Universität Trier

Fachbereich IV - Informatik

Christoph Meinel: Informatik I

Vorlesungsskript WS 1993/94

LaTeX-Satz: Joachim Marder, Kerstin Faber, Johannes Lehnert

Inhaltsverzeichnis

Was ist Informatik

> "Informatik ist die Wissenschaft von der systematischen Verarbeitung von Informationen, besonders der automatischen Verarbeitung mit Hilfe von Computern."[1]

Die Informatik kann eigentlich keiner der drei Wissenschaftsformen (Naturwissenschaft, Geisteswissenschaft, Ingenieurwissenschaft) zugeordnet werden. Ein geeigneter Begriff wäre "Strukturwissenschaft", der von C.F. von Weizäcker geprägt worden ist, da es in der Informatik häufig um Untersuchung von Sprachstrukturen, Programmstrukturen, Datenstrukturen, Rechnerstrukturen, o.ä. geht. Die Methoden der Informatik sind:

- formal-mathematisch
- technisch-ingenieurwissenschaftlich
- praktisch-experimentell.

Bis etwa 1960 war die Informatik Spezialgebiet anderer Disziplinen, wie etwa der Logik, Mathematik, E-Technik. Inzwischen hat sie sich zu einem zusammenhängenden, theoretisch fundierten Gebäude einer neuen Grundlagenwissenschaft entwickelt.

Hinausgehend über das Programmierhandwerk und die Rechnerbenutzung geht es um Fragen der:

- Programm-Technik (Software-Engineering)
- Software-Projektmanagement
- abstrakte Maschinen- und Berechnungsmodelle
  - Grenzen der Berechenbarkeit
  - Komplexitätsanalyse
- parallele und verteilte Rechner
- Datenbanken, künstliche Intelligenz
- Logik, Zahlentheorie, Operations Research, Kodierungstheorie, ...

[1] aus: Duden Informatik

1 Algorithmen – der zentrale Begriff der Informatik

1.1 Der Algorithmenbegriff

1.1.1 Definition

Unter einem *Algorithmus* versteht man eine Verarbeitungsvorschrift, die so präzise formuliert ist (möglicherweise in einer formalen (Programmier-)Sprache), daß sie "mechanisch", z.B. von einem Computer, ausgeführt werden kann. Aus der Präzision der sprachlichen Darstellung des Algorithmus muß die Abfolge der einzelnen Verarbeitungsschritte (= Elementaroperationen) eindeutig hervorgehen. Die Abarbeitung eines Algorithmus bezeichnet man als *Prozeß* , das Ausführungsorgan als *Prozessor*.

Beispiel: Alltagsalgorithmen

Prozeß	**Algorithmus**	**Elementaropertionen**
Pullover stricken	Strickmuster	eine links, eine rechts eine fallen lassen
Kuchen backen	Rezept	nimm 3 Eier ...
Modellschiff bauen	Bauanleitung	befestige Teil A an Teil B
Konzert	Partitur	Noten

Ein Algorithmus gibt an, wie Eingabedaten schrittweise umgewandelt werden in Ausgabedaten, d.h. ein Algorithmus beschreibt eine Funktion: $f : E \to A$ wobei E die zulässigen Eingaben und A die zulässigen Ausgaben sind.

Bemerkung: Nicht jede Abbildung läßt sich durch einen Algorithmus realisieren. (↑Vorlesung Berechnungstheorie)

1.1.2 Charakteristische Eigenschaften von Algorithmen

Abstrahierung: Allgemein löst ein Algorithmus eine *Klasse* von Problemenl; die Auswahl eines einzelnen Problems erfolgt über *Parameter*.

Determiniertheit: Algorithmen sind im allgemeinen determiniert, d.h. mit gleichen Eingabedaten und gleichem Startzustand wird stets ein gleiches Ergebnis geliefert. (↑nichtdefinierter Algorithmus, ↑Komplexitätstheorie)

Finitheit: Die Beschreibung eines Algorithmus besitzt endliche Länge. Die bei der Abarbeitung eines Algorithmus entstehenden Datenstrukturen und Zwischenergebnisse sind endlich.

Terminierung: Algorithmen, die nach endlich vielen Schritten ein Resultat liefern heißen terminierend. Meist sind nur terminierende Algorithmen von Interesse. Ausnahme: z.B. Betriebssysteme.

Determinismus: Ein Algorithmus heißt deterministisch, falls zu jedem Zeitpunkt seiner Ausführung höchstens eine Möglichkeit zur Fortsetzung besteht. (↑nichtdeterministische Algorithmen, ↑statische Algorithmen)

Beispiel: Euklidischer Algorithmus

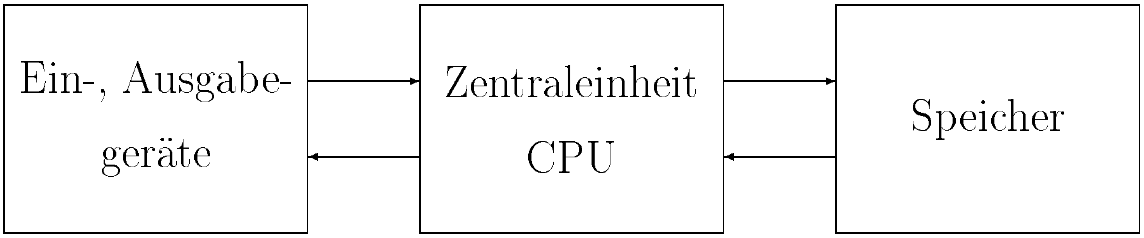
Aufbau: Seien $a, b \in \mathbb{N}, b \neq 0$. Bestimme $ggT(a, b)$

Algorithmus:

1. Falls $a = b$ brich Berechnung ab, es gilt $ggT(a, b) = a$. Sonst gehe zu Schritt 2.
2. Falls $a > b$ ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Sonst gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

1.2 Computer – bevorzugtes Algorithmenausführungsorgan

1.2.1 typischer Aufbau eines Computers



Ein-, Ausgabegeräte (= input/output-devices): Sie ermöglichen die Eingabe des Algorithmus mit der Eingabe der Daten in den Hauptspeicher und Ausgabe der erzielten Resultate.

Zentraleinheit (= central processing unit = CPU): Sie führt die Basisoperationen aus.

Speicher (= memory) Er enthält:
- ausführbare Operationen des Algorithmus
- Daten, mit denen der Algorithmus ausgeführt werden soll

Diese Komponenten bilden die *Hardware*, die physikalischen Einheiten, aus denen ein Computer besteht.

1.2.2 Hervorstehende Merkmale von Computern

Geschwindigkeit: Komplexe Algorithmen können schnell ausgeführt werden. Unterscheidung in praktisch durchführbare und undurchführbare Prozesse in der ↑Komplexitätstheorie.

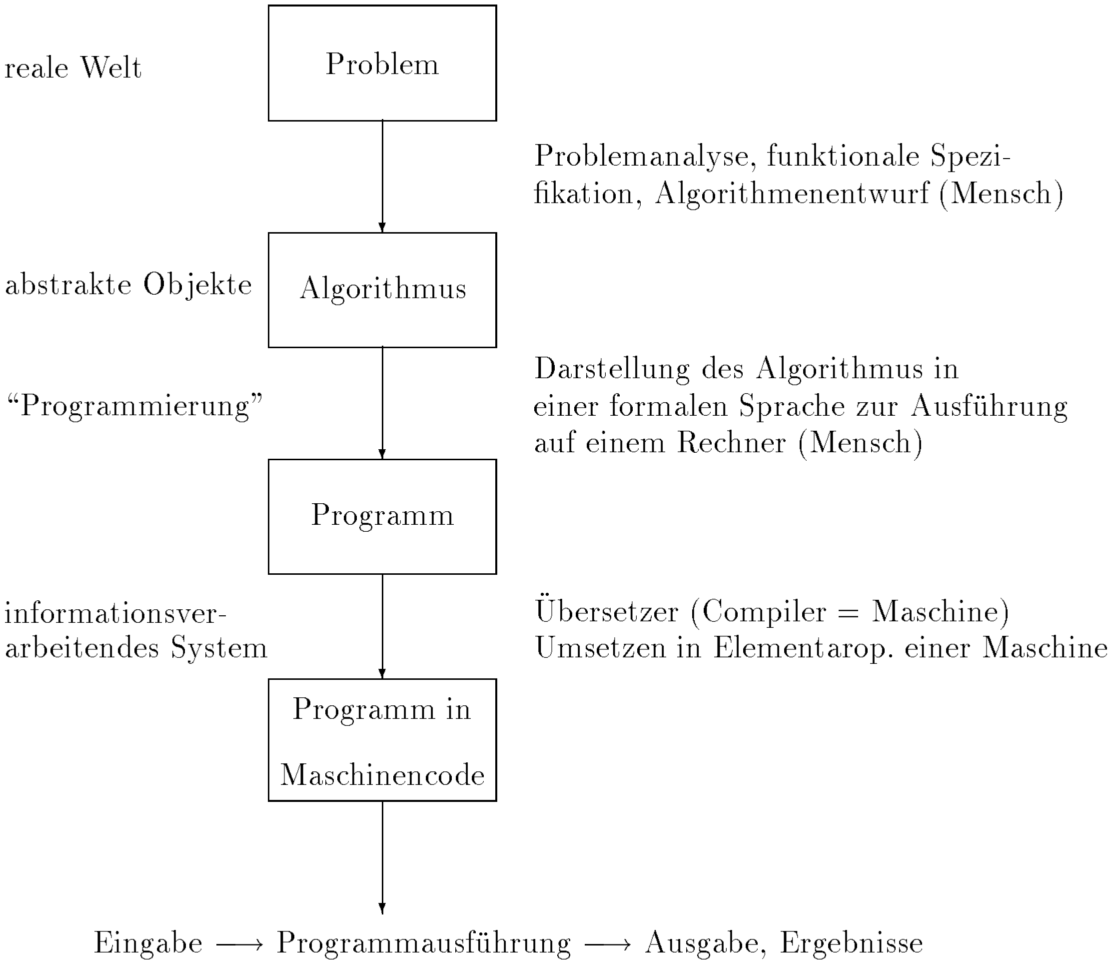
Zuverlässigkeit: Computer machen selten Fehler. Auftretende Fehler werden im allgemeinen durch Irrtümer in den auszuführenden Algorithmen bzw. durch fehlerhafte Eingabedaten verursacht.

Speicherkapazitäten: Der Computer besitzt die Fähigkeit, riesige Datenmengen zu speichern und schnell auf sie zugreifen zu können.

Kosten: Die niedrigen laufenden Kosten sind ausschlaggebend für viele Computeranwendungen.

Algorithmen lassen sich darstellen durch die Angabe eines (mathematisch exakt definierten) *Maschinenmodells* ("virtuelle Maschine"), die den Algorithmus schrittweise ausführt. Einfachste universelle Maschinenmodelle sind: Turingmaschine (1936) und Registermaschine. Bei diesen Maschinenmodellen werden vereinfachende Annahmen getroffen, so z.B. daß unbegrenzt Speicher zur Verfügung stehen. Zur Definition eines Maschinenmodells genügt die Angabe der Grundbefehle (Elementaroperationen), der zulässigen Speicherbereiche sowie die Angabe der Kontrollstrukturen zur Festlegung der erlaubten Verknüpfung von Grundbefehlen.

1.3 Problem → Algorithmus → Programm



1.3.1 Schritte bei der Algorithmen- und Programmentwicklung

Der Entwurf von Algorithmen sollte unabhängig sein von Programmiersprache und Rechner. Folgende Schritte sind bei der Algorithmen- und Programmentwicklung durchzuführen:

1. Problemanalyse, Problemspezifikation, Problemabstraktion
2. Algorithmenentwurf
3. Korrektheitsnachweis, Verifikation
4. Aufwandsanalyse
5. Programmierung

1.3.2 Prinzipbeispiel

1. *Problemformulierung*
   "Finde die jüngste Person im Raum!"

2. (a) *Problemanalyse*
   - Wie sieht die Lösung aus?
   - Was ist bekannt?
   - Gibt es eine Lösung?
   - Gibt es genau eine Lösung?

   $\vdots$

   (b) *Problemabstraktion*

- Was ist "Alter"?	$\Rightarrow$	positive ganzen Zahlen
- Was ist "jüngste"?	$\Rightarrow$	Ordnungsrelation definiert auf ganze Zahl
- Was heißt "finde"?	$\Rightarrow$	Altersangaben müssen bekannt sein, also eingelesen werden
- Was heißt "Person"?	$\Rightarrow$	Altersangaben $a_1, \ldots, a_n$ die mit den Personen 1,...,n identifiziert sind

   (c) *Problemspezifikation*

   gegeben: Folge $a_1, \ldots, a_n$ von positiven ganzen Zahlen

   gesucht: kleinster Positionsindex j mit: $a_j = \min\{a_1, \ldots, a_n\}$
   ("funktionale Spezifikation")

3. *Algorithmenentwurf*
   Hierunter versteht man die Ableitung eines Verfahrens aus der Spezifikation. Es gibt viele Lösungsmöglichkeiten, erwünscht sind vor allem folgende Eigenschaften: elegant, kurz, verständlich, transparent, schnell, leicht modifizierbar. Oft sind jedoch nicht alle diese Eigenschaften miteinander vereinbar.

Algorithmus *MinAlter*:

> Setze $j := 1$ und $x := a_j$
> (* Das ist die Lösung falls $n = 1$ *)
>
> (* Suchlauf *)
> Setze $i := 2$
> Solange $i \leq n$ gilt wiederhole:
>   falls $a_i < x$, setze $j := i$ und $x := a_i$
>   erhöhe i um 1
> Das Ergebnis ist j.

Bei dem *Prinzip der schrittweisen Verfeinerung* werden in der Entwurfsphase zuerst die groben Strukturen der Algorithmen aufgeschrieben, die dann nach und nach verfeinert werden. Es werden dabei unter anderem folgende Ziele verfolgt:

- Reduzierung der Komplexität des Entwurfprozesses
- bessere Aussicht auf korrekten Algrithmus

Bei der *Teilproblembestimmung* wird das Problem in einzelne Teilprobleme zerlegt, die dann getrennt gelöst werden können. (↑Modularisierung)

4. *Korrektheitsnachweis, Verifikation*
   - Terminiert der Algorithmus?
   - Liefert er das gewünschte Resultat?

Behauptung: Der Algorithmus *MinAlter* terminiert und liefert das gewünschte Resultat.

Beweis: Vollständige Induktion über n.

I.Behauptung: $x = \min\{a_1, \ldots, a_n\} = a_j$ wobei j minimal ist und der Algorithmus nach $n-1$ Durchläufen der Schleife terminiert.

I.Anfang: $n = 1$. Beweis durch Simulation:

(1) $j := 1$ und $x := a_1$

(2) $i := 2$. Schleife läuft nicht, weil $i \leq n$

(3) $j := 1 \quad \Rightarrow$ Induktionsbehauptung gilt

I.Schritt: $n \rightarrow n+1$
Induktionsvoraussetzung: Behauptung gilt für n.
Sei $n' = n + 1$.

(1) $j := 1$ und $x := a_j$.

(2) $i := 2$
Nach $(n-1)$ Durchläufen gilt nach Induktionsvoraussetzung $x = \min\{a_1, \ldots, a_n\} = a_j$.
Letzter Durchlauf: i erreicht n' und der Algorithmus terminiert nach dem $(n'-1)$-ten Durchlauf. Falls $a_{n'} \geq x$,

dann gilt die Induktionsbehauptung für n. Sonst erhalten j und x neue Werte und die Induktionsbehauptung gilt für n'.
(j ist minimal, da j und x nur neue Werte bekommen, falls $a_i < x$)

5. *Aufwandsanalyse*
   Die Aufwandsanalyse untersucht den Zeit- und evtl. den Speicherplatzbedarf des Algorithmus. Die Zeit wird gemessen in der Zahl der auszuführenden Elementaroperationen (E).

Operation	*Aufwand*	*Zahl der Wiederholungen*
setze $j := 1, x := a_j$	2E	1
setze $i := 2$	1E	1
teste $i \leq n$	1E	n
teste $a_i < x$	1E	$n - 1$
setze $j := i, x := a_j$	2E	$n - 1$ maximal, 0 minimal
erhöhe i um 1	1E	$n - 1$

Günstigster Fall: $T_{min}(n) = 3 + n + 2(n-1)$ E

Schlechtester Fall: $T_{max}(n) = 3 + n + 4(n-1)$ E

Mittlerer Fall: $T_{mittel}(n) = 3n \text{ E} + \sum_{i=2}^{n} \frac{1}{i}$ E

Beweis: (Mittlerer Fall)

Annahme: Alle Permutationen von $(a_1, \ldots, a_n)$ sind gleich wahrscheinlich, alle a_i sind verschieden.

Sei A_j die Anzahl der a_i $(1 \leq i \leq j)$ für die gilt: "$a_i < x$" bei j Eingaben $(a_1, \ldots, a_j)$.

Zu zeigen: $A_j = \sum_{i=2}^{j} \frac{1}{i}$ für alle $j < n$

Vollständige Induktion über j:

I.Anfang: $j = 2$
Es gibt die zwei Fälle $a_1 < a_2$ oder $a_2 < a_1$ die beide gleichwahrscheinlich sind. Es gilt also: $A_2 = 0 \cdot \frac{1}{2} + 1 \cdot \frac{1}{2} = \frac{1}{2}$

I.Schritt: $j \to j + 1$
$\underbrace{a_1, \ldots, a_j}_{A_j}, a_{j+1}$ seien zufällig gewählt.
$\Rightarrow A_j = \sum_{i=2}^{j} \frac{1}{i}$ (nach Induktionsvoraussetzung)
a_{j+1} ist minimal unter den $a_1, \ldots, a_{j+1}$ mit einer Wahrscheinlichkeit von $\frac{1}{j+1}$.

$$\begin{aligned} \Rightarrow A_{j+1} &= A_j(1 - \frac{1}{j+1}) + (A_j + 1)\frac{1}{j+1} \\ &= A_j + \frac{1}{j+1} \\ &= \sum_{i=2}^{j} \frac{1}{i} + \frac{1}{j+1} \\ &= \sum_{i=2}^{j+1} \frac{1}{i} \end{aligned}$$

Wegen $\ln j \ \leq \ \sum_{i=1}^{n} \frac{1}{i} \ \leq \ \ln j + 1$ gilt:

$(3n - \ln n - 1)$ E $\ \leq \ T_{mittel}(n) \ \leq \ (3n + \ln n)$ E

Bemerkung: Die Verschiedenheitsannahme ist problematisch, denn in der Praxis können die a_i auch gleich sein. Außerdem ist nicht garantiert, daß jede Permutation in der Praxis immer gleich wahrscheinlich ist. Die idealisierten Annahmen sind jedoch die Voraussetzung, um eine theoretische Analyse überhaupt ausführen zu können. (↑ingenieurwissenschaftlicher Charakter der Informatik)

6. *Programmierung*
   - Wie kommen die Daten in den Rechner?
   - Welche Programmiersprache steht zur Verfügung?
   - Wie wird das Ergebnis ausgegeben?
   - Wie wird das Programm gegliedert?

   Der Algorithmus *MinAlter* ist beispielhaft in MODULA–2 implementiert. Die Operation MinIndex ist dabei abgespalten worden, um sie auch in anderen Programmen einsetzen zu können. Die Altersangaben müssen eingegeben werden und die Ausgabe erfolgt in verständlichem Text:

```
MODULE JuengstePerson;
FROM Inout IMPORT ReadCard, WriteCard, WriteString, WriteLn;
CONST max = 100; (* maximale Personenzahl *)
TYPE Folge = ARRAY [1..max] OF CARDINAL;
VAR a: Folge;
j,i,n: CARDINAL;

PROCEDURE MinIndex(n: CARDINAL; VAR a: Folge): CARDINAL;
(* liefert den Minimalindex im Feld a[1] ... a[n] *)
VAR i,j,x: CARDINAL;
```

```
BEGIN
  j := 1; x := a[j];
  i := 2;
  WHILE i <= n DO (* Schleife *)
    IF a[i] < x THEN j:=i; x := a[j] END;
    i := i+1;
  END;
  RETURN j;
END MinIndex;

BEGIN (* Hauptprogramm *)
  REPEAT
    WriteString('Bitte Anzahl der Personen eingeben: ');
    ReadCard(n);
    WriteLn;
  UNTIL (n >= 1) AND (n <= max);
  i := 1;
  WHILE i<n DO
    WriteString('Bitte Alter der ');
    WriteCard(i,3);
    WriteString(. Person eingeben: );
    ReadCard(a[i]);
    WriteLn;
    i := i+1;
  END;
  j:=MinIndex(n,a);
  WriteLn; WriteLn;
  WriteString('Juengste Person ist Nr. ');
  WriteCard(j,3);
  WriteLn;
END JuengstePerson.
```

1.4 Ablaufsteuerung von Algorithmen

Algorithmen bestehen aus Elementarschritten, aber wie sind diese angeordnet? Die Ablaufsteuerung wird durch *Kontrollstrukturen* realisiert.

1.4.1 Sequenz (Folge von Anweisungen)

- Zu jedem Zeitpunkt wird nur ein Schritt ausgeführt
- Jeder Schritt wird genau einmal ausgeführt. Keiner wird wiederholt, keiner wird ausgelassen.
- Die Reihenfolge, in der die Schritte ausgeführt werden, ist die gleiche Folge, in der die Schritte niedergeschrieben sind. (d.h. nacheinander)

- Mit Beendigung des letzten Schrittes endet der Algorithmus

Mit Sequenz als einziger Kontrollstruktur können nur sehr unflexible und starre Algorithmen erhalten werden.

1.4.2 Selektion (bedingte Auswahl)

Die Kontrollstruktur der Selektion erlaubt die bedingte Ausführung von Anweisungen:

(a) *einfache Form*

falls *Bedingung*
  dann *Anweisung*

(b) *bedingte Form mit Alternative* (allgemeine Form)

falls *Bedingung*
  dann *Anweisung 1*
  sonst *Anweisung 2*

Die einfache Form ist ein Spezialfall der allgemeinen Form, wobei *Anweisung 2* die leere Anweisung ist.

(c) *Mehrfachauswahl*

falls *Bedingung 1* dann *Anweisung 1*
  Bedingung 2 dann *Anweisung 2*
  ⋮
  Bedingung n dann *Anweisung n*
  sonst *Anweisung n+1*

Die Bedingungen 1 bis n müssen sich gegenseitig ausschließen. Die eingerückte Schreibweise verdeutlicht, welche Algorithmenschritte zu welcher Auswahl gehören.

1.4.3 Iteration (Wiederholung, Schleife)

Unter einer Iteration versteht man die wiederholte Ausführung einer Folge von Anweisungen. Das Ende wird durch eine Abbruchbedingung bestimmt.

(a) wiederhole
  Anweisung (“Schleifenkörper”)
bis *Bedingung* (“Abbruchbedingung”)

Spezialfall: Endlosschleife

wiederhole *Anweisung*
immer

Eine Endlosschleife entsteht häufig unbeabsichtigt, weil die Abbruchbedingung nicht korrekt formuliert ist. Endlosalgorithmen können beabsichtigt sein, wenn sie über die Zeit ein stets wiederkehrendes Problem stets aufs neue lösen. Sie sind nicht korrekt, wenn sie unendlich lange brauchen, um ein Problem zu lösen.

(b) solange *Bedingung*
führe aus *Anweisung* ("Schleifenkörper")

Die *Anweisung* in der Schleife wird ausgeführt, solange *Bedingung* erfüllt ist.

Der Unterschied zwischen den beiden Varianten liegt darin, daß in (a) die *Bedingung* *nach* der Ausführung des Schleifenrumpfes geprüft wird, in (b) jedoch wird die *Bedingung* *vor* der Ausführung überprüft. (a) und (b) sind äquivalent, d.h. sie lassen sich mit bedingten Anweisungen ineinander überführen.

1.4.4 Beispiel für die Verknüpfung von Kontrollstrukturen: Bubble–Sort

Aufgabe: Sortiere eine Liste von n Namen in alphabetischer Reihenfolge (siehe auch Kapitel 2.4.9 Seite 48)

Algorithmus:
```
wiederhole
   beginne mit Listenanfang
   setze Wiederholungszähler auf 0
   wiederhole
      erhöhe Wiederholungszähler um 1
      falls Name alphabetisch vor dem des Nachfolger
         dann: Tausche Name und Nachfolger
         merke, daß Vertauschung vorkam
   bis Wiederholungszähler = n − 1
bis keine Vertauschung im Durchlauf vorkam
```

Beispiel: Eine Liste mit fünf Namen wird von Bubblesort folgendermaßen sortiert:

Ur-Liste	1. Durchlauf	2. Durchlauf	3. Durchlauf
Hans	Anna	Anna	Anna
Anna	Hans	Hans	Bernd
Susi	Susi	Bernd	Hans
Uwe	Bernd	Susi	Susi
Bernd	Uwe	Uwe	Uwe

Im 4. Durchgang wird keine Vertauschung mehr durchgeführt, so daß der Algorithmus terminiert.

1.4.5 Bedeutung von Selektion, Sequenz, Iteration

Satz:
Falls es möglich ist, einen Algorithmus zur Beschreibung einer bestimmten Prozedur zu entwerfen, dann kann ein solcher Algorithmus unter alleiniger Verwendung der Kontrollstrukturen der Sequenz, der Selektion und der Iteration entworfen werden, d.h. *Sequenz, Selektion und Iteration reichen aus, um jeden Algorithmus zu beschreiben.*

Der Beweis erfolgt in der ↑Berechnungstheorie.

Bemerkung: Die Hinzunahme weiterer Kontrollstrukturen erleichtert das Leben.

1.4.6 Die goto-Anweisung

goto G
$\vdots$
G: *Anweisung*

Die goto-Anweisung ist inhaltlich nicht nötig. Man verliert leicht die Übersicht bei Verwendung vieler goto-Anweisungen, d.h. der Algorithmus verliert an Klarheit. Außerdem verursacht sie technische Schwierigkeiten, z.B. bei Verwendung innerhalb von Schleifenkörpern.

Bemerkung: Vermeide goto-Anweisungen!

1.5 Informale und graphische Beschreibung von Algorithmen

Informal bedeutet unabhängig von einer bestimmten Programmiersprache.

1.5.1 Umgangssprache, stilisiertes Deutsch

Diese Art der Beschreibung geschieht unter Verwendung möglichst präziser Anweisungen und einer klaren Beschreibung der Ablaufsteuerung.

Beispiel: Euklidischer Algorithmus
Eingabe: Zwei natürliche Zahlen a, b, wobei $b \neq 0$ gilt
Algorithmus:

1. Falls $a = b$ brich Berechnung ab, es gilt $ggT(a, b) = a$. Sonst gehe zu Schritt 2.
2. Falls $a > b$ ersetze a durch $a - b$ und setze Berechnung in Schritt 1 fort. Sonst gehe zu Schritt 3.
3. Es gilt $a < b$. Ersetze b durch $b - a$ und setze Berechnung in Schritt 1 fort.

1.5.2 Pseudocode

Dies ist eine programmähnliche Beschreibung, die zum Teil unvollständig ist (ohne formale Ein-/Ausgabebeschreibung usw.).

Beispiel: Algorithmus $ggT(a, b)$
Objekte: $a, b \in I\!N, b \neq 0$

> solange $a = b$ führe aus
>   falls $A > b$
>   dann : $A := A - b$
>   sonst: $b := b - a$
> ggT:=a
> End.

