∈A und sp∈Speicher mit

$f_W(a)=sp$

gilt:

$f_W(a(i))=sp[f_R(i)]$ für alle i mit $0\le i\le n$.

Eigentlich müßte es hier heißen: $f_W(\pi_{i,n}(a))=f_R(\pi_{i,n})(f_W(a))=f_R(\pi_{i,n})(sp)=sp[i']$. g ist hier die Projektion $\pi_{i,n}$ auf die i-te Komponente von a.

Die Implementierung besteht also darin, eine geeignete Umrechnung der Indizes $i\to i'=f_R(\pi_{i,n})$ zu bestimmen, so daß $a(i)=sp[i']$. Eine solche Umrechnung f_R bezeichnet man als **Adreßfunktion** (location-function, Abk. loc)

Sei a_0 die Anfangsadresse, ab der a in sp abgelegt werden soll. Dann ist offenbar

$i'=loc(i)=f_R(i)=a_0+i,\ 0\le i\le n$.

Benötigen die Werte von T zur Unterbringung im Speicher c Worte, so gilt offenbar

$$i' = loc(i) = a_0 + c \cdot i,\ 0 \le i \le n,$$

und

$$f_W(a(i)) = sp(f_R(i)) = sp(loc(i)) \ldots sp(loc(i)+c-1).$$

Mehrdimensionale Arrays.

Nun betrachten wir ein 2-dimensionales Array

typ A≡array (nat [0..m],nat [0..n]) of T.

Hier müssen wir die Elemente eines Objekts a vom Typ A in geschickter Weise linear anordnen. Man kann dies zeilenweise (Abb. 2) tun oder spaltenweise oder auch diagonalenweise. Wir entscheiden uns hier für die zeilenweise Anordnung (andere Anordnungen siehe Übungen).

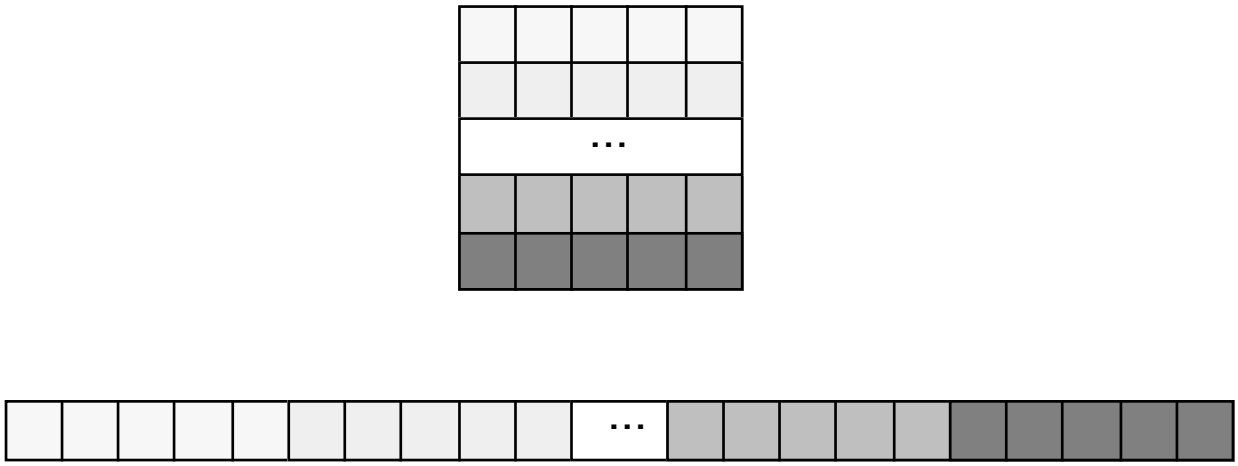


Abb. 2: Zeilenweise Anordnung eines 2-dimensionalen Arrays im Speicher

Ist a_0 die Anfangsadresse, so bestimmt sich die loc-Funktion, die jetzt zwei Argumente besitzt, zu

$$loc(i,j) = a_0 + c(n+1) \cdot i + c \cdot j,\ 0 \le i \le m,\ 0 \le j \le n,$$

und

$$f_W(a(i,j)) = sp(f_R(i,j)) = sp(loc(i,j)) \ldots sp(loc(i,j)+c-1).$$

Nun der allgemeine Fall. Gegeben sei der Typ

typ A≡array (nat [0..n_1],nat [0..n_2],...,nat [0..n_k]) of T.

Durch Übertragung erhält man bei zeilenweiser Anordnung folgende Adreßfunktion:

$$\begin{aligned} loc(d_1,d_2,\ldots,d_k) = \ & a_0 \\ & + c \cdot i_1 (n_2+1) \cdot \ldots \cdot (n_k+1) \\ & + c \cdot i_2 (n_3+1) \cdot \ldots \cdot (n_k+1) \\ & \ldots \\ & + c \cdot i_{k-1} (n_k+1) \\ & + c \cdot i_k \end{aligned}$$

$$= \quad a_0 + \sum_{j=1}^{k} c_j \cdot i_j$$

mit $$c_j = c \prod_{l=j+1}^{k} (n_l + 1).$$

Die loc-Funktion bei allgemeinen Indextypen der Form $\mathrm{nat}[m_i..n_i]$ wird in den Übungen behandelt.

Stets ist bei der Implementierung auf die Effizienz zu achten: Für die loc-Funktion bedeutet das, daß sie nach Möglichkeit eine lineare Funktion sein soll, um beim Zugriff auf ein Array-Element eine schnelle Auswertung zu sichern. Tatsächlich ist die loc-Funktion in allen obigen Fällen linear.

3.4 Implementierung von Records

Die Implementierung eines Recordtyps der allgemeinen Form

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

orientiert sich an der Speicherung von Arrays. Man legt die Komponenten $x.t_1,...,x.t_n$ eines Objekts $x \in T$ beginnend bei einer Speicherzelle a_0 sequentiell in den Speicherzellen ab. Benötigt man zur Speicherung der Objekte von T_i jeweils c_i Speicherzellen, so lautet die loc-Funktion

$$\mathrm{loc}(t_i) = a_0 + c_1 + c_2 + \ldots + c_{i-1}.$$

Wegen der Unterschiedlichkeit der c_i gibt es i.a. keine geschlossene Form für die Adreßfunktion. Daher ist man gezwungen, über den Bezeichner auf die Komponenten zuzugreifen statt über Indizes. Die jeweiligen Werte $c_1+c_2+...+c_{i-1}$ (sog. **Offsets**) werden vom Übersetzer zusammen mit a_0 und den Namen t_i in einer Tabelle gespeichert, stehen also zum Ablauf des Programms zur Verfügung (Abb. 3).

	a_0
t_1	0
t_2	c_1
...	...
t_n	$c_1+c_2+\ldots+c_{n-1}$

Abb. 3: Tabelle mit Anfangsadresse und Offsets

3.5 Implementierung von Mengen

Potenzmengentypen der allgemeinen Form

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

sind in Programmiersprachen i.a. nur erlaubt, wenn die Wertemenge des Grundtyps D' endlich ist. Eine mögliche Implementierung unendlicher (berechenbarer) Mengen haben wir bereits in Abschnitt 2 durchgesprochen. Sei fortan die Wertemenge von $D'=\{d_1,...,d_n\}$.

Eine übliche Implementierung einer beliebigen Menge $M \in D$ erfolgt (wie bei unendlichen Mengen) durch Übergang zur *charakteristischen Funktion*

$$\chi_M: D' \rightarrow \text{bool mit}$$

$$\chi_M(x) = \begin{cases} \text{true, falls } x \in M, \\ \text{false, sonst.} \end{cases}$$

Bei endlichen Grundtypen kann man nun die charakteristische Funktion durch eine endliche Folge von n booleschen Werten

$$\chi_M=(\chi_M(d_1),...,\chi_M(d_n))$$

realisieren, die man als 0-1-Folgen (0 für false, 1 für true) in einem oder mehreren Speicherworten unterbringt. Für jedes mögliche Element aus D' ist hier explizit abzulesen, ob es zu M gehört oder nicht.

Beispiel: Die Menge $M=\{1,3,4,7\} \in 2^{\{1,2,3,4,5,6,7,8,9,10\}}$ ist durch folgende 0-1-Folge dargestellt:

1 0 1 1 0 0 1 0 0 0.

Mit dieser Darstellung sind auch die gängigen Mengenoperationen leicht als boolesche Funktionen zu beschreiben:

Vereinigung zweier Mengen: stellenweise ODER-Funktion
Durchschnitt zweier Mengen: stellenweise UND-Funktion
Komplement einer Menge: stellenweise Negation
Elementabfrage $d_i \in M$: Test, ob i-te Stelle =1

Alle diese Operationen sind vor allem dann sehr effizient realisierbar, wenn die Anzahl n der Elemente nicht größer ist als die Länge eines Speicherwortes, da viele Prozessoren über spezielle Operationen verfügen, die sich genau auf Speicherworte beziehen.

3.6 Implementierung der Parameterübergabe

Sei im folgenden x der aktuelle und a der formale Parameter.

call by value.

Implementiert wird diese Übergabe durch Kopieren des Wertes von x in einen neuen gleichgroßen Speicherbereich, der beim Aufruf angelegt und unter dem Bezeichner a angesprochen wird (Abb. 4).

Handelt es sich bei x um ein Objekt eines strukturierten Typs (etwa ein großes Array) mit einer Vielzahl von elementaren Werten, so werdenalle diese Werte einzeln kopiert. Da dies mit hohem zusätzlichem Speicher- und Zeitaufwand verbunden ist, sollte man in allen diesen Fällen die effizientere call-by-reference-Übergabe vorzziehen, auch wenn über den Parameter tatsächlich keine Werte nach außen gegeben werden.

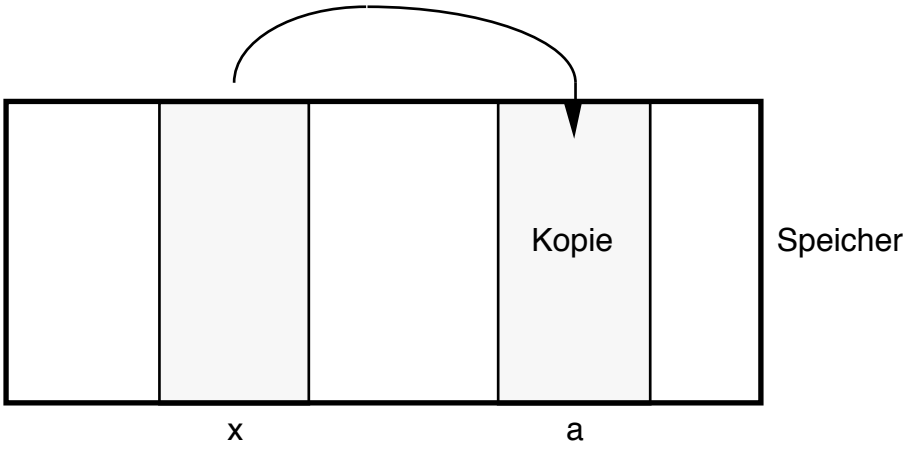


Abb. 4: call-by-value-Übergabe

call by reference.

Vermöge call-by-reference wird der formale Parameter a innerhalb der Prozedur oder Funktion als Synonym für den aktuellen Parameter x verwendet. Dazu wird eine neue Speicherzelle angelegt, die unter dem Bezeichner a angesprochen und in der ein Verweis (ein Zeiger, eine **Referenz**) auf x abgelegt wird (Abb. 5). Der Verweis besteht aus der Adresse der ersten Speicherzelle, an der x im Speicher beginnt. Es sind keine Kopiervorgänge von Werten nötig.

