Universität Leipzig
Institut für Informatik

Studienjahr 1996/97

Vorlesung
Grundkurs Praktische Informatik

S. Gerber

Inhaltsverzeichnis

I Digitale Informationsverarbeitung

1. Historische Entwicklung

700 v.Chr.	Abakus (computare, counter, chu pan)
80 v.Chr.	Räderwerk von Antikythera, Differentialgetriebe
500 n. Chr.	Stellenwertsystem mit arabischen Ziffern (Orient)
1250	Lullus (Algorithmiker)
1500	Dezimalsystem mit arabischen Ziffern (Spanien)
1620	Nepersche Rechenstäbe (Multiplikation)
1623	Schickard, Addiermaschine mit Übertrag
1641	Pascal, Addiermaschine mit 8 Rechenstellen
1650	Patridge, Rechenschieber (Analogrechner)
1674	Leibniz, Rechenmaschine für vier Grundrechenarten mit Staffelwalze und Übertragsschlitten
1680	Leibniz, ars magna, Algorithmenbegriff, Logik, Kalkülaufbau
1730	Falcon, automatischer Webstuhl, feste Programmsteuerung
1823	Babbage, difference engine, Programmsteuerungsprinzip
1890	Hollerith, Lochkartenmaschine mit steckbarem Programm
1934	Zuse, programmierbarer Rechner auf Relaisbasis, Gleitpunktarithmetik
1944	Aiken, Mark I, sequentieller Rechner
1946	Neumann, Goldstine, Princeton-Maschine, speicherprogrammiert
	Mauchly, Eniac, Rechenautomat mit Elektronenröhren
1954	Rechner der 1. Generation, IBM 650 (USA), Zuse 1-3 (Hersfeld), Oprema (Jena), Assemblersprachen, FORTRAN (formula translator)
1957	Rechner der 2. Generation mit Transistoren, IBM 1401 (USA)
1959	COBOL als Programmiersprache für kommerzielle Aufgaben
1960	ALGOL als algorithmische Programmiersprache
	ZRA 1 (Zeiss Jena), D1-D3 (Technische Hochschule Dresden)
1964	BASIC als Programmiersprache für Anwender
1970	Rechner der 3. Generation mit integrierten Schaltungen
	PASCAL als Programmiersprache für Ausbildung und Forschung
1972	Programmiersprache C
1975	Mikroprozessortechnik
1980	Rechnernetze, Multiuser-Systeme, Supercomputer, Personal-Computer,
	Diskettenspeicher, Datenbanken, Objektorientierte Sprachen (Smalltalk)
1986	Notebook-Computer, Parallel-Rechner, Festplatten, Display-Schirme
1990	CD-ROM, GB-Platten, Palmtops, Internet, Electronic-Mail
1994	Multimediale Systeme, Netzsprachen, Mobil-Kommunikation

2. Algorithmenbegriff

Beispielalgorithmus

Um 1700 v. Chr. fordert der König Minos von Kreta von der Stadt Athen jährlich einen Tribut von sieben Jungfrauen und sieben Jünglingen als Opfer für den Minotaurus, einem Fabelwesen mit menschlichem Körper und Stierkopf, der in einem Labyrinth auf Kreta haust und viel Unheil über die Insel bringt.
Theseus, der Sohn des Königs Ägeus von Athen, will diesen Minotaurus besiegen und reist mit dem nächsten Opfertribut nach Kreta. Ariadne, die Tochter des Königs Minos, verliebt sich alsbald in Theseus und will ihm beim Kampf gegen Minotaurus helfen. Damit sich Theseus in dem unbekannten Labyrinth nicht verirrt und den Kampf gegen Minotaurus erfolgreich bestehen kann, übergibt sie ihm den berühmten Faden und das Zauberschwert.
Mit Hilfe des Fadens findet Theseus den Minotaurus, kann ihn mit dem Zauberschwert töten und gelangt über den Faden sicher zu Ariadne zurück, die ihn am Eingang des Labyrinths freudig in ihre Arme schließt.
Es erhebt sich die Frage, wie muß Theseus verfahren, um in dem unbekannten Labyrinth den Minotaurus aufzuspüren und wieder sicher zu Ariadne zurückzukehren.
Das Labyrith stellen wir uns als eine endliche Menge von Knotenpunkten vor, die durch Strecken (Gänge) verbunden sind. Wir wählen einen Knoten als Startknoten, auf dem Ariadne steht. Man finde nun ein Verfahren, mit dem man für beliebige Labyrinthe feststellen kann, ob der Knoten, auf dem sich der Minotaurus eventuell befindet, auf einem von Ariadne ausgehenden Weg aus Strecken erreichbar ist, dieser aufgesucht werden kann und man auf einem kreisfreien Weg, wo jeder Knoten höchstens einmal durchlaufen wird, wieder zu Ariadne zurückfindet.

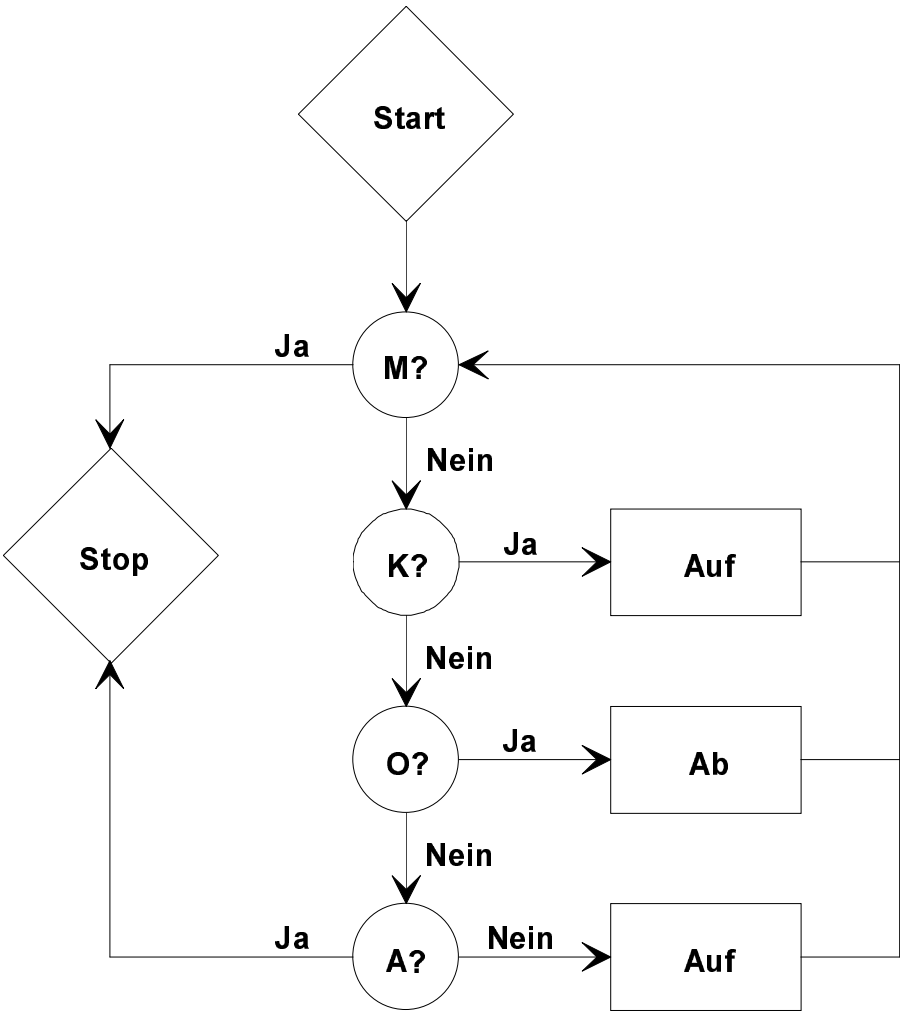
Aktionen und Eigenschaften:

1. Theseus- Aktionen:
   - Ab - Theseus wickelt Faden auf noch nicht durchlaufener Strecke (0-Strecke) ab.
   - Auf - Theseus nimmt Faden auf einer bereits durchlaufenen Strecke auf.

2. Streckeneigenschaften:
   - 0- noch nicht durchlaufen
   - 1- einmal durchlaufen
   - 2- zweimal durchlaufen

3. Knoteneigenschaften:
   - A Knoten auf dem sich Ariadne befindet.
   - M Knoten auf dem der Minotaurus sitzt.
   - O Knoten von dem mindestens eine noch nicht durchlaufene Strecke ausgeht
   - K Knoten über den schon der Faden läuft (Kreisweg).

4. Verfahren:



Behauptungen:

1. *Theseus erreicht nach endlich vielen Aktionen Ariadne oder Minotaurus (Termination).*
2. *Vom Minotaurus gelangt Theseus entlang dem Faden auf einem kreisfreien Weg zu Ariadne zurück.*
3. *Bei Halt auf Ariadne ist Minotaurus nicht erreichbar.*

Zum Beweis dieser Behauptungen bietet es sich an, zunächst folgende Hilfsbehautungen nachzuweisen.

Hilfsbehauptung I: Nach jeder Aktion von Theseus, trifft einer der beiden Fälle zu:

a) Wenn keine 1-Strecken existieren, steht Theseus bei Ariadne.

b) Wenn 1-Strecken existieren, dann bilden diese einen kreisfreien Weg von Theseus zu Ariadne.

Hilfsbehauptung II: Theseus durchläuft keine 2-Strecke.

Beweis der Hilfsbehauptung I:
(Vollständige Induktion über die Anzahl der Theseus-Aktionen)

Induktions-Anfang: Nach 0-Aktionen steht Theseus bei Ariadne und es gibt noch keine 1-Strecken, d.h. Fall a) liegt vor.

Induktions-Annahme: Nach n-Aktionen trifft Fall a) oder b) zu.

Induktions-Schluß: Bei der (n+1)-ten Aktion sind folgende Fälle zu unterscheiden:

- Nach n Aktionen lag Fall a) vor, d.h. Theseus steht bei Ariadne. Dann durchläuft Theseus mit der Aktion Ab eine von Ariadne ausgehende 0-Strecke und erreicht einen Knoten, der nach dieser Aktion mit Ariadne durch eine 1-Strecke verbunden ist, d.h. Fall b) liegt vor, oder es sind keine 0-Stecken vorhanden und Theseus bleibt bei Ariadne, d.h. Fall a) liegt vor.
- Nach n Aktionen lag Fall b) vor, d.h. der Knoten, auf dem Theseus steht, ist mit Ariadne durch einen kreisfreien Weg aus 1-Strecken verbunden.
  Entsprechend der Eigenschaft dieses Knotens hat Theseus folgende Aktionsmöglichkeit:

bei M-Eigenschaft	Stop, d.h. Fall b) bleibt erhalten.
bei K-Eigenschaft	Auf, d.h. kreisfreier Weg an 1-Strecken wird verkürzt und die dabei durchlaufene Stecke wird zur 2-Strecke. Steht danach Theseus bei Ariadne liegt der Fall a vor, sonst Fall b).
bei O-Eigenschaft	Ab, d.h. kreisfreier Weg aus 1-Strecken wird verlängert. Es entsteht Fall b).
bei A-Eigenschaft	Es entsteht ein Kreisweg und Knoten hat K-Eigenschaft, d.h. Reaktion wie bei K-Eigenschaft.
sonst	Auf, d.h. analog K-Eigenschaft.

Nach n+1 Aktionen trifft also wieder Fall a) oder b) zu. Demnach gilt dies nach jeder endlichen Anzahl von Aktionen.

Beweis der Hilfsbehauptung II: Angenommen, Theseus durchläuft eine 2-Strecke. Dann könnte er nur die Aktion Auf ausführen. Die Strecke müßte also 1-Strecke sein, was im Widerspruch zur Annahme steht.

Beweis der Behauptungen:

zu 1) Da die Anzahl der Strecken endlich ist, und jede Strecke höchstens zweimal durchlaufen werden kann, bleibt Theseus nach endlich vielen Aktionen bei Ariadne oder Minotaurus stehen.

zu 2) Wenn Theseus bei Minotaurus steht, trifft nach Hilfsbehauptung I Fall b) zu, also ist Theseus mit Ariadne durch einen kreisfreien Weg von 1-Strecken verbunden.

zu 3) Wenn Theseus bei Ariadne anhält, dann hat dieser Knoten nach Verfahren weder K- noch O-Eigenschaft. Alle von Ariadne ausgehenden Strecken sind 2-Strecken. Der Faden ist vollständig aufgewickelt, d.h. es existieren keine 1-Strecken.
Angenommen, Minotaurus wäre dennoch erreichbar, dann müßte ein Weg von Ariadne zu Minotaurus existieren, und alle bei Minotaurus endenden Strecken wären 0-Strecken, da Theseus den Minotaurus nicht erreicht hat. Sei die Strecke vom Knoten k zum Knoten k' die erste 0-Strecke auf diesem Weg von Ariadne zu Minotaurus. Wenn Theseus k erreicht, findet er dort K Eigenschaft vor, sonst würde die Strecke von k nach k' durch die Aktion Ab zur 1-Strecke. Zu k führen also zwei 1-Strecken. Da bei Erreichen von Ariadne keine 1-Strecken mehr existieren, muß Theseus beide 1-Strecken nochmals durchlaufen, d.h. Theseus kehrt wieder zu k zurück. Da dann wieder K-Eigenschaft vorliegen muß, käme Theseus immer wieder zu k zurück, könnte also Ariadne nicht erreichen.
Also kann eine derartiger Weg von Ariadne zu Minotaurus nicht existieren, d.h. Minotaurus ist im Labyrinth von Ariadne ausgehend nicht erreichbar.

Eigenschaften von Algorithmen

Anschaulich verstehen wir unter einem Algorithmus (Verfahren) eine Vorschrift, nach der gegebene Größen (Eingangswerte) durch ein endliches System von Regeln (Befehle) in eindeutig bestimmter Weise (deterministisch) in andere Größen (Ausgangswerte) umgeformt werden.

Wir fordern folgende *Eigenschaften*:

1. Finitheit	Die Vorschrift läßt sich durch eine endliche Beschreibung (Zeichenfolge) angeben.
2. Effektivität	Die Regeln sind effektiv ausführbar.
3. Termination	Die Ausführung erfolgt in endlich vielen Schritten.
4. Determiniertheit	Die Ausführung ist eindeutig.
5. Effizienz	Die Ausführung ist mit vertretbarem Aufwand verbunden.
6. Akzeptanz	Die Vorschrift ist verständlich und nutzerfreundlich.

Betrachtungsweisen:
a) *als Regelsystem* (Operationssystem)
b) *als algorithmischer Prozess* (Operationsfolge, Folge von Befehlen)
c) *als Verhaltensfunktion* (Abbildung zwischen Eingangs- und Ausgangswerten)

Um den Begriff des Algorithmus näher zu bestimmen, wählen wir eine endliche nichtleere Menge X von Grundzeichen (etwa a, b, c, 1, 2, ...) und bilden daraus durch Hintereinanderschreiben (Verketten) Zeichenfolgen (Wörter), wie etwa abc , aa1aa, ...
Die Gesamtheit aller aus Zeichen aus X bildbaren Zeichenfolgen bezeichnen wir durch X*. Das Verketten von Wörtern aus X* liefert wieder Wörter aus X* und kann als assoziative[1] Operation über X* aufgefaßt werden.
Die Menge X wird Alphabet genannt und muß so beschaffen sein, daß jedes Wort aus X* auf genau eine Weise in seine Bestandteile aus X zerlegt werden kann.
Die Menge X ={|, ||}, wo || durch Hintereinanderschreiben von zwei | Elementen entsteht, erfüllt diese Eigenschaft z.B. nicht, da das Wort ||| aus Elementen von X verschiedenartig gebildet werden kann. Eindeutige Zerlegbarkeit kann durch spezielle Trennzeichen erzwungen werden.
Als Regeln bezeichnen wir Wortpaare (p,q) aus Elementen von X*. Die Anwendung der Regel (p,q) auf das Wort v = rps liefert das Wort w = rqs , wo r das kürzeste wählbare

$$v=rps \xrightarrow[(p,q)]{} rqs=w;$$

Wort bezeichnet, d.h. das von links her erste Auftreten von p in v wird an dieser Stelle durch q substituiert. Sollte p in v nicht vorkommen, so ist die Regel (p, q) auf v nicht anwendbar.

1) *Bei vorgegebener endlicher Menge R von Regeln heißt ein Wort w genau dann aus v direkt ableitbar, wenn eine Regel aus R existiert, deren Anwendung aus v das Wort w erzeugt (in Zeichen:* $v \xrightarrow[R]{} w$*).*
2) *w heißt aus v genau dann ableitbar (in Zeichen:* $v \xrightarrow[R]{*} w$*), wenn eine Folge* $w_0, w_1, \ldots, w_n$ *mit* $v=w_0$ *und* $w=w_n$ *existiert und für alle* $0 \le i < n$ *gilt:* $w_i \xrightarrow[R]{} w_{i+1}$.

Die als *Ableitung* bezeichnete Wortfolge $w_0, w_1, ..., w_n$ wird durch v und R nicht eindeutig festgelegt, da in jedem Schritt verschiedene Regeln aus R anwendbar sein können. Deshalb bringen wir die Regeln aus R in eine Ordnung, indem wir Regelfolgen bilden.
Eine Folge ρ von Regeln aus R bezeichnen wir als Programm über X und wählen eine Teilmenge R' von R als abbrechende Regeln aus. Durch Anwendung eines Programms $\rho = r_1; r_2; ...; r_n$ auf ein Wort v entsteht eine Folge von Wörtern $w_1, w_2, ..., w_n$ wobei w_{i+1} aus w_i durch Anwendung derjenigen Regel r_k aus p entsteht, die unter den auf w_i anwendbaren Regeln aus ρ den kleinsten Index k hat, d.h. $k = Min(j|w_i \xrightarrow[r_j]{} w_{i+1})$.
Die Folge der w_i bricht ab (ρ *terminiert*), falls eine abbrechende Regel aus R' ange-

[1] Eine binäre Operation ∘ heißt assoziativ, wenn für alle p, q, r gilt:
$(p \circ q) \circ r = p \circ (q \circ r)$.

wendet wird oder keine Regel aus ρ anwendbar ist. Das Programm p angewendet auf v terminiert mit w (in Zeichen: $v \xrightarrow[p]{} w\downarrow$).

Beispiel: Gegeben sei das Programm:

$\rho = ((ab, c); (a, aa); (c, d); (b, e))$ mit der abbrechenden Regel $r_3 = (c, d)$.
$\quad\quad r_1 \quad\quad r_2 \quad\quad r_3 \quad\quad r_4$

Ableitungen:

1.) $\underline{ab}b \xrightarrow[r_1]{} \underline{c}b \xrightarrow[r_3]{} db$
Terminierung durch Anwendung der abbrechenden Regel (c, d)

2.) $b\underline{a}c \xrightarrow[r_2]{} b\underline{a}ac \xrightarrow[r_2]{} b\underline{a}aac \xrightarrow[r_2]{} \ldots$
Regel r_2 ist wiederholt anwendbar, Verfahren terminiert nicht.

3.) $\underline{b}deb \xrightarrow[r_4]{} ede\underline{b} \xrightarrow[r_4]{} edee$
Verfahren terminiert, da keine weitere Regel anwendbar ist.

An diesem Beispiel ist zu erkennen, daß ein Verfahren auf verschiedene Weise terminieren kann, es zeigt aber auch, daß die Einführung abbrechender Regeln weder notwendig noch hinreichend für eine Terminierung ist.

Berechenbarkeit: Das Programm ρ transformiert ein Wort v in das Wort w, falls die durch Anwendung von ρ auf v erzeugte Wortfolge mit w abbricht, d.h. die Anwendung terminiert. Mit anderen Worten, das Programm ρ erzeugt eine (allgemein partielle) Funktion φ_p über der Wortmenge X*. (Wortfunktion über X)
Umgekehrt heißt eine Wortfunktion φ über X berechenbar[1], wenn es ein Programm ρ über X gibt, so daß für alle Wörter v, w über X gilt: φ(v) = w genau dann, wenn die durch Anwendung des Programms ρ auf v erzeugte Wortfolge mit w abbricht, d.h., p angewendet auf v mit w terminiert ($v \xrightarrow[p]{} w\downarrow$).

Beispiel:

Wir wählen X={0,1} und betrachten die Funktion φ, mit

$$\varphi(v) = \begin{cases} 0 & \text{gerade} \\ \text{falls die Anzahl der Stellen in v, die mit 1 besetzt sind} & \text{ist.} \\ 1 & \text{ungerade} \end{cases}$$

d.h. für φ(01101001) = 0 , da das Element 1 viermal (gerade) in v enthalten ist.
Hingegen ist φ(110100011) = 1, da das Element 1 fünfmal (ungerade) in v vorkommt.

[1]Wurde so erstmals von A. Markow 1947 eingeführt.

Behauptung:
Die Funktion φ wird durch das Programm p=((10, 1); (01,1); (11, 0); (00, 0))
(ohne abbrechende Regeln) berechnet.

Beispielberechnung:
$01\underline{10}1001 \xrightarrow{(10,1)} 0111\underline{10}01 \xrightarrow{(10,1)} 011\underline{10}1 \xrightarrow{(10,1)} \underline{01}111 \xrightarrow{(01,1)} \underline{11}11 \xrightarrow{(11,0)} \underline{01}1 \xrightarrow{(01,1)} \underline{11} \xrightarrow{(11,0)} 0$

Entscheidbarkeit, Aufzählbarkeit:

- entscheidbare Menge
  Eine Wortmenge M über X (mit den Elementen 0,1) heißt genau dann *entscheidbar*, wenn die charakteristische Funktion χ_M dieser Menge M berechenbar ist. Dabei gilt für die charakteristische Funktion χ_M der Menge:

$$\chi_M(\mathrm{w}) = \begin{cases} 0 & w \in M; \\ & gdw. \\ 1 & sonst, \end{cases}$$

  d.h., ein Wort w gehört genau dann zu M, falls $\chi_M(w) = 0$, sonst ist $\chi_M(w) = 1$. χ_M liefert also ein Entscheidungsverfahren für die Menge M.

- aufzählbare Menge
  Eine nichtleere Wortmenge M über X heißt genau dann *aufzählbar*, wenn eine berechenbare Funktion φ von der Menge der natürlichen Zahlen in die Wortmenge über X existiert, so daß der Wertebereich von φ die Menge M ist, d.h. $W(\varphi) = M$.

(Die Menge N der natürlichen Zahlen kann als Wortmenge über einem einelementigen Alphabet {I} aufgefaßt werden, wobei die natürlichen Zahl n durch das Wort I^{n+1} dargestellt ist, welches durch Hintereinanderschreiben von n+1 Zeichen I entsteht.)

Beispiel: Wir betrachten die Menge aller ungeraden natürlichen Zahlen als Wortmenge über {I}, d.h. M={| $^{2n+2}$: n ∈ N}.
Die Funktion $\varphi(I^{n+1}) = I^{2n+2}$ kann über dem Alphabet X = {0,I} durch das Programm p= ((0I, II0); (I0, II); (I, I0)) mit der abbrechenden Regel: (I0, II) berechnet werden.

a) φ(I) = II (nat. Zahl 1): $I \xrightarrow{(I,I0)} I0 \xrightarrow{(I0,II)} II$

b) φ(II) = IIII (nat. Zahl 3): $II \xrightarrow{(I,I0)} I0I \xrightarrow{(0I,II0)} III0 \xrightarrow{(I0,II)} IIII$

c) φ(III) = IIIII : $III \xrightarrow{(I,I0)} I0II \xrightarrow{(0I,II0)} III0I \xrightarrow{(0I,II0)} IIII0 \xrightarrow{(I0,II)} IIIII$

Die Funktion φ ist berechenbar und der Wertebereich von φ ist die Menge M.
Die Menge aller ungeraden natürlichen Zahlen ist demnach aufzählbar.

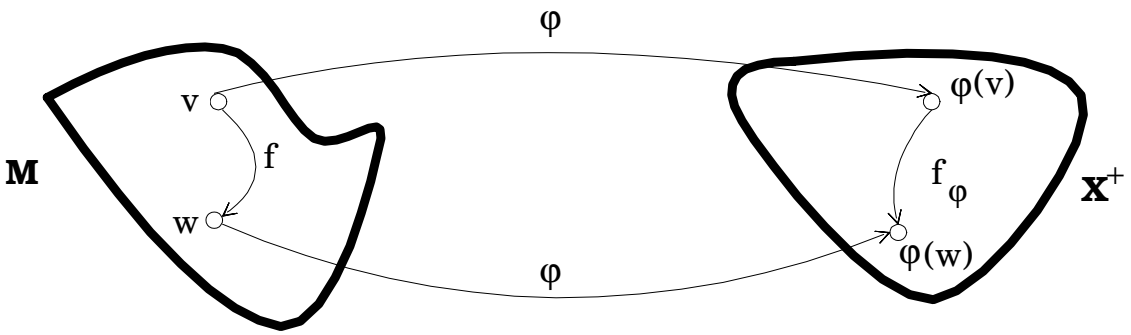
3. Digitale Informationsdarstellung

Codierung

Die zur Informationsdarstellung benutzten Objekte sind als Strukturen, im einfachsten Fall Wörter, über einer nichtleeren endlichen Grundmenge (Alphabet) X aufzufassen. (Auf der untersten Ebene der Rechner können nur 0,1-Folgen dargestellt werden.) Die durch die Verfahren zu manipulierenden Größen (Zahlen z.B.) müssen über dieser Grundmenge codiert werden.
Unter einer Codierung einer Menge M über dem Alphabet X verstehen wir eine injektive[1)] Abbildung φ von M in die Menge X^+ der nichtleeren Wörter über X. $\varphi(w)$ soll Codewort von w heißen.

Beispiel: Darstellung der natürlichen Zahlen durch |-Folgen.
N = Menge der natürlichen Zahlen
X= { | }
$\varphi(0)$= |; $\varphi(1)$= || ; ... ; $\varphi(n)= |^{n+1}$, wobei $\varphi(i)$ das Codewort von i ist.
Hier ist φ sogar eineindeutig, d.h., eine Bijektion.

Um mit den Codewörtern ähnlich arbeiten zu können, wie mit den Elementen (Urbildern) aus M, müssen etwa n-stellige Funktionen f über M ($f: M^n \longrightarrow M$) in geeignete n-stellige Funktionen f_φ über den Codewörtern über X ($f_\varphi: X^{+n} \longrightarrow X^+$) transformiert werden, so daß für alle $m_i \in M$ gilt: $\varphi(f(m_1, ..., m_n)) = f_\varphi(\varphi(m_1), ..., \varphi(m_n))$.
Das Codewort des Funktionswertes entspricht der Anwendung der zugeordneten Funktion auf die Codewörter der Argumente. (Mit anderen Worten, die Kodierung φ erzeugt einen Homomorphismus von der Struktur (M,f) in die Struktur (X^+,f_φ).)



Beispiel: Der Addition + für natürliche Zahlen entspricht bei der obigen Codierung der Funktion (+) über |-Folgen mit $|^m$ (+) $|^n = |^{m+n-1}$, so daß für beliebige natürliche Zahlen m, n wie gefordert gilt: $\varphi(m+n) = \varphi(m)$ (+) $\varphi(n)$.

[1)] Eine Abbildung φ heißt injektiv, wenn sie eindeutig ist und aus $\varphi(x) = \varphi(y)$ immer x=y folgt.

Zahlcodierungen (Stellenwertcodierung)

Sei M die Menge der natürlichen Zahlen und X die endliche Zeichen-Menge {0, 1, 2, ..., 8, 9, A, B, ...} bestehend aus b (Basis) Elementen. Die Zeichen aus X nennen wir Ziffern. In der Menge X erklären wir in natürlicher Weise (etwa wie angegeben) eine Ordnung und bezeichnen durch $\overline{x}$ die um eins verringerte Ordungszahl (natürliche Zahl) des Zeichens (Ziffer) x in dieser Ordnung, z.B. $\overline{A}$ entspricht der natürlichen Zahl 10. D.h. wir bezeichnen durch $\overline{x}$ den Zahlenwert des Zeichens x.

Codierung natürlicher Zahlen

Satz: *Zu jeder natürlichen Zahl z und jeder Basis b ≥ 2 gibt es genau eine natürliche Zahl n≥0 und eine Ziffernfolge $\varphi_b(z)=x_n\, x_{n-1} \ldots\ x_1\, x_0$, über X mit $x_n \neq 0$ für $n \neq 0$ und*

$$z = \overline{x}_n * b^n + \overline{x}_{n-1} * b^{n-1} + \ldots + \overline{x}_1 * b + \overline{x}_0 = \sum_{i=0}^{n} \overline{x}_i * b^i$$

($\varphi_b(z)$ heißt direkter Code von z zur Basis b)

Beweis:
Zum Beweis des Satzes nutzen wir folgende Konstruktion:

$$n = \text{Min}\,(i \mid i \in N \text{ und } z < b^{i+1})$$

(z.B. für b = 2, z = 5 ist n = 2, weil $5 < 2^{2+1}$ und $5 > 2^{1+1}$)
Für z = 0 wählen wir n = 0 und $x_0 = 0$.
Für $z \neq 0$ bilden wir durch Abspalten von Vielfachen von b^i:

$z = \overline{x}_n * b^n + r_n$ mit $0 < \overline{x}_n < b$ und $0 \le r_n < b^n$,
$r_n = \overline{x}_{n-1} * b^{n-1} + r_{n-1}$ mit $0 \le \overline{x}_{n-1} < b$ und $0 \le r_{n-1} < b^{n-1}$,
...
$r_2 = \overline{x}_1 * b + r_1$ mit $0 \le \overline{x}_1 < b$ und $0 \le r_1 < b$,
$r_1 = \overline{x}_0$.

Insgesamt erhält man $z = \sum_{i=0}^{n} \overline{x}_i * b^i$ und $\varphi_b(z) = x_n \ldots x_0$.

Unter Verwendung des *Horner-Schemas* kann die Folge $x_n \ldots x_0$ auch aus $z_i = \overline{x}_n b^{n-i} + \ldots + \overline{x}_i$ $(i \ge 0)$ durch $z_i = z_{i+1} b + \overline{x}_i$ für $0 \le i \le n$ und $z_{n+1} = 0$ erzeugt werden. Nach wiederholter ganzzahliger Division von z durch b entstehen die $\overline{x}_i$ als Reste in aufsteigender Folge $\overline{x}_0, \overline{x}_1, \ldots \overline{x}_n$.

Des weiteren ist die Eindeutigkeit dieser Darstellung zu zeigen. Diesen Beweis führen wir indirekt.

Annahme: Es existiert eine weitere Folge $x'_{n'}\, x'_{n'-1} \ldots\ x'_1\, x'_0$ mit $z = \sum_{i=0}^{n'} x_i' * b^i$.

Durch Differenzbildung entsteht $0 = \sum_{i=0}^{l} y_i b^i$ mit l = Max(n' , n) und

$$y_i = \begin{cases} \overline{x}_i - \overline{x}_i' & i \le \text{Min}(n',n) \\ \overline{x}_i & \text{falls} \quad n' < i \le n, \\ -\overline{x}_i' & n < i \le n'. \end{cases}$$

Wegen $|y_i| < b$ für alle i gilt:

$\left|y_1 b^1\right| = \left|\sum_{i=0}^{l-1} y_i b^i\right| \le (b-1) \sum_{i=0}^{l-1} b^i = b^l - 1$ und deshalb weiter $\left|y_l b^l\right| < b^l$.

Also ist $|y_1| < 1$ und damit $y_1 = 0$. Demnach gilt für alle i: $y_i = 0$ und deshalb n' = n und $\overline{x}'_i = \overline{x}_i$. Die Darstellung $\varphi_b(z) = x_n \ldots x_0$ ist demnach eindeutig. (Wenn keine Verwechslungen zu befürchten sind, werden wir $\overline{x}$ mit x identifizieren.)

3.2.2 Beispiel

Die Zahl Einhundertfünfundzwanzig (125) soll zur Basis b=2 (binär) codiert werden. Wir erhalten n=6, da $2^6 < 125 < 2^7$ ist. Durch Abspalten von Vielfachen von 2^i entsteht:

125= **1** * 2^6 + 61
61 = **1** * 2^5 + 29
29 = **1** * 2^4 + 13
13 = **1** * 2^3 + 5
5 = **1** * 2^2 + 1
1 = **0** * 2^1 + 1
1 = **1** * 2^0 + 0

Oder durch wiederholte ganzzahlige Division durch 2.
125 : 2 = 62 Rest **1**
62 : 2 = 31 Rest **0**
31 : 2 = 15 Rest **1**
15 : 2 = 7 Rest **1**
7 : 2 = 3 Rest **1**
3 : 2 = 1 Rest **1**
1 : 2 = 0 Rest **1**

Der *direkte Code* von 125 zur Basis 2 lautet $\varphi_2(125) = 1111101$ (Binärdarstellung).

Dieselbe Zahl 125 soll hexadezimal (zur Basis 16) dargestellt werden. Wir erhalten jetzt n = 1, da $16^1 < 125 < 16^2$. Durch Abspalten von Vielfachen von 16^i bzw. wiederholte Division durch 16 entsteht

125 = **7** * 16^1 + 13 bzw. 125 : 16 = 7 Rest **13**
13 = **13** * 16^0 + 0 7 : 16 = 0 Rest **7**.

Der *direkte Code* von 125 zur Basis 16 lautet $\varphi_{16}(125)$ = 7D (Hexadezimaldarstellung).

Codierung ganzer Zahlen

Satz: *Zu jeder ganzen Zahl $z \neq 0$ und jeder Basis $b \geq 2$ gibt es genau eine natürliche Zahl n und eine Ziffernfolge $\varphi_b(z) = x_{n+1}\, x_n \ldots x_1\, x_0$ über X mit $x_n \neq 0$, $x_{n+1} \in \{0,1\}$ und $z = \pm\sum_{i=0}^{n} \overline{x}_i b^i$ für $x_{n+1} = \begin{matrix}0\\1\end{matrix}$.*

($\varphi_b(z)$ heißt Vorzeichenbetragsdarstellung von z zur Basis b).

Beweis: Für |z| gibt es nach dem vorhergehenden Satz genau ein n und genau eine Ziffernfolge $\varphi_b(|z|) = x_n \ldots x_1\, x_0$ mit $|z| = \sum_{i=0}^{n} \overline{x}_i b^i$. Wir wählen $x_{n+1} = \begin{matrix}0\\1\end{matrix}$ *falls* $z \begin{matrix}>\\<\end{matrix} 0$. Dann ist die Ziffernfolge $\varphi_b(z) = x_{n+1}\, x_n \ldots x_0$ eindeutig und erfüllt die im Satz angegebenen Eigenschaften.

Beachte: Für die Zahl 0 kann n = 0 mit x_{n+1} = 0 oder 1 und x_n = 0 gewählt werden (Normalisierung).

Beispiel: Die ganze Zahl Minusfünfunddreißig (-35) kann in Vorzeichenbetragsdarstellung zu verschiedenen Basen wie folgt codiert werden.

b = 2, $\varphi_2(z)$ = 1100011, n = 5
b = 8, $\varphi_8(z)$ = 143, n = 1
b =10, $\varphi_{10}(z)$ = 135, n = 1
b =16, $\varphi_{16}(z)$ =123, n = 1

Negative Zahlen können auch durch Komplementdarstellung codiert werden.

Codierung Rationale Zahlen

Jede nichtnegative rationale Zahl z kann zunächst eindeutig in einen ganzzahligen Anteil $z_1 \geq 0$ und einen gebrochenen Anteil $0 \leq z_2 < 1$ zerlegt werden mit

$$| z | = z_1 + z_2.$$

Für z_1 können wir zunächst als natürliche Zahl eindeutig den direkten Code $\varphi_b(z_1) = x_n \ldots x_1 x_0$ bestimmen.
Für den gebrochenen Anteil z_2 existieren (nicht notwendig eindeutig) zwei natürliche Zahlen $x,y \neq 0$ mit $z_2 = x/y$ und $x < y$.
Die Zerlegung ist eindeutig, wenn x und y als teilerfremd vorausgesetzt werden. In diesem Fall bezeichnen wir durch $y = N[z_2]$ den Nenner von z_2.
Durch Abspalten der negativen Potenzen von b entsteht die folgende Zerlegungsfolge mit ganzzahligem $\overline{x}_{-1}$ und r_i:

$z_2 = x/y = \overline{x}_{-1}b^{-1} + r_1/yb$ d.h., $bx = \overline{x}_{-1}y + r_1$ mit $0 \leq \overline{x}_{-1} < b$, $0 \leq r_1 < y$

$r_1/yb = \overline{x}_{-2}b^{-2} + r_2/yb^2$ d.h. $br_1 = \overline{x}_{-2}\, y + r_2$ mit $0 \leq \overline{x}_{-2} < b$, $0 \leq r_2 < y$

...

$\frac{r_{m-1}}{yb^{m-1}} = \overline{x}_{-m}b^{-m} + r_m/yb^m$ d.h. $br_{m-1} = \overline{x}_{-m}y + r_m$ mit $0 \leq \overline{x}_{-m} < b$, $0 \leq r_m < y$

Insgesamt erhält man: $z_2 = x/y = \sum_{i=1}^{m} \overline{x}_{-i} b^{-i} + \frac{r_m}{y} b^{-m}$

($\overline{x}_{-i}$ bezeichnet den Zahlenwert der Ziffer x_{-i})
Eine Darstellung von z_2 wäre dann gefunden, wenn $r_m = 0$ wird. Darüber gilt der folgende Satz.

Satz: *Für jedes Paar teilerfremder natürlicher Zahlen $x,y \neq 0$ mit $x<y$ und jede ganze Zahl $b>1$ existiert ein $m>0$ so, daß $r_m=0$ gdw. y Teiler von b^m.*

Beweis:

a) ⟶ **Voraussetzung:** $r_m = 0$
Behauptung: y ist Teiler von b^m
Beweis:

Wegen $x = y\sum_{i=1}^{m} \overline{x}_{-i} b^{-i}$ gilt

$x \cdot b^m = y\sum_{i=1}^{m} \overline{x}_{-i} b^{m-i}$ und b^{m-i} ganz.

Also ist y Teiler von $x \cdot b^m$. Unter der Voraussetzung ggT(x,y)=1 ist dann y Teiler von b^m.

b) ⟵ **Voraussetzung:** y ist Teiler von b^m
Behauptung: $r_m=0$
Beweis:

Nach Voraussetzung ist $\frac{b^m}{y}$ und damit auch

$$\frac{r_m}{y} = \frac{x \cdot b^m}{y} - b^m \cdot \sum_{i=1}^{m} \overline{x}_{-i} b^{-i} = \frac{x \cdot b^m}{y} - \sum_{i=1}^{m} \overline{x}_{-i} b^{m-i}$$

eine ganze Zahl. Wegen der Bedingung $r_m < y$ und $y \neq 0$ muß $r_m = 0$ gelten.

q.e.d.

Satz *Zu jeder rationalen Zahl z≠0 und jeder Basis b≥2, wo jeder Primfaktor von N[z] auch Primfaktor von b ist, gibt es <u>genau ein</u> Paar natürlicher Zahlen m,n und eine Ziffernfolge* $\varphi_b(z) = x_{n+1}\, x_n \ldots x_0\, x_{-1} \ldots x_{-m}$, *mit* $x_{n+1} \in \{0,1\}$ *und* $x_n \neq 0$ *für m,n ≠0, so daß* $z = \pm \sum_{i=-m}^{n} \overline{x}_i \cdot b^i$ für $x_{n+1} = \begin{matrix} 0 \\ 1 \end{matrix}$.

Beispiel:

Gegeben: $z = \frac{21}{4} = 5 + \frac{1}{4}$

$N(z) = 4$

$b = 2$

Gesucht: $\varphi_2(z)$

Da N(z) und b nur den Primfaktor 2 besitzen, ist die Voraussetzung des Satzes erfüllt und wir bestimmen m=2, n=2 und $\varphi_b(z)$ = 010101. In der üblich gewohnten Kommadarstellung, wäre das Komma vor der zweiten Stelle von rechts zu lesen.

Beweis des Satzes:
(Wir behandeln nur den gebrochenen Anteil $z_2 = x/y$ von |z|.)
Seien $y = p_1^{k_1} \cdot p_2^{k_2} \cdot \ldots \cdot p_r^{k_r}$
und $b = p_1^{l_1} \cdot p_2^{l_2} \cdot \ldots \cdot p_s^{l_s}$ die Primfaktorzerlegungen mit den i-ten Primzahlen p_i.
Nach Voraussetzung muß für alle i; mit $k_i > 0$ auch $l_i > 0$ sein.
Für die m-te Potenz von b gilt dann:

$$b^m = p_1^{k_1} \cdot \ldots \cdot p_r^{k_r} \cdot p_1^{l_1 \cdot m - k_1} \cdot \ldots \cdot p_r^{l_r \cdot m - k_r} \cdot p_{r+1}^{l_{r+1} \cdot m} \cdot \ldots \cdot p_s^{l_s \cdot m},$$

d.h. y ist Teiler von b^m. Die oben angegebene Konstruktion der Ziffernfolge

liefert $r_m=0$, dh. $z_2 = \sum_{i=1}^{m} \overline{x}_{-i} b^{-i}$.

q.e.d.

Indirekter Beweis für die Eindeutigkeit der Darstellung von z_2:

Angenommen, es existiert eine Folge $x'_{-1}... x'_{-m}$, für die ebenfalls $z_2 = \sum_{i=1}^{m'} \overline{x}'_{-i} b^{-i}$ gilt. Die Differenz der beiden Summen ergibt :

$\sum_{i=1}^{l} y_{-i} b^{-i} = 0$, wobei l = Max (m' ,m) und $y = \overline{x}_{-i} - \overline{x}'_{-i}$ ganz.

Daraus entsteht

$\left| y_{-1} \cdot b^{-1} \right| = \left| \sum_{i=2}^{l} y_{-i} \cdot b^{-i} \right|$, mit $|y_{-i}| \leq b-1$ ganzzahlig.

Durch Abschätzung nach oben erhalten wir

$\left| y_{-1} \right| \leq (b - 1) \cdot \sum_{i=1}^{l-1} b^{-i}$ und weiter $\left| y_{-1} \right| \cdot b^{l} \leq (b - 1) \cdot \sum_{i=1}^{l-1} b^{i} = b^{l} - b.$

Also $|y_{-1}| < 1$, woraus $y_{-1} = 0$ und $\overline{x}_{-1} = \overline{x}'_{-1}$ folgt.

Dies gilt demnach auch für alle i > 1, womit die Eindeutigkeit bewiesen wäre.

q.e.d.

Konvertierung

In der Regel sind die Zahlen durch ihre Codes zur Basis 10 (Dezimaldarstellung) gegeben. Um daraus die Codierung zu anderen Basen zu bestimmen (Konvertierung), benutzen wir das Hornerschema.

Problem: Für die Zahl z sei der Code

$$\varphi'_{b'}(z) = x'_{n'} ... x'_1 x'_0, x'_{-1} ... x'_{-m'}$$

zur Basis b' gegeben. Gesucht ist die Codierung

$$\varphi_b(z) = x_n ... x_1 x_0, x_{-1} ... x_{-m}$$

von z zur Basis b.
(Die Konvertierung kann als Codierung der Wörter über der Menge von Ziffern x_i aufgefaßt werden.)