1.5.3 Flußdiagramme

Dies ist eine weitgehend normierte Methode zur graphischen Darstellung von Programmabläufen. Flußdiagramme können folgende Elemente enthalten:

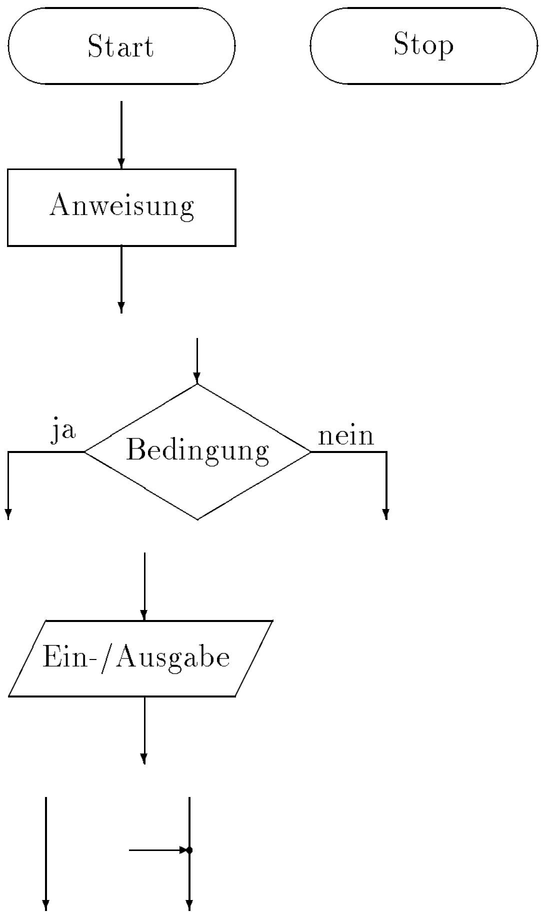
Einstiegs-/Ausstiegsstelle

elementare Anweisung

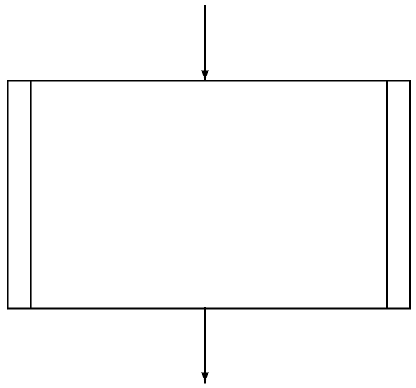
Selektion

Eingabe, Ausgabe

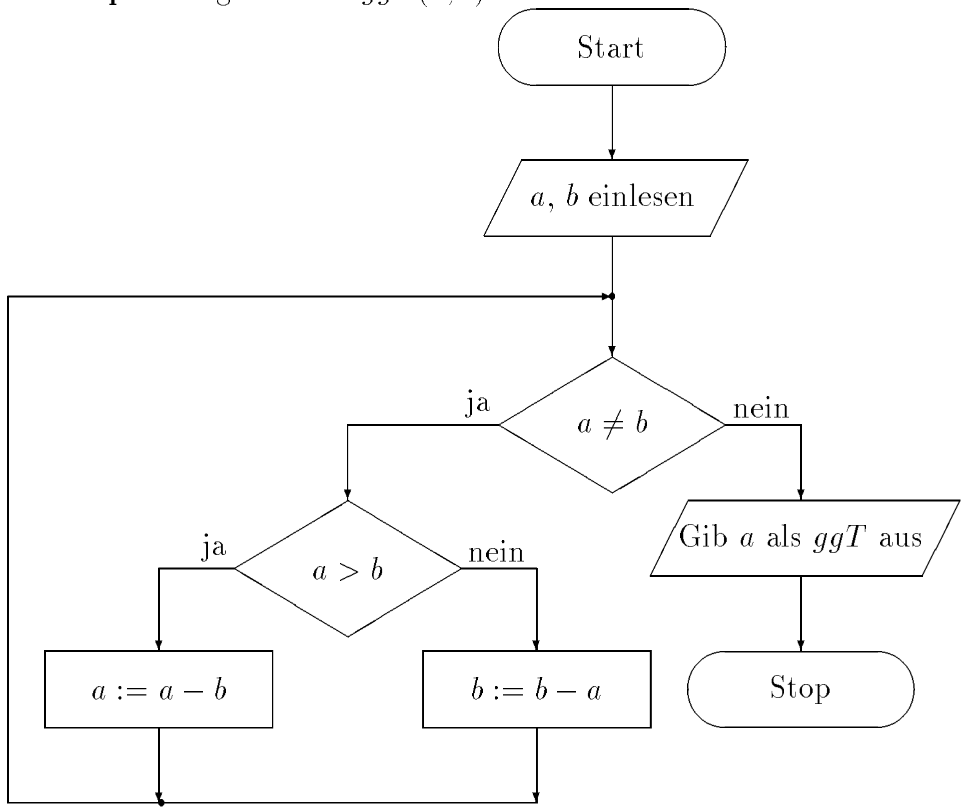
Ablauflinie, Zusammenführung



Unterprogramm



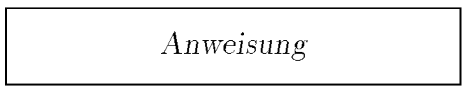
Beispiel: Algorithmus $ggT(a, b)$



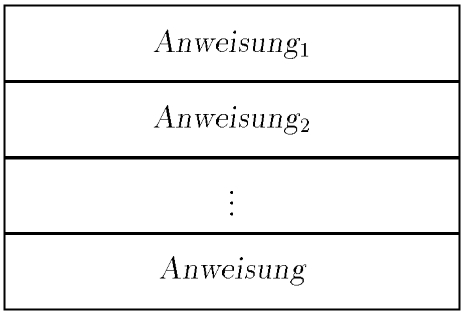
1.5.4 Struktogramme (Nassi-Schneidermann-Diagramme)

Struktogramme sind graphische Darstellungsmittel für Programme, bei denen jede einzelne Aktion in einen Strukturblock eingetragen wird. Es existieren folgende Strukturblöcke:

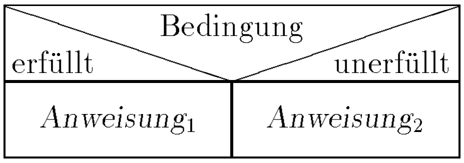
Elementaranweisung:



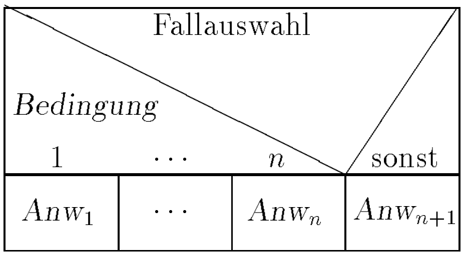
Sequenz:



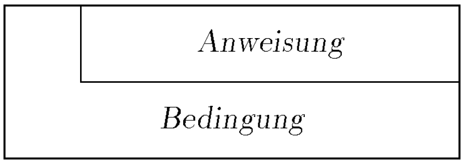
Selektion:



Selektion mit Mehrfachauswahl:



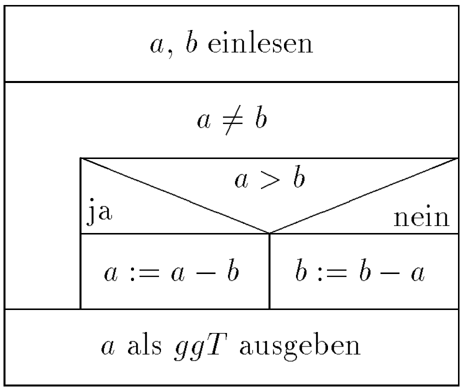
Iteration:



wiederhole *Anweisung*
bis *Bedingung*

solange *Bedingung* gilt
führe *Anweisung* aus

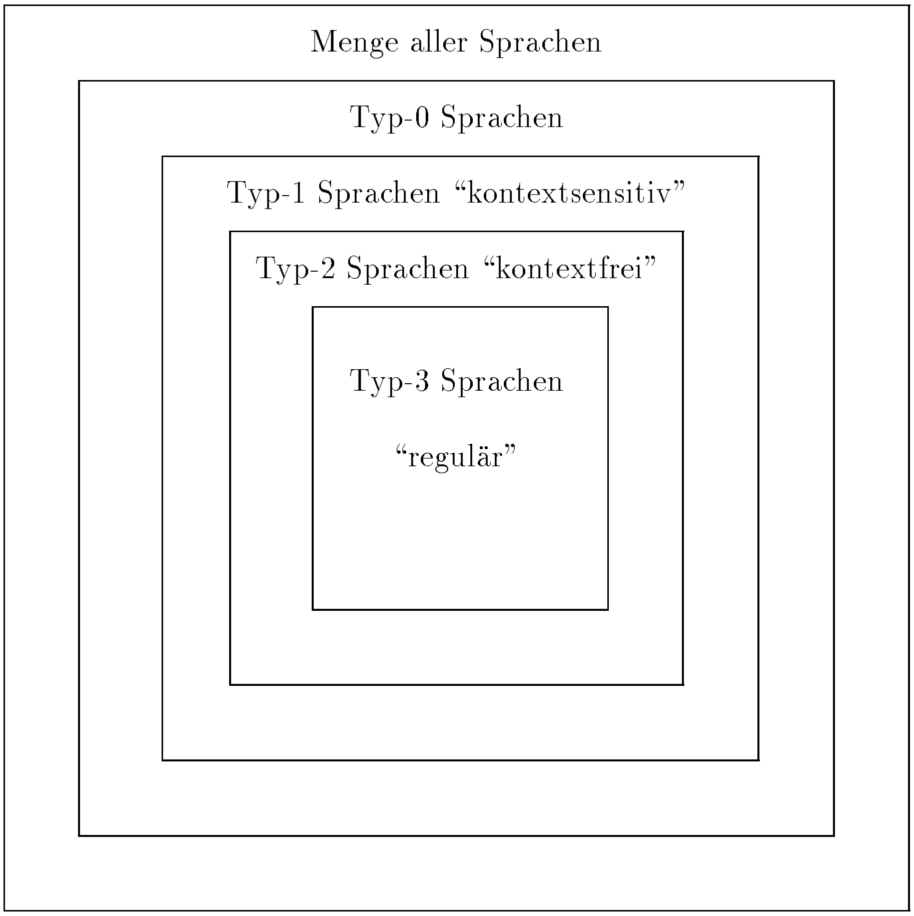
Beispiel: Algorithmus $ggT(a, b)$



1.6 Programmierung – Aufbereitung eines Algorithmus zur Ausführung durch einen Computer

Um einen Algorithmus durch einen Computer ausführen zu lassen, muß er in eindeutiger und formalisierter Form vorliegen. Dazu dienen *Programmiersprachen*, dies sind Sprachen zur Formulierung von Rechenvorschriften, d.h. von Algorithmen und Datenstrukturen, die von einem Computer ausgeführt werden können. Sie stellen die wichtigste Schnittstelle zwischen Benutzer und Computer dar. Der Benutzer muß die durch den Computer auszuführende Bearbeitung von Problemen in einer Programmiersprache formulieren. Zur Vermeidung von Mehrdeutigkeiten müssen *Syntax* (Grammatik) und *Semantik* (Bedeutung) einer Programmiersprache eindeutig definiert sein. Die meisten Programmiersprachen sind *kontextsensitiv* (↑Theorie der Automaten und formalen Sprachen), werden aber mit Hilfe von kontextfreien Grammatiken und umgangssprachlichen Zusätzen (z.B. "Jede Variable muß deklariert sein.") beschrieben.

Es existiert folgende Hierarchie der formalen Sprachen ("*Chomsky Hierarchie*"):



Je nach Nähe zur Hardware unterscheidet man:

- *Maschinensprachen* (direkte Programmierung der Hardware)
- niedere maschinenorientierte Programmiersprachen oder Assembler-Sprachen (Programmierung der Hardware mit symbolischen Namen)
- höherere oder problemorientierte Programmiersprachen (von der Hardware unabhängig, an den zu bearbeitenden Problemen orientiert)

Höhere Sprachen werden in niedere Sprachen mit Hilfe von *Übersetzern* (Compiler, Assemblierer) übertragen.

1.6.1 Syntax von Programmiersprachen

Die *Syntax* ("Lehre vom Satzbau") ist die formale Beschreibung des Aufbaus der Sätze und Wörter, die zu einer (Programmier-) Sprache gehören. Die Syntax einer Programmiersprache legt fest, welche Zeichenreihen korrekt formulierte Programme der Sprache sind und welche nicht. Zur Automatisierung des Tests auf syntaktische Korrektheit (↑Übersetzer) muß die Syntax formal beschrieben werden. Dazu gibt es u.a. folgende Methoden:

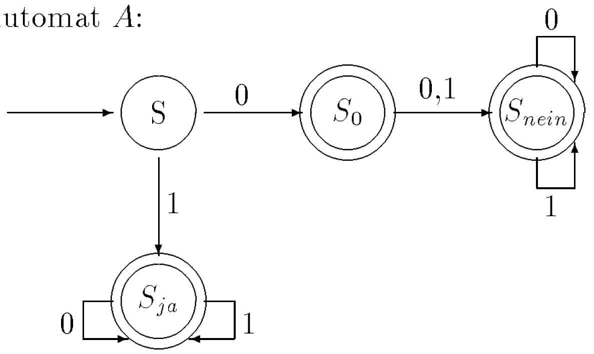
(a) Angabe eines die Syntax erkennenden endlichen Automaten

(b) Angabe einer die Sprache erzeugenden *Grammatik*

Beispiel: $L = \{\omega \in \{0,1\}^* | \omega$ ist eine Binärzahl ohne führende Nullen$\}$

(a) Endliche Automaten werden durch Zustandsüberführungsdiagramme beschrieben:

Automat A:



(◎ terminale Zustände; S_{ja}, S_0 akzeptierende Zustände, S_{nein} verwerfender Zustand)

z.B.: $\omega_0 = 110$: A terminiert bei ω_0 in $S_{ja} \Rightarrow\ \omega_0 \in L$
$\omega_1 = 0110$: A terminiert bei ω_1 in $S_{nein} \Rightarrow\ \omega_0 \notin L$

(b) Grammatik: Die von G erzeugte Sprache ist L.

$$G = (\underbrace{\{S, A\}}_{\text{"Variablen"}}, \underbrace{\{0,1\}}_{\text{"Terminalalphabet"}}, \underbrace{\mathrm{P}}_{\text{"Produktion"}}, \underbrace{\mathrm{S}}_{\text{"Startsymbol"}})$$

$$P = \{S \to 0,\ S \to 1A,\ A \to 1A,\ A \to 0A,\ A \to \varepsilon\}$$

z.B.: $\omega_0 = 110 : S \to 1A \to 11A \to 110A \to 110 \Rightarrow\ \omega_0 \in L$
$\omega_1 = 0110$: $S \to 0$ läßt sich nicht ableiten $\Rightarrow\ \omega_0 \notin L$

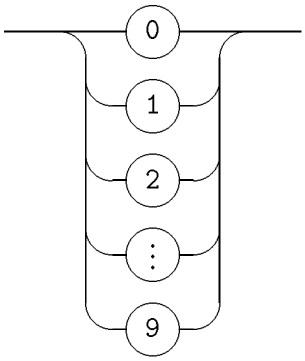
Die Syntax höherer Programmiersprachen wird mit Hilfe von *Syntaxdiagrammen* bzw. in *(erweiterter) Backus–Naur–Form* beschrieben.

1.6.2 Syntaxdiagramme

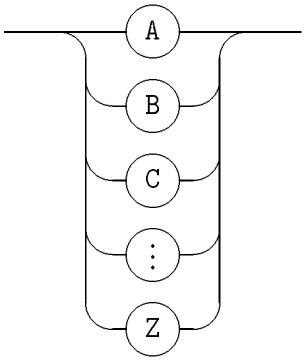
Das Syntaxdiagramm ist eine Technik zur graphischen Darstellung kontextfreier Grammatiken. Die Grundbausteine sind: *rechteckige* und *runde Kästen*, die *Beschriftungen* enthalten und durch *Pfeile* miteinander verbunden sind. Jedes Diagramm erhält eine Bezeichnung (frei wählbarer Name) und enthält genau einen hineingehenden und einen hinausgehenden Pfeil. Bezeichnungen in einem rechteckigen Kasten beziehen sich auf ein jeweils anderes Diagramm mit dieser Bezeichnung, das an der Stelle des rechteckigen Kastens eingesetzt zu denken ist. Bezeichnungen in einem runden Kasten sind Text der formalen Sprache, der an der betreffenden Stelle vorgesehen ist. Syntaktisch korrekter Text wird erhalten, indem das Syntaxdiagramm vom Eingangspfeil zum Ausgangspfeil in einer möglichen Weise durchlaufen wird und dabei alle Texte in runden Kästen, auf die man trifft, notiert werden.

Beispiele: (MODULA-2)

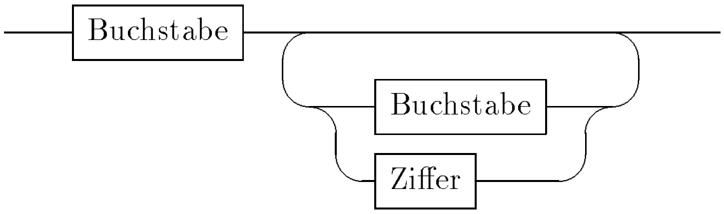
Ziffer



Buchstabe

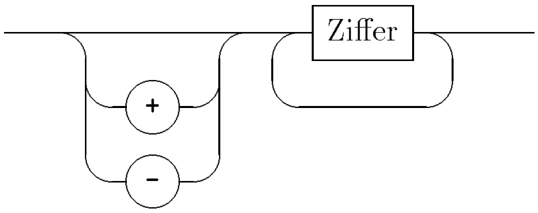


Bezeichner

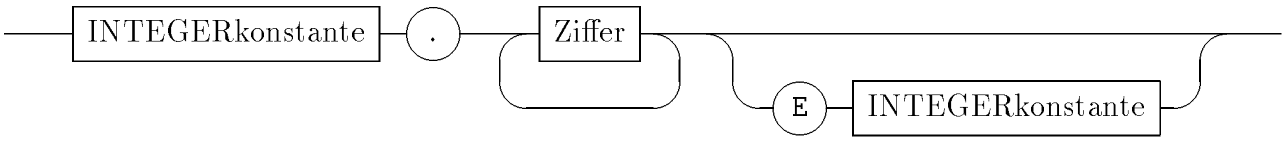


syntaktisch korrekte Bezeichner: `X, RST, VZ, D1543AB,` ...
syntaktisch nicht korrekte Bezeichner: `5, 4AR532,` ...

INTEGERkonstante



REALkonstante

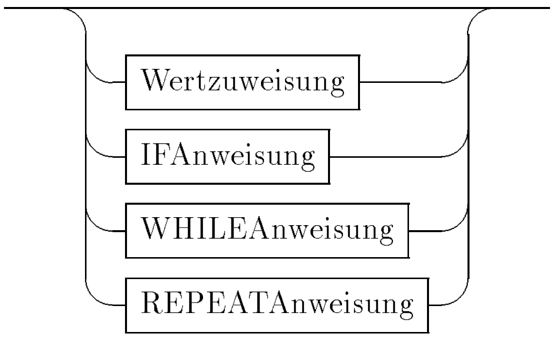


syntaktisch korrekte `REAL`-Konstanten: `0.0, 4.2E-18, -576.0E-111,` ...
syntaktisch nicht korrekte `REAL`-Konstante: `326, .22EA,` ...

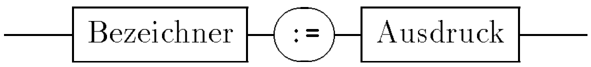
Syntaxdiagramme können auch rekursiv sein, d.h. im Diagramm mit dem Bezeichner X (bzw. in einem von X indirekt angegebenen Diagramm) kommt X selbst wieder vor.

Beispiele: (MODULA-2)

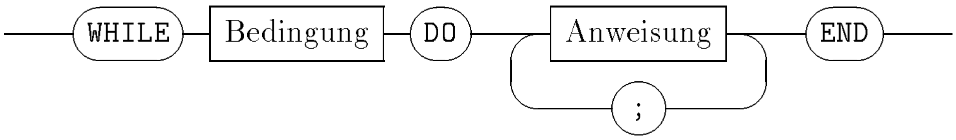
Anweisung



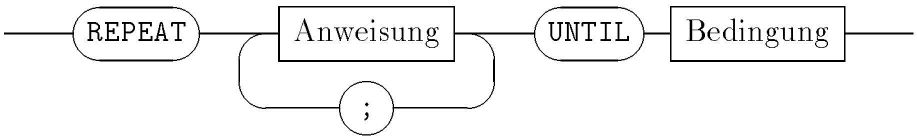
Wertzuweisung



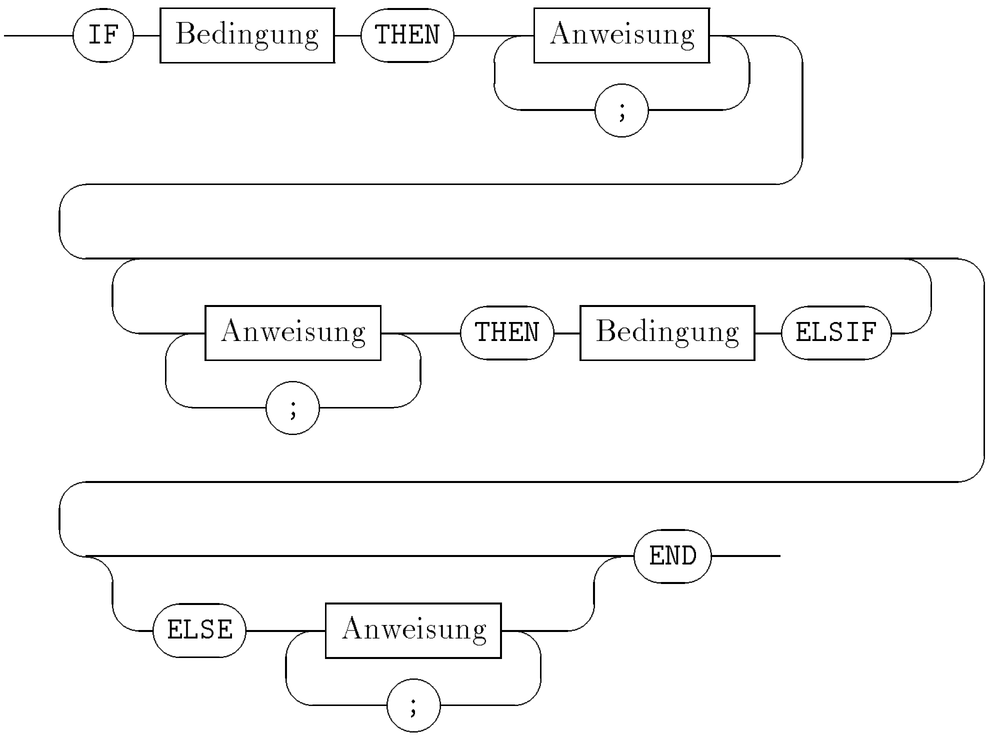
WHILE-Anweisung



REPEAT-Anweisung



IF-Anweisung



Beispiele für syntaktisch korrekte Anweisungen:

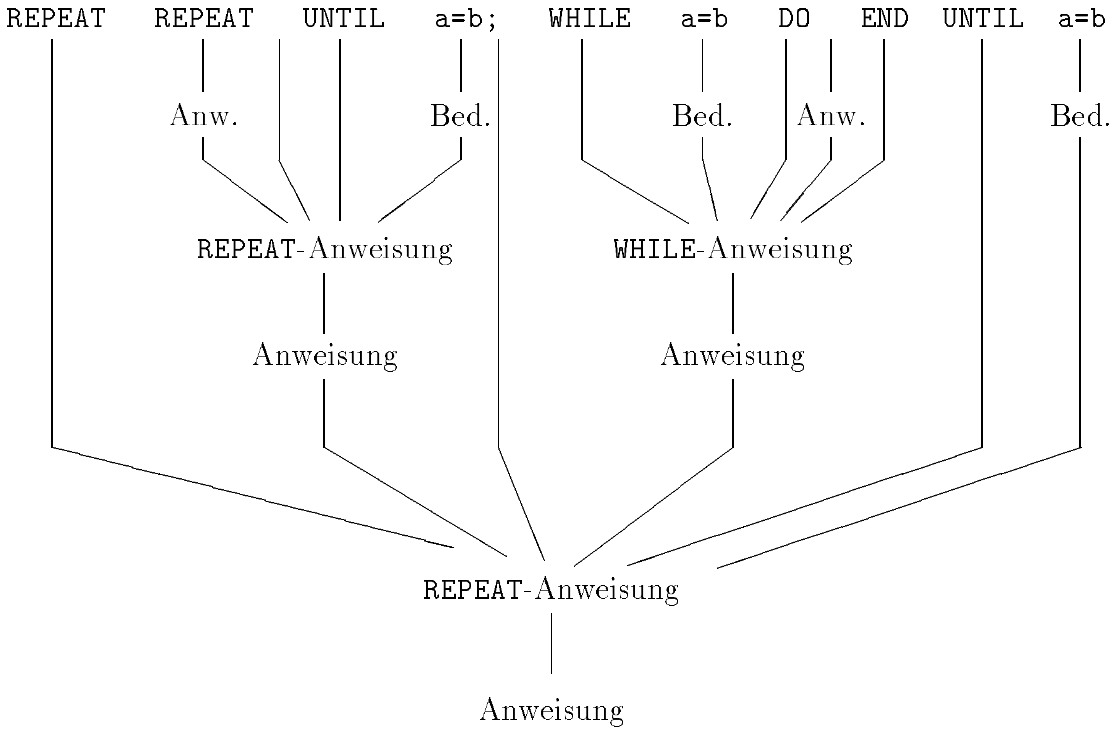
(a) `IF a=b THEN WHILE a=b DO a:=b; REPEAT a:=b; c:=e; UNTIL a=b; END; ELSIF a=b THEN END`

(b) `REPEAT REPEAT UNTIL a=b; WHILE a=b DO END UNTIL a=b`

(c) `WHILE a=b DO IF a=b THEN a:=b END END`

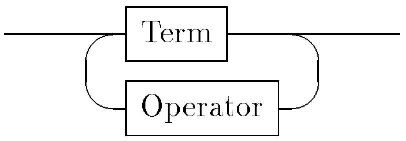
Bemerkungen:
- `a=b` steht für eine Bedingung;
- Man beachte das Auftreten "leerer" Anweisungen;
- Jedes Beispiel liefert Anweisungen, die "im inneren" weitere Anweisungen enthalten können;

Ableitungsbaum für (b):

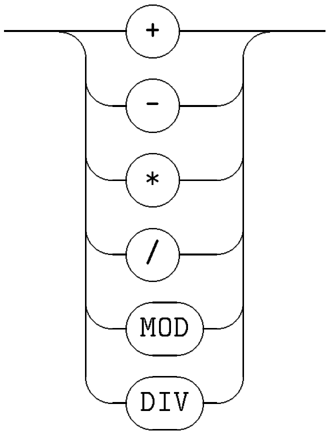


Beispiel: Ausdruck (vereinfacht)

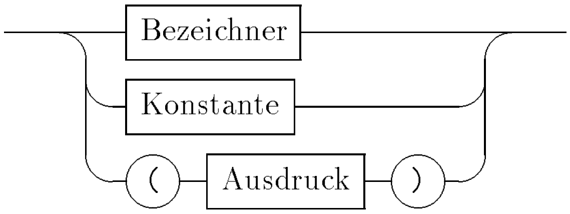
Ausdruck



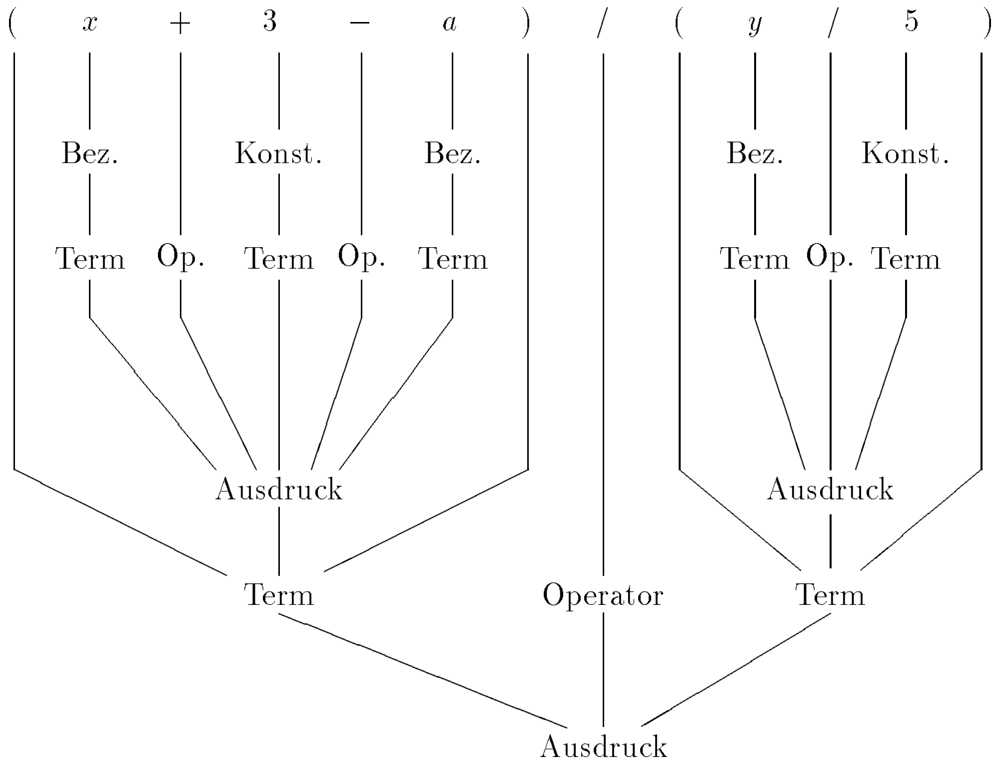
Operator



Term

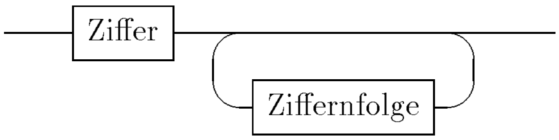


Beispiel für einen syntaktisch korrekten Ausdruck: $(x + 3 - a)/(y/5)$
Ableitungsbaum:

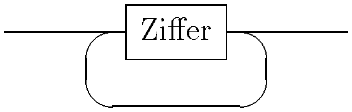


Manche rekursiven Diagramme lassen sich in äquivalente iterative Diagramme überführen. Beispiel:

Ziffernfolge (rekursiv):

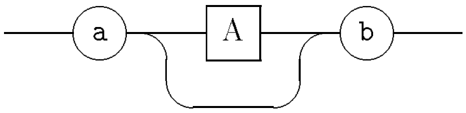


Ziffernfolge (iterativ):



Aber: Nicht alle rekursiven Diagramme können in iterative umgewandelt werden.
Beispiel: Die Sprache $L := \{a^n b^n, n \geq 1\}$

A



Dies ist die Definition einer Sprache L, die aus gleich vielen a's und b's besteht. Die Sprache L ist kontextfrei, aber nicht regulär.

Es läßt sich folgendes feststellen:

Fakt 1: Reguläre Sprachen (Typ 3) sind genau die Sprachen, die sich durch Syntaxdiagramme ohne Rekursion beschreiben lassen.

Fakt 2: Kontextfreie Sprachen (Typ 2) sind genau die Sprachen, die sich durch (rekursive) Syntaxdiagramme darstellen lassen. (↑Automatentheorie und formale Sprachen)

Das Äquivalenzproblem für Syntaxdiagramme ist *unentscheidbar*, d.h. es gibt keinen Algorithmus der entscheidet, ob zwei vorgegebene Diagramme dieselbe Sprache beschreiben. (↑Berechnungstheorie)

1.6.3 (Erweiterte) Backus-Naur-Form

Bei der (erweiterten) Backus-Naur-Form (EBNF) werden die Regeln in folgender Form angegeben: ⟨*linke Seite*⟩ ::= rechte Seite. Dabei gelten folgende Abkürzungen:

⟨...⟩	kennzeichnen Regelbezeichner
|	Alternative
{ ... }	der Inhalt in den geschweiften Klammern darf beliebig oft wiederholt werden (auch null mal)
[...]	Der Inhalt in den Klammern darf null oder einmal vorkommen

Beispiele:

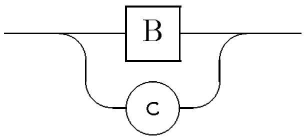
- ⟨*Ziffer*⟩ ::= 0|1|2|3|4|5|6|7|8|9
- ⟨*Buchstabe*⟩ ::= A|B|C|D|...|X|Y|Z
- ⟨*Bezeichner*⟩::= ⟨*Buchstabe*⟩ { ⟨*Buchstabe*⟩ | ⟨*Ziffer*⟩ }
- ⟨*REAL-Konstante*⟩ ::=
  [“+″|“−″]⟨*Ziffer*⟩{⟨*Ziffer*⟩}“.”{⟨*Ziffer*⟩}[E[“ +″ |“−″]⟨*Ziffer*⟩{⟨*Ziffer*⟩}]
- ⟨*REPEAT-Anweiung*⟩ ::= REPEAT ⟨*Anweisung*⟩ {“;” ⟨*Anweisung*⟩ } UNTIL ⟨*Bedingung*⟩

Satz:
Syntaxdiagramme und die EBNF haben die gleiche Ausdrucksstärke, d.h. ist ein Programm durch Syntaxdiagramme beschreibbar, dann auch durch EBNF und umgekehrt. (Mittels der EBNF kann man also genau die kontextfreien Sprachen beschreiben)

Beweis (Syntaxdiagramme $\Rightarrow$ EBNF):
Wir müssen jede EBNF–Regel durch geeignete Syntaxdiagramme simulieren, zum Beispiel:

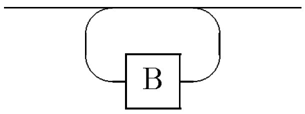
- $\langle A \rangle ::= \langle B \rangle \mid \mathrm{C}$ wird simuliert durch:

A



- $\langle A \rangle ::= \{\langle B \rangle\}$ wird simuliert durch:

A



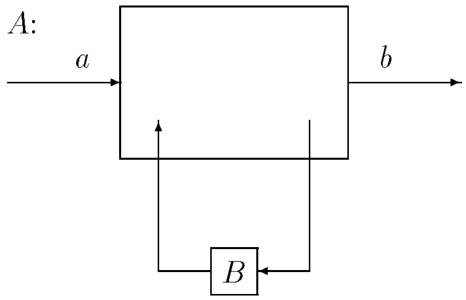
- etc.

Beweis (EBNF $\Rightarrow$ Syntaxdiagramme):
Es ist zu zeigen, daß jedes Syntaxdiagramm durch eine EBNF-Regel simuliert werden kann. Dies geschieht durch vollständige Induktion über die Anzahl der im Diagramm vorkommenden Kästen.

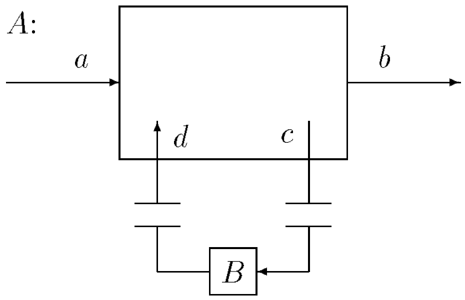
I.Anfang: Die Behauptung ist für 0 bzw. 1 Kasten trivial.