Die call-by-reference-Übergabe sollte aus Effizienzgründen auch dann gewählt werden, wenn keine Ergebnisse über den formalen Parameter nach außen gelangen sollen, also eigentlich eine call-by-value-Übergabe geboten wäre, der aktuelle Parameter jedoch ein Objekt mit einer größeren Zahl elementarer Daten ist. Denn in diesem Falle würde die call-by-value-Übergabe zu einer Vielzahl von Kopiervorgängen und einem hohem Speicherplatzverbrauch führen. Ist die Prozedur oder Funktion rekursiv, multipliziert sich der Zeit- und Platzbedarf. Es ist aber sicherzustellen, daß beim Übergang von call-by-value zu call-by-reference keine unerwünschten Seiteneffekte möglich werden.

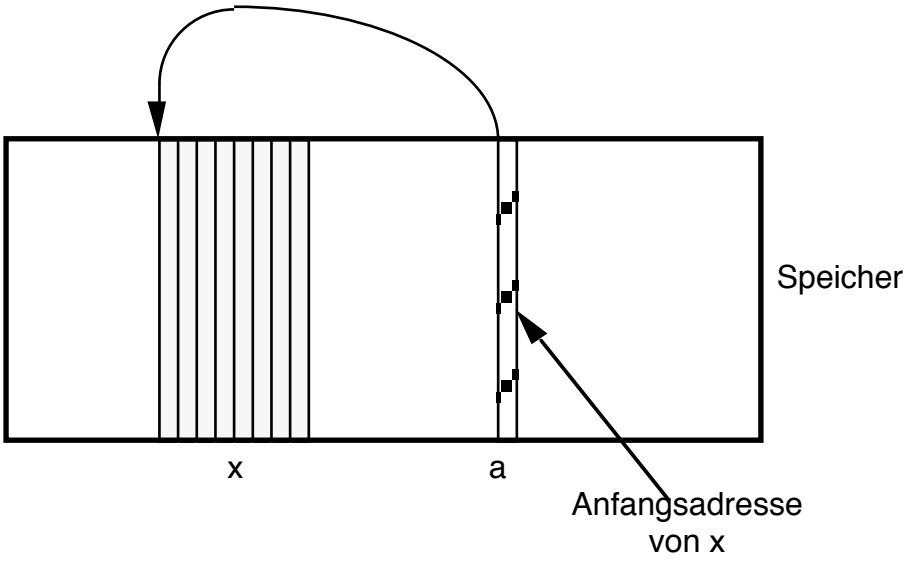


Abb. 5: call-by-reference-Übergabe

3.7 Implementierung von Stacks und Queues

Wir implementieren Stacks und Queues zunächst in Form *sequentieller Speicherverfahren*, d.h. die Einträge werden der Reihe nach in aufeinanderfolgenden Speicherzellen abgelegt. Später folgen dann Implementierungen durch *verkettete Speicherverfahren:* Hierbei werden die Einträge an beliebigen Stellen im Speicher verstreut untergebracht, und die Herstellung der Reihenfolge der Datenelemente erfolgt durch Verweise/Zeiger/Referenzen zwischen den Daten.

Stacks.

Für die Implementierung eines Stacks bietet sich die Verwendung eines Arrays an, wobei dann allerdings eine feste maximale Größe nicht überschritten werden darf (Abb. 6).

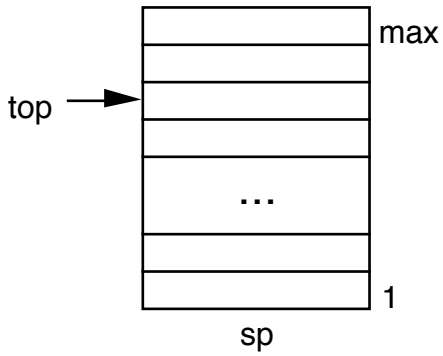


Abb. 6: Stack

Bei push und pop ist die Wahl der Parameterübergabeart klar: call-by-reference, weil Modifikationen des Stacks nach außen dringen sollen. Wir wählen jedoch auch bei den beiden Funktionen top und is_empty die call-by-reference-Übergabe, auch wenn der Stackinhalt nicht

verändert wird, denn wir wollen ausschließen, daß durch die call-by-value-Übergabe in einem zeitaufwendigen Verfahren jeweils eine komplette Kopie des Stacks angelegt wird:

```
const max=...;
type data=...;
     stack=record
               top: 0..max;
               sp: array [1..max] of data
           end;
function is_empty(var s: stack): boolean;
begin
     is_empty:=(s.top=0)
end;
procedure push(x: data; var s: stack);
begin
     with s do
     begin
          if top=max then "Overflow" else
          begin
               top:=top+1;
               sp[top]:=x
          end
     end
end;
procedure pop(var s: stack);
begin
     with s do
          if is_empty(s) then "Underflow" else top:=top-1
end;
function top(var s: stack): data;
begin
     if is_empty(s) then "Underflow" else top:=s.sp[s.top]
end;
procedure empty(var s: stack);
begin
     s.top:=0
end.
```

Initialisierung:

```
var s: stack;
     empty(s).
```

Beispiel: Wir schreiben ein Programm in PASCAL zur Syntaxanalyse von Zeichenfolgen auf korrekte Klammerung. Die Syntax definieren wir mit einer BNF-Grammatik G=(N,T,P,S) wie folgt: N={<Wort>}, T={(,),[,]}, S=<Wort>, und P enthält die Produktion:

<Wort>::= (<Wort>) <Wort> | [<Wort>] <Wort> | ε.

L(G) besteht aus allen Zeichenfolgen mit den Klammern (,),[,], die als korrekte Klammerung anzusehen sind.

Das Programm verwendet einen Stack, auf den jede gelesene "Klammer auf" abgelegt wird. Liest man eine "Klammer zu", so wird sie mit dem obersten Eintrag des Stacks verglichen und bei Gleichheit vom Stack gelöscht. Bei Ungleichheit liegt ein Syntaxfehler auf der Eingabe vor. Wird im Laufe der Analyse auf das oberste Element zugegriffen, obwohl der Stack leer ist, so liegt ebenfalls ein Syntaxfehler vor. Das gleiche gilt, wenn nach Lesen der gesamten Eingabe noch Symbole auf dem Stack stehen. Das Programm:

```
program Klammersyntax (imput,output);
type data=char;
<hier steht die Definition des Stacks und der Zugriffsprozeduren>
var   s: stack;
      ch: char;
procedure error;
begin
      writeln('Syntaxfehler')
end;
begin
      empty(s);
      while not eof do
      begin
            read(ch);
            case ch of
                  '(','[': push(s,ch);
                  ')': if is_empty(s) then error else
                              if top(s)='(' then pop(s) else error
                  ']': if is_empty(s) then error else
                              if top(s)='[' then pop(s) else error
                  otherwise: error
            end;
      end;
      if is_empty(s) then writeln('Korrekte Klammerung') else error
end.
```

Queues.

Auch Queues implementiert man häufig durch Arrays, wobei man sich dann anfangs wieder für eine maximale Größe der Queue entscheiden muß. Da nun jedoch stets nur hinten Elemente eingefügt und nur vorne Elemente ausgefügt werden, die Queue also keine Verankerung mehr im Speicher besitzt, muß man dafür sorgen, daß die Queue innerhalb des vorgesehenen Speicherbereichs bleibt und nicht den gesamten Speicher "durchwandert". Man realisiert die Queue daher zyklisch und stellt sich den vorgesehenen Speicherbereich zu einem Ring zusammengebogen vor. Anfang und Ende werden nun durch zwei Indizes anfang und ende markiert, die mit den Modifikationen der Queue im Gegenuhrzeigersinn durch das Feld wandern (Abb. 7). anfang weist dabei immer auf das Feldelement unmittelbar *vor* Beginn der Queue.

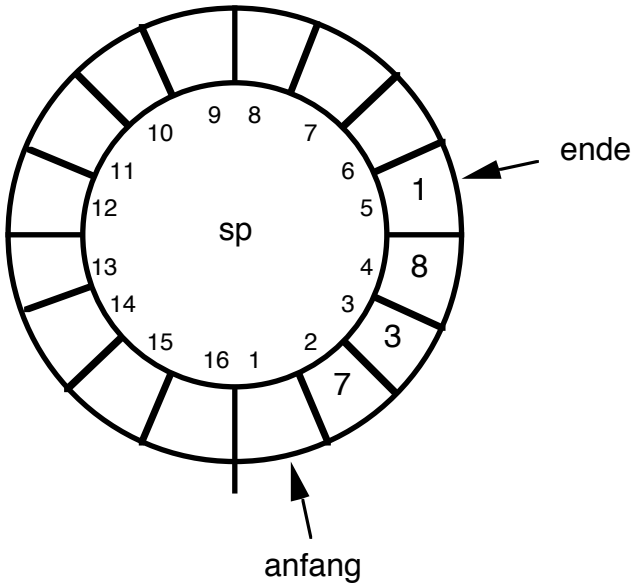


Abb. 7: Queue

Auch bei Queues wählt man aus Effizienzgründen wieder generell die Parameterübergabe call-by-reference:

```
const max=...;
type data=...;
     queue=record
               anfang, ende: 1..max;
               sp: array [1..max] of data
           end;
procedure enter(x: data; var q: queue);
begin
     with q do
     begin
          if ende=max then ende:=1 else ende:=ende+1;
          if ende=anfang then "Overflow" else sp[ende]:=x
     end
end;
function is_empty(var q: queue): boolean;
begin
     is_empty:=(q.anfang=q.ende)
end;
procedure remove(var q: queue);
begin
     with q do
          if is_empty(q) then "Underflow" else
               if anfang=max then anfang:=1 else anfang:=anfang+1
end;
function first(var q: queue): data;
begin
     if is_empty(q) then "Underflow" else
          if anfang=max then first:=q.sp[1] else first:=q.sp[anfang+1]
end;
procedure empty(var q: queue);
begin
     q.anfang:=max; q.ende:=max
end.
```

Initialisierung: var q: queue;
empty(q).

Overflows und Underflows.

Mit *Underflow* (*Unterlauf*) bezeichnet man allgemein die Situation, aus einer leeren Datenstruktur ein Element entfernen zu wollen. *Overflow* (*Überlauf*) nennt man umgekehrt die Situation, in eine volle Datenstruktur ein Element einfügen zu wollen. "Voll" bestimmt sich dabei relativ zum vorgesehenen oder insgesamt verfügbaren Speicher.

Underflow beruht i.a. auf einem fehlerhaften Programm, Overflow auf zu geringem vorgesehenen Speicher, also zu kleinem max. Man kann dann max nach Belieben vergrößern auf die Gefahr hin, daß es dann immer noch zu klein ist.

Eine elegantere Lösung, die zudem die gleichzeitige Benutzung mehrerer Stacks (für Queues geht es analog) unterstützt, besteht darin, eine freizügige Definition der einzelnen max zuzulassen, so daß erst dann ein Overflow angezeigt wird, wenn die Summe der Längen aller Stacks max überschreitet. In diesem Fall käme es nicht so sehr auf das einzelne Verhalten der Stacks an. Vielmehr würde sich das langsame Wachstum einzelner Stacks und das schnelle Wachstum anderer Stacks gegenseitig ausgleichen, so daß insgesamt eine bessere Ausnutzung des Speichers und eine geringere Overflow-Häufigkeit erreicht würde.

Diese Idee wollen wir im folgenden für Stacks implementieren. Wir gehen aus von einem Speicher mit l Speicherzellen:

type Speicher=array [1..l] of data.

In diesem Speicher werden n homogene Stacks der Reihe nach angelegt. Die einzelnen Stacks besitzen nun keine feste Verankerung mehr, d.h. wir haben daher neben der Menge der top-Zeiger noch eine Menge von base-Zeigern (Abb. 8), die den Anfang jedes Stacks markieren. Der Zeiger base[n+1] dient dabei der Vereinfachung der Algorithmen. Beide Mengen implementieren wir als Arrays. Die Stack-Operationen werden nun um einen weiteren Parameter ergänzt, der die Nummer des angesprochenen Stacks bezeichnet. Ein Überlauf eines Stacks tritt immer dann auf, wenn das Einfügen eines Elementes zu einer Überschneidung zweier Stacks führen würde (top[i]>base[i+1]).