Lösung:

Wir betrachten die Zerlegung von z: $|z| = z_1 + z_2$, mit z_1 als natürliche Zahl und $0 \le z_2 < 1$;

z_1 und z_2 berechnen sich nach dem Hornerschema:

$$z_1 = \overline{x}_0 + b(\overline{x}_1 + \underbrace{b(\underbrace{\overline{x}_2 + \ldots + b \cdot \overline{x}_n \ldots)}_{S_2})}_{S_1} = s_0$$

$$\text{mit } s_k = \sum_{i=k}^{n} \overline{x}_i b^{i-k} \ge 0 \text{ und } 0 \le \overline{x}_k \le b-1,$$

$$z_2 = b^{-1}(\overline{x}_{-1} + \underbrace{b^{-1}(\overline{x}_{-2} + \underbrace{b^{-1}(\overline{x}_{-3} + \ldots + b^{-1}\overline{x}_{-m} \ldots)}_{t_2})}_{t_1}) = t_0$$

$$\text{mit } t_k = \sum_{i=k+1}^{m} \overline{x}_{-i} b^{k-i} < 1 \text{ und } 0 \le \overline{x}_{-k} \le b-1.$$

Damit ergibt sich mit $s_0 = z_1$ und $t_0 = z_2$ für die $\overline{x}_k$ und $\overline{x}_{-k}$ die Bildungsvorschriften:

$$\overline{x}_k = s_k - b \cdot s_{k+1} \quad \textit{für } 0 \le k \le n,$$
$$\overline{x}_{-k} = t_{k-1} \cdot b - t_k \quad \textit{für } 1 \le k \le m,$$

wobei $n = \text{Min}(k \mid s_{k+1} = 0)$ und $m = \text{Min}(k \mid t_k = 0)$ falls die Voraussetzung des letzten Satzes für z erfüllt ist. Bei der Durchführung des Verfahrens entsteht $\overline{x}_k$ als Rest der ganzzahligen Division s_k/b und $\overline{x}_{-k}$ als ganzer Anteil von $t_{k-1} * b$, mit:

$$\overline{x}_k = \text{Rest}(s_k/b); \quad \overline{x}_{-k} = \text{Ganz}(t_{k-1} * b).$$

Zu beachten ist, daß bei der Anwendung des Hornerschemas die Rechenoperationen im Zahlenbereich der neuen Basis ausgeführt werden müssen. Die Rechenverfahren und die Eigenschaften bleiben bei Basiswechsel erhalten und die Operationen selber sind zu den entsprechenden Basen auszuführen.

Beispiel:

Gegeben: Binärcode $\varphi_2(\mid z \mid) = 11001$ von z=25/8 mit m=3
Gesucht: Code $\varphi_8(|z|)$ zur Basis 8.
Bestimmung der $\overline{x}_k$:

$\overline{x}_0 = \text{Rest}(3/8) = 3$

$\overline{x}_{-1} = \text{Ganz}(0{,}125 * 8) = 1.$

Damit entsteht $\varphi_8(|z|) = 31$ mit m'=1.

Zahlendarstellung im Rechner

Um Zahlen im Computer darzustellen, wobei ein günstiges Verhältnis zwischen der Genauigkeit und dem benötigten Speicherplatz erreicht werden soll, gibt es verschiedene Möglichkeiten, auf die wir nachfolgend näher eingehen. Im konkreten Fall ist die Codierung und die Anzahl der Ziffern festzulegen.

Festkommadarstellung

Direkte Codierung

Bei vorgegebener Basis b und Stellenzahlen n, m mit der Codierung $\varphi_h(|z|) = x_n \ldots x_1\ x_0\ x_{-1} \ldots x_{-m}$ ist die Zahl $|z| = \sum_{i=-m}^{n} \bar{x}_i b^i$ ein Vielfaches von b^{-m} und der darstellbare Zahlenbereich ergibt sich zu $b^{-m} \le |z| \le b^{n+1} - b^{-m}$. Die Stellung des Kommas ist zwischen x_0 und x_{-1} fixiert.
Als Normierung gilt $x_n \neq 0$ für $n \neq 0$. Für Rechnerdarstellungen werden Codierungen zu Basen 2^k verwendet, da die dadurch entstehenden Codes immer mit endlich vielen Ziffern in die rechnerinterne Binärdarstellung umsetzbar sind. Durch diese Festlegung (Basis und Stellenzahl) wird die Menge der darstellbaren Zahlen weiter eingeschränkt. Es ist also mit Fehlern bereits bei der Zahleneingabe zu rechnen.

Darstellung negativer Zahlen:

a) *Vorzeichen-Betrags-Darstellung:*

Die Codierung von |z|, wie vorstehend angegeben, wird durch eine Vorzeichenstelle x_{n+1} ergänzt, wofür gilt $x_{n+1} = \begin{matrix} 0 \\ 1 \end{matrix}$ falls z $\begin{matrix} \ge \\ < \end{matrix}$ 0 unabhängig von den genannten Normierungen.

b) *(b - 1)-Komplement*

Hier wird ein von der Basis b und der Zahl z abhängiger Wert c zu z so addiert, so daß z' = z + c > 0 ist. Zur Codierung wird der direkte Code von z' verwendet.
Beim (b-1)- oder 9er Komplement ist

$$c = b^{n+2} - b^{-m} = \sum_{i=-m}^{n+1}(b-1)b^i ,$$

wobei n+1 die Anzahl der Stellen vor, und m die Anzahl der Stellen nach dem Komma angibt.
Das Vorzeichen von z ergibt sich aus der ersten Ziffer x_{n+1} von z' , die 0 für z>0 oder (b-1) für z<0 sein kann.
Beim Addieren im (b-1)-Komplement wird der an der (n+2)-ten Stelle entstehende Übertrag auf die (-m)-te Stelle addiert (Endübertrag).
Dies kann einfach durch ein Ringspeicher mit m+n+1 Stellen realisiert werden.

Beispiel:

Gegeben: $\varphi_{16}(z)$= 1AB1D,3; m = 1, n=3
in Vorzeichen-Betrags-Darstellung.
Gesucht: Darstellung im (b-1)-Komplement

Lösung:

z' = z+c

$c = b^{n+2} - b^{-m} = 16^5 - 16^{-1}$

Rechnung:

$$\begin{array}{rl} & FFFFF,F = \varphi_{16}(c) \\ - & 0AB1D,3 = \varphi_{16}(z) \\ \hline & F54E2,C = \varphi_{16}(z') \end{array}$$

c) b -Komplement

Eine andere Darstellungsform ist das b-Komplement, das sich von Fall b) nur durch den zu addierenden Wert c unterscheidet. Hier ist $c=b^{n+2}$. Beim Rechnen im b-Komplement ist keine Übertragsbehandlung notwendig. (Übertragsziffer entfällt).

Beispiel:

Gegeben: $\varphi_{16}(z)$= 1AB1D,3; m = 1, n=3
in Vorzeichen-Betrags-Darstellung.
Gesucht: Darstellung im b-Komplement

Lösung:

z' = z+c

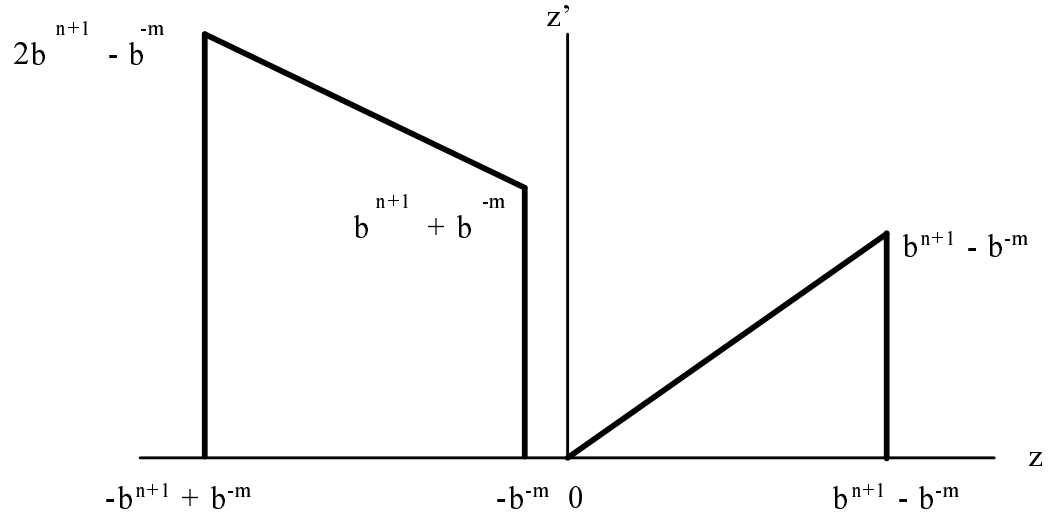
$c = b^{n+2} = 16^5$

Rechnung:

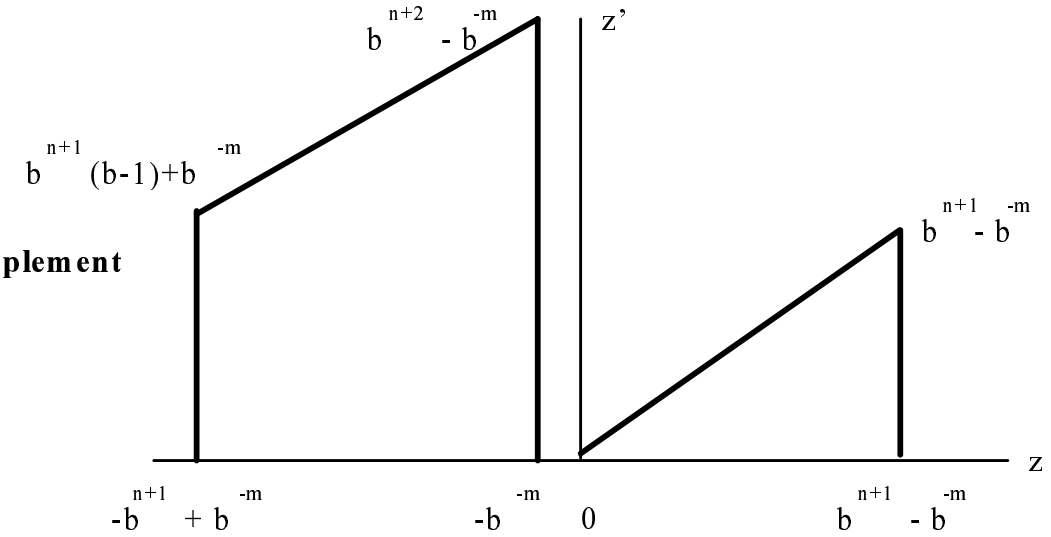
$$\begin{array}{rl} & 100000,0 = \varphi_{16}(c) \\ - & 0AB1D,3 = \varphi_{16}(z) \\ \hline & F54E2,D = \varphi_{16}(z') \end{array}$$

Die Verwendung von sehr großen und sehr kleinen Zahlen in Festkommadarstellung bringt einen hohen Speicheraufwand mit sich.
Deshalb empfiehlt sich eine Darstellung durch Mantisse und Exponent. Zuvor werden die Darstellungsbereiche von Festkommazahlen graphisch verdeutlicht.
Dabei bezeichnet z die tatsächliche Zahl und z' die zur Darstellung benutzte Zahl.

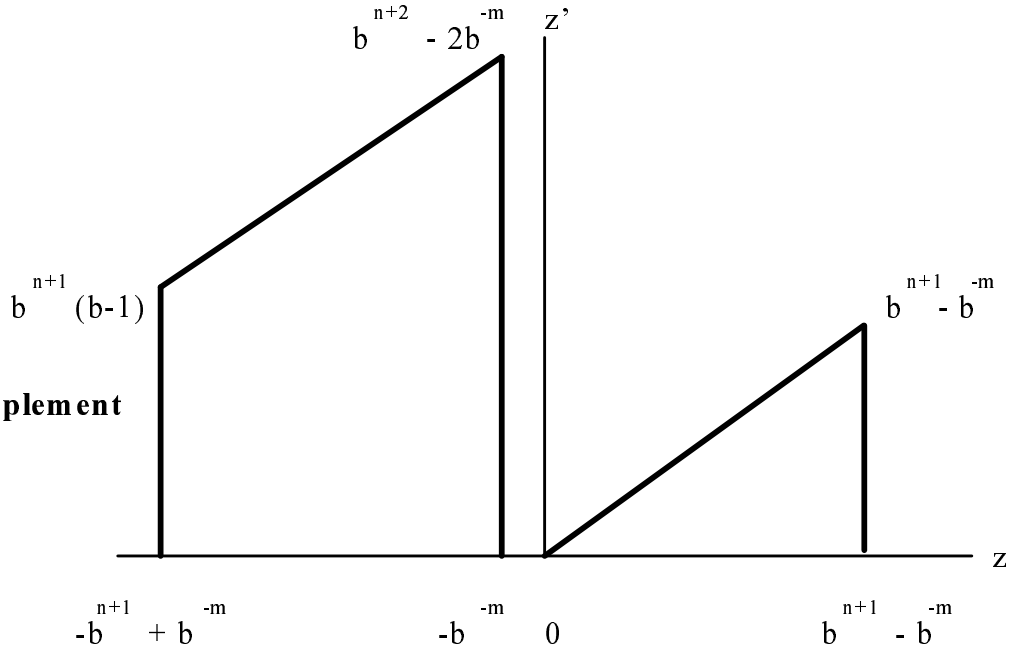
direkt



b-Komplement



(b-1)-Komplement



Gleitkommadarstellung

Die zu codierende Zahl z wird als Paar (z_0,e) aus Exponent e und Mantisse z_0 aufgefaßt und zur Basis b dargestellt als $z = \pm z_0 * b^e$, mit $|z_0| < 1$ und e ganze Zahl.
Mantisse z_0 und Exponent e werden in Festkommadarstellung codiert, zu:

$$\varphi_b(|z_0|) = x_{-1}\, x_{-2} \ldots x_{-m}.$$
$$\varphi_b(e) = x_n\, x_{n-1} \ldots x_0.$$

Als Normierung wird häufig e positiv und $x_{-1} \neq 0$ für $z \neq 0$ gewählt. Insgesamt hat die Gleitkommacodierung die Gestalt

VZ - Code	Mantissen - Code	Exponenten - Code

wo der Zahlexponent durch Addition von b^{n+1} in den positiven Codierungsexponenten $e' := e + b^{n+1}$ umgewandelt wird, d.h. $1 \leq e' < 2 * b^{n+1}$.
|z| wird demnach in Gleitkommadarstellung codiert durch:

$$\varphi'_b(|z|) = x_{-1} \ldots x_{-m}\, x'_{n+1}\, x'_n \ldots x'_0, \text{ wo}$$
$$\varphi'_b(z_0) = x_{-1} \ldots x_{-m} \text{ und } \varphi_b(e') = x'_{n+1} \ldots x'_0.$$

Beispiel 1:

$z = \frac{1}{16}$

b=10: $\varphi'_{10}(|z_0|) = 625$ und e=-1, n=1 ergibt e' = 100-1 = 99.
In *Gleitkommadarstellung* $\varphi'_{10}(|z|) = 625099$, mit m=3.

Beispiel 2:

b=2; $\varphi'_2(|z_0|) = 1$ und e=-3, n=2 ergibt e' = 8-3 =5;
In *Gleitkommadarstellung* $\varphi'_2(|z|) = 100101$ mit m=1.

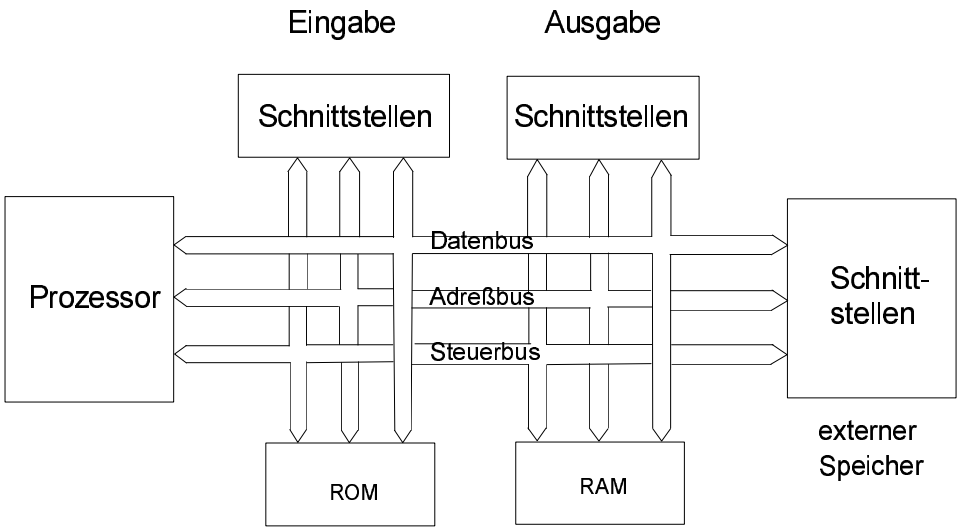
Beispiel 3:

b=8; $\varphi'_8(|z_0|) = 4$, e=-1, n=1 ergibt e' = 64 -1 = 63
In *Gleitkommadarstellung* $\varphi'_8(|z|) = 4077$ mit m=1.

Für die Zahlendarstellung in einem konkreten Rechner (Maschinenzahlen) ist die Länge von Mantisse und Exponent beschränkt. Überschreiten die Zahlen den im Rechner darstellbaren Zahlenbereich, treten Genauigkeitsverluste ein. Häufig werden Standardlängen vorausgesetzt.

4. Klassischer Digitalrechner

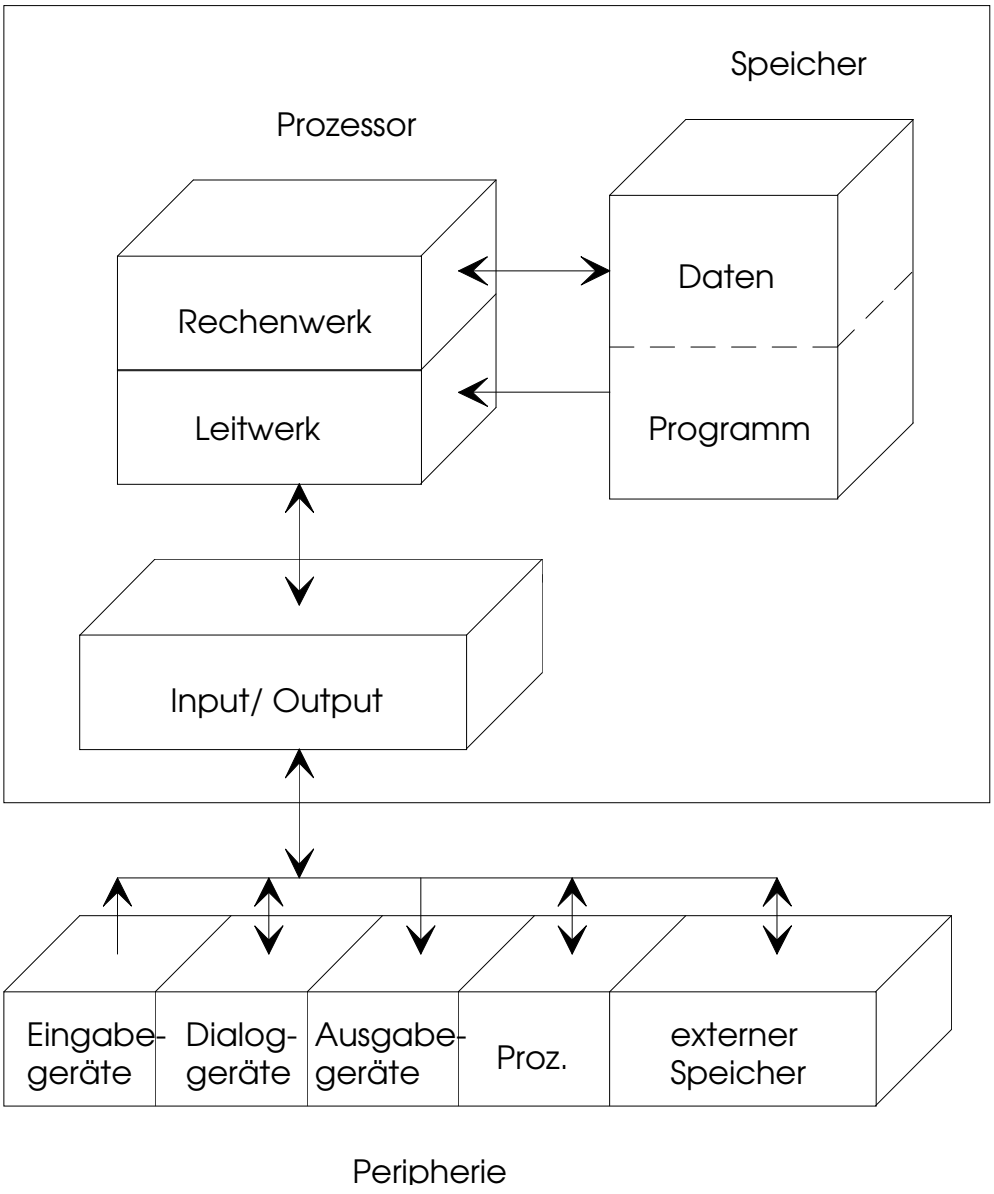
Unter einem Digitalrechner (Computer) verstehen wir ein universelles informationsverarbeitendes System, welches Folgen von Operationen (arithmetische, logische, u.a.) durch vorzugebende Programme gesteuert, selbsttätig (automatisch) auszuführen gestattet.
Die Universalität besteht darin, daß im Rahmen der vom Rechner ausführbaren Operationen beliebige Algorithmen (Verfahren) in entsprechende Programme transformiert (übersetzt) werden können.
Das Programm besteht aus einer Folge von Befehlen, die von einem oder mehreren Prozessoren ausgeführt werden, der über Leitungen (Daten-, Adreß-, Steuerbus) mit Arbeitsspeichern und Schnittstellen zur Außenwelt (Ein-, Ausgabemöglichkeiten, externe Speicher etc.) verbunden ist.



Allgemeiner Aufbau

von Neumann-Architektur

Im klassischen Rechnermodell, das der deutsch-ungarische Mathematiker von Neumann (1903-1957) vorgeschlagen hat, werden als Hauptbestandteile das Rechenwerk, das Leitwerk und der Speicher ausgezeichnet.



Leitwerk und Rechenwerk bilden den Prozessor. Die wichtigste Eigenschaft der von Neumann-Architektur besteht im gemeinsamen Speicher für Daten und Programme. Man spricht deshalb auch von speicher-programmierter Verarbeitung, bei der das Programm selbsttätig verändert werden kann. Den Verkehr mit der Außenwelt gewährleistet eine Input/Output-Einheit.

Speicher

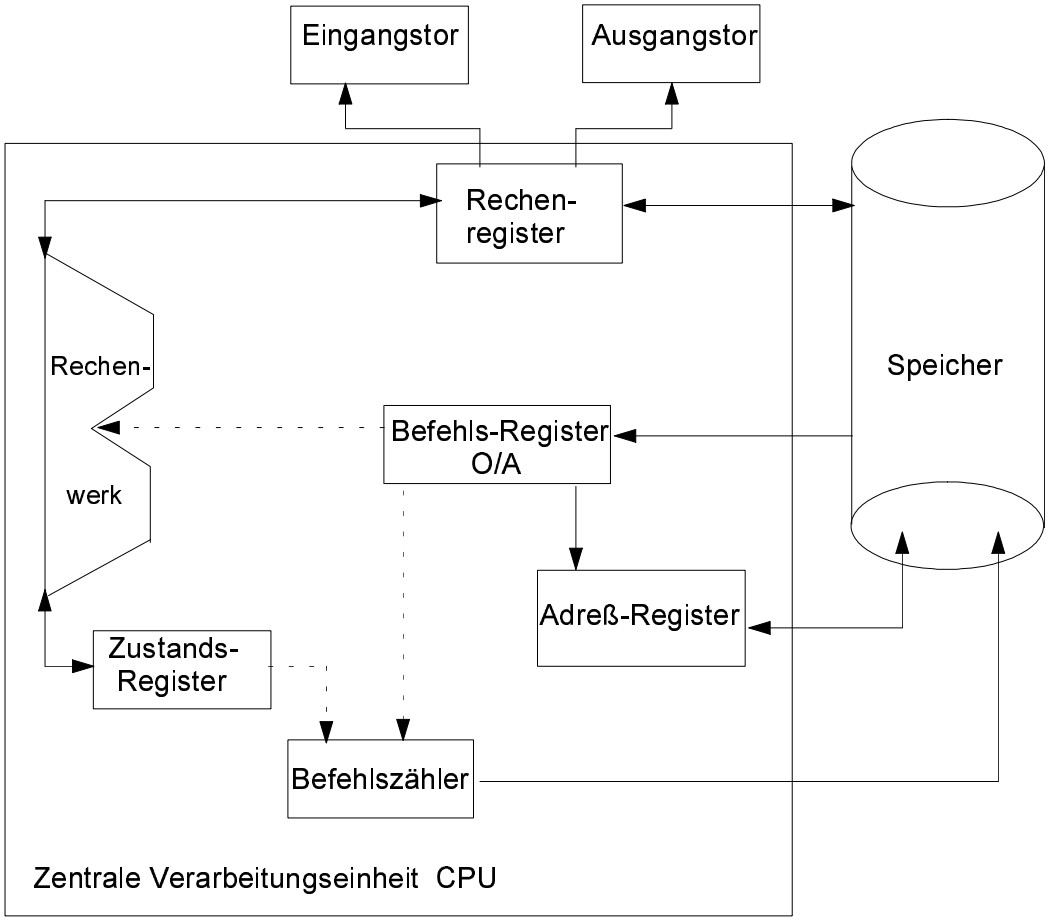
Der Speicher dient zur Aufbewahrung der benötigten Objekt- und Befehls-Informationen als Zeichenfolgen (Codewörter, fast immer linear) in einheitlicher Form. Der Speicher ist in einzelne Zellen unterteilt, die jeweils ein Wort aufnehmen, dessen Länge vielfach durch die Verarbeitungsbreite des Prozessors bestimmt wird (16 bzw. 32 Dualstellen [Bit]).
Den Zellen sind Adressen zugeordnet, über die auf die Wörter (Daten- und Befehle) zugegriffen werden kann
Die Unterscheidung in Daten- und Befehlswörter bezieht sich auf die Art der Verarbeitung dieser Wörter. Die Datenwörter codieren die Informationen über Objekte, die entsprechend dem Programm zu verarbeiten sind und die Befehlswörter codieren Informationen über die auszuführenden Operationen.
Zur Ausführung der Programme sind die Wörter aus dem Speicher zum Prozessor zu transportieren. Nach Ausführung der entsprechenden Operationen werden die Ergebniswörter im Speicher abgelegt. Dieser Speicher wird auch als Arbeitsspeicher bezeichnet. Größere Datenmengen werden auf externen Speichern gehalten.

Leitwerk (Steuerwerk)

Das Leitwerk hat folgende Aufgaben:

- Programmbefehle aus dem Speicher laden;
- Befehle decodieren;
- Befehle interpretieren;
- die an der Befehlsausführung beteiligten Funktionseinheiten mit nötigen Informationen bzw. Signalen zu versorgen.

Zur Bewältigung dieser Aufgaben besitzt das Leitwerk in der Regel eigene Speicher (Register). Solche Register sind der Befehlszähler, der die Adresse des nächsten auszuführenden Befehls (anfangs die Startadresse) enthält, das Befehlsregister, in dem sich der abzuarbeitende aktuelle Befehl befindet, das Rechenregister, welches das Ergebnis der ausgeführten Operation aufnimmt, und weitere Register, die die Befehlsausführung unterstützen . Die Befehlsabarbeitung erfolgt bei unserem Modell streng sequentiell in dem Zyklus Laden, Decodieren, Interpretieren und Ausführen. Jeder Befehl besteht aus einem Operationsteil und einem Operandenteil.



Der Operationsteil gibt an, welche Operation auszuführen ist, und der Operandenteil stellt die für die Operation benötigten Informationen über die Operanden zur Verfügung. Bestimmte Befehle (z.B. Halt) benötigen keine Operanden.

Rechenwerk

Im Rechenwerk wird die eigentliche Verarbeitung der Daten vorgenommen. Es werden hier sowohl arithmetische als auch boolesche (logische) Operationen ausgeführt.
Zur Ausführung dieser Operationen benötigt das Rechenwerk zusätzliche Arbeitsregister (Speicher). Die Aufgabe von Registern besteht darin, Informationen aufzunehmen, diese kurzfristig zu speichern und bei Bedarf wieder zur Verfügung zu stellen.
Um das Ergebnis einer Operation auswerten zu können, werden spezielle Eigenschaften im sogenannten Statusregister (Zustandsregister) gespeichert (z.B. ob das Ergebnis einer arithmetischen Operation negativ ist).
Neben den arithmetischen und booleschen Operationen können durch das Rechenwerk auch Transfer- und Schiebeoperationen mit Registerinhalten ausgeführt werden.

Ein- und Ausgabe

Die Input/Output-Einheit dient als Tor (Eingabe und Ausgabe) zur Außenwelt. Sie erfüllt in der Regel ebenfalls Speicher-, Steuer- und Verarbeitungsfunktionen, die hier nicht weiter präzisiert werden sollen.
Über diese Tore sind die peripheren Einheiten (z.B. zur Eingabe [Maus, Tastatur], Ausgabe [Drucker, Bildschirm], externe Speicher [Platten]) mit der Zentraleinheit verbunden.

Programmierung

Ein Programm besteht aus einzelnen Befehlen, deren Ausführung den gewünschten Berechnungsprozeß realisiert. Die Befehle selbst haben z.B. die folgende allgemeine Struktur:

Befehlsaufbau:

Operationsteil	Adreßteil
- legt die auszuführende Operation (durch Operationscode) fest	- legt den bzw. die Operanden (z.B. durch eine Adresse) fest

Man unterscheidet folgende Befehlsarten:

- *Transportbefehle, zum Austausch von Daten zwischen Speicher und Zentraleinheit bzw. zwischen Registern*
- *Datenmanipulationsbefehle, zur Ausführung von arithmetischen bzw. logischen Operationen*
- *Steuer- und Organisationsbefehle, zur Änderung von Systemzuständen bzw. des Abarbeitungsregimes*
- *Ein- und Ausgabebefehle, zum Informationsaustausch mit der Umwelt.*

Adreßinformationen werden u.a. benötigt zur:

- *Lokalisierung von Daten und Befehlen im Speicher,*
- *zum Spezifizieren von Operanden für die Ausführung von Operationen,*
- *zur Festlegung von Geräten und Registern.*

Man unterscheidet dabei verschiedene **Adressierungsarten:**

a) Implizite Adressierung

Bei dieser Adressierungsart ist der bzw. sind die Operand(en), z.B. durch Angabe eines oder mehrerer Register, im Operationsteil codiert.

Man unterscheidet:

- *direkte Registeradressierung:* Dabei beinhaltet das betreffende Register den Operanden.
  Bsp.: Lade den Inhalt des Registers A in das Register B.

- *indirekte Registeradressierung:* Der Inhalt des angesprochenen Registers gibt die Adresse des Operanden an
  Bsp.: Lade das Register A mit dem Inhalt der Adresse aus Register B.
  Beide Formen treten auch gemischt auf.

b) Konsekutive Adressierung

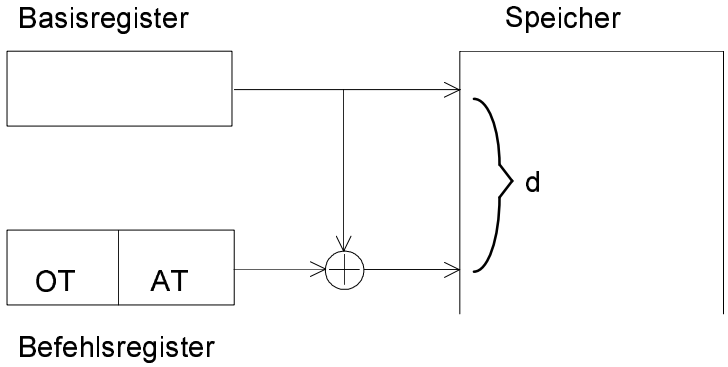
Das Merkmal dieser Adressierungsart besteht darin, daß sich die Informationen über die Operanden außerhalb des Operationscodes befinden, d.h. es wird ein expliziter Adreßteil angegeben. Man unterscheidet die:

- *unmittelbare Adressierung:* Der oder die Operanden selbst stehen im Adreßteil des Befehls. Man spricht in diesem Zusammenhang auch von Direktoperanden.
  Bsp. Addiere zu vorgegebenem Register eine im Befehl angegebene Konstante.

- *direkte Adressierung:* Die im Adreßteil angegebene Adresse bezeichnet die Speicheradresse des Operanden.
  Bsp.: Lade Register mit dem Inhalt der Speicherzelle 3!

- *relative Adressierung*: Die Adresse des Operanden wird berechnet aus dem Inhalt eines festen Registers (Basisregister), in der sich die Basisadresse befindet, zu dem die im Adreßteil des Befehls angegebene Adresse addiert wird.
  Das Basisregister adressiert einen Speicherabschnitt (Seite). Die Adresse im Adreßteil gibt die Distanz d zur Basisadresse an.

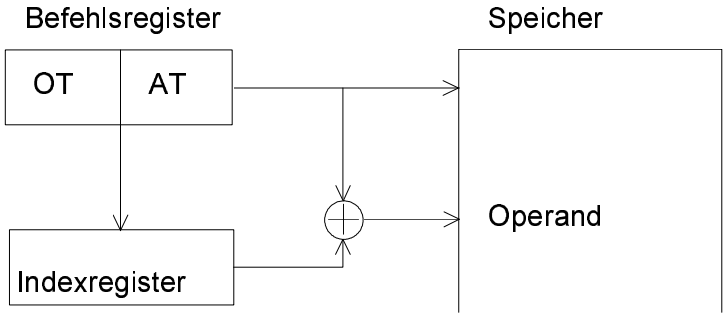
Operandenadresse = Basisadresse + Adresse im Adreßteil



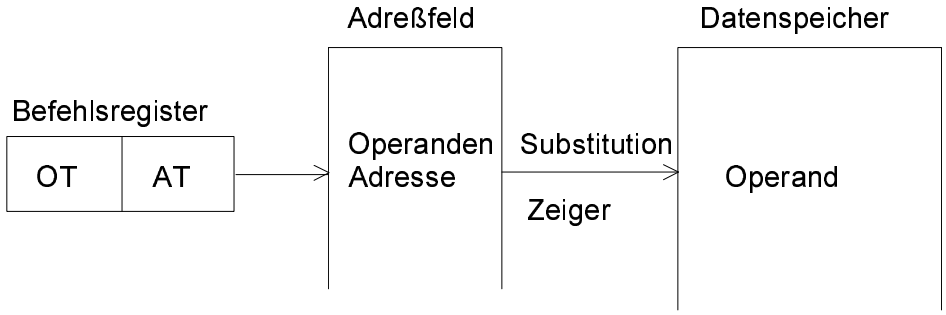
Die relative Adressierung ist Voraussetzung zur Verschiebung von Programmen im Arbeitsspeicher.

- *indexierte Adressierung:* Hierbei wird die Operandenadresse durch die Addition der im Adreßteil angegeben Adresse und dem Inhalt eines anzugebenden Registers (Indexregister) gebildet. Die Festlegung des Indexregisters kann im Operationsteil oder im Operandenteil erfolgen.

Operandenadresse = Adresse im Adreßteil + Inhalt des angegebenen Indexregisters



- *indirekte Adressierung:* Durch den Befehl wird die Adresse der Operandenadresse festgelegt. Bei der Befehlsausführung ist eine Adreßsubstitution vorzunehmen (dynamische Speicherverwaltung).
  Die Bestimmung der Anfangsadressen vom Adreßfeld und Datenspeicherabschnitt kann wieder durch vorgegebene Register realisiert werden. Datenumspeicherungen können durch Änderungen im Adreßfeld erreicht werden.



Modellrechner

Einadreßmaschine

Als Modell eines oben skizzierten Rechners betrachten wir zunächst ein Befehlssystem mit einer Adresse pro Befehl. Der Adreßteil AT des Befehls kann eine Adresse aufnehmen, die Operanden- oder Befehlsadresse sein kann.

Als Register verwenden wir:

RR	*Resultatsregister*
BR	*Befehlsregister*
AR	*Adreßregister*
BZ	*Befehlszähler*
ZR	*Zustandsregister*

Folgende Operationen stehen zur Verfügung:

° arithmetische Operationen

$\rightarrow \Rightarrow$ Transportoperation ohne, mit Löschen des Resultatregisters

H Steueroperationen

S unbedingter Sprung

$S \leq$, $S >$ bedingte Sprünge

Die Arbeitsweise des Rechners, d.h. die Ausführung der einzelnen Befehle, wird durch folgende Tabelle festgelegt.

Operationsteil des Befehls < OT >		Befehlsausführung <RR>, <BZ> gegeben
Operations-symbol	Bezeichnung	 := <<BZ>> ; <AR> := <AT> ; <BZ> := <BZ> + 1 (gilt für alle Befehle)
o	arithmetische Operation + - *	<RR> := <RR> o <<AR>> ; <ZR> := 1 falls <RR> 0 (0: >, 1: ≤)
$\rightarrow$	ohne Löschen Transport	<<AR>> := <RR> ; <ZR> := wie bei o
$\Rightarrow$	mit Löschen	<<AR>> := <RR> ; <RR>: = 0 ; <ZR> := 1
S	unbedingter Sprung	<BZ> := <AR>
$S_{\leq}$	kleiner/gleich	<BZ> := <AR> falls <ZR> := 1
$S_{>}$	größer	<BZ> := <AR> falls <ZR> := 0
H	Halt	Ausführung beenden
	keine Operation	----------

Das Resultatsregister RR nimmt die Zwischenergebnisse von arithmetischen Operationen auf. Im Befehlszähler BZ befindet sich die Adresse des auszuführenden Befehls. Dieser wird aus der Speicherzelle, die durch den Befehlszähler adressiert ist in das Befehlsregister BR geladen. Das Adreßregister AR nimmt die im Adreßteil AT dieses Befehls befindliche Adresse auf.
Danach wird der BZ um 1 (bzw. Anzahl der Zellen, die ein Befehl einnimmt) erhöht. Damit wird der Befehl mit der nächsten Adresse (fortlaufende Nummerierung vorausgesetzt), als der nach Ausführung des aktuellen Befehls im BZ vorbereitet.

Dabei verstehen wir unter <R> den Inhalt des Registers R. Mit <<R>> ist somit der Inhalt der Speicherzelle gemeint, die duch den Inhalt des Registers adressiert wird.

Der Inhalt des Operationsteils OT des Befehls in BR bestimmt die auszuführende Operation. Bei den arithmetischen Operationen (z.B. +, -, *) wird der Inhalt des Resultatsregisters RR als erster Operand mit dem Inhalt der durch das AR adressierten Speicherzelle als zweiten Operand verknüpft und das Ergebnis der Operation im RR abgelegt. Ist das Ergebnis gößer Null, so wird das ZR auf 0 ansonsten auf 1 (Status) gesetzt.
Um den Datentransport zwischen Resultatsregister und Speicher zu realisieren, stehen zwei Transport-Befehle (Transport ohne Löschen, Transport mit Löschen) zur Verfügung. Bei beiden wird der Inhalt des RR in die durch AR adressierte Speicherzelle gebracht. Beim Transport mit Löschen wird nach Ausführung des Transports der Inhalt des RR gelöscht (auf Null gesetzt). Beim Transport ohne Löschen bleibt der Inhalt des RR unverändert erhalten.
Durch die Sprungbefehle kann die Abarbeitungsfolge der Befehle entsprechend der Ordnung ihrer Adressen verändert werden. Die neue Adresse wird in den Befehlszähler geladen und überschreibt damit die vorbereitend um eins erhöhte aktuelle Befehlsadresse. Beim unbedingten Sprung wird dies immer ausgeführt, bei den bedingten Sprüngen, nur bei erfüllter Bedingung, d.h. entsprechend der Stellung des Zustandsregisters ZR. Ansonsten wird die gegebene Abarbeitungsfolge beibehalten.
Der Befehl HALT bewirkt ein Abbrechen der Abarbeitungsfolge, d.h. die Programmausführung wird beendet.
Der Operationsteil eines Befehls kann auch leer sein, d.h. nur der Befehlszählerinhalt wird wie oben beschrieben um eins erhöht.
Der Einfachheit halber wurde hier angenommen, daß wir es nur mit einfachen Objekten (z.B. Integergrößen) zu tun haben, die in einer Speicherzelle zu speichern sind.
Codierungsfragen sollen in unserem Modell unberücksichtigt bleiben. Wir arbeiten mit den vereinbarten Zeichen und Nummern als Pseudo-Code.

Beispielprogramm

Betrachten wir die Berechnung der Fakultät x! für natürliche Zahlen durch das Verfahren

$t_0 = x-1 , p_0 = x$

$p_{i+1} = p_i * t_i$ für $0 \leq i < x$ mit $x! = p_{(x-1)}$ bzw. $t_{(x-1)} = 0$

$t_{i+1} = t_i - 1$

Für die einzelnen Schritte ergibt sich :

i	t_i	p_i
0	x-1	x
1	x-2	x∗(x-1)
...	...	...
(x-2)	1	x∗(x-1)∗...∗2
(x-1)	0	x!

Das folgende Programm realisiert dieses Verfahrens auf einem Einadreßrechner mit der angegebenen Adressierung.

Die Zellen mit den Adressen 14,15 und 16 dienen als Datenspeicher. Auf Adresse 14 steht die Zahl minus Eins, die zur schrittweisen Dekrementierung von t benötigt wird. Die Speicheradresse 15 repräsentiert die Variable t und die Speicheradresse 16 die Variable p. Nach Beendigung des Programms steht das Ergebnis der Berechnung im Resultatsregister und auf der Adresse 16.
Der Befehl auf Adresse 0 belegt die Speicheradresse 16 mit dem Startwert (p_0=x). Ist der Eingabewert x kleiner oder gleich Null, so springt das Programm bedingt zur Adresse 9.

Adresse	Operationsteil	Adreßteil	<RR>	Bemerkung
0	→	16	x	p:=x
1	$S_{\leq}$	9		x ≤ 0?
2	+	14	t-1	
3	→	15		t:=t-1
4	$S_{\leq}$	12	t ≤ 0?	Abbruch des Verfahrens
5	∗	16	p∗t	
6	⇒	16	0	p:=p∗t
7	+	15	t	Lade t in RR
8	S	2		wiederhole Verfahren für i+1
9	⇒	15	0	
10	-	14	+1	x! = 1
11	⇒	16	0	
12	+	16	x!	Lade x! in RR
13	H	-		Ende der Programmabarb.
14	-1			
15	t			
16	p			

Die eigentliche Berechnung erfolgt auf den Adressen 2 bis 8. In Speicherzelle 2 wird eine Addition ausgeführt, die gleichbedeutend ist mit der Dekrementierung von t. Das Ergebnis dieser Subtraktion wird mit dem Befehl auf Adresse 3 in die Speicherzelle 15 transportiert. Ist dieses Ergebnis kleiner oder gleich Null so wird zur Adresse 13

gesprungen. Ansonsten wird eine arithmetische Operation ausgeführt, indem die Werte t und p multipliziert werden. Dabei hat das RR den aktuellen Wert von t und der Operand wird durch den Inhalt der Speicherzelle 16 dargestellt. Das Ergebnis dieser Multiplikation wird in die Speicherzelle 16 transportiert und gleichzeitig das Rechenregister gelöscht. In Adresse 7 wird durch einen Additionsbefehl das Rechenregister wieder mit dem Wert von t geladen. Anschließend erfolgt ein unbedingter Sprung zur Adresse 2, womit der gleiche Zyklus wiederholt wird und zwar bis die Abbruchbedingung in Adresse 4 erfüllt ist ($t \leq 0$).
Nach Abarbeitung des Befehls auf Adresse 13 bricht das Programm ab.