I.Annahme: Die Behauptung gilt für Diagramme mit n Kästen.

I.Schritt: Es sei A ein Syntaxdiagramm mit $(n+1)$ Kästen:



Sei $\boxed{B}$ ein beliebiger Kasten aus A. Nun wird $\boxed{B}$ aus dem Diagramm herausgelöst.



Wir erhalten nun vier Teildiagramme D_{ab}, D_{ac}, D_{db} und D_{dc} mit jeweils höchstens n Kästchen (D_{ij} bezeichnet das Diagramm mit dem Eingang $i \in \{a, d\}$ und dem Ausgang $j \in \{b, c\}$). Für alle vier Teildiagramme existiert nach Induktionsannahme eine EBNF-Regel:

$\langle D_{ab} \rangle$ ::= ...; $\langle D_{ac} \rangle$ ::= ...; $\langle D_{db} \rangle$ ::= ...; $\langle D_{dc} \rangle$ ::= ...;

Damit erhalten wir für A folgende EBNF-Regel:

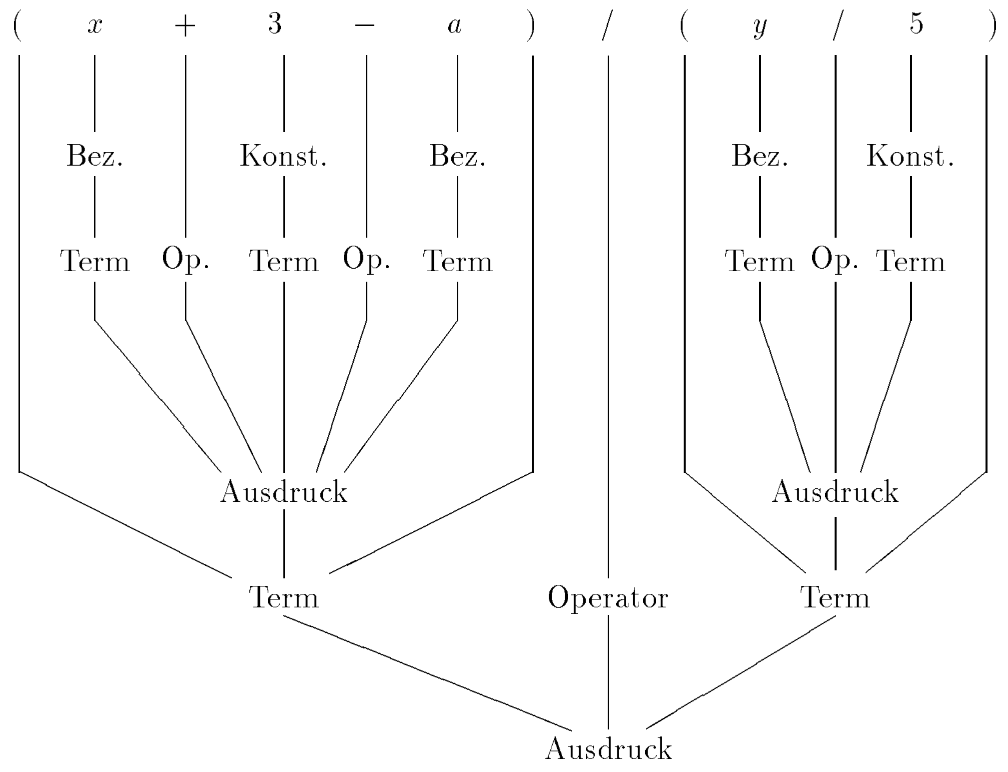
$\langle A \rangle$ ::= $\langle D_{ab} \rangle \mid \langle D_{ac} \rangle \langle B \rangle \; \{\langle D_{dc} \rangle \langle B \rangle\} \; \langle D_{db} \rangle$

(Um von a nach b zu gelangen benötigt man entweder B nicht, oder man geht von a nach c , durchläuft B, wiederholt evtl. die Strecke d–c und anschließend B beliebig oft, um dann von d zum Ausgang b zu gehen.)

1.6.4 Syntaktische Analyse

Die syntaktische Analyse eines (Quell-)Programms wird durch die Ableitung eines Ableitungsbaumes realisiert; ist das nicht möglich, so erfolgt eine Fehlermeldung. Die syntaktische Analyse auf dem Rechner erfolgt mit Hilfe eines *Parsers* (Zerteiler).

Beispiel:
Der Parser für MODULA–2 übersetzt $(x+3-a)/(y/5)$ als syntaktisch korrekten Ausdruck in:



Mit Hilfe des Ableitungsbaumes werden dann bei semantischer Analyse weitere Überprüfungen auf Korrektheit durchgeführt und die Erzeugung des Codes vorbereitet.

1.6.5 Semantik von Programmiersprachen

Semantik ist die Lehre von der inhaltlichen Bedeutung der syntaktisch korrekten aber zunächst sinnleeren Sätze (Programme) einer (Programmier)-Sprache. Die Semantik wurde mangels Beschreibungsmethoden früher fast ausschließlich *informal* beschrieben. Anknüpfend an die mathematische Notation war das Gefühl verbreitet, daß klar ist, was ein Programm bedeutet. Seit den 70er Jahren gibt es aber erhebliche Fortschritte bei den formalen Beschreibungsmethoden des Semantik. (↑Vorlesung "Semantik")

Ansätze zur Beschreibung der Semantik von Programmiersprachen sind:

- Übersetzersemantik
- operationale Semantik
- denotationelle Semantik
- axiomatische Semantik

1.6.6 Semantische Analyse

Die semantische Analyse ist eine Phase bei der Übersetzung eines Programms und erfolgt nach der syntaktischen Analyse anhand eines Ableitungsbaumes. Sie erfüllt im wesentlichen zwei Aufgaben:

1. Die Überprüfung des Ableitungsbaumes auf Einhaltung kontextsensitiver Zusatzbedingungen:
   Die syntaktische Analyse kann nur Fehler in Bezug auf den kontextfreien Kern der Sprache auffinden, nicht jedoch in Bezug auf die kontextsensitiven Zusatzbedingungen wie zum Beispiel:

   "Jede Variable muß ihrem Datentyp entsprechen." oder

   "Jeder verwendete Bezeichner muß deklariert sein."

2. Vorbereitung der Codeerzeugung:
   Der Ableitungsbaum wird durchlaufen und dabei werden die Adressen für Variablen, Ausdrücke und evtl. Anweisungen innerhalb des Zielprogramms zugeordnet. Dies geschieht mit Hilfe von *Symboltabellen*:

Bezeichner	Typ	Adresse
...	...	...
X	INTEGER	100
Y	REAL	104
Z	REAL	112
...	...	...

1.6.7 Überblick über Programmiersprachen

Es gibt schätzungsweise mehr als 1000 verschiedene Programmiersprachen, die meisten davon sind auf bestimmte Problemfelder zugeschnitten. Weite Verbreitung haben davon jedoch nur etwa 20 Progammiersprachen gefunden. Wir haben Programmiersprachen bereits folgendermaßen eingeteilt:

- Maschinensprachen (direkte Programmierung der Hardware)
- niedere maschinenorientierte Programmiersprachen oder Assembler-Sprachen (Programmierung der Hardware mit symbolischen Namen)
- höherere oder problemorientierte Programmiersprachen (von der Hardware unabhängig, an den zu bearbeitenden Problemen orientiert)

Jeder Programmiersprache liegt ein bestimmtes Konzept zu Grunde. Auf der Basis dieser Konzepte unterteilt man die höheren Programmiersprachen in folgende Kategorien:

Imperative Programmiersprachen:

Bei diesen Sprachen besteht ein Programm aus einer Folge von Befehlen an den Computer, wie z.B:

> "Schreibe in die Variable X den Wert 7."
> "Springe an die Stelle x im Programm."

Wesentlich an diesen Sprachen ist das *Variablenkonzept*: Eingabewerte werden in Variablen (Speicherzellen) gespeichert und weiterverarbeitet. In diesen Sprachen spiegelt sich deutlich die Architektur des ↑von–Neumann–Rechners wider. Beispiele für imperative Programmiersprachen sind: ADA, ALGOL, BASIC, C, FORTRAN, MODULA–2, PASCAL, ...

Funktionale Programmiersprachen:

Die Programme werden als Funktionen von Eingabedaten (den Parameter) in eine Menge von Ausgabedaten betrachtet. Hierbei werden komplexere Funktionen durch Zusammensetzung aus einfacheren Funktionen definiert. Ein Beispiel für eine funktionale Programmiersprache ist LISP.

Logik-basierte Programmiersprachen:

Bei diesen Sprachen werden die Programme als Beweise in einem System von Tatsachen und Schlußfolgerungen aufgefaßt: Der Anwender gibt eine Menge von Fakten (gültige Prädikate) und Regeln (Aussagen, wie man aus Fakten neue Fakten gewinnen kann) vor, und die Aufgabe des Rechners ist es, eine gestellte Frage als richtig oder falsch zu beantworten. Ein Beispiel für eine Logik-basierte Programmiersprache ist PROLOG.

Objektorientierte Programmiersprachen:

Bei diesen Sprachen werden alle zum Lösen eines Problems notwendigen Informationen (Daten, Anweisungen, Regeln) in sogenannten *Objekten* zusammengefaßt. Diese Objekte können durch Senden von Nachrichten miteinander kommunizieren und Informationen austauschen. Jedes Objekt gehört zu einer bestimmten *Klasse*, durch die festgelegt ist, welche Nachrichten es versteht, und wie es auf diese reagiert. Beispiele für objektorientierte Programmiersprachen sind C++ und SMALLTALK.

2 Die Grundbausteine der Programmierung in MODULA–2

Wir haben bereits die Syntax von einigen MODULA–2 Konstrukten kennengelernt. Eine vollständige Aufstellung der Syntax kann die Vorlesung jedoch nicht liefern. Dies ist MODULA–2–Büchern zu entnehmen. Es soll hier mehr Grundsätzliches zur Programmierung am Beispiel von MODULA–2 angesprochen werden.

Literatur:

- N. Wirth: "Programmieren in MODULA–2"(1976), "Algorithmen und Datenstrukturen in MODULA–2" (1986)
- G. Blaschek, G. Pomberger, F. Ritziger: "Einführung in die Programmierung mit MODULA–2"

2.1 Grundsätzlicher Aufbau eines MODULA–2 Programms

MODULA–2 ist eine imperative Programmiersprache, die auf PASCAL aufbaut und als wichtigste Erweiterung ein Modulkonzept besitzt. Ein MODULA–2–Programm ist ein "Programmodul", das wiederum implementiert werden kann unter Benutzung einer Reihe von anderen Modulen. Die Syntax für ein Programmodul lautet:

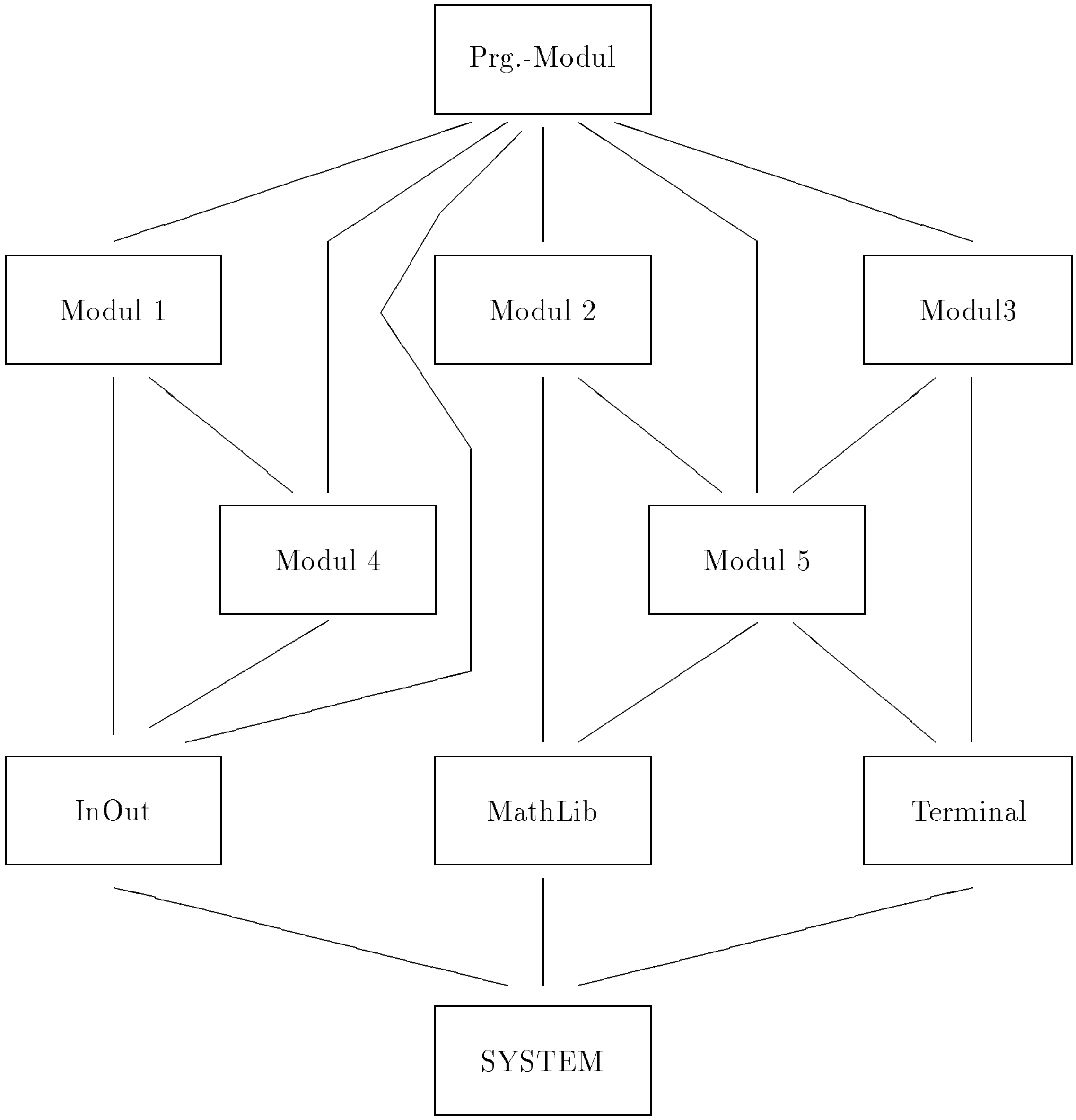
`MODULE` ⟨*Bezeichner*⟩";" [⟨*Importliste*⟩] [⟨*Deklarationen*⟩]
`BEGIN` [Anweisung] {";" ⟨*Anweisung*⟩} `END` ⟨*Bezeichner*⟩"."

Der *Modulbezeichner* am Anfang (nach `MODULE`) und am Ende (vor dem Punkt) müssen identisch sein. *Achtung:* Diese Bedingung kann nicht durch ein Syntaxdiagramm alleine ausgedrückt werden.

Die *Importliste* legt fest, welche Objekte (Konstanten, Variablen, Typen, Prozeduren) aus anderen Modulen innerhalb des vorliegenden Programmoduls bekannt gemacht werden sollen, um sie dann im Anweisungsteil verwenden zu können. Zu importierende Objekte sind u.a. Ein– und Ausgabeprozeduren, die durch Bibliotheksmodule bereitgestellt werden. Eine Importliste kann beispielsweise folgendermaßen aussehen:

```
FROM InOut IMPORT WriteCard, WriteLn, ReadCard, Done;
FROM RealInOut IMPORT ReadReal, WriteReal;
```

Durch das Importieren von Objekten entsteht eine Hierachie zwischen den Modulen, die am Programmodul direkt oder indirekt beteiligt sind. Diese kann z.B. folgendermaßen aussehen:



Letztendlich importieren alle Module aus dem Pseudo–Modul `SYSTEM`, das im allgemeinen Teil des Compilers und nicht explizit als MODULA–2 Programm implementiert ist.

Die in einem Modul verwendeten Objekte werden durch *Deklarationen* eingeführt. Dies kann z.B. folgendermaßen aussehen:

```
CONST max   = 100;
      pi    = 3.14;

TYPE  farbe = (schwarz, rot, gelb);
      card  = CARDINAL;
      tag   = (Mo, Di, Mi, Do, Fr, Sa, So);
```

```
VAR   x,y : REAL;
      i   : card;
      f   : farbe;
      j   : INTEGER;
      c   : tag;

PROCEDURE ggt1( ... ) ...;
```

MODULA–2 schreibt die Deklaration jedes verwendeten Objekts vor. Mit dieser Vorgehensweise werden verschiedene Ziele verfolgt:

- Die Deklaration verbindet einen Namen mit einem Datentyp, d.h. der Compiler kann bereits vor Ausführung des Programms die mißbräuchliche Verwendung des zugehörigen Objekts entdecken und als Fehler melden.
- Häufig verwendete Konstanten können mit Namen versehen werden, d.h. das Programm wird wartungsfreundlicher und übersichtlicher.
- MODULA–2 hat wie schon PASCAL ein Typkonzept. Darüberhinaus ist noch die Deklaration eigener Datentypen erlaubt. Die Typdeklaration vergibt einen Namen als “Synonym” für diesen Datentyp.
- Der Compiler kann aufgrund der Deklaration von Variablen für diese Speicherplatz passender Länge reservieren. Außerdem kann zur Laufzeit überprüft werden, ob Variablen gültige Werte annehmen.

2.2 Anweisungen in MODULA–2

2.2.1 Wertzuweisung

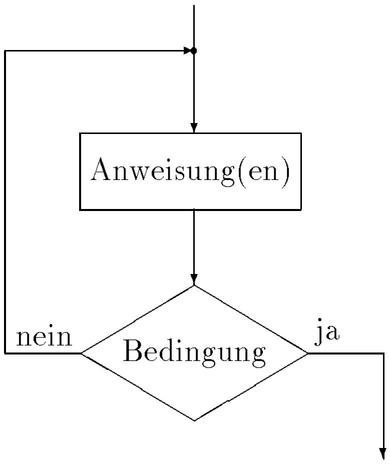
Syntax: ⟨*Bezeichner*⟩ “`:=`” ⟨*Ausdruck*⟩

Semantik: Der ⟨*Ausdruck*⟩ wird ausgewertet und der dabei berechnete Wert wird der durch ⟨*Bezeichner*⟩ bezeichneten Variable zugewiesen, ihr alter Wert geht dabei verloren. Zusätzlich gilt, daß der Typ des Bezeichners *zuweisungskompatibel* mit dem Typ des Bezeichners sein muß (d.h. im wesentlichen übereinstimmen muß).

2.2.2 `REPEAT`–Schleife

Syntax: `REPEAT` ⟨*Anweisungsfolge*⟩ `UNTIL` ⟨*Bedingung*⟩

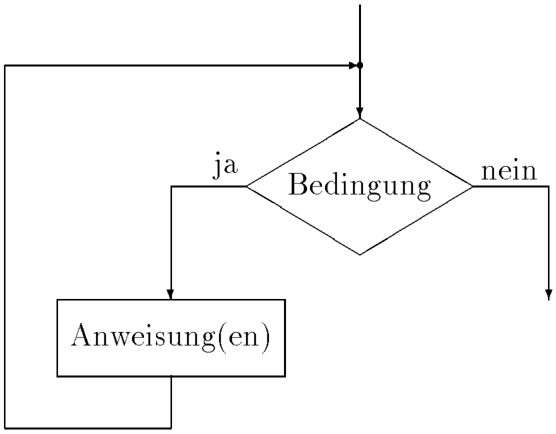
Semantik: Die ⟨*Anweisungsfolge*⟩ wird so lange wiederholt, bis die ⟨*Bedingung*⟩ (= Ausdruck vom Typ `BOOLEAN`) erfüllt ist. Das entsprechendes Flußdiagramm sieht folgendermaßen aus:



2.2.3 `WHILE`–Schleife

Syntax: `WHILE` ⟨*Bedingung*⟩ `DO` ⟨*Anweisungsfolge*⟩ `END`

Semantik: Solange die ⟨*Bedingung*⟩ (die zuerst geprüft wird) erfüllt ist, wird die ⟨*Anweisungsfolge*⟩ wiederholt. Das entsprechendes Flußdiagramm sieht folgendermaßen aus:



Bemerkung: `REPEAT` und `WHILE`–Schleifen dienen zur Bildung "iterativer" Schleifen, die solange ausgeführt werden, bis ein bestimmtes Abbruchkriterium erfüllt ist. Die `WHILE`–Schleife bricht ab, wenn die Schleifenbedingung `FALSE` ergibt, die `REPEAT`–Schleife endet, wenn die Schleifenbedingung `TRUE` ergibt. Die `REPEAT` und die `WHILE`–Schleife lassen sich ineinander überführen, die Transformation sieht folgendermaßen aus:

(a) `WHILE` *Bedingung* `DO` *Anweisung* `END`

kann umgewandelt werden in:

```
IF Bedingung THEN
  REPEAT Anweisung UNTIL NOT Bedingung
END
```

(b) `REPEAT` *Anweisung* `UNTIL` *Bedingung*

kann simuliert werden durch:

```
Anweisung;
WHILE NOT Bedingung DO Anweisung END
```

oder durch die Einführung einer Booleschen Hilfsvariablen `Looping`:

```
Looping := TRUE;
WHILE Looping DO
  Anweisung;
  Looping := NOT Bedingung;
END
```

2.2.4 `LOOP`–Schleife und `EXIT`–Anweisung

Syntax: `LOOP` ⟨*Anweisungsfolge*⟩ `END`

Semantik: Die ⟨*Anweisungsfolge*⟩ wird wiederholt. Erst wenn die `EXIT`–Anweisung ausgeführt wird, wird die Schleife verlassen.

Beispiel:

```
LOOP
  ⋮
  IF Abbruchbedingung THEN EXIT END;
  ⋮
END
```

Bemerkung: Die `EXIT`–Anweisung besteht nur aus dem Schlüsselwort `EXIT` und darf nur in `LOOP`–Schleifen auftreten. `LOOP`–Anweisungen bilden "Endlosschleifen", die durch eine `EXIT`–Anweisung verlassen werden können. Das Programm wird dann hinter `END` fortgesetzt. `LOOP`–Anweisungen können auch geschachtelt werden, d.h. der Rumpf kann seinerseits weitere `LOOP`–Anweisungen enthalten. Eine `EXIT`–Anweisung bewirkt jeweils nur das Verlassen der aktuellen Schleife. Die `LOOP`–Anweisung ist die allgemeinste Form der Schleifenbildung:

```
WHILE Bedingung DO                 LOOP
  Anweisung         entspricht       IF NOT Bedingung THEN
                                       EXIT END
END                                  Anweisung
                                   END

REPEAT                             LOOP
  Anweisung         entspricht       Anweisung
UNTIL Bedingung                      IF Bedingung THEN EXIT END;
                                   END
```

Ein Vorteil der `LOOP`–Anweisung ist, daß sie flexibler als die `WHILE` und `REPEAT`–Anweisung ist, denn eine `LOOP`–Anweisung kann überall innerhalb der Schleife verlassen werden. Eine `LOOP`–Anweisung kann auch mehrere `EXIT`–Anweisungen enthalten, so daß mehrere Abbruchbedingungen an verschiedenen Stellen vorgesehen werden können. Nachteilig bei der `LOOP`–Anweisung ist die schlechtere Lesbarkeit des Quelltextes gegenüber der `REPEAT` bzw. der `WHILE`–Anweisung.

2.2.5 `FOR`–Anweisung

Syntax: `FOR` ⟨*Bezeichner*⟩ "`:=`" ⟨*Ausdruck*⟩$_1$ `TO` ⟨*Ausdruck*⟩$_2$ [`BY` ⟨*Konstante*⟩] `DO` ⟨*Anweisungsfolge*⟩ `END`

Semantik: ⟨*Bezeichner*⟩ bezeichnet eine Variable ("Laufvariable") vom Datentyp `INTEGER`, `CARDINAL`, `BOOLEAN`, `CHAR`, Aufzählungstyp oder Teilbereichstyp ("Ordinaltypen"), die nicht aus einem anderen Modul importiert oder formaler Parameter einer Prozedur ist. ⟨*Ausdruck*⟩$_1$ und ⟨*Ausdruck*⟩$_2$ müssen *ausdruckskompatibel* zur Laufvariablen sein. Die Laufvariable nimmt der Reihe nach die Werte von ⟨*Ausdruck*⟩$_1$ bis ⟨*Ausdruck*⟩$_2$ an mit einer Schrittweite von 1 oder, falls angegeben, mit der Schrittweite der ⟨*Konstante*⟩($\neq$0). Mit jedem Wert der Laufvariablen wird die ⟨*Anweisungsfolge*⟩ einmal ausgeführt, die Laufvariable darf innerhalb der Anweisungsfolge nicht verändert werden. Nach Beendigung der Schleife ist ihr Wert undefiniert.

Bemerkung: `FOR`–Schleifen enden grundsätzlich immer nach endlich vielen Schritten (im Unterschied zu `LOOP`, `WHILE` und `REPEAT`–Schleifen), vorausgesetzt, die Anweisungen enden im Inneren. Wir können also `FOR`–Schleifen mit `LOOP`, `WHILE` oder `REPEAT`–Schleifen nachbilden, aber nicht umgekehrt. (↑ Berechnungstheorie)
Die Transformation sieht folgendermaßen aus:

```
FOR i:=Ausdruck1 TO Ausdruck2 BY Ausdruck3 DO Anweisung END
```

kann überführt werden in:

```
i:=Ausdruck1
t:=Ausdruck2 (* Eine evtl. Änderung von Ausdruck2 im Laufe
                der FOR-Schleife hat keine Auswirkung auf die
                Anzahl der Schleifendurchläufe *)
WHILE ((i<=t) AND (Ausdruck3>0)) OR
      ((i>=t) AND (Ausdruck3<0)) DO
  Anweisung
  i:=i+Ausdruck3
END
```

2.2.6 `IF`–Anweisung

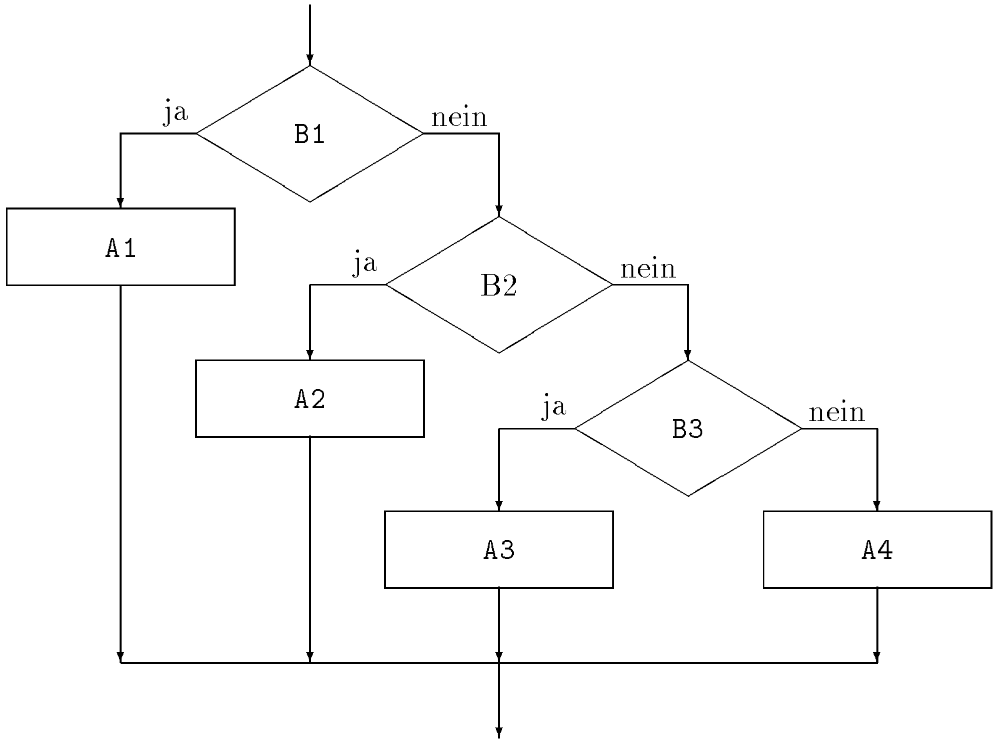
Syntax: `IF` ⟨*Bedingung*⟩$_1$ `THEN` ⟨*Anweisungsfolge*⟩$_1$
{`ELSIF` ⟨*Bedingung*⟩$_2$ `THEN` ⟨*Anweisungsfolge*⟩$_2$}
[`ELSE` ⟨*Anweisungsfolge*⟩$_3$ `END`]

Semantik: ⟨*Bedingung*⟩$_1$ und ⟨*Bedingung*⟩$_2$ müssen vom Typ `BOOLEAN` sein. Falls ⟨*Bedingung*⟩$_1$ erfüllt ist, wird ⟨*Anweisungsfolge*⟩$_1$ ausgeführt und die Ausführung der `IF`–Anweisung beendet. Andernfalls werden, wenn vorhanden, die ⟨*Bedingung(en)*⟩$_2$ geprüft und im positiven Fall die entsprechende ⟨*Anweisungsfolge*⟩$_2$ ausgeführt. Falls keine `IF` oder `ELSIF` Bedingung zutrifft und falls ein `ELSE`–Teil vorhanden ist, so wird ⟨*Anweisungsfolge*⟩$_3$ ausgeführt.

Beispiel:

```
IF B1 THEN A1
  ELSIF B2 THEN A2
  ELSIF B3 THEN A3
  ELSE A4
END
```

Ein Flußdiagramm für dieses Beispiel sieht folgendermaßen aus:



2.2.7 `CASE`–Anweisung

Syntax: `CASE` ⟨*Ausdruck*⟩ `OF`
⟨*Konstante*⟩ {“,” ⟨*Konstante*⟩}“:” ⟨*Anweisungsfolge*⟩
{“|” ⟨*Konstante*⟩ {“,” ⟨*Konstante*⟩}“:” ⟨*Anweisungsfolge*⟩ }
[`ELSE` ⟨*Anweisungsfolge*⟩]
`END`

Semantik: Zuerst wird ⟨*Ausdruck*⟩ ausgewertet. Falls der Wert des Ausdrucks in einer der Konstantenlisten vorkommt, so wird die entsprechende Anweisungsfolge ausgeführt und die `CASE`–Anweisung danach beendet. Wenn keiner der Konstanten mit dem Wert des Ausdrucks übereinstimmt, und falls ein ELSE–Teil vorhanden ist, so wird die entsprechende Anweisungsfolge ausgeführt. Es wird eine Fehlermeldung ausgegeben, falls keine der Konstantenlisten zutrifft und kein ELSE–Teil vorhanden ist.

Bemerkung: Anstelle von Konstanten sind in MODULA–2 auch Bereichsangaben der Form ⟨*Konstante*⟩“..”⟨*Konstante*⟩ zul”assig.

Beispiel:
```
CASE n OF
  1, 8, 9      : A1;
  12           : A2;
  3..7, 20, 29 : A3;
  ELSE A4;
END
```

2.3 Beispielprogramm: Arithmetisches Mittel

Gegeben: Eine Zahlenfolge $(x_1, \ldots, x_n)$ von reellen Zahlen.

Gesucht: Das arithmetische Mittel M und die Varianz D.