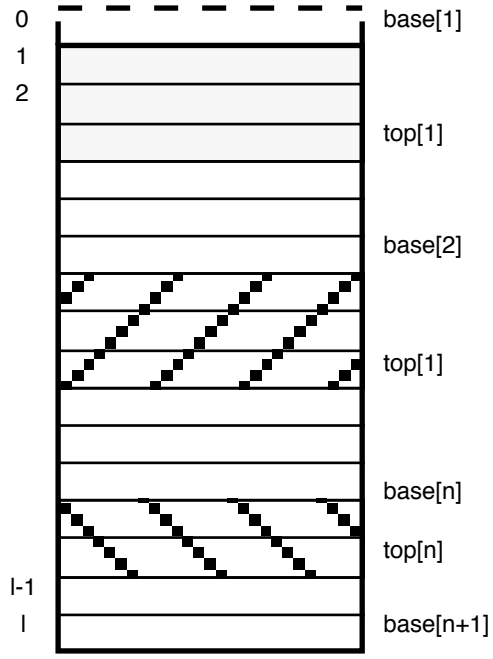


Abb. 8: n homogene Stacks

Die Definition:

```
type data=...;
     Speicher=array [1..l] of data;
     anzahl=1..n;
     n_stack=record
                top: array [anzahl] of 0..l+1;
                base: array [1..n+1] of 0..l;
                sp: Speicher
               end;
procedure push(x: data; i: anzahl; var s: n_stack);
begin
     with s do
     begin
          top[i]:=top[i]+1;
          if top[i]>base[i+1] then "Overflow" else sp[top[i]]:=x
     end
end;
function is_empty(i: anzahl; var s: n_stack): boolean;
begin
     is_empty:=(s.top[i]=s.base[i])
end.
procedure pop(i: anzahl; var s: n_stack);
begin
     with s do
          if is_empty(i,s) then "Underflow" else top[i]:=top[i]-1
end;
function top(i: anzahl; var s: n_stack): data;
begin
     with s do
          if is_empty(i,s) then "Underflow" else top:=sp[top[i]]
end;
procedure empty(var s: n_stack);
var i: integer;
begin
```

```
        with s do
        begin
            base[n+1]:=l;
            for i:=1 to n do
            begin
                base[i]:=(l*(i-1)) div n;
                top[i]:=(l*(i-1)) div n
            end
        end
    end.
```

Die Initialisierung

```
var s: stack;
empty(s)
```

verteilt die anfangs leeren Stacks gleichmäßig über den Speicherbereich.

Wie reagiert man nun auf einen Overflow? Angenommen beim Stack i tritt eine Overflow-Situation auf, der insgesamt zur Verfügung stehende Speicher ist aber noch nicht vollständig belegt. Man kann dann entweder alle Stacks so verschieben, daß bei Stack i genau ein freier Platz entsteht, um die Anforderung zu befriedigen. Dies ist jedoch auf längere Sicht unklug, da möglicherweise gleich anschließend erneut eine push-Anforderung bei Stack i ansteht, so daß wieder eine zeitaufwendige Umordnung notwendig wird. Sinnvoller ist es, im Rahmen der ersten Umordnung gleich eine längerfristige Lösung anzustreben und dafür zu sorgen, daß bis zur nächsten Umordnung eine möglichst große Zahl von Stack-Operationen möglich wird. Dies ist die Idee des **Garwick-Algorithmus**: Es werden mit jeder Umordnung jedem Stack im Rahmen der noch zur Verfügung stehenden Speicherzellen mehrere Plätze zugewiesen. Die genaue Anzahl richtet sich nach dem Verhalten des Stacks in der Vergangenheit seit der letzten Umordnung: Stacks, die in der Vergangenheit stark gewachsen sind, erhalten mehr zusätzliche Plätze zugewiesen als Stacks, die in der Vergangenheit nicht oder nur wenig gewachsen sind. Diese Überlegung beruht auf vielen statistischen Beobachtungen, die in ein sog. *Lokalitätsprinzip* gemündet haben: Programme verhalten sich innerhalb einer gewissen zukünftigen Zeitspanne etwa so, wie sie sich innerhalb ihren unmittelbaren Vergangenheit verhalten haben.

Für den Garwick-Algorithmus ergänzen wir unsere Implementierung für n_stacks um einige Komponenten:

```
type n_stack=record
                ...
                oldtop: array [anzahl] of 0..l;
                newbase: array [1..n+1] of 0..l;
                d: array [anzahl] of 0..l
              end;
```

Die Prozedur zur Behandlung eines Overflows wird wie folgt implementiert:

```
procedure overflow(var s: n_stack);
var   a,b: real;
      sum, inc: integer;
      j: 1..n;
begin
      with s do
      begin
```

```
            sum:=l; inc:=0;
            for j:=1 to n do
            begin
                  sum:=sum-(top[j]-base[j]);
                  if top[j]>oldtop[j] then
                  begin
                        d[j]:=top[j]-oldtop[j];
                        inc:=inc+d[j]
                  end else d[j]:=0;
            end;
            if sum<0 then "Fehler: Speicher ist voll" else
            begin
                  a:=(0.1*sum)/n; b:=(0.9*sum)/inc;
                  newbase[1]:=base[1];
                  for j:=2 to n do
                        newbase[j]:=newbase[j-1]+(top[j-1]-base[j-1])+
                                    trunc(a)+trunc(b*d[j-1]);
                  umordnen(s);
                  for j:=1 to n do oldtop[j]:=top[j]
            end
      end
end;
procedure umordnen(var s: n_stack);
var   j,j1: 2..n;
      k: 2..n+1;
begin
      with s do
      begin
            j:=2;
            while j<=n do
            begin
                  k:=j;
                  if newbase[k]<=base[k] then verschieben(s,k) else
                  begin
                        while newbase[k+1] >base[k+1] do k:=k+1;
                        for j1:=k downto j do verschieben(s,j1)
                  end;
                  j:=k+1
            end
      end
end;
procedure verschieben(var s: n_stack; m: integer);
var   delta: integer;
      j2:= 0..l;
begin
      with s do
      begin
            delta:=newbase[m]-base[m];
            if delta<>0 then
            begin
                  if delta>0 then
                        for j2:=top[m] downto base[m]+1 do sp[j2+delta]:=sp[j2]
                  else
                        for j2:=base[m]+1 to top[m] do sp[j2+delta]:=sp[j2];
                  base[m]:= newbase[m];
                  top[m]:=top[m]+delta
            end
      end
end.
```

Der Garwick-Algorithmus arbeitet nicht nur für Stacks, sondern für alle relativ zu einer Basis adressierten Tafeln, als z.B. auch für Queues. In diesem Falle liegen statt der Arrays base und top die Arrays für anfang und ende der Queues vor.

Algorithmische Analyse.
Bisher liegen für den Garwick-Algorithmus keine exakten theoretischen Effizienzanalysen vor. Man ist weitgehend auf die Ergebnisse experimenteller Untersuchungen angewiesen. So ist der Garwick-Algorithmus sehr effizient, wenn der Speicher etwa zur Hälfte gefüllt ist.
Problematisch ist sein Verhalten, wenn – insbesondere bei fehlerhaften Programmen – der Speicherbedarf nicht ausreicht. Kurz vorher wird dann noch ungeheuer viel Zeit für die fortlaufenden Umordnungen verschwendet. Möglicher Lösungsansatz: Man stoppt die Prozedur overflow mit einem Fehler, wenn die Zahl der freien Speicherplätze ein gewisses Minimum min>0 unterschreitet. In overflow ist dazu die Abfrage sum<0 durch sum<min zu ersetzen.

Zusammenfassung.
Wir haben nun sequentielle Speicherverfahren für Stacks und Queues kennengelernt. Insgesamt können wir die Merkmale sequentieller Verfahren wie folgt zusammenfassen: Sequentielle Speicherverfahren
- erlauben einen schnellen direkten Zugriff auf Datenelemente,
- erfordern ein kompliziertes Einfügen von Datenelementen, bei dem meist ein Umordnen der Datenelemente erforderlich ist, um für den neuen Eintrag Platz zu schaffen,
- sorgen oft für eine unzureichende Speicherausnutzung, weil für evtl. einzutragende Elemente Platz reserviert werden muß, oder weil Speicherbereinigungsalgorithmen, wie der Garwick-Algorithmus, nur bei halbgefülltem Speicher effizient sind (vgl. Abfrage "sum<min").

3.8 Implementierung von Rekursion

In diesem Abschnitt befassen wir uns sowohl mit der Implementierung von rekursiven Datentypen als auch mit der Rekursion im Kontrollbereich.
Zunächst zu den Datentypen. Betrachten wir zur Motivation die Definition eines markierten binären Baumes in FUN

```
typ Markierung ≡ {a,b,c,d,e};
typ Baum ≡ {leer} | (Baum,Markierung,Baum).
```

Ein Objekt vom Typ Baum ist z.B.

```
B=((leer,e,leer),a,((leer,b,leer),c,(leer,b,leer))).
```

Wie bringt man Bäume wie B im Speicher unter? Offenbar ist eine Darstellung der Form gem. Abb. 9 oben, in der die einzelnen Bestandteile linear im Speicher unter gebracht sind wenig geeignet. Denn nun gestalten sich Ein- und Ausfügeoperationen recht aufwendig, weil mit jeder dieser Operationen eine größere Speicherumordnung verbunden. Andererseits

kann man in Abb. 9 unten nicht in einen Speicherbereich "hineinschachteln", da man ja zu Beginn nicht die endgültige Größe der Objekte kennt.

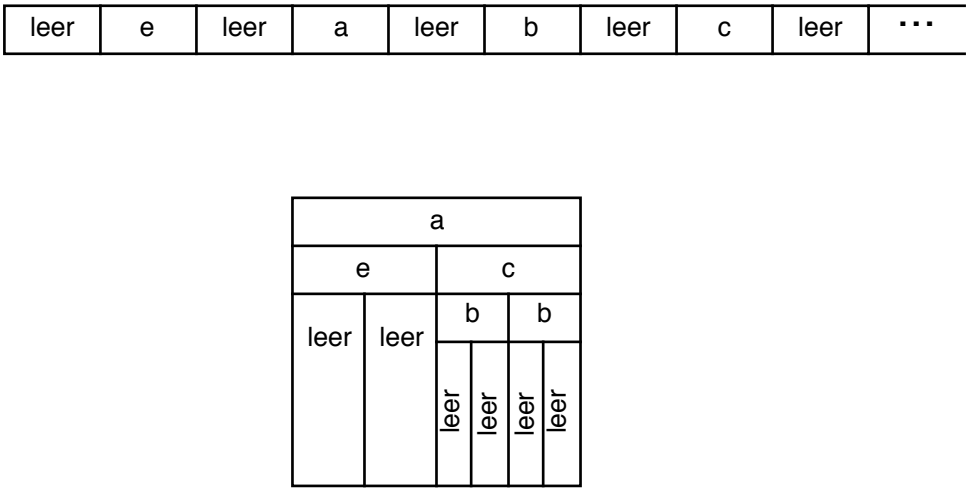
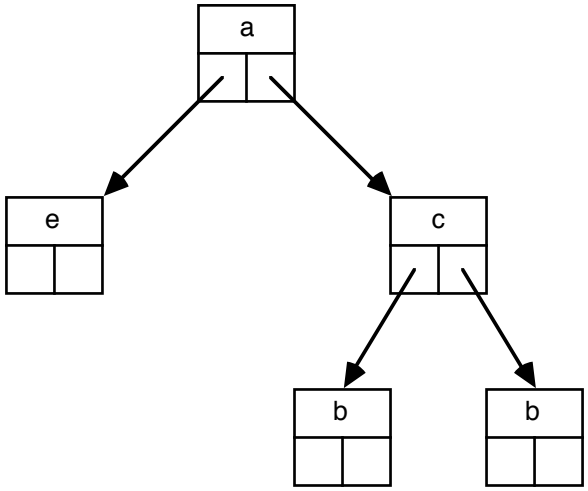


Abb. 9: Ungeeignete Speichermöglichkeiten für Bäume

Zur Lösung dieses Problems nutzt man die Referenztechnik, die bereits bei der Realisierung der call-by-reference-Übergabe genutzt wurde. Anstelle einer Schachtelungsstruktur gem. Abb. 9 geht man über zu einer verketteten Struktur gem. Abb. 10.