Dreiadreßmaschine

Der Adreßteil AT des Befehls beim Modell einer Dreiadreßmaschnie kann drei Adressen $\langle AT \rangle_i$ aufnehmen, die Operanden- oder Befehlsadressen sein können.
Das Resultatsregister RR, der Befehlszähler BZ und das Befehlsregister BR werden wie bisher verwendet. Um die drei Adressen aufnehmen zu können, verfügen wir nun über drei Adreßregister:

AR1	*Adreßregister 1*
AR2	*Adreßregister 2*
AR3	*Adreßregister 3*

Durch die folgende Tabelle wird die Arbeitsweise des Rechners aufgezeigt, wobei die Inhalte von Rechenregister und Befehlszähler wiederum vorgegeben sind:

Operationsteil des Befehls OT		Befehlsausführung <RR>, <BZ> gegeben
Operations-symbol	Bezeichnung	 := <<BZ>> ; <AR1> := $\langle AT \rangle_1$; <AR2> := $\langle AT \rangle_2$; <AR3> := $\langle AT \rangle_3$ <BZ> := <BZ> + 1 (für alle Befehle)
o	arithmetische Operationen + - *	<RR> := <<AR1> > o <<AR2>> ; <<AR3>> := <RR>
S	Sprung	<<AR3>> := <RR> ; <BZ> := <AR1> falls <RR> ≤ 0 <AR2> <RR> > 0
H	keine Operation	----------

Eine Befehlsabarbeitung erfolgt in ihren Befehlsphasen analog zur Einadreßmaschine.
Bei arithmetischen Operationen wird der Inhalt der durch AR1 adressierten Speicherzelle mit dem Inhalt der durch AR2 adressierten Speicherzelle arithmetisch

verknüpft. Das Ergebnis befindet sich anschließend im Resultatsregister und wird zusätzlich auf der durch das AR3 adressierten Speicherzelle abgelegt.
Um die Abarbeitungsfolge der Befehle zu verändern, wird der Sprungbefehl benutzt. Ist der Inhalt des Resultatregisters kleiner-gleich Null, so wird dem Befehlszähler der Inhalt des AR1 zugewiesen, ansonsten wird der Inhalt des AR2 als Adresse für den nächsten auszuführenden Befehl verwendet. Dabei wird jedesmal der Speicherzelle, die durch den Inhalt von AR3 adressiert wird, der Inhalt des Resultatregisters zugewiesen.
Ist der Operationsteil eines Befehls leer, so bewirkt dies lediglich eine Erhöhung des Befehlszählerinhaltes.
Die bei der Einadreßmaschine verwendeten Transportbefehle (<<AR>>:=<RR>,...) müssen nicht gesondert aufgeführt werden, da sie in den arithmetischen Operationen implizit enthalten sind.

Beispielprogramm

Zum Vergleich betrachten wir wie in 4.4.1 die Berechnung der Fakultät x! als Beispielprogramm und dem analogen Verfahren:

$t_0=x$, $p_0=1$ mit
$p_{i+1} = p_i * t_i$
$t_{i+1} = t_i - 1$

i	t_i	p_i
0	x	1
1	x-1	x
...	...	...
(x-1)	1	x∗(x-1)∗...∗2
x	0	x!

Das Verfahren wird durch folgendes Dreiadreßprogramm realisiert:

Adresse	Operationsteil	Adreßteil	<RR> = x	
0	S	5 1 8	x	t:=x
1	*	7 7 9	+1	p:=1
2	*	9 8 9	p · t	p:= p · t
3	+	8 7 8	t-1	t:= t - 1
4	S	6 2 8		t > 0 ?
5	*	7 7 9	+1	
6	H	- - -		
7		-1	-1	
8			t	
9			p	

Die Zellen 7, 8 und 9 stellen den Datenspeicher dar, wobei auf der Speicherzelle 7 die Konstante -1, auf der Speicherzelle 8 der Wert von t und auf der Speicherzelle 9 der Wert von p gespeichert wird. Auf Speicherstelle 9 befindet sich somit auch das Endergebnis.
Mit dem Sprungbefehl auf Adresse 0 wird der Inhalt des RR (x, Startwert von t) in die Speicherzelle 8 transportiert. Ist die Zahl x kleiner gleich Null, so wird zum Befehl mit der Adresse 5 gesprungen, der auf die Speicherzelle 9 eine 1 als Endergebnis schreibt. Durch den anschließenden H-Befehl wird das Programm beendet.

Ist x dagegen größer als 0 wird mit der Abarbeitung des Programms auf Adresse 1 fortgesetzt. Hier wird durch Multiplikation des Inhaltes der Speicherzelle 7 (Konstante -1) mit sich selbst eine 1 erzeugt und auf Adresse 9 als Anfangswert für p abgespeichert.
Anschließend wird eine Schleife durchlaufen, in der durch den Multplikationsbefehl auf Adresse 2, p mit t multipliziert wird. Durch den nächsten Befehl auf Adresse 3 wird der Wert von t in Speicherzelle 8 dekrementiert. Diese Schleife wird solange durchlaufen bis t Null ist. Die Schleifensteuerung wird durch den Sprungbefehl auf Adresse 4 realisiert.
Das Ergebnis der Berechnung steht schließlich auf Adresse 9.

Unterprogramme

Jede Aufgabe, die mit unserem Rechnermodell bearbeitet werden soll, muß in eine Folge von Befehlen (Programm) transformiert werden. Deren Abarbeitung entsprechend der obigen Beschreibung löst die vorliegende Aufgabenstellung, d.h., führt eine geeignete Speichertransformation durch.
Häufig verwendete Teilaufgaben, wie z.B. Berechnungen von ein- oder mehrstelligen Funktionen (sin x, e^x...), können als Programme vorgefertigt und als Unterprogramme in andere Programme integriert werden. Diese Methode kann als ein erster Schritt zur Automatisierung der Programmerzeugung angesehen werden.

Die Nutzung vorgefertigter Programme kann erfolgen als:

- ***Offene Unterprogramme***
  Bei der Benutzung als offenes Unterprogramm, werden die Programme an entsprechender Stelle im Hauptprogramm eingefügt.

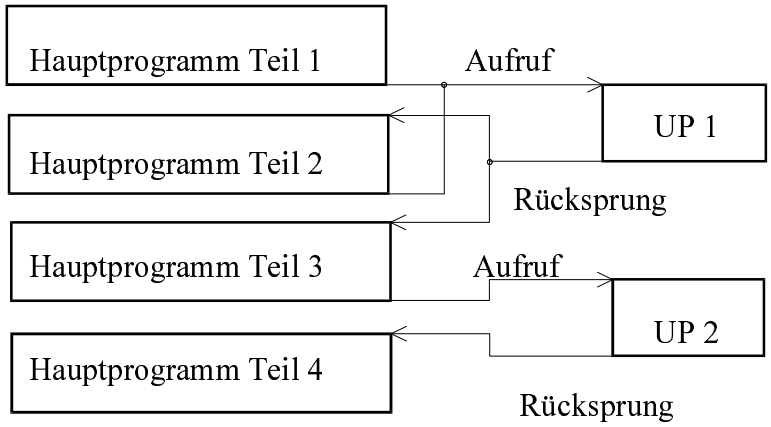
Hauptprogramm Teil1
Unterprogramm1...
Hauptprogramm Teil2
Unterprogramm2...
Hauptprogramm Teil3
Unterprogramm3...
Hauptprogramm Teil4

Häufig wird das gleiche Programm an mehreren Stellen als Unterprogramm benötigt.

Dann bietet sich die Nutzung an als

- ***Geschlossene Unterprogramme***
  Hier befinden sich die Unterprogramme außerhalb des Hauptprogramms. Dabei existiert nur ein Exemplar des jeweiligen Unterprogramms, welches von verschiedenen Stellen im Hauptprogramm, aufgerufen wird. Nach Beendigung des Unterprogramms wird an die aufrufende Stelle des Hauptprogramms zurückgesprungen und die Abarbeitung des Haupt-Programms beim nächsten Befehl fortgesetzt.

Um den Rücksprung zum Hauptprogramm zu realisieren, muß die Adresse, an der das Hauptprogramm fortgesetzt werden soll, vor dem Sprung ins Unterprogramm gespeichert werden.
Das Unterprogramm benötigt Informationen vom Hauptprogramm und umgekehrt muß das Unterprogramm seine Ergebnisse an das Hauptprogramm zurückliefern. Dies wird durch eine Parametervermittlung gewährleistet.

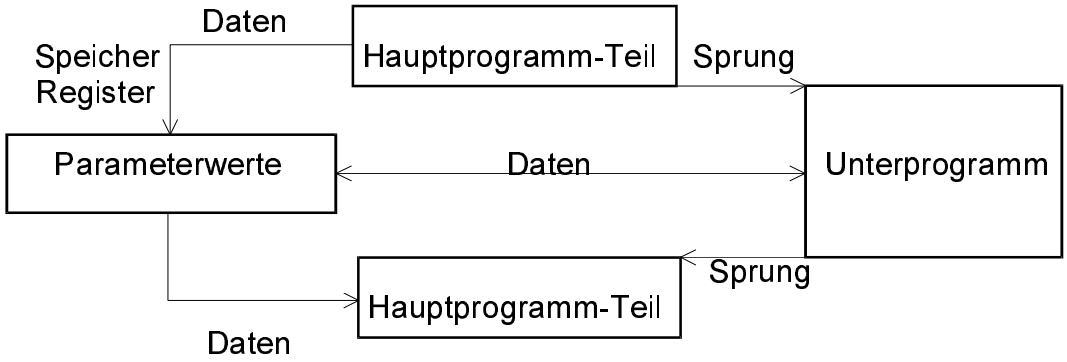


Parametervermittlung

Ein Unterprogramm tauscht mit dem aufrufenden Hauptprogramm Informationen (Parameter) aus. Hier sind etwa Argumente für Berechnungen oder deren Adressen dem Unterprogramm mitzuteilen und Ergebnisse aus dem Unterprogramm an das Hauptprogramm zurückzuliefern.
Es existieren verschiedene Möglichkeiten ein Unterprogramm mit Parametern zu versorgen oder Ergebnisparameter des Hauptprogramm zu vermitteln.

a) Parameter in festen Speicherzellen oder Registern

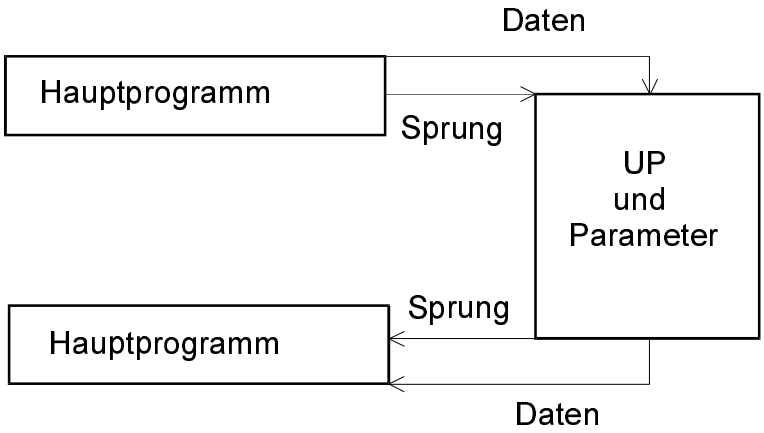
Die Werte der aktuellen Parameter werden vom Hauptprogramm in vordefinierte Speicherzellen bzw. Register geladen, von wo aus sie das Unterprogramm selbst übernimmt. Ergebnisse werden umgekehrt vom Unterprogramm zur Weiterverarbeitung durch das Hauptprogramm an bestimmte Speicherzellen bzw. Registern übergeben.



b) Parameter fest im Unterprogramm.

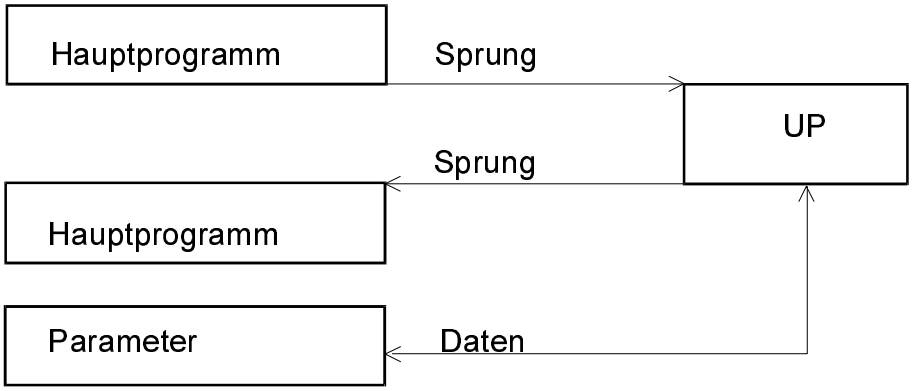
Da die Parameter fest im Unterprogramm integriert sind, müssen diese vor dem Sprung ins Unterprogramm duch das Hauptprogramm aktualisiert werden, d.h. sie werden im Unterprogramm gespeichert. Im Unterprogramm werden die aktuellen Parameter dann als Werte für die formalen Parameter verwendet. Die Ergebnisparameter werden in analoger Weise gespeichert. Anschließend werden die Ergebnisparameter durch das Hauptprogramm übernommen und dort weiter verarbeitet.

Auf die Parameter kann somit relativ zur Unterprogrammanfangsadresse zugegriffen werden.



c) Parameter fest im Hauptprogramm.

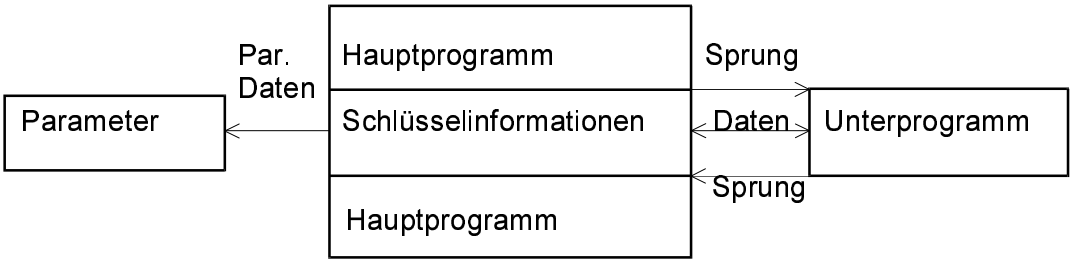
Die Parameter stehen im Hauptprogramm zur Verfügung und besitzen dort eine feste Adresse. Dabei werden die Parameter im Hauptprogramm zuerst aktualisiert. Nach Aufruf des Unterprogramms werden die aktuellen Parameter aus dem Hauptprogramm ins Unterprogramm überführt. Die aktuellen Parameter werden als Werte für die formalen Parameter verwendet. Vor dem Rücksprung ins Hauptprogramm werden die Ergebnisparameter an das Hauptprogramm übergeben.



d) Parameter als Schlüsselinformationen im Hauptprogramm

Schlüsselinformationen können sein:

- Parameterwerte (einzelne Parameter),
- Adressen von Parametern (einzelne Parameter, die mehrfach benutzt werden),
- Leitadresse von Parameterfeldern (z.B. mehere Parameter, Vektoren, Matrizen),
- Leitadresse von Programmen (z.B. von weiteren Unterprogrammen, die in Unterprogrammen verwendet werden).



Zur Übergabe von Parametern, Schlüsselinformationen bzw. Rücksprung-Adressen werden Adressensubstitutionen ausgeführt.

Die Übergabe von Werten erfolgt über Wertparameter und die von Adressen als Referenzparameter.

5. Daten- und Steuerstrukturen

Die in den Algorithmen verwendeten Objekte und die darüber ausgeführten Operationen bzw. Funktionen bilden als Einheit die sogenannten Datentypen.

Ein Datentyp bestimmt die Menge von Objekten, die Konstanten, Variablen oder Ausdrücke als Werte annehmen bzw. durch Funktionen berechnet werden. Der Typ eines durch Konstante, Variable oder Ausdruck bezeichneten Wertes muß vereinbart werden, falls dies nicht durch Standardvorgaben geregelt ist.
Operationen bzw. Funktionen erwarten Argumente eines bestimmten Typs und liefern Ergebnisse eines bestimmten Typs.

Bevor wir uns mit den Datentypen im allgemeinen oder deren Anwendung in konkreten Programmiersprachen beschäftigen, wollen wir uns dem Aufbau der Objektbereiche selbst zuwenden, d.h., die Struktur unserer (Daten-)Objekte untersuchen. Man spricht dann von **Datenstrukturen**, durch die der Aufbau von Objektbereichen mit Hilfe von Operationen, den sogenannten Konstruktoren, erfolgt.

Ausgangspunkt sind dabei die als elementar vorausgesetzten **unstrukturierten** (skalare) Objekttypen, wie Zahlen, Zeichen oder endliche Objektmengen, die nachfolgend beschrieben sind.

a) Typ enum (Aufzählung)

Eine Variable vom Aufzählungstyp bezeichnet eine Menge von Objekten mit einer festgelegten Ordnung.

enum t = (w_0, w_1, ... , w_n)
wobei (i-1) die Ordnungszahl von w_i und w_i vor w_{i+1}

Beispiel: enum Farbe = (rot, grün, gelb, blau, schwarz);

Folgende Operationen sind für den Aufzählungstyp häufig vordefiniert:
succ(w_i)= w_{i+1} (Nachfolger von w_i)
pred(w_i)= w_{i-1} (Vorgänger von w_i)

b) Typ boolean (Wahrheitswerte)

Eine Variable des Typs boolean kann nur die Werte 0 oder 1 (Wahrheitswerte) annehmen. Dabei entspricht 0 dem Wert false (falsch) und 1 dem Wert true (wahr).
Vordefiniert sind hier häufig die logischen (Booleschen) Operationen *oder* (Alternative), *und* (Konjunktion), *not* (Negation).

c) Typ byte word (nat. Zahlen)

Der Typ byte umfaßt einen Teilbereich der natürlichen Zahlen (0 -- 255), die durch 8 bit codiert werden können. Es sind alle Operationen möglich, die auch für natürlichen Zahlen erlaubt sind.

d) Typ integer (ganze Zahlen)

Der Typ legt jeweils Abschnittsmengen der Menge der ganzen Zahlen fest. Es gelten alle Operationen für ganze Zahlen (z.B. +, -, ∗, div, mod, succ, pred) und Vergleiche (z.B. < , =).
Als Codierungsbasis wird neben 10 häufig eine Zweierpotenz wie 8 (oktal) oder 16 (hexadezimal) verwendet. Diese Basis wird meist durch ein Zusatzzeichen am Anfang des Codewortes angezeigt.
Vorsicht: Es bleiben nicht alle Gesetze und Eigenschaften der ganzen Zahlen erhalten (Die Assoziativgesetze gelten allgemein nicht mehr).
Häufig wird mit dem Typ integer der Wertebereich der Zahlen -32768 bis +32767 verbunden. In Abhängigkeit von den zugelassenen Beschreibungsmitteln bzw. des verwendeten Rechnertyps sind auch unterschiedliche Wertbereiche gebräuchlich, die dann durch den Zusatz short bzw. long gekennzeichnet sind.
Zur Beschreibung von natürlichen Zahlen wurde vielfach zusätzlich ein Typ card (Cardinal) eingeführt.

e) Typ char (Zeichen)

Dieser Typ legt eine Codierung für alle Zeichen einer endlichen geordneten Zeichenmenge fest. Die Länge des Codes hängt von dem gewählten Codierungsalphabet und der Anzahl der zu unterscheidenden Zeichen ab, z.B. 1 Byte bei 256 Zeichen über einem binären Alphabet.
Meist werden Typtransferoperationen: char $\longleftrightarrow$ integer verwendet.
Z.B. ord (c) entspricht der Ordinalzahl von Zeichen c und
char (n) dem Zeichen mit dem Code n.

f) Typ string (Zeichenketten)

Der Typ umfaßt Folgen (Strings, Zeichenketten) von Objekten (Zeichen) vom Typ char bis zu einer bestimmten im Beschreibungsmittel festgelegten maximalen Länge. Anfang und Ende des Strings sind durch Sonderzeichen markiert. Die einzelnen Zeichen des Strings werden byteweise codiert.
Als Operationen über Strings werden u.a. häufig die Verkettung zweier Strings durch Hintereinanderschreiben, die Auswahl first des ersten Zeichens oder last des letzten Zeichens u.a. eingeführt.
Strings können auch als lineare Listen (strukturierter Datentyp) aufgefaßt werden, deren Elemente Zeichen sind.

g) Typ real (rationale Zahlen)

Der Typ umfaßt Teilbereiche der rationalen Zahlen.
Ein real-Objekt wird als Gleitkommazahl interpretiert. Es setzt sich aus drei Komponenten m (Mantisse), e (Exponent), b (Basis) zusammmen, womit die Zahl $n = m * b^e$ beschrieben wird. Häufig ist b als Zweierpotenz vorausgesetzt.
Der Umfang des Typs hängt vom Verwendungszweck und von der benutzten Rechnerbasis ab. Häufig werden neben den normalen real Objekten noch Objekte vom Typ double für das Rechnen mit
höherer Genauigkeit benutzt. Die definierten Operationen sind +, -, ∗, / mit zugehörigen Eigenschaften.

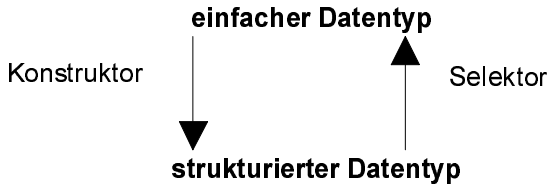
Vorsicht: Bei Division fast gleichgroßer Zahlen tritt eine Auslöschung und bei Division durch kleine Zahlen tritt ein Überlauf auf.
Transferoperationen: integer → real (unproblematisch), real → integer (Abschneiden von Stellen mit negativem Zahlexponenten).

Ausgehend von den skalaren Typen werden durch **Konstruktoren** zusammengesetzte **strukturierte** Typen erzeugt. Dabei unterscheiden wir zwischen **statischen** und **dynamischen** Strukturen.

Strukturierter Typ

Statischer Typ	Dynamischer Typ
(Feld, Verbund, Folge)	(Liste, Graph, Baum)

Ein Objekt vom strukturierter Typ ist durch die Art seiner Strukturierung und durch seinen Komponenten-Typ bzw. seine Komponenten-Typen gekennzeichnet und besteht aus einer Menge von Objekten niederer Struktur, im einfachsten Fall sind diese skalar.
Der Aufbau der Objekte erfolgt durch **Konstruktoren**, ihre Zerlegung in Bestandteile durch **Selektoren**.



Häufig werden folgende Strukturen verwendet:

Statischer Typ

a) Typ array (Feld)

Arrays haben eine bestimmte Anzahl von Komponenten eines vorzugebenden Typs (Komponenten- Typ).
Sei T' ein beliebiger Typ und i ein skalarer (einfacher) Typ, dann ist type T = array [i] of T ' eine Typdefinition .
Schreibweise: x = array [0..n] of *Komponententyp;*
d.h. es wird ein Feld mit n Elementen vom Typ *Komponententyp* definiert. Arrays können auch mehrdimensional sein,
y = array[0..a, 0..b, ... , 0..z] of *Komponententyp;*
Der Aufruf (Selektion) von Feldelementen erfolgt über deren Index durch x [i] bzw. y [i, j, ... , k] .

b) Typ record (Verbund, Struktur)

Ein record ist die Zusammenfassung einer festen Anzahl von Elementen unterschiedlichen Typs. Die Definition des record-Typs legt den Typ und den Namen (Selektor) jedes Elementes fest. Jeder Name darf in einer Typdefinition nur einmal als Selektor auftreten. Mehrere Selektoren für Elemente gleichen Typs können zusammengefaßt werden.

Beispieldefinition:

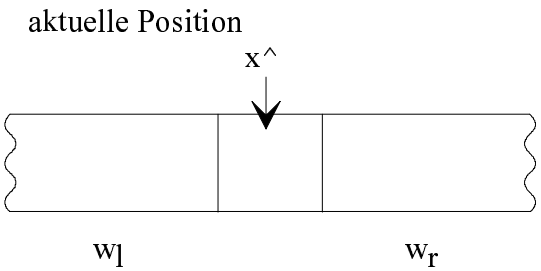
```
type Datumtyp = record
                    Jahr: integer;
                    Monat: string [ 10 ];
                    Tag: 1..31;
```

Die Auswahl (Selektion) eines Elementes einer vereinbarten Variablen Datum vom Datumtyp erfolgt durch Angabe von Variablenname und Selektor z.B. in der Form Datum .Tag .
Bei geschachtelten Records werden die einzelnen Selektoren jeweils durch . getrennt.

c) Typ file (Datei)

Objekte dieses Typs bestehen aus einer potentiell unendlichen Folge von Komponenten eines beliebigen Typs.
Schreibweise: type N = file of *Komponententyp;*
Files können als statische oder dynamische Strukturen betrachtet werden. Da Files wegen ihres Umfanges im allgemeinen auch auf "linearen" Hintergrundspeicher abgelegt werden, ist lediglich ein rein sequentieller Zugriff möglich. Über Files wird die Ein- und Ausgabe von Daten beschrieben.

Standardoperationen:
Alle Operationen auf Files basieren auf der Nutzung einer Puffervariablen x^ , die mit der Filevariablen x verbunden ist. Diese Puffervariable hat den Komponententyp und nimmt das Element an der aktuellen File-position an. Über die Puffervariable kann in das File geschrieben oder aus dem File gelesen werden. Das File wird durch die Puffervariable in zwei Teile (links und rechts von x^) gegliedert,



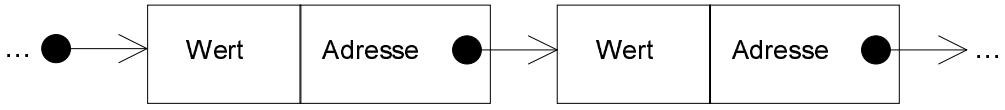
Für Files sind die folgenden Operationen üblich:

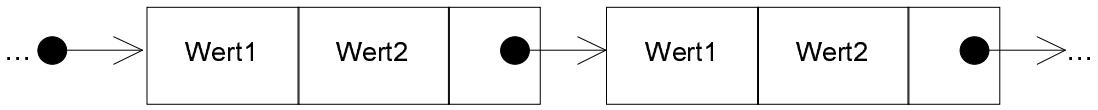
rewrite (x) ⟶ Löschen bzw. Erstellen eines leeren Files x
put (x, b) ⟶ Anfügen des Elements b an das File x
reset (x) ⟶ Setzen der Puffervariable an den Fileanfang
get (x) ⟶ Weiterrücken der Puffervariablen x^
eof (x) ⟶ Abfrage auf Fileende
open (x) ⟶ Öffnen des Files x
close (x) ⟶ Schließen des Files x

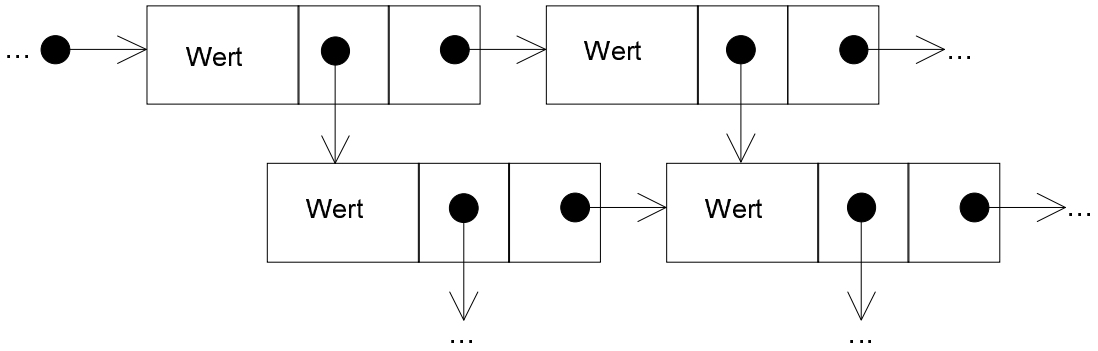
Die bisher betrachteten Strukturen (außer bestimmte Strukturen vom Typ file) sind statisch, d.h. die Anzahl der enthaltenen Elemente und damit der zu reservierende Speicherplatz ist von vornherein festgelegt. Es muß daher immer ein ausreichend großer Speicherraum zur Verfügung gestellt werden. Im konkreten Fall bleibt dann häufig ein großer Teil des Speichers ungenutzt. Um dies möglichst zu vermeiden und den Speicher effektiv zu nutzen, werden Datenstrukturen vom **dynamischen Typ** eingeführt.

Zeiger (Pointer)

Eine Variable vom Typ Zeiger kann als Werte Adressen bzw. Namen von anderen Strukturen eines bestimmten Typs annehmen. Im einfachsten Fall kann ein Zeiger Speicheradressen von Daten angeben. Er kann aber auch die Adresse einer Variablen, eines Records, einer Procedur oder einer Funktion sein. Mit Hilfe von Zeigern kann man Typen beliebiger Struktur und Größe aufbauen. Diese Objekte werden in einem offenen Speicher (Halde) gehalten, der vom Programmierer dynamisch verwaltet werden kann. Die Objekte auf der Halde haben keinen festen Speicherplatz und sind nur über die Zeiger erreichbar. Während die Ausgangspunkte der Zeiger fest sind, können ihre Zielpunkte umgestellt werden. Der Zeiger auf das Gesamtobjekt muß als Ankerpunkt im Speicher festgehalten werden.
Eine übliche Anwendung von Zeigern ist es, verkettete Listen von Records zu erzeugen. In diesem Sinne kann ein Zeiger als Referenz betrachtet werden, die z.B. auf ein Listenelement verweist. Jedes Listenelement enthält eine Referenz (Zeiger) zum nächsten Element der Liste.







Ein Bestandteil eines Listenelementes wird ausgewählt, indem man den Puffer Zeigervariable^ auf das Element setzt und **Element.Bestandteil** einer Variablen zuweist. Der Zeiger des letzten Listenelements sollte auf die leere Liste NIL (Listenende) gesetzt werden.

Als Operationen sind für den Typ Zeiger üblich:

New (Zeigervariable) -- richtet einen Zeiger ein und reserviert automatisch soviel Speicher, wie der jeweilige Datentyp beansprucht.

Dispose (Zeigervariable) -- gibt den Platz, den die Zeigervariable belegt hat, wieder frei, d.h. die Zeigervariable wird gelöscht.

Release (Zeigervariable) -- löscht die gesamte Struktur, die die Zeigervariable referiert.

Beispiele:

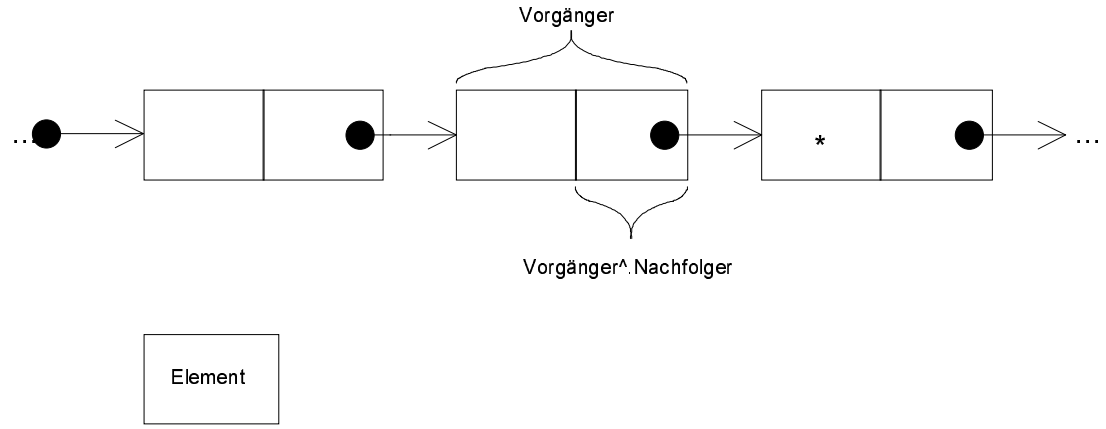
1. Erstellen einer neuen Liste:

Operationsfolge:

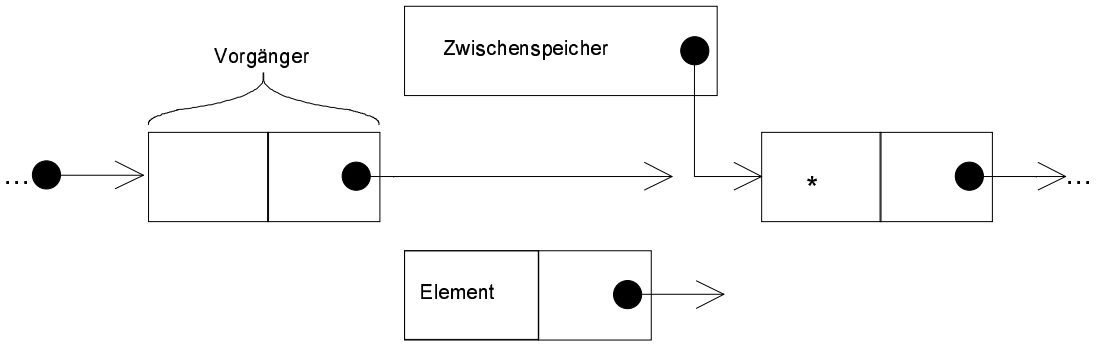
```
new(Unterliste);
Unterliste.Wert;
Unterliste^.Nachfolger:=NIL;
Erstellen:=Unterliste;
```

2. Einfügen eines Elements:

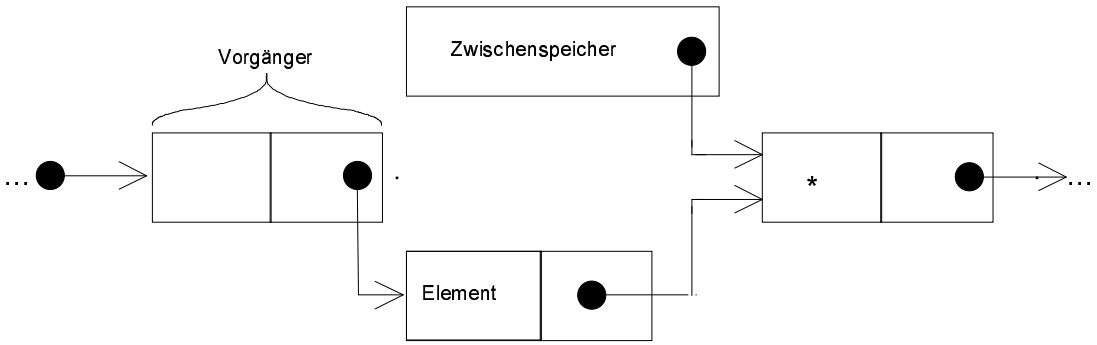
a)



b)



c)

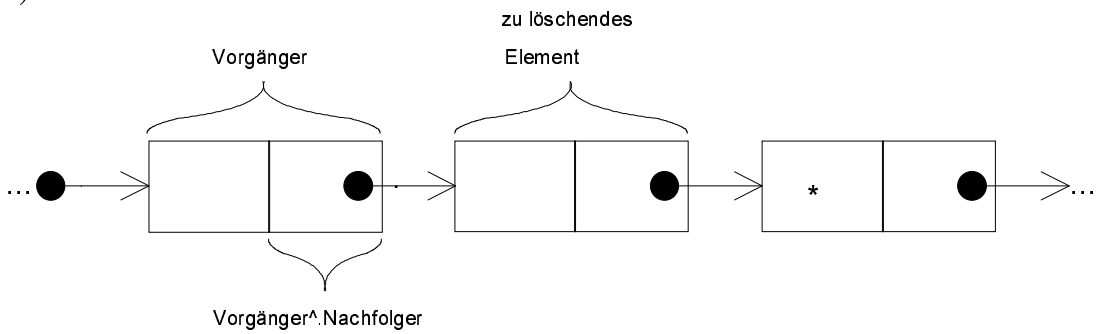


Operationsfolge: Zwischenspeicher:= Vorgänger^.Nachf;
new (Unterliste);
Vorgänger^.Nachf:=Unterliste;
Unterliste.Wert:=Element;

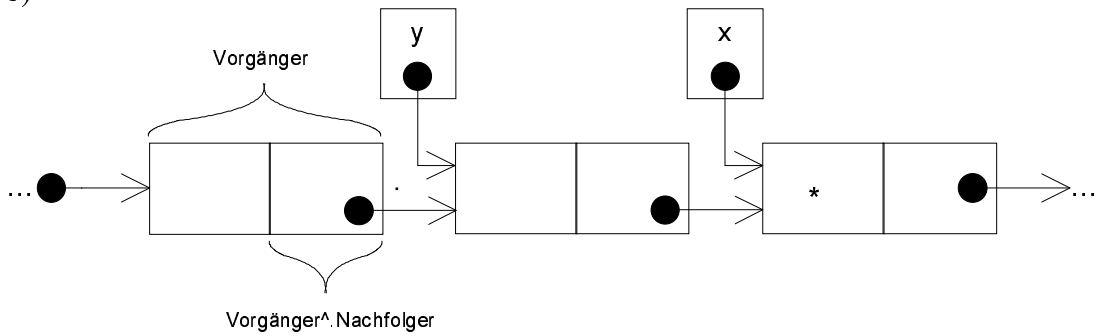
Unterliste^.Nachf:=Zwischenspeicher;

3. Löschen eines Elements in der Liste:

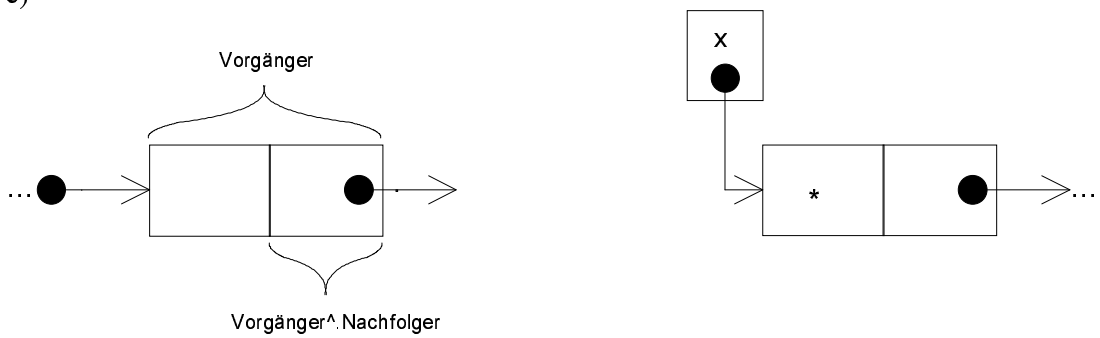
a)



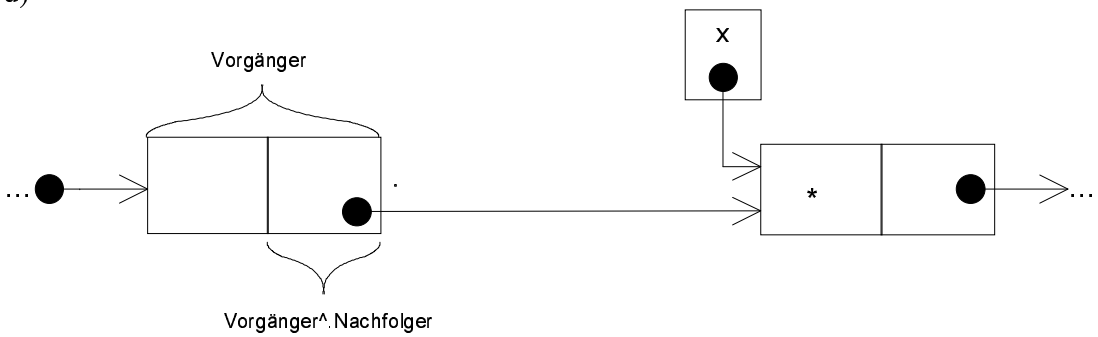
b)



c)



d)



Operationsfolge:

```
y:= Vorgänger^.Nachf;
x:= y^.Nachf;
dispose (y);
Vorgänger^.Nachf:=x;
```

Beachte: Da Listenelemente, auf die kein Pointer zeigt, nicht mehr erreicht werden können, ist es wichtig, Zwischenspeicher zu verwenden. Beim Löschen sollten nicht nur die Zeigervariable gelöscht, sondern auch die Speicherplätze wieder freigegeben werden (garbage collection).

Steuerstrukturen

Durch die Steuer- oder Kontrollstrukturen wird die Reihenfolge der auszuführenden Operationen über den spezifizierten Datenstrukturen in einem Programm festgelegt. Unter verschiedenen Programmierparadigmen sind diese Strukturen unterschiedlich ausgeprägt. Hier soll zunächst auf die Strukturen bei imperativer (anweisungsorientierter) Programmierung eingegangen werden. Auf die Steuerung der Systemkomponenten, abgesehen von einfachen Ein- und Ausgabe-Befehlen, soll später eingegangen werden. Bei der syntaktischen Beschreibung dieser Strukturen soll nicht auf eine konkrete Programmiersprache Bezug genommen werden. Im konkreten Fall ist die Syntax der jeweiligen Sprache zu beachten.

Syntaxdarstellung

Zunächst müssen wir die Beschreibungsmittel vereinbaren, die wir für die Notation der Strukturen verwenden wollen.

Es werden folgende Bezeichnungen vereinbart:

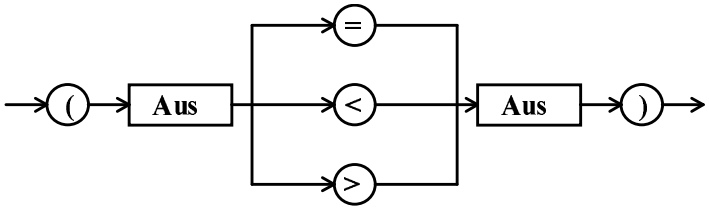
Grundbereiche:	C	Objektstukturen
	O	Operationen über Objektstrukturen
	F	Funktionen über Objektenstrukturen
	R	Relationen (Eigenschaften) über Objektstrukturen

Konstanten	$c \in C$ (z.B. für Gleit- und Festkommazahlen)
Wahrheitswertkonst.	$\underline{0}$ und $\underline{1}$ ($\underline{0}$ für false, $\underline{1}$ für true)
Namen (Variable)	$v \in V$ (Zeichen und Zeichenketten) Namen für Objekte
Operationszeichen	$o \in O$ (z. B. +, ∗, -, /, ..., ¬, ∧, ∨) ein- und zweistellig
Funktionszeichen	$f^s \in F$ (s>0 Stellenzahl oder Anzahl der Argumente von f)
Relationszeichen	$r \in R$ (z. B. =, <, >, ...) zweistellig
Trennzeichen	**(** , **)** , **[** , **]** , **{** , **}** , **:** , **:=** , **;** , **,**
Einfache Ausdrücke	EAus :: Konstanten c , Namen v
Ausdrücke (Terme)	Aus :: EAus , o Aus , (Aus o Aus) , f^m(Aus1, ..., Ausn)
Vergleiche	Vgl :: (Aus r Aus)
Bedingungen	Bed :: $\underline{0}$, $\underline{1}$, Vgl, ¬Bed, (Bed ∨ Bed), (Bed ∧ Bed)
Belegungen	Bel (partielle Funktion, die Namen Objektstrukturen als Speicherinhalte zuordnet)
Werte	<H> (Inhalt von H) Wert eines Ausdruckes
reservierte Namen	**begin** , **end** , **if** , **then** , **else** , **while** , **do** , **repeat** , **until** , **for** , **to** , **case** , **of** , **switch** , **goto** , **break** , **continue**

Den Namen werden als Variablen jeweils gewisse Objekte bzw. Objektstrukturen zugeordnet. Diese Zuordnung wird als Belegung bezeichnet und kann als (partielle) Funktion b von der Namensmenge in die Objektmenge aufgefaßt werden. Nach Interpretation Int der Grundbereiche mit konkreten Objekten, Operationen, Funktionen bzw. Relationen können den Ausdrücken, Vergleichen und Bedingungen bei vorgegebener Belegung b Werte zugeordnet werden.