Es gilt:

$$
\begin{aligned}
M &= \frac{1}{n}\sum_{i=1}^{n} x_i \\
D &= \frac{1}{n}\sum_{i=1}^{n}(x_i - M)^2 \\
&= \ldots \\
&= (\frac{1}{n}\sum_{i=1}^{n} x_i^2) - M^2
\end{aligned}
$$

Bei Benutzung der zweiten Form müssen die x_i zur Berechnung von D nicht zwischengespeichert werden.

Programm:
```
MODULE Statistik;

FROM RealInOut IMPORT Done, ReadReal, WriteReal;
FROM InOut     IMPORT WriteString, WriteLine;

VAR s, s2, x: REAL (* s = Summe der Eingabewerte
         s2= Summe der Quadrate der Eingabewerte *)
    n       : CARDINAL;
```

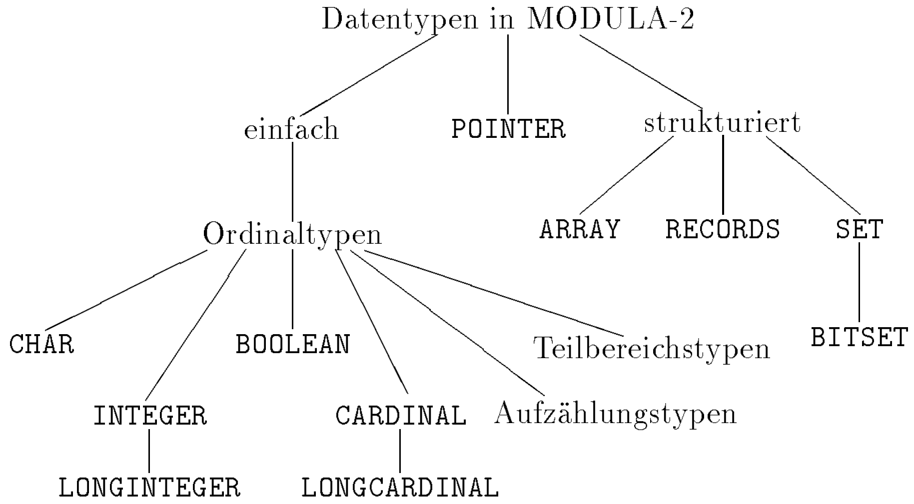
```
BEGIN
  s := 0.0; s2 := 0.0; n := 0;
  LOOP
    WriteLn; WriteString('Gib eine Zahl ein: ');
    ReadReal(x);
    IF NOT Done THEN EXIT END;
    (* Done=FALSE wenn unzulaessige Eingabe erfolgte *)
    s := s+x; s2 := s2+x*x; INC(n);
    (* INC erhoeht den Wert um 1 *)
  END;

  IF n>0 THEN
    WriteLn; WriteString('Mittelwert = ');
    WriteReal(s/FLOAT(n), 10);
    WriteLn; WriteString('Varianz    = ');
    WriteReal(s2/FLOAT(n) - s*s/FLOAT(n*n), 10);
    WriteLn;
  END;
END Statistik.
```

2.4 Datentypen in MODULA–2

MODULA–2 stellt dem Programmierer verschiedene *Datentypen* standardmäßig zur Verfügung: `INTEGER`, `CARDINAL`, `REAL`, `BOOLEAN`, `CHAR`, `BITSET`. Darüberhinaus können komplexe Datentypen aus einfachen Grundtypen aufgebaut werden. Dazu dienen die Schlüsselworte `SET, ARRAY,RECORD`. Jeder Datentyp definiert eine Wertemenge und eine Reihe von Operationen, die auf den Elementen dieser Menge angewandt werden dürfen.

Neben den vordefinierten *einfachen* Datentypen gibt es *strukturierte* Datentypen, die aus anderen Typen aufgebaut sind und somit von diesen abhängen.



2.4.1 CARDINAL

Der Typ CARDINAL repräsentiert die Menge der nichtnegativen ganzen Zahlen im Bereich von 0 bis MAX(CARDINAL) . Werden beispielsweise 16 Bit für die Darstellung von INTEGER–Zahlen verwendet, so gilt: MAX(CARDINAL)=65 535. Bei 32–Bit Rechnern gilt: MAX(CARDINAL) = 4 294 967 295.

Mit CARDINAL–Objekten sind folgende Operationen möglich:

+	Addition
–	Subtraktion
*	Multiplikation
DIV	ganzzahlige Division
MOD	Divisionsrestbildung

Das Ergebnis ist wieder vom Typ CARDINAL. Sollte es nicht im zulässigen Bereich liegen, so erfolgt ein Abbruch wegen eines “overflow error”. Des weiteren sind folgende Vergleichsoperationen anwendbar: =, <, >, <=, >=, <>. Das Ergebnis der Vergleichsoperatoren ist ein Boolescher Wert.

2.4.2 INTEGER

Der Typ INTEGER repräsentiert ganze Zahlen im Bereich MIN(INTEGER) (negative Zahl) bis MAX(INTEGER) (positive Zahl). Für einen 16-Bit-Rechner gilt MIN(INTEGER) = –32768 und MAX(INTEGER) = 32767. Die möglichen Operatoren sind die gleichen wie beim Typ CARDINAL.

Negative Zahlen werden beim Typ INTEGER intern in *Zweierkomplementdarstellung* gespeichert, die beispielsweise für 3 Bit folgendermaßen aussieht:

Interne Darstellung	INTEGER	CARDINAL
000	0	0
001	1	1
010	2	2
011	3	3
100	-4	4
101	-3	5
110	-2	6
111	-1	7

Im Bereich von 0 bis MAX(INTEGER) sind INTEGER und CARDINAL kompatibel. Der Vorteil der Zweierkomplementdarstellung liegt darin, daß rechnerinterne arithmetische Operationen, wie die Feststellung des Vorzeichens, Addition und Subtraktion einfach realisierbar sind.

Beispiel:

$$\begin{array}{rcr} 0110 & \leftarrow & +\,6 \\ +\;1101 & \leftarrow & -\,3 \\ \hline 10011 & \rightarrow & 3 \end{array}$$

Ein eventueller Übertrag wird einfach abgeschnitten.

2.4.3 BOOLEAN

Der Datentyp `BOOLEAN` repräsentiert die Wahrheitswerte `TRUE` und `FALSE`, somit gilt: `BOOLEAN = {FALSE, TRUE}`. Mögliche *Operationen* sind `AND`, `OR` und `NOT`. Auf `BOOLEAN` anwendbare *Vergleichsoperatoren* sind: `=, <, >, <=, >=, <>`, wobei gilt: `FALSE(0) < TRUE(1)`

Beispiel:

AND	F	T
F	F	F
T	F	T

OR	F	T
F	F	T
T	T	T

Die syntaktische Einheit “Bedingung” steht für einen Ausdruck vom Typ `BOOLEAN`.

Beispiel:

```
b := TRUE
c := NOT( (b OR TRUE) AND (x > 0) )
```

2.4.4 CHAR

Der Datentyp `CHAR` repräsentiert die Menge der auf dem Rechner darstellbaren Zeichen. Jedem Zeichen ist eine Nummer zugeordnet, die anzeigt, an welcher Stelle dieses Zeichen innerhalb des Zeichensatzes steht. Nach der *ASCII–Norm* (“American Standard Code for Information Interchange”) hat z.B. “A” die Nummer 65.

Beispiel:

```
CONST  c  = 'A';
       ch = 'B';
VAR    d  : CHAR;
       ⋮
       d  := '+';
```

Nicht druckbare Zeichen werden durch Positionsnummer (in ASCII–Tabelle) und Anfügung des Buchstaben `C` repräsentiert.

Beispiel: `c := 12C;`
oder: `c := CHR(12);`

Die Standardfunktion `ORD` liefert bei Eingabe eines Zeichens dessen Ordnungs– oder *Ordinalzahl* (=`CARDINAL`). Die Standardfunktion `CHR` liefert bei Eingabe einer `CARDINAL`–Zahl das entsprechende Zeichen.

Beispiel: Zeichenweises Einlesen einer `CARDINAL`–Zahl:

```
x := 0;
Read(ch) (* Read wird von InOut importiert *)
WHILE ('0' <= ch) AND (ch <= 9) DO
  x := 10 * x +(ORD(ch) - ORD('0'));
  Read(ch);
END;
```

2.4.5 `REAL`

Der Datentyp `REAL` repräsentiert reelle Zahlen.

Bemerkung: Wegen der Endlichkeit der internen Darstellung können nur endlich viele rationale Zahlen dargestellt werden.

Übiche Darstellung:

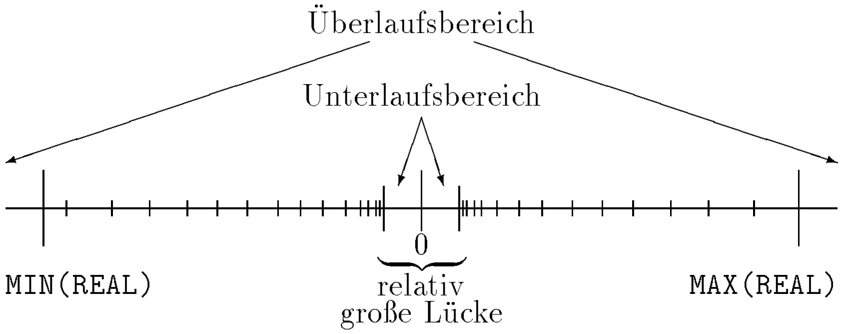
±	Mantisse 4 byte = 32 bits	±	Exponent 1 byte = 8 bits

"normierte" Darstellung : Mantisse: $0.a_1a_2 \ldots a_n$ mit $a_1 \neq 0$

Beispiel: 3.14 wird dargestellt als: $0.314 * 10^1$ (Dezimalsystem)
$0.11001001 * 2^2$ (Dualsystem)

- 0 erfordert eine Sonderdarstellung (wegen $a_1 \neq 0$).
- Im Dualsystem kann bei normierter Darstellung auf die Speicherung von a_1 verzichtet werden, da stets $a_1 = 1$.

Verteilung der darstellbaren reellen Zahlen auf einer Zahlengerade:



Beispiel zur Problematik der `REAL`–Arithmetik Die folgenden Algorithmen brechen nach endlich vielen Schritten ab, obwohl sie mathematisch beide nie enden dürften:

```
x := 1.0;
n := 0;
REPEAT
  x := x / 2.0;
  INC(n);
UNTIL x = 0.0;
WriteCard(n,10);
```

Abbruch(PC) : bei 127

```
x := 1.0;
n := 0;
REPEAT
  x := x / 2.0;
  INC(n);
UNTIL x + 1.0 = 1.0;
WriteCard(n,10);
```

Abbruch(PC) : bei 24

Mit `REAL`–Objekten sind folgende Operationen möglich:

+	Addition
–	Subtraktion
*	Multiplikation
/	Division

Sowie Vergleichsoperationen

Zur Typumwandlung zwischen `REAL` und `INTEGER` (bzw. `CARDINAL`) dienen die Standardfunktionen `FLOAT` und `TRUNC`.

Beispiel:
```
FLOAT (3)     = 3.0   (Umwandlung von INTEGER in REAL)
TRUNC (3.14)  = 3     (Umwandlung von REAL in INTEGER)
TRUNC (-4.9)  = -4
```
(`TRUNC` – abschneiden der letzten Stelle)

2.4.6 Aufzählungstypen

Die zulässigen Werte für diesen Datentyp werden explizit aufgezählt. Einzelne Aufzählungskonstanten werden mit Ordinalzahl in der Reihenfolge versehen, in der sie aufgeschrieben sind; die erste Konstante trägt die Nummer 0. Ein Vergleich zwischen den Werten ist also möglich.

Beispiel:
```
TYPE farbe = (rot, blau, gelb);
VAR f,g : farbe;
    i : CARDINAL;
    ⋮
  f := rot;
  INC(f);            (* f = blau *)
  i := ORD(f);       (* i = 1 *)
  g := VAL(farbe,2); (* g = gelb *)
```

Zulässige Operationen sind:

Wertzuweisung
Vergleich
`INC`, `DEC`
`ORD`, `VAL`

2.4.7 Teilbereichstypen

Teilbereichstypen werden zur Einschränkung des Wertebereichs von Typen verwendet, deren Elemente mit Ordinalzahlen versehen sind (`INTEGER, CARDINAL, BOOLEAN, CHAR,` Aufzählungstypen). Die Deklaration zeigt an, daß sich die Elemente dieses Typs nur zwischen fest vorgegebenen Grenzen bewegen dürfen.

Vorteile:

- gibt dem Programmierer erhöhte Sicherheit gegen Programmierfehler
- ermöglicht dem Compiler einen Code zu erzeugen, der zur Laufzeit die Einhaltung der vom Programmierer gemachten Einschränkungen überprüft

Beispiel:
```
TYPE  monat         = [1..12];
      buchstabe     = ['a'..'z'];
      kleininteger  = [-10..10];
```

2.4.8 Operationen und Standardprozeduren

Die wichtigsten Operationen im Überblick:

Operator	Argumenttyp	Ergebnistyp

`+,-,*`	`REAL`
	`INTEGER`
	`CARDINAL`
		`REAL`
		`INTEGER`
		`CARDINAL`

/	`REAL`	`REAL`
`DIV,MOD`	`INTEGER` `CARDINAL`	`INTEGER` `CARDINAL`
`AND, OR, NOT`	`BOOLEAN`	`BOOLEAN`
`=, <, >, <=, >=, <>`	einfache Typen	`BOOLEAN`

Die wichtigsten Standardprozeduren im Überblick:

Prozedurname	Argumenttyp	Ergebnistyp
`ABS`	`REAL` `INTEGER`	`REAL` `INTEGER`
`CHR`	`CARDINAL`	`CHAR`
`INC, DEC`	jeder Ordinaltyp	derselbe Typ
`FLOAT`	`INTEGER`	`REAL`
`TRUNC`	`REAL`	`INTEGER`
`ORD`	jeder Ordinaltyp	`CARDINAL`
`ODD`	`CARDINAL` `INTEGER`	`BOOLEAN` `BOOLEAN`

Es gibt verschiedene *Standardprozeduren*, die auf verschiedenen Datentypen unterschiedlich operieren. Sie müssen *nicht* explizit importiert werden.

2.4.9 `ARRAY`

`ARRAY`s dienen der Darstellung von Folgen von Daten des selben Typs. Der Zugriff auf die einzelnen Daten erfolgt über den *Index*. Obwohl theoretisch jeder Ordinaltyp für die Indexmenge verwendet werden kann, kommen aus Speichergründen lediglich Teilbereichstypen in Frage. Ihr Syntax lautet:

⟨*ArrayType*⟩ ::= `ARRAY` ⟨*SimpleType*⟩ {“,” ⟨*SimpleType*⟩} `OF` ⟨*type*⟩

Beispiel:

```
VAR x : ARRAY [1..100],[rot,gelb,blau] OF REAL;
  ⋮
 x[45,gelb] := 3.14;
```

“`ARRAY T1, T2 OF T3`” ist darstellbar als “`ARRAY T1 OF ARRAY T2 OF T3`”.

Typische Beispiele für `ARRAY`s

Einfaches Beispiel:

geg.: Feld a[1]..a[n] (`OF CARDINAL`) und x
ges.: kleinster Index i mit a[i] = x;
falls x nicht vorhanden, setze i=0.

Programmstück:

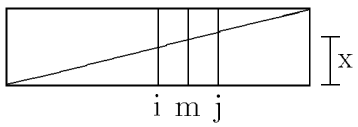
```
CONST N= ...
VAR a: ARRAY[1..N] OF CARDINAL;
      ⋮
  i := 1;
  WHILE (i <= N) AND (a[i] <> x) DO;
    INC(i);
  END;
  IF i = N+1 THEN i := 0 END;
```

Achtung: Das Programmstück ist nur korrekt, wenn ein Boolescher Ausdruck der Form "`B1 AND B2`" ausgewertet wird in der Form: "`IF NOT B1 THEN RETURN FALSE ELSE RETURN B2`"; der vordere wird also vor dem hinteren Ausdruck ausgewertet. Ansonsten führt der Test im Falle i = N+1 zu einer Fehlermeldung (es soll auf ein nichtdefiniertes `ARRAY`–Element a[N+1] zugegriffen werden). Immer korrekt ist hingegen folgendes Programmstück:

```
i := 1;
LOOP
  IF i > N THEN i := 0; EXIT END;
  IF a[i] = x THEN EXIT END;
  INC(i);
END;
```

Binäre Suche

Wäre das Feld beim "einfachen Beispiel" (oben) vorsortiert, dann könnte das Problem viel effizienter gelöst werden. Sind die Elemente in aufsteigender (oder absteigender) Form geordnet, so kann man immer das Feld halbieren. (Annahme: a[i] sind paarweise verschieden)




```
i := 1; j := N;
LOOP
  IF i > j THEN i := 0; EXIT END;
  m := (i + j) DIV 2;
  IF x = a[m] THEN i := m; EXIT END;
  ELSIF x < a[m] THEN j := m-1;
  ELSE i := m+1 END;
END;
```

Prinzip der "Binären Suche"

Es sind $\approx \log_2 N$ viele Suchschritte, d.h. Schleifendurchläufe erforderlich. Bei Realisierung des "einfachen Beispiels" wären im schlechtesten Fall N Schleifendurchläufe erforderlich.

Naive Sortierverfahren

1. **Sortieren durch direkte Auswahl**

```
CONST N=...;
VAR a : ARRAY [1..N] OF REAL;
  i,j : CARDINAL;
  x : REAL;
    ⋮
FOR i := 1 TO N-1 DO
  FOR j := i+1 TO N DO
    IF a[i] > a[j] THEN x := a[i];
      a[i] := a[j];
      a[j] := x;
    END;
  END;
END;
```

   Idee: Der Reihe nach wird das Minimum der Teilfolge a[i]..a[N] durch Vertauschen auf Position i gebracht; dann wird Verfahren mit erhöhtem i–Wert wiederholt.

2. **Bubble–Sort**

```
VAR b : BOOLEAN;
    ⋮
REPEAT
  b := TRUE;
  FOR i := 1 TO N-1 DO
    IF a[i] > a[i+1] THEN x := a[i];
      a[i] := a[i+1];
      a[i+1] := x;
      b := FALSE;
    END;
  END;
UNTIL b;
```

   Idee: Das `ARRAY` wird linear durchsucht. Falls benachbarte Elemente nicht in richtiger Reihenfolge liegen, werden sie vertauscht. Dieser Prozeß wiederholt sich, bis keine Vertauschung mehr stattfindet (wird mit booleschen Ausdruck b getestet, sogenanntes "flag").

Aufwandsanalyse:

Beim Sortieren durch direkte Auswahl wird das Innere der FOR–Schleife insgesamt

$$\sum_{i=1}^{N-1} \quad \sum_{j=i+1}^{N} 1 = \sum_{i=1}^{N-1}(N-i) = \sum_{i=1}^{N-1} i = \frac{N*(N-1)}{2} = \frac{N^2}{2} - \frac{N}{2}$$

oft durchlaufen.

Im günstigsten Fall ist Bubble–Sort linear.
Im schlechtesten Fall (kleinstes Element an Position N) ist Bubble–Sort quadratisch).

Auch im Mittel hat Bubble–Sort quadratische Laufzeit.

Bemerkung: Es gibt andere Sortierverfahren, die im *ungünstigsten* Fall mit Aufwand $N * \log N$ auskommen. (↑ Info–2; z.B. Heap–Sort)

Matrizenmultiplikation

`ARRAY`s entsprechen auf Datenebene der `FOR`–Schleife auf Anweisungsebene.

geg.:

$$m \times n \quad \text{Matrix} \quad A = \begin{pmatrix} a_{11} & \dots & a_{1n} \\ \vdots & & \vdots \\ a_{m1} & \dots & a_{mn} \end{pmatrix}$$

$$n \times l \quad \text{Matrix} \quad B = \begin{pmatrix} b_{11} & \dots & b_{1l} \\ \vdots & & \vdots \\ b_{n1} & \dots & b_{nl} \end{pmatrix}$$

Das Produkt $A * B = C$ ist definiert als:

$$C = \begin{pmatrix} c_{11} & \dots & c_{1l} \\ \vdots & & \vdots \\ c_{m1} & \dots & c_{ml} \end{pmatrix} \text{ mit } c_{ij} = \sum_{k=1}^{n} a_{ik} * b_{kj}$$

Als Programm geschrieben:

```
VAR a : ARRAY[1..m],[1..n] OF REAL;
    b : ARRAY[1..n],[1..l] OF REAL;
    c : ARRAY[1..m],[1..l] OF REAL;
    i,j,k : CARDINAL;
    s : REAL;
    ⋮
```

```
FOR i := 1 TO m DO
  FOR j := 1 TO l DO
    s := 0.0;
    FOR k := 1 TO n DO
      s := s + a[i,k] * b[k,j];
    END;
    c[i,j] := s;
  END;
END;
```

Dieser Matrizenmultiplikationsalgorithmus benötigt $n * l * m$ Schritte (z.B. $N = n = m = l$ also $\Rightarrow O(N^3)$ Schritte). Es gibt effizientere Verfahren ("*Strassensche Matrizenmultiplikation*"), die nur $O(N^{\log_2 7})$ ($\log_2 7 \approx 2,81$) Schritte benötigen.

Der momentane Weltrekord liegt bei $O(N^{2,376})$ Schritten.

Strings `ARRAY`s mit Elementtyp `CHAR` heißen Strings.

```
VAR s : ARRAY T OF CHAR
```

(Wirth: Indextyp `T` sollte Teilbereichstyp `T=[0..N-1]` für hinreichend große `N` sein.)

Bei Strings sind vereinfachte Zuweisungen möglich: `s := 'abc'`. Diese Anweisung hat den gleichen Effekt wie:

```
s[MIN(T)]   := 'a';
s[MIN(T)+1] := 'b';
s[MIN(T)+2] := 'c';
s[MIN(T)+3] := 0C;
   ⋮
s[MAX(T)] := 0C;
```

(`0C` = nicht druckbares Zeichen NULL; Position 0 der ASCII–Tabelle)

Die Prozedur WriteString im Modul InOut erwartet einen String als erstes Argument. Die Prozedur ReadString benutzt als Argument eine Variable vom Typ String.

2.4.10 SET

`SET` ist ein Datentyp zur Repräsentation von Mengen, also von Elementen eines gemeinsamen Grundtyps.

Syntax: ⟨*SetType*⟩ ::= `SET OF` ⟨*SimpleType.*⟩

Der Typ darf höchstens W Elemente haben (W ist systemspezifisch, z.B. W = 16 oder W = 32) und muß Aufzählungs– oder Teilbereichstyp sein.

Beispiel:

```
SET OF [3..5]
SET OF [eisen..kupfer]
SET OF BOOLEAN
```

Die *Mengenkonstanten* werden durch geschweifte Klammern mit vorangestelltem Grundtyp gekennzeichnet.

Beispiel:
```
TYPE farbe = (rot,gruen,gelb);
CONST rg = farbe{rot,gruen};
    alle = farbe{rot,gruen,gelb};
VAR x : SET OF farbe;
   ⋮
x := farbe{};
x := rg + (farbe{rot,gelb}/alle);
IF rot IN x THEN ...
```

Operationen:

+	: Mengenvereinigung	: `e IN s+t` $\Longleftrightarrow$ `(e IN s) OR (e IN t)`
$*$	: Mengenschnitt	: `e IN s*t` $\Longleftrightarrow$ `(e IN s) AND (e IN t)`
$-$	: Differenz	: `e IN s-t` $\Longleftrightarrow$ `(e IN s) AND NOT (e IN t)`
/	: symmetrische Differenz	: `e IN s/t` $\Longleftrightarrow$ `(e IN s) <> (e IN t)`
IN	bedeutet "$\in$"	
=	: =	
<>	: $\neq$	
<=	: $\subseteq$	
>=	: $\supseteq$	

`INCL`: Standardprozedur zum Hinzufügen eines Elementes zu einer Menge. Die Anweisung `INCL (s,x)` hat die Wirkung `s := s + {(Wert von)x}`.

`EXCL`: Standardprozedur zum Herausnehmen eines Elementes aus einer Menge. Die Anweisung `EXCL (s,x)` hat die Wirkung `s := s - {(Wert von)x}`

Der Standardtyp `BITSET` entspricht dem Typ `SET OF [0..W-1]`, wobei W die Wortlänge des Rechners ist. Bei `BITSET`–Konstanten entfällt die Angabe des Grundtyps.

2.4.11 `RECORD`

`RECORD`s ermöglichen einen logischen Zusammenhang von Daten unterschiedlicher Typen.

Syntax:

⟨*RecordType*⟩ ::= `RECORD` ⟨*FieldListSequence*⟩ `END`.
⟨*FieldListSequence*⟩ ::= ⟨*FieldList*⟩ {";" ⟨*FieldList*⟩}.
⟨*FieldList*⟩ ::= [⟨*IdentList*⟩ ":" ⟨*type*⟩ |
    `CASE` [⟨*ident*⟩] ":" ⟨*qualident*⟩ `OF` ⟨*variant*⟩ {"—" ⟨*variant*⟩}
    [`ELSE` ⟨*FieldListSequence*⟩] `END`].
⟨*variant*⟩ ::= ⟨*CaseLabelList*⟩ ":" ⟨*FieldListSequence*⟩.
⟨*CaseLabelList*⟩ ::= ⟨*CaseLabels*⟩ {"," ⟨*ConstExpression*⟩.}.
⟨*CaseLabel*⟩ ::= ⟨*ConstExpression*⟩ [".." ⟨*ConstExpression*⟩].

Beispiel :

```
TYPE Person = RECORD
                Name : ARRAY[1..30] of CHAR;
                Geschlecht : (m,w);
                FamStand : (verh, ledig, gesch, verw);
                Gehalt : CARDINAL;
                GebDatum : RECORD
                             Tag : [1..31];
                             Monat : (Jan, Feb, Mae, Apr
                                      Mai, Jun, Jul, Aug,
                                      Sep, Okt, Nov, Dec);
                             Jahr: CARDINAL;
                           END;
              END;

VAR Willi : Person;
    Fussballmannschaft : ARRAY[1..12] of Person;
    ⋮
Willi.Name := 'Meyer';
Willi.Geschlecht := m;
Willi.Gehalt := 2500;
Fussballmannschaft[10].GebDatum.Jahr := 1958;
```

`WITH`-**Anweisung:** Wir hatten im Zusammenhang mit `RECORD`s die `CASE`–Anweisung besprochen. Des weiteren kann man die Zuweisung der Komponenten von `RECORD`s mit der `WITH`–Anweisung kompakter schreiben:

```
WITH Willi DO
    Name := 'Meyer';
    Geschlecht := m;
      ⋮
    Gehalt := 2500;
    WITH GebDatum DO
        Tag := 12;
          ⋮
    END;
END;
```

Syntax:

⟨*WithStatement*⟩ ::= `WITH` ⟨*Designator*⟩ DO ⟨*StatementSequence*⟩ END.

Erweiterung des `RECORD`–Konzepts (`RECORD`s mit Varianten) entnehmen Sie bitte Modula–Büchern.

2.4.12 `POINTER`

Pointer werden in Kapitel 2.7 besprochen.

2.5 Unterprogramme (Prozeduren)

2.5.1 Syntax der `PROCEDURE` Anweisung

Aufgaben, die in verschiedenen Teilprogrammen immer wieder – eventuell mit wechselnden Parametern – benötigt werden, werden als Unterprogramme strukturiert. Gleiches gilt für logisch in sich abgeschlossene Aufgaben.

Syntax (Deklaration einer Prozedur):

⟨*ProcedureDeclaration*⟩ ::= ⟨*ProcedureHeading*⟩ ";" ⟨*block ident*⟩.
⟨*ProcedureHeading*⟩ ::= `PROCEDURE` ⟨*ident*⟩ [⟨*FormaleParameters*⟩].
⟨*block*⟩ ::= {⟨*declaration*⟩} [`BEGIN` ⟨*StatementSequence*⟩] `END`.
⟨*declaration*⟩ ::= `CONST` {⟨*ConstDeclaration*⟩ ";"} |
  `TYPE` {⟨*TypeDeclaration*⟩ ";"} |
  `VAR` {⟨*VariableDeclaration*⟩ ";"} |
  ⟨*ProcedureDeclaration*⟩ ";" |
  ⟨*Module Declaration*⟩ ";".

Beispiel:

$\texttt{PROCEDURE ggt(}\underbrace{\texttt{a,b : CARDINAL, VAR c : CARDINAL}}_{\text{"formale Parameter"}}\texttt{);}$

```
VAR r : CARDINAL;
  BEGIN
    REPEAT
      r := a MOD b;
      a := b;
      b := r;
    UNTIL r = 0;
    c := a;
  END ggT;
```

Möglicher Ausschnitt aus dem Hauptprogramm:

```
x := 10;
ggT(x,15,y);
WriteCard(y,10);    (* Ausgabe = 5 *)
ggT(x,x+1,x);
WriteCard(x,10);    (* Ausgabe = 1 *)
```

2.5.2 Bedeutung

"top–down" Entwurf von Programmen: Der top–down Entwurf ist eine Methodik bei der Erstellung von Programmen, die eine Folge von Daten verarbeiten sollen. Die Folge beginnt mit einem "Header"–Datensatz, der globale Informationen enthält. Dieser wird gefolgt von N ähnlichen, individuellen Datensätzen (Units), die lokale Informationen enthalten.

"top–down" Spezifikation eines Programms:

```
Read Header;
Process Header;
Write Header;
FOR i := 1 TO N DO
    Read Unit;
    Process Unit;
    Write Unit;
END;
```

Damit liegt die Grobstruktur des Programms fest. Im nächsten Schritt werden einzelne Prozeduren ausformuliert, wobei eventuell "tiefer liegende Prozeduren" abgelegt werden. Diese Methode heißt "*schrittweise Verfeinerung*".

"bottom–up" Entwurf von Programmen: Der "bottom–up" Entwurf ist die umgekehrte Methode des "top–down" Entwurfes. Hierbei beginnt man mit einfachsten, elementaren, systemnahen Prozeduren und setzt diese zu immer mächtigeren "Tools" zusammen.

2.5.3 Gültigkeitsbereich der Prozedurparameter und lokal deklarierter Variablen

Der Gültigkeitsbereich der Prozedurparameter (= Parameter der Prozedur) erstreckt sich nur über die Prozedur selbst (und über Prozeduren, die im Deklarationsteil der Prozedur eingeführt werden). Objekte mit gleichem Namen außerhalb der Prozedur sind von der Prozedur *nicht* erreichtbar.

2.5.4 Lebensdauer einer Variable

Die Lebensdauer einer Variable ist in MODULA–2 die Zeit, während der die sie deklarierende Prozedur aktiv ist. Bei Wiederaufruf einer Prozedur geschieht eine neue "*Inkarnation*" der Variable (mit möglicherweise anderem Speicherplatz), d.h. der alte Wert ist im allgemeinen *nicht* verfügbar.

Variablen sollen so lokal wie möglich deklariert werden. Das hilft, Programmierfehler zu vermeiden und Speicherplatz zu sparen ("*pulsierender Speicher*").