... oder als mögliche Anordnung im Speicher ...

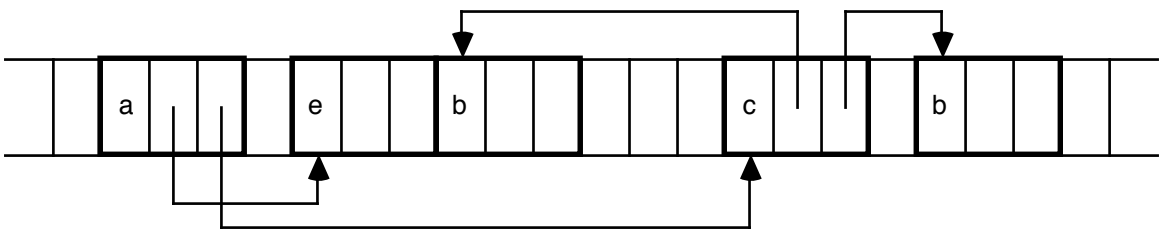


Abb. 10: Verkettete Speichermöglichkeit für Bäume

Auf diese Weise gewinnt man zwei Vorteile:

- Zum einen bestehen die einzelnen Bausteine eines Baumes aus identischen, einfach zu beschreibenden Objekten desselben Grundtyps;
- zum anderen können die Bausteine nun an beliebige freie Stellen im Speicher abgelegt werden, sofern die entsprechende Verkettung durch Zeiger sicher gestellt wird. Ein- und Ausfügen wird dadurch vergleichsweise unkompliziert.

In PASCAL beschreibt man diese Darstellung folgendermaßen:

```
type Baum=record
            marke: markierung;
            lt, rt: ↑Baum
         end.
```

Hierbei ist der Typ ↑Baum ein Datentyp, dessen Wertemenge die Menge aller Verweise auf Werte vom Typ Baum ist. Maschinenorientiert gesehen handelt es sich dabei um Adressen von Speicherzellen oder Gruppen aufeinanderfolgender Speicherzellen, in denen Objekte vom Typ Baum abgelegt werden können. Die Überprüfung, ob in Speicherzellen Objekte des deklarierten Typs in korrekter Weise abgelegt werden, erfolgt bereits zur Übersetzungszeit des Programms.

3.8.1 Zeiger

Allgemeines Definitionsschema für Zeigertypen: Ist T ein beliebiger Datentyp, so bezeichnet refT=(W,R) definiert durch

```
type refT=↑T
```

den zu T gehörigen Zeigertyp. Die Wertemenge W von refT ist die Menge aller Zeiger auf Objekte vom Typ T.

Eine in jedem Zeigertyp vorhandene (also eigentlich polymorphe) Konstante ist das Objekt

nil∈W.

Ein Zeiger mit Wert nil verweist zur Zeit auf kein Objekt (Beachte: nil ist nicht dasselbe wie undefiniert).

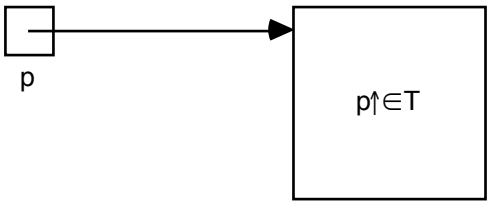
In der Operationsmenge R gibt es drei Standardoperationen für Zeiger: new, dispose und ↑ (Dereferenzierung). Für einen Zeiger

```
var p: refT
```

gilt: Die Prozedur

```
new(p)
```

erzeugt ein (neues) Objekt vom Typ T und weist p eine Referenz auf das Objekt zu (Abb. 11).



... oder als Speicherbild ...

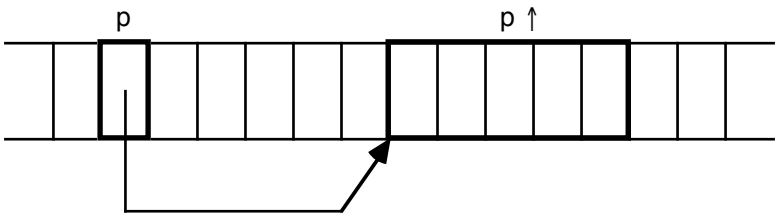


Abb. 11: Wirkung von new

Für den umgekehrten Effekt sorgt dispose:

```
dispose(p)
```

löscht den Speicherbereich, den das Objekt belegt, auf das p weist. Tatsächlich wird dieser Speicherbereich aber meist nicht unmittelbar gelöscht, sondern nur als "gelöscht" markiert. Erst durch einen späteren Lauf eines Speicherbereinigungsalgorithmus (etwa vergleichbar zum Garwick-Algorithmus) wird der Speicherbereich dem freien Speicher zugeordnet und kann dann tatsächlich wiederbelegt werden.

Die (Postfix-)Operation ↑ realisiert die Dereferenzierung eines Zeigerobjekts: Für p∈refT liefert

```
p↑
```

das Objekt, auf das p verweist; also gilt p↑∈T. Mittels p↑ läuft man also anschaulich dem Zeiger nach und landet beim Objekt, auf das verwiesen wird. Zwischen p und p↑ ist daher streng zu unterscheiden, ferner zwischen Anweisungen der Form p:=q und p↑:=q↑. Verboten sind offenbar p:=q↑ und p↑:=q (Warum?).

Beispiel:

```
type T=record
          x: integer
       end;
var p, q, r: ↑T
```

Abb. 12 zeigt die Situation bei Ausführung der folgenden Anweisungen:

```
new(p); p↑.x:=5;
new(q); q↑.x:=7;
new(r); r↑.x:=9;
(* die Situation an dieser Stelle zeigt Abb. 12 links *)
p:=q;
```

```
r↑:=q↑;
(* die Situation an dieser Stelle zeigt Abb. 12 rechts *)
```

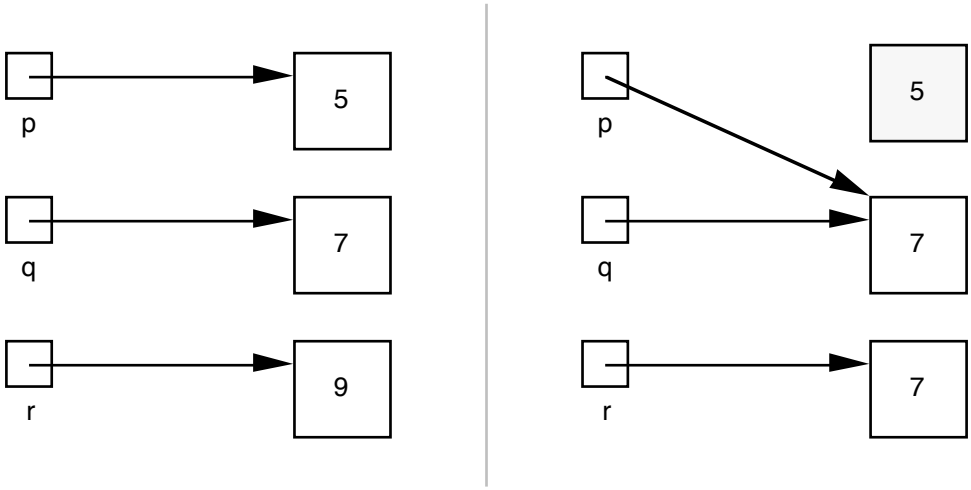


Abb. 12: Programmsituationen

Mittels Zeigern gelingt nun die effiziente Implementierung einer Vielzahl rekursiver Datentypen, die wir im folgenden durchsprechen.

3.8.2 Implementierung von Sequenzen

Eine polymorphe Linkssequenz (oder mit identischer Implementierung auch eine Rechtssequenz) über Objekten vom Typ D der Form

```
typ L(D)≡{leer} | (D,L)
```

implementieren wir als durch Zeiger verkettete Folge von Elementen des Typs D.
In PASCAL also:

```
type T=record
         inhalt: D
         next: ↑T
       end.
```

Jedes Objekt vom Typ T besteht nun aus einer Komponente inhalt vom Typ D, die die eigentliche Information trägt, und aus einem Zeiger, der auf ein weiteres Objekt vom Typ T verweist, den Nachfolger in der Sequenz (Abb. 13).

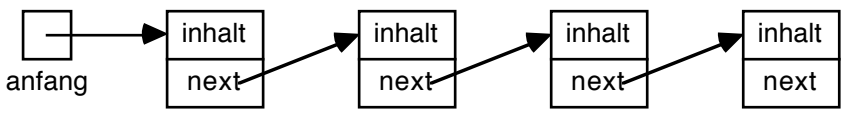


Abb. 13: Implementierung einer Links-/Rechtssequenz als (einfach verkettete) lineare Liste

Typische Operationen auf Sequenzen werden nun in PASCAL durch folgende Programmstücke implementiert. Generell seien dazu die Deklarationen

```
var p,q,r,anfang: ↑T
```

gegeben.

Aufbau einer Liste mit n Elementen, deren jeweiliger Inhalt $t_1,...,t_n$ ist:

```
anfang:=nil;
for i:=n downto 1 do
begin
      new(p);
      p↑.next:=anfang;
      p↑.inhalt:=ti;
      anfang:=p
end.
```

Man beachte, daß die Liste hierbei von hinten nach vorne aufgebaut wird. Eine andere Version wird in den Übungen behandelt.

Einfügen eines Listenelementes mit dem Inhalt t hinter p↑:

```
new(q);
q↑.inhalt:=t;
q↑.next:=p↑.next;
p↑.next:=q.
```

Entfernen des Nachfolgers von p↑:

```
p↑.next:=p↑.next↑.next.
```

Entfernen des Elements p↑ selbst unter der Voraussetzung, daß p↑ einen Nachfolger besitzt:

```
p↑:=p↑.next↑.
```

Beispiel: Einlesen einer beliebigen Folge von Zeichen und Ausgabe in umgekehrter Reihenfolge mithilfe einer linearen Liste:

```
program palindrom(input,output);
type zeichen=record
                    inhalt: char;
                    next: ↑zeichen
             end;
var  p, anfang: ↑zeichen;
     ch: char;
begin
     anfang:=nil;
     while not eof do
     begin
          new(p); p↑.next:=anfang;
          read(ch); p↑.inhalt:=ch;
          anfang:=p
     end;
     while anfang≠nil do
     begin
          write(anfang↑.ch);
          anfang:=anfang↑.next
     end
end.
```

Je nach Art der am häufigsten vorkommenden Operationen verwendet man eine Reihe weiterer Implementierungen von linearen Listen, die zwar etwas mehr Speicherplatz benötigen, dafür aber gewisse Zugriffsoperationen vereinfachen.

Kreisförmige Verkettung.
Der next-Zeiger des letzten Listenelements verweist wieder auf den Anfang der Liste (Abb. 14). Nun ist jedes Element der Liste von jedem anderen Element aus erreichbar. Jedes Element kann nun als Anfangselement dienen.

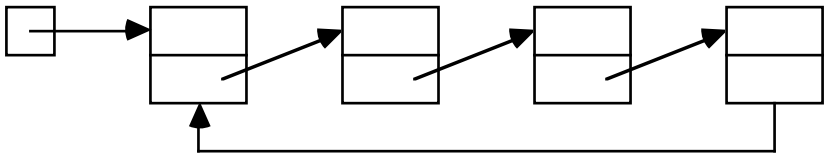


Abb. 14: Kreisförmig verkettete lineare Liste

Doppelte Verkettung.
Jedes Listenelement besitzt zwei Zeiger, einen auf den Vorgänger und einen auf den Nachfolger. Zusätzlich gibt es ein ausgezeichnetes Listenelement, den Kopf, der als Anker der Liste dient, keine Information trägt und nicht gelöscht werden darf (Abb. 15). Bei doppelter Verkettung vereinfacht sich das Ein- und Ausfügen von Listenelementen. Zugleich ist ein beliebiges Vor- und Zurücklaufen möglich. Zur Implementierung wird auf die Übungen verwiesen.