Syntax	Wertbestimmung (b, Int gegeben)
EAus	$< c > = \text{Int } c$
	$< v > = b (v)$
Aus	$< o_1 H > = \text{Int } o_1 (< H >)$
	$< (H_1 o_2 H_2) > = \text{Int } o_2 (< H_1 >, < H_2 >)$
	$< f\text{^}s (H_1, ... , H_s) > = \text{Int } f\text{^}s (< H_1 >, ... , < H_s >)$
Vgl	$< (H_1 r H_2) > =$ true falls $(< H_1 > , < H_2 >) \in \text{Int } r$, sonst false
Bed	$< \underline{1} > =$ true , $< \underline{0} > =$ false , $< \neg H > = \text{not} < H >$
	$< (H_1 \vee H_2) > = < H_1 >$ oder $< H_2 >$, $< (H_1 \wedge H_2) > = < H_1 >$ und $< H_2 >$

Beispiel:
Der (vollständig geklammerte) Vergleichsausdruck ((x + y) ≤ ((1 - x) ∗ y)) nimmt unter der üblichen Interpretation der Operations- bzw. Relationszeichen bei der Belegung der Variablen x mit 1 und y mit 2 den Wert falsch an, bei der Belegung x mit 0 und y mit 1 aber den Wert wahr.
Häufig wird in der Literatur zur Beschreibung der Steuerstrukturen auch die Form der Syntaxdiagramme gewählt. Dabei beschreiben die in eckigen Kästen notierten Bezeichnungen andere syntaktische Begriffe (Metabezeichnungen), während in runden Kästen Grundzeichen bzw. reservierte Zeichenketten stehen. Die Zeichenfolgen, die dem jeweiligen syntaktischen Begriff entsprechen, der durch das Diagramm beschrieben wird, entstehen durch die möglichen Diagrammdurchläufe in Pfeilrichtung. Für den syntaktischen Begriff Vergleich Vgl entsteht auf diese Weise das folgende Diagramm.



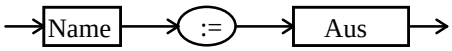
Damit werden als Steuerstrukturen die Anweisungen beschrieben. Diese Anweisungen, die auch geschachtelt auftreten können, erzeugen Belegungsänderungen, deren Wirkung in den Tabellen rechts charakterisiert ist.

Anweisungen Anw

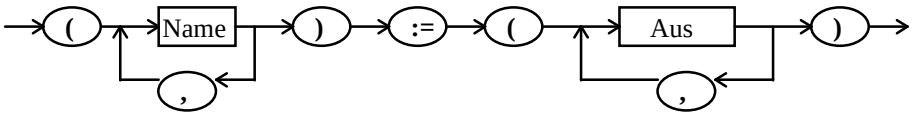
einfache Anweisung EAnw	v := Aus	Der Variablen *v* wird der Wert des Ausdruckes *Aus* zugeordnet.
simultane Anweisung SAnw	$(v_1, \ldots, v_n) := (Aus_1, \ldots, Aus_n)$	Für $1 \le i \le n$ wird der Variablen v_i der Wert des Ausdruckes Aus_i zugeordnet.
Anweisungsfolge Anwf	Anw Anw ; Anwf	sequentielle Ausführung von Anweisungen
Anweisungsblock Anwbl	**begin** Anwf **end**	Anweisungen der Anwf werden zusammengefaßt .
markierte Anw MAnw	C **:** Anw	Die Anw ist über C erreichbar (adressierbar) .

Syntaxdiagramme:

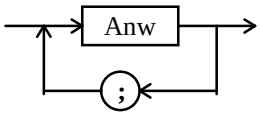
EAnw



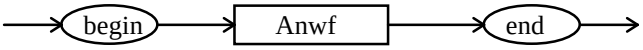
SAnw



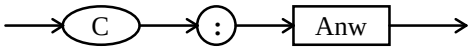
Anwf



Anwbl



Manw

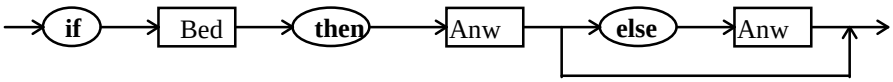


Bedingte Anweisungen BAnw

verkürzte BAnw	**if** Bed **then** Anw bzw. **if** (Aus) Anw	Wenn die Bed bzw. Aus ≠ 0 erfüllt ist, wird die *Anw* ausgeführt
vollständige BAnw (Auch als else-if-Ketten möglich.)	**if** Bed **then** Anw1 **else** Anw2 bzw. **if** (Aus) Anw1 **else** Anw2	Wenn die Bed bzw. Aus ≠ 0 erfüllt ist, wird die Anw1, sonst die Anw2 ausgeführt.

Syntaxdiagramm:

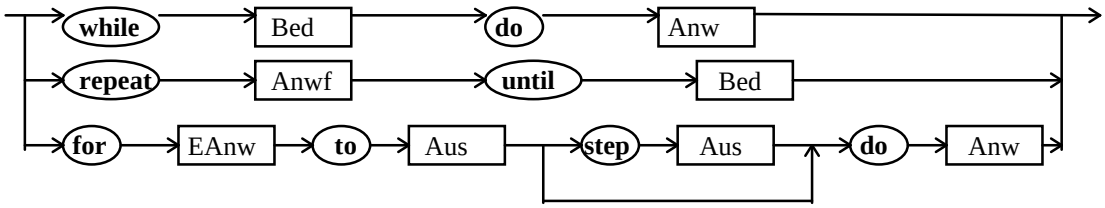
BAnw



Wiederholanweisungen WAnw

while-do-Schleife bzw. while-Schleife	**while** Bed **do** Anw bzw. **while (** Bed **)** Anw	Solange die *Bed* erfüllt ist, wird die *Anw* ausgeführt.
repeat-until-Schleife bzw. do-while-Schleife	**repeat** Anwf **until** Bed bzw. **do** Anwf **while (** Bed **)**	Die Anweisungfolge *Anwf* wird solange ausgeführt, bis die *Bed* erfüllt bzw. nicht mehr erfüllt ist.
for-to-Schleife	**for** EAnw **to** Aus1 **step** Aus2 **do** Anw bzw. **for (** EAnw ; Bed ; EAnw°**)** Anw	Die (Lauf)Variable in *EAnw* wird nach jedem Schleifen-durchlauf (bei der die Anwei-sung *Anw* ausgeführt wird) um eine durch den Wert von Aus2 festgelegte Schrittweite erhöht bzw. durch EAnw° geändert, bis der durch Aus1 festgelegte Endwert erreicht oder über-schritten bzw. die Bed verletzt ist.

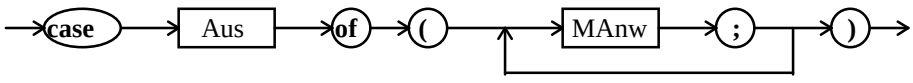
Syntaxdiagramm: WAnw



Auswahlanweisungen AAnw

case-of-Anweisung bzw. switch-Anweisung	**case** Aus **of (** c_1 **:** Anw_1; ... c_n **:** Anw_n **;)** bzw. **switch (** Aus **) { case** c_1 **:** Anw_1; ... c_n **:** Anw_n**; }**	Falls der Wert von *Aus* mit dem Wert von c_i überein-stimmt, wird die Anweisung Anw_i ausgeführt bzw. dort fortgesetzt (Verzweigung).

Syntaxdiagramm: AAnw



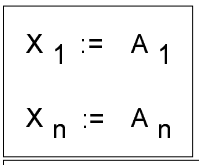
Unterbrechungs - Anweisungen (unstrukturiert)

Abbruch	**break**	Schleifenabbruch.
Fortsetzung	**continue**	Abbruch und Fortsetzung der umgebenden Schleife.
Sprung	**goto** C	Programm wird bei Marke C fortgesetzt(Sprunganweisung)

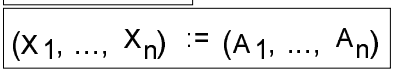
Unstrukturierte Notationen werden auch durch sogenannte Ablaufdiagramme beschrieben, auf die hier verzichtet werden soll.

Strukturierte Darstellung (Struktogramme)

Ergibtanweisungen:
Einfache Anweisung

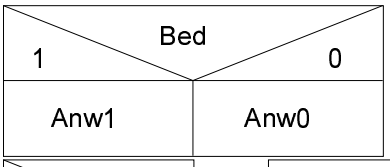


Simultane Anweisung

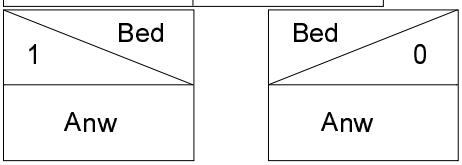


Bedingte Anweisungen:

if-then-else-Anweisung

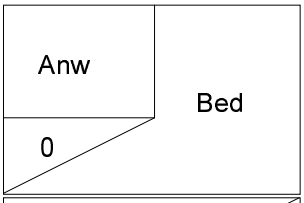


if-then-Anweisungen

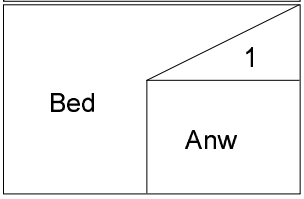


Wiederholanweisungen:

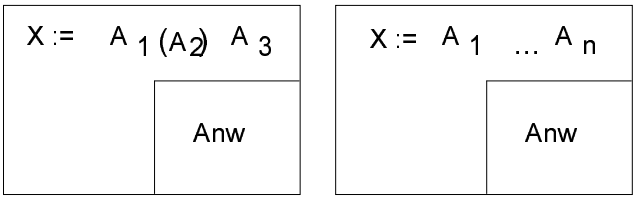
repeat-until-Schleife



while-do-Schleife

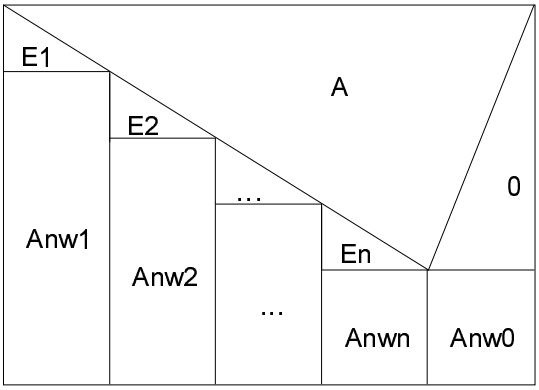


for-to-Schleife



***Auswahlanweisungen**:*

case-of-Anweisung



Kompositionsarten von Anweisungen

Entsprechend der Art der Ausführung von Anweisungen unterscheidet man verschiedene Kompositionen:

a) sequentiell (hintereinander)	p_1; p_2 oder $p_2(p_1)$	Die Anweisungen werden hintereinander ausgeführt.
b) selektiv (wahlweise)	if- then- Anweisung case- of- Anweisung switch-Anweisung	Die auszuführende Anweisung wird aus mehreren Anweisungen ausgewählt .
c) induktiv/ iterativ (wiederholt)	while- do- Anweisung repeat- until- Anweisung for- to- Schleife	Die Anzahl der Durchläufe des Programms hängt vom Wert einer Bedingung ab. Die Anzahl der Durchläufe ist vorher festgelegt.
d) verteilt (konkurrent, parallel)	[p_1 , p_2]	Die Reihenfolge der Ausführung von p_1 und p_2 ist beliebig. Daher können sie beispielsweise auf verschiedene Prozessoren verteilt werden.

6. Prozedurale Programmierung

Variable

Das Konzept der Variablen ist für den anweisungsorientierten imperativen Programmierstil und den entsprechenden Programmiersprachen fundamental. Durch Anweisungen werden Speicherbelegungen verändert, die selbst als Zuordnungen von Objektstrukturen zu Variablen aufzufassen sind. Die Variable ist dabei programmtechnisch als Objekt anzusehen, welches ein anderes Objekt als Wert zugeordnet erhält. Dieser Wert kann über den Namen der Variablen erreicht und manipuliert werden. Die Variable stellt dabei eine Referenz von Namen zu den Speicherzellen her, in denen dieser Wert als Datenstruktur abgelegt ist. Der Wert der Variablen entspricht dem Inhalt von Speicherzellen.
Variablen können in Programmen angelegt und benutzt werden, aber auch unabhängig davon u.a. als Namen für Dateien existieren. Variable in Programmen müssen erklärt werden, insbesondere ist der Typ der Objektstrukturen anzugeben, die sie als Werte annehmen oder über ihre Namen referieren sollen. Dies geschieht durch Deklarationen, die vor der Benutzung der darauf bezugnehmenden Variablen auszuführen sind. Sie stellen eine Zuordnung von Variablenname und Typ dar, durch die bei Ausführung Speicherplatz für als Werte der Variablen möglichen Objektstrukturen reserviert wird.
Syntaktisch wird dies durch die Deklaration **var** Namenliste **:** Typ zum Ausdruck gebracht, wobei die Variablennamen in der Liste durch Kommata getrennt sind.
Im einfachsten Fall handelt es sich um Variable für skalare Basistypen, wie z.B. bei den Variablen mit den Namen m und n in der Deklaration **var** m , n : integer, womit Speicherzellen referiert werden, die Objekte vom Typ integer aufnehmen können. Bei Ausführung der Deklaration in einem Programm werden diese Speicher reserviert, ohne das ihr Inhalt vorgegeben oder bekannt ist. Dieser Inhalt wird erst durch Anweisungen wie n := 1 definiert bzw. durch n := (n+1) gelesen und überschreibend verändert.
In die durch die Variable referierten Speicherbereich können nur Werte abgelegt werden, die den in der Deklaration dieser Variablen vereinbarten Typ besitzen (Typkompatibilität). Die Daten, die eingegeben oder gespeichert werden sollen, müssen den deklarierten Typen entsprechen oder in diese transformiert werden.

Variable für strukturierte Typen

Variable können auch strukturierte Werte annehmen (strukturierte Variable) und bestehen dann aus Komponenten, denen wieder Variable entsprechen. Die Werte dieser Komponenten können selektiv gelesen und verändert werden.
Im nachfolgenden Beispiel handelt es sich bei d um eine **Recordvariable** und bei ds um eine **Feldvariable**. Die Variablen d . t bzw. d . m als Komponenten von d besitzen den Typ integer bzw. string. ds [n] bzw. ds [n] . t bzw. ds [n] . m sind Komponenten von ds mit dem Typ Datum bzw. integer bzw. string.

```
type  Monat = ( Jan , Feb , ... , Dez ) ;
      Datum = record
```

```
                        m : Monat ;
                        t  : 1 .. 31 ;
var  d  :  Datum ;
     ds : array [ 1 .. n ] of Datum
...
d . t  :=  15 ;  d . m  :=  Jan ;
...
ds [n] . m  :=  Dez ; ds [n] . t := 31 ;
...
ds [1]  :=  d ;
```

Feldvariable können **statisch** sein. Ihre Indexmenge wird vor Ablauf des Programms (Compilezeit) festgelegt.

```
const n = 10 ;
type Vector = array [1 .. n]  of  real ;
var a, b : Vector ;
...
b := a ; read (a) ; read (b) ;
```

Bei **dynamischen** Feldvariablen wird die Indexmenge zur Laufzeit des Programms festgelegt.

```
type <Aus i > : integer ; n , m : integer ;
      Vector = array [Aus 1 .. Aus 2] of real ;
var a : Vector (1..10) ;
    b : Vector (m .. n) ;
...
a := b ;
```

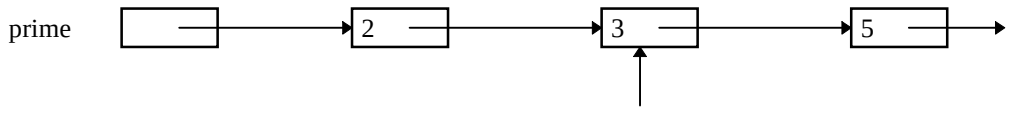
Bei **flexiblen** Feldvariablen ist die Indexmenge durch Wertzuweisung im Programm frei veränderbar.

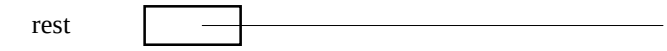
```
type Vector = array [1 .. n] of real ; ...
var   a : Vector ;
...
n := (n - 1) ; read (a) ;
```

Haldenvariable (Zeigervariable)

Variable können durch Befehl angelegt und gelöscht werden. Solche Variable sind im Gegensatz zu denen, die durch eine Deklaration eingeführt werden, anonym und durch Zeiger erreichbar. Sie sind selektiv überschreibbar.

```
typ Intliste = ^ Intknoten
    Intknoten = record kopf : integer,
                       rumpf : Intliste ;
var prim , rest : Intliste ;
...
prime := rest ;
...
prime.kopf := 2 ; prime.rumpf := rest ;
```





Anlegen : allocate liefert Zeiger auf Variable, erst danach zugreifbar
Löschen : dispose trennt Zeigen von Variable, danach nicht mehr zugreifbar

Programme

Bevor die Werte (Speicherinhalte) von Variablen in Anweisungen verwendet bzw. verändert werden können, muß der Typ dieser Variablen vereinbart werden. Werden Variable verwendet, deren Typ vorher nicht erklärt wurde, dann steht ihnen kein geeigneter Speicherbereich zur Verfügung. Die Variable kann demnach nicht oder nicht korrekt angesprochen werden, so daß ein Fehler bei der Abarbeitung der entsprechenden Anweisung entsteht.
Neben Variablen können auch Namen für Konstante eingeführt werden, die vor Benutzung ebenfalls deklariert werden müssen. Anweisungen können zu Unterprogrammen (Prozeduren, Funktionen) zusammengefaßt und über einen Namen aufgerufen werden. Dazu sind entsprechende Unterprogramm-Deklarationen einzuführen.

Wir verwenden folgende Syntaxbeschreibung:

Programm	**program** Name **;** Rumpf
Rumpf	Deklarationsblock Anweisungsblock
Deklarationsblock	FolgevonDeklarationen
FolgevonXyz	leer Xyz **;** FolgevonXyz
Deklaration	**const** FolgevonKonstantendeklarationen **type** FolgevonTypdeklarationen **var** FolgevonVariablendeklarationen FolgevonUnterprogrammdeklarationen
Konstantendeklaration	Name = KonstantenAusdruck **;**
Typdeklaration	Name = Typ **;**
Variablendeklaration	Namenliste **:** Typ **;**
Namenliste	Name Name **,** Namenliste

Unterprogrammdeklaration Prozedurdeklaration
Funktionsdeklaration

Prozedurdeklaration **procedure** Name Parameterteil **;** Rumpf **;**

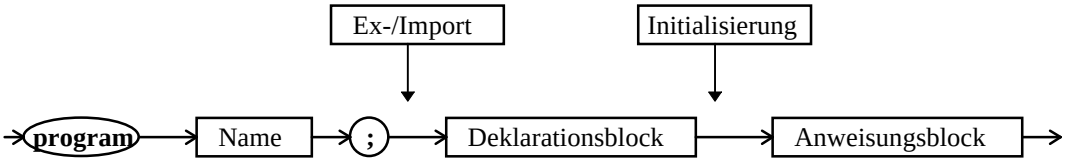
Funktionsdeklaration **function** Name Parameterteil **:** Typ ; Rumpf **;**

Parameterteil **(** Parameterfolge **)**

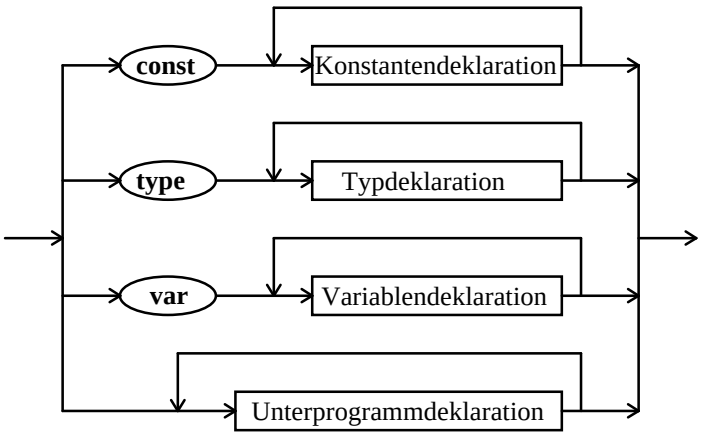
Parameterfolge Variablendeklaration
Variablendeklaration **;** Parameterfolge

Syntaxdiagramme

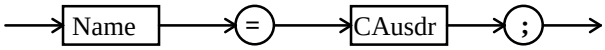
Programm



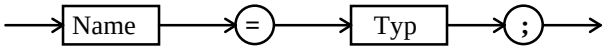
Deklaration



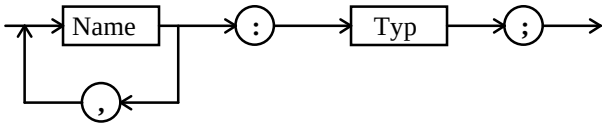
Konstantendeklaration



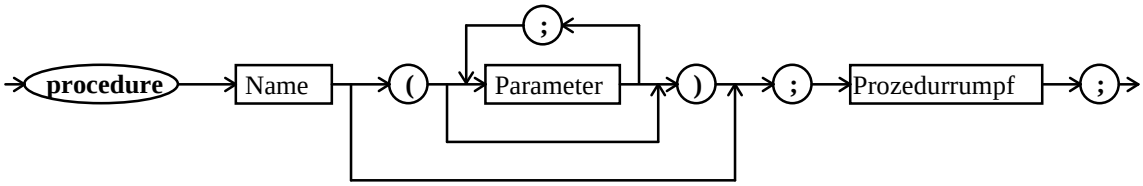
Typdeklaration



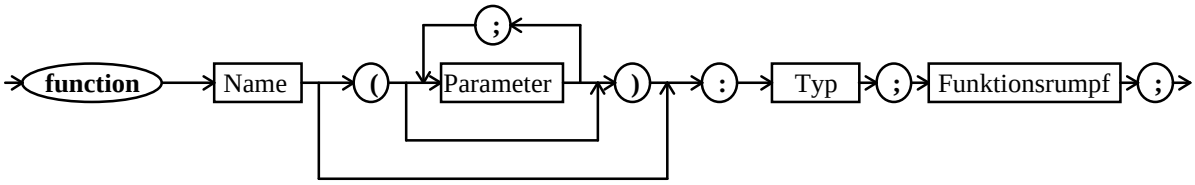
Variablendeklaration



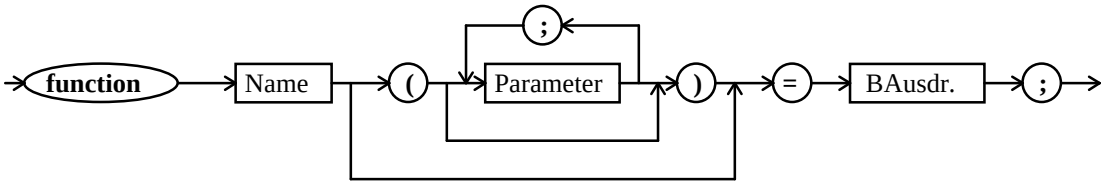
Prozedurdeklaration



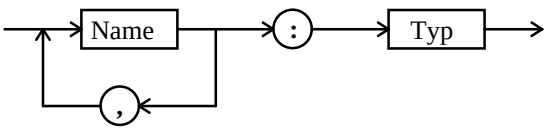
Funktionsdeklaration



oder



Paramter



Prozeduren

In sich abgeschlossene Teilprogramme bzw. Programme für häufig auftretende Verarbeitungsfunktionen werden in Form von Prozeduren oder Funktionen als Unterprogramme verwendet und damit die Übersichtlichkeit des Gesamtprogrammes erhöht bzw. seine Struktur klarer beschrieben. Solche Unterprogramme sollten so erstellt werden, daß sie unabhängig von dem Kontext, in dem sie geschrieben wurden, vielfach verwendbar sind.

Die Bedeutung von Prozeduren liegt in ihren Seiteneffekten (Belegungsänderungen), zum Beispiel der Übergabe von Werten. Zur Definition einer Prozedur gehören folgende Angaben:

- der Bezeichner, unter dem die Prozedur aufgerufen wird,
- für jeden formalen Ein- und Ausgabe-Parameter einen Bezeichner und dessen Typ,
- den Deklarationsteil, lokale, d.h. auf die Prozedur beschränkte, Variablen, Konstanten, Prozeduren, Funktionen und
- einen Anweisungsblock (eingeschlossen in BEGIN und END) mit den in der Prozedur auszuführenden Anweisungen.

Beispiel:

```
program Beispiel ;
    procedure Text3 ( a [1] , a [2] , a [3] : string [10] ) ;        {Prozedur: Text3}
    var i : Integer ;
    {Variablendeklaration}
    begin
        for i := 1 to 3 do
            write ( a[i] , '  ' ) ;
    end ;

begin                                                                {Hauptprogramm}
    Text3 ( 'Funktionen' , 'und' , 'Prozeduren.' ) ;                 {Prozeduraufruf}
end.
```

Die Prozedur wird im Hauptprogramm aufgerufen, indem der Prozedurname gefolgt von drei durch Kommata getrennte (aktuelle) Parameter (Zeichenketten) in Klammern angegeben wird.

BEACHTE: Beim Aufruf der Prozedur müssen immer so viele aktuelle Parameter angegeben werden, wie im Prozedurkopf formale Parameter vereinbart sind. Der Typ der aktuellen Parameter muß dem Typ der formalen Parameter entsprechen.

Funktionen

Eine Funktion (auszuführender Ausdruck) liefert ein Objekt, das an das Hauptprogramm zurückgegeben wird. Eine Funktion beginnt mit dem Terminalsymbol **function** gefolgt vom Namen der Funktion. Der Typ des zurückzugebenden Objektes (Funktionswert) wird nach der Liste der formalen Eingabeparameter getrennt durch einen Doppelpunkt festgelegt. Der Aufruf der Funktion im Hauptprogramm erfolgt wie üblich durch den Funktionsnamen gefolgt von der Liste der aktuellen Parameter.

Beispiel: Funktion zur Berechnung des Volumens eines Quaders.

```
program Beispiel
var  vol : integer ;                              {Variablendeklaration}
function Volumen ( Länge , Breite , Höhe : integer ) : integer ;
        begin
              Volumen := ( (Länge * Breite) * Höhe) ;                  {oder}
              return ( (Länge * Breite) * Höhe) ;                {Return-Anweisung}
        end ;

begin                                             {Hauptprogramm}
        vol := Volumen (5, 7, 1) ;                      {Funktionsaufruf, Wertzuweisung}
        write ( 'Volumen: ', vol ) ;              {Ausgabe}
end.
```

Lebenszeit und Gültigkeitsbereich

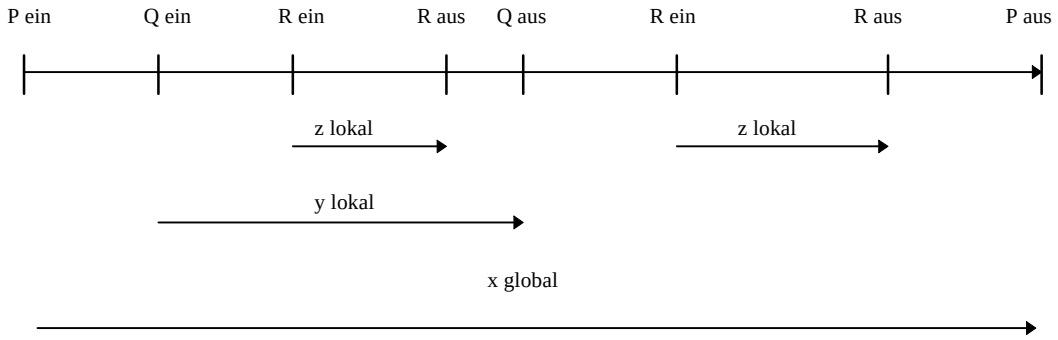
Jede deklarierte Variable besitzt eine Lebenszeit, in der das von ihr referierte Objekt existiert und angesprochen werden kann. Diese entspricht normalerweise der Aktivierungsdauer des Blockes, der die Deklaration für diese Variable enthält und ist dynamisch an die Laufzeit des Programms gebunden.
Der Gültigkeitsbereich (Reichweite) einer Variablen bezeichnet denjenigen Teil des Programms, in dem die Variable mit gleicher Bedeutung (gleiche Speicherzuordnung) benutzt wird. Die Gültigkeitsbereiche, hier nur statisch betrachtet, können geschachtelt auftreten. Verschiedene Objekte können unter dem gleichen Namen auftreten. Aufgerufen werden können die Werte nur im Gültigkeitsbereich.
Wir unterscheiden ferner **persistente** Variablen, deren Lebenszeit die Aktivierung des Programms überdauert, und **transiente** Variablen, deren Lebenszeit an die Aktivierung des Programms gebunden ist.
Lokale Variable sind innerhalb eines Blockes vereinbart und nur dort gültig. Variable, die für das gesamte Programm vereinbart sind, werden **global** genannt.

<u>Beispiel</u>

```
program  P ;
        var x : ... ;
        procedure Q ;
                var y : ... ;
                 begin
                        ... R  ...
                 end ;
```

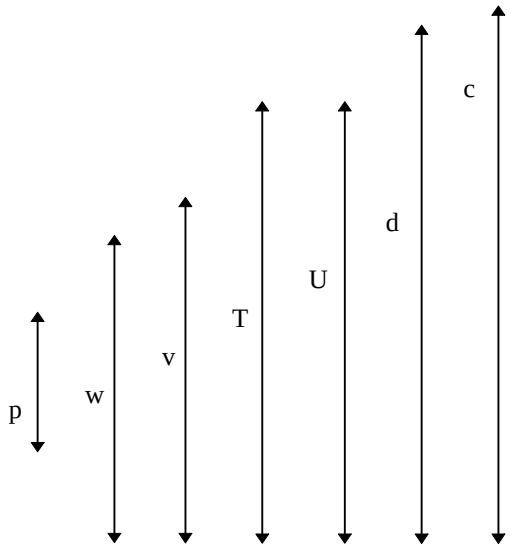
```
procedure R ;
            var z : ... ;
             begin
                      ...
             end ;

begin
      ... Q ... R  ....
end .
```

Im Beispielprogramm besitzen die Variablen x, y, z bei der unten vorgegebenen Abarbeitung (Laufzeit) folgende Lebensbereiche:



Nicht nur Variable, sondern auch andere durch Deklarationen oder Definitionen eingeführte Bezeichner besitzen einen Gültigkeitsbereich, gebunden an den Programmblock, in dem sie deklariert werden.

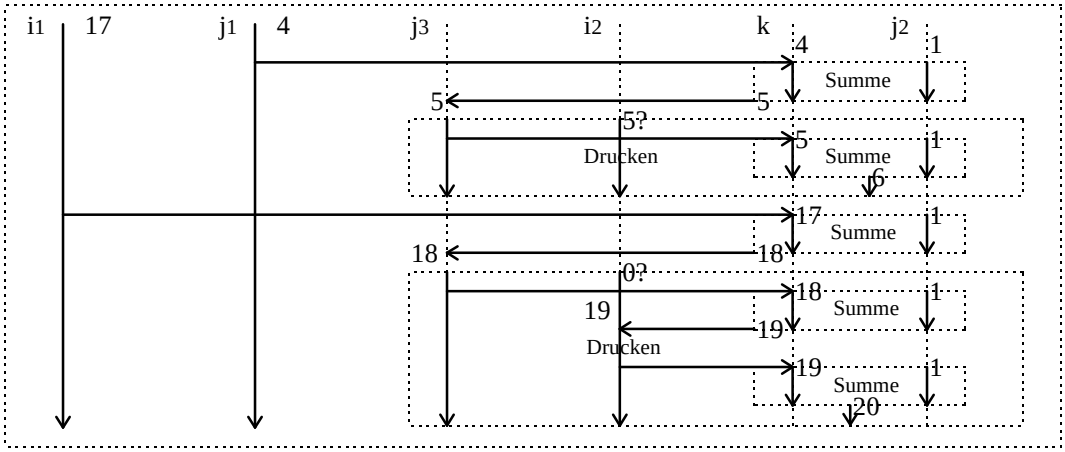
```
const  c = ... ;
       d = ... ;

type  T =  ^ U ;
      U =  ... T ... ;

var   v : T ;
     w : ... ;

procedure  p ;

      begin
            ...  p  ...  v  ...  ;
      end ;
begin
        ...
end.
```



Im nachfolgenden Programm werden die Bezeichner i und j in verschiedenen Gültigkeitsbereichen mit verschiedenen Bedeutungen und Referenzen benutzt, die durch Indizierung unterschieden werden sollen.

```
program p ;
var i1, j1 : integer ;
function Summe ( k : integer ) : integer ;
var j2 : integer ;
begin if i1 < 10 then j2 := 10 else j2 := 1 ;
      return ( j2 + k )
end ;
procedure Drucken ( j3 : integer ) ;
var i2 : integer ;
begin if i2 ≠ j3 then i2 := Summe ( j3 ) ;
      write ( Summe ( i2 ) )
end ;
begin i1 := 17 ; j1 := 4 ;
      Drucken ( Summe ( j1 ) ) ;
      Drucken ( Summe ( i1 ) )
 end
```

Für die Lebensdauer der einzelnen Variablen und deren zugeordnete Werte erhält man für den beschriebenen Ablauf im Beispiel folgende Ergebnisse:



Die in der Prozedur Drucken deklarierte lokale Variable i wird im Anweisungsblock dieser Prozedur bei obigen Ablauf aufgerufen, bevor ihr ein Wert zugeordnet wurde (Laufzeitfehler). Im Beispiel wird dazu an den mit ? gekennzeichneten Stellen ein bestimmter Wert vorausgesetzt. Um solche Laufzeitfehler zu vermeiden, muß vor Benutzung lokaler Variablen geprüft werden, ob ihr durch die Prozedur bereits ein Wert zugewiesen ist. Ansonsten ist die Verwendbarkeit dieser Prozedur eingeschränkt bzw. führt zu Fehlverhalten des Gesamtprogramms.

Parameterübergabe (Kopiermechanismus)

Die beim Aufruf von Prozeduren und Funktionen verwendeten aktuellen Parameter müssen zu den in der Deklaration angegebenen formalen Parametern in Beziehung gesetzt (kopiert) werden. Dafür gibt es unterschiedliche Mechanismen, deren wichtigsten in nachfolgend zusammengestellt sind:

	Call by Value	**Call by Reference**	**Call by Name**
Übergabe	Wert	Speicherzuordnung	Name
Kopie	Daten	Adressen	Bezeichner
Auswertung	strikt	normal	normal
Zeitpunkt	Aufruf	Laufzeit	Übersetzung
Sprache	Pascal	Fortran	Algol

Diese Mechanismen können in Programmiersprachen auch gemischt auftreten, wie z.B. Call by Value und Call by Reference in Pascal. Ihre Auswahl kann durch die Syntax der Sprache erfolgen.
Als Parameter kommen außer Bezeichner von Dateien alle anderen deklarierten Namen in Frage. Auch Prozedur- und Funktionsnamen können übergeben werden. Man unterscheidet demnach Konstantenparameter, Variablen(Referenz)parameter, Prozedurparameter, Funktionsparameter.
Die formalen Parameter sind lokale Variablen einer Abstraktion (Unterprogramm). Beim Eintritt in die Abstraktion wird in die lokale Variable kopiert, beim Austritt aus

der Abstraktion in die globale Variable. Die lokale Variable wird beim Eintritt in die Abstraktion angelegt und beim Austritt wieder gelöscht.

Rekursion und Iteration

Wie bereits bei der syntaktischen Beschreibung der Steuerstrukturen geschehen, können auch Programme Funktionen oder Prozeduren enthalten, in deren Deklaration ihre eigenen Funktions- bzw. Prozedurnamen wieder auftreten, die sich beim Programmlauf dann selbst aufrufen. Derartige Beschreibungen bzw. Funktionen und Prozeduren heißen **rekursiv**. So kann z.B. die Fakultätsfunktion fac durch die Gleichung

$$\text{fac}(n) = \begin{cases} 1 & n \leq 1 \\ (n * \text{fac}(n-1)) & \text{sonst} \end{cases} \quad \text{falls}$$

rekursiv definiert werden. (Die Funktion fac ist Lösung der Gleichung.) Die Werte von fac für beliebige natürliche Zahlen n sind durch die Gleichung eindeutig festgelegt, da die natürlichen Zahlen vollständig geordnet sind und ausgehend von n mit fortlaufender Bildung von n-1 (Vorgänger) diese Ordnung bis zu 1 (kleinstes Element) durchlaufen wird. Wenn der Wert von fac für eine Zahl (n-1) bekannt ist, kann er durch den Ausdruck im Fall sonst für den Nachfolger n berechnet werden.
Im Beispiel der Funktion fac kann dies durch folgendes Programm notiert werden:

```
function fac (arg : integer ) : integer ;
        begin
            if arg ≤ 1 then return 1
                    else return ( arg * fac (arg - 1 ) )
         end ;
```

Bei jedem Aufruf von fac müssen neue Inkarnationen von arg und fac erzeugt werden.
Der Wert von arg ist jeweils zu speichern. Die Berechnung von fac ist zu unterbrechen und mit dem Wert von (arg - 1) für arg neu zu starten.
Bei Erreichen von 1 (Abbruchkriterium) ist der Wert von fac für die letzte Inkarnation berechnet. Durch sukzessives Multiplizieren mit den Werten der Inkarnationen von arg in umgekehrter Reihenfolge ihrer Speicherung wird schließlich der Wert von fac für den anfänglichen Wert von arg erhalten. Der Speicher ist nach dem Prinzip last in - first out als Kellerspeicher zu verwalten. Für arg = 4 ergibt sich folgender Ablauf:

arg		return-Anweisung		fac
4	↓	**return** (4 * fac (3))	↑	24
3	↓	**return** (3 * fac (2))	↑	6
2	↓	**return** (2 * fac (1))	↑	2
1	↓	**return** 1	↑	1

Bei derartigen rekursiven Programmen, der Selbstaufruf kann direkt oder indirekt über andere Unterprogramme erfolgen, ist immer der Speicheraufwand für den Keller zu berücksichtigen. Insbesondere bei mehrfachen Rekursionen kann dies zu erheblichen Speicherverbrauch und zu einer großen Anzahl von rekursiven Aufrufen (Rechenzeit) führen.

Ein bekanntes Beispiel für Rekursion wurde von Fibonacci angegeben. Er berechnete als fib (n) die Anzahl der Kaninchen-Paare, die nach n Jahren leben würden, wenn im ersten Jahr ein Paar exisitiert, jedes Paar vom zweiten Jahr an ein weiteres Paar als Nachwuchs hat und die Kaninchen eine unbeschränkte Lebensdauer besitzen. Damit entsteht die Zahlenfolge :

Jahr	1	2	3	4	5	6	7	8	9	10	30
Paare	1	1	2	3	5	8	13	21	34	55	832040

Als Funktion läßt sich fib definieren durch die Gleichungen:

$fib(n) = 1$ falls $n \leq 2$

$= (fib(n-1) + fib(n-2))$ falls $n > 2$

Als Funktionsprozedur erhält man damit das rekursive Programm:

```
function fib ( arg : integer ) : integer ;
        begin
            if arg ≤ 2 then return 1
                    else return ( fib ( arg - 1 ) + fib ( arg -2 ) )
        end ;
```

Der Aufwand bei der Berechnung von Fibonacci-Zahlen fib (n) mit diesem Programm ist proportional der Anzahl c_n der Aufrufe von fib. Für c_1 und c_2 ergibt sich 1.
Weiter gilt $c_n = 1 + c_{n-1} + c_{n-2}$ für $n > 2$ und $c_{n-1} = 1 + c_{n-2} + c_{n-3}$ für $n > 3$.
Zusammen erhält man $c_n = 2 + 2c_{n-2} + c_{n-3} > 2c_{n-2} \ldots > 2^{n/2-1}c_2$ und
$c_n = 2c_{n-1} - c_{n-3} < 2c_{n-1} \ldots < 2^{n-1}c_1$.
Also ergibt sich $2^{n/2-1} < c_n < 2^{n-1}$, d.h., ein exponentielles Wachstum von c bei wachsendem n. Bei $n \geq 40$ entsteht eine unverhältnismäßig hohe Rechenzeit. Außerdem entsteht eine große Rekursionstiefe, was zu einem inakzeptablen Aufwand für den Kellerspeicher führt.
Die Effizienz der Berechnung kann hier dadurch deutlich gesteigert werden, wenn die Funktion fib nicht durch ein rekursives, sondern durch ein iteratives Verfahren berechnet wird. Ein solches zeichnet sich dadurch aus, daß nur bestimmte Teilabschnitte des Verfahrens gesteuert durch eine Abbruchbedingung wiederholt durchlaufen werden, ohne daß das gesamte Verfahren selbst wieder neu angestoßen wird. Der Fortschritt des Verfahrens ist zu speichern und wird im nächsten Durchlauf wieder verwendet. Mit den vorher beschriebenen Wiederholanweisungen läßt sich dies im Beispiel der Fibonacci-Zahlen **iterativ** wie folgt ausdrücken:

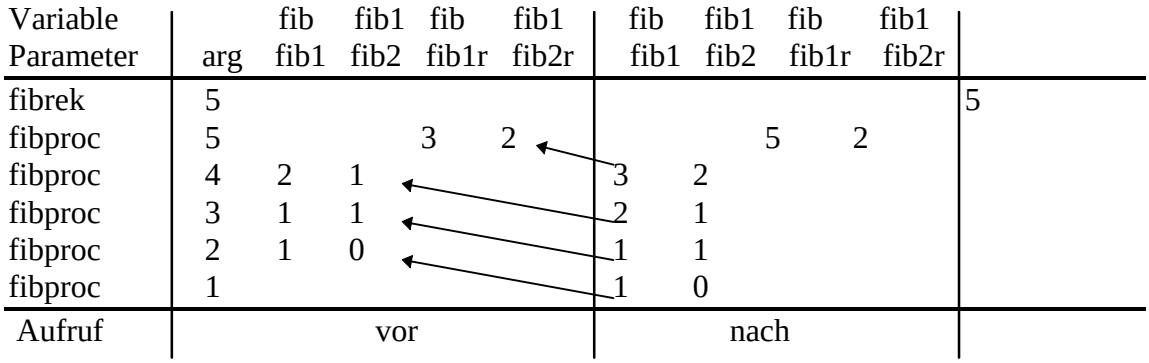
```
function fibit ( arg : integer ) : integer ;
        var i , x , y , fib : integer ;
        begin
          if arg = 0  then return error
                  else  begin
                          x :=  0 ; fib := 1 ;
                          for i := 2 to n do begin
                                              y := ( fib + x ) ;
                                              x := fib ;
                                              fib := y
                                            end ;
                          return fib
                        end
        end ;
```

Die zuletzt berechneten Fibonacci-Zahlen sind jeweils in x und fib gespeichert. Als nächste Fibanocci-Zahl entsteht der Wert von y. Für den nächsten Durchlauf wird dann fib auf x und y auf fib umgespeichert. (Dies kann auch durch Rotation erreicht werden.) Insgesamt werden hier zur Berechnung von fibit (n) in der for-to-Schleife (n - 1) solche Umspeicherungen vorgenommen, was einem linearen Aufwand entspricht. Die Funktion selbst wird nur einmal aufgerufen.
Neben der for-to-Schleife können auch die repeat-until- (bzw. do-while-) Schleife oder die while-do- (bzw. while-) Schleife benutzt werden. Dabei ist zu beachten, daß bei repeat die Schleife mindestens einmal durchlaufen wird, bevor die Abbruchbedingung geprüft wird. Bei while hingegen wird erst die Abbruchbedingung geprüft und danach die Anweisungen der Schleife ausgeführt. Letzteres reduziert im allgemeinen die Fehleranfälligkeit.