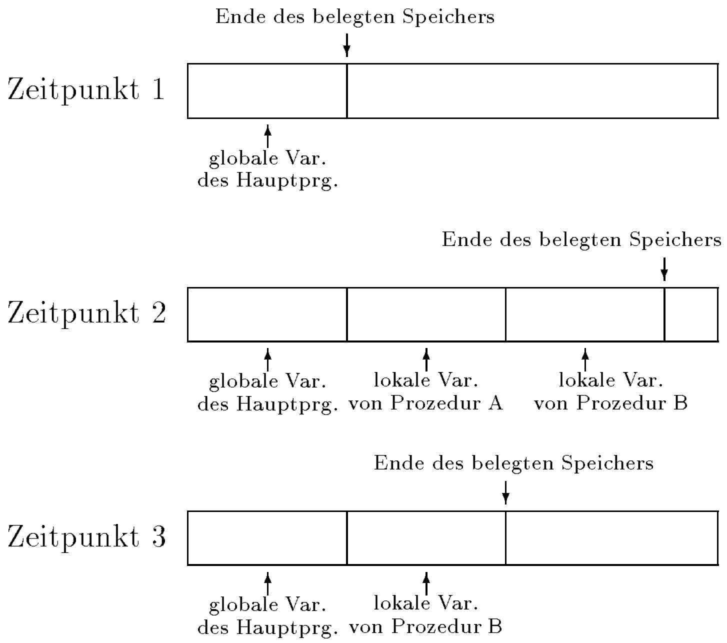
Beispiel:

```
MODULE test;
VAR ...
  ⋮
PROCEDURE A;
  VAR ...
  BEGIN
  ...
  B;
  ...
  END A;
```

```
PROCEDURE B;
  VAR ...
  BEGIN
  ...
  END B;

BEGIN         (* Zeitpunkt 1 *)
  A           (* Zeitpunkt 2 *)
  B           (* Zeitpunkt 3 *)
END test.
```

2.5.5 Speicherbelegung



Dieses Prinzip ist besonders bedeutungsvoll bei *rekursiven* , d.h. sich selbst aufrufenden Prozeduren.

Beispiel: Gäbe es einen Aufruf von A in B,
so wäre A,B rekursiv.

Speicherbelegung z.B.:
globale Variablen des Hauptprogramms
lokale Variablen der 1. Inkarnation von A;
lokale Variablen der 1. Inkarnation von B;
lokale Variablen der 2. Inkarnation von A;
⋮

2.5.6 Parameterübergabe

Man unterscheidet zwei Arten von Parametern:

Werteparameter ("Call by Value")

Beim Aufruf von ggT (`PROCEDURE ggT (a,b : CARDINAL, ...);` vgl. 2.5.1) werden für a und b neue, lokale Variablen eingerichtet, die ihre Startwerte aus dem aufrufenden Programm erhalten.

Nach der Parameterübergabe sind der aktuelle Parameter (im aufrufenden Programm) und der formale Parameter (in der Prozedur) völlig entkoppelt. Die Veränderung der formalen Parameter ist ohne Auswirkung auf das aufrufende Programm. Diese Art der Parameterübergabe dient der Eingabe von Werten.

Variablenparameter ("Call by Reference")

Dem formalen Parameter c (`PROCEDURE ggT (...,VAR c : CARDINAL);` vgl. 2.5.1) wird kein eigener Speicherplatz zugeordnet. Bei dem Aufruf der Prozedur mit dem aktuellem Parameter x wird ein Verweis ("Zeiger", "Pointer", "Reference") von c nach x eingerichtet, so daß jede Aktion innerhalb der Prozedur, die an c gerichtet ist (z.B. Wertzuweisung) eigentlich mit x ausgeführt wird.

Der aktuelle Parameter im aufrufenden Programm muß eine Variable (vom gleichen Typ) und darf kein beliebiger Ausdruck sein wie bei Werteparametern. Variablenparameter sollten nur verwendet werden, wenn eine Wirkung in dem aufrufenden Programm erwünscht ist, sonst nehme man Werteparameter.

Ausnahme: Wenn der Parameter z.B. ein großes `ARRAY` ist, kann damit die Deklaration eines zweiten, lokalen `ARRAY`s verhindert werden, welche bei der Parameterübergabe beim Prozeduraufruf neben der Umspeicherung des aktuellen `ARRAY`s vorgenommen werden müßte.

Achtung: Wenn der aktuelle Variablenparameter eine `ARRAY`–Komponente ist, so wird der Index nur einmal bei der Parameterübergabe ausgewertet.

Beispiel:

```
VAR a : ARRAY[1..10] OF REAL;
    i : CARDINAL;

PROCEDURE test(VAR x : REAL);
  BEGIN
    i := i+1;
    WriteReal(x,10);
  END test;

BEGIN      (* Hauptprogramm *)
  a[1] := 2,5;
  a[2] := 7,8;
  i    := 1;
  test(a[i]);
END.
```

führt auf Ausgabe 2,5 (und nicht 7,8)

Die Anweisung `i := i+1` erzeugt eine Wirkung auf eine Variable des aufrufenden Programms, welche nicht über die Parameterliste geschieht. So etwas heißt *Seiteneffekt*. Seiteneffekte sollten grundsätzlich vermieden werden!

2.5.7 Funktionsprozeduren

Funktionsprozeduren sind spezielle Prozeduren, die einen Ausgabewert oder "Funktionswert" liefern. Funktionsprozeduren werden im aufrufenden Programm syntaktisch wie ein Ausdruck und nicht wie eine Anweisung behandelt.

Syntax:

`PROCEDURE` ⟨*Prozedur–Bezeichner*⟩ (⟨*Parameterliste*⟩): ⟨*Typ–Bezeichner*⟩;

Im Inhalt des Prozedurkörpers muß mindestens einmal die Anweisung `RETURN` mit einem nachfolgenden Ausdruck (vom entsprechenden Typ) erscheinen. Dieser Ausdruck wird ausgewertet und liefert den Wert der Prozedur. Die Prozedur wird daraufhin verlassen.

Achtung: Strukturierte Typen sind als Ausgabewerte der Prozedur verboten.

Beispiel:

1.
```
PROCEDURE ggt (a,b : CARDINAL) : CARDINAL;
  VAR r : CARDINAL;
  BEGIN
    LOOP
      r := a MOD b;
      IF r = 0 THEN RETURN b END;
      a := b;
      b := r;
    END
  END ggt;
```

2.
```
PROCEDURE Wurzel (x : REAL) : REAL;
(* bestimmt Quadratwurzel aus positiven REAL-Zahlen nach
  Biselektionsmethode *)
  CONST eps = 0.000001;   (* Genauigkeit *)
  VAR u,o,m : REAL;
```

```
    BEGIN
      u := 0.0;
      o := x;
      WHILE o-u > eps DO
        m := (o+u) / 2.0;
        IF m*m > x THEN o := m
          ELSE u := m
        END;
      END;
    RETURN (o+u) / 2.0;
                 m
  END Wurzel;
```

Bemerkung: Die Anweisung `RETURN` (ohne nachfolgenden Ausdruck) kann auch in der eigentlichen Prozedur (d.h. nicht Funktionsprozedur) verwendet werden, um die Prozedur vorzeitig zu verlassen.

2.5.8 Offene `ARRAY`–Parameter

Ist der formale Parameter ein `ARRAY`, so muß der aktuelle Parameter auch ein `ARRAY` des selben Typs mit den selben Indexgrenzen sein. Diese Strenge in der Typanbindung kann durch sogenannte "offene `ARRAY`–Parameter" durchbrochen werden. Bei offenen `ARRAY`–Parametern werden die Indexgrenzen beim formalen Parameter weggelassen.

Beispiel: `PROCEDURE P (VAR a : ARRAY OF REAL);`

Innerhalb von `P` ist `a` als `ARRAY` mit unterer Indexgrenze 0 organisiert. Die obere Indexgrenze ergibt sich beim Aufruf von `P` als (obere Grenze – untere Grenze) das aktuellen `ARRAY`s. Die Differenz kann zur Laufzeit durch die Standardprozedur `HIGH` erfragt werden.

Beispiel:

Das Hauptprogramm enthalte die folgende Deklaration:

```
VAR b : ARRAY [5..10] OF REAL;
    c : ARRAY [-10..20] OF REAL;
```

Die Prozedur P sei folgendermaßen definiert:

```
PROCEDURE P (VAR a : ARRAY OF REAL);
  VAR i : CARDINAL;
  BEGIN
    FOR i := 0 TO HIGH(a) DO
      WriteReal( a[i],10 )
    END;
  END P;
```

Durch die Aufrufe `P(b)`; bzw. `P(c)`; im Hauptprogramm wird erreicht, daß beide `ARRAY`s ausgegeben werden:

Im ersten Fall gilt : `HIGH(a)` = 10 - 5 = 5,
im zweiten Fall gilt : `HIGH(a)` = 20 - (-10) = 30.

Zuordnung der `ARRAY`s:

a[0]	→	b[5]	a[0]	→	c[-10]
a[1]	→	b[6]	a[1]	→	c[-9]
a[2]	→	b[7]	a[2]	→	c[-8]
	⋮			⋮	
a[5]	→	b[10]	a[30]	→	c[20]

2.5.9 Prozedurtypen und Prozedurvariablen

In MODULA–2 ist die Verwendung von Prozedurvariablen in gleicher Weise erlaubt wie die Verwendung von `REAL`– oder `CARDINAL`–Variablen erlaubt sind.

Beispiel :

```
  ⋮
TYPE Funkt = PROCEDURE (REAL) : REAL;
     Proc  = PROCEDURE (VAR REAL);

VAR x : Funkt;
    y : Proc;
    z,w : REAL;
  ⋮
BEGIN      (* Hauptprogramm *)
  x := sin;    (* aus MathLib importieren *)
  y := ReadReal;     (* aus RealInOut importieren *)
  y(z);
  w := x(3.14);
```

Prozedurvariablen müssen wie alle Variablen deklariert werden. Die Deklaration eines Prozedurtyps (in `TYPE`–Vereinbarung) beginnt mit dem Wort `PROCEDURE`. In der darauf folgenden Liste müssen alle Parametertypen durch "," getrennt aufgeführt werden. Falls Variablenparameter vorkommen, muß auch `VAR` in dieser Liste an entsprechender Position erscheinen.

Anwendung: Prozeduren erhalten Parameter vom Prozedurtyp.

Beispiel: Eine noch unbekannte Funktion f soll im Intervall [a,b] numerisch integriert werden. Wir teilen dazu $[a,b]$ in n Intervalle der Länge $\Delta x = \frac{b-a}{n}$ und nähern $\int\limits_a^b f(x)\,dx$ durch $\sum\limits_{i=0}^{n-1} \Delta x * f(x + \frac{\Delta x}{2} + i * \Delta x)$ an.

Programmstück:

```
TYPE FUNKTION = PROCEDURE (REAL) : REAL;
  ⋮
PROCEDURE integral (a,b : REAL, f : FUNKTION, n : CARDINAL): REAL;
  VAR s, deltaX : REAL;
    i : CARDINAL;

BEGIN
  deltaX := (b-a) / FLOAT(n);
  s := 0.0;
  FOR i := 0 TO n-1 DO
    s := s+deltaX * f(a+deltaX/2.0 + FLOAT(i)*deltaX)
  END;
  RETURN s;
END integral;
  ⋮
BEGIN      (* Hauptprogramm *)
  WriteReal( integral(0,3.14,sin,100),20);
  WriteReal( integral(0.5,2.0,cos,200),20);
    ⋮
```

2.6 Rekursive Prozeduren

2.6.1 Definition

Rekursive Prozeduren sind Prozeduren, die sich selbst, eventuell indirekt über weitere Prozeduren, aufrufen. Durch verschiedene rekursive Aufrufe derselben Prozedur kann es zu einem Zeitpunkt im Ablauf des Programms verschiedene sogenannte "*Inkarnationen*" der Prozedur (Reservierung von Speicherplatz für lokale Variablen und Aufruf der Prozedur) geben. Die verschiedenen aktiven Inkarnationen besitzen jeweils *eigenständige* Speicherbereiche für die lokalen Variablen und Parameter.

Typischer Aufbau einer rekursiven Prozedur:

```
PROCEDURE p(...);
  ⋮
BEGIN
  IF <Rekursionsende erreicht>
    THEN <nicht rekursiver Teil>
    ELSE p(...);     (* ein oder mehrere rekursive Aufrufe *)
END;
```

Der Test, ob das Rekursionsende erreicht ist, richtet sich meistens an einen Parameter in der Prozedur:

```
PROCEDURE p(n : CARDINAL, ...);
  ⋮
BEGIN
  IF n = 0
    THEN <nicht rekursiver Teil>
    ELSE p(n-1,...);    (* ein oder mehrere Aufrufe von
                           p mit Parameterwert(en) <n *)
    END
END p;
```

2.6.2 Ackermann–Funktion

Die Ackermann–Funktion ist eine “hoch”–rekursive Funktion. Sie kann jede durch primitive Rekursion erzeugte Funktion majorisieren.
→ Übung
→ Vorlesung : Berechnungstheorie

Definition:

$$\begin{array}{lll} A(0,y) & = y+1 & \\ A(x,0) & = A(x\text{-}1,1) & \text{für } x>0 \\ A(x,y) & = A(x\text{-}1,A(x,y\text{-}1)) & \text{für } x>0,\ y>0 \end{array}$$

Die Funktion ist wohldefiniert: Jede Eingabe (x,y) endet nach endlich vielen Schritten beim Rekursionsanfang (erste Zeile der Definition).

Beispiel:

$$\begin{array}{ll} A(1,2) & = A(0,A(1,1)) \\ & = A(0,A(0,A(1,0))) \\ & = A(0,A(0,A(0,1))) \\ & = A(0,A(0,2)) \\ & = A(0,3) \\ & = 4 \end{array}$$

Die Ackermann–Funktion hat ein unvorstellbares Wachstum. A(4,2) hat beispielsweise 19729 Dezimalstellen!

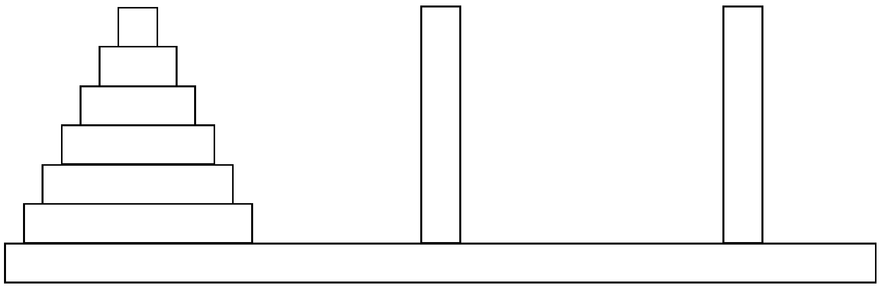
Aus der rekursiven Definition folgt die rekursive Prozedur :

```
PROCEDURE A(x,y : CARDINAL) : CARDINAL;
BEGIN
  IF x = 0 THEN RETURN y+1
    ELSIF y = 0 THEN RETURN A(x-1,1)
    ELSE RETURN A(x-1,A(x,y-1))
  END;
END A;
```

Bemerkung: Die Prozedur benötigt *keine* lokalen Variablen. "Zwischenergebnisse" werden durch die Existenz der verschiedenen Inkarnationen der Prozedur gespeichert. Diese Art der Programmierung heißt *funktionales Programmieren.* Es gibt keine lokalen Variablen; stattdessen werden zur Berechnung von Funktionswerten lediglich weitere Funktionsprozeduren aufgerufen (eventl. rekursiv). Ein Beispiel einer funktionalen Programmiersprache ist LISP (Vgl. 1.6.7).

2.6.3 Türme von Hanoi – Einpersonenspiel

Problem:



Bewege den Stapel von links nach rechts. In jedem Zug darf genau ein Ring bewegt werden, so daß nie ein größerer Ring auf einem kleineren Ring liegt.

Rekursive Lösungsidee:

Die Turmaufgabe für 0 Ringe ist trivial (Rekursionsanfang).

Um die Turmaufgabe mit n $>$ 0 Ringen zu lösen, bewege die oberen n-1 Ringe von dem linken Stab auf den mittleren Stab (Rekursionsvoraussetzung).

Bewege den unteren Ring von links nach rechts, bewege wieder unter Ausnutzung der Rekursion den Turm der Höhe n-1 von der Mitte nach rechts (Rekursionsschritt).

Programmbeispiel:

```
MODULE Hanoi;

FROM InOut IMPORT WriteString, WriteLn, ReadCard;
TYPE Platz[1..3];    (* Startposition *)
VAR hoehe : CARDINAL;    (* Höhe des Turms *)

PROCEDURE Drucke(x : Platz);
BEGIN
  CASE x OF
    1 : WriteString('links') |
    2 : WriteString('mitte') |
    3 : WriteString('rechts') |
  END    (* of CASE *)
END Drucke;
```

```
PROCEDURE bewegeTurm(h : CARDINAL; a,b : Platz);
(* gibt aus, wie Turm der Höhe h von a nach b bewegt wird *)
BEGIN
  IF h > 0 THEN
    bewegeTurm(h-1,a,6-a-b);
    WriteString('Lege eine Scheibe von ');
    Drucke(a);
    WriteString(' nach ');
    Drucke(b);
    WriteLn;
    bewegeTurm(h-1,6-a-b,b);
  END
END bewegeTurm;

BEGIN    (* Hauptprogramm *)
  WriteString('Höhe = ');
  ReadCard(hoehe);
  WriteLn;
  bewegeTurm(hoehe,1,3);
END Hanoi.
```

Die Eingabe 4 führt zu folgender Ausgabe:

```
Lege eine Scheibe von  links   nach  mitte
Lege eine Scheibe von  links   nach  rechts
Lege eine Scheibe von  mitte   nach  rechts
Lege eine Scheibe von  links   nach  mitte
Lege eine Scheibe von  rechts  nach  links
Lege eine Scheibe von  rechts  nach  mitte
Lege eine Scheibe von  links   nach  mitte
Lege eine Scheibe von  links   nach  rechts
Lege eine Scheibe von  mitte   nach  rechts
Lege eine Scheibe von  mitte   nach  links
Lege eine Scheibe von  rechts  nach  links
Lege eine Scheibe von  mitte   nach  rechts
Lege eine Scheibe von  links   nach  mitte
Lege eine Scheibe von  links   nach  rechts
Lege eine Scheibe von  mitte   nach  rechts
```

Anmerkung: Es gibt auch einen einfachen, nicht rekursiven Algorithmus:

```
LOOP
  <bewege kleinste Scheibe um eine Position im
    ''Uhrzeigersinn''(1→2,2→3,3→1)>
  IF <Aufgabe erledigt> THEN EXIT END;
  <mache den einzigen möglichen Zug, der nicht die kleinste
    Scheibe betrifft>
END;
```

2.6.4 Vorgehensweise beim Schreiben rekursiver Prozeduren

Bevor der Algorithmus ausprogrammiert wird, wird eine genaue Spezifikation, die sogenannte "Schnittstellenbeschreibung" vorgenommen. Sie gibt an, was der Algorithmus tun soll und welche Rolle die Parameter spielen. Die Schnittstellenbeschreibung kann später als Kommentar eingefügt werden.

Auf der Basis dieser Schnittstellenbeschreibung wird der rekursive Algorithmus dann konstruiert. Zu Beginn wird zuerst der Rekursionsanfang, in dem keine Rekursion mehr stattfindet, (meist) mit einer `IF`–Abfrage abgefragt. Dann folgt die Programmierung des rekursiven Zweiges. Dabei wird so getan, als ob die gerade zu programmierende rekursive Prozedur bereits existiert. Deren Inneres ist aber bis auf die durch die Schnittstellenbeschreibung eindeutig festgelegte Wirkung unbekannt. Auf die Korrektheit dieser Prozedur vertrauend, wird der rekursive Zweig ausprogrammiert. (Analogie: Induktionsbeweis !)

2.6.5 Mini–Compiler

Rekursive Prozeduren eignen sich gut zum Schreiben von "recursive descent" Compilern.

Hierbei wird jedem Syntaxdiagramm (jeder BNF–Formel) eine Prozedur zugeordnet. Sobald im Diagramm ein Name eines anderen Diagramms auftaucht, so bedeutet das auf Prozedurebene nichts anderes als ein Prozeduraufruf (evtl. rekursiv).

Sobald Eingabezeichen (wie bei ovalen Kästchen) zu verarbeiten sind, werden die entsprechenden Zeichen vom Eingabegerät gelesen. Bei Alternativverzweigungen (Bsp.: `ELSE/ELSIF`) muß man sich das nächste (die k nächsten) zu verarbeitende Zeichen ansehen, um zu entscheiden, welchem Pfad durch das Syntaxdiagramm zu folgen ist (Vorgehensweise: *look ahead*. Programmiersprachen, bei denen dieses look ahead stets einwandfrei funktioniert heißen LL(1)–Sprachen).

Wir wollen einen Mini–Compiler für die Auswertung von arithmetischen Wertzuweisungen schreiben:

Syntax: Wertzuweisung

$\langle assignment \rangle$ ::= $\langle designator \rangle$ ":=" $\langle expression \rangle$.

- Variablen sollen aus kleinen Buchstaben a,...,z bestehen.
- Wir definieren Ausdrücke so, daß unser Mini–Compiler die "Punkt–vor–Strich–Regel" kennt: $a + b * c$ bedeutet also $a + (b * c)$ und nicht $(a + b) * c$.

Wir zerlegen dazu den Ausdruck in Terme und Faktoren:

Ausdruck:

$\langle Ausdruck \rangle$ ::= $\langle Term \rangle$ { $\langle AddOperator \rangle$ $\langle Term \rangle$ }.

$\langle AddOperator \rangle$ ::= "+" | "-".

Term:

$\langle Term\rangle$ ::= $\langle Faktor\rangle$ { $\langle MulOperator\rangle$ $\langle Faktor\rangle$ }.

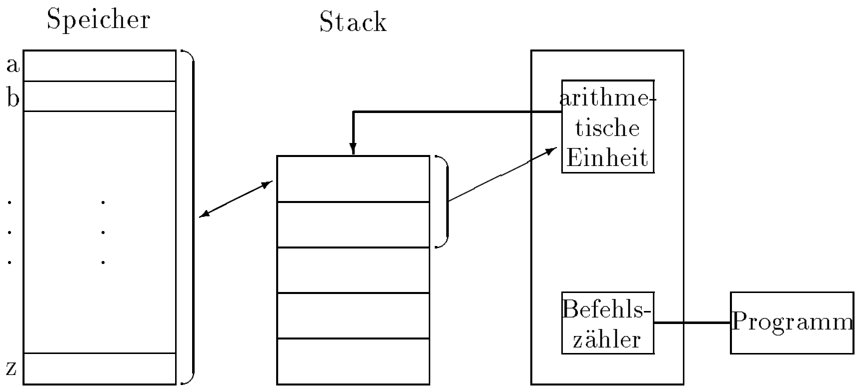
$\langle MulOperator\rangle$ ::= "*" | "/".

Faktor:

$\langle Faktor\rangle$ ::= $\langle Variable\rangle$ | "(" $\langle Ausdruck\rangle$ ")".

Wir haben damit die Syntax der "Programmiersprache" festgelegt.

Nun legen wir die *Zielmaschine* fest, auf der das übersetzte Programm laufen soll. Dazu stellen wir uns folgende Architektur vor:



Speicher: Er dient zur Aufnahme der Variablen a,...,z.

Stack: Der Stack ist eine Folge von Registern, die als Kellerspeicher ("first in – last out", LIFO) organisiert sind, d.h. lediglich auf das oberste Register ist zugreifbar. Beim Einspeichern verschieben sich alle Speicherinhalte nach "unten", beim Herausnehmen nach "oben".

mögliche Operationen:

- STORE x	:	Oberstes Stackelement wird entfernt und in den Variablenspeicher unter x abgelegt.
- LOAD x	:	Die Variable x wird vom Speicher geholt und in den Stack geschrieben.
- ADD	:	Die obersten beiden Stackelemente werden herausgeholt und in der Arithmetikeinheit addiert. Das Ergebnis wird im Stack abgelegt.
- SUB	:	Analog wie ADD.
- MULT	:	Analog wie ADD.
- DIV	:	Analog wie ADD.

Maschinenprogramm: Als Maschinenprogramm bezeichnet man eine Folge von Befehlen aus der obigen Liste, die der Reihe nach abgearbeitet werden. Unser Mini–Compiler soll a := b+c*d in folgendes Maschinenprogramm überführen:

LOAD b
LOAD c
LOAD d

Stack:
d
c
b

MULT

Stack:
$c * d$
b

ADD

Stack:
$b + c * d$

STORE a

falsch wäre:
LOAD b
LOAD c

Stack:
c
b

ADD

Stack:
$b + c$

LOAD d

Stack:
d
$b + c$

MULT

Stack:
$(b + c) * d$

STORE a

Mini–Compiler

```
MODULE Compiler;

FROM InOut IMPORT WriteCard, ReadString, WriteString,
                  WriteLn;
VAR i : CARDINAL;
    next : CHAR;
    s : ARRAY[1..80] OF CHAR;

PROCEDURE Lese;
  BEGIN
    INC(i);
    next := s[i]
  END Lese;

PROCEDURE Wertzuweisung;
  VAR ch : CHAR;
  BEGIN
    CASE next OF
      'a',...'z' : ch := next;
                   Lese;
                   IF next <>':' THEN Fehler END;
                   Lese;
                   IF next <>'=' THEN Fehler END;
                   Lese;
                   Ausdruck;
                   Ausgabe(6,ch)
             ELSE : Fehler
    END
  END Wertzuweisung;

PROCEDURE Ausdruck;
  BEGIN
    Term;
    LOOP
      CASE next OF
        0C,')' : RETURN|
           '+' : Lese;
                 Term;
                 Ausgabe(1,' ')|
           '-' : Lese;
                 Term;
                 Ausgabe(2,' ')|
          ELSE : Fehler
      END
```

```
    END
  END Ausdruck;

PROCEDURE Term;
  BEGIN
    Faktor;
    LOOP
      CASE next OF
        0C,'+','-',')' : RETURN|
                   '*' : Lese;
                         Faktor;
                         Ausgabe(3,' ')|
                   '/' : Lese;
                         Faktor;
                         Ausgabe(4,' ')|
                  ELSE : Fehler
      END
    END
  END Term;

PROCEDURE Faktor;
  BEGIN
    CASE next OF
      'a',..,'z' : Ausgabe(5,next);
                   Lese;
                   RETURN|
             '(' : Lese;
                   Ausdruck;
                   IF next <>')' THEN Fehler
                   ELSE Lese; RETURN
                   END|
            ELSE : Fehler;
    END
  END Faktor;

PROCEDURE Fehler;
  BEGIN
    WriteLn;
    WriteString('Fehler in Position');
    WriteCard(i,2);
    HALT
  END Fehler;

PROCEDURE Ausgabe(n : CARDINAL, c : ARRAY OF CHAR);
  BEGIN
    CASE n OF
```

```
        1 : WriteString('ADD')|
        2 : WriteString('SUB')|
        3 : WriteString('MULT')|
        4 : WriteString('DIV')|
        5 : WriteString('LOAD')
            WriteString(c)|
        6 : WriteString('STORE')|
            WriteString(c)
      END
      WriteLn
    END Ausgabe;

  BEGIN    (* Hauptprogramm *)
    ReadString(s);
    i := 0;
    Lese;
    WriteLn;
    Wertzuweisung;
    IF next <> 0C THEN Fehler END
  END Compiler.
```

Unser Mini–Compiler produziert z.B. bei der Eingabe

a := b-c*d+(c*(e+f))/h

das folgende Programm:

LOAD b
LOAD c
LOAD d
MULT
SUB
LOAD c
LOAD e
LOAD f
ADD
MULT
LOAD h
DIV
ADD
STORE a

Wir können uns die rekursive "Aufrufstruktur" in dem folgenden Ablaufprotokoll veranschaulichen, bei dem jede Rekursionstiefe eingerückt wird. (CALL ... bedeutet Aufruf, RETURN ... Ausstieg; Ausgaben des Programms werden durch ⇒ gekennzeichnet).

```
  a := b-c*d+(c*(e+f))/h

CALL Wertzuweisung
  CALL Ausdruck
    CALL Term
      CALL Faktor
        ⇒ LOAD b
      RETURN Faktor
    RETURN Term
    CALL Term
      CALL Faktor
        ⇒ LOAD c
      RETURN Faktor
      CALL Faktor
        ⇒ LOAD d
      RETURN Faktor
      ⇒ MULT
    RETURN Term
    ⇒ SUB
    CALL Term
      CALL Faktor
        CALL Ausdruck
          CALL Term
            CALL Faktor
            ⇒ LOAD c
            RETURN Faktor
            CALL Faktor
              CALL Ausdruck
                CALL Term
                  CALL Faktor
                  ⇒ LOAD e
                  RETURN Faktor
                RETURN Term
                CALL Term
                  CALL Faktor
                  ⇒ LOAD f
                  RETURN Faktor
                RETURN Term
                ⇒ ADD
              RETURN Ausdruck
            RETURN Faktor
            ⇒ MULT
          RETURN Term
        RETURN Ausdruck
      RETURN Faktor
```

CALL Faktor
⇒ LOAD h
RETURN Faktor
⇒ DIV
RETURN Term
⇒ ADD
RETURN Ausdruck
⇒ STORE a
RETURN Wertzuweisung

Übungsaufgabe: Erweitere den Mini–Compiler um weitere Programmstrukturen.

Weitere Zielmaschinenbefehle:

- GOTO n : Ausführung eines unbedingten Sprungs zu der Anweisung mit der Marke n.
- JZERO n : Falls oberstes STACK–Element gleich 0, springe zu der Anweisung mit der Marke n. Das oberste STACK–Element wird dabei entfernt.

Befehle im Maschinenprogramm können durch paarweise verschiedene Marken (n–Anweisung) markiert werden.

- Eine `IF`–Anweisung der Form :

  `IF` Ausdruck `THEN` Anweisung `ELSE` Anweisung `END`

  sollte übersetzt werden in:

  Maschinencode für Ausdruck
  JZERO n_1
  Maschinencode für Anweisung$_1$
  GOTO n_2
  n_1 : Maschinencode für Anweisung$_2$
  n_x : ...

- Eine `WHILE`–Schleife der Form :

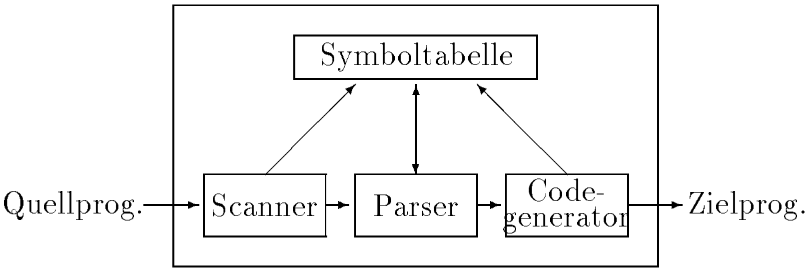
  `WHILE` Ausdruck `DO` Anweisung `END`

  sollte übersetzt werden in :

  n_1 : Maschinencode für Ausdruck
  JZERO n_2
  Maschinencode für Anweisung
  GOTO n_1
  n_2 : ...

Grundsätzlicher Aufbau eines Compilers

(→ Vorlesung Compilerbau, Syntaxanalyse)



lexikalische Analyse

$\boxed{\text{Quellprogramm}} \xrightarrow{\text{Scanner}} \boxed{\text{Folge von Token}}$

Token sind grundlegende Symbole der Quellsprache (z.B. Schlüsselworte, Bezeichner, ...).