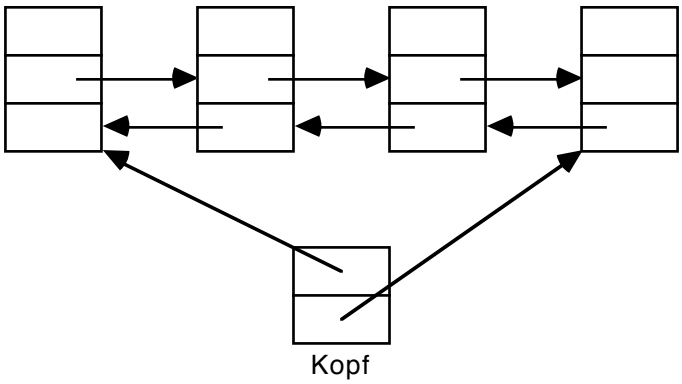


Abb. 15: Doppelt verkettete lineare Liste

3.8.3 Implementierung von Bäumen

Einen ersten Eindruck von einer möglichen Implementierung binärer Bäume haben wir bereits zu Beginn von Abschnitt 3.8 gewonnen. Kümmern wir uns nun um allgemeine Bäume. Da man die Zahl der Söhne eines Knotens nicht im voraus kennt und diese Zahl auch nicht nach oben beschränkt ist, muß man die Söhne in Form einer linearen Liste erfassen (Abb. 16). Oder man schätzt den zu erwartenden durchschnittlichen Grad eines Knotens vorher ab und legt ein Array von Zeigern mit entsprechend vielen Komponenten an. Überschreitet die Zahl der Söhne dann ausnahmsweise die Arraygröße, so erzeugt man einen Hilfsknoten (Abb. 17).

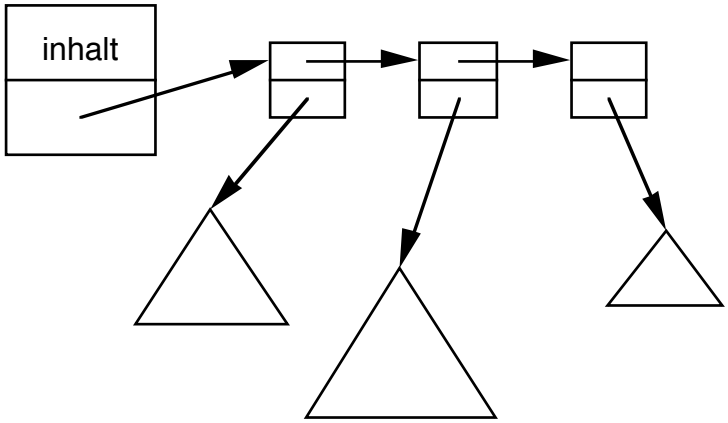


Abb. 16: Implementierung von Bäumen durch lineare Listen

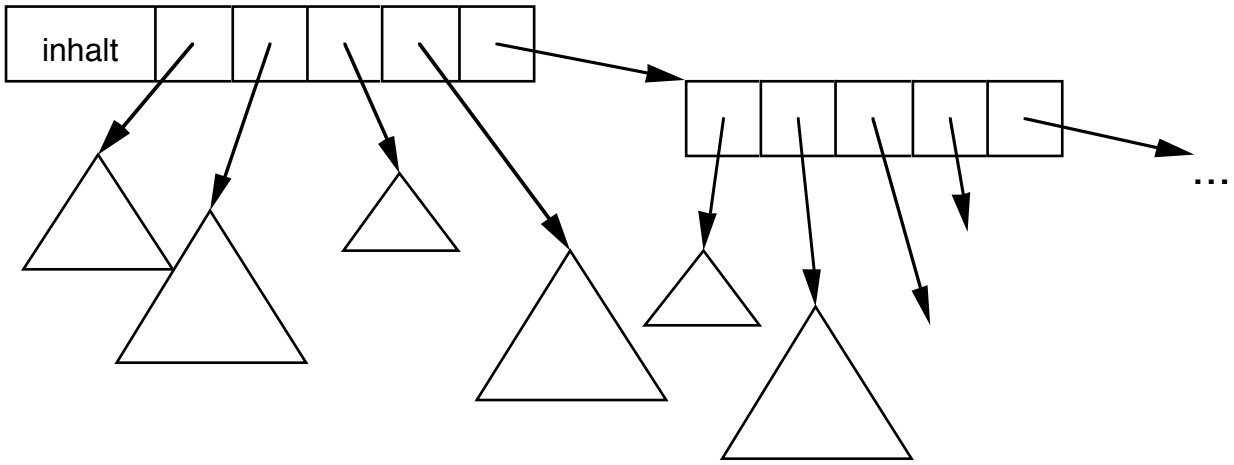


Abb. 17: Implementierung von Bäumen durch Arrays mit Überlauf

Beide Implementierungen besitzen Nachteile bei der Speicherausnutzung. Bei der ersten werden für jeden Knoten mit k Söhnen 2k+1 Zeiger benötigt, bei der zweiten ist eine Vielzahl von Zeigern ungenutzt, nämlich die von den Blättern. Warum?
Sei B ein Baum mit n Knoten, die mit Ausnahme der Blätter jeweils k Söhne besitzen. B besitzt also nk Zeiger. Wegen der n Knoten sind genau n-1 Zeiger ungleich nil. Folglich sind insgesamt n(k-1)+1 Zeiger gleich nil und damit praktisch überflüssig. Schon bei ternären Bäumen sind damit ca. 2/3 der Zeiger überflüssig.
Daher spielen in der Informatik binäre Bäume eine solch große Rolle, weil sie eine bestmögliche Speicherausnutzung sichern. Zugleich gestatten sie es aber, Bäume beliebiger Ordnung effizient zu "simulieren". Die Idee besteht hierbei darin, einen der beiden Verweise umzudefinieren. Der linke Zeiger verweist weiterhin auf den ersten Sohn des Knotens, der rechte jedoch auf den Bruder des Knotens. Diese Idee überträgt sich auch auf Wälder. Formal definiert man:

Sei $F=(T_1,...,T_n)$ ein Wald. Für einen Baum T bezeichne w(T) die Wurzel, für einen Knoten x sei lt(x) der linke und rt(x) der rechte Sohn. Der zu F gehörige binäre Baum B(F) ist wie folgt definiert:

1) Für n=0 ist B(F) der leere Baum.
2) Für n>0 ist

$w(B(F))=w(T_1)$,
$lt(w(B(F)))=B(F')$ mit F'=Wald der Teilbäume von $w(T_1)$,
$rt(w(B(F)))=B(T_2,...,T_n)$.

Beispiel: s. Übungen.

Im folgenden beziehen wir uns also immer nur auf binäre Bäume, die in PASCAL wie folgt definiert seien:

```
type data=...
     knoten=record
                inhalt: data;
                lt, rt: ↑knoten
            end.
```

Ein Objekt

```
var b: ↑knoten
```

repräsentiert dann einen Baum. Für b=nil ist der Baum leer.

Baumdurchlauf.

Wir haben bereits in der Vorlesung Algorithmen, Daten, Programme I verschiedene Algorithmen zum Durchlaufen eines Baumes kennengelernt. Diese übertragen sich in natürlicher Weise auf die obige Implementierung, z.B. für den inorder-Durchlauf:

```
procedure inorder (b: ↑knoten);
begin
    if b<>nil then
    begin
        inorder(b↑.lt);
        write(b↑.inhalt);
        inorder(b↑.rt)
    end
end.
```

Aufbau eines Baumes.

In beinahe umgekehrter Weise kann man binäre Bäume aufbauen. Man ordnet dazu die Knoteninhalte z.B. in preorder-Reihenfolge an, muß aber in dieser Reihenfolge noch die Stellen ergänzen, an denen ein Teilbaum leer ist. Dazu setzen wir einen Bindestrich.

Beispiel: Die Eingabefolge

ABC--DE--FG---HI--JKL--M--N--

ist eine preorder-Darstellung des Baumes aus Abb. 18

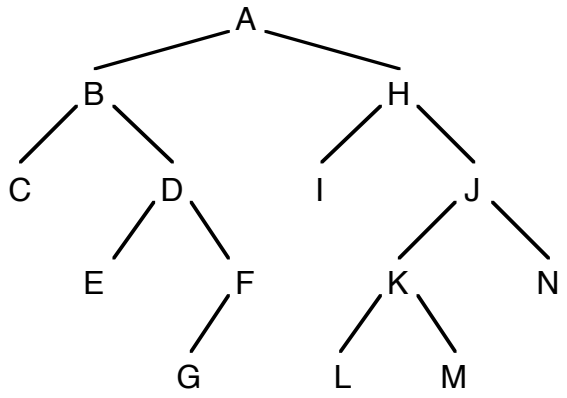


Abb. 18

Den Aufbau leistet nun die Prozedur:

```
procedure Aufbau(var p: ↑knoten);
var ch: char;
begin
      read(ch);
      if ch<>'-' then
      begin
            new(p); p↑.inhalt:=ch;
            Aufbau(p↑.lt);
            Aufbau(p↑.rt)
      end else p:=nil
end.
```

3.8.4 Rekursion im Kontrollbereich

In Abschnitt 10.4.2 der Vorlesung "Algorithmen, Daten, Programme I" haben wir die Formularmaschine vorgestellt, mit der man in systematischer (und automatisierbarer) Weise rekursive Funktionen berechnen kann. Wie implementiert man diese Formularmaschine auf einem Rechner? Schauen wir uns dazu noch einmal das Formular des ggT an.

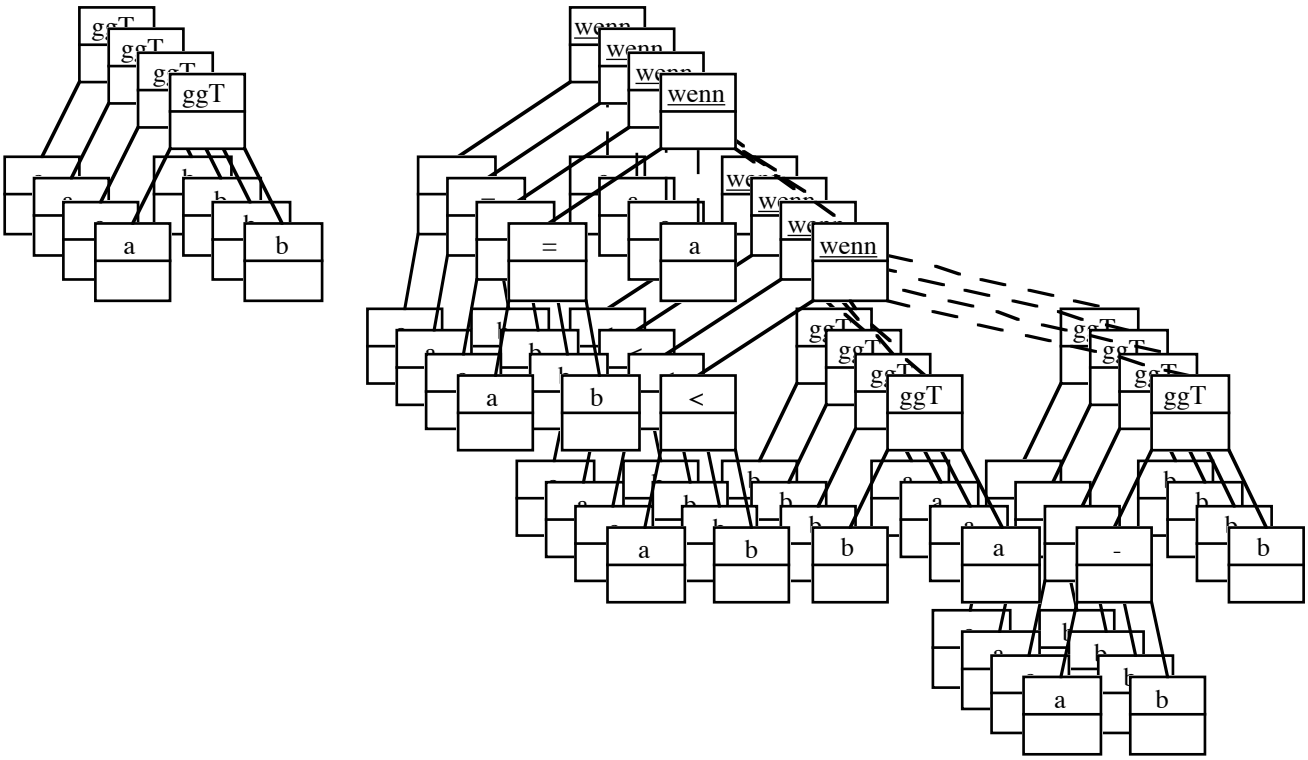


Abb. 19: Formular für ggT

Die Formularmaschine legt für jede Inkarnation der Funktion ein eigenes Formular an. In einer ersten Annäherung an eine effiziente Implementierung können wir alle diese strukturell identischen Formulare zu einem einzigen zusammenfassen, indem wir sie quasi übereinanderlegen (Abb. 19), dann die jeweils oberen Hälften aller Kästchen zu einem vereinigen und nur die jeweiligen Wertezellen für jede Inkarnation zurückbehalten, also die unteren Hälften aller Knoten getrennt halten. An jedem Knoten liegen nun *Folgen* von Werten gleichen Datentyps vor, von denen für die aktuelle Auswertung einer Inkarnation nur die zuletzt eingetragenen Werte jeder Folge von Bedeutung sind.