Daß auch rekursive Lösungen effizient sein können, belegt das folgende Programm, bei dem eine rekursive Prozedur mit den Referenzparametern fib und fib1 verwendet wird. Bei diesen formalen Parametern soll der früher erwähnte Kopiermechanismus nach Call by Reference vorliegen, was durch die Variablendeklarartion **var** in der Parameterliste ausgedrückt ist.

```
function fibrek ( arg : integer ) : integer ;
        procedure fibproc ( arg : Integer ; var fib , fib1 : intger ) ;
                var fib2 : integer ;
                begin if arg ≤ 1 then begin fib : = arg ; fib1 := 0 end ;
                                 else  begin
                                         fibproc ( ( arg - 1 ) , fib1 , fib2 ) ;
                                         fib := ( fib1 + fib2 )
                                       end
                end ;

        var fib1r , fib2r : integer ;
        begin  fibproc ( arg , fib1r , fib2r ) ;
               return fib1r
        end ;
```

Die verwendete Prozedur fibproc liefert den Wert fib für das Argument arg und den Wert fib1 für das Argument (arg - 1). Die Variable fib2 wird in zweifacher Bedeutung verwendet, direkt und als Ergebnis-Parameter, aber nur einmal berechnet. Die Anzahl der Aufrufe von fibproc in der Funktion fibrek steigt nur noch linear mit dem Argument. Eine Beispielberechnung von fibrek (4) zeigt die nachfolgende Tabelle, in der die linke Tafel den Variablen-Keller vor und die rechte Tafel den Variablen-Keller nach den jeweiligen Prozeduraufrufen darstellt.



Der hierbei erzielte Effizenzgewinn gegenüber der vorhergehenden rekursiven Lösung wird durch eine erhöhte Komplexität des Programms erkauft. Im prozeduralen Programmierstil würde man die iterative Lösung sicher bevorzugen. Bei Paradigmenwechsel, etwa zum funktionalen Programmierstil, wird dies im allgemeinen nicht mehr gelten.

Regeln für die prozedurale Programmierung (im Kleinen)

- Zerlegung des Gesamtverfahrens in überschaubare und hierarchisch gegliederte Teile (Bausteine)

- Konzeption funktioneller umgebungsunabhängiger Einheiten (Module)

- Unterprogramme wiederverwendbar konzipieren

- Objekttypen problemadäquat definieren

- Abstraktion der Daten (Struktur, Eigenschaften, Typ, Zugriffsoperation) festlegen

- auf Lokalität von Variablen und abgegrenzte Gültigkeitsbereiche achten

- unnötigen Export aus und Import in Unterprogramme vermeiden

- Kommunikation zwischen Programmteilen mit möglichst wenig Parametern (Schnittstellen)

- Variable nicht über (Unter-)Programmgrenzen hinaus verwenden

- Verwendung weitgehend selbsterklärender Bezeichner

- Programmnotationen strukturieren, aus der syntaktische Struktur erkennbar ist

- Unterprogramme und Anweisungen kommentieren, so daß daraus deren Bedeutung ableitbar ist

- Typ von Konstanten und Variablen deklarieren

- auf Konsistenz von Daten- und Steuerstrukturen achten

- Variable vor Benutzung initialisieren (Vermeidung von Laufzeitfehlern)

- unkontrollierte Sprünge (goto- Anweisungen) vermeiden

- Eingabefehler und Typverletzungen möglichst ausschließen (Robustheit)

- bei rekursiven Programmen Keller- und Aufruftiefe (Speicher- und Zeiteffizienz) beachten

- weitgehende Unabhängigkeit von Compiler und Gerätekonfiguration anstreben

- Laufzeitverhalten und Ressourcenverbrauch erfassen (Qualitätsparameter)

- Testprotokolle festlegen (Eingabetest, Laufzeittest, Vollständigkeitstest)

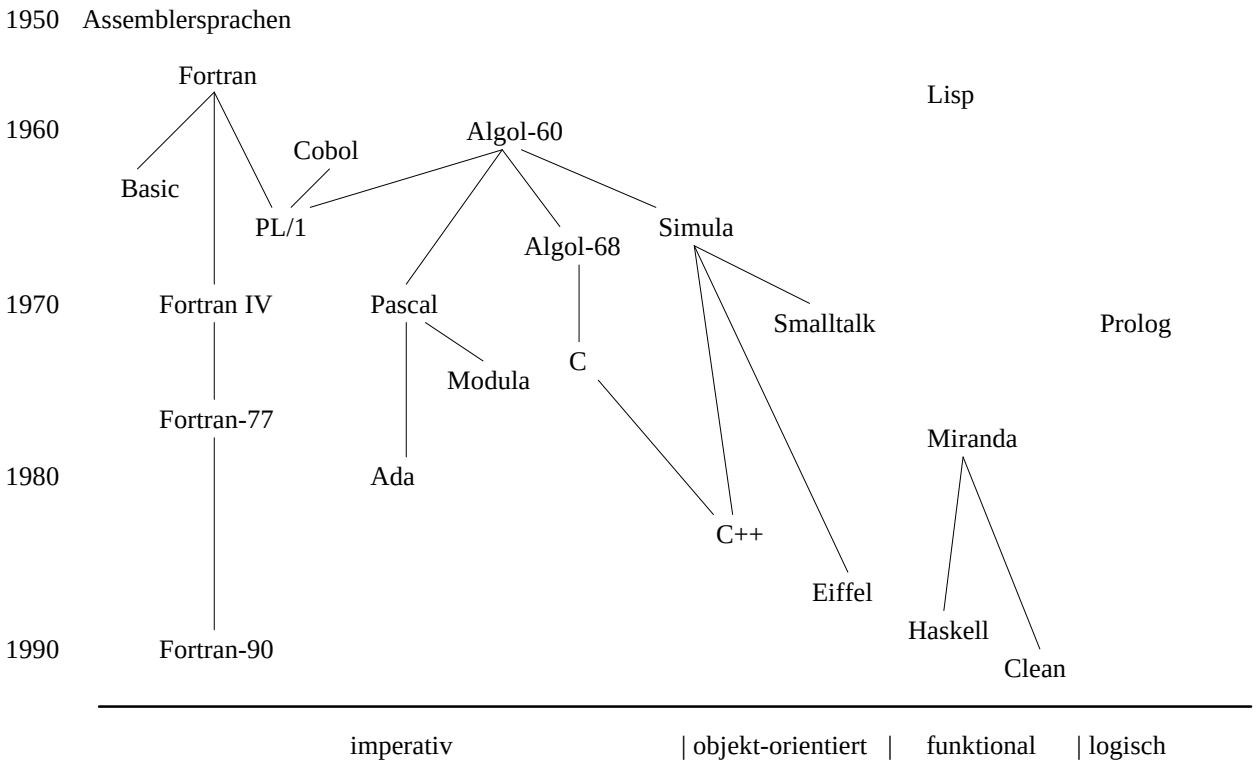
II Programmierung und Programmiersprachen

1. Entwicklung der Programmiersprachen

Programmiersprachen führen Notationsvorschriften für Algorithmen bzw. der von diesen beschriebenen Prozesse ein. Je nachdem welche Konstrukte in der jeweiligen Sprache zugrundegelegt werden, handelt es sich um Notationsformen die dem Computer (Maschinen-, Assemblersprachen) oder dem Algorithmus bzw. dem Nutzerkonzept (Problemorientierte Sprachen) näher stehen. Die Programmiersprache dient als Verständigungsmittel zwischen Maschine und Nutzer. Die Übertragung der problemorientierten Nutzersprache bzw. einer ihr nahestehenden Notationsform in die Maschinensprache erfordert einen Übersetzungsvorgang. Bei dieser Übertragung werden häufig Zwischensprachen verwendet, die den Übersetzungsvorgang in mehreren Schritten auszuführen gestatten.

Programmiersprachen werden seit den fünfziger Jahren erfolgreich eingesetzt. Für unterschiedliche Problemklassen sind unterschiedliche Sprachen und Sprachtypen nach unterschiedlichen Konzepten entwickelt worden. Einen Ausschnitt dieser Entwicklung zeigt die folgende Übersicht.

Für konkrete Problemklassen wurden anwendungsspezifische Sprachen entwickelt, deren Allgemeinheit durch zunehmende Abstraktion gesichert wird. Von großer Bedeutung für die Akzeptanz und Effektivität von Programmiersprachen sind Software-Entwicklungs-Umgebungen zur Erzeugung von fehlerfreier, robuster und effizienter Software auf der Basis anwendungsnaher Benutzeroberflächen.



2. Programmierparadigmen

Jeder Programmiersprache liegt ein bestimmtes Konzept (pragmatisches Begriffssystem) zugrunde. Nach diesem Konzept, auch Paradigma genannt, kann man verschiedene Klassen von Programmiersprachen unterscheiden:

Stil	Beschreibung	Definitionsform	Aktivierungskonzept	Ausführungskonzept	Beispiel
imperativ	prozedural	Prozedur	Prozedur-Aufruf	Wertzuordnung über Variablen	PASCAL
applikativ	funktional	Funktion	Funktionsanwendung mit Parametervermittlung	Wertbestimmung	Miranda
deklarativ	logisch	Regelsystem	Unifikation	Ableitung (Beweis)	PROLOG
direktiv	objektorientiert	Objektklasse	Instantierung	Kommunikation zwischen Klassen	C++

Programmbeispiel

Die verschiedenen Programmierparadigmen sollen an Hand eines Beispiels demonstriert werden. Dabei verwenden wir die im früher eingeführten Notationen bzw. sind diese in Anlehnung an Funktions- und Relationsschreibweisen selbsterklärend.
Wir betrachten nach Appelrath, Ludewig [1] folgende Problemstellung:

Ein Kunde möchte in einem Milchladen eine bestimmte Anzahl (*Soll*) von Litern Milch kaufen und bringt dazu ein Gefäß ohne Maßeinteilung mit, welches ein Fassungsvermögen von *Gmax* Litern hat. Der Verkäufer besitzt seinerseits einen Eimer mit einer Eichmarke bei *Emax* Litern. Wie kann der Verkäufer den Wunsch des Kunden erfüllen?

"Prozeduraler Milchladen"

Betrachten wir zunächst ein prozedurales Lösungskonzept. Bei *Soll = 4*, *Gmax = 7* und *Emax = 10* wäre vom Verkäufer folgende Handlungsfolge durchzuführen.

Handlungsfolge für 4 Liter	danach sind Liter im Gefäß	Liter im Eimer
Fülle G	7	0
Umfüllen von G nach E	0	7
Fülle G	7	7
Umfüllen von G nach E	4	10
Ausgießen E	4	0

Nach fünf Schritten befinden sich 4 Liter im Gefäß, das Ziel ist erreicht.
Wollte man dagegen 5 Liter kaufen, so muß eine neue Handlungsfolge durchgeführt werden:

Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;
Ausgießen G; Umfüllen E nach G;
Fülle E; Umfüllen E nach G; Ausgießen G; Umfüllen E nach G;

Diese Folge liefert das gewünschte Ergebnis von 5 Litern im Gefäß. Verschiedene Handlungen wiederholen sich. Die "Abweichung" im 13. Schritt kommt dadurch zustande, daß nach dem Umfüllen ein Rest im Eimer verblieben ist.
Diese Folge kann durch eine WHILE-DO-Schleife beschrieben werden.(Solange in E noch mindestens so viel Milch ist, wie maximal in G hineinpaßt, führe aus ...): WHILE E ≥ 7 DO... . Die Abbruchbedingung des Programms ist erfüllt, wenn sich die gewünschte Menge Milch im Gefäß befindet.

In PASCAL-ähnlicher proceduraler Notation kann die Handlungsfolge beschrieben werden durch:

```
begin
    G:=0; E:=0    (G leer; E leer)
    while G ≠ 5 do
      begin
        E:= 10;    (Fülle E)
        while E ≥ 7 do
          begin
            E:= E - ( 7 - G ); G:= 7;    (Umfüllen von E nach G)
            G:= 0;    (Ausgießen von G)
          end;
        G:= G + E ;   (Umfüllen von E nach G)
        E:= 0;          (E leer)
      end;
end
```

Wollen wir den Algorithmus verallgemeinern, so ersetzen wir die Größen von Gefäß und Eimer durch Variablen (Gmax und Emax). Es gilt dabei zu beachten, daß das Problem nicht in jedem Fall lösbar ist. Eine hinreichende Bedingung für die Lösbarkeit ist, daß das gewünschte Ergebnis *Soll* ein Vielfaches des größten gemeinsamen Teilers von Gmax und Emax ist.
Der allgemeine Algorithmus nach dem prozeduralen Paradigma besteht aus folgenden Proceduren mit den Typ-Deklarationen für die Variablen E (Eimer), G (Gefäß) , Gmax (Fassungsvermögen Gefäß), Emax (Fassungsvermögen Eimer), Soll (Sollmenge):

```
program PROCMilchladen
 var E, G, Gmax, Emax, Soll: Integer;

procedure LeereG;
 begin
    Aktion(' Ausgießen G' );
     G:=0;
end;

procedure FülleE;
 begin
   Aktion (' Fülle E ');
   E:= Emax;
 end;

procedure EnachG;
 begin
    Aktion (' Umfüllen E nach G. ');
    while (E>0) and (G<Gmax) do
      begin
        E:=E - 1;
        G:=G + 1;
      end;
 end;

function ggT (a, b: Integer): Integer;
 var Rest: Integer;
 begin
     while a>0 do
       begin
         Rest:=b mod a;
         b:= a;
         a:= Rest;
       end;
     ggT:= b;
end;
```

```
begin     {Hauptprogramm}
 if  Soll mod ggT( Gmax, Emax) = 0  then
   begin
     G:=0;
     E:=0;
     while G<>Soll do
       begin
         FülleE;
         EnachG;
         repeat
             LeereG;
             EnachG;
          until E=0;
       end;
   end
  else Aktion( ' Bedienung ablehnen. ');
end;
```

Bei Gmax = 31, Emax = 40 und Soll = 27 ergibt sich in Ausführung des Programms folgende Aktionsfolge:
Fülle E; Umfüllen E nach G, Ausgießen G, Umfüllen E nach G
Fülle E; Umfüllen E nach G, Ausgießen G, Umfüllen E nach G
Fülle E; Umfüllen E nach G, Ausgießen G, Umfüllen E nach G

Funktionaler "Milchladen"

Beim funktionalen Paradigma interessieren die auszuführenden Funktionen und deren Auswertung. Die Lösung des Problems ist durch sukzessives Verändern der Füllzustände von Eimer und Gefäß zu erreichen. Wir führen deshalb eine Funktion S (*Situation*) ein, die abhängt von den Parametern:

x	(Istmenge im Eimer)
y	(Istmenge im Gefäß)
Emax	(Fassungsvermögen des Eimers)
Gmax	(Fassungvermögen des Gefäßes)
Soll	(Sollmenge)

Die neu entstehende Situation nach Ausführung einer Aktion läßt sich als Funktion S wie folgt beschreiben:

S (x, y, Emax, Gmax, Soll) =

falls y = Soll dann Aktion *NIL*, (keine Aktion) sonst
falls y = Gmax dann "AUSGIESSEN G" und *S(x,0,Emax,Gmax,Soll)*; sonst

falls x = 0	dann "FÜLLE E" und *S(Emax,y,Emax,Gmax,Soll)*; sonst
falls	dann "UMFÜLLEN E NACH G" und
x+y ≥ Gmax	*S(x-Gmax+y, Gmax, Emax, Gmax, Soll)*; sonst
falls	dann "UMFÜLLEN E NACH G" und
x+y < Gmax	*S(0,x+y, Emax, Gmax, Soll)*;

Wir beschreiben diesen Sachverhalt durch eine Funktionsdefinition wie folgt:

```
define function S (x, y, Emax, Gmax, Soll)=
 if  y = Soll then NIL
  else if  y = Gmax then  ('Ausgießen von G ')∗ S(x, 0, Emax, Gmax, Soll)
     else if  x=0 then  ('Fülle E ') ∗ S(Emax, y, Emax, Gmax, Soll )
          else if  x+y >=Gmax then  ('Umfüllen von E nach G ') ∗
          S(x-Gmax+y, Gmax, Emax, Gmax, Soll)
               else ( 'Umfüllen von E nach G') ∗ S(0, x+y, Emax, Gmax, Soll)
```

Durch ∗ wird das Verketten (Hintereinanderausführung) von Funktionen bzw. Aktionen bezeichnet.
Die Funktion S ruft sich selbst wieder auf, d.h. sie ist rekursiv definiert.
Mit Hilfe einer zweiten Funktion A legen wir die Anfangswerte fest und schließen Fehleingaben (keine Lösungsmöglichkeit) aus.
Damit erhalten wir folgende rekursive Definition:

```
define function A( Emax, Gmax, Soll) =
  if  Gmax < 1 then  ('Gefäß nicht vorhanden !') ∗ NIL
  else if  Soll > Gmax then  ('Gefäß zu klein') ∗ NIL
       else if  Soll < 0 then ('Negative Literzahl nicht möglich !') ∗ NIL
            else if  Gmax > Emax then  ('Rollentausch Eimer/ Gefäß') ∗
            A(Gmax, Emax, Soll)
                 else if  Soll mod ggT(Emax, Gmax)  ≠ 0 then
                      ('Keine Lösung möglich') ∗ NIL
                      else S(0, 0, Emax, Gmax, Soll)
```

Innerhalb dieser Funktion wird eine weitere Funktion zur Berechnung des "Größten gemeinsamen Teilers" aufgerufen (durch ggT(Emax, Gmax)). Diese kann rekursiv wie folgt definiert werden:

```
define function ggT(a, b) =
  if  b = 0 then a
   else ggT(b, a mod b)
```

Ein funktionales Programm besteht also aus einer Reihe von Funktionsdefinitionen und einem initialen Funktionsaufruf. Dieser lautet in unserem Fall: A(k, l, m). Es wird nach diesem Aufruf die Schrittfolge erstellt, um m Liter mit Hilfe eines k-Liter großen Eimers und eines l-Liter großen Gefäßes abzumessen. Jede zu berechnende Funktion muß vorher definiert sein.

Für das Beispiel Soll = 6, Gmax = 7, Emax = 10 ergeben sich folgende Funktionsaufrufe:

	A(10,7,6)
	S(0,0,10,7,6)
Fülle E	S(10,0,10,7,6)
Umfüllen von E nach G	S(3,7,10,7,6)
Ausgießen G	S(3,0,10,7,6)
Umfüllen von E nach G	S(0,3,10,7,6)
Fülle E	S(10,3,10,7,6)
Umfüllen von E nach G	S(6,7,10,7,6)
Ausgießen G	S(6,0,10,7,6)
Umfüllen von E nach G	S(0,6,10,7,6)
	NIL

Logikbasierter "Milchladen"

Im logikbasierten Paradigma wird das Problem durch *Eigenschaften* (Prädikate) in Form von *Tatsachen* (Fakten*)* und *Beziehungen* (Regeln) beschrieben.
Die Ziel-Eigenschaft (Ziel) besteht darin, im Gefäß (y) durch eine Aktionsfolge (p) die Sollmenge (Soll) herzustellen: Ziel(Soll, p).
Die durch eine Aktionsfolge(p) hergestellte Situation "x Liter im Eimer und y Liter im Gefäß" wird durch die Eigenschaft *Situation(x,y,p)* beschrieben. Die Eigenschaft *gültig(x,y)* besagt, daß die genannte Situation überhaupt möglich ist.
In unserem Beipiel ist diese Eigenschaft erfüllt, wenn $(x \geq 0)$ und $(y \geq x)$ und $(x \leq Emax)$ und $(y \leq Gmax)$. Diese Beziehung schreiben wir als Regel in der Form:

$$gültig\ (x, y) \longleftarrow (x \geq 0),\ (y \geq x),\ (x \leq Emax),\ (y \leq Gmax),$$

links von ⟵ steht dabei die Folgerung aus den rechts von ⟵ stehenden Voraussetzungen. Die logische Und-Verknüpfung wird durch Kommata notiert.
Ähnliche Beziehungen können für die Herstellung von Situationen notiert werden:

1.)
situation (x, y, [p, "Fülle E"]) ← situation (0, y, p), (x = Emax)
2.)
situation (x, y, [p, "Fülle G"]) ← situation (x, 0, p), (y = Gmax)
3.)
situation (x, y, [p, "Ausgießen E"]) ← situation (u, y, p), (x = 0)
4.)
situation (x , y, [p, "Ausgießen G"]) ← situation (x, v, p), (y = 0)
5.)
situation (x, y, [p, "Von G nach E"]) ← situation (u, v, p), (x = u + v), (x < Emax), (y = 0)

6.)
situation (x, y, [p, "Von E nach G"]) ← situation (u, v, p), (y = u+v),
(y < Gmax), (x = 0)

7.)
situation (x, y, [p, "Von G nach E"]) ← situation (u, v, p), (x = Emax),
(x + y = u + v)

8.)
situation (x, y, [p, "Von E nach G"]) ← situation (u, v, p), (y=Gmax),
(x + y = u + v)

Die erste Regel besagt: Wenn mit der Aktionsfolge p die Situation im Eimer 0 Liter und im Gefäß y Liter erzeugt wurde und x=Emax ist, dann wird nach der Aktionsfolge p; "Fülle E" die Situation x Liter im Eimer und y Liter im Gefäß vorliegen.

Die Variablen gelten nur für jeweils eine Regel und werden bei jeder Anwendung neu erzeugt, d.h sie sind nur *lokal* gültig.
Nach der Aktionsfolge p ist die Zieleigenschaft *Ziel(y,p)* hergestellt, wenn durch diese Aktionsfolge die Eigenschaft *Situation(x,y,p)* erfüllt wird und diese Situation außerdem gültig ist, d.h. die Eigenschaft *gültig(x,y)* zutrifft. Diese Beziehung kann durch die Regel

Ziel(y,p)←———*situaiton(x,y,p), gültig(x,y)*

beschrieben werden.
Neben den obigen Regeln braucht man eine Reihe von Fakten (Situationen ohne Voraussetzungen).
In unserem Beispiel legen wir entsprechend der Problemstellung die Fakten:

für p : situation (0, 0, []) ←———
für Emax : (Emax = 10) ←———
für Gmax : (Gmax=7) ←———

fest, wobei [] die leere Aktionsfolge NIL bezeichnet.
Wir fragen danach, welche Aktionsfolge p führt zur Zieleigenschaft " Soll = drei Liter im Gefäß", also ziel(3 , p).
Der *Logik-Interpreter* versucht, die gewünschte Eigenschaft aus den Fakten und Regeln herzuleiten, indem er die Variablen schrittweise mit Werten belegt, so daß die Fakten und Regeln erfüllt werden.
Welche der obigen Fakten und Regeln in welcher Reihenfolge benutzt werden ist dabei nicht vorgeschrieben.

Das Vertauschen von Regeln kann dazu führen, daß das Programm in unendliche Schleifen läuft, d.h. niemals mit der erfüllten Zieleigenschaft abbricht. Die Strategie für die Anwendung der Regeln beeinflußt die Effektivität des Verfahrens wesentlich. Um effiziente Programme im logikbasierten Paradigma zu entwicklen, sind Kenntnisse über die vom Logik-Interpreter benutzte Strategie hilfreich.

Beispiel: Mit den oben festgelegten Fakten wird die Zieleigenschaft Ziel(3,p) gesucht. Die Herleitung des Ziels wird duch die folgende Ableitungskette beschrieben.

	Regel
situation(0,0,[])	
situation(10,0,["Fülle E"])	1
situation(3,7,["Fülle E","Von E nach G"])	8
situation(3,0,["Fülle E","Von E nach G","Ausgießen G"])	4
situation(0,3,["Fülle E","Von E nach G","Ausgießen G","Von E nach G"])	6
gültig(0,3)	

Ziel(3,["Fülle E","Von E nach G","Ausgießen G","Von E nach G"])

Objektorientierte Programmierung

Das objektorientierte Paradigma basiert auf der Definition von *Objekten*, die an der Problemstellung beteiligt sind. In unserem Fall sind das:

- der Eimer (mit Fassungsvermögen Emax)
- das Gefäß (mit Fassungsvermögen Gmax)
- der Milchtank (Speicher)
- der Verkäufer (Laden)

Jedes dieser Objekte hat bestimmte Eigenschaften, die Klassen gleichartiger Objekte bestimmen. Die Objekte einer Klasse können Operationen (Methoden) ausführen.

Eimer und Gefäß sind z.B. Behälter, die sich nur durch ihr Fassungsvermögen unterscheiden. Die Objektklasse Behälter hat damit folgende Eigenschaften:

1. Maximales Fassungsvermögen MAX (natürliche Zahl)
2. Aktueller Inhalt INH (natürliche Zahl zwischen 0 und MAX)
3. Kann auf das maximale Fassungsvermögen aufgefüllt werden.
4. Kann entleert werden.
5. Kann von einem anderen Behälter Milch bis zur Differenz aus dem maximalen Fassungsvermögen und dem aktuellen Inhalt zugeschüttet bekommen.

Als Methoden kommen hier *Füllen*, *Leeren*, *Eingießen*, *Ausgießen* in Betracht. Wir notieren dies durch:

Variablen MAX: integer, (MAX > 0)
INH : integer, (0 ≤ INH), (INH ≤ MAX), anfangs ist INH = 0

Methoden

Füllen: Milchtank (MAX - INH) Liter entnehmen;
Leeren: An Milchtank INH-Liter schicken;
Eingießen: Umschütten von (MAX - INH) Litern aus Behälter A;
Ausgießen von X Litern:
falls X < INH dann gib X Liter ab, vermindere INH um X;
sonst gib INH Liter ab, setze INH = 0;

Zur sprachlichen Präzisierung der Definition verwenden wir die schon eingeführten prozeduralen Anweisungen und bringen die Zuordnung von Variablen und Methoden zu den Objekten wie bei den Listenstrukturen durch Eimer.MAX oder Gefäß.Füllen zum Ausdruck.
Damit kann die Objektklasse Behälter wie folgt definiert werden:

Definition Objektklasse Behälter:

Var MAX, INH: integer
A:Behälter
with(MAX > 0),(0 ≤ INH) (INH ≤ MAX)
initial: INH := 0

Methods

Füllen: begin Milchtank.Laden(MAX - INH); INH:= MAX end;
Leeren: begin Milchtank.Speichern(INH); INH:= 0 end;
Eingießen: INH:= INH + A.Eingießen ;
Ausgießen (X:integer): integer; begin if X < INH then INH := INH - X
else X:= INH;
INH := 0; return X
end;

end Objektklasse Behälter;

Konkrete Behälter erzeugt man durch den Befehl *new*, indem man den Behältern Namen gibt und einige oder alle Variablen mit Werten belegt:
Eimer := new Behälter (MAX = 10, A = Gefäß);
Gefäß:= new Behälter (MAX = 7, A = Eimer);
Da der Tank, im Unterschied zu Eimer und Gefäß passiv bleibt, definieren wir eine neue Objektklasse Tank:

Definition Objektklasse Tank;

Methods

Laden (i: integer) : NIL;
Speichern (i: integer) : NIL;

end Objektklasse Tank;

Milchtank := new Tank;

Zur Steuerung des Verkaufs-Prozesses wird ein Steuerobjekt Verkäufer eingeführt, der die Methodenanwendung der Objekte überwacht bzw. veranlaßt.
Dazu führen wir die Objektklasse *Steuerung* ein.

Definition Objektklasse Steuerung;

```
 Var Soll: integer;
       Bgroß, Bklein: Behälter;
        with (0 ≤ Soll), (Soll ≤Bklein.MAX);
 Methods
   Bediene: if Soll = Bklein.MAX then Bklein.Füllen
                                 else while Bklein.INH ≠ Soll do
                                       if Bklein.INH = Bklein.MAX
                                          then Bklein.Leeren
                                          else if Bgroß.INH = 0
                                               then Bgroß.Füllen
                                               else Bklein.Eingießen ;
```

end Objektklasse Steuerung;

Im Beispiel setzen wir

Verkäufer := new Steuerung (Soll=5, Bgroß = Eimer, Bklein = Gefäß);

Der Auftrag Verkäufer.Bediene veranlaßt den Verkäufer seine Methode *Bediene* auszuführen. Dadurch kommt im Beispiel folgende Auftragsfolge zustande:

	Inhalt nach Ausführung	
Auftrag	*Eimer*	*Gefäß*
Eimer.Füllen, Milchtank.Laden	10	0
Gefäß.Eingießen, Eimer.Ausgießen	3	7
Gefäß.Leeren, Milchtank.Speichern	3	0
Gefäß.Eingießen, Eimer.Ausgießen	0	3
Eimer.Füllen, Milchtank.Laden	10	3
Gefäß.Eingießen, Eimer.Ausgießen	6	7
Gefäß.Leeren, Milchtank.Speichern	6	0
Gefäß.Eingießen, Eimer.Ausgießen	0	6
Eimer.Füllen, Milchtank.Laden	10	6
Gefäß.Eingießen, Eimer.Ausgießen	9	7
Gefäß.Leeren, Milchtank.Speichern	9	0
Gefäß.Eingießen, Eimer.Ausgießen	2	7
Gefäß.Leeren, Milchtank.Speichern	2	0
Gefäß.Eingießen, Eimer.Ausgießen	0	2
Eimer.Füllen, Milchtank.Laden	10	2
Gefäß.Eingießen, Eimer.Ausgießen	5	7
Gefäß.Leeren, Milchtank.Speichern	5	0
Gefäß.Eingießen, Eimer.Ausgießen	0	5

Nicht jedes Paradigma ist für jedes Problem gleich gut geeignet. Deshalb hängt die Auswahl der Beschreibungssprache (Programmiersprache) auch von der gegebenen Problemklasse ab. Für die Effizienz und Akzeptanz der entstehenden Programme ist deren Überprüfbarkeit und Übersichtlichkeit wesentlich, die von den verschiedenen Paradigmen unterschiedlich unterstützt wird.

3. Spezifikation und Verifikation von Programmen

Im Zusammenhang mit der Konstruktion von Programmen, stellt sich die Frage: „Was leistet das Programm?“ oder „Erfüllt das Programm die gestellten Erwartungen?“ oder „Wie ist ein Programm mit gewünschter Wirkung zu konstruieren?“.
Die erste Fragestellung betrifft die Beschreibung dessen, was durch das Programm bewirkt wird, d.h. eine Definition der Wirkungsweise von Bestandteilen aus denen das Programm besteht. Man spricht dann von der *Semantik* der Programmnotation.
Die zweite Frage betrifft den Nachweis von Eigenschaften, die das Programm erfüllen soll. Das Programm soll bezüglich gegebener Bedingungen (Eingangs- und Ausgangsbedingungen) *verifiziert* werden.
Die dritte Frage betrifft die Konstruktionsmöglichkeiten zur Erzeugung *korrekter* Programme zu gegebenen Anforderungen. Dabei beziehen sich die Anforderungen auf die Erfüllung von Eingangs- und Ausgangsbedingungen des Programms und seiner Teile. Man unterscheidet dabei eine *partielle* Korrektheit, bei der das Programm nichts Falsches produziert, und die *vollständige* Korrektheit, bei der das Programm bei erfüllter Eingangsbedingung immer Richtiges produziert, d.h. die Ausgangsbedingung wird immer erfüllt.
Diese drei Fragen hängen eng zusammen und wir diskutieren nachfolgend Ansätze zu deren Beantwortung.

Vor- und Nachbedingung

Von jeder Programmausführung erwarten wir ein bestimmtes Ergebnis, welches als ***Zusicherung*** (*postcondition**)*** formuliert wird.
Beispiel: von der Berechnung des Doppelten einer Zahl x; Doppelt(x) erwartet man das Zweifache der Zahl x als Ergebnis. Die Zusicherung könnte also lauten:

$$\text{Doppelt}\ (\ x\ ) = 2 * x$$

Dabei werden bestimmte Voraussetzungen über den Eingangswert x gemacht, die als ***Vorbedingungen*** (*precondition*) formuliert werden.

Mögliche Vorbedingungen in unserem Beispiel wären:

$$x \text{ ist INTEGER; } | x | \leq \text{MAX INTEGER}/2.$$

Die zweite Bedingung dient dabei der Absicherung, daß der Integer-Bereich nicht überschritten werden kann, da die Operation sonst kein Ergebnis liefert. Man wird nach Möglichkeit eine solche Vorbedingung anstreben, die möglichst keine Einschränkungen der Eingangswerte eines Programms erzwingt. Deshalb suchen wir die ***schwächste Vorbedingung*** (*weakest precondition* oder *wp*), um die Leistung des Programms voll "auszuschöpfen", d.h. auf möglichst viele Eingangswerte anwendbar zu machen.

Ein Programm läßt sich bezüglich seines Effekts also beschreiben durch:
- die Zusicherung (Q) für das Ergebnis
- die Vorbedingung (P) an die Eingabe.

Diese Angaben spielen bei der Verifikation eine zentrale Rolle.

Notation:
Seien S ein Programm oder ein Programmteil, P (precondition) eine Aussage über die Parameter des Programms, die vor dessen Ausführung gilt, und Q (post-condition) eine Aussage, die nach Ausführung von S gilt. Dann beschreiben wir diesen Zusammenhang durch:

P { S } Q

Die so festgelegte Notation kann verschiedene Bedeutungen besitzen.

a) Definition (**Def**) einer einzelnen Anweisung **S**	Durch Angabe der Wirkung der Anweisungen, definieren wir die *Semantik* der Sprache.
b) Beschreibung eines Programms oder Programmteils **S** (**Bes**)	Aus der Definition der einzelnen Befehle werden die Aussagen P und Q *synthetisiert*.
c) Spezifikation (Anforderung) des Programmes **S** (**Spec**)	Die Anforderungen an ein noch zu schaffendes Programm werden durch P und Q *spezifiziert*.

Im allgemeinen ist aus dem Kontext ersichtlich, welche Bedeutung verwendet wird.

Den Nachweis, daß Spezifikation (was soll das Programm tun) und Beschreibung (was tut das Programm) verträglich sind, bezeichnet man als *Verifikation* des Programms. Dabei ist zu zeigen:

$$(P_{Spec} \Rightarrow P_{Bes}) \wedge (Q_{Bes} \Rightarrow Q_{Spec})$$

d.h., die spezifizierte Vorbedingung P_{Spez} impliziert die realisierte Vorbedingung P_{Bes} des Programms und die realisierte Nachbedingung Q_{Bes} impliziert die spezifizierte Nachbedingung Q_{Spez}.
In der Regel ist Q_{Spec} durch die Aufgabenstellung vorgegeben (gesuchtes Ergebnis), davon ausgehend wird Q_{Bes} aus Q_{Spec} verallgemeinert, d.h. Q_{Bes} ist eine allgemeinere Aussage, in der Q_{Spec} enthalten ist.
Die gesuchte Vorbedingung P_{Bes} sollte so schwach wie möglich sein:

$P_{Bes} = wp(S, Q_{Bes})$ (Schwächste Vorbedingung von S zur Erfüllung von Q_{Bes}).

Bei der Erstellung einer Programmbeschreibung gehen wir also rückwärts vor, von der angestrebten Nachbedingung zur unvermeidlichen Vorbedingung. Ein Programm ist um so robuster oder flexibler, um so weniger Vorbedingungen gestellt sind. (Idealerweise: Vorbedingung TRUE - immer erfüllt, oder: Vorbedingung FALSE - Programm liefert unter keinen Umständen das richtige Ergebnis.)

Beispiel:
In einem Feld F aus Integer-Werten mit der Indizierung 1 ... 1000, in dem nur die Werte 0 bis 20 auftreten, sollen alle Stellen gefunden werden, die mit einem bestimmten Wert a besetzt sind.

Die Nachbedingung für das Ergebnis i folgt aus der Aufgabenstellung:

$Q_{Spec} \equiv 1 \le i \le 1000 \wedge F[i] = a.$

Zur Realisierung empfiehlt es sich, immer das erste Vorkommen von a zu suchen.

$Q_{Bes} \equiv Q_{Spec} \wedge (F[j] = a \Rightarrow j \ge i)$

Q_{Spec} ist in Q_{Bes} enthalten; Q_{Bes} impliziert Q_{Spec}.

Die Vorbedingung muß sicherstellen, daß a überhaupt in F vorkommt:

d.h., $P_{Bes} \equiv$ es existiert ein i mit: $1 \le i \le 1000 \wedge F[i] = a$.

Durch die Problemstellung ist vorgegeben:

$P_{Spec} \equiv$ es existiert ein $1 \le i \le 1000$ mit: $F[i] = a \wedge$ für alle j: $F[j] \in \{0... 20\}$

d.h., P_{Spec} impliziert P_{Bes}.

Semantik der Anweisungen

(Angabe der schwächsten Vorbedingung WP zu vorgegebener Nachbedingung Q.)

Anweisung	**Semantik**	**Erklärung**
leere Anweisung	wp (leer, Q) ≡ Q	Was nach der Ausführung der leeren Anweisung gelten soll, muß schon vorher gelten.
Wertzuweisung	wp (x:= E, Q(x)) ≡ Q(E) Vorauss.: E ist definiert und E liefert Element vom Typ(x)	E muß definiert und typkompatibel zu x sein. Die Vorbedingung ergibt sich aus der Nachbedingung, durch Ersetzung aller Vorkommen von x durch den Ausdruck E. E kann x enthalten (keine Rekursion)
Sequenz	wp (S1; S2, Q) ≡ wp (S1, wp (S2, Q))	Die schwächste Vorbedingung der zweiten Anweisung S2 ist die Nachbedingung der Anweisung S1.
Alternative	wp (if B then S1 else S2, Q) ≡(B ∧ wp (S1, Q)) ∨ (nicht B ∧ wp (S2, Q))	Wenn die Bedingung B erfüllt ist, dann hat die Alternative die schwächste Vorbedingung des then-Zweiges S_1, sonst die des else-Zweiges S_2.
Bedingte Anweisung	wp (if B then S , Q) ≡(B ∧ wp (S, Q) ∨ (nicht B ∧ Q)	Wenn die Bedingung B erfüllt ist, hat die bedingte Anweisung die schwächste Vorbedingung der Anweisung S, andernfalls galt die Nachbedingung schon vorher.

Beispiel:

Gegeben sei die folgende Anweisung S:

IF x < 0 THEN x := -x ELSE x := x - 1 ;

mit x Element Integer und der Nachbedingung $Q \equiv x \geq 0$.
Dann ergibt sich die schwächste Vorbedingung von S bezüglich der Nachbedingung Q, wie vorher eingeführt zu:

$$\begin{aligned} wp(S, Q) &\equiv ((x < 0) \wedge wp(x:= -x, x \geq 0)) \vee ((x \geq 0) \wedge wp(x:= x-1, x \geq 0)) \\ &\equiv ((x < 0) \wedge (-x \geq 0)) \quad \vee \quad ((x = 0) \wedge (x-1 \geq 0)) \\ &\equiv ((x < 0) \wedge (x \leq 0)) \quad \vee \quad ((x \geq 0) \wedge (x > 0)) \\ &\equiv (x < 0) \quad \vee \quad (x > 0) \\ &\equiv \mathbf{x \neq 0} \end{aligned}$$

Die schwächste Vorbedingung, um nach Ausführung der Anweisung die Nachbedingung zu erfüllen, ist also $x \neq 0$. Eine hinreichende Vorbedingung wäre also $x > 10$, da sie $x \neq 0$ impliziert. Die Bedingung $x<10$ wäre dagegen nicht hinreichend.

Zur Beschreibung der Wirkung von Wiederholanweisungen geben wir zunächst ein Beispiel.

Gegeben sei das folgende Programmfragment S:

```
begin
  while i ≤ n do
    begin
      t := t * x;
      i := i + 1;
    end;
end;
```

Damit wird die n-te Potenz von x berechnet. Die Nachbedingung lautet also $Q(t, i) \equiv t = x^n$. Wir suchen nun die schwächste Vorbedingung $wp_j(S,Q(t,i))$ von S in Abhängigkeit von der Anzahl j der Schleifendurchläufe.
Den Schleifenrumpf t := t * x; i:= i + 1 bezeichnen wir durch T.

kein Durchlauf	$wp_0 (S, Q (t, i))$	$\equiv i > n \wedge Q (t, i)$ (Wegen i>n wird T niemals durchlaufen)
ein Durchlauf	$wp_1 (S, Q (t, i))$	$\equiv i \leq n \wedge wp (T, i > n \wedge Q (t, i))$
zwei Durchläufe	$wp_2 (S, Q (t, i))$	$\equiv i \leq n \wedge wp (T, i \leq n \wedge wp (T, i > n \wedge Q (t, i)))$

Die Bedingung $i \leq n$ garantiert, daß die Schleife T durchlaufen wird. Die Abbruchbedingung ($i > n$) muß nach der letzten Ausführung der Schleife erfüllt sein und steht deshalb in der entsprechenden Nachbedingung.
Da die Anzahl der Durchläufe beliebig sein kann, gilt allgemein:

$$wp\ (S, Q\ (t, i\ )) \equiv wp_0\ (S; Q) \vee wp_1\ (S; Q) \vee ...$$

Für den Schleifenrumpf kann man für die Nachbedingung $Q(t,i) \equiv i \leq n \wedge t = x^i$ aus der Definition der einfachen Anweisung bzw. der Sequenz die schwächste Vorbedingung

$$\begin{aligned} wp(T,Q(t,i)) &= wp\ (t:= t * x, wp(i:=i+1, (i \leq n \wedge t=x^i))) \\ &= wp\ (t:= t * x, (i+1 \leq n \wedge t=x^{i+1})) \\ &= (i+1 \leq n \ \wedge\ t * x = x^{i+1}) \\ &= (\ \ i < n \ \wedge \ \ \ t = x^i) \end{aligned}$$

erhalten.
Anmerkung: $t = x^i$ bleibt durch t unverändert (invariant).