Der Scanner liest das Quellprogramm und erzeugt eine Folge von Token.

syntaktische Analyse

$\boxed{\text{Folge von Token}} \xrightarrow{\text{Parser}} \boxed{\text{Ableitungsbaum}}$

Der Parser enthält eine Folge von Token und produziert einen Ableitungsbaum entsprechend der Syntax der Quellsprache.

semantische Analyse, Codegenerierung und –optimierung

$\boxed{\text{Ableitungsbaum}} \xrightarrow{\text{Codegenerator}} \boxed{\text{Zielprogramm}}$

Die semantische Analyse wird anhand des Ableitungsbaumes durchgeführt (z.B. Variablendeklaration; werden Variablen entsprechend ihres Typs verwendet? ; ...)

Der Codegenerator wird vorbereitet.

Der Codegenerator erzeugt ein Zielprogramm, das im allgemeinen noch optimiert wird (z.B. im Hinblick auf den benötigten Speicherplatz, die Anzahl der Befehle, ...).

Man unterscheidet *Mehr–Pass–Compiler* (die Prozesse laufen getrennt ab) und *Ein–Pass–Compiler* (wie z.B. unser Mini–Compiler).

2.6.6 Rekursion versus Iteration

Rekursive Prozeduren lassen sich stets nicht-rekursiv, also *iterativ* realisieren, z.B. durch manuelles Programmieren des “pulsierenden Speichers” mit einem oder mehreren Stacks.

Beispiel: Die Ackermann–Funktion

Wir geben eine iterative Prozedur zur Berechnung der Ackermann–Funktion $A(x, y)$ unter der Verwendung eines Stacks s an. Der Stack s wird der Einfachheit halber als `ARRAY` programmiert. Ist zu einem Zeitpunkt der Stackinhalt $s[k]\ s[k-1]\ \ldots\ s[1]$ so bedeutet das, daß noch $A(s[1], A(s[2], \ldots A(s[k-1], s[k]) \ldots))$ zu berechnen ist. Wir initialisieren den Stack mit $s[1] := x$ und $s[2] := y$. Die Rechnung ist beendet, falls der Stack die Höhe 1 hat. Das Ergebnis steht dann in $s[1]$.

```
PROCEDURE A(x,y: CARDINAL): CARDINAL;
CONST maxstacksize=1000;
VAR s: ARRAY [1..maxstacksize] OF CARDINAL;
    stacktop: CARDINAL;
BEGIN
  s[1] := x; s[2] := y; stacktop := 2;
  WHILE stacktop>1 DO
    IF s[stacktop-1] = 0 THEN
      s[stacktop-1] := s[stacktop]+1;
      DEC(stacktop);
    ELSIF s[stacktop] = 0 THEN
      s[stacktop] := 1;
      DEC(s[stacktop-1]);
    ELSE
      s[stacktop+1] := s[stacktop]-1;
      s[stacktop] := s[stacktop-1];
      DEC(s[stacktop-1]);
      INC(stacktop);
    END; (* IF *)
  END; (* WHILE *)
  RETURN s[1];
END A;
```

Bemerkung: Die rekursive Lösung ist meist eleganter und durchschaubarer. Die iterative Lösung ist vorzuziehen, falls die rekursive Lösung ineffizient ist.

Beispiel: Fibonacci-Funktion

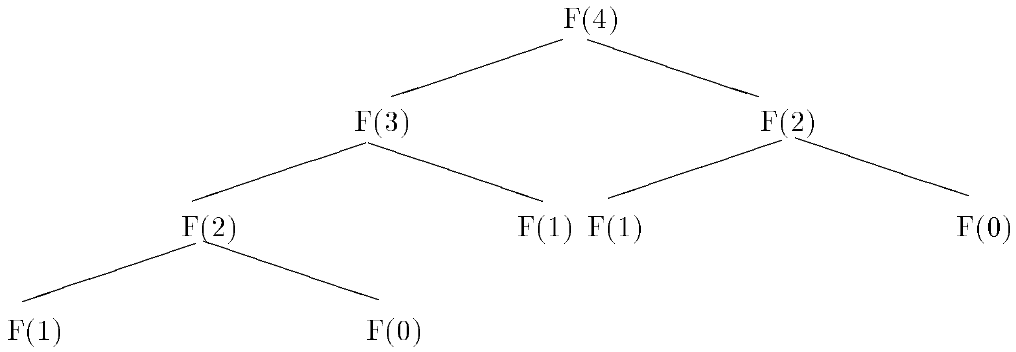
Rekursive Definition:

$$F(0) = 0$$
$$F(1) = 1$$
$$F(n) = F(n-1) + F(n-2)$$

- Direkte Umsetzung in rekursive Prozedur:

```
PROCEDURE F(n: CARDINAL): CARDINAL;
BEGIN
  IF n <= 1 THEN RETURN n
  ELSE RETURN(F(n-1)+F(n-2))
  END;
END F;
```

Diese Prozedur ist sehr ineffizient, bei Aufruf von $F(n)$ werden zwei voneinander unabhängige Aufrufe (von $F(n-1)$ und $F(n-2)$) ausgelöst, die ihrerseits wieder verzweigen.
Der resultierende Zeitaufwand, d.h. die Anzahl der zu bearbeitenden Aufrufe, ist exponentiell in n:
z.B. Aufrufstruktur für $F(4)$



- iterative Lösung: Die iterative Lösung benötigt nur n viele Berechnungsschritte:

```
PROCEDURE F(n: CARDINAL): CARDINAL;
VAR a,b,i: CARDINAL; BEGIN
  IF n=0 THEN RETURN 0
  a:=0; b:=1;
  FOR i:=1 TO n-1 DO
    a:=a+b;
    b:=b-a; (* altes a *)
  END;
  RETURN b;
END F;
```

- Es gibt sogar eine Lösung, die mit $\log n$ vielen Schritten auskommt:

Idee: es gilt: $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \begin{pmatrix} F(n-1) & F(n) \\ F(n) & F(n+1) \end{pmatrix}$ für $n > 0$

Beweis: durch vollständige Induktion (Übungsaufgabe)

Die n-te Potenz von $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}$ kann rekursiv berechnet werden:
$\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n = \left(\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{\frac{n}{2}}\right)^2$, d.h. berechne rekursiv $\begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^{\frac{n}{2}}$ und bilde dann das Quadrat dieser Matrix.
Ist $T(n)$ die Berechnungsformel für dieses Verfahren, ausgewertet auf einer $n \times n$ Matrix, so gilt:
$T(n) = T(\frac{n}{2}) + const$ also $T(n) = c \cdot \log_2 n + d$ für geeignete Konstanten c und d.
$\Rightarrow$ die rekursive Methode muß nicht ineffizient sein!

2.6.7 Divide and Conquer

Rekursive Prozeduren dienen der Implementierung des „Teile und Herrsche“-Prinzips („Divide and Conquer“).
Beispiel: *Multiplikation zweier n-ziffriger Zahlen*

- **„Schulmethode“**
  Zur Multiplikation zweier n-stelliger Zahlen wird eine quadratische Tabelle der Länge n erzeugt:

1	2	3	4	*	5	6	7	8
		6	1	7	0			
			7	4	0	4		
				8	6	3	8	
					9	8	7	2
		7	0	0	6	6	5	2

  Der Aufwand ist im wesentlichen n^2 und als Elementaroperationen werden die Ausführung 1-ziffriger Multiplikationen und Additionen benötigt („kleines 1×1“).

- **Divide and Conquer nach Karatsuba**
  Wir zerlegen zwei n-ziffrige Zahlen x und y in zwei gleiche Teile x_1, x_2 und y_1, y_2, wobei wir annehmen, daß $n = 2^x$.
  Dann kann man x und y folgendermaßen darstellen: $x = x_1 \cdot b^{\frac{n}{2}} + x_2$ und $y = y_1 \cdot b^{\frac{n}{2}} + y_2$, wobei b die Basis des zugrundeliegenden Zahlensystems ist. Es gilt nun:
$$\begin{aligned} x \cdot y &= (x_1 b^{\frac{n}{2}} + x_2) \cdot (y_1 b^{\frac{n}{2}} + y_2) \\ &= x_1 y_1 b^n + (x_1 y_2 + x_2 y_1) \cdot b^{\frac{n}{2}} + x_2 y_2 \\ &= x_1 y_1 b^n + (x_1 y_1 + x_2 y_2 - (x_2 - x_1) \cdot (y_2 - y_1)) \cdot b^{\frac{n}{2}} + x_2 y_2 \end{aligned}$$

  Wir haben die Multiplikation von Zahlen der Länge n auf drei Multiplikationen der Länge $\frac{n}{2}$ zurückgeführt, d.h. mit drei Aufrufen des Multiplikationsalgorithmus für Zahlen der Länge $\frac{n}{2}$ erhält man einen Multiplikationsalgorithmus für Zahlen der Länge n.

Aufwand:
Sei $A(n)$ der Aufwand des Multiplikationsalgorithmus für Zahlen der Länge n. Für Additionen und Subtraktionen mit „Shifts“ (Multiplikationen mit $b^{\frac{n}{2}}$) genügt linearer Aufwand. Man erhält also für $A(n)$:

$$A(n) = 3 \cdot A(\frac{n}{2}) + c \cdot n$$

$$A(1) = a$$

Behauptung: $A(n) = (a + 2c) \cdot n^{\log_2 3} - 2 \cdot c \cdot n$

Beweis: *durch vollständige Induktion.*

I.Anfang: $A(1) = (a + 2c) - 2c = a$

I.Voraussetzung: Behauptung gelte für $k = \frac{n}{2}$

I.Schritt:
$$\begin{aligned} A(n) &= 3 \cdot A(\tfrac{n}{2}) + c \cdot n \\ &= 3 \cdot ((a + 2c) \cdot (\tfrac{n}{2})^{\log_2 3} - 2 \cdot c \cdot \tfrac{n}{2}) + cn \\ &= \tfrac{3}{2^{\log_2 3}}(a + 2c) \cdot n^{\log_2 3} - 3 \cdot c \cdot n + c \cdot n \\ &= (a + 2c) \cdot n^{\log_2 3} - 2 \cdot c \cdot n \end{aligned}$$

Anwendungsbeispiel: *Matrizenmultiplikation*

- **Schulmethode** Drei ineinandergeschachtelte `FOR`-Schleifen mit einem Aufwand von n^3
- **schnelle Matrizenmultiplikation nach Strassen**
  Seien $A, B\, n \times n$ Matrizen, $n = 2^k$, A_{ij}, B_{ij} $\frac{n}{2} \times \frac{n}{2}$ Matrizen.

$$A = \begin{pmatrix} A_{11} & A_{12} \\ A_{21} & A_{22} \end{pmatrix} \qquad B = \begin{pmatrix} B_{11} & B_{12} \\ B_{21} & B_{22} \end{pmatrix}$$

Dann gilt:

$$A \cdot B = \begin{pmatrix} A_{11}B_{11} + A_{12}B_{21} & A_{11}B_{12} + A_{12}B_{22} \\ A_{21}B_{11} + A_{22}B_{21} & A_{21}B_{12} + A_{22}B_{22} \end{pmatrix}$$

Wir können die $n \times n$ Matrixmultiplikation durch sieben Aufrufe von $\frac{n}{2} \times \frac{n}{2}$ Matrixmultiplikationen und die durch linearen Aufwand zu realisierenden Additionen und Subtraktionen ausführen. Seien $P_1, \ldots, P_7$ wie folgt definiert:

$$P_1 = (A_{12} - A_{22}) \cdot (B_{21} + B_{22})$$

$$P_2 = (A_{11} + A_{22}) \cdot (B_{11} + B_{22})$$

$$P_3 = (A_{11} - A_{21}) \cdot (B_{11} + B_{12})$$

$$P_4 = (A_{11} + A_{12}) \cdot B_{22}$$

$$P_5 = A_{11} \cdot (B_{12} - B_{22})$$

$$P_6 = A_{22} \cdot (B_{21} - B_{11})$$
$$P_7 = (A_{21} + A_{22}) \cdot B_{11}$$

Dann gilt:

$$A_{11}B_{11} + A_{12}B_{21} = P_1 + P_2 - P_4 + P_6$$
$$A_{11}B_{12} + A_{12}B_{22} = P_4 + P_5$$
$$A_{21}B_{11} + A_{22}B_{21} = P_6 + P_7$$
$$A_{21}B_{12} + A_{22}B_{22} = P_2 - P_3 + P_5 - P_7$$

Aufwand: $A(n) = 7 \cdot A(\frac{n}{2}) + d \cdot n^2$
Lösung: $A(n) \approx c \cdot n^{\log_2 7} = c \cdot n^{2.81...}$
Bemerkung: $n^{2.81...}$ wächst asymptotisch (d.h. für wachsende $n \to \infty$) langsamer als n^3, allerdings ist die Konstante c ziemlich groß. Daher kommt der Vorteil der Strassenmultiplikation erst bei $n \geq n_0$ zum Tragen, wobei n_0 relativ groß ist.
Als Ausweg bietet sich die *Kombination* der beiden Verfahren an: zuerst wird das rekursive Verfahren von Strassen angewendet, bis n unter den Schwellenwert n_0 fällt. Dann wird mit der Schulmethode fortgefahren. Der Aufwand für dieses Verfahren läßt sich wie folgt ausdrücken:

$$A(n) = \begin{cases} e \cdot n^3 & n < n_0 \\ 7 \cdot A(\frac{n}{2}) + d \cdot n^2 & n \geq n_0 \end{cases}$$

Bei diesem Verfahren erhält man zumindest kleinere Konstanten.

Das Master-Theorem

Das *Master-Theorem* liefert die Lösung für allgemeine Rekursionsgleichungen von „Divide and Conquer“-Algorithmen. Bei diesen Algorithmen wird ein Problem der Größe n in b Teilprobleme der Größe $\frac{n}{c}$ unterteilt. Diese Teilprobleme werden rekursiv gelöst und die Einzellösungen werden zur Gesamtlösung zusammengesetzt. Der Aufwand für ein solches Verfahren läßt sich durch folgende *Rekursionsgleichung* ausdrücken:

$$A(n) = b \cdot A(\frac{n}{c}) \ + \ (Aufwand\ zur\ Zerlegung\ und\ Zusammensetzung)$$

Genauer ausgedrückt:

$$A(n) = \begin{cases} O(1) \\ b \cdot A(\frac{n}{c}) + O(n^k) \end{cases}$$

Das *Master-Theorem* liefert für diese Gleichung folgende Lösung:

$$A(n) = \begin{cases} O(\log n) & b = 1, k = 0 \\ O(n) & b < c, k = 1 \\ O(n \cdot \log n) & b = c, k = 1 \\ O(n^{log_c b}) & b > c, k < \log_c b \end{cases}$$

In unseren Beispielen wäre also einzusetzen:

Fibonacci-Funktion: $b = 1, c = 2, k = 0$

Karatsuba-Multiplikation: $b = 3, c = 2, k = 1$

Strassen-Multiplikation: $b = 7, c = 2, k = 2$

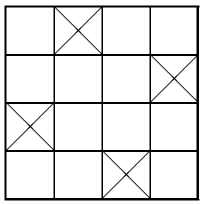
Definition: $f(n)$ heißt $O(g(n))$, falls Zahlen c, n_0 existieren mit $f(n) \leq c \cdot g(n)$ für alle $n \geq n_0$

2.6.8 Backtracking

Backtracking-Verfahren („Trail and Error“-Verfahren) sind Lösungsverfahren, bei denen man versucht, eine Teillösung des Problems systematisch zu einer Gesamtlösung auszubauen. Falls der Ausbau einer vorliegenden Teillösung nicht mehr möglich ist („Sackgasse“), werden einer oder mehrere der letzten Teilschritte rückgängig gemacht. Dann wird versucht, die reduzierte Teillösung auf einem anderen Weg zu einer Gesamtlösung auszubauen. Auch bei diesen Verfahren lassen sich rekursive Prozeduren vorteilhaft einsetzen.
Zur Lösung eines Problems durch Backtracking kann man ein generelles Schema angeben:

```
PROCEDURE backtrack(<Teillösung>);
BEGIN
  IF <vollständige Lösung erreicht> THEN <Gib Lösung an>
  ELSE
    FOR <jede zulässige Teillösung> DO
      backtrack(<Teillösung mit Erweiterung>);
    END;
  END;
END backtrack;
```

Im Hauptprogramm wird die Prozedur dann mit `backtrack(<leere Lösung>)` aufgerufen.
Beispiel: *N-Damen-Problem*
Bei diesem Problem geht es darum, N Damen auf einem $N \times N$ großen Schachbrett so aufzustellen, daß sich keine zwei gegenseitig bedrohen.
Lösung für $N = 4$:



Programm zur Lösung des Problems:

```
MODULE Queens;
    FROM InOut IMPORT WriteCard,WriteLn;
    CONST N = 8;
    TYPE NN = [1..N]
        L = ARRAY NN OF NN;
    VAR a: L; (* a[i]=j bedeutet, daß Dame der i-ten Zeile auf j-te
    Spalte zu setzen ist *)
    PROCEDURE Suche(t: NN);
    (* rekursive Prozedur; gibt alle Lösungen an, die duch Fortsetung
    der Anfangspositionen a[1],...,a[t] möglich sind. Methode:
    backtracking *)

      PROCEDURE Drucke;
      (* gibt eine Lösung aus *)
      VAR i: NN;
      BEGIN
        FOR i := 1 to N DO
          WriteCard(a[i],3);
        END;
        WriteLn;
      END Drucke;

      PROCEDURE Consistent(): BOOLEAN;
      VAR i: NN;
      BEGIN
        FOR i:=1 TO t DO
          IF a[i] = a[t+1] THEN (* gleiche Spalte *)
            RETURN FALSE;
          END;
          IF ABS(INTEGER(a[i] - a[t+1]))=INTEGER(t+1-i) THEN
            RETURN FALSE; (* gleiche Diagonale *)
          END;
        END;
        RETURN TRUE;
      END Consistent;

    VAR i: NN;
    BEGIN (* Suche *)
      IF t = N THEN Drucke (*Ende der Rekursion)
      ELSE
        FOR i := 1 TO N DO
          a[t+1] := i;
          IF Consistent() THEN Suche(t+1) END;
        END;
      END;
```

```
END Suche;

BEGIN (* Hauptprogramm *)
  Suche(0);
END Queens.
```

Wichtige Anwendungen des *Backtracking*-Verfahrens sind:

- Lösungssuche in `PROLOG`
- Lösungen von NP-vollständigen Problemen
- Auswertung von Spielbäumen (Schachprogramme,...)

Dabei ist zu beachten, daß die Rechenzeit exponentiell mit der Suchtiefe wächst. Daher eignen sich Backtracking-Verfahren in der Praxis nur, falls durch Zusatzbedingungen vorab möglichst viele Sackgassen ausgeschlossen werden können.
↑*Branch and Bround-Verfahren*

2.7 Zeiger und dynamische Datenstrukturen

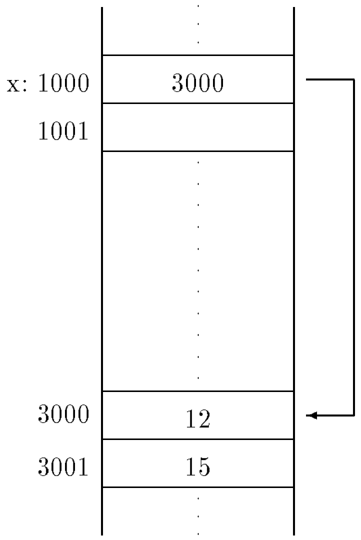
Bisher gilt für Variablen und Datentypen:

- Die Lebensdauer einer Variablen ist gekoppelt an den Aufruf bzw. das Verlassen der Prozedur, in der sie deklariert ist (System des *pulsierenden Stacks*) .
- Daten wie `ARRAY, RECORD` etc. sind *statische Objekte*, d.h. ihre Größe und die Zugriffsmechanismen werden im Deklarationsteil der betreffenden Prozedur festgelegt und können dann nicht mehr verändert werden (z.B. die Zahl der Feldelemente)

2.7.1 `POINTER`-Variablen

Wir können dynamische Datenstrukturen mit Hilfe von `POINTER`-Variablen einführen:

> Eine Zeigervariable (`POINTER`) ist eine Variable, die auf ein anderes Datenobjekt (eines bestimmten Typs) „zeigt", d.h. Werte der `POINTER`-Variablen sind „Verweise" auf andere Objekte.



Die `POINTER`-Variable x ist zum Zeitpunkt ihrer Deklaration eine Speicherzelle mit der Adresse 1000. Im Laufe der Abarbeitung des Programmes erhält die Speicherzelle den Wert 3000. Über die `POINTER`-Variable x kann so ein Datenobjekt, das bei der Speicherposition 3000 beginnt, angesprochen werden (z.B. `ARRAY` mit den ersten Komponenten 12,15,...)

Deklaration von `POINTER`-Variablen:

⟨*Pointer-Variable*⟩ ::= `POINTER TO` ⟨*Datentyp*⟩

Beispiel:

```
TYPE Feld = ARRAY[1..100] OF CARDINAL;
VAR x : POINTER TO Feld;
```

Wichtige Eigenschaften von `POINTER`-Variablen:

- Das Datenobjekt, auf das die `POINTER`-Variable zeigt, hat keinen eigenen Namen, es können also mehrere Quellen angesprochen werden. Zugriff erhält man indirekt über die `POINTER`-Variable.

- **Notation:**
  `x` bedeutet `POINTER`-Variable
  `x^` bezeichnet das Datenobjekt, auf das `x` verweist.

- Variablen und Konstanten lassen sich in *Referenzstufen* einteilen:

Konstanten	heißen Objekte der Referenzstufe 0
(einfache) Variablen	heißen Objekte der Referenzstufe 1
`POINTER`-Variablen	heißen Objekte der Referenzstufe 2
`POINTER`-Variablen, die auf `POINTER`-Variablen der Referenzstufe 2 verweisen	heißen Objekte der Referenzstufe 3
⋮	⋮

Die Referenzstufe gibt an, wie oft auf den Speicher zugegriffen werden muß, bis das „eigentliche“ Datenobjekt erreicht ist. Das Aufsuchen des addressierten Objektes („Nachlaufen des Zeigers“) heißt „*Dereferenzierung*“.

- Durch die Deklaration einer `POINTER`-Variablen wird zunächst nur im pulsierenden Stack ein Speicherplatz für `x` selbst generiert und nicht für das Datenobjekt, auf das `x` weist. Eine Speicheraufforderung dafür erfolgt dynamisch während des Programmablaufs mittels der Prozeduren `NEW` und `ALLOCATE` aus dem `MODULE STORAGE`.
  Beispiel: Sei `x POINTER TO T`. `NEW(x)` bzw. `ALLOCATE(x,SIZE(T))` bewirkt, daß der Speicherbereich, der der Größe des Typs `T` entspricht, reserviert wird und `x` ein Zeigerwert auf diesen Bereich zugewiesen wird.

- Die Lebensdauer des so erzeugten Datenobjektes vom Typ `T` erstreckt sich über den gesamten Ablauf des Programms. Der Speicherplatz kann nur explizit mittels der Prozeduren `DISPOSE` bzw. `DEALLOCATE` wieder freigegeben werden.
  `DISPOSE` bzw. `DEALLOCATE(x,SIZE(T))` meldet dem System, daß der Speicherplatz für das Objekt des Typs `T`, auf das `x` zeigt, wieder verfügbar ist.

- Das Belegen und wieder Freigeben von Speicher mittels `NEW` bzw. `ALLOCATE` und `DISPOSE` bzw. `DEALLOCATE` ist unabhängig von der Prozedur-Information und steht damit *nicht* im Einklang mit dem „pulsierenden Speicher“ und erfordert eine *eigene Speicherverwaltung*, den sogenannten *Heap* (Haufen).
  Stack und Heap stehen in Konkurrenz um den noch freien Speicherplatz:

Stack	⟶ freier Speicher ⟵	Heap

  In dem Moment, in dem sich während des Programmablaufs Stack und Heap überlappen, wird ein Laufzeitfehler „Speicherüberlauf“ ausgelöst.
  Die Verwendung von `POINTER`-Variablen verlangt deshalb große Disziplin, weil der Programmierer selbst über die Speicherbelegung und -freigabe entscheidet.

- **Beispiele für Speicherüberlauf und Speicherverschnitt**

  1. ```
     REPEAT
       ALLOCATE(x,SIZE(T));
     UNTIL FALSE;
     ```
     erzeugt einen *Speicherüberlauf*, da unbegrenzt viel Speicherplatz reserviert wird.

  2. ```
     ALLOCATE(x,SIZE(T));
     ALLOCATE(y,SIZE(T));
     x := y;
     ```
     Zunächst werden zwei Datenobjekte vom Typ `T` erzeugt, auf die `x` bzw. `y` zeigen. Dann wird `x` der (Zeiger-)Wert von `y` zugewiesen. Das Datenobjekt,

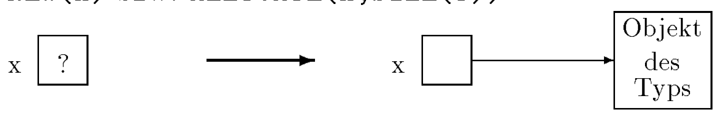
auf das `x` ursprünglich zeigte, kann nun nicht mehr angesprochen werden und steht auch nicht mehr für Aufrufe von `DISPOSE` bzw. `DEALLOCATE` zur Verfügung (=*Speicherverschnitt*).

- Wie alle Variablen sind `POINTER`-Variablen zu Beginn ihrer Lebensdauer bei der Deklaration zunächst undefiniert. Es gibt drei Möglichkeiten der Wertzuweisung:

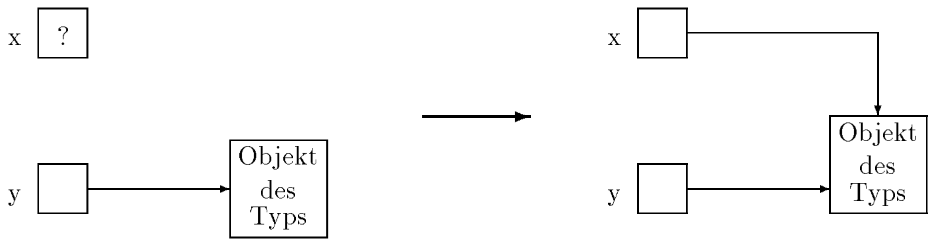
  1. Speicherbelegung mittels `NEW` bzw. `ALLOCATE`
  2. Wertzuweisung eines anderen Zeigers (auf ein Objekt gleichen Typs)
  3. Zuweisung der `POINTER`-Konstanten `NIL` (vordefinierte Konstante, die den „Zeiger auf Nichts“ darstellt)

Veranschaulichung:

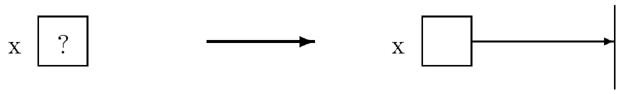
1. `NEW(x)` bzw. `ALLOCATE(x,SIZE(T))`



2. `x := y`



3. `x := NIL`



2.7.2 Dynamische Datenstrukturen

Die Bedeutung der Zeigervariablen liegt in der Möglichkeit der Schaffung *dynamischer Datenstrukturen.*

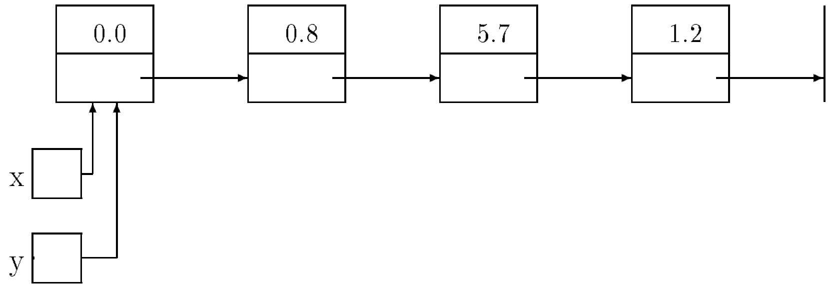
1. **Listen**

```
TYPE Zeiger = POINTER TO Objekt;
     Objekt = RECORD
                inhalt: REAL;
                next  : Zeiger;
              END;
VAR x,y: Zeiger;
```

Wir können damit wie folgt eine Struktur aufbauen:

```
y := NIL;
REPEAT
  ALLOCATE(x,SIZE(Objekt));
  ReadReal(x^.inhalt);
  x^.next := y;
  y := x;
UNTIL x^.inhalt = 0.0;
```

Werden nun z.B. die Zellwerte 1.2, 5.7, 0.8, 0.0 eingelesen, so ergibt sich folgende Situation:



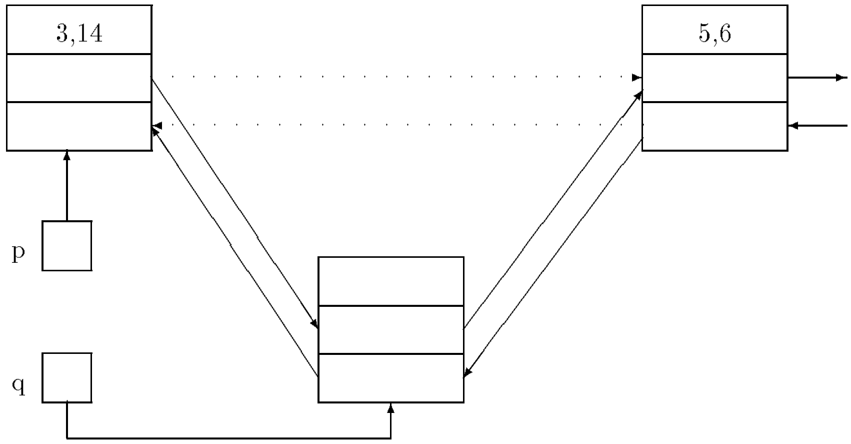
Wir erhalten in dynamischer Weise eine Datenstruktur, deren Zugriffsmechanismen nach Art eines Stacks funktionieren, d.h. nur auf das zuletzt eingelesene Element kann zugegriffen werden.
Um ein Element auszulesen („Pop"-Operation), ist folgende Prozedur nötig:

```
IF x = NIL THEN
  Fehler;
ELSE
  resultat := x^.inhalt;
  y := x;
  x := x^.next;
  DEALLOCATE(y,SIZE(Objekt));
END;
```

2. **Doppelt verkettete Listen**
   Oft werden Listen doppelt „verzeigert" („verkettet"). Eine Zeigervariable weist von vorne nach hinten, die zweite von hinten nach vorne:

```
TYPE Zeiger = POINTER TO Objekt;
     Objekt = RECORD
                inhalt: ...;
                next  : Zeiger;
                prev  : Zeiger;
              END;
```

Wir wollen z.B. ein Datenobjekt, auf das `q` zeigt, zwischen das Datenobjekt, auf das `p` zeigt und seinen rechten Nachbarn einfügen.