Man führt also bei jedem Aufruf an jedem Knoten eine neue Wertezelle ein, die man während der Abarbeitung der Inkarnation ausfüllt. Nach Abarbeitung muß man auf die zugehörigen Wertezellen nicht mehr zugreifen und kann sie daher löschen. Dieses Muster an Zugriffsoperationen auf die Werte paßt unmittelbar zum Datentyp Stack. Faßt man alle Wertezellen sowie weitere unten näher beschriebene Informationen einer Inkarnation zu einem Stackelement zusammen, so geht der Aufruf einer Funktion einher mit dem push aller Wertezellen, von denen einige (die aktuellen Parameter) bereits Werte tragen, andere erst im Verlauf der Auswertung der Inkarnation Werte bekommen. Nach Beendigung wird mit einem pop der zugehörige Wertesatz vernichtet.

Der Stack ist damit das entscheidende Hilfsmittel zur effizienten Realisierung von Rekursion im Kontrollbereich.

Wie geht man nun praktisch vor, um eine rekursive Funktion aus FUN in PASCAL zu implementieren?

1. Schritt: Man beseitigt geschachtelte Funktionsaufrufe. Dazu sind ggf. lokale Hilfsvariablen einzuführen.
   Beispiel: Ersetze den Aufruf "f(g(x))" durch "h:=g(x); f(h)".
2. Schritt: Der Rumpf der Funktion wird am Programmanfang eingefügt und erhält zu Beginn eine Marke X als Sprungziel. Eine weitere Marke Y steht unmittelbar hinter dem Rumpf. Vor den Rumpf wird ein Sprung auf Y eingefügt. Der Rumpf soll ja nicht gleich zu Beginn ausgeführt werden.
3. Schritt: Man definiert einen Stack, bei dem jeder Eintrag in Form eines Records alle Parameter, alle lokalen Variablen, eine Variable für den berechneten Funktionswert einer Inkarnation sowie die Rücksprungadresse enthält, zu der nach Abarbeitung der Funktion verzweigt werden muß (direkt hinter der Aufrufstelle). Der Funktionsrumpf arbeitet nur mit dem obersten Stackelement. Eigene Variablen oder Parameter besitzt er nicht.
   Beispiel: Unter Verwendung der allgemeinen Definition von Stacks aus Abschnitt 3.7 definiert man den Typ data für die Funkton ggT durch

```
type data=record
            a,b: integer;          für die Parameter
            marke: (M1,M2,M3);     für die drei möglichen Rücksprung-
                                   marken zu den drei Aufrufen von ggT
            ggT: integer           für den Funktionswert
          end.
```

4. Schritt: Jeder Funktionsaufruf wird in dieser Reihenfolge ersetzt durch
   a) eine Zuweisung der Parameter und der Rücksprungadresse (s. d)) an ein neues Record-Element
   b) eine push-Operation dieses Records auf den Stack
   c) einen Sprung zur Anfangsmarke des Funktionsrumpfs
   d) die Einführung einer Rücksprungmarke.
5. Schritt: Am Schluß des Funktionsrumpfes wird folgende Anweisungsfolge eingefügt:
   a) Auslesen der Rücksprungadresse aus dem obersten Stackelement
   a) Auslesen des Funktionswertes aus dem obersten Stackelement
   b) Entfernen des obersten Stackelementes
   c) Weitergabe des Funktionswertes an das jetzt oberste Stackelement
   d) Sprung zur Rücksprungadresse.

Bemerkung: Nicht alle oben beschriebenen Schritte funktionieren unmittelbar in PASCAL. Dies betrifft vor allem die Verwendung von Marken. Zur konkreten Realisierung schauen Sie am besten in ein PASCAL-Handbuch.

Beispiel: Wir vollziehen die Schritte für die Funktion ggT nach:

```
funktion ggT (a:nat,b:nat) → nat ≡
wenn a=b dann a sonst
     wenn a<b dann ggT (b,a) sonst ggT (a-b,b) ende ende.
```

1. Schritt: Eigentlich ist hier ggT (a-b,b) durch c:=a-b; ggT (c,b) zu ersetzen. Dies können wir uns aber sparen, weil die Subtraktion elementar ist (wir haben in diesen Fällen auch bei der Formularmaschine kein neues Formular angelegt.)

2. Schritt: Aufstellung des Programms:

```
program ggt(input,output);
begin
      goto B;
      A: wenn a=b dann a sonst
            wenn a<b dann ggT (b,a) sonst ggT (a-b,b) ende ende;
      B: readln(a,b);
      h:=ggt(a,b); writeln(h)
end.
```

3. Schritt: Stackdefinition:

```
program ggt(input,output);
type data=record
                  a,b: integer;
                  marke: (M1,M2,M3);
                  ggt: integer
            end;
      stack=... <übliche Definition unter Verwendung von data>
begin
      goto B;
      A: if a=b then a else
            if a<b then ggT (b,a) else ggT (a-b,b);
      B: readln(a,b);
      h:=ggt(a,b); writeln(h)
end.
```

4. und 5. Schritt: Modifikation der Aufrufe und Beendigung des Funktionsrumpfes:

```
program ggt(input,output);
type data=record
                  a,b: integer;
                  marke: (M1,M2,M3);
                  ggt: integer
            end;
      stack=... <übliche Definition unter Verwendung von data>
var   x: data;
      s: stack;
begin
      goto B;
      A: if s.sp[s.top].a=s.sp[s.top].b then s.sp[s.top].ggT:=s.sp[s.top].a else
            if s.sp[s.top].a<s.sp[s.top].b then
            begin
                  x.a:=s.sp[s.top].b; x.b:=s.sp[s.top].a; x.marke:=M1; push(x,s);
                  goto A;
                  M1:
            end else
            begin
                  x.a:=s.sp[s.top].a-s.sp[s.top].b; x.b:=s.sp[s.top].b;
                  x.marke:=M2; push(x,s);
                  goto A;
                  M2:
            end;
      x:=top(s);
      pop(s);
      s.sp[s.top].ggT:=x.ggT
      goto x.marke;
```

```
        B: readln(a,b);
        x.a:=a; x.b:=b; x.marke:=M3; push(x,s);
        goto A;
        M3:
        h:=s.sp[s.top].ggT; writeln(h)
end.
```

Die kritische Größe bei der Implementierung rekursiver Funktionen ist offenbar der Stack s, zum einen wegen des Speicherbedarfs zum anderen wegen der Laufzeit für die Zugriffsoperationen auf den Stack. Bevor man also rekursive Funktionen verwendet und den Übersetzer mit der Realisierung der obigen Implementierungsvorschrift beauftragt, sollte man sich vergewissern, daß die Rekursionstiefe und -breite, die der Größe des Stacks entspricht, einen relativ zum Problem vernünftigen Umfang nicht überschreitet. Eine solche Analyse führt dann oftmals auch zu einem nicht-rekursiven Algorithmus, der dann jedoch manchmal weniger gut lesbar ist als der rekursive, weil seine Struktur die rekursive Struktur des Problems weniger deutlich widerspiegelt. Ein extremes Beispiel ist die Fibonacci-Funktion mit

f(0)=0,

f(1)=1,

f(n)=f(n-1)+f(n-2) für n≥2,

bei der die Größe des Stacks exponentiell mit n wächst. Andererseits gibt es aber auch eine effiziente nicht-rekursive Lösung, die ebenso übersichtlich ist wie die rekursive.

Es gibt eine Vielzahl von Funktionen, die man ohne Stack implementieren kann, wobei die nicht-rekursive Lösung der rekursiven hinsichtlich Übersichtlichkeit und Systematik in nichts nachsteht. Zu dieser Klasse gehören u.a. diejenigen Funktionen, bei denen der rekursive Aufruf stets am Anfang (*head recursion*) oder stets am Ende (*tail recursion*) erfolgt. Diese Funktionen lassen sich durch eine einfache while- oder repeat-Schleife implementieren.

Genauer: Das Funktionsschema

```
funktion P x → ... ≡
wenn B dann a sonst g(x,P(f(x))) ende
```

mit der *kommutativen* Funktion g und dem Aufruf P(y) ersetzt man durch das Schleifenschema

```
P:=a; j:=y;
while not B do
begin
        P:=g(j,P);
        j:=f(j)
end.
```

Beispiel: Für die Fakultätsfunktion

```
funktion fak x: nat→nat ≡
wenn x=0 dann 1 sonst x*fak (x-1) ende.
```

erhält man für den Aufruf fak n das Programmstück

```
fak:=1; j:=n;
while not (x=0) do
begin
```

```
        fak:=j*fak;
        j:=j-1
end.
```

3.9 Graphen

Ein Graph ist ein anschauliches mathematisches Modell zur Beschreibung von Objekten, die untereinander in gewisser Beziehung stehen. Beispiele sind chemische Strukturformeln, Verkehrsnetze oder Verwandschaftsbeziehungen (Abb. 20).

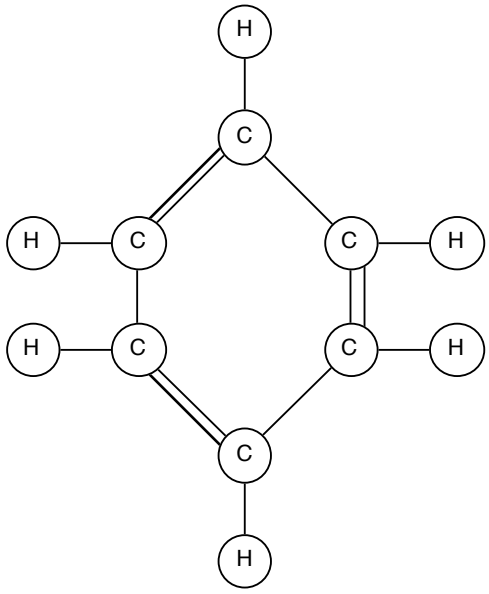


Abb. 20: Chemische Strukturformel für Benzol

Die erstmalige Benutzung von Graphen läßt sich recht genau zurückverfolgen: Es war das Königsbereger Brückenproblem, das L. Euler 1736 löste: In der Innenstadt von Königsberg vereinen sich der Alte und der Neue Pregel zum Pregelfluß. Im 18. Jahrhundert führten über die Flußläufe sieben Brücken, die das Süd-, Nord-, Ost- und Inselgebiet miteinander verbanden (Abb. 21). Angenommen man befindet sich auf irgendeinem der Gebiete N, S, I oder O. Gibt es einen Weg vom Ausgangspunkt, bei dem man jede Brücke genau einmal passiert und schließlich zum Ausgangsort zuürckkehrt? Zwar ist das Problem endlich und daher durch Aufzählung aller Wege zu lösen, Euler fand jedoch einen eleganteren Zugang und zeigte, daß es keinen solchen Weg gibt, indem er das Problem auf ein Graphenproblem zurückführte. Er ersetzte jede Landmasse N, S, I und O durch einen Knoten im Graphen und jede Brücke zwischen zwei Gebieten durch eine Kante zwischen den zugehörigen Knoten. Abb. 22 zeigt den entsprechenden Graphen.