Das Einsetzen von T ergibt:

$$\begin{aligned} wp_1\ (S, Q\ (t, i)) &\equiv i \leq n \wedge wp\ (\ t := t * x\ ;\ i := i + 1, \\ &\qquad\qquad i > n \ \wedge\ Q\ (t, i)) \\ &\equiv i \leq n \wedge\ i + 1 > n \ \wedge\ Q\ (\ t * x, i + 1\ ) \\ &\equiv i = n \wedge\ Q\ (\ t * x, i + 1\ ) \\ wp_2(S, Q\ (\ t, i\ )) &\equiv i \leq n \wedge wp\ (t:=t * x;\ i:= i + 1, i \leq n \\ &\qquad \wedge wp\ (t:=t * x;\ i:= i + 1, i > n \\ &\qquad \wedge Q\ (\ t\ ,\ i\ ))) \\ &\equiv i \leq n \wedge wp\ (t:=t * x;\ i:= i + 1, i = n \\ &\qquad \wedge Q\ (\ t * x, i + 1\ )) \\ &\equiv i \leq n \wedge i + 1 = n \wedge Q\ (\ t * x^{\,2}\ , i + 2\ ) \\ &\equiv i+1 = n \wedge Q\ (\ t * x^2\ , i + 2\ ) \end{aligned}$$

Offensichtlich lautet die allgemeine Form für j Durchläufe:
$wp_j(S, Q\ (\ t, i\ )) \equiv (i + j - 1 = n \ \wedge\ \ Q(t * x^j, i + j))$
Insgesamt entsteht damit für wp(S, Q(t,i)) durch Alternative aller wp_j:

$$\begin{aligned} wp\ (S, Q(t, i)) &\equiv\ wp_0\ (S, Q(t,i)) \vee wp_1(S, Q(t,i)) \vee wp_2(S, Q(t,i)) \vee ... \\ &\equiv\ (i > n \ \wedge\ Q(\ t, i\ )) \vee \text{ es existiert ein } j \in N: (i + j - 1 = n \ \wedge\ \ Q(t * x^j, i + j)) \end{aligned}$$

Damit haben wir eine endliche Beschreibung für das gesamte Programmfragment S erreicht.
Setzen wir nun die Nachbedingung $Q(\ t, i\ ) \equiv (t = x^n)$ ein, so erhalten wir:

$$\begin{aligned} wp\ (S, Q(\ t, i\ )) &\equiv (i > n \ \wedge\ t = x^n) \ \vee\ (\text{existiert } j \in N: (\ i+j-1=n \ \wedge\ t*x^j = x^n)) \\ &\equiv (i > n \ \wedge\ t = x^n) \ \vee\ (n + 1 - i > 0 \ \wedge\ t * x^{(n+1-i)} = x^n) \\ &\equiv (i > n \ \wedge\ t = x^n) \ \vee\ (i \leq n \wedge (\ t = x^{i-1} \ \ \vee\ x = 0)) \end{aligned}$$

Dies ist die schwächste Vorbedingung unserer while-Schleife. Da sie aber beim Einbau in ein Programm einen zu großen Aufwand erfordern würde, suchen wir nun eine stärkere, aber einfachere Vorbedingung.
Das erste Alternativglied ($t = x^n$ bzw. $t*x^j = x^n$) ist in der Regel nicht generell zu erfüllen, da das Ergebnis (x^n) bekannt sein muß.
Eine mögliche stärkere Vorbedingung ist:

$$P \equiv (i = 1 \wedge t = 1 \wedge n \geq 0)$$

Es ist leicht nachzuweisen, daß diese Bedingung die schwächste Vorbedingung wp(S, Q(t,i)) impliziert.

Um P zu erfüllen erweitern wir unser Programmfragment zu S'

```
i := 1;  t := 1;
while i ≤ n do
  begin
    i := i + 1;
    t := t * x;
  end;
```

Damit lautet die schwächste Vorbedingung ($n \geq 0$) .
Um auch diese Bedingung noch zu erfüllen, wäre eine Eingangsprüfung für n erforderlich. Das Programmfragment würde dann lauten

```
 if n ≥ 0 then (S')
          else  (Fehlermeldung);
```

Danach ist die Vorbedingung immer erfüllt (TRUE).

Um immer ein richtiges Ergebnis zu garantieren, müssen wir die *totale Korrektheit* nachweisen.
Die Schleife terminiert, wenn die Schleife überhaupt nicht oder nur endlich oft ausgeführt wird und wenn ein Schleifendurchlauf nur endlich viele Schritte umfaßt.
Für n = 0 wird die Schleife nicht, für n > 0 n-mal durchlaufen. Die Schleife umfaßt zwei Operationen, also terminiert unser Programmfragment.

***Beachte**: Die totale Korrektheit folgt nicht notwendig aus der partiellen Korrektheit.*

Beschreibung einer Schleife mittels Invariante

In diesem Abschnitt wird ein anderer Ansatz für die Verifikation eines iterativen Programmes benutzt, der vielfach leichter anwendbar ist.
Dazu betrachten wir das Programmfragment

while B do S end.

Festlegungen:
V: sei die Gesamtheit aller Variablen, die von der Schleife verändert werden.
B: die von V abhängige Wiederholbedingung der Schleife.
h: die durch den Schleifenrumpf S realisierte Veränderung h(V) der Variablenwerte.
$h^j(V_0)$: j-malige Anwendung von h auf V_0 . $V_j = h^j(V_0)$ bezeichnet die dabei entstehenden Variablenwerte.

Eine **Invariante** INV ist eine Aussage über V, die gültig ist, solange die Wiederholbedingung geprüft wird. Es gilt also:

$$INV\ (V_0) = TRUE$$

Außerdem muß gelten: $B(V_0) \wedge ... \wedge B(V_m) \Rightarrow$
$INV\ (h(V_m)) = INV\ (V_{m+1}) = TRUE$

Insgesamt läßt sich die Invariante INV charakterisieren durch:

$$INV\ (V_0) \wedge B\ (V_0) \wedge B\ (V_1) \wedge ... \wedge B\ (V_m) \Rightarrow INV\ (h\ (V_m)).$$

Die Invariante beschreibt also einen Zusammenhang der Variablen, der erhalten bleibt, solange die Schleife durchlaufen wird, beim Abbruch der Iteration ist $B\ (V_j)$ nicht mehr erfüllt, d.h., dann gilt:

$$INV\ (V_j) \wedge \text{nicht } B\ (V_j).$$

Bei richtiger Wahl der Invariante läßt sich daraus die Nachbedingung Q ableiten:

$$(INV(V_j) \wedge \text{ nicht } B(V_j)) \Rightarrow Q(V_j)$$

Zusammenfassung:
Wie wir sehen muß eine Invariante folgende drei Bedingungen erfüllen:
- Sie muß am Schleifenanfang gelten.
- Sie muß, sofern B erfüllt war, unter Anwendung von h tatsächlich erhalten bleiben.
- Sie muß hinreichend aussagekräftig sein, um nach Abbruch der Schleife den Schluß auf die Nachbedingung Q zu gestatten.

Beispiel:
Betrachten wir das Programmfragment zur Berechnung von x^n. Die Invariante ist:

$$INV\ (i, t) \quad \equiv \quad i \leq n+1 \ \wedge \ t = x^{i-1}.$$

Der Beweis der Richtigkeit erfolgt über vollständige Induktion:

Induktionsanfang: Für n = 0 ist nach Voraussetzung i = 1 und t = 1, damit ist

$$INV\ (i_0, t_0) \quad \equiv \ (i_0 \leq n+1 \ \wedge \ t_0 = x^{i0-1}) \equiv (1 \ \leq 0 + 1 \wedge 1 = x^0)$$

eine wahre Aussage.

Seien die Invariante und die Wiederholbedingung gültig für i_m und t_m (Werte von i und t nach dem m-ten Schleifendurchlauf). Dann gilt:

$$INV\ (i_m, t_m) \wedge B\ (i_m) \quad \equiv \ (i_m \leq n+1 \wedge \ \ t_m = x_m{}^{im-1} \ \wedge \ i_m \leq n).$$

Wegen h(t, i) = [t ∗ x, i + 1] gilt weiter:

$$t_{m+1} = t_m * x = x_m{}^{im-1} * x = x_m{}^{im} \ \wedge \ i_{m+1} = i_m + 1.$$

Also gilt nach Übergang auf i_{m+1} auch:

$$(t_{m+1} = x_{m+1}{}^{im+1 \ - 1} \ \wedge \ (i_{m+1} \leq n+1)) \equiv INV\ (i_{m+1}, t_{m+1}) \qquad \text{q.e.d.}$$

Eine hinreichende Vorbedingung dafür, daß die Invariante am Anfang erfüllt ist, lautet

$$P(i, t, n) \equiv [i=1 \ \wedge \ n \geq 0 \ \wedge \ t=1] \ \Rightarrow INV\ (i, t).$$

Mit der Wiederholbedingung B(i) ≡ i ≤ n folgt als Nachbedingung:

$$INV\ (V) \wedge \text{nicht } B(V) \equiv \ (i \leq n+1 \ \wedge \ t = x^{i-1} \ \wedge \ i > n)$$
$$\equiv \ (i = n+1 \ \wedge \ t = x^{i-1} \ \Rightarrow t = x^n) \equiv \ Q\ (t, i)$$

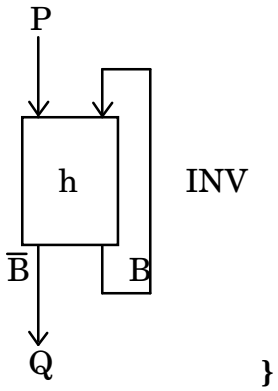
Mit Hilfe der so gewonnenen Invarianten können Programme konstruiert werden, die in jedem Fall das richtige Ergebnis liefern (totale Korrektheit).

Konstruktion iterativer Programme

Bei der Konstruktion korrekter iterativer Programme und Schleifen ist wie folgt vorzugehen:

1.) Festlegung der *Nachbedingung* Q (Ziel der Konstruktion).

2.) "Finden" einer geeigneten *Invariante* INV.
(Das Problem wird in Teilprobleme zerlegt, die alle die Invariante erfüllen, wobei h der Funktionskörper ist.)



3.) Festlegung der *Vorbedingung* P und Ableitung der notwendigen *Initialisierung* mit dem *Nachweis*, daß P die Invariante impliziert.
(Eingangsprüfung oder Anforderungen an vorangehenden Programmteil.)

$P \Rightarrow INV$

4.) Festlegung der *Wiederholbedingung* B.

B

5.) Nachweis, daß die in 2.) gefundene Bedingung tatsächlich eine Invariante ist, d.h., bei Wiederholung und nach Ende der Schleife gilt.

$INV \wedge B \Rightarrow INV \cdot h$

6.) Nachweis, daß bei nicht mehr gültiger Wiederholbedingung die Nachbedingung Q gilt.

$INV \wedge \overline{B} \Rightarrow Q$

Ist Schritt 6.) nicht erfolgreich, so muß der Vorgang ab Schritt 2.) wiederholt werden.

Beispiel:

Iteratives Programm zur Berechnung der Fakultät

1.) $Q \equiv y = n\,!$

2.) $INV \equiv y = i\,! \ \wedge \ 0 \leq i \leq n.$
Aus der Fakultätseigenschaft $(i+1)! = i! * (i+1)$ erhalten wir für die Implementierung des Schleifenrumpfes
*S = (i := i + 1; y := y * i;)*
und bezüglich Q als schwächste Vorbedingung
$wp(S, Q) \equiv wp(i:=i+1, wp(y:= y * i, Q(i, y)))$
$\equiv wp(i:=i+1, Q(i, y * i))$
$\equiv Q(i+1, y * (i+1))$.
Daraus folgt die Transformation $h(i, y) = [i+1, y * (i+1)]$, die zusammen mit der Wiederholbedingung (4.) die Invariante garantiert.

3.) Aus der Anfangsbedingung 0! = 1 erhalten wir die Vorbedingung
$P \equiv (y = 1 \wedge i = 0 \wedge n \geq 0)$
Die Invariante folgt aus P, da $(y = 1 = i! \wedge 0 = i \leq n)$ gilt.

4.) Als Wiederholbedingung mit dem Endwert n ergibt sich
$B \equiv (i < n)$.

5.) Es gilt $INV (V) \wedge B (V) \Rightarrow INV (h(V))$, denn aus
$(y = i! \wedge 0 \leq i \leq n \wedge i < n)$ folgt
$(y * (i+1) = h(y) = i! * (i+1) = h(i)! \wedge i+1 = h(i) \leq n)$.

6.) Abschließend gilt:
$INV (V) \wedge$ nicht $B (V) \Rightarrow Q (V)$, denn aus $(y = i! \wedge 0 \leq i \leq n \wedge i \geq n)$
folgt $(y = i! \wedge i = n)$ und daraus weiter $(y = n!)$.

Damit ist das Programm erfolgreich konstruiert.

Als Implementierung in PASCAL ähnlicher Notation erhalten wir:

```
if n ≥ 0 then
  y := 1; i:=0;
  while i < n do
     begin
       i  := i +1;
       y := y * i;
     end;
   else (Fehlermeldung);
```

In diesem Fall empfiehlt sich die Anwendung einer for-Schleife mit Inkrementierung (Erhöhung) von i am Schleifenende (y ist mit i+1 zu multiplizieren).

```
y := 1;
for j:=1 to n do y := y * (i+1);
```

Die vollständige Funktion zur Berechnung der Fakultät könnte dann wie folgt aussehen:

```
function fakultaet (Argument: integer): integer ;
  var Produkt: integer ;
        Index: integer ;
  begin
    Produkt := 1;
    for Index := 1 to Argument do
      begin
       Produkt := Produkt * Index ;
      end ;
       Fakultaet := Produkt ;
end ;
```

Zusammenfassung

Eine **Spezifikation** präzisiert die Anforderungen an ein Programm. In der **Beschreibung** eines Programms wird festgelegt, wie die spezifizierte Anforderung durch Anwendung von Anweisungen der definierten Programmiersprache erfüllt wird. Durch die **Verifikation** wird nachgewiesen, das Beschreibung und Spezifikation verträglich sind, d.h., die Programmbeschreibung erfüllt die spezifizierten Anforderungen.
In Abhängigkeit von der Interpretationssicht, kann die Aussage

$$P \{S\} Q,$$

wo P,Q Bedingungen (Vor-, Nachbedingung) und S ein Programm, zur Definition der Semantik, zur Beschreibung der Programmeigenschaften, zur Spezifikation oder zur Verifikation des Programms dienen.
Endziel ist immer die Erzeugung korrekter Software, die das tut was sie tun soll.
S heißt ***partiell korrekt*** bzgl. (P, Q), falls unter der Bedingung, daß die Eingangswerte P erfüllen und das Programm S terminiert, die Ausgangswerte nach Programmausführung die Bedingung Q erfüllen. (Es entsteht kein falsches Ergebnis.)
S heißt ***total korrekt*** bzgl. (P, Q), falls unter der Bedingung, daß die Eingangswerte P erfüllen, das Programm S terminiert und die Ausgangswerte die Bedingungen Q erfüllen. (Es entsteht immer ein richtiges Ergebnis.)

III. Algorithmen und Datenstrukturen

1. Effizienz und Komplexität

Zur Lösung eines Problems existieren im allgemeinen verschiedene Verfahren. Deshalb ist nicht nur entscheidend, daß ein Programm eine Aufgabe erfüllt, vielmehr ist auch von Interesse wie und mit welchem Aufwand (Speicherverbrauch und Rechenzeit) dies geschieht. Daher führen wir zur Bewertung von Funktionen und Algorithmen die Begriffe Komplexität und Effizienz ein.
Bewertungskriterien für die Effizienz von Algorithmen sind die jeweils in Anspruch genommenen *Ressourcen* (*Zeit* und *Speicher*). Die Komplexität einer Funktion hingegen wird durch den Berechnungsaufwand (die *Zahl notwendiger Operationen)* in Abhängigkeit vom Umfang des zu lösenden Problems beschrieben. Die Komplexität stellt dabei die untere Schranke für die Effizienz der die Funktion beschreibenden Algorithmen dar. Die Effizienz ist die obere Schranke für die Komplexität der durch den Algorithmus definierten Funktione. Folgende Tabelle zeigt dies im Überblick:

Effizienz von Algorithmen	*Komlexität von Funktionen*
Die Effizienz eines Algorithmus ist eine obere Schranke für die Komplexität der Funktion, die durch den Algorithmus definiert wird.	Die Komplexität einer Funktion ist eine untere Schranke für die Effizienz von Algorithmen, die diese Funktion beschreiben.
Praktisches Problem	Theoretisches Problem
Auswahl effektiver Algorithmen	Schranken für Funktionen

Vergleicht man verschiedene Algorithmen, so möchte man eine möglichst allgemeine vom Rechner und den Eingabedaten unabhängige Aussage treffen. Man unterscheidet daher verschiedene *Methoden der Effizienzbeurteilung.*

1.)	Rechner und Daten konstant	Vergleich über *Zeitmessung* (z.B. interne Uhr)
2.)	Rechner konstant, Daten beliebig	Vergleich durch *Zählung* der notwendigen CPU-Operationen
3.)	Rechner und Daten beliebig	*Berechnung* der notwendigen Operationen in Abhängigkeit konkreter Parameter (z.B. Anzahl der Elemente)

Wenn der Datenumfang beliebig groß sein kann, sind insbesondere *asymptotische* Abschätzungen sinnvoll.

Beispiel:
Seien V_i die Anzahl der Vergleiche und S_i die Anzahl der Wertzuweisungen des Algorithmus A_i , der eine Datenstruktur der Größe n (natürliche Zahl) bearbeitet. Für zwei Algorithmen A_1 und A_2 gelte:
$V_1 = S_1 = n \log(n)$ $\quad\quad$ $V_2 = n^2$ $S_2 = n$

Frage: Welcher Algorithmus ist für $n \to \infty$ schneller ?

Im konkreten Fall hängt die Beantwortung der Frage von der Ausführungszeit für Vergleiche und Wertzuweisungen sowie vom Umfang der Datenmenge ab.
Unabhängig davon kann schon eine allgemeine Aussage getroffen werden:
Übersteigt n einen gewissen Wert, so ist Algorithmus A_1 schneller, denn sein Aufwand wächst proportional $n \cdot \log(n)$, der von A_2 jedoch proportional n^2.

Es sei Σ eine Zeichenmenge (Alphabet), Σ^* die Menge der darüber bildbaren Zeichenketten (Wörter) und w (bzw. v) ein Wort aus Σ^* (Eingabe- bzw. Ausgabewort).
Der Algorithmus A möge eine Funktion f bestimmen, die von Σ^* nach Σ^* abbildet ($f: \Sigma^* \to \Sigma^*$).

Wir verwenden weiter folgende Bezeichnungen:

s (w)	Schrittzahl für Transformation von w in v Voraussetzung: A ist auf w anwendbar, hält bei w und liefert v.
$\|w\| = n$	Länge von w (Anzahl der Zeichen in w)
$s_A(n)$	Vom Algorithmus A benötigte maximale Schrittzahl zur Vearbeitung eines Wortes der Länge n (Effizienzmaß von A) $s_A(n) = \text{Max}\{ s(w) \mid w \in \Sigma^*, \|w\| = n\}$
p (w)	Anzahl der Speicherplätze, die der Algorithmus A zur Verarbeitung der Eingabe w benötigt.
$P_A(n)$	Maximale Speicherbelastung von A für Wörter der Länge n $p_A(n) = \text{Max} \{ p(w) \mid w \in \Sigma^* , \|w\| = n \}$

Bei der Aufwandsabschätzung sind nur die Größenverhältnisse für den Aufwand von Interesse (z.B.: lineares-, polynomiales- oder exponentielles Wachstum).
Zur Beschreibung von Komplexitätsmaßen benutzen wir die Landau-Notation:
f und g seien Funktionen über N (f,g: $N \to N$).
Nachfolgend definieren wir Ordnungs-Eigenschaften dieser Funktionen hinsichtlich ihrer Komplexität (Wachstum).

1.)	groß O	$g \in O(f)$	g wächst nicht schneller als f. $\exists c>0\ \exists n_0 \in N\ (\forall n \geq n_0 : g(n) \leq c \cdot f(n))$, d.h., es läßt sich eine Zahl c größer Null und eine nat. Zahl n_0 finden, so daß für alle n, die größer als n_0 sind, der Wert von g an der Stelle n kleiner gleich dem c-fachen des Wertes von f an der Stelle n ist. (Der Quotient $\frac{g(n)}{f(n)}$ bleibt beschränkt.)
2.)	klein o	$g \in o(f)$	g wächst *langsamer* als f $\forall c>0\ \exists n_0 \in N\ (\forall n \geq n_0 : g(n) \leq c \cdot f(n))$; Im Gegensatz zu 1.) muß hier für <u>jede</u> frei wählbare Konstante c größer Null, von einer Zahl n_0 ab der Funktionswert von g kleiner gleich dem c-fachen des Wertes von f an dieser Stelle sein. (Der Quotient $\frac{g(n)}{f(n)}$ geht gegen Null.)
3.)	groß Ω	$g \in \Omega(f)$	genau dann, wenn $f \in O(g)$, d.h., g wächst mindestens so schnell wie f;
4.)	klein ω	$g \in \omega(f)$	genau dann, wenn $f \in o(g)$, d.h., g wächst schneller als f;
5.)	Deta Θ	$g \in \Theta(f)$	genau dann, wenn $f \in O(g)$ und $g \in O(f)$ gilt, d.h. f und g wachsen gleich schnell.

Funktionsbeispiele:

x > 0 ganz
g(x) = 3x+7
$g \in O(x)$, denn $3x+7 \leq c \cdot x$ für $\forall x \geq x_0$,
mit c=5 und x_0=4.

$f(x) = x^2$
$g \in o(f)$, denn $3x + 7 \leq c \cdot x^2$ für $c > 0$ und $x > 5/c$.

$h(x) = 2^x$
$f \in o(h)$, denn $x^2 \leq c \cdot 2^x$ für c>0 und $x > 1/c$.

Programmbeispiele:
Vergleichen wir nun eine rekursive und eine iterative Lösung zur Berechnung der Fibonacci-Zahlen bezüglich ihres Aufwandes: (Zahl der Funktionsaufrufe bzw. Wertzuordnung)

rekursiv

```
function fibr(n:integer): integer;
 begin
    if n ≤ 2 then return 1
    else  return (fibr(n-1)+fibr(n-2))
  end;
```

Aufwand:
$fibr(n) \in \Omega(2^n)$
exponentieller Aufwand
c_n sei die Anzahl der Funktionsaufrufe bei Berechnung von fibr(n).
$c_1 = c_2 = 1$
$c_n = 1 + c_{n-1} + c_{n-2}$ für $n > 2$
$c_{n-1} = 1 + c_{n-2} + c_{n-3}$ für $n > 3$

$c_n = 2 + 2\ c_{n-2} + cn\text{-}3 > 2c_{n-2} > 4c_{n-4}$
$\ldots > 2^{(n \text{ DIV } 2)-1}\ c_2,$
also $c_n > 2^{(n \text{ DIV } 2)-1}$ für $n > 2$

iterativ

```
function fibi(n:integer): integer;
 var x,y: integer;
 begin
     x :=1;  y :=1; z :=1;
     while n >1 do
       begin
          n :=n-1;
          z :=x ;
          x :=x+y ;
          y :=z
        end ;
     return(z)
 end;
```

Aufwand:
$fibi(n) \in O(n)$
linearer Aufwand

Nach Initialisierung der Variablen wird der Schleifenrumpf (n-1)mal ausgeführt.

Zum besseren Vergleich führen wir Zeitkomplexitätsklassen ein.

Definition:
Ein Algorithmus A lese ein Wort $w \in \Sigma^*$ ein und gebe ein Wort $v \in \Delta^*$ aus. A berechnet also eine Funktion

$$f_A: \Sigma^* \rightarrow \Delta^* .$$

A terminiert dabei für alle Wörter aus dem Definitionsbereich von f_A. Man unterscheidet i.a. folgende drei zentralen *Zeitkomplexitätsklassen:*

A heißt:

- *linear-zeitbeschränkt*, wenn $f_A \in O(n)$.
- *polynomial-zeitbeschränkt*, wenn $\exists k \in N$, so daß $f_A \in O(n^k)$.
- *exponentiell-zeitbeschränkt*, wenn $\exists k \in N$, so daß $f_A \in O(k^n)$.

Linear-zeitbeschränkte Algorithmen sind z.B. das sequentielle Bearbeiten einer Liste. Typische polynomial-zeitbeschränkt Algorithmen sind primitive Sortierverfahren wie z.B. Bubble-Sort. Backtracking-Algorithmen sind Vertreter von exponentiell-zeitbeschränkten Verfahren.
Die Klasse der polynomial-zeitbeschränkten Algorithmen sind von besonderem Interesse, da viele Algorithmen aus der Praxis in dieser Klasse liegen.

Nichtdeterministische Algorithmen

Definition:
Man nennt einen Algorithmus A *nichtdeterministisch,* wenn A das Sprachelement **OR** (in beliebiger Anzahl) enthält:

OR (Anw1, Anw2) *bedeutet, daß entweder die Anweisung Anw1 oder die Anweisung Anw2 ausgeführt wird. Die Auswahl zwischen den beiden Anweisungen ist* ***willkürlich****.*
(Von mehreren Möglichkeiten wird eine geraten.)

Wir suchen nun die "kürzeste" Berechnung solcher Algorithmen, wozu wir wiederum die Schrittzahlfunktion heranziehen und ohne Beschränkung der Allgemeinheit annehmen, daß die Algorithmen als Ergebnis nur TRUE oder FALSE liefern..

Für $p \in \Sigma^*$ sei $s_A(p)$ die kleinste Schrittzahl in der der Algorithmus A nach Eingabe von p bei geschickter Auswahl an jedem OR das Ergebnis TRUE liefert.

Zur Charakterisierung von A abstrahieren wir vom konkreten Wort und betrachten $s_A(n) = Max\{ s_A(p) \mid p \in \Sigma^n \}$ die maximale Schrittzahl für Wörter p der Länge n.

A heißt *nichtdeterministisch polynomial-zeitbeschränkt*, wenn es ein $k \in N$ gibt, so daß $s_A \in O(n^k)$.

Der nichtdeterministische Fall stellt eine sichere obere Schranke bezüglich des Rechenaufwandes dar, der vom deterministischen Algorithmus nicht übertroffen werden kann.
Die Zeitkomplexitätsklassen für den nichtdeterministischen Fall zeigen dasselbe Verhalten wie im deterministischen Fall.

Komplexität von Funktionen und Sprachen

Wir abstrahieren von einem konkreten Algorithmus und gehen zur Charakterisierung der Komplexität von *Funktionen* über. Die Funktion $f_A : \Sigma^* \rightarrow \Delta^*$ werde durch das Verfahren A berechnet.

Man unterscheidet dabei in Beziehung zum Algorithmus:
- *A berechnet f_A, d.h. wird A angewendet auf p, entsteht als Ergebnis $f_A(p)$*
- *A entscheidet f_A (d.h. $(p, q) \in f_A$ gdw. A auf p anwendbar und als Ergebnis der Anwendung q entsteht)*

Definition:
Eine Funktion $f : \Sigma^* \rightarrow \Delta^*$ heißt ***polynomial-zeitberechenbar (pzb)*** gdw. es einen polynomial-zeitbeschränkten Algorithmus A gibt, der f berechnet.

Im weiteren kann man die ***Komplexität von Sprachen*** charakterisieren, indem man als Kriterium den Aufwand für die Entscheidung nimmt, ob ein Wort zur Sprache gehört oder nicht.

Sei $L \subseteq \Sigma^*$, C_L charakteristische Funktion von L mit:

$$C_L(w) = \begin{cases} \text{TRUE} & w \in L \\ & \text{falls} \\ \text{FALSE} & w \notin L \end{cases}$$

Definition:
Eine Menge, Sprache oder ein Problem L heißt ***polynomial-zeitbeschränkt***, wenn die charakteristische Funktion C_L polynomial-zeitberechenbar ist.

Definition:
a) P (bzw. NP) := {$L \subseteq \Sigma^*$; wobei L (nichtdeterministisch) polynomial-zeitbeschränkt } heißt die ***Klasse der (n)pzb-Sprachen****.*

b) $\underline{P}$*(bzw.* $\underline{NP}$*):={f:* $\Sigma^* \rightarrow \Delta^*$*; wobei f (nichtdeterministisch) polynomial-zeitberechenbar} heißt die* ***Klasse der (n)pzb-Funktionen****.*

Definition:

Seien $L, L' \subseteq \Sigma^*$

a) L' heißt *polynomial reduzierbar* auf L gdw. eine pzb Funktion $f:\Sigma^* \to \Sigma^*$ existiert, so daß gilt:

$$\forall w \in \Sigma^* \; (w \in L' \leftrightarrow f(w) \in L)$$

b) L heißt *NP-hart* gdw. für jedes $L' \in NP$ gilt:

L' ist polynomial reduzierbar auf L.

c) L heißt *NP-vollständig* gdw. gilt:

$L \in NP$ und L ist NP-hart.

Als Beispiel für ein NP-vollständiges Problem sei das Rucksackproblem genannt, bei dem zur gegebenen Folge $b, a_1, \ldots, a_n$ natürlicher Zahlen eine Auswahl I der a_i gesucht wird, für die $|b - \sum_{i \in I} a_i|$ minimal wird.

Übungsaufgaben:

1.) Beweisen Sie:
$O(f) + O(f) = O(f)$
$O(f) \cdot O(g) = O(f \cdot g)$
$f \in \Theta(2^{\log f})$
$\log(n!) \in \Theta(n \log(n))$ für jede natürliche Zahl n

2.) Entscheiden Sie für $g,h \in O(f)$:
$h(n) + g(n) \in O(f)$
$h(n) - g(n) \in o(f)$
$\frac{h(n)}{g(n)} \in O(1)$
$h \in O(g)$

3.) Vergleichen Sie die asymptotischen Aufwände für folgende zwei Verfahren, wenn für die Vergleiche eine assoziative Relation vorausgesetzt wird:
   a) n Objekte ordnen durch paarweises Vergleichen!
   b) n Objekte ordnen durch wählen eines Bezugsobjektes und ordnen bezüglich dieses Bezugsobjektes!

2. Graphen und Bäume

Graphstrukturen werden in der Informatik in vielfältiger Weise eingesetzt, z.B. bei Sortier- und Suchverfahren, bei der Sprachverarbeitung und in der Speicherverwaltung. Graphen bestehen aus Punkten (auch Knoten genannt) und Kanten, die die Knoten miteinander verbinden. Graphen können gerichtet oder ungerichtet sein, je nachdem, ob die Kanten eine Durchlaufrichtung besitzen oder nicht.

Graphen

Definition:

- Ein Paar G = (P, K) heißt *ungerichteter Graph*, wenn :
  P eine nichtleere Menge (Knotenmenge) und
  $K \subseteq 2^P$ eine Menge (Kantenmenge) ein- oder zweielementigen Teilmengen von P bezeichnet.

- Eine Kante $\{p\} \in K$ als einelementige Teilmenge von P heißt eine S*chlinge* .

- Ein Graph heißt *schlingenlos,* gdw. alle Kanten zweielementige Teilmengen von p, d.h. keine Schlingen, sind.

- Zwei Knoten p_1 und p_2 heißen genau dann *benachbart*, wenn
  $$\{p_1, p_2\} \in K \quad \text{oder} \quad \{p_1\} \in K \text{ und } p_1 = p_2 .$$

- Eine Folge $k = (p_0, p_1, ..., p_n)$ von Knoten mit $n>0$ und $\{p_i, p_{i+1}\} \in K$ für alle $0 \le i < n$ heißt *Kette* der Länge n in G, d.h. je zwei aufeinanderfolgende Knoten der Folge sind durch eine Kante verbunden.
  p_0 , p_n heißen *Endpunkte* der Kette,
  $\{p_j \mid 1 \le j < n\}$ heißt die Menge der *inneren Knoten* der Kette.

- Eine Kette $(p_0, p_1 , ..., p_n)$ heißt *geschlossen*, wenn $p_0 = p_n$, sonst *offen*.

- Eine Kette $(p_0, p_1, ..., p_n)$ heißt *Kantenzug*, wenn keine Kante mehrfach in der Kette auftritt, d.h., für alle $0 \le i \ne j <= n$ gilt $\{p_i, p_{i+1}\} \ne \{p_j, p_{j+1}\}$.

- Eine Kette $(p_0, p_1, ..., p_n)$ heißt *Weg* der Länge n, wenn alle Knoten der Kette paarweise verschieden sind, d.h., für alle $0 \le i \ne j < n$ gilt $p_i \ne p_j$.

- Ein Kantenzug $\bar{k} = (p_0, p_1, ..., p_n)$ heißt *Kreis* (oder *Zyklus*), wenn $\bar{k}$ geschlossen ist und $(p_0, p_1, ..., p_{n-1})$ ein Weg der Länge (n-1).

- Ein Graph G heißt *kreisfrei* (oder *azyklisch)*, falls er keinen Kreis enthält.

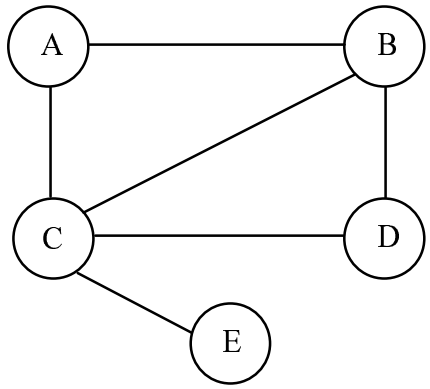
- Ein Graph G = (P, K) mit $|P| = 1$ und $K = \varnothing$ heißt ein *Punktgraph.*

- Ein Graph G = (P, K) heißt *endlich*, wenn seine Knotenmenge P (und damit auch
  seine Kantenmenge K) endlich ist.

- Ein Graph G=(P, K) heißt *zusammenhängend*, wenn für je zwei verschiedene Punkte p,q $\in$P ein Weg (p_0, p_1, ..., p_n) mit p_0 = p und p_n = q existiert, d.h., alle Knoten paarweise durch Wege verbunden sind.

Beispiel:

(A, B, C, D, B, C) ist *Kette*
[aber kein Kantenzug, da Kante {B, C} zweimal auftritt].
(A, C, D, B, C, E) ist *Kantenzug*
[aber kein Weg, da Knoten C zwei -mal im Kantenzug enthalten].
(A, C, D, B) ist *Weg*
[aber kein Kreis, da A≠B].
(A, C, D, B, A) ist *Kreis.*
Der Graph ist zusammenhängend.



Definition:
Die Struktur T = (P', K') heißt *Teilgraph* des Graphen G=(P, K), falls gilt:
P' $\subseteq$ P
K' $\subseteq$ K.
Liegen echte Teilmengenrelationen vor, so sprechen wir von einem *echten* Teilgraph.

Definition:
Die Struktur U = (P', K') heißt *Untergraph* des Graphen G=(P, K), falls gilt:
P' $\subseteq$ P
K' = K/P'.
(Die Menge K/P' enthält alle Kanten k aus K, die Knoten aus P' verbinden, d.h., K' entsteht durch Einschränkung von K auf P'.)

Definition:
Der Graph G' = (P', K') heißt *Komponente* des Graphen G=(P, K), falls gilt:
a) G' ist Untergraph von G
b) G' ist zusammenhängend
c) G' ist maximal, d.h. es existert kein p $\in$ P\P', so daß der durch p erweiterte Graph G' ein zusammenhängender Untergraph von G ist.

Definition:
Ein Paar G = (P, K) heißt *gerichteter Graph*, falls gilt:
- P nichtleere Menge (Knotenmenge)
- K $\subseteq$ P x P (Kantenrelation)

Bezeichnungen:
- (p_1,p_2) $\in$K, heißt Kante von p_1 nach p_2
- p_1 heißt der *Anfangsknoten (Quelle)* der Kante bzw. p_1 *Vorgänger* von p_2
- p_2 heißt der *Endknoten (Ziel)* der Kante bzw. p_2 *Nachfolger* von p_1

Ein Knoten p eines gerichteten Graphen heißt
- *maximal*, falls kein Knoten q mit (p,q) ∈ K als Nachfolger existiert bzw.
- *minimal*, falls kein Knoten q mit (q, p) ∈ K als Vorgänger existiert.

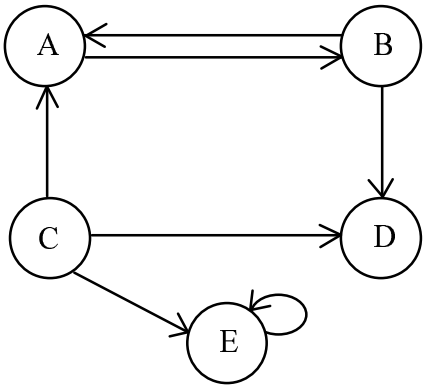
Die vorstehenden Begriffsbildungen für ungerichtete Graphen übertragen sich unter Berücksichtigung der Kantenrichtung entsprechend auf gerichtete Graphen.
Der wesentliche Unterschied zwischen ungerichteten und gerichteten Graphen besteht darin, daß eine Kante als geordnetes Paar aufgefaßt wird, d.h. zwischen den Knoten einer Kante eine Richtung (Ordnung) festgelegt ist.

Definition:

Ein gerichteter Graph G heißt *stark zusammenhängend*, wenn für alle Knoten p ≠ q sowohl ein Weg von p nach q als auch von q nach p existiert.

Beispiel:

Der Knoten D ist maximal.
Der Knoten C ist minimal.
(E, E) ∈ K ist eine Schleife.
Der Graph ist nicht stark zusammenhängend, da von E nach C kein Weg existiert.



Im folgenden werden wir einige Eigenschaften von gerichteten Graphen G = (P, K) genauer betrachten.

Voraussetzung: p ≠ q ∈P und W_G (p, q) sei die Menge aller Wege (Wegmenge) in G von p nach q.
Dann gilt $W_G (p, q) \neq \varnothing$ *gdw.* $(p, q) \in K^+$.
[K^+ bezeichnet dabei die transitive Hülle von K, d.h., $K^+ = U_{l>0} K^l$.]
Es gibt genau dann einen Weg von p nach q, wenn (p, q) Element der Hülle von K ist, d.h. K^+ bildet die Menge aller Wege in G.

Definition:

Die natürliche Zahl $d(p,q) = \text{Min}\{ l(w) \mid w \in W_G(p, q)$: Weg von p nach q in G$\}$ heißt
Abstand der Knoten p und q im Graphen G.
(l(w) bezeichnet die Länge des Weges w.)

Satz: *Der Graph G ist kreisfrei gdw.* $K^+ \cap Id(P) = \emptyset$.

Den Beweis des Satzes führen wir indirekt.

$\Rightarrow$ Wenn $K^+ \cap Id(P) \neq \emptyset$, dann $\exists\, p \in P$ mit $(p,p) \in K^+ = \bigcup_{l>0} K^l$,

d.h., es existiert eine nat.Zahl $k>1$ mit $(p,p) \in K^k$.
Man wähle k so, daß $(p,p) \notin K^i$ für $i < k$ gilt.
Dann existiert ein Weg $(p_0, ..., p_k) \in W_G(p, p)$ mit $p_0 = p = p_k$.
Dieser Weg $(p_0, ..., p_k)$ ist wiederholungsfrei, denn
sonst würde $i \neq j$ existieren mit: $0 \leq i < j < k$ mit $p_i = p_j$.
Dann gilt aber:
für $i = 0$: $(p_j, ..., p_k) \in W_G(p, p)$ mit Länge $(k - j) < k$;
für $i > 0$: $(p_0, ..., p_{i-1}, p_j, ..., p_k) \in W_G(p, p)$ mit Länge $k-(j-i) < k$,
somit $(p,p) \in K^{(k-(j-i))}$.
Widerspruch zu $(p, p) \notin K^i$ mit $i < k$.

$\Leftarrow$ Wenn ein Kreis $(p_0, ..., p_n)$ aus K^+ mit $p_0 = p = p_n$ existiert, dann ist $(p, p) \in K^+$ und damit $K^+ \cap Id(P) \neq \emptyset$.

q.e.d

Satz: *Wenn ein* $p \in P$ *mit* $\{p\} \times P \setminus \{p\} \subseteq K^+$ *existiert, d.h. von p gibt es einen Weg zu jedem anderen Knoten, dann ist G zusammenhängend.*

Beweis: Zu zeigen ist: Für $q,r \in P$ gilt: $W_G(q, r) \neq \emptyset$

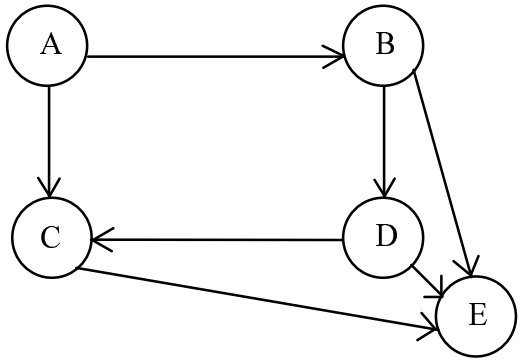
a) Wenn $p = q \neq r$, so gilt $(q,r) \in K^+$. Damit existiert ein Weg von q nach r, und somit ist $W_G(q, r) \neq \emptyset$.
b) $r = p \neq q$ analog a)
c) Bei $q,r \neq p$ gilt: (p, q) und $(p, r) \in K^+$,
d.h. $(p, p_1, ..., p_{n-1}, q) \in W_G(p, q)$ und $(p, p_n, ..., p_m, r) \in W_G(p, r)$.
Dann ist aber auch $(q, p_{n-1}, ..., p_1, p, p_n, ..., p_m, r) \in W_G(q, r)$.

q.e.d.

Satz: *G ist genau dann stark zusammenhängend, wenn* $P \times P \subseteq (K \cup K^{-1} \cup Id(P))^+$.

Definition:
Der gerichtete Graph G heißt *Netzplan*, falls G endlich, kreisfrei ist und genau ein maximales und genau ein minimales Element existiert.

Beispiel:
A ist ein minimales Element.
E ist ein maximales Element.
Jedem Knoten p kann eine Stufe S(p) zugeordnet werden, die die Länge (Abstand) des längsten Weges vom minimalen Element zum Knoten p angibt:
S(A) = 0, S(B) = 1, S(C) = 3, S(D) = 2
S(E) = 4



Darstellung von endlichen Graphen

Neben der graphischen Darstellung gibt es für endliche Graphen effektive Notationen, die für Implementierungen geeigneter sind. Dazu bieten sich z.B. Matrizen oder verkettete Listen an. Wir unterscheiden bei der Matrizendarstellung die Inzidenzmatrix I_G und die Adjazenzmatrix A_G.

Sei G = (P, K) ein endlicher und gerichteter Graph mit
P = $\{p_1, ..., p_n\}$, I_P Indexbereich von P; $p_i \in P$,
K = $\{k_1, ..., k_m\}$, I_K Indexbereich von K; $k_i \in K$.