Dies erreicht man auf folgende Weise:

```
p^.next^.prev := q;
q^.next := p^.next;
q^.prev := p;
q^.next := q;
```

Der Parameterübergabemechanismus „*Call by Reference*" ist eng mit `POINTERN` verbunden:

```
VAR x: CARDINAL;
PROCEDURE A(VAR z: CARDINAL);   (* Zeile a *)
BEGIN
  z := z+1;                     (* Zeile b *)
END A;

A(x);                           (* Zeile c *)
```

Der `MODULA-2` Compiler erzeugt folgenden Maschinencode:

- Der *Variablenparameter* `z` (Zeile a) wird wie ein *Werteparameter* vom Typ `POINTER TO CARDINAL` aufgefaßt.
- Die Anweisung in Zeile b wird aufgefaßt wie `z^:=z^+1;`
- Der Aufruf von A in Zeile c erfolgt im Sinne einer Werteparameterübergabe:

```
z := Adresse von x;
Unterprogrammsprung nach A;
```

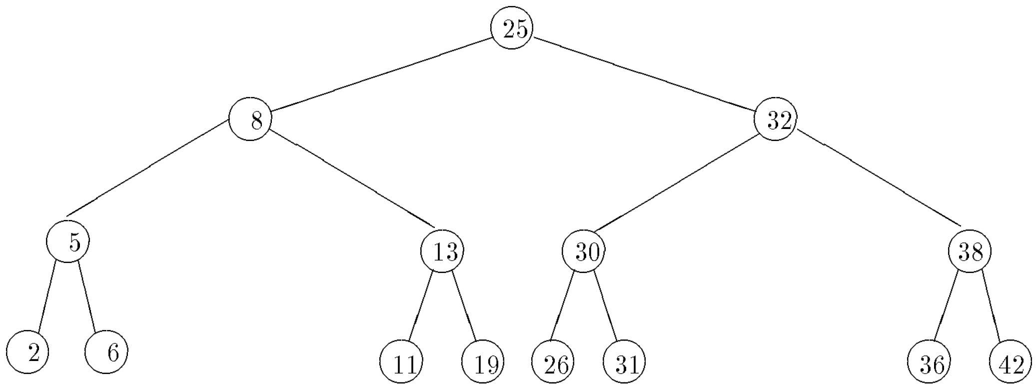
2.7.3 Suchbäume und AVL-Bäume

Wir können mit `POINTERN` Baumstrukturen aufbauen. Dazu betrachten wir folgendes Problem:

Wir wollen `CARDINAL`-Zahlen so einlesen und abspeichern, daß möglichst schnell entschieden werden kann, ob eine gegebene Zahl zu einer eingelesenen Menge gehört oder nicht.
Idee: „Suchbäume“
Man speichert die Zahlen in den Knoten eines *binären Baumes* (ungerichteter Graph ohne Zyklen mit einer Wurzel), so daß für jeden Teilbaum gilt:

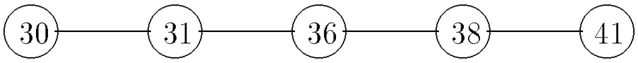
Zahlen im linken Teilbaum < Zahl in der Wurzel < Zahlen im rechten Teilbaum

Die Eingabe der Zahlenfolge 25, 8, 32, 5, 2, 6, 13, 11, 19, 26, 30, 31, 36, 38, 41 liefert folgenden Baum:



Bei der Zahl 35 benötigt man 4 Schritte, um zu sehen, daß 35 in der eingelesenen Menge nicht vorhanden ist. Bei einer *linearen Liste* hätte man dazu 15 Schritte benötigt. Man kann die 35 (34, 33) hinzufügen, allerdings entsteht dann ein langer Ast und der Baum gerät aus dem Gleichgewicht.

Wenn man aus einer vorsortierten Zahlenfolge wie z.B. $41, 38, 36, 31, 30$ einen Baum aufbauen will, so ergibt sich ein zur einfachen Liste entarteter Baum:



Eigenschaften eines vollständigen binären Suchbaums:

- Ein binärer Suchbaum heißt vollständig, falls er bei Baumtiefe t 2^t Knoten hat.
- In jedem vollständigen binären Baum hat jeder Knoten 2 oder 0 Nachfolger (0 Nachfolger: „Blätter“)
- Ein vollständiger binärer Baum der Tiefe t hat $n = 2^0 + 2^1 + 2^2 + \ldots + 2^t = 2^{t+1} - 1$ Knoten.

Aufwandsanalyse: Beim Suchvorgang ist eine Zahl x gegeben. Zu entscheiden ist, ob x im Suchbaum vorkommt.
Wir beginnen mit der Suche in der Wurzel und vergleichen den dort gespeicherten Wert mit x, stellen dabei fest, ob x im linken oder rechten Teilbaum (nach gleichem Prinzip) zu suchen ist. Erreicht man ein Blatt, ohne x gefunden zu haben, so ist x im Baum nicht vorhanden.
Im schlechtesten Fall werden $t + 1 = \log_2 n + O(1)$ viele Suchschritte benötigt.

Mögliche Implementation eines binären Suchbaumes:

```
TYPE Zeiger = POINTER TO Knoten;
     Knoten = RECORD
                inhalt: CARDINAL;
                links : Zeiger;
                rechts: Zeiger;
              END;
VAR baum,p: Zeiger;
```

Das folgende Programmfragment liest `CARDINAL`-Zahlen ein und fügt sie in den Baum ein, bis 0 eingegeben wird.

```
baum := NIL;
LOOP
  ReadCard(x);
  IF x=0 THEN EXIT END;
  ALLOCATE(p,SIZE(Knoten));
  WITH p^ DO
    inhalt := x;
    links  := NIL;
    rechts := NIL;
  END;
  Einfuegen(p,baum);
```

```
END;

PROCEDURE Einfuegen(p: Zeiger; VAR baum: Zeiger);
VAR q: Zeiger;
BEGIN
  IF baum = NIL THEN (* leerer Baum *)
    baum := p;
  ELSE
    q := baum;
    LOOP
      IF p^.inhalt > q^.inhalt THEN
        IF q^.rechts <> NIL THEN
          q := q^.rechts
        ELSE
          q^.rechts := p;
          EXIT;
        END;
      ELSE
        IF q^.links <> NIL THEN
          q := q^.links;
        ELSE
          q^.links := p;
          EXIT;
        END;
      END; (* IF *)
    END; (* LOOP *)
  END; (* IF *)
END Einfuegen;
```

Bemerkung:
Einfügen ist *nicht* rekursiv, Bäume aber sind rekursive Datenstrukturen. Daher ist es sinnvoll, einen rekursiven Algorithmus für das Einfügen zu schreiben:

```
PROCEDURE Einfuegen(p: Zeiger; VAR baum: Zeiger);
BEGIN
  IF baum = NIL THEN
    baum := p;
  ELSE
    IF p^.inhalt > baum^.inhalt THEN
      Einfuegen(p,baum^.rechts); (* in rechten Teilbaum einfuegen *)
    ELSE
      Einfuegen(p,baum^.links);  (* in linken Teilbaum einfuegen *)
    END;
  END;
END Einfuegen;
```

Suchbäume haben bei Eingabe von n Elementen im schlechtesten Fall die Tiefe n, d.h. das Verwalten von n Elementen erfordert im schlechtesten Fall einen Aufwand $O(n)$. Daher sucht man eine Datenstruktur, die auch im schlechtesten Fall n Elemente mit einem Suchaufwand von $O(\log n)$ verwaltet:

AVL-Bäume
Erfinder der *AVL-Bäume* waren Adelson, Velski und Landis. *AVL-Bäume* sind „*höhenbalanciert*", d.h. für jeden Knoten des Baumes gilt:

$$|(\textit{Tiefe des linken Teilbaumes}) - (\textit{Tiefe des rechten Teilbaumes})| \leq 1$$

Wir müssen also beim Einfügen und beim Löschen „Balance" halten, wobei sich die Balance des Knotens K folgendermaßen ergibt:

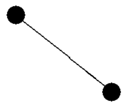
$$b(K) := (\textit{Tiefe des linken Teilbaums von } K) - (\textit{Tiefe des rechten Teilbaumes})$$

Dabei gilt stets: $b(K) \in \{-1, 0, 1\}$
Als Datenstruktur für den AVL-Baum wählen wir:

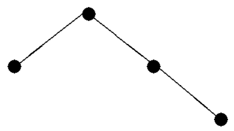
```
TYPE Zeiger: POINTER TO AVL;
     AVL   : RECORD
               inhalt : ...;
               links  : Zeiger;
               rechts : Zeiger;
               balance: [-1,...,1]
             END;
```

Maximal unbalancierte AVL-Bäume:
B_0:

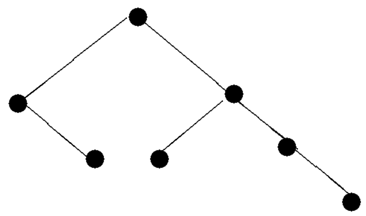
B_1:



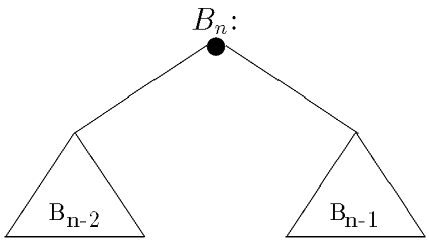
B_2:



B_3:



Allgemein gilt:



Im schlechtesten Fall gilt bei einer Tiefe t für die Zahl der Knoten $k = K(t)$:

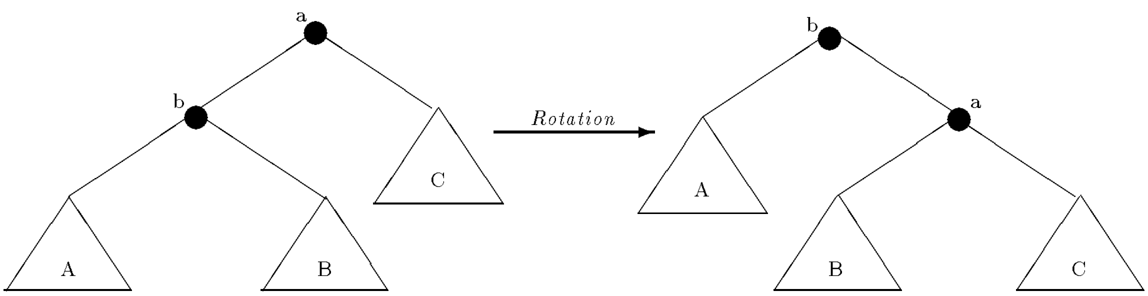
$$K(t) = 1 + K(t-1) + K(t-2) \text{ mit } K(0) = 1 \text{ und } K(1) = 2$$

Es ergibt sich also:

$$K(t) \geq 2^{\frac{t}{2}}, \text{ also } t \leq 2 \cdot \log_2 K$$

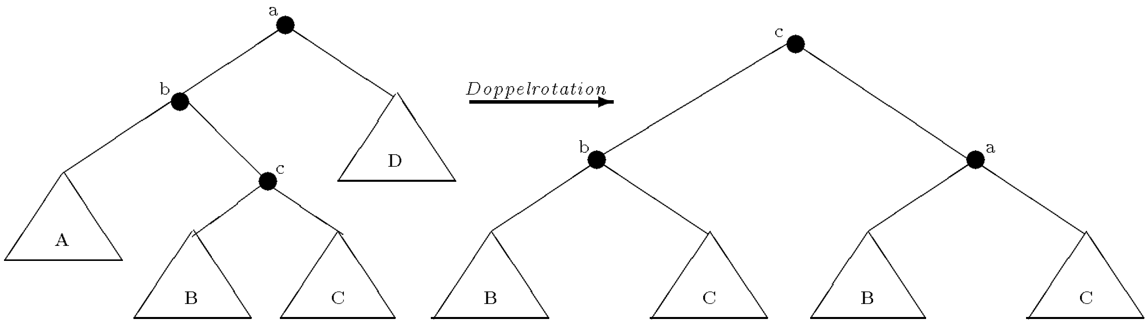
Einfügen eines neuen Elementes in einen AVL-Baum:
Wir fügen Knoten wie im gewöhnlichen Suchbaum als neues Blatt ein. Dabei können sich in allen Knoten auf dem Weg zur Wurzel die Balancewerte ändern. Eventuell entstehen Balancewerte -2 oder 2.
In diesem Fall müssen wir den Baum umstrukturieren, um wieder zulässige Balancewerte in allen Knoten herzustellen.

1. Fall: Die Einordnung verläuft über Knoten b nach a. Dann ist bei a der linke Teilbaum zu tief (d.h. es wurde in Teilbaum A eingefügt).
Die *Rebalancierung* erfolgt mittels *Rotation*:



2. Fall: Die Einordnung erfolgt über die Knoten c, b nach a. Dann ist bei a die Balance nicht mehr gegeben, da der linke Teilbaum zu tief ist. (Es wurde in Teilbaum B oder C eingefügt)

Die Rebalancierung erfolgt mittels *Doppelrotation*:



Da die Umstrukturierung lediglich entlang des Pfades vom neu eingefügten Knoten bis zur Wurzel verlief, ergibt sich eine Komplexität von $O(\log n)$, da AVL-Bäume höchstens die Tiefe $O(logn)$ haben.
Das *Löschen* eines Elementes ist komplizierter, allerdings muß auch hier lediglich entlang eines Pfades umstrukturiert werden, was wiederrum eine Komplextität $O(\log n)$ ergibt.

2.8 Das Modulkonzept und das Prinzip der seperaten Übersetzung

Aus Gründen der Übersichtlichkeit und geringerer Fehleranfälligkeit werden große Programme in mehrere Module zerlegt.
Ein Modul bildet das „*Programmodul*", alle anderen Module sind unterteilt in:

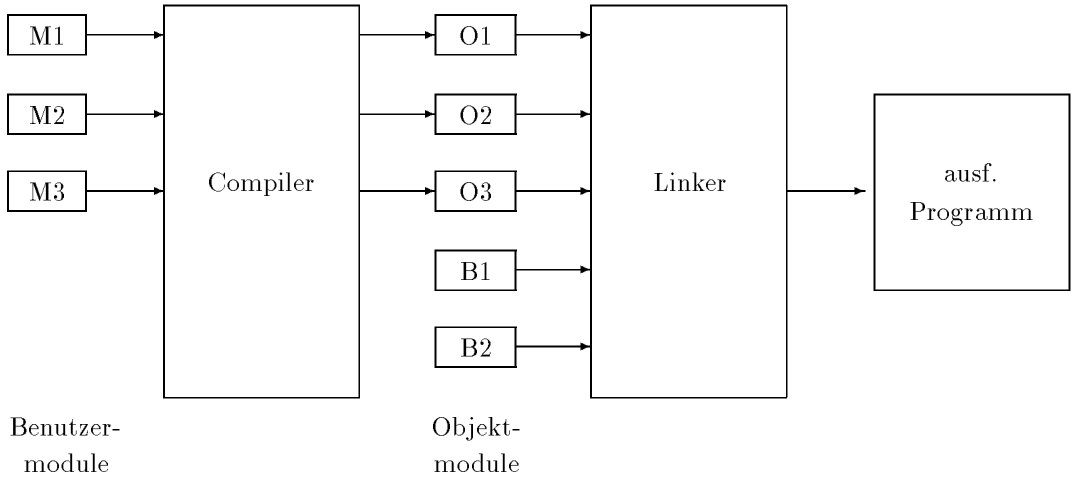
- *Schnittstelle* (`DEFINITION MODULE`)
- *Implementation* (`IMPLEMENTATION MODULE`).

In der *Schnittstelle* wird festgelegt, welche Objekte (Konstanten, Variablen, Typen, Prozeduren) nach außen „sichtbar" sind und eventuell von anderen Modulen importiert werden können.
Im *Implementierungsteil* können Modul-intern durchaus weitere Objekte deklariert sein, die für die Implementierung notwendig sind, aber nicht exportiert werden. („*Geheimnisprinzip*")
Vorteile dieser Lösung:

- Der Übersetzer kann die Schnittstellenüberprüfung unabhängig von der Implementierung vornehmen.
- Die Änderung eines Implementationsteils bringt keine Änderung der Schnittstelle und damit der anderen Module mit sich.
  (Dies wird beim „*rapid–prototyping*" ausgenutzt, wo zunächst mit „irgendwie"-Implementierungen gearbeitet wird)
- Verschiedene Module können von verschiedenen Programmierern erstellt werden.

- Die Bearbeitung ist jeweils auf eine relativ kleine, überschaubare und möglichst in sich geschlossene Einheit (Modul) begrenzt.

- Jedes Modul bildet eine *Übersetzungseinheit*, d.h. Module können separat übersetzt werden. Bei einer Änderung braucht nur das geänderte Modul bzw. die von der Änderung betroffenen Module neu übersetzt zu werden. Daraus ergibt sich eine große Zeitersparnis bei großen Programmen.



- Durch den Übersetzungsvorgang werden *Objektmodule* erzeugt. Diese enthalten das übersetzte Programm in Maschinensprache, allerdings mit einigen symbolischen Zusatzinformationen. Objektdateien sind noch nicht ausführbar, z.B. kann bei Aufruf einer Prozedur in einem anderen Modul zu diesem Zeitpunkt noch keine Absolutadresse für den Anfang der Prozedur im Speicher gegeben werden.

- Der *Binder/Linker* fügt alle Objektdateien zusammen und setzt Absolutadressen ein. So entsteht eine *ausführbare Datei.*

- Der *Lader/Loader* bringt das übersetzte und ausführbare Programm in den Speicher und veranlaßt die Ausführung.
  Oft stellt das Betriebssystem einen *Bindeloader* zur Verfügung, der beide Vorgänge zusammenfaßt.
  Die Implementations-, Definitionsmodule und Objektdateien sind jeweils in seperaten Dateien gespeichert. Die Dateiverwaltung des Betriebssystems notiert sich mit jedem Kreieren bzw. Ändern einer Datei jeweils Datum und Uhrzeit (*Zeitmarke*). Anhand dieser mit jeder Datei angelegten Information kann vom Programm (meist mit dem Namen `MAKE`) automatisch festgestellt werden, welche Module in der aktuellsten Version als Objektdatei vorliegen und welche Module neu übersetzt werden müssen, um für den Bindevorgang nur Objektdateien zur Verfügung zu haben, die alle auf dem neuesten Stand sind. Dazu muß z.B. überprüft werden, ob die Zeitmarke des Implementationsmoduls älter ist als die Zeitmarke der entsprechenden Objektdatei.

Syntax und Anwendungsbeispiel:

- *Syntax der Schnittstelle*

```
DEFINITION MODULE <Name>;
     <Importliste>;
     <Definition der zu exportierenden Objekte
      (bei Prozeduren nur der Prozedurkopf)>
END <Name>.
```

- *Syntax des Implementationsteils*

```
IMPLEMENTATION MODULE <Name>;
     <Importliste>;
     <Deklaration der lokalen Objekte>;
BEGIN
  <Initialisierung der lokalen und zu exportierenden Objekte>;
END <Name>.
```

Zu jedem Definitionsmodul gehört ein Implementationsmodul des selben Namens. Im Implementationsmodul werden die zu exportierenden Prozeduren ausprogrammiert und Variablen werden im Modulkörper – falls notwendig – initialisiert.

Beispiel:

Modul zur Erzeugung von Zufallszahlen nach der multiplikativen Kongruenzmethode von *Kruskal*.

```
DEFINITION MODULE Zufall;
  PROCEDURE RandomCard(a,b: CARDINAL): CARDINAL;
  (* liefert eine Zufallszahl im Intervall a,...,b *)
END Zufall.

IMPLEMENTATION MODULE Zufall;
VAR zz: CARDINAL;

PROCEDURE RandomCard(a,b: CARDINAL): CARDINAL;
BEGIN
  zz := (125*zz) MOD 8192;
  RETURN ((zz DIV 4) MOD (b - a)) + a;
END RandomCard;

BEGIN
  zz := 1;  (* Startwert *)
END Zufall.
```

2.9 Abstrakte Datentypen (ADT)

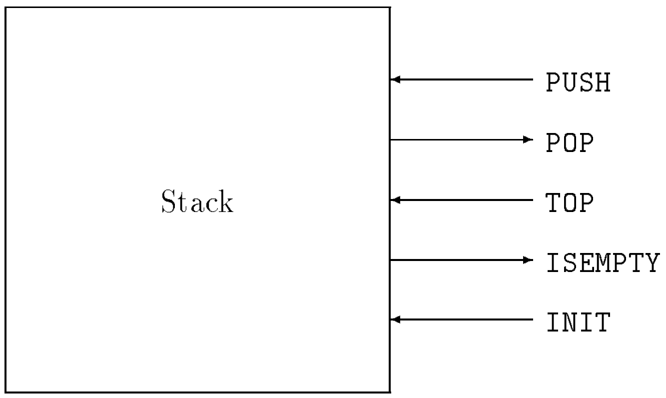
Ein Datentyp ist eine Zusammenfassung von Wertebereich und darauf möglichen Operationen zu einer Einheit.
Stehen die Eigenschaften des Datentyps im Vordergrund und nicht die konkrete Realisierung, so spricht man vom *abstrakten Datentyp (ADT)* im Unterschied zum *konkreten Datentyp*.
Programmiersprachen wie MODULA-2 stellen zunächst eine Reihe von konkreten Datentypen zur Verfügung, z.B.:

```
CARDINAL, REAL, INTEGER
RECORD
ARRAY
SET
```

Der Begriff des Datentyps umfaßt nicht nur die Art der Daten, sondern auch die enthaltenen Zugriffsoperationen, z.B.:

Zugriffsoperationen	(z.B. bei `ARRAY` über Index)
Zusammensetzungsoperationen	(z.B. bei `SET` mittels $+, *$)
Elementbeziehung	(z.B. bei `SET` mit `IN`)

Beispiel: *Stack* (Kellerspeicher)



Wir können einen Stack als abstrakten Datentyp auffassen. Ein Stack beinhaltet:

- (eine nicht näher spezifizierte) innere Darstellung der Daten
- verschiedene Zugriffsoperationen:

`PUSH`	Einfügen von Elementen (am Anfang)
`POP`	Auslesen von Elementen (am Anfang)
`TOP`	„Ansehen“ des Anfangselementes (ohne Herausnahme)
`ISEMPTY`	Test, ob der Stack leer ist
`INIT`	Erzeugung eines neuen Stacks

Wir können das Verhalten eines abstrakten Datentyps vollkommen implementierungsunabhängig durch *Axiome* spezifizieren:

- **Angabe der Definitions- und Wertebereiche:**

`PUSH`	Stack $\times$ Element	$\longrightarrow$	Stack
`POP`	Stack	$\longrightarrow$	Stack
`TOP`	Stack	$\longrightarrow$	Element
`ISEMPTY`	Stack	$\longrightarrow$	`BOOLEAN`
`INIT`	$\emptyset$	$\longrightarrow$	Stack

- **Axiome:**
  1. `ISEMPTY(INIT)`
  2. $\neg$`ISEMPTY(PUSH(x,e))`
  3. `POP(PUSH(x,e)) = x`
  4. `TOP(PUSH(x,e)) = e`
  5. $\neg$`ISEMPTY(x)` $\longrightarrow$ `(x = PUSH(POP(x),TOP(x))`

Mit Hilfe dieser Axiome läßt sich z.B. beweisen, daß gilt:

`TOP(POP(PUSH(PUSH(x,a),b))) = a`

Beweis:
Aus Axiom 3 folgt: `POP(PUSH(PUSH(x,a),b) = PUSH(x,a)`, und damit
`TOP(POP(PUSH(PUSH(x,a),b))) = TOP(PUSH(x,a))`.
Axiom 4 liefert `TOP(PUSH(x,a))=0`

Wir können mit gewissen Abstrichen abstrakte Datentypen in MODULA-2 programmieren (noch besser geeignet sind objektorientierte Sprachen).
Die Implementierung spezifischer Details können vollständig im Implementationsmodul „versteckt“ werden. Über die Schnittstelle (Definitionsmodul) stellt sich die Datenstruktur dar, die sich wie ein abstrakter Stack verhält:

```
DEFINITION MODULE ADTStack;
  TYPE Stack;              (* opaker Typ *)
       Element = CARDINAL; (* Basistyp muss angegeben werden *)
  PROCEDURE PUSH(s: Stack; x: Element): Stack;
  PROCEDURE POP(s: Stack): Stack;
  PROCEDURE TOP(s: Stack): Element;
  PROCEDURE ISEMPTY(s: Stack): BOOLEAN;
  PROCEDURE INIT(): Stack;
END ADTStack.
```

1. **Implementation mittels verketteter Liste:**

```
IMPLEMENTATION MODULE ADTStack;
FROM Storage IMPORT ALLOCATE,DEALLOCATE;
TYPE Stack = POINTER TO Objekt;
     Objekt= RECORD
               Inhalt: Element;
               next  : Stack;
             END;

PROCEDURE PUSH(s: Stack; x: Element): Stack;
VAR t: Stack;
BEGIN
  ALLOCATE(t,SIZE(Objekt));
  t^.Inhalt := x;
  t^.next   := s;
  RETURN t;
END;

PROCEDURE POP(s: Stack): Stack;
VAR t: Stack;
BEGIN
  t := s^.next;
  DEALLOCATE(s,SIZE(Objekt));
  RETURN t;
END POP;

PROCEDURE TOP(s: Stack): Element;
BEGIN
  RETURN s^.Inhalt;
END TOP;

PROCEDURE ISEMPTY(s: Stack): BOOLEAN;
BEGIN
  RETURN (s = NIL);
END ISEMPTY;

PROCEDURE INIT(): Stack;
BEGIN
  RETURN NIL;
END Init;

END ADTStack.
```

2. **Alternative Realisierung mittels** `ARRAY`**s:**

```
IMPLEMENTATION MODULE ADTStack;
FROM Storage IMPORT ALLOCATE;
CONST max=1000;
TYPE Feld  = ARRAY[1..max] OF Element;
     Stack = POINTER TO Objekt;
     Objekt= RECORD
               f: Feld;
               i: CARDINAL;
             END;
PROCEDURE PUSH(s: Stack; x: Element): Stack;
BEGIN
  INC(s^.i);
  s^.f[s^.i] := x;
  RETURN s;
END PUSH;

PROCEDURE POP(s: Stack): Stack;
BEGIN
  DEC(s^.i);
  RETURN s;
END;

PROCEDURE ISEMPTY(s: Stack): BOOLEAN;
BEGIN
  RETURN(s^.i = 0);
END ISEMPTY;

PROCEDURE INIT(): Stack;
VAR s: Stack;
BEGIN
  ALLOCATE(s,SIZE(Objekt));
  s^.i := 0;
  RETURN s;
END INIT;

END ADTStack.
```

Bemerkung:

- Wir haben die Prozeduren als Funktionen realisiert, um mit der mathematischen Axiomatisierung in Einklang zu stehen.
  Normalerweise würde man z.B. `POP` als `PROCEDURE POP(VAR s: Stack)` realisieren.

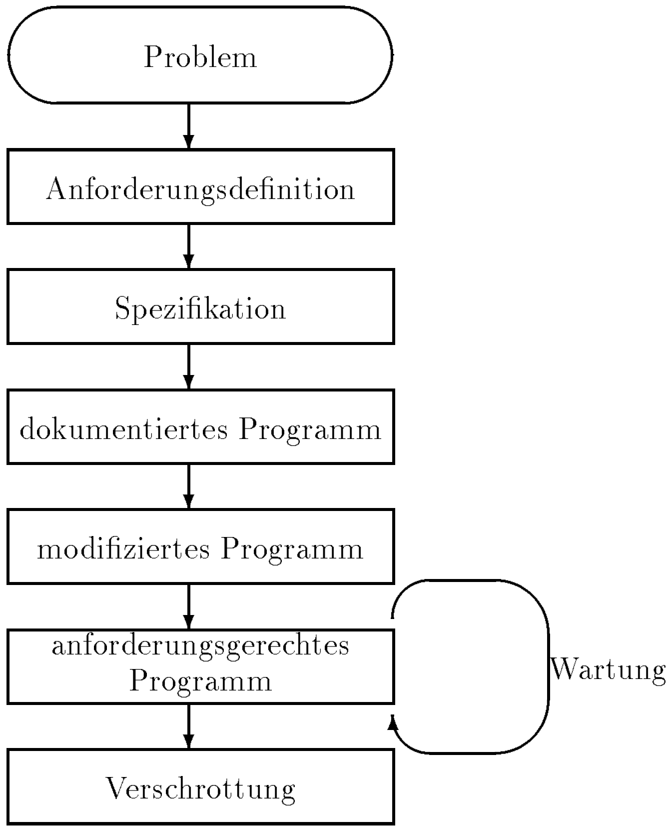
- *opake Typen* dürfen in MODULA-2 im Implementationsteil nur als `POINTER TO` realisiert werden. Demzufolge wird für die Variablen eines opaken Typs Speicherplatz auf dem Heap eingerichtet, weshalb man Initialisierungsroutinen braucht.

2.10 Bemerkungen zum Software-Engineering

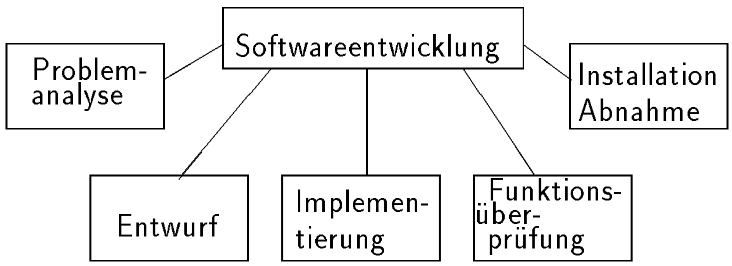
- Software-Engineering: Studium und Anwendung von Prinzipien, Methoden und Techniken zum Entwurf und zur Implementierung von Programmen und Programmsystemen.
  Der Begriff entstand in den 60er Jahren zur Zeit der sogenannten *Softwarekrise*, als man erkannte, daß größere Programme grundsätzlich mit Fehlern behaftet zu sein scheinen, und daß der Aufwand zur Fehlersuche und -beseitigung überproportional zur Programmgröße wächst.
  Ziele des Software-Engineering sind das Erreichen von mehr:

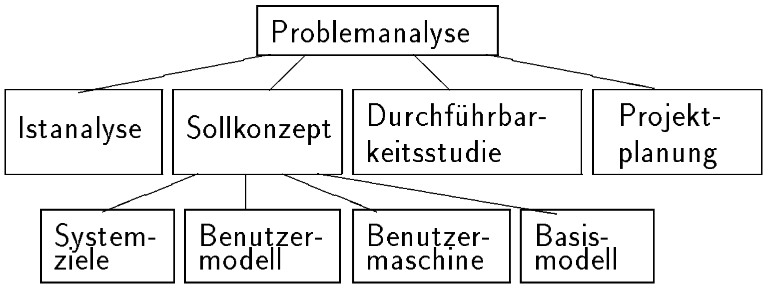
  – *Benutzerfreundlichkeit* (leichte Handhabung)
  – *Zuverlässigkeit* (auch in Extremsituationen)
  – *Wartbarkeit* (auch durch Fremde)
  – *Anpassbarkeit* (an weitere Benutzeranforderungen)
  – *Portabilität* (z.B. durch scharfe Trennung von maschinenabhängigen und -unabhängigen Teilen)
  – *Effizienz* (Verwendung möglichst weniger Ressourcen)
  – *Ergonomie* (dem Menschen angemessene Dialogform).