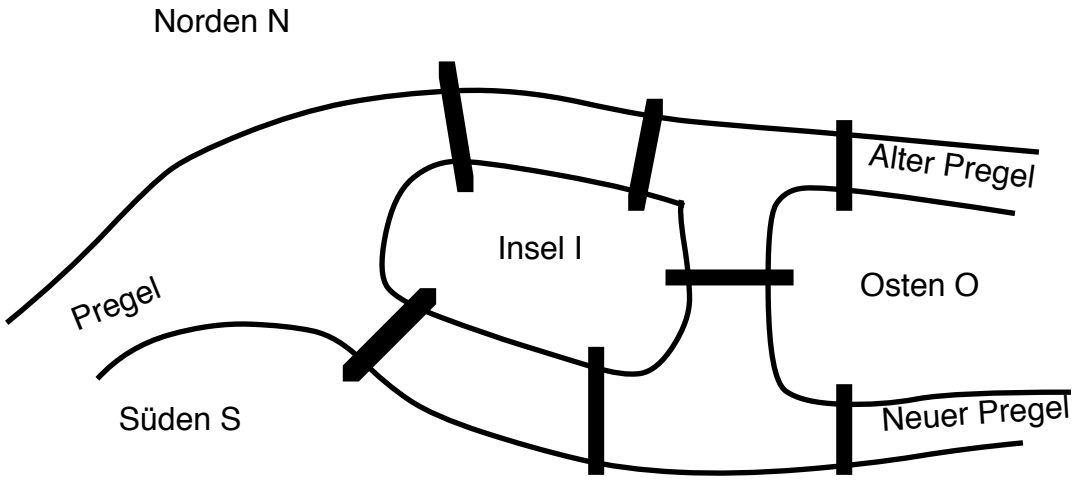


Abb. 21: Königsberger Brückenproblem

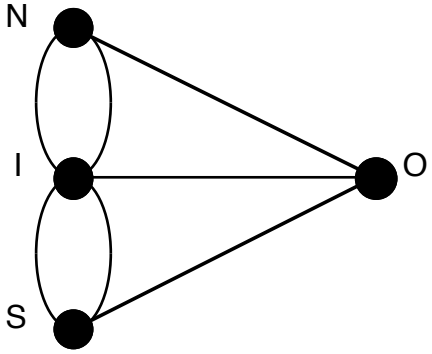


Abb. 22: Graph zum Königsberger Brückenproblem

Auf Graphen übertragen lautet das Königsberger Brückenproblem: Gibt es zu jedem Knoten einen Weg, der jede Kante genau einmal durchläuft und am Ende wird am Ausgangsknoten ankommt? Einen solchen Weg nennt man *Eulerscher Kreis*.
Euler bewies für beliebige Graphen, daß ein Eulerscher Kreis dann und nur dann existiert, wenn der Grad jedes Knotens eine gerade Zahl ist, d.h. wenn an jedem Knoten eine gerade Anzahl von Kanten anliegt. Der Graph in Abb. 22 erfüllt diese Bedingung nicht. Folglich existiert auch der gesuchte Rundweg über die Brücken des Pregel nicht.
Das Gegenstück, dessen Lösung wesentlich schwieriger ist und für das man bis heute keine exakte Charakterisierung kennt, ist das *Hamiltonsche Kreisproblem*, bei dem es darum geht, jeden Knoten (statt jeder Kante) genau einmal zu besuchen.
Nun zu den genauen Definitionen.

Definition B:
Sei $V \neq \emptyset$ eine endliche Menge und E eine Menge von ein- und zweielementigen Teilmengen von V. Dann heißt G=(V,E) **ungerichteter Graph**. V (oder V(G), wenn Unterscheidungen nötig sind) ist die Menge der **Knoten**, E (oder E(G)) die Menge der **Kanten**. Ist $\{x,y\} \in E$ eine Kante, so sind x und y **adjazent**. Der **Grad** d(v) eines Knotens v ist definiert durch $d(v)=|\{u \in V \mid \{u,v\} \in E\}|$.

Sei $P=(v_0,v_1,...,v_{n-1},v_n)$ ein (n+1)-Tupel von Knoten von G. P ist ein **Weg** der Länge n zwischen v_0 und v_n, wenn $\{v_{i-1},v_i\}\in E$ für $1\le i\le n$. P heißt **einfach**, wenn $v_i\neq v_j$ für $0\le i<j\le n$. Ein Weg $P=(v_0,v_1,...,v_{n-1},v_n)$ heißt **Zyklus** der Länge n, falls $n\ge 3$, $v_n=v_0$ und $(v_0,v_1,...,v_{n-1})$ ein einfacher Weg ist. Ein Graph ist **azyklisch**, wenn er keinen Zyklus enthält. u und v sind **verbunden**, falls es einen Weg zwischen u und v gibt. G ist **zusammenhängend**, wenn jedes Paar $x,y\in V$ verbunden ist.
Ein Graph G'=(V',E') heißt **Teilgraph** von G, falls $V'\subseteq V$ und $E'\subseteq E$ ist. Ein zusammenhängender Teilgraph von G heißt **Zusammenhangskomponente**.
Für eine Teilmenge $V'\subseteq V$ ist $G\backslash V'$ definiert als der Graph $(V\backslash V',\{\{u,v\}\in E \mid u,v\in V\backslash V'\})$.
Ein Graph G=(V,E) heißt **gerichteter Graph**, wenn $E\subseteq V\times V$ ist. Man unterscheidet hier den **Eingangsgrad** $d^+(v)=|\{(u,v) \mid (u,v)\in E\}|$ und den **Ausgangsgrad** $d^-(v)=|\{(v,u) \mid (v,u)\in E\}|$.
Die Begriffe von Weg, Zyklus, azyklisch usw. übertragen sich von ungerichteten auf gerichtete Graphen sinngemäß.

Beispiel: Abb. 23 zeigt den ungerichteten Graphen G=(V,E) mit V={a,b,c,d} und E={{a,c},{a,d},{b,d},{b},{c,d}}, Abb. 24 den gerichteten Graphen G=(V',E') mit V'=V und E'={(a,c),(d,a),(b,d),(b,b),(d,c)}.

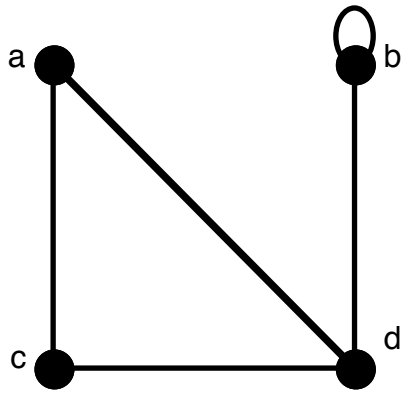


Abb. 23: Ungerichteter Graph

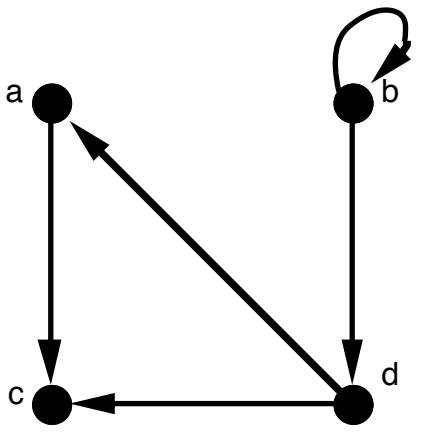


Abb. 24: Gerichteter Graph

Implementierung von Graphen.

Eine Implementierung von Graphen sieht vor, daß die Knoten beginnend bei 1 fortlaufend durchnumeriert werden. Die Kantenbeziehungen werden durch eine boolesche Matrix, die *Adjazenzmatrix*, dargestellt. Bei n Knoten definiert man diese Adjazenzmatrix $A=(a_{ij})$ durch

$$a_{ij} = \begin{cases} \text{true}, & \text{falls } \{i,j\}\in E \text{ im ungerichteten Graphen bzw.} \\ & (i,j)\in E \text{ im gerichteten Graphen} \\ \text{false}, & \text{sonst.} \end{cases}$$

Bei ungerichteten Graphen ist die Adjazenzmatrix immer symmetrisch, d.h. $a_{ij}=a_{ji}$ für alle i,j.

Beispiel: Die Adjazenzmatrix der Graphen aus Abb. 23 und 24 lauten

$$A = \begin{matrix} \text{false} & \text{false} & \text{true} & \text{true} \\ \text{false} & \text{true} & \text{false} & \text{true} \\ \text{true} & \text{false} & \text{false} & \text{true} \\ \text{true} & \text{true} & \text{true} & \text{false} \end{matrix} \qquad A' = \begin{matrix} \text{false} & \text{false} & \text{true} & \text{false} \\ \text{false} & \text{true} & \text{false} & \text{true} \\ \text{false} & \text{false} & \text{false} & \text{false} \\ \text{true} & \text{false} & \text{true} & \text{false} \end{matrix}$$

Der Nachteil von Adjazenzmatrizen ist der hohe Speicherplatzbedarf *unabhängig* von der Größe des Graphen: es werden stets $|V|^2$ Speicherzellen benötigt, auch wenn der Graph nur wenige oder überhaupt keine Kante besitzt. In der Praxis werden verkettete Darstellungen bevorzugt, bei denen ein Speicherplatzbedarf entsteht, der nur linear mit der Größe des Graphen wächst, also ungefähr $c(|V|+|E|)$ für eine Konstante c beträgt. Diese Eigenschaft besitzt die Implementierung durch *Adjazenzlisten*. Hierbei werden die Knoten in irgendeiner Reihenfolge linear verkettet; an jedem Knoten hängt eine Liste von Verweisen auf die zu diesem adjazenten Knoten. Abb. 25 zeigt eine Darstellung des Graphen aus Abb. 24.

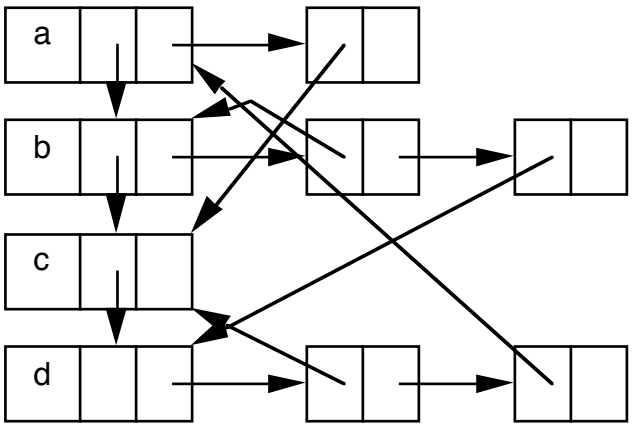


Abb. 25: Adjazenzlistendarstellung zum Graphen aus Abb. 24

Durchlaufen von Graphen.

Zweck ist das systematische Durchsuchen eines Graphen und ggf. die Ausgabe aller Knotenmarkierungen.

Alle Durchlaufverfahren durch einen Graphen G=(V,E) benötigen einen Startknoten v∈V. Ausgegeben werden sollen alle Knoten, die mit v verbunden sind. Wir gehen im folgenden von einem zusammenhängenden Graphen G aus. Besteht ein Graph aus mehreren Komponenten, so muß der Algorithmus für jede einzelne Komponente aufgerufen werden. Wir unterscheiden zwei wesentlich verschiedene Durchlaufsarten.