Definition: *Inzidenzmatrix* I_G (Knoten-Kanten-Matrix)

I_G: $I_P \times I_K \rightarrow \{-1, 0, 1\}$ mit

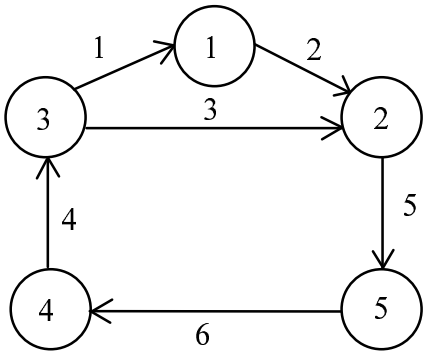
$$I_G(i,j) = \begin{cases} -1 & \text{es existiert ein p mit } (p_i, p) = k_j, \text{ d.h. } p_i \text{ Quelle von } k_j \\ 0 & \text{falls sonst} \\ 1 & \text{es existiert ein p mit } (p, p_i) = k_j, \text{ d.h. } p_i \text{ Ziel von } k_j \end{cases}$$

Definition: *Adjazenzmatrix* A_G (Knoten-Knoten-Matrix)

A_G: $I_P \times I_P \rightarrow \{1, 0\}$ mit

$$A_G(i,j) = \begin{cases} 1 & (p_i, p_j) \in K \text{ d.h. es existiert eine Kante von } p_i \text{ nach } p_j \\ 0 & \text{falls sonst} \end{cases}$$

Beispiel:



Kanten ***Knoten***

$$I_G = \begin{matrix} +1 & -1 & 0 & 0 & 0 & 0 \\ 0 & +1 & +1 & 0 & -1 & 0 \\ -1 & 0 & -1 & +1 & 0 & 0 \\ 0 & 0 & 0 & -1 & 0 & +1 \\ 0 & 0 & 0 & 0 & +1 & -1 \end{matrix} \text{ Knoten} \qquad A_G = \begin{matrix} 0 & 1 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 \\ 1 & 1 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 \\ 0 & 0 & 0 & 1 & 0 \end{matrix} \text{ Knoten}$$

Eigenschaften: $A^l_G(i, j) = |W^l_G(p_i, p_j)|$ (Anzahl der Wege von p_i nach p_j mit der Länge l), d.h., A^l_G ist Adjazenzmatrix des Graphen $G^l = (P, K^l)$.

Satz:
G ist kreisfrei gdw. ein l existiert mit $A^l_G \neq 0_{Matrix}$ *und für alle* $t > l$ *gilt:* $A^t_G = 0_{Matrix}$,

(Bei kreisfreien Graphen gibt es einen längsten Weg. Die Anzahl der Wege insgesamt ist beschränkt, wohingegen ein Kreis beliebig oft durchlaufen werden kann und somit keiner Längenbeschränkung existiert.)

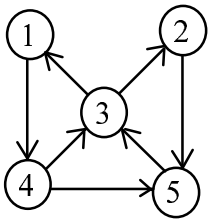
Satz:
Wenn $|P| = n$ *und* $p_i, p_j \in P$, *so gilt:* $W_G(p_i, p_j) = \varnothing$ *gdw.* $A^l_G = 0_{Matrix}$ *für alle* $1 \leq l \leq n$.

Weiter wollen wir den Begriff der Basis eines Graphen einführen. Dazu benötigen wir zunächst den Begriff der Erreichbarkeit.

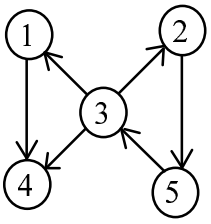
Definition:
Sei G =(P,K) gerichteter Graph, so heißt
a) $q \in P$ *erreichbar* von $p \in P$ im Graph G gdw. $W_G(p,q) \neq \varnothing$ oder p=q ist.
b) $Q' \subseteq P$ *erreichbar* von $p \in P$ im Graph G gdw. für alle $q \in Q'$gilt: q ist erreichbar von p.
c) $Q' \subseteq P$ *erreichbar* von $P' \subseteq P$ im Graph G gdw. für alle $q \in Q'$ ein $p \in P'$ existiert, so daß q erreichbar von p in G ist.

Definition:
$\varnothing \neq B \subseteq P$ heißt eine *Basis* (Knotenbasis) des Graphen G=(P, K), falls:
a) P erreichbar von B in G und
b) für alle $B' \subset B$ gilt: P ist nicht erreichbar von B' in G .

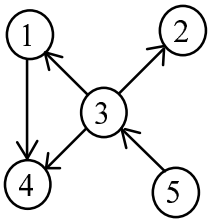
Beispiel



Basen: {1};{2};{3};{4};{5}



Basen: {2};{3};{5}



Basis: {5}

Satz:

1) Jeder (gerichtete) Graph besitzt eine Basis.
2) B ist eine Basis des Graphen G=(P, K) gdw.
   a) P erreichbar von B in G und
   b) wenn $p \neq q$ und $p,q \in B$, so $W_G(p, q) = \emptyset$ (Minimalität von B)
      Beweis
      $\Rightarrow$ Falls $W_G(p, q) \neq \emptyset$, dann P erreichbar $B\backslash\{q\}$ in G.
      Dies steht im Widerspruch zur Minimalität von B.

      $\Leftarrow$ Angenommen es existiert eine Basis $B' \subset B$ von G, dann würde ein $p \in B'$ und ein $q \in B\backslash B'$ existieren, wofür $W_G(p, q) \neq \emptyset$ und $p \neq q$ gilt.
      Dies steht im Widerspruch zur Minimalität von B .

3) Jeder zyklenfreie gerichtete endliche Graph besitzt mit der Menge der minimalen Knoten *genau* eine Basis .

Bäume

Zyklenfreie und zusammenhängende Graphen werden auch als Bäume bezeichnet. Durch Bäume können hierarchische Beziehungen zwischen Objekten beschrieben werden. Wir beschränken uns dabei auf gerichtete Graphen.

Definition:
- Ein (gerichteter) Graph G=(P, K) heißt *Baum (Tree)*, wenn er zusammenhängend und kreisfrei ist. Besitzt dieser Baum außerdem genau einen minimalen Knoten (Wurzel), dann heißt G *Wurzelbaum*.

- Ein Graph heißt *Wald*, wenn alle maximal zusammenhängenden Untergraphen (Komponenten) wieder Bäume sind.

- Ein Knoten p(bzw. q) heißt *Vorgänger* von q(bzw. *Nachfolger* von p), wenn $(p,q) \in K$.
$Vg_G(q) = \{p \mid (p, q) \in K\}$, p Vorgänger (Vater) von q .
$Nf_G(p) = \{q \mid (p, q) \in K\}$, q Nachfolger (Sohn) von p .

- Ein Knoten, der keinen Nachfolger besitzt, heißt *Blatt*: $Grad_G(p)=0$. Für die *Blattmenge*
B_G gilt somit $B_G = \{ p \mid Nf_G(p) = \emptyset\}$. Die Blattmenge zusammen mit der Wurzel bildet
den *Rand* $R_G = \{p \mid Nf_G(p) = \emptyset \vee p \text{ minimal}\}$ von G.

- Knoten, die weder minimal noch maximal sind, heißen *innere* Knoten.

Satz:
Ein Wurzelbaum G = (P, K) mit der Wurzel r hat folgende Eigenschaften:
a) Für jeden beliebigen Knoten $p \in P$ (ungleich Wurzel r) gibt es genau einen Weg von der Wurzel zum Knoten p, d.h. $\forall p \in P: p \neq r \rightarrow |W_G(r, p)| = 1$.

b) Jeder von der Wurzel verschiedene Knoten hat genau einen Vorgänger, d.h.

$$\forall p \in P: p \neq r \rightarrow |Vg_G(p)| = 1$$

Beweis:

Annahme: $q, q' \in Vg_G(p)$, d.h. $(q, p) \in K \wedge (q', p) \in K$.

Da der Graph G zusammenhängend ist, muß eine Kette $\bar{k}_1 = (r, ..., q)$ von r nach q und eine Kette $\bar{k}_2 = (r, ..., q')$ von r nach q' existieren . Bei $q \neq q'$ müssen zwei Ketten von r nach p existieren, im Widerspruch zur Baumeigenschaft (Zyklenfreiheit) von G.

q.e.d.

Definition:

- Unter der *Stufe*$_G(p)$ versteht man den um 1 erhöhten Abstand des Knotens p von der Wurzel r .

$$Stufe_G(p) = \begin{cases} 1 & p = r \\ l(w_G(r,p)) + 1 & \textit{falls } w_G(r,p) \text{ Weg in G von r nach p} \end{cases}$$

$Stufe_G(p) = Stufe_G(q) + 1$ falls $(q, p) \in K$, d.h. p Nachfolger von q

- Unter der *Höhe* eines Baumes versteht man die maximale Stufe seiner Knoten.

$$Höhe(G) = Max\{Stufe_G(p): p \in P\}$$

- Unter dem *Grad*$_G(p)$ versteht man die Anzahl der von p ausgehenden Kanten.

$$Grad_G(p) = |Nf_G(p)|$$

- Der Baum G heißt k-*beschränkt* , falls für alle $p \in P$ gilt $Grad_G(p) \leq k$.

- Ein 2-beschränkter Baum heißt *Binärbaum*.

- Ein Baum heißt *vollständiger* k-Baum (oder k-*gleichverzweigt* oder *perfekt*), falls für alle $p \in P$, die nicht Blatt sind, gilt: $Grad_G(p) = k$,
d.h., jeder Knoten hat entweder k oder keinen Nachfolger.

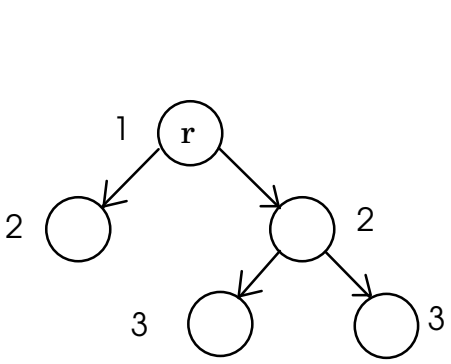
- Ein Baum heißt *ausgeglichen*, falls für alle Blätter $p \in B_G$ gilt:
$Stufe_G(p) = Höhe(G)$ oder $Stufe_G(p) = Höhe(G) - 1$.

- Ein Baum heißt *vollständig ausgeglichen*, falls
  a) er ausgeglichen ist und
  b) für alle Knoten $p \in P$, die nicht Blätter sind, gilt:
  die Anzahl der Knoten in den Teilbäumen unterscheidet sich höchstens um 1.

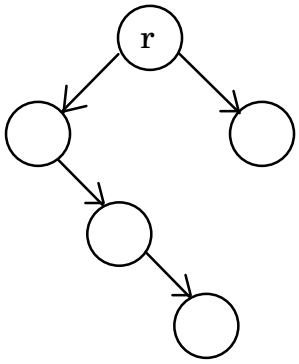
- Ein Baum G heißt *höhenbalanciert* oder *AVL-Baum*, falls für alle Knoten p,q,t gilt: aus $p,q \in Nf_G(t)$ folgt $|Höhe\ G(p) - Höhe\ G(q)| \leq 1$.
(G(p) bzw. G(q) sind Unterbäume mit der Wurzel p bzw. q.)

Insbesondere die höhenbalancierten Bäume werden später zum Sortieren und Suchen verwendet.

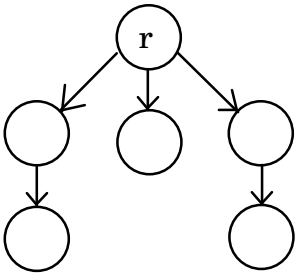
Beispiele:



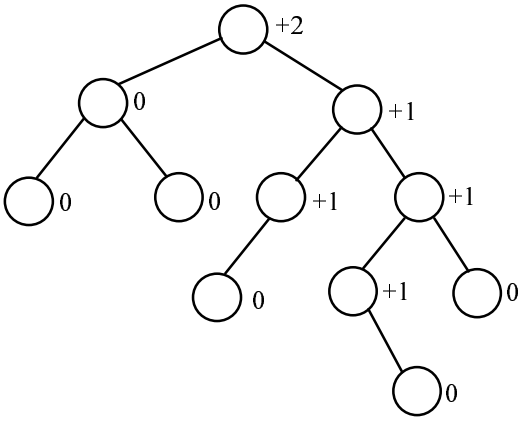
2-beschränkt
vollständiger 2-Baum



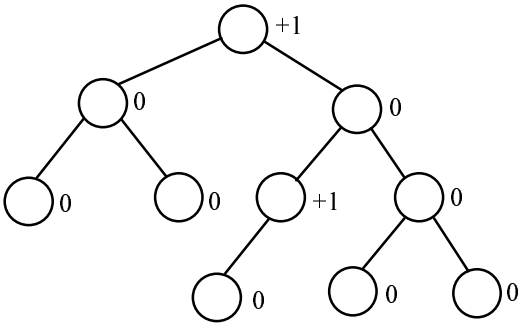
2-beschränkt
kein vollständiger 2-Baum



3-beschränkt
ausgeglichen
kein vollständiger 3-Baum



Kein AVL-Baum, da für die Wurzel
der Höhendifferenzbetrag > 1 ist.



AVL - Baum

Die Beträge der Höhendifferenz der Unterbäume sind am Knoten angegeben.

Satz:
Für jeden Baum G=(P, K) gilt:

1) Ist die Anzahl der Knoten größer als 1, so enthält der Rand R_G von G mindestens zwei Knoten, d.h. aus $|P| > 1$ folgt $|R_G| > 1$.

2) Die Anzahl der Kanten ist um eins kleiner als die der Knoten, d.h. $|K| = |P| - 1$.

3) Es existiert kein echter zusammenhängender Teilgraph G' = (P', K') mit P=P' und $K \subset K'$, d.h. es kann keine Kante aus G entfernt werden, ohne den Zusammenhang von G aufzugeben.

4) Ist G ein zyklenfreier Graph und $|K| = |P| - 1$, so ist G ein Baum.

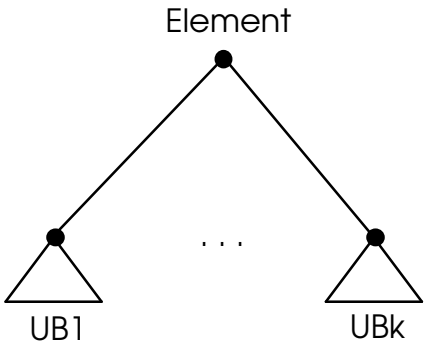
5) G ist ein endlicher Baum gdw. G endlich, zusammenhängend ist

und |K| = |P| - 1 gilt.

Bäume als Datenstrukturen

Ein k-beschränkter Baum läßt sich in Pascal-ähnlicher Notation wie folgt darstellen:

```
Type Baum = ^Knoten;
        Knoten =
        record
            Element: Elementtyp;
            UB1: Baum;
                 ...
            UBk: Baum;
         end;
```



(Dabei können UB1 bis UBk als Unterbäume aufgefaßt werden, die einer definierten Ordnung unterliegen.)

Besitzt ein Knoten weniger als k Nachfolger (Söhne), so zeigen die restlichen Zeiger auf NIL. Daraus erwächst das Problem, daß für Bäume mit großem k (Grad), die nicht vollständig ausgeglichen sind, die Anzahl der Pointer, die auf NIL zeigen, sehr groß ist. Daher versucht man einen k-beschränkten Baum als Binärbaum darzustellen, wodurch sich die Anzahl der NIL-Zeiger verringert.
Wir bezeichnen die beiden Nachfolger dann meist mit *linker* und *rechter Sohn* anstatt mit UB1 und UB2.

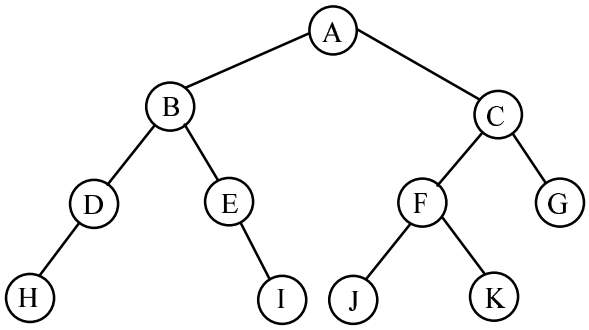
Eine oft gestellte Aufgabe besteht darin, einen Baum derart zu durchlaufen, daß jeder Knoten genau einmal besucht wird. Wir wollen für unsere weiteren Betrachtungen von einem Binärbaum ausgehen. Dabei sind insbesondere drei Arten des Durchlaufes von Interesse, der Inorder-, Preorder- und Postorder-Durchlauf.
Hierbei wird eine lineare Ordnung auf der Knotenmenge des Baumes definiert.

Inorder- besuche zuerst linken Sohn, dann den Vater und danach den rechten Sohn
(linker Sohn, Vater, rechter Sohn)

Preorder- besuche zuerst Vater, dann den linken Sohn und danach den rechten Sohn
(Vater, linker Sohn, rechter Sohn)

Postorder- besuche erst linken Sohn, dann rechten Sohn und danach den Vater
(linker Sohn, rechter Sohn, Vater)

Beispiel:



Inorder: H D B E I A J F K C G
Preorder: A B D H E I C F J K G
Postorder: H D I E B J K F G C A

Abschließend geben wir eine Pascal-ähnliche Implementierung dieser Durchläufe unter Nutzung des oben angegebenen Typs Baum.

```
procedure
inorder(tree:Baum);
begin
  if tree ≠ NIL then
    begin
    inorder(tree^.linkerSohn);
    write(tree^.Element);
    inorder(tree^.rechterSohn)
    end;
end;
```

```
procedure
preorder(tree:Baum);
begin
  if tree ≠ NIL then
    begin
    write(tree^.Element);
    preorder(tree^.linkerSohn);
    preorder(tree^.rechterSohn)
    end;
end;
```

```
procedure
postorder(tree:Baum);
begin
  if tree ≠ NIL then
  begin
  postorder(tree^.linkerSohn);
  postorder(tree^.rechterSohn);
  write(tree^.Element)
  end;
end;
```

(Die Funktion write schreibt den jeweiligen Elementknoten bzw. dessen Namen.)

3. Sortieralgorithmen

In Verbindung mit dem Speichern und Wiederauffinden von diskreten Objekten spielen Sortieralgorithmen eine entscheidende Rolle. Dabei kann man das Sortieren als eine Funktion auffassen, die aus einer ungeordneten Folge von Werten eine geordnete Folge als Ergebnis liefert.

Etwas präziser formuliert, läßt sich das Sortierproblem wie folgt beschreiben:

> Gegeben sei eine endliche Folge $S = s_1, s_2, ..., s_n$ von Elementen einer Menge M und eine auf M definierte lineare Ordnung ≤. Gesucht ist eine Permutation f von $\{1, ..., n\}$, wobei $s_{f(i)} \leq s_{f(i+1)}$ für alle $1 \leq i < n$ gilt.
> Als Ergebnis erhält man die sortierte (geordnete) Folge $S' = s_{f(1)} ... s_{f(n)}$.

Sortierverfahren, d.h. Verfahren zur Herstellung einer sortierten Folge, lassen sich nach verschieden Kriterien klassifizieren

1.) *interne und externe Sortierverfahren*

Von *internen* Sortierverfahren spricht man, wenn die gesamte zu sortierende Folge im Hauptspeicher gehalten werden kann. Auf diese Daten kann wahlfrei zugegriffen werden. (Als Datenstruktur bietet sich z.B. ARRAY an.)
Bei *externen* Verfahren kann nur einen Teil der zu sortierenden Folge im Hauptspeicher aufgenommen werden. Die gesamte Folge wird auf externen Speichermedien gehalten. Diese Verfahren finden insbesondere bei sehr großen Datenbeständen Anwendung. Der Zugriff auf die Daten ist i.d.R. sequentiell (Als Datenstruktur können z.B. Stapel verwendet werden.)

2.) *Effizienz*

In vielen Sortieralgorithmen sind die wesentlichen Operationen der Vergleich zweier Schlüssel (eindeutige Auszeichnung eines diskreten Objekt) bzw. der Tausch zweier Schlüsselwerte. Wir bezeichnen die Anzahl der Vergleiche mit V und die Anzahl der Tauschoperationen mit T . Die Aufwandsabschätzung wird im folgenden für den *worst case* ($[V_{max}, T_{max}]$ invers sortierte Ausgangsfolge) den *averege case* $[V_{\varnothing}, T_{\varnothing}]$ und den *best case* ($[V_{min}, T_{min}]$, Ausgangsfolge bereits sortiert) verglichen.

3.) *Stabile Sortierverfahren*

Ein Sortierverfahren heißt *stabil*, wenn sichergestellt ist, daß sich die Reihenfolge von Daten mit gleichem Schlüsselwert durch das Sortieren nicht verändert, d.h. für die zu sortierende Folge $S = s_1 \ldots s_n$ gilt:

$$\text{Aus } s_i = s_j \wedge i < j \text{ folgt } f(i) < f(j) \text{ für alle } 1 \le i, j \le n .$$

4.) *Methoden*

Gegeben sei die Folge $S = s_1 \ldots s_n$. und die lineare Ordnung $<$. Die Folge S kann unter Verwendung verschiedener Methoden sortiert werden, wobei die wichtigsten Sortiermethoden im folgenden kurz erläutert werden. Als Beispiel werden typische Sortierverfahren genannt, die auf speziellen Methoden basieren.

Methode	**Beschreibung**	**Beispiel**
Aussuchen (Auswählen)	Man *suche* in der Folge S ein Element s_i mit einer bestimmten Eigenschaft, z.B. das kleinste Element in der betrachteten Folge und bringe es an seine endgültige Position in der Folge. Anschließend verfahre man mit der Restfolge (ohne s_i) wie oben beschrieben weiter.	- Direktes Aussuchen - Heapsort
Einfügen	Die sortierte Folge entsteht durch sukzessives Einfügen von Elementen in eine bis dahin entstandene Teilfolge entsprechend einer vorgegebenen Ordnung.	- Direktes Einfügen - Binäres Einfügen

Austauschen	In der Folge S sucht man Elemente s_i und s_j, die die vorgegebene Ordnung nicht erfüllen und *vertauscht* diese. (Wie entscheidend die richtige Strategie zum Auffinden solcher Elemente ist, zeigen die verschiedenen Aufwandsabschätzungen der angegebenen Beispielverfahren.)	- Bubblesort - Shellsort - Heapsort - Quicksort
Mischen	Seien S_1 bzw. S_2 geordnete Teilfolgen der Länge k bzw. l, dann werden diese zu einer geordneten Gesamtfolge S der Länge k + l *gemischt*, indem die jeweils nächsten beiden Elemente der Folgen verglichen, das kleinere der beiden in S übernommen und aus der Ausgangsfolge entfernt wird.	- Einphasen-Mischen - Zweiphasen-Mischen - natürliches Mischen
Streuen und Sammeln	Die Elemente aus S werden auf Fächer (Zellen) verteilt. Dabei nimmt das i-te Fach die Anzahl der Vorkommen von a_i in S auf. Gibt man nun der Reihe nach alle Elemente a_1 .. a_n so oft aus, wie es die Anzahl in den Fächern angibt, erhält man eine sortierte Folge. Diese etwas intuitivere Vorstellung trug dazu bei, daß diese Methode häufig auch als *Sortieren durch Verteilen* bezeichnet wird.	- Bucketsort

Interne Sortierverfahren

1. Selection Sort (Direktes Aussuchen)

Einer der einfachsten Sortieralgorithmen gestaltet sich wie folgt:

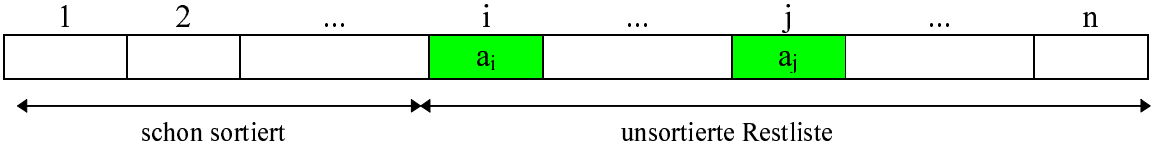
- finde das kleinste Element im Feld
- tausche es mit dem ersten Element aus
- wiederhole die Prozedur mit dem Restfeld ohne erstem Element

Die Prozedur für das Sortieren des Feldes a[1],...,a[n] in Pascal-ähnlicher Notation kann wie folgt beschrieben werden:

```
procedure selection;
  var i, j, min, t : integer;
  begin
    for i:=1 to n-1 do                    {durchlaufe 1.bis vorletztes Element}
     begin min:=i;                        {setze Minimum als Vergleichselement}
           for j:= i+1 to n do             {durchlaufe Restfeld}
              if a[j]<a[min] then  min:=j;  {suche kleineres Element}
             t:=a[min]; a[min]:=a[i]; a[i]:=t   {tausche Elemente}
     end;
   end;
```

Der Zeiger wandert durch das Feld a[1] ... a[n] und sucht das kleinste Element a[min], tauscht dieses mit dem ersten Element a[1] aus und betrachtet im nächsten Schritt das Restfeld, welches durch Weglassen des ersten Elementes entsteht.
Der Sortiervorgang ist abgeschlossen, wenn das Restfeld nur noch aus einem Element besteht.

a_j kleinstes Element in der Restliste, wird an die Stelle i gebracht



Der Vorteil dieses Verfahrens liegt in der geringen Anzahl der Tauschoperationen.

Aufwandsabschätzung:

$T_{max} = T_{min} = T_{\varnothing} = n-1$ (Drei Zuweisungen entsprechen einer Tauschoperation.)

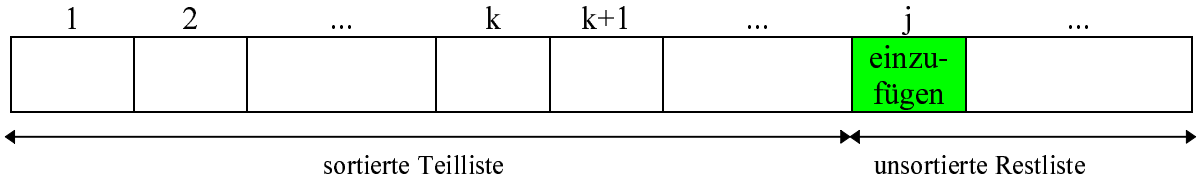
$V_{min} = V_{max} = V_{\varnothing} = \frac{n(n-1)}{2}$

2. Insertion Sort (Sortieren durch Einfügen)

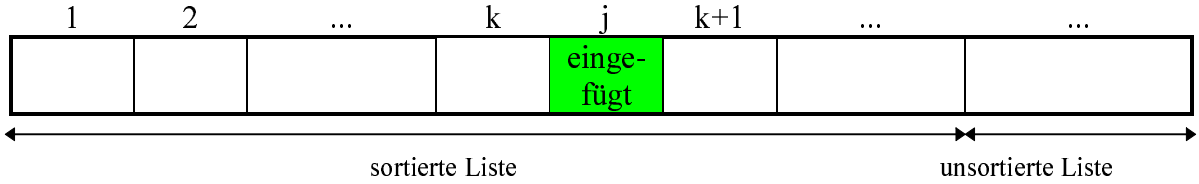
Ein dem Selection Sort ähnlicher, jedoch flexiblerer Algorithmus, ist Insertion sort. Dabei wird das erste Element der unsortierten Liste an die richtige Stelle in der bereits sortierten Liste eingefügt. Die Suche findet somit im sortierten Feld statt.

Beispiel: Element j ist größer als Element k, aber kleiner als k + 1.

vor dem Durchlauf



nach einem Durchlauf
(Man beachte die Verschiebung der Feldinizes des sortierten Feldes um 1 f ür i >k.)



Der Nachteil dieses Verfahrens ist die größere Anzahl nötiger Tauschoperationen, da immer alle größeren Elemente der sortierten Liste verschoben werden müssen.

Die Implementierung in Pascal-ähnlicher Notation:

```
procedure insertion;
    var i, j, v: integer;
    begin
    for i:=2 to n do        {1.Element schon sortiert, daher beginne bei Element 2
                             (Voraussetzung Feldlänge > 1)}
      begin
        v:=a[i]; j:=i;                  {v einzufügendes Element}
        while a[j-1]>v ∧ j > 1 do       {solange Element im Feld größer, verschiebe
                                         dessen Pos. um 1, beachte  j > 1}
            begin a[j] := a[j-1]; j:=j-1 end;
        a[j]:=v;                        {sonst  ersetze a[j]}
      end;
    end;
```

Anstatt der Bedingung j > 1 im while-Konstrukt könnte auch ein Wächter (a[0]) eingesetzt werden, um den Fall, daß das einzufügende Element das kleinste im Feld ist, zu erkennen.

Aufwandsabschätzung:

$V_{min} = n-1; \quad V_{max} = \sum_{i=2}^{n} i = \frac{n(n+1)}{2} - 1; \quad V_{\varnothing} = \frac{n^2 + n - 2}{4}$

$T_{min} = n-1; \quad T_{max} = (n+1) + \sum_{i=2}^{n} i = \frac{n^2 + 3n}{2} \quad ; \quad T_{\varnothing} = \sum_{i=2}^{n} (1 + \frac{i-1}{2}) = \frac{n^2 + 3n - 4}{4}$

3. Bubble Sort (Sortieren durch direktes Austauschen)

Bei diesem Verfahren steigen die großen Elemente ähnlich Blasen (engl. bubble) ans Ende der Liste. Bei jedem Durchlauf werden Nachbarn in der Liste miteinander verglichen, gegebenenfalls werden sie vertauscht. Die Liste ist sortiert, wenn bei einem Durchlauf kein Tausch erfolgt.
Da bei jedem Durchlauf das jeweils größte Element ans Ende transportiert wird, kann die im weiteren betrachtete Teilliste um ein Element verkürzt werden.

Als Implementierung in Pascal-ähnlicher Notation für ein Feld a[1], ... ,a[n] erhält man:

```
procedure bubble_sort;
    var i, j, k : integer;
    begin
      for i := n downto 1 do            {durchlaufe das Feld absteigend}
        for j := 2 to i do              {durchlaufe Feld aufsteigend  bis Index i}
          if a[j - 1] > a[j ] then      {vergleiche benachbarte Elemente}
            begin
              t := a[j - 1] ;           {tausche benachbarte Elemente}
              a[j - 1] := a[j ] ;
              a[j ] := t
            end
    end;
```

Die Aufwandsabschätzung ergibt:

$$V_{min} = V_{max} = V_{\varnothing} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}$$

$$T_{min} = 0; \qquad T_{max} = \sum_{i=1}^{n-1} i = \frac{n(n-1)}{2}; \qquad T_{\varnothing} = \frac{n(n-1)}{4}$$

(Wird mit alternativer Durchlaufrichtung gearbeitet, d.h. wechseln die „Blasen“ bei jedem Durchgang ihre Laufrichtung, so wird dieses Verfahren als *Shaker-Sort* bezeichnet.)

Schnelle Sortierverfahren

1. Shellsort

Während bislang, z.B. bei Bubblesort, immer nur benachbarte Elemente verglichen und getauscht wurden, ist es mit diesem Verfahren möglich, weiter voneinander entfernte Elemente zu vergleichen.

Dazu wählt man eine *Distanz (Abstand)* h der zu vergleichenden Elemente. Durch schrittweises Verringern der Distanz wird eine gegebene (grobe) Vorsortierung ständig verfeinert, um schließlich mit der Distanz h = 1 eine sortierte Liste zu erhalten. Die Wahl der Distanzen kann willkürlich erfolgen, sie muß nur mit h = 1 enden. Es gibt sehr günstige und sehr ungünstige Folgen von Distanzen, so daß die Distanz-Auswahl eine nicht unwesentliche Rolle spielt.

Beispiel:

Liste	44	60	8	14	5	12
h=4	44	60	8	14	5	12
	5	60	8	14	44	12
h=3	5	12	8	14	44	60
h=1	5	12	8	14	44	60
h=1	5	8	12	14	44	60

Implementierung in Pascal-ähnlicher Notation für das Feld a[1], ... ,a[n]:

```
procedure shellsort ( var a : array [1..n] of integer );
var i, v, h, t : integer;

begin
  h := 1;
  repeat h := 3*h + 1 until h > n ;    {erzeugt kleinsten Distanzwert, der größer N ist }
  h := h div 3 ;                                  { bestimme Distanz h }
  while h > 0 do
```

```
  begin
    for i := h + 1 to n do          {durchlaufe alle Elemente mit Index größer Distanz}
        begin
            v := i - h ;                    {Vergleichs-Index}
            while v > 0 do
                if a[v] > a[v + h] then     {Prüfung, ob Elemente zu vertauschen
sind}
                begin
                    t := a[v];
                  a[v] := a[v + h];         {Elementetausch}
                a[v+h] := t;
                    v := v - h;             {veringere Vergleichs-Index um Distanz}
                end
                else v := 0                 {setze Abbruch-Bed. für while-Anw.}
        end;
        h := h div 3                            {verringere Distanz}
  end;
end;
```

Geeignete Distanzfolgen sind nach KNUTH z.B.:

$h_{k-1} = 3 * h_k + 1$

$h_{k-1} = 2 * h_k + 1.$

Aufwandsabschätzung:

Zur Effizienz von Shellsort können keine genauen Angaben gemacht werden, da diese von der Auswahl der Distanzfolgen abhängig ist. Auch ein Vergleich verschiedener Distanzenfolgen gestaltet sich schwierig, da natürlich immer auch das zu sortierende Feld zu beachten ist. Shellsort führt mit obiger Distanzenfolge niemals mehr als $n^{3/2}$ Vergleiche aus.

2. Heapsort

Ein vollständiger Binärbaum heißt ein Heap oder hat Heap-Eigenschaft, wenn die Werte der Vaterknoten niemals kleiner als die ihrer Söhne sind.
Für die Nummerierung der Knoten a_i soll folgender Zusammenhang gelten:

$$Nf(a_i) = \begin{cases} (a_{2i}, a_{2i+1}) & 2 \\ (a_{2i}) \quad falls \;\; grad(a_i) = 1 & \\ \varnothing & 0 \end{cases}$$

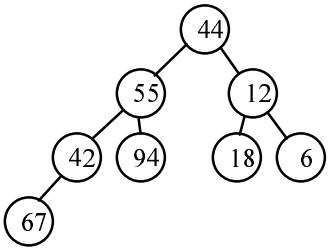
Die erwähnte Heap-Eigenschaft besagt dann, daß für alle Knoten a_i mit grad > 0 gilt :

$$a_i \geq \max (a_{2i} , a_{2i+1})$$

Durch Vergleichen der Söhne mit ihrem Vaterknoten wird schrittweise (von unten beginnend) die Heapeigenschaft hergestellt. Nach einem Durchlauf steht das größte Element in der Wurzel des Baumes (a_1). Durch Vertauschen von Wurzel und letztem

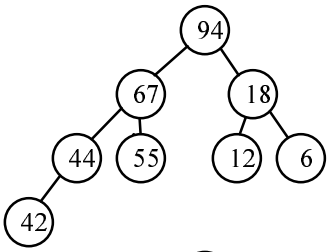
Element, sowie Weglassen des letzten (und größten) Elements entsteht ein neuer Baum. Dieser Vorgang wird solange wiederholt, bis nur noch ein Element übrigbleibt.

Beispiel:



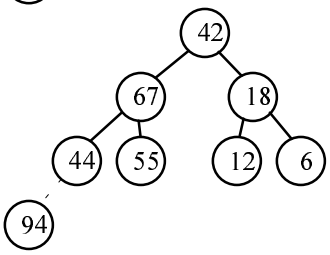
-Baum verletzt Heapbedingung
Aufbau des Heap von unten:

1. 67 ⇔ 42
2. 18 ⇔ 12
3. 94 ⇔ 55
4. 94 ⇔ 44
5. 67 ⇔ 44

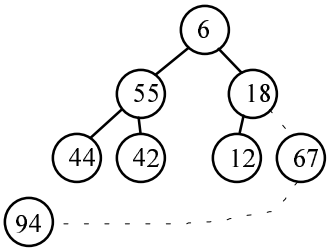


-Heapbedingung wurde hergestellt

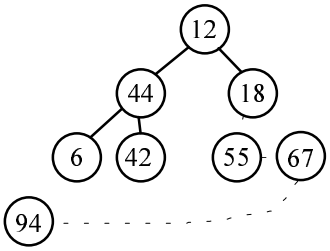
6. 94 ⇔ 42
7. 94 als letztes und größtes Element der Liste abspalten



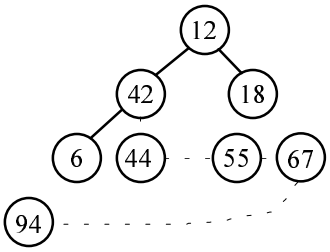
1. 67 ⇔ 42
2. 42 ⇔ 55 Heap hergestellt
3. 67 ⇔ 6
4. 67 abspalten



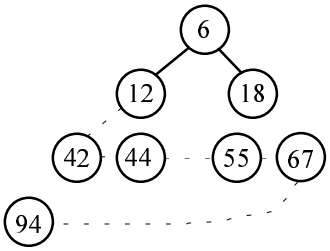
1. 55 ⇔ 6
2. 44 ⇔ 6 Heap hergestellt
3. 55 ⇔ 12
4. 55 abspalten



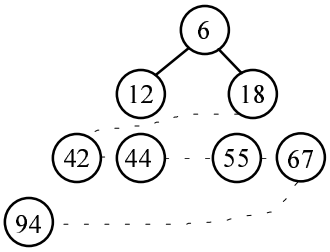
1. 44 ⇔ 12
2. 42 ⇔ 12 Heap hergestellt
3. 44 ⇔ 12
4. 44 abspalten



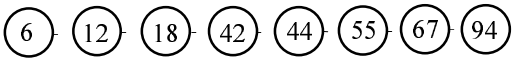
1. 42 ⇔ 12 Heap hergestellt
2. 42 ⇔ 6
3. 42 abspalten



1. 18 ⇔ 6 Heap hergestellt
2. 18 ⇔ 6
3. 18 abspalten



1. 12 ⇔ 6 Heap hergestellt
2. 12 ⇔ 6
3. 12 abspalten



Sortierte Folge

Implementierung in Pascal-ähnlicher Notation:

Prozedur zum Herstellen der Heapeigenschaft:

```
procedure heaprek ( wurzel, max : integer );
var s,t : integer;                {max = maximale Anzahl der Knoten im Baum}
begin
        s := 2 * wurzel;                 { Wurzel eines Unterbaumes }
        if s ≤ max then                                        { falls Unterbaum existiert }
          begin
              if (s < max) and ( a[s+1] > a[s] ) then
                        s := s + 1;                            { größeren Unterbaum suchen }
               if a [s] > a[wurzel] then
                 begin
                          t := a[s];
                       a[s] := a[wurzel];                { falls Wurzel kleiner, tausche}
                   a[wurzel] := t;
                    heaprek ( s, max )                  { Wiederholung im Unterbaum }
                 end;
           end;
end;
```

Prozedur zum Abspalten des größten (obersten) Elementes:

```
procedure heapsort;
var i, t: integer;
begin
        for i := n div 2 downto 1 do                    { Heap wird aufgebaut }
            heaprek ( i, n );              {n - Knotenanzahl des Gesamtbaumes}
        for i := n - 1 downto 1 do                      { Heap wird  abgebaut }
            begin
                    t := a[1];
                a[1] := a[i + 1];                   {Element in Heap setzen}
            a[i + 1] := t;
            heaprek ( 1, i )                {restlichen Teilbaum bearbeiten}
            end;
end;
```

Aufwandsabschätzung:
Für den worst case gilt $T_{max} < n \text{ ld } n$ und $V_{max} < 2n \text{ ld } n$,
denn $T_1 = n - 1$, (Aufwand um die Heapeigenschaft am Anfang herzustellen),
$k \leq \log(n+1)$; $n = 2^k - 1$ mit $|Nf(a_k)| \leq 1$ (Wiederherstellung der Heapeigenschaft) und damit $T_2 = n(k-1) - (k-1) = (n-1)(k-1)$. Insgesamt also
$T_{max} = T_1 + T_2 = (n-1)\, k \leq (n-1) \text{ ld } (n+1) < n \text{ ld } n$ und
$V_{max} = (n-1)(k-1) * 2 \leq 2(n - 1) (\log (n +1) -1) < 2n \text{ ld } n$.

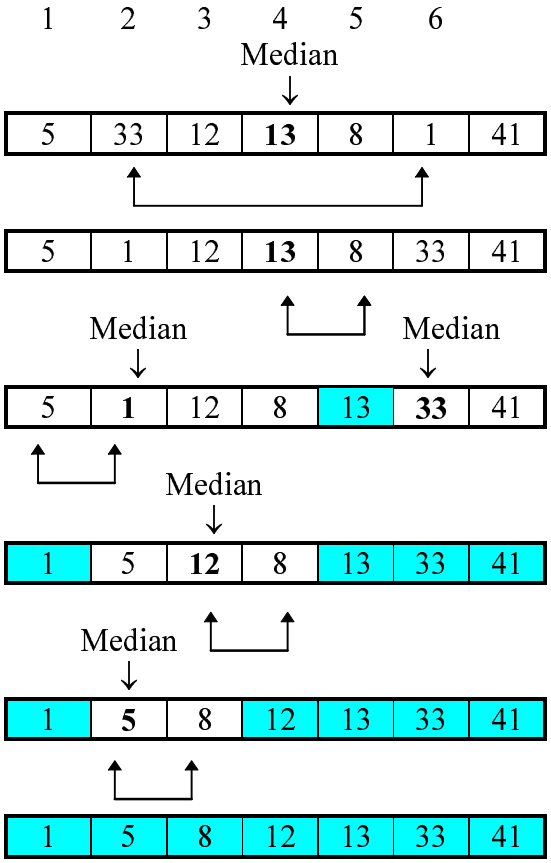
Demnach sind T_{max} und V_{max} von der Ordnung O(n log n) und damit besser als $O(n^2)$ für Bubblesort.

3. Quicksort

Quicksort ist der schnellste bekannte interne Sortieralgorithmus, da hier das Austauschen am effizientesten realisiert wird. Der grundlegende Algorithmus wurde 1960 von C. A. R. Hoare entwickelt. Aufgrund der einfachen Implementation und der breiten Anwendungsmöglichkeiten ist Quicksort sehr weit verbreitet.

Das Grundprinzip von Quicksort ist Teile und Herrsche . Man wählt aus der zu sortierenden Liste ein Element aus (im günstigsten Fall den Medianwert [n div 2]). Nun durchsucht man das Feld von links nach einem größeren, von rechts nach einem kleineren oder gleich großen Element, und tauscht diese. Dieser Vorgang wird solange wiederholt, bis sich die beiden Laufindizes treffen.
Entstanden sind zwei Teilfelder, links das der kleiner-gleichen, rechts das der größeren Elemente, dazwischen das ausgewählte Element.
Der Algorithmus wird nun solange auf die entstehenden Teillisten angewendet, bis jede Teilliste aus nur genau einem Element besteht, diese sind naturgemäß sortiert.

Beispiel:



Der Medianwert dient als Schlüsselindex. Dieser Index kann jedoch innerhalb der Feldgrenzen beliebig gewählt werden.