Der **Tiefendurchlauf** (**depth first search**, Abk. **dfs**) besucht ausgehend von einem Knoten zunächst einen beliebigen adjazenten Knoten, macht von dort rekursiv weiter und besucht zu diesem Knoten einen beliebigen adjazenten Knoten usw. Erst wenn von einem Knoten kein unbesuchter Nachbar mehr gefunden wird, geht der Algorithmus zurück und versucht von bereits besuchten Knoten unbesuchte Nachbarn zu finden. Der Algorithmus geht also zunächst in die "Tiefe" des Graphen und erst dann in die "Breite".

Beim **Breitendurchlauf** (**breadth first search**, Abk. **bfs**) geht man von einem Knoten, der gerade besucht wird, zunächst zu allen adjazenten Knoten, bevor deren adjazente Knoten besucht werden. Man geht also zunächst in die "Breite" und erst dann in die "Tiefe" des Graphen.

Sei G=(V,E) ein Graph mit |V|=n Knoten. Wir verwenden eine Pseudo-PASCAL-Notation, die die Struktur der Algorithmen besser deutlich werden läßt. In beiden Fällen benötigen wir zur Buchhaltung der besuchten Knoten ein Hilfsarray

```
var   besucht: array [V] of boolean.
```

Zur Implementierung des dfs:

```
var   besucht: array [V] of boolean;
procedure dfs(v: knoten);
begin
      besucht[v]:=true; write(v);
      for all w∈V: {v,w}∈E do
            if not besucht[w] then dfs(w)
end.
```

Zur Implementierung des bfs: Kern des Algorithmus ist eine Queue, in die man im Laufe der Verarbeitung jeweils alle adjazenten Knoten eines besuchten Knotens einträgt. Die Knoten werden dann stets in der Reihenfolge besucht, in der sie in der Queue abgelegt sind:

```
var   besucht: array [V] of boolean;
      q: queue of V;
procedure bfs(v: knoten);
begin
      besucht[v]:=true;
      enter(v,q);
      while not is_empty(q) do
      begin
            v:=first(q); write(v);
            remove(q);
            for all w∈V: {v,w}∈E do
                  if not besucht[w] then
                  begin
                        besucht[w]:=true;
                        enter(w,q)
```

```
                end
            end
        end.
```

Bei geeigneter Implementierung durch Adjazenzlisten benötigen beide Algorithmen proportional zur Größe von G (=|V|+|E|) viele Schritte und ebenso viel Speicher (für den Stack bzw. die Queue).

Beispiel: Für den Graphen aus Abb. 26 sind u.a. folgende Durchläufe möglich:
dfs: 1 2 9 10 6 5 3 8 4 7
bfs: 1 2 4 6 9 3 7 8 5 10
In beiden Fällen hängt die genaue Besuchsreihenfolge von der Implementierung der Anweisung for all ab.

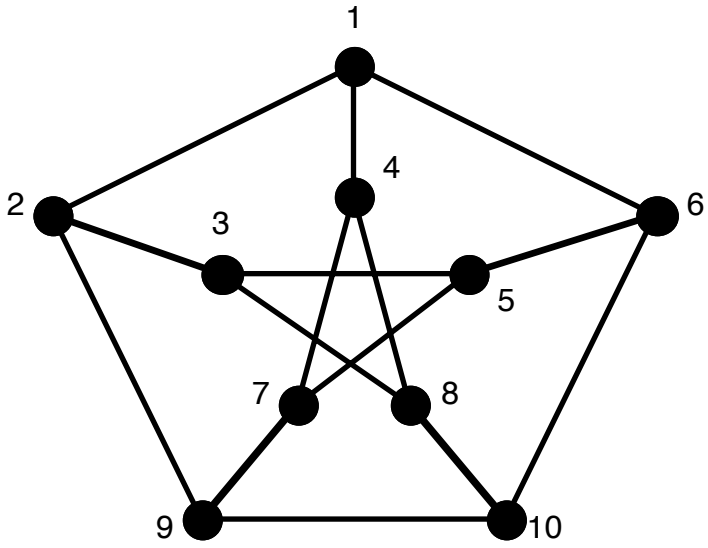


Abb. 26: Graph

3.10 Schlußbeispiel: Topologisches Sortieren

Wir wollen in diesem Abschnitt zusammenfassend eine Reihe von Begriffen an einem größeren Beispiel erläutern; dazu gehört die Anwendung von Graphen, Queues, verketteten Strukturen usw.

Beim topologischen Sortieren handelt es sich um eine Form des Sortierens, bei der auf der Menge $M=\{a_1,...,a_n\}$ der zu sortierenden Elemente keine vollständige sondern nur eine partielle Ordnung vorliegt, d.h. nicht alle Elemente sind miteinander vergleichbar. Ziel ist es, diese partiell geordnete Menge so anzuordnen, daß für alle $i,j\in\{1,...,n\}$ mit $i\neq j$ aus $a_i<a_j$ immer $i<j$ folgt.
Beispiel: Gegeben sei die Menge $M=\{a_1,...,a_9\}$ mit $a_1<a_3$, $a_3<a_7$, $a_7<a_4$, $a_7<a_5$, $a_4<a_6$, $a_9<a_2$, $a_9<a_5$, $a_2<a_8$, $a_5<a_8$, $a_8<a_6$. Eine topologische Sortierung lautet:

$a_1,a_3,a_7,a_4,a_9,a_2,a_5,a_8,a_6$.

Eine andere Sortierung ist:

$a_9,a_1,a_2,a_3,a_7,a_5,a_8,a_4,a_6$.

Anwendungen:

1) *Projektplanung*. Ein Projekt, z.B. ein Hausbau, besteht aus einer Reihe von Arbeitsvorgängen, die teils voneinander abhängig teils unabhängig durchgeführt werden können. Man ordne die Arbeitsgänge zeitlich so an, daß kein Arbeitsgang beginnt, bevor nicht alle Vorlaufarbeitsgänge abgeschlossen sind.
2) *Schreiben eine Lehrbuchs*. In einem Lehrbuch werden verschiedene Begriffe definiert. Teilweise greifen Definitionen auf andere Begriffe zurück. Diese müssen also schon vorher eingeführt sein. Man ordne die Begriffsdefinitionen so an, daß jede Definition nur bereits bekannte Begriffe verwendet.
3) *Studienplan*. Vorlesungen bauen aufeinander auf, oder sie können ohne Vorkenntnisse besucht werden. Man ordne die Vorlesungen zeitlich so an, daß alle zu einer Vorlesung notwendigen Kenntnisse in früheren Vorlesungen erworben werden können.

Partielle Ordnungen stellt man häufig durch gerichtete Graphen dar. Die Elemente der Menge M werden durch Knoten repräsentiert; vom Knoten v zum Knoten w verläuft eine gerichtete Kante, wenn v<w gilt, genauer:

$$G=(V,E) \text{ mit } V=M,\ E=\{(v,w) \mid v<w\}.$$

Der entstehende Graph ist immer azyklisch, denn sonst läge keine Ordnung vor, weil eines der Axiome für Ordnungen verletzt wäre (welches?).

Beispiel: Abb. 27 zeigt die Darstellung der partiellen Ordnung aus obigem Beispiel als Graph, Abb. 28 enthält eine topologische Sortierung.

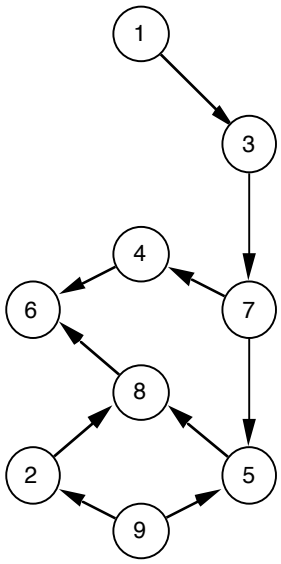


Abb. 27: Menge M mit partieller Ordnung als Graph

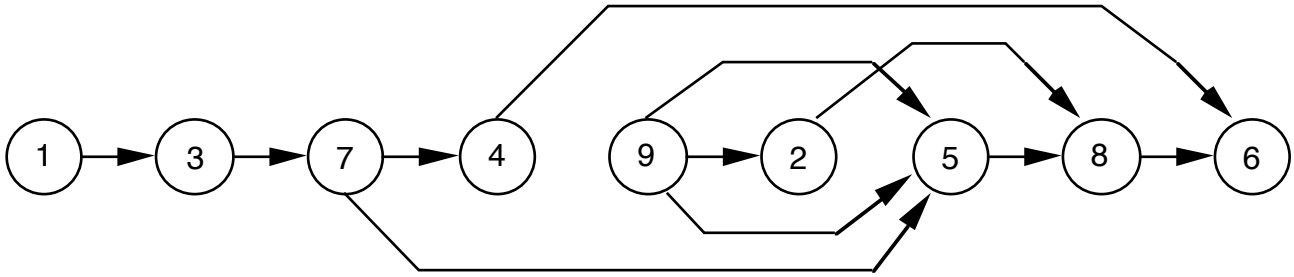


Abb. 28: topologische Sortierung von M als Graph

Eine einfache Lösungsidee besteht darin, einen Knoten ohne Vorgänger, also mit Eingangsgrad 0 zu suchen. Dieser Knoten existiert immer, weil der Ausgangsgraph azyklisch ist. Bezogen etwa auf eine Projektplanung, kann dieser Vorgang in jedem Fall als erstes ausgeführt werden. Anschließend entfernt man den Knoten (und alle ausgehenden Kanten) und wendet das Verfahren auf den restlichen Graphen erneut an, der offenbar ebenfalls azyklisch ist.

Der Algorithmus in Pseudo-PASCAL:

```
procedure topsort (G=(V,E): graph);
begin
    if V≠∅ then
    begin
        Wähle v∈V mit d+(v)=0;
        writeln(v);
        topsort(G\{v})
    end
end.
```

Implementierung.

Um zu einem möglichst effizienten Algorithmus zu kommen, muß man eine geeignete Datenstruktur wählen und vor allem den o.g. Schritt "Wähle $v \in V$ mit $d^+(v)=0$" geschickt implementieren, indem man dafür sorgt, daß der Graph nicht nach jedem Löschen eines Knotens erneut vollständig nach Knoten mit Eingangsgrad 0 durchsucht werden muß. Die Idee besteht darin, zu Beginn einmalig alle Eingangsgrade zu bestimmen und dann bei jedem Löschen eines Knotens nur die Eingangsgrade der adjazenten Knoten zu aktualisieren, denn offenbar bleiben die Eingangsgrade aller übrigen Knoten durch die Löschung unbeeinflußt. Die jeweils vorliegenden Knoten mit Eingangsgrad 0 werden in einer Queue abgelegt, nacheinander verarbeitet und aus der Queue entfernt.

Dazu benötigt man ein Array zu Abspeicherung der jeweiligen Eingangsgrade aller Knoten.

Der Algorithmus in Pseudo-PASCAL:

```
var   d+: array [V] of integer;
      q: queue of V;
begin
      read(G=(V,E));
      empty(q);
      for all v∈V do d+(v):=0;
      for all v∈V do
      begin
```

```
            for all w∈V: (w,v)∈E do d+(v):=d+(v)+1;
            if d+(v)=0 then enter(v,q)
        end;
        while not is_empty(q) do
        begin
            v:=first(q); remove(q);
            writeln(v,N);
            for all w∈V: (v,w)∈E do
            begin
                d+(w):=d+(w)-1;
                if d+(w)=0 then enter(w,q)
            end
        end
end.
```

Im Moment können wir uns nur davon überzeugen, daß der obige Algorithmus gegenüber dem zuvor angegebenen relativ effizient ist. Wir können seine Effizienz aber noch nicht genau bestimmen. Dazu fehlt uns noch ein exaktes numerisches Maß, um Algorithmen genauer zu vergleichen. Dieses Maß und die zugehörigen Meßverfahren werden wir im nächsten Kapitel einführen.