Vorzüge von Quicksort:

- arbeitet am Ort, d.h. benötigt nur einen kleinen Hilfs-Speicher
- für n Elemente durchschnittlich nur $n \log n$ Operationen
- kurze innere Schleife

Nachteile:

- Rekursivität (Beseitigung führt zu kompliziertem Verfahren)
- n^2 Operationen im ungünstigen Fall
- Algorithmus ist störanfällig (Fehler in der Implementierung können unbemerkt bleiben und nur für bestimmte Daten falsche Ergebnisse liefern)

Implementierung in Pascal-ähnlicher Notation:

```
procedure quicksort (l,r: integer);
  var v, t, i, j: integer;
  begin
        if r>l then                    {rechte Feldgrenze größer 1}
          begin
```

```
            v:= a[r];                {Vergleichselement}
            i := l - 1;              {Laufindex, durchläuft Feld von links}
            j := r;                  {Laufindex, durchläuft Feld von rechts}
            repeat
               repeat i := i+1 until a[i] >=v;  {suche Elemente, die nicht dem}
               repeat j :=j - 1 until a[j] <=v;   {Ordnungskriterium entprechen}
               t:=a[i];   a[i]:=a[j];  a[j]:=t;       {vertausche Elemente}
            until j <= i;
            a[j]:=a[i]; a[i]:=a[r]; a[r]:=t;  {vertausche Elemente}
            quicksort (l, i - 1);                    {bearbeite linkes Teilfeld}
            quicksort (i + 1, r);                    {bearbeite rechtes Teilfeld}
         end;
   end;
```

Aufwandsabschätzung:
Ist ein sortiertes Feld gegeben, so findet keine Tauschoperation statt (T_{min} =0). Geht man von einer Teilung aus, die stets in der Mitte des betrachteten Intervalls liegt, so ergeben sich pro Rekursionsstufe im Mittel n/4 Tauschoperationen, da bei der Hälfte der im Intervall liegenden Elemente ein Tausch durchgeführt wird.
Die Anzahl der Rekursionsstufen beträgt ld(n) und damit die Anzahl der Tauschoperationen T = ¼ n ld(n) . Betrachtet man die Vergleichsoperationen, so sind diese im worst case von der Ordnung $O(n^2)$. Dieser Fall tritt dann ein, wenn der maximale (minimale) Wert im unsortierten Feld als Vergleichselement verwendet wird. Im best case beträgt die Anzahl der Vergleichsoperationen n ld(n).
Quicksort besitzt ein durchschnittliches Laufzeitverhalten von O(n log n).

Insgesamt also V_{min} =n ld(n), V_{max} =$O(n^2)$, $V_{\varnothing}$ =2 n ln(n) und T_{min} =0 , T_{max} =$\frac{1}{4}$ n ld(n) , $T_{\varnothing}$ =n ln(n) .

Externe Sortierverfahren

Eine Reihe von Sortiervorgängen müssen auf großen Datenmengen ausgeführt werden. Da diese Daten oftmals nicht komplett in den Hauptspeicher eines Systems geladen werden können, ist es nötig, die Daten auf einem anderen Speichermedium zu sortieren. Methoden , die dieses Konzept verwirklichen, nennt man externe Sortierverfahren.

Sortieren durch Mischen

Das Grundkonzept besteht darin, Teile der Datei vorzusortieren und dann zu größeren Teilen zusammenzufassen.
Zuerst werden die Daten in zwei Hilfsdateien so aufgeteilt, daß Segmente mit bereits sortierten Teilen entstehen. Danach wird jeweils ein Segment der einen mit einem Segement der anderen *gemischt*, d.h. zu einem größeren, wiederum sortierten Teil zusammengefaßt, indem die ersten Elemente miteinander verglichen werden, das kleinere wird in die Gesamtdatei geschrieben, das größere wird mit dem nächsten des anderen Bandes verglichen. Dieser Vorgang wiederholt sich solange, bis eines der Bänder leer ist.
Damit entstehen größere sortierte Segmente in der Gesamtdatei, mit denen der Vorgang solange wiederholt wird, bis die Gesamtdatei aus genau einem Segment besteht.

Natürliches Mischen

Beim natürlichen Mischen wird eine etwaige Vorsortierung der Datei berücksichtigt. Die Segmente werden so festgelegt, daß aufeinanderfolgende Elemente in der richtigen Reihenfolge in einem Segment zusammengefaßt werden.

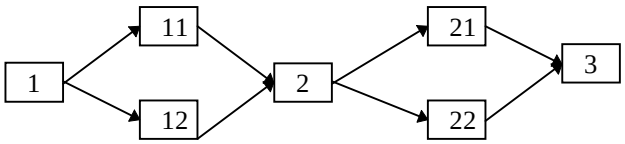
Beispiel:

Gesamtdatei 1:	44	55	12	42	94	18	6	67
Hilfsdatei 11:	44	55	| 18					
Hilfsdatei 12:	12	42	94	| 6	67			

durch Mischen von HD 1 und HD 2 entsteht:

Gesamtdatei 2:	12	42	44	55	94	| 6	18	67
Hilfsdatei 21:	12	42	44	55	94			
Hilfsdatei 22:	6	18	67					
Mischen:								
Gesamtdatei 3:	6	12	18	42	44	55	67	94

Aufwand: 2 mal Verteilen, 2 mal Mischen



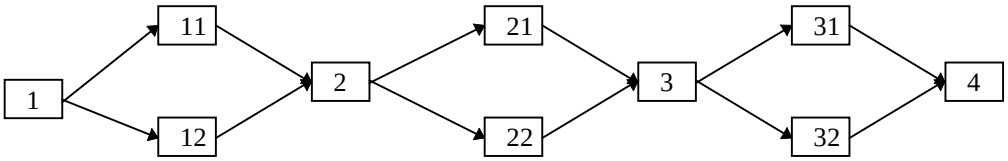
Direktes Mischen

Beim direkten Mischen bleibt eine Vorsortierung unberücksichtigt, zu Beginn besteht jedes Segment aus genau einem Element, welcher per Definition sortiert ist.

Beispiel:

Gesamtdatei 1:	44	55	12	42	94	18	6	67
Hilfsdatei 11:	44 |	12 |	94 |	6				
Hilfsdatei 12:	55 |	42 |	18 |	67				
Mischen								
Gesamtdatei 2:	44	55 |	12	42 |	18	94 |	6	67
Hilfsdatei 21:	44	55 |	18	94				
Hilfsdatei 22:	12	42 |	6	67				
Mischen								
Gesamtdatei 3:	12	42	44	55 |	6	18	67	94
Hilfsdatei 31:	12	42	44	55				
Hilfsdatei 32:	6	18	67	94				
Mischen								
Gesamtdatei 4:	6	12	18	42	44	55	67	94

Aufwand: 3 mal Verteilen, 3 mal Mischen



Ausgeglichenes Mehrwege-Mischen

Statt der zwei Hilfsdateien können auch mehrere verwendet werden, dies ergibt sich aus der Tatsache, daß die Schreib- und Lesezugriffe die dominanten Aktionen im Programm sind, und sich damit eine Verringerung dieser Zugriffe deutlich auf die Leistungsfähigkeit des Algorithmus auswirken kann.

Beispiel:

Annahmen: - Speicher kann drei Datensätze gleichzeitig speichern
- beliebig viele Ein- und Ausgabebänder

Gesamtdatei: ASO RTI NGA NDM ERG ING EXA MPL E

Verteilen: Segmente aus drei Elementen, die im Haupspeicher sortiert (internes Verfahren) werden und anschließend auf die Bänder 1,2 und 3 verteilt werden;

Band 1: AOS DMN AEX
Band 2: IRT EGR LMP
Band 3: AGN GIN E

Mischen: Die drei ersten Segmente der Bänder (AOS, IRT, AGN) werden auf Band 4, die zweiten (DMN, EGR, GIN) auf Band 5 und die dritten (AEX, LMP, E) auf Band 6 gemischt. Dazu werden jeweils die drei ersten Elememte gelesen (Band 1: A, Band2: I, Band 3: A), das kleinste zurückgeschrieben (A) und dafür das nächste gelesen (Band 1: O) , bis alle Elemente zurückgeschrieben sind.

Band 4: AAGINORST
Band 5: DEGGIMNNR
Band 6: AEELMPX

Mischen:

Band 1: AAADEEEGGGIILMMNNNOPRRSTX

Die 25 Elemente wurden in zwei Durchläufen gemischt, ein Vergleich mit den Verfahren auf drei Bändern zeigt die verbesserte Leistung deutlich (zwei bzw. drei Durchläufe bei 8 Elementen).

Mergesort

Eine Implementierung des Sortierens durch Mischen beruht auf folgendem rekursiven Algorithmus: Sortieren einer Datei durch Teilen der Datei in zwei Hälften, die sortiert und dann gemischt werden.

Implementierung in Pascal-ähnlicher Notation:
- Sortieren eines Feldes a[l..r] unter Verwendung des Hilfsfeldes b[l..r]

```
procedure mergesort (l, r:integer);
  var i, j, k, m: integer;
  begin
        if r-1 >0 then
          begin
                m:=(r+1) div 2;              {Mittelindex von Intervall bestimmen}
                mergesort (l,m);                    {bearbeite linkes Teilfeld}
                mergesort (m+1, r);                 {bearbeite rechtes Teilfeld}
                for i:=m downto 1 do b[i]:=a[i];          {initialisiere Hilfsfeld b}
                for j:=m+1 to r do b[r+m+1-j]:=a[j]; {Zuweisung in umgek. Reihenfolge}
                for k:=1 to r do             {durchl. Hilfsfeld b, Start: i=1,j=r}
                  if b[i]<b[j] then                       {Vergleich}
                        begin
                          a[k]:=b[i];                     {Mische auf Feld a}
                          i:=i+1
                        end;
                  else begin
                          a[k]:=b[j ];                    {Mische auf Feld a}
                          j :=j - 1
                        end;
          end;
  end;
```

Aufwandsabschätzung:

Beginnt man bei n Elementen mit Läufen (aufsteigend geordnete Teilfolge von Elementen) der Länge 1, so beträgt die maximale Rekursionstiefe (Phasen) ld n . In jeder Phase, die sich aus mehreren Läufen zusammensetzt, werden n Vergleiche durchgeführt. Der Aufwand für Vergleichsoperationen ist somit von der Ordnung O (n ln n).

4. Suchverfahren

Aus einer Menge von Daten sind häufig Elemente mit einer vorgegebenen Eigenschaft zu selektieren. Dieser Suchvorgang wird in der Regel über einer geordneten Datenmenge durchgeführt.
Liegen ungeordnete Daten vor, so müßte jedes Element der Datenmenge einzeln überprüft werden. In einem solchen Fall spricht man von "Linearem Suchen". Bei Vorgabe einer Ordnung als Zusatzinformation über die Daten kann das Suchen effektiver gestaltet werden.
Suchverfahren können nach internen und externen Verfahren klassifiziert werden. Interne Suchverfahren halten während des Suchvorgangs sämtliche Daten im

Hauptspeicher, wohingegen externe Verfahren nur einen Teil des Datenbestandes im Hauptspeicher halten können und den Rest auf Sekundärspeicher auslagern.
Um in einer Datenmenge zu suchen, müssen die einzelnen Daten eindeutig identifiziert werden. Dies geschieht mit einem Schlüssel (Adresse).
Wir wollen in einem Feld (a) vom Typ Suchtyp mit Hilfe eines sogenannten *Suchschlüssels* (x) das erste Vorkommen des Schlüsselwertes (Schlüssel_1) finden, der mit dem Suchschlüssel übereinstimmt.
Die entsprechende Datenstruktur kann in Pascal-ähnlicher Notation wie folgt spezifiziert werden.

```
Type Suchtyp = record
                  Schlüssel_1:   Schlüsseltyp;
                        ...
                  Inhalt_1:      Inhaltstyp;
                        ...
               end;

var      a : array [1..n] of Suchtyp;
         x: Schlüsseltyp;
```

Unser Ziel wird es sein, nach Resultat = MIN { i : a[i].Schlüssel_1 = x} zu suchen.

Lineares Suchen

Die einfachste, zugleich aber auch aufwendigste, Möglichkeit, dieses gestellte Ziel zu erreichen, besteht im sequentiellen (linearen) Durchsuchen des gegebenen Feldes. Bei dieser Vorgehensweise wird Element für Element geprüft, ob der Schlüsselwert gleich dem Suchschlüssel ist.

Das entsprechende Verfahren in Pascal-ähnlicher Notation lautet:

a: **array of** Suchtyp

```
function suchen (x:Schlüsseltyp): integer ;
  var i: integer ;
  begin
        i:=1
        while ((i ≤ n) and  (a[i].Schlüssel_1 ≠ x )) do  i:=i+1;
        return (i)
  end;
```

Die Prozedur terminiert mit dem ersten Index i, an dem das Ziel erfüllt ist (i ≤ n) oder mit dem Endindex des Feldes (i = n+1).

Aufwandsabschätzung:

Geht man davon aus, daß das gesuchte Element in der Menge enthalten ist, so haben wir minimalen Aufwand, wenn das erste Schlüssel-Element des Feldes mit dem Suchschüssel übereinstimmt. Im worst case finden *n* -Vergleiche statt und die Schleife wird *n-1* mal durchlaufen, d.h. das n-te Feld-Element ist das Gesuchte. Daraus ergibt sich, daß im Mittel *(n-1)/2* Schleifendurchläufe und *n/2* Vergleiche stattfinden. Lineares Suchen erfordert demnach einen linearen Aufwand mit $V_{min}=1$, $V_{max}=n$ und $V_{\varnothing}=\frac{n}{2}$.

Eine Beschleunigung des vorgestellten Verfahrens läßt sich erreichen, wenn der Vergleich (i <= n) aus dem WHILE-Konstrukt entfernt wird und dafür eine Feldende-Marke benutzt wird. Belegt man die Feldende-Marke mit dem Wert des Suchschlüssels, so wird sichergestellt, daß das Verfahren immer terminiert. Der Funktionsrumpf würde dann wie folgt aussehen:

```
begin
        a[n+1].Schlüssel_1: = x;        {sicheres Abbruch-Kriterium für while}
        i:=1;
        while ( a[i].Schlüssel_1≠ x ) do   i:=i+1;
        return(i)
end ;
```

Binäres Suchen

Ein effizienterer Algorithmus für das Suchen in geordneten Datenmengen ist das *Binäre Suchen*.
Das zu suchende Objekt wird mit dem Mittel-Element des Feldes verglichen; ist es gleich dem Suchschlüssel, wäre die Suche erfolgreich gewesen, ist es jedoch kleiner (bezüglich der gegebenen Ordnung), betrachtet man das linke Teilfeld, ist es größer, so wird mit dem rechten Teilfeld fortgefahren. Nach dem gleichen Prinzip verfahre man mit den Teilfeldern. Es wird im folgenden ein rekursiver und ein iterativer Algorithmus zur Lösung dieser Aufgabe in Pascal-ähnlicher Notation angegeben.

Rekursiver Algorithmus:

```
function  Suchen(L,R: Integer): boolean ;
begin
        if L ≤ R then
                VerglPos := (L + R) div 2;
                if a.[VerglPos].Schlüssel_1  >  x then
                        Suchen (L, VerglPos - 1);
                else
                if a.[VerglPos].Schlüssel_1  < x then
                        Suchen (VerglPos +1, R );
                 else  return (TRUE) ;
        else return (FALSE)
end;
```

Bei jedem Schritt wird die Anzahl der zu prüfenden Elemente halbiert, d.h. in m Schritten kann ein Datenumfang von $n = 2^m$ Elementen bearbeitet werden. Die Komplexität ist demnach von der Ordnung Θ (ld n).

Iterativer Algorithmus:

```
function bin_suchen (L,R: integer, x:typ): integer ;
 var   VerglPos: integer;
       find: boolean;
 begin
       find := FALSE;
       while ((L ≤ R) and not find)  do
      begin
            VerglPos := ( L + R ) div 2;
            if x < a[VerglPos]. Schlüssel_1 then R := VerglPos - 1
            else     if x > a[VerglPos]. Schlüssel_1 then L := VerglPos + 1
                     else find := TRUE
       end;
       if find then return(VerglPos)              {gefunden, gib VerglPos zurück}
       else return(0);                        { nicht gefunden,  gib Wert 0 zurück}
end;
```

Beim Aufruf der Funktion wird L (Links) mit 1 und und R (Rechts) mit der Länge des Feldes (n) initialisiert. Wurde das Element gefunden, so wird die Position zurückgegeben, ansonsten der Wert "0". Mit dieser Funktion wird nicht zwingend das Element mit dem kleinsten Index gefunden.

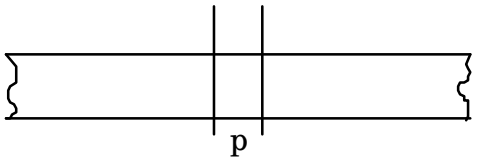
Aufwandsabschätzung:

$V_{min} = 1; V_{max} = \text{ld } n;$

Der Algorithmus ist effizienter als das einfache Suchen. Für große Datenmengen, in denen häufig gesucht wird, kann durch Sortieren der Daten und Anwenden der binären Suche eine wesentliche Leistungssteigerung erzielt werden.

Muster-Suche

Bisher beschränkte sich die Suche in einer Datenmenge auf ein einzelnes Element, in der Regel auf den Suchschüssel. Wir wollen diesen Ansatz erweitern, indem ganze Zeichenketten in einem vorgegebenen String gesucht werden.
Dazu betrachten wir den String *s* , in dem wir das Muster (pattern) *p* suchen.



String *s*

Probleme: a) Ist *p* Teilwort von *s*, d.h. existiert q_1, q_2 mit $s = q_1\ p\ q_2$?
b) Gesucht wird das erste Auftreten von p in $s = q_1\ p\ q_2$.

Beispiel:
s = b a b a b a
p = a b a, q_1 = b, q_2 = b a

Vorgegeben seien die Objektstrukturen:

```
s : array [0 .. n - 1] of char;
p : array [0 .. m - 1] of char;          mit m ≤ n.
```

Das einfachste Verfahren festzustellen, ob *p* in *s* enthalten ist, ist das *sequentielle Durchsuchen* von s.

Dieses Verfahren kann in Pascal-ähnlicher Notation wie folgt beschrieben werden:

```
function mustersuchen: integer;
  var i, j : integer;                    {i Index im  Textfeld, j Index im Muster}

  begin
       i:=1; j :=1;
       repeat
         if a[i] = p[j ]
              then begin
                     i:=i+1;  j :=j+1;
                   end
             else begin
                     i:=i-j+2;  j := 1;
                   end;
        until (j > m) or (i >n);          {Muster oder Text vollst. durchlaufen}
        if j >m then
                  return (i - m)          {Muster gefunden}
             else  return (0)             {Muster nicht gefunden}
  end;
```

Das Programm verwendet zwei Zeiger (i auf Text und j auf Muster). Solange beide Zeiger auf übereinstimmende Zeichen zeigen, werden sie inkrementiert. Erreicht j das Ende des Musters, so ist es im Text enthalten.
Stimmen die beiden Zeichen nicht überein, so wird j auf den Anfang des Musters zurückgesetzt. Der Index i wird so verändert, das es der Verschiebung des Musters um eine Position zum Text entspricht.
Als *Aufwandsabschätzung* ergibt sich : $V_{max} = m * n$, $V_{min} = m$.

Verfahren von Knuth - Morris - Pratt (KMP)

Beim sequentiellen Suchen wurde das Muster gegenüber dem Text um eine Stelle nach rechts verschoben. Das KMP-Verfahren (D.E. Knuth, J.H. Morris und V.R.Pratt) versucht das Muster um mehr als eine Stelle gegenüber dem Text zu verschieben.

Das Verfahren vergleicht wie bisher zwei Zeichenketten von links nach rechts. Stimmen zwei Zeichen überein so wird die Suche beim nächsten Zeichen fortgesetzt. Sind die beiden Zeichen ungleich, erhebt sich die Frage, um wieviele Stellen nach rechts verschoben werden muß und wovon die Verschiebung abhängig ist.

Beispiel:

- ohne Wiederholung im Muster

s =	I	N	F	O	R	M	A	T	I	K
p =	I	N	F	O	S	T	A	N	D	
					I	N	F	O	S	T	A	N	D

Da keine Wiederholungen im Muster auftreten, kann das Muster beim nächsten Versuch an der Stelle angesetzt werden, wo der Unterschied auftrat. Im Beispiel kann somit bei der 5. Postion im Text fortgefahren werden.

mit Wiederholungen im Muster:

s =	G	E	G	E	N	S	A	T	Z	
p =	G	E	G	E	B	E	N			
			G	E	G	E	B	E	N	

Im nächsten Versuch kann das Muster auf den Anfang der Wiederholung verschoben werden.

Schlußfolgerung:

Die Verschiebung ist abhängig vom Muster und der Ungleichheitsstelle.
Für jede Stelle j des Musters muß eine Verschiebezahl dj ermittelt werden, die angibt an welcher Stelle im Muster der Vergleich mit dem Text fortgesetzt werden soll, wenn an der Stelle j die Ungleichheit zwischen Text und Muster auftrat. Die Verschiebewerte dj werden bestimmt, indem das Muster p mit sich selbst verglichen wird.

Dazu betrachten wir das folgende in Pascal-ähnlicher Notation beschriebene Verfahren :

```
procedure Verschiebezahl ;
  var   k: integer;

  begin
        k := 0;
        d [0] := -1;
        for i:= 1 to (musterlänge -1) do
        begin
```

```
            if  p [i] := p [ k ] then
                    d [i] := d [k]
            else
                    d [i] := k;
            while ((k ≥ 0) and (p [i] ≠ p [k])) do k:= d[k];
            k:= k + 1
        end;
  end;
```

```
function kmpsuchen : integer ;
  var i, j: integer;                    {Index i durchläuft Muster, j durchläuft Text}
  begin
        Verschiebezahl ;                {Verschiebezahlen initialisieren}
        i:= 0; j:= 0;

        while ( (i < musterlänge) and (j < textlänge)) do
        begin  {Text noch nicht vollst. durchlaufen und Muster noch nicht gefunden}
              while ((i ≥ 0)  and  (a [j] ≠ p [i]))  do
              begin
                    i := d [i]                          {verschiebe pattern-Index}
              end;
              i := i + 1; j := j + 1                    {vergleiche nächste Zeichen}
        end;
        if  (i ≥  musterlänge) then
             return(j - musterlänge)  {Muster gefunden; Rückgabe Musteranfangsindex}
        else return(-1)                 {Muster nicht gefunden, Rückgabe -1}
  end;
```

Der vollständige Algorithmus ergibt sich aus dem einfachen Verfahren, dessen Verschiebung um eine Stelle gemäß KMP modifiziert wird.
Der Text wird bei Durchführung des Suchvorgangs sequentiell durchlaufen werden.

Zur *Aufwandsabschätzung* stellen wir zunächst fest, daß das Muster vollstänig durchlaufen werden muß. Die minimale Anzahl der Vergleiche ist demnach $V_{min} = m$.
Betrachten wir die Anzahl der maximalen Vergleichsoperationen, so ist diese deutlich geringer als bei den einfachen Verfahren. Dies ergibt sich daraus, daß der Textzeiger während des Suchens nicht zurückgesetzt wird und somit maximal (n-1)-mal nach rechts wandert. Diese Aktion ist jeweils mit einem Vergleich verbunden, also (n-1) Vergleichen. Zusätzlich sind noch die Vergleiche zu betrachten, die entstehen, wenn das Muster verschoben wird (im worst-case muß an der Abbruch-Stelle erneut verglichen werden). Das Muster kann maximal um (n-m)-mal Stellen verschoben werden. Insgesamt ergibt sich als Abschätzung der maximalen Vergleichsoperationen $V_{max} < 2 \cdot n - m$.

Der Algorithmus von Boyer - Moore

Ähnlich dem Ansatz von KMP wird hier versucht, aus dem Muster Informationen über die Verschiebezahlen zu erhalten, die wiederum in einer Tabelle gespeichert werden. Im Gegensatz zu KMP erfolgt der Vergleich von Text und Muster aber von rechts nach links. Es wird also zuerst das letzte Zeichen des Musters mit dem entsprechenden Text-Zeichen verglichen. Sind die Zeichen gleich, wird mit dem vorletzten Zeichen des Musters analog fortgefahren. Tritt Ungleichheit auf, so wird aus dem Textzeichen, das mit dem *letzten* Muster-Zeichen verglichen wurde, die Verschiebung ermittelt. Existiert an dieser Stelle z.B. ein Text-Zeichen, welches überhaupt nicht im Muster enthalten ist, so kann das Muster bis hinter dieses Zeichen verschoben werden.

Beispiel : Es wird geprüft, ob das letzte Zeichen *S* von *GEBES* gleich dem Zeichen *N* ist. Da *N* im Muster nicht auftritt, kann sofort an der 6. Stelle der Zeichenkette fortgesetzt werden. Beim KMP-Verfahren müßte hingegen ab 3. Stelle geprüft werden.

s =	G	E	G	E	N	S	A	T	Z
p =	G	E	B	E	S				
						G	E	B	E	S

Verschiebetabelle:
(Erstes Auftretens von rechts im Muster; Musterende bei Position 0)

d [A] = 5; da nicht im Text enthalten
d [B] = 2;
d [E] = 1;
d [G] = 4;
sonst d [...] = 5.

Die Tabelle zur Verwaltung der Verschiebezahlen wird für alle Zeichen angelegt. Kommt ein Zeichen im Muster nicht vor, so wird die Länge des Musters gespeichert.
Ein Nachteil gegenüber dem KMP-Verfahren besteht darin, daß der Text nicht sequentiell durchlaufen werden kann.
Das entsprechende Verfahren kann in Pascal-ähnlicher Notation wie folgt beschrieben werden:

```
procedure Verschiebezahlen ;
  var   k: integer;
  begin
        for k := ord('A') to ord('z') do        {Verschiebezahlen  initialisieren}
                d[k] := musterlänge;
        for k := 0 to  (musterlänge - 2) do   {k durchläuft Muster bis vorletztes Element}
            d[ord(p[k])] := musterlänge - k - 1          {Verschiebezahlen berechnen}
  end;


function bmsuchen : integer;
  var i, j,v: integer;                    {i Musterindex, j Textindex, v Vergleichsindex}
  begin
        Verschiebezahlen;                          {Verschiebezahlen initialisieren}
        i:= musterlänge - 1; j:= i;
       while ( (i ≥ 0) and (j < textlänge)) do          {Text noch nicht durchlaufen,
```

```
    begin                                        Muster noch nicht gefunden}
          i :=  musterlänge -1;
          v:= j;
          loop
          begin if (i < 0) then exit;            {Textstelle gefunden}
                if  ( p [i] ≠ a [v] ) then       {verschieben}
                 begin
                    j := j +  d[ ord(a[j]) ];
                    exit
                 end;
                 i := i - 1; v := v -1                    {rückwärts vergleichen}
          end;
    end;
    if  (i <0 ) then
          return(j - musterlänge + 1)   {Musterbeginn gefunden}
    else  return(-1)                    {Muster nicht gefunden}
end;
```

Aufwandsabschätzung: worst case $V_{max} = n \cdot m$
best case $V_{min} = m$

Im worst case ist der Aufwand nicht besser als beim linearen Suchen. Dafür verbessert sich jedoch die Effizienz des Verfahrens. Mindestens m Vergleiche zwischen Text und Muster sind erforderlich.

Hashing

Beim Hasing wird aus dem Suchargument direkt ein Schlüssel berechnet.
Möchte man z.B. im Telefonbuch *Herrn Ackel* suchen, so gibt der erste Buchstabe des Namens bereits eine genauere Lokalisierung. Bei üblicher lexikographischen Ordnung steht der Name *Ackel* im Telefonbuch vor dem Namen *Zuckel.* Den Namen *Ackel* würde man demnach von vorn und *Zuckel* von hinten suchen.
Haben die zu speichernden Objekte eine komplexere Struktur, so nutzt man eine Komponente bzw. eine Kombination dieser Komponenten als Wert, der die Abbildung in den Speicher ermöglicht. Dieser Wert wird häufig als *Schlüssel* bezeichnet und könnte z.B. der Nachname einer Person sein. Man betrachte nun eine Funktion h, die als *Hash-Funktion* oder auch als *Schlüsseltransformation* bezeichnet wird:

$$h : D \rightarrow A.$$

dabei ist A der allgemeiner Adreßbereich und D der Schlüssel-Wertebereich,
Die Funktion h ist surjektiv und sollte effizient berechnet werden können. Die zu speichernden Schlüssel sollen möglichst gleichmäßig über den Speicher verteilt sein.

Beispiel: Man bilde die 12 Monatsnamen Januar bis Dezember auf n = 17 Speicherzellen ab. Dabei wird ein Monatsname als eine Zeichenkette x_1 bis x_k aufgefaßt. Als einfache Hashfunktion bietet an:

$$h(x_1 \ldots x_k) = \sum_{i=1}^{k} N(x_i) \bmod n$$

Unter Berücksichtigung der ersten drei Zeichen der Monatsnamen (JAN ... DEZ) erhalten wir mit N(A) = 1; ... ; N(Z) = 26 die Hashfunktion:

$h(x_1, ..., x_k) = (N(x_1) + N(x_2) + N(x_3)) \bmod 17$

Damit ergibt sich folgende Speicherverteilung:

0	NOV	9	
1	APR, DEZ	10	
2	MÄR	11	JUN
3		12	AUG, OKT
4		13	FEB
5		14	
6	MAI, SEP	15	
7		16	
8	JAN		

Wie wir sehen, werden durch die Hashfunktion verschiedene Monate einundderselben Speicherzelle zugeordnet (z.B MAI und SEP der Speicherzelle 6). Diese Erscheinung wird als *Kollision* bezeichnet.
Hashverfahren unterscheiden sich in der Art der Kollisionsbehandlung und in der zugrundegelegten Speicherbegrenzung.
Von *offenen Hashverfahren* spricht man, wenn beliebig viele Schlüssel aufgenommen werden können.
Wird von vornherein die Anzahl der Speicherzellen durch einen konstanten Wert festgelegt, so wird dies als *geschlossenes Hashing* bezeichnet.

Perfektes Hashing

Unter perfektem Hashing wollen wir eine injektive Abbildung der Schlüsselwerte auf die Speicherzellen ohne Kollision verstehen. Mit steigender Anzahl der zu speichernden Elemente wird diese Vorgehensweise ineffektiver bzw. unmöglich. Die Verwendung eines direkten Schlüssels ist dann praktisch nicht mehr durchführbar.

Das perfekte Hashing wird nur dann sinnvoll sein, wenn folgende Voraussetzungen erfüllt sind:

a) Die Schlüsselmenge ist bekannt (konstant).
b) Die Anzahl der Schüssel darf nicht größer sein, als die Anzahl der Einträge, die die Hashtabelle zur Verfügung stellt.

Die Hashtabelle sollte auch nicht zu groß gewählt werden, da sonst Speicherplatz verschenkt wird.

Beispiel:

Die Personennamen Agnes , Annegret , Arthur , Beat , Boris , Cäsar , Clements , Dagmar , Dagobert und Dolores sollen in einem Feld mit Hilfe einer Hashfunktion abgespeichert werden.

Folgende Hashfunktion wäre möglich:

h (Pname**) =** (3 * (**ORD** (Pname[0]) + (**ORD**(Pname[length(Pname)]))

Die Funktion ORD wird auf ein einzelnes Zeichen (Buchstabe) angewendet und ordnet diesem Zeichen einen Integer-Wert zu.

- angewendet auf Endzeichen (EZ):	ORD (r) = 0
	ORD (s) = 1
	ORD (t) = 2
- angewendet auf Anfangszeichen (AZ):	ORD (A) = 0
	ORD (B) = 1
	ORD (C) = 2
	ORD (D) = 3

(Durch length bezeichnen wir die Längenfunktion für Zeichenketten.)Die Hashfunktion h liefert für die Personennamen gerade die Zahlen 0 für *Arthur* bis 11 für *Dorit*.

Kollisionsbehandlung

Da die Voraussetzungen für perfektes Hashing nur sehr selten erfüllt sind, kommt der Kollisionsbehandlung ein besonderes Bedeutung zu.
Man wird versuchen, die Hashfunktion so zu wählen, daß möglichst wenig Kollisonen entstehen und die Funktion selbst effizient implementiert werden kann.
Es wird empfohlen, die Hashtabelle nicht mehr als 80 % auszulasten, da eine größere Auslastung zu einer wesentlich höheren Kollisionsrate führt. Für die Länge der Hash-Tabelle sollte eine Primzahl gewählen werden. In einer Hashfunktion h= (7*ord(x_0) + 11*ord(x_{m-1}) + 19*m) mod n sollten die Konstanten 7, 11, 19 teilerfremd zu n sein.
Bei der *externen Kollisionsbehandlung* lagert man die betreffenden Werte auf eine Halde aus.

Beispiel:
Ein einfaches Verfahren zur Kollisionsbehandlung kann unter Verwendung eines Varianten-Records angegeben werden. Die Elemente der Hashtabelle sind mit *belegt := FALSE* zu initialisieren. Wird eine Speicherstelle besetzt, so muß der Pointer zum Nachfolgeelement (next) auf NIL gesetzt werden. Treffen weitere Elemente auf den gleichen Feldindex, so hängt man diese durch Verkettung an die bereits bestehende Liste an.

```
type nextpointer = pointer to varecord;
 varecord = record
        case  belegt: boolean of
                FALSE: { leer }
                TRUE:   Schlüssel: integer;
                        Inhalt: InhaltTyp;
                        next: nextpointer
        end;

var HashTab          : array [0.. n-1] of varecord;
```

Nimmt man die Möglichkeit der externen Kollisionsbehandlung nicht in Anspruch, muß bei Kollision in den noch verbliebenen unbelegten Speicherplätzen gespeichert werden. Man spricht dann von *interner Kollisionsbehandlung*.

Lineares Sondieren

Wenn zum ersten Mal eine Kollision eintritt, wird in der Hashtabelle sequentiell nach einem freien Platz gesucht (sondiert). Eine entsprechende Hashfunktion ist z.B.

$h_0(x) = h(x)$
$h_i(x) = (h(x) + c * i) \bmod k \quad \text{mit} \quad 1 \leq i \leq k-1.$

Kommt es während der Sondierung wiederum zur Kollision, so spricht man von *Sekundärkollision*. Um den Speicher möglichst gut auszulasten, sollten die Konstanten c und k teilerfremd sein. Prinzipiell bietet dieses Verfahren verglichen mit c = 1 keine Vorteile, da sich auch hier Zyklen bilden. (Vielfache von c)

Beispiel: c = 8; n = 100:
Es ergibt sich nach 200/8 Schritten ein Zyklus, d.h. es wird nur ein Viertel der Tabelle durchlaufen.

Quadratisches Sondieren

Im Gegensatz zum linearen Sondieren, wo c konstant blieb, wird es hier durch die folgende Vorschrift verändert:

$c_1 = 1,$
$c_{i+1} = c_i + 2.$

Damit ergibt sich die Hashfunktion $h_i(x) = h(h(x) + c_i) \bmod k = (h(x) + i^2) \bmod k.$

Quadratisches Sondieren ergibt keine Verbesserung bei Primärkollisionen. Die Wahrscheinlichkeit für die Bildung von längeren Zyklen bei der Sekundärkollision ist deutlich geringer als beim linearen Sondieren.

Doppel-Hashing

Beim Doppel-Hashing werden zwei Hashfunktionen h und h' verwendet. Diese sollten so gewählt werden, daß Doppelkollisonen gegenüber Primärkollisionen eine wesentlich geringere Wahrscheinlichkeit besitzen. Als Hashfunktion wäre dann etwa

$$h_i(x) = (h(x) + h'(x) * i^2) \bmod m$$

möglich. Das Problem ist hier, zwei derart unabhängige Funktionen h und h' zu finden.

Literatur (Auswahl) :

Appelrath, H.J.; Ludewig, J.: Skriptum Informatik - eine konventielle Einführung, Teubner, 1991

Bauer, F.L.; Goos, G.: Informatik - eine einführende Übersicht, Springer, 1990

Bauer, F.L.; Wössner, H.: Algorithmische Sprache und Programmentwicklung, Springer, 1984

Goldschlager, L.; Lister, A.: Informatik - eine moderne Einführung, Hanser, 1986

Hotz; G.: Einführung in die Informatik, Teubner, 1990

Loeckx, J.; Mehlhorn, K.; Wilhelm,R.: Grundlagen der Programmiersprachen, Teubner, 1986

Rembold, U.: Einführung in die Informatik, Hanser, 1987

Schefe, P.: Informatik - eine konstruktive Einführung, Bibl. Inst. Mannheim, 1987

Aufgaben zur Vorlesung „Digitale Informationsverarbeitung“ im Wintersemester 1996/97

Algorithmenbegriff

1. Beschreiben Sie ein Verfahren zur Suche eines Buches in einer Bibliothek. Beachten Sie dabei, daß das gesuchte Buch in der Bibliothek eventuell nicht geführt wird oder gegenwärtig entliehen ist.

2. Beschreiben Sie ein Verfahren zur Berechnung der kleinsten Primzahl, die eine gegebene Zahl übertrifft, und begründen Sie Ihr Verfahren.

3. Beschreiben Sie ein Verfahren, welches a) den Buchstaben bestimmt, der in einer Laufschrift bzw. b) das Wort bestimmt, das in einem Buch am häufigsten auftritt.

4. Zeigen Sie, daß die Funktion f, die für jedes Wort über {0, 1} mit einer geraden (bzw. ungeraden) Anzahl von 1 als Funktionswert 0 (bzw. 1) liefert, berechenbar ist. Geben Sie dazu ein (Markov-) Programm an und weisen Sie nach, daß dieses Programm die Funktion f berechnet.

5. Zeigen Sie, daß die Menge aller geraden natürlichen Zahlen n als |-Folgen der Länge 2n+1 aufzählbar ist.

6. Konstruieren Sie ein (Markov-) Programm, welches das m-fache einer natürlichen Zahl n (|-Folge der Länge n+1) berechnet.

Digitale Informationsdarstellung

7. Ergänzen Sie die fehlenden Angaben in nachfolgender Tabelle.

Basis 2	Basis 5	Basis 8	Basis 10	Basis 16
110101111				
	43231			
		35127		
			734219	
				B6AC8E
0,000101				
	0,0314			
		0,123		
				0,7AB

8. Codieren Sie die Zahl einhundertsiebenunddreißig Fünftel zur Basis 5, 7 und 15.

9. Multiplizieren Sie im jeweiligen Zahlensystem und stellen Sie die Gegenprobe durch entsprechende
Division:
Basis 2: 101,011 * 0,1011 ; Basis 7: 14,03 * 2,6 ; Basis 8: 5,7 * 3,2 ; Basis 16: A9,0E * 12,CD.

10. Stellen Sie die Dezimalzahlen x = 71 und y = -52 im Bereich achtstelliger Binärcodes als
Zweierkomplement dar, und berechnen Sie zur Basis 2 die Ausdrücke x+y und x-y .

11. Geben Sie die normierte Gleitkommadarstellung der Zahl ein Dreiundzwanzigstel auf vier Stellen
genau jeweils zur Basis zwei, drei und acht an.

12. Wieviele Druckseiten können auf a) einer 1,4 MB Diskette
und b) einer 1,2 GB CD
gespeichert werden, wenn jedes Zeichen durch ein Byte kodiert wird und eine Seite 40 Zeilen zu je
60 Anschlägen umfaßt.

Klassischer Digitalrechner

13. Führen Sie das in der Vorlesung entworfene Programm zur Berechnung der Fakultätsfunktion für
n = 4 manuell aus und protokollieren Sie die Veränderungen der Register- bzw. Speicherinhalte.

14. Entwerfen Sie ein Programm zur Berechnung des größten gemeinsamen Teilers zweier natürlicher
Zahlen für
a) eine Einadreßmaschine,
b) eine Dreiadreßmaschine.

15. Entwerfen Sie ein Programm für eine Fünfadreßmaschine zu dem Verfahren aus Aufgabe 2.

Daten- und Steuerstrukturen

16. Entwerfen Sie eine Datenstruktur Labyrinth, die zur Beschreibung des Theseus-Algorithmus verwendet werden kann.

17. Geben Sie eine Datenstruktur für den Buchkatalog einer Bibliothek an und definieren Sie dazu geeignete Operationen zur Beschreibung des Verfahrens aus Aufgabe 1.

18. Welche Strukturen und Wertmengen werden durch folgende Typen beschrieben:

```
type Eins = ( Dresden, Erfurt, Magdeburg, Potsdam, Schwerin);
     Zwei = 1 .. 5;
     Drei = record  s : Eins, t : string [10], z : Zwei;
     Vier = array [0 .. 9] of Drei;
     Fünf = pointer to Element
            Element = record a: Vier, b: Fünf;
```

19. Entwerfen Sie eine Datenstruktur zur Verwaltung von Übungsgruppen mit maximal 25 Studenten nach Studiengängen getrennt unter Angabe von Name, Vorname und Immatrikulationsnummer der Studenten.

20. Welche der folgenden Zeichenfolgen sind Ausdrücke?
Wenn ja, von welcher Art. Wenn nein, aus welchem Grund.

a) $(x < y) = (x + 2)$
b) $((x - 3) * (y - 1)$
c) $(x \wedge y + 3)$
d) $(x - (y * z)) = 2$

21. Berechnen Sie den Wert der folgenden Ausdrücke bei den angegebenen Belegungen
b(x, y, z) = (2, 1, 0), b(x, y, z) = (0, 1, 2), b(x, y, z) = (1, 0, 1) !

a) $(x > y) = \neg (y \geq z)$
b) $((x - 1) < ((y + 1) * z))$
c) $(x \geq (y + 1 - x)) \wedge (\neg (0 < y) \vee (y \geq 0))$

22. Welche Belegungsänderungen werden von den folgenden Anweisungen erzeugt?

a) **begin** S **:=** 0 **; for** i **:=** 1 **to** n **step** 1 **do if** $x_i > 0$ **then** S **:=** (S + x_i) **end**
b) **while** i $\leq$ n **do begin** t := (t * x) ; i := (i + 1) **end**

23. Geben Sie Anweisungen an, die die durch folgende Funktionen beschriebenen Belegungsänderungen geeigneter Variablen erzeugen!

a) Größter gemeinsamer Teiler von zwei natürlichen Zahlen
b) Skalarprodukt zweier Vektoren
c) Stellenwertkodierung römischer Zahlwörter zur Basis 10

24. Bestimmen Sie die Lebenszeit und die Gültigkeitsbereiche der Variablen in dem nachfolgenden Programm:

```
program  P ;
    var m : integer ;

    procedure  R  ( n : integer ) ;
        begin
        if  n > 0  then  R ( n-1 )
        end ;

    begin
          R ( 2 )
    end
```

25. Geben Sie ein Struktogramm und ein Programm zur Konvertierung von Codewörtern zwischen verschiedenen Zahlenbasen an.

26. Geben Sie eine Prozedur an, die die Knoteninhalte (Typ integer) eines geordneten Baumes in der gegebenen Ordnung Ord in eine Liste stellt. [Ord (Vater) < Ord (Kind)]
Benutzen Sie diese Prozedur für ein Programm zur Ausgabe dieser Liste.

27. Schreiben Sie eine iterative und eine rekursive Prozedur zur Multiplikation zweier Matrizen von Integer-Objekten und verwenden Sie diese Prozeduren zum Ausdrucken (write) der Ergebnismatrix.

28. Formen Sie die folgende rekursive Prozedur in eine iterative Prozedur um.

```
procedure p ( i : integer) ;
        begin
            if  Bed ( i )          {von i abhängige Bedingung}
            then begin
                    i := Aus1(i) ;      {Ausdruck in i}
                    p ( i )
                end
             else  i := Aus2(i)             {Ausdruck in i}
        end ;
```