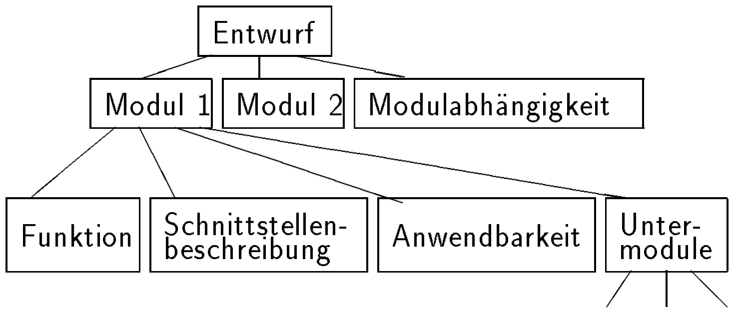
- Grundlage der Untersuchung ist das folgende Phasenmodell:

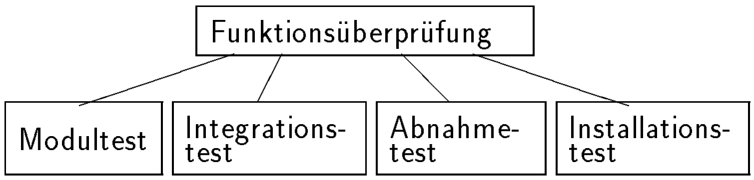


- **Elemente der Softwareentwicklung:**









1. **Problemanalyse**
   Die *Problemanalyse* zergliedert sich in:

   - *Istanalyse*: untersucht und beschreibt das vorliegende System; Bestandsaufnahme;
   - *Sollkonzept*: Ziele werden formuliert; Entwicklung eines Benutzermodells; Maschinenanforderungen und Benutzerschnittstellen festlegen;
   - *Durchführbarkeitsstudie*: Kosten-/Nutzenanalyse;
   - *Projektplanung*: Erstellung von Zeitplänen, Verteilung der Personalressourcen.

   Angaben zur Problemanalyse (insbesondere das Sollkonzept) werden in einem „Pflichtenheft" als Teil des Vertragsdokumentes zwischen Auftraggeber und Softwarehersteller festgehalten.

2. **Entwurf**
   Der Programmierer entwickelt Modelle des Gesamtsystems, das die in der Problemanalyse formulierten Anforderungen erfüllt. Der „Abstand" zwischen Benutzermaschine und vorhandenem bzw. zu beschaffenden Computersystem (Basismaschine) muß überbrückt werden.
   Dabei können die Prinzipien der Modularisierung ausgenutzt werden: Zerlegung der Aufgabe in einzelne Module, Hierarchisierung, abstrakte Datentypen, Datenkapselung,...
   Grundsätzliche Entwurfsprinzipien:

   - *top-down-Methode*: „schrittweise Verfeinerung";
   - *bottom-up-Methode*: Erstellung immer mächtigerer „Tools" aus bereits vorhandenen.

   Beide Methoden werden in der Praxis simultan verwendet.

Allgemeine Entwurfsprinzipien:

- jedes Modul verwende möglichst wenige andere Module;
- die Zahl der Annahmen über andere Module (Länge der Schnittstellenbeschreibung) soll möglichst klein bleiben;
- die interne Struktur eines Moduls soll anderen verborgen bleiben (Geheimnisprinzip), Seiteneffekte sollen vermieden werden (Black-Box-Prinzip).

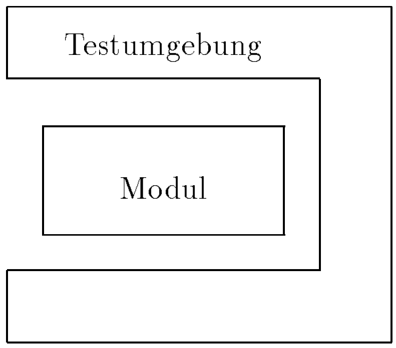
Die Ergebnisse der Entwurfsaktivitäten sind in der Spezifikation festgehalten, in der für jedes Modul seine Funktion und Schnittstelle, sowie die Voraussetzungen für seine Anwendbarkeit festgehalten werden.
Dafür existieren *formalisierte Spezifikationssprachen*.

3. **Implementation**
Allgemeine Prinzipien bei der Implementation sind:

- strukturierte und dokumentierte Programmdarstellung;
- verständliche Wahl von Namen für Variablen, Prozeduren, etc.;
- Kommentare einfügen;
- Verwendung von Standardprogrammelementen (keine „Tricks").

4. **Funktionsüberprüfung(Testen)**
Als Funktionsüberprüfung bezeichnet man die Überprüfung des Ein- und Ausgabeverhaltens des Programms anhand der Spezifikation.
Durch *Testen* kann man höchstens feststellen, ob Fehler anwesend sind, nicht aber, ob sie abwesend sind. Der Nachweis der Abwesenheit von Fehlern setzt die Anwendung formaler Methoden der Programmverifikation voraus, die meist zu aufwendig sind- ($\uparrow$*Informatik II*)

- *Modultest*: Testen der Module in einer eigens dafür geschaffenen Testumgebung:



Betriebssysteme stellen Testhilfen zur Verfügung (`trace, dump,...`).

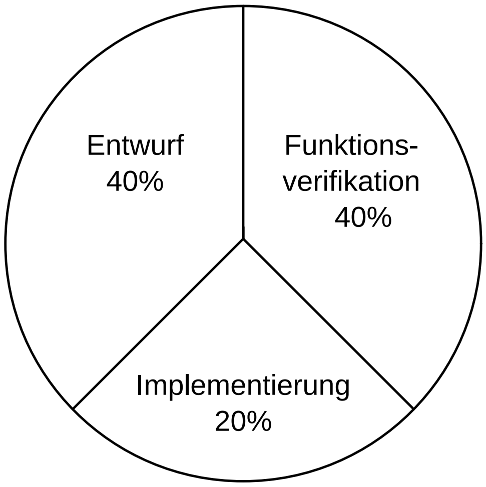
- *Integrationstest*: Funktionsüberprüfung des Gesamtsystems anhand von Anforderungsspezifikationen;

- *Installationstest*: Test auf Einsatz-Hardware (erforderlich, falls diese von der Entwicklungs-Hardware verschieden ist);
- *Abnahmetest*: Der Auftraggeber testet das Programm.

5. **Leistungsüberprüfung**

6. **Installation, Abnahme**

Allgemeine Beobachtung:



Dokumentation:
Der Verlauf und die Ergebnisse aller Phasen müssen *projektbegleitend* dokumentiert werden:

- Benutzerdokumentation (Gebrauchsanweisung);
- Entwicklungsdokumentation;
- technische Dokumentation.

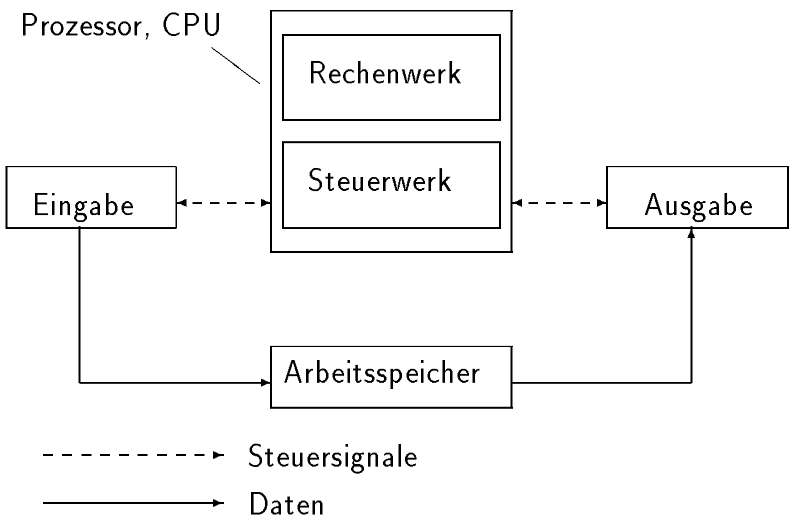
Bemerkung:
Das besprochene Phasenmodell ist ein wenig überholt. Das „*rapid–prototyping*" ist stark im Kommen:
Fast alle Phasen der Softwareentwicklung werden in einem Durchgang relativ schnell durchlaufen, um zu „Prototypen" zu kommen. Anhand der Erfahrungen mit diesen Prototypen werden die einzelenen Arbeitsvorgänge noch einmal überdacht und verfeinert.

3 Der Aufbau von Computern

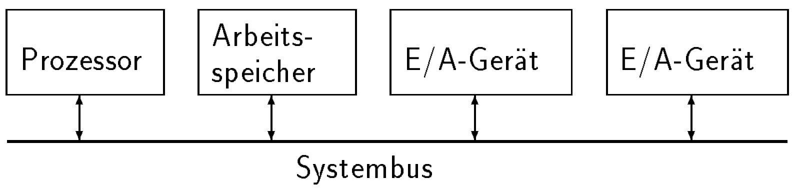
Nach der summarischen Vorstellung des Aufbaus von Computern im Abschnitt 1.2 soll nun detaillierter beschrieben werden, wie Computer "eigentlich funktionieren".

3.1 Der prinzipielle Aufbau von Computern

Es wurde bereits der prinzipielle Aufbau von Computern besprochen:



Rechen- und Steuerwerk bilden den *Prozessor* (CPU – *C*entral *P*rocessing *U*nit). Die Ein- und Ausgabegeräte, sowie die externen Speicher (Diskettenlaufwerke, Magnetplattenspeicher, Magnetbandspeicher, ...) heißen *periphäre Geräte*.
Die Verbindung der einzelnen Bestandteile wird über den Systembus organisiert:



Zunächst benötigt ein Computer ein Programm in Maschinensprache (bestehend aus Anweisungen wie LAOD, STORE, ADD, ...), um arbeiten zu können. Verschiedene Befehle der Machinensprache haben gewisse Ähnlichkeit miteinander. Sie können mit Hilfe einer kleineren Anzahl von grundlegenden Operationen, den *Mikroanweisungen* (micro instructions) ausgedrückt werden. Aus ökonomischen Gründen sind Computer so aufgebaut, daß sie eher Mikroanweisungen ausführen als Programme in Maschinensprache. *Interpreter* (geschrieben in Form von Mikroanweisungen) sagen dem Computer, wie jeder Befehl der Maschinensprache zu lesen, zu verstehen und auszuführen ist ("mikroprogrammierte Computer"). (Hat im allgemeinen nichts mit Mikrocomputern zu tun.)

Die *Software* unfaßt alle Programme, die zum Computer gehören. Unter *Firmware* versteht man Programme, die unter Verwendung von Mikroanweisungen geschrieben sind, sog. "Mikrogramme". (z.B. Interpeter). Die *Hardware* meint alle physikalischen Einrichtungen, aus denen ein Computer besteht. Sie besteht fast vollständig aus elektronischen Bauelementen. Prinzipiell könnten aber auch andere Bauteile verwendet werden, z.B. mechanische Bausteine (Babbage) oder pneumatische Systeme.

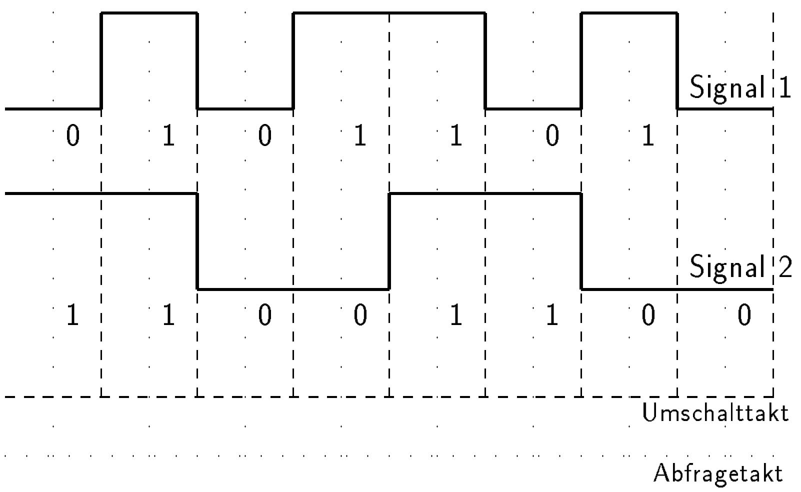
3.2 Schaltelemente

3.2.1 Binäre Signale und Taktsteuerung

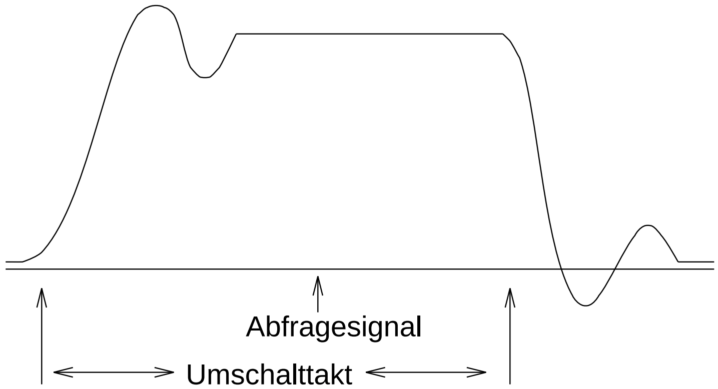
Alle Daten, die zwischen Teilen eines Computers transportiert werden, sind als binäre elektrische Signale codiert. Typische Spannungswerte sind:

- 0 Volt (Masse, niedrig, low)
- 5 Volt (hoch, high)

Der zeitliche Spannungsverlauf auf einer Leitung wechselt immer Sprunghaft zwischen diesen Werten, die Steuerung erfolgt durch Taktsignale:



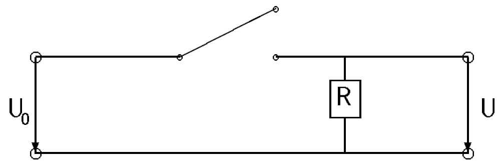
Aus physikalischen Gründen ist kein echter Rechteckverlauf realisierbar, sondern es findet ein sog. Einschwingen statt. Aus diesem Grund ist der *Abfragetakt* gegen den *Umschalttakt* verschoben.

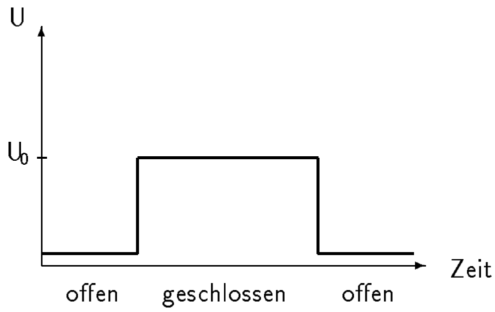


Es kann aber vom genauen elektrischen Verhalten abstrahiert werden und von der "diskreten" Anschauung ausgegangen werden. Das zeitliche Verhalten $x(t)$ wird dabei nicht als kontinuierliche Funktion angesehen, sondern als diskrete Funktion mit den Werten $x_0, x_1, x_2, \ldots$ (Diskrete Mathematik ist von großer Bedeutung für die Informatik). Der "übliche" Abstand der Taktsignale liegt in der Größenordnung von 40 nsek, das entspricht 25 Millionen Taktsignalen pro Sekunde, also einer Taktfrequenz von 25 MHz.

3.2.2 Elektronische Schalter: Transistoren

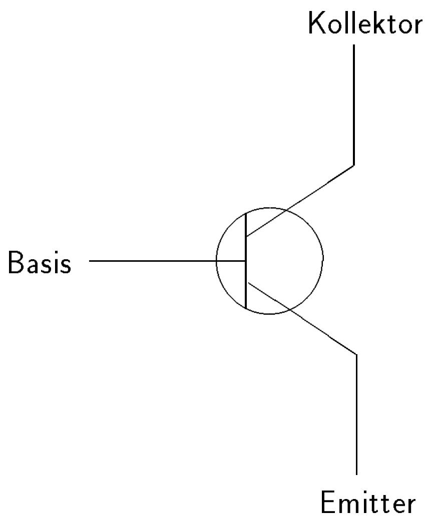
Zur Erzeugung rechteckiger Spannungsverläufe werden Schalter benötigt. Ist der Schalter offen, so fließt kein Strom und die Spannung am Widerstand R ist 0. Ist der Schalter geschlossen, so liegt am Widerstand R die Spannung U_0 an.



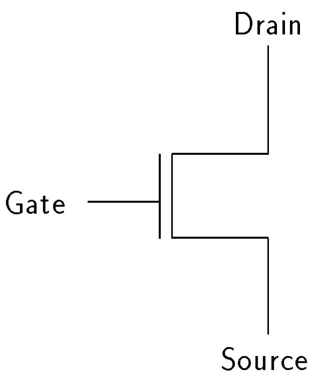


Elektronische Schalter werden durch Halbleiter-Transistoren realisiert. Man unterscheidet dabei folgende Typen:

1. Bipolarer Transistor

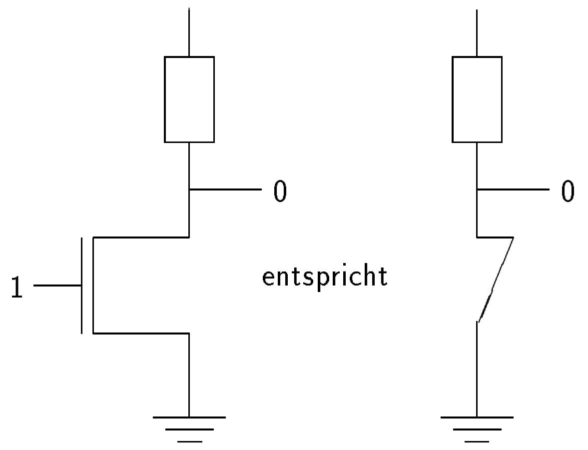


2. MOS-Transistor (metal-oxide-semiconductor)

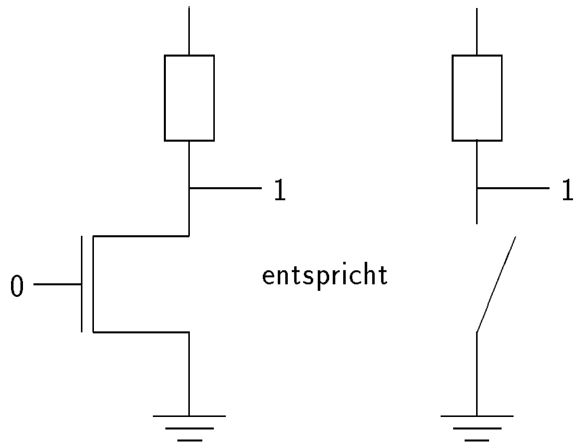


Die Strecke Source-Drain stellt einen Widerstand dar, dessen Größe durch die Spannung am Gate verändert werden kann. Ist die Spannung Gate-Source gleich 0, dann ist der Widerstand (fast) unendlich groß. Ist die Gate-Source-Spannung gleich der Source-Spannung, dann ist der Widerstand annähernd null. Das heißt: Ein Transistor ist ein Schalter, der über Gate geöffnet und geschlossen werden kann.

Ein Transistor wird durch eine positive Gate–Spannung in Durchlaßrichtung gepolt, d.h. der Schalter wird geschlossen:



Ein Transistor wird durch die Gate–Spannung null in Sperrichtung gepolt, d.h. der Schalter wird geöffnet:



Bei der physikalischen Realisierung von Transistoren wird die Eigenschaft einiger Materialien (z.B. Silizium) ausgenutzt, die bei wohldosierten Verunreinigungen den elektrischen Strom nur in eine Richtung leiten. Die Technik der integrierten Schaltungen vereinigt Transistoren, Widerstände und Verbindungsleitungen auf einem Chip.

3.3 Schaltnetze und Schaltwerke

Wir verknüpfen jetzt einzelne Schaltelemente zu größeren Schaltungen. Es wird dabei zwischen Schaltungen ohne Speicherverhalten (sogen. Schaltnetze) und solchen mit Speicherverhalten (sogen. Schaltwerke) unterschieden.

3.3.1 Schaltnetze oder kombinatorische Schaltkreise

Mathematisch gesehen realisieren Schaltnetze 0-1-wertige Funktionen (Boolesche Funktionen, logische Funktionen, Schaltfunktionen):

$$f : \{0,1\}^n \to \{0,1\}^m$$

Solche Funktionen können aus einigen wenigen Grundfunktionen aufgebaut werden:

- Negation (NOT):

$$\neg : \{0,1\} \to \{0,1\};\ \neg(x) = \overline{x}$$

x	$\overline{x}$
0	1
1	0

- Konjunktion (AND):

$$AND : \{0,1\}^2 \to \{0,1\};\ AND(x,y) = x \wedge y$$

x	y	$x \wedge y$
0	0	0
0	1	0
1	0	0
1	1	1

- Disjunktion (OR):

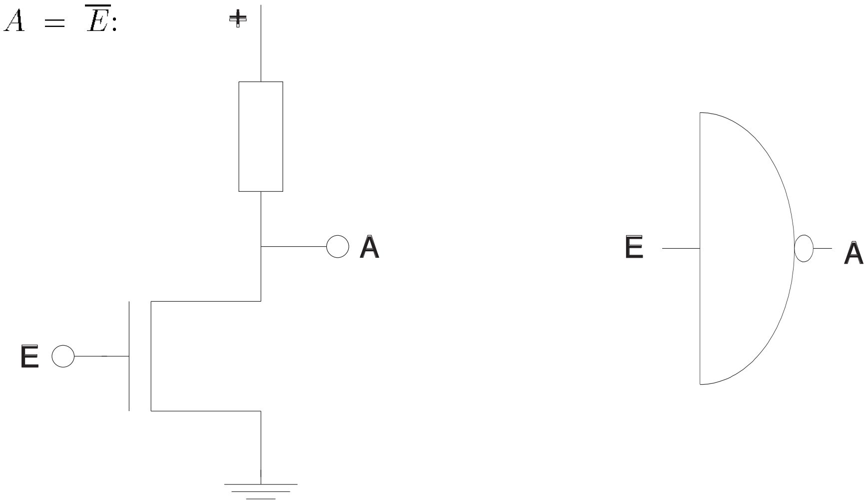
$$OR : \{0,1\}^2 \to \{0,1\};\ OR(x,y) = x \vee y$$

x	y	$x \vee y$
0	0	0
0	1	1
1	0	1
1	1	1

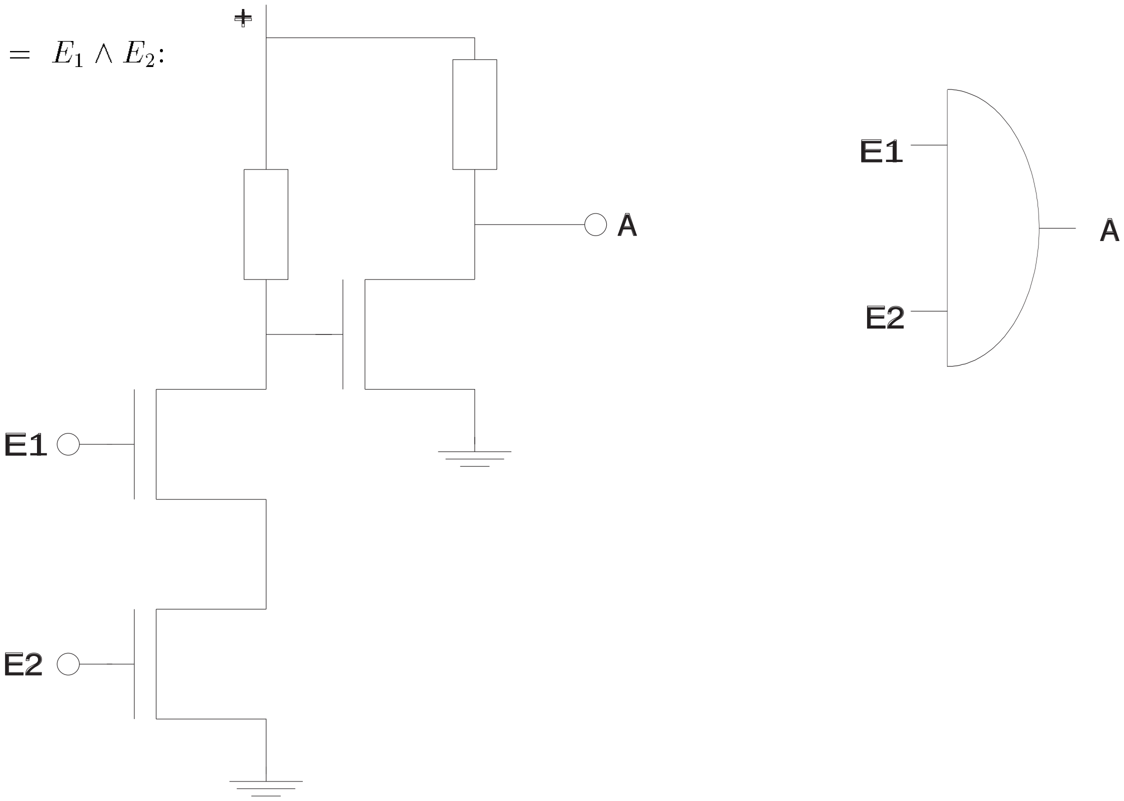
Diese Funktionen können durch folgende einfache Verknüpfungsschaltungen (sogen. Gatter, Gates) physikalisch realisiert werden:

Symbolische Darstellung:

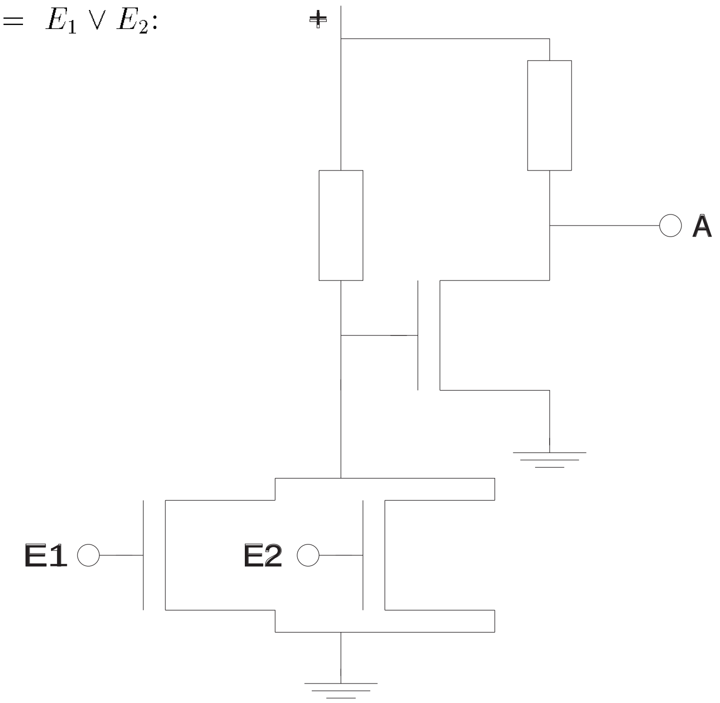
$A = \overline{E}$:

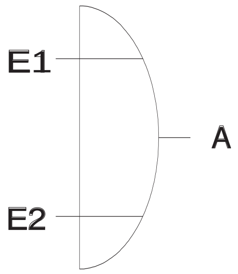


$A = E_1 \wedge E_2$:



$A = E_1 \vee E_2$:





Schaltnetze sind Zusammenschaltungen von Gattern ohne Rückkoppelung. (logische Schaltungen)

Beispiel: Wir betrachten folgende logische Aussage:

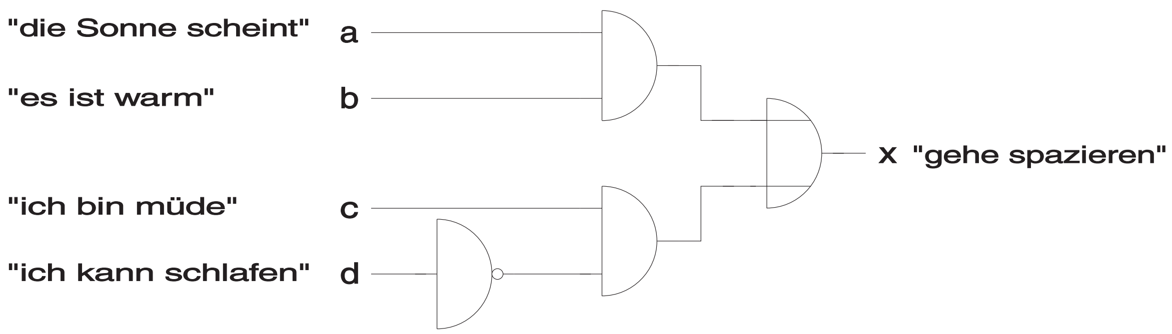
“Wenn $\underbrace{\text{die Sonne scheint}}_{a}$ *und* es $\underbrace{\text{warm ist}}_{b}$

oder $\underbrace{\text{wenn ich müde bin}}_{c}$ *und nicht* $\underbrace{\text{schlafen kann}}_{d}$

gehe ich spazieren.”

Die Gesamtaussage kann wahr (symbolisiert durch 1) oder falsch (symbolisiert durch 0) sein. Der Wahrheitswert kann durch Schaltnetze aus den Wahrheitswerten der Einzelaussagen berechnet werden:

$$x = (a \wedge b) \vee (c \wedge \overline{d})$$

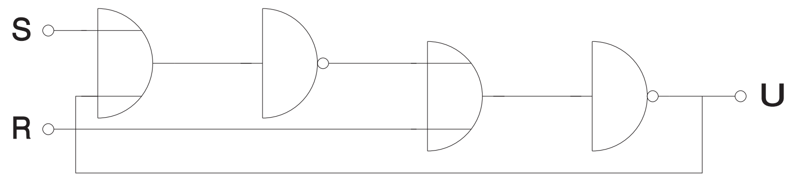


Die Ausgangssignale hängen lediglich vom momentanen Wert der Eingangssignale ab.

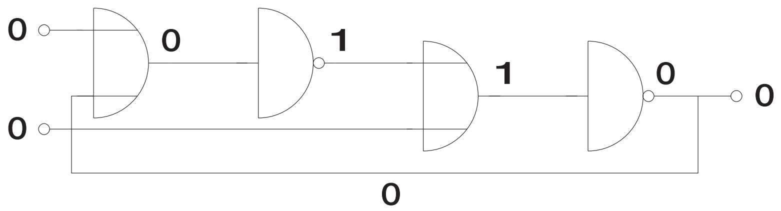
3.3.2 Schaltwerke oder sequentielle Schaltkreise

Neben Schaltnetzen werden auch Schaltungen benötigt, die Informationen (z.B. 1 Bit) Speichern können, d.h. die Ausgabe soll nicht nur von der momentanen Eingabe, sofern auch vom "Zustand" der Schaltung abhängen. Das mathematische Modell einer solchen Schaltung ist das des "endlichen Automaten". (↑Vorlesung Automatentheorie)

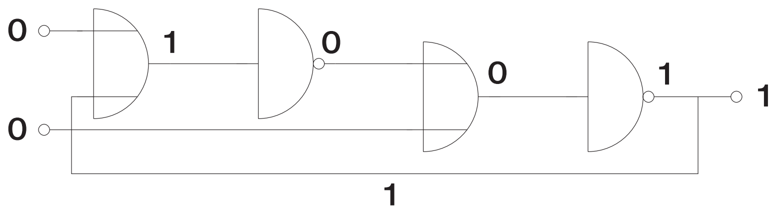
Das einfachste Speicherelement kann 1 Bit, d.h. zwei Zustände 0 oder 1, speichern. Dafür werden hauptsächlich sog. "Flip-Flops" verwendet. Ein *ungetaktetes Flip-Flop* ist eine rückgekoppelte Gatterschaltung aus zwei Oder-Nicht-Verknüpfungen:



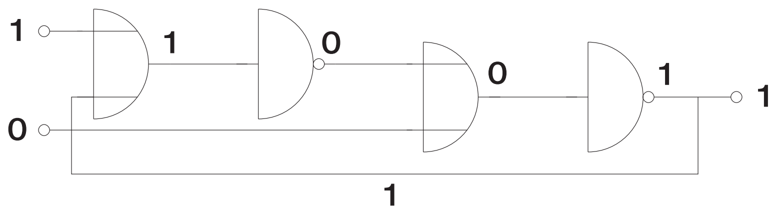
Es gilt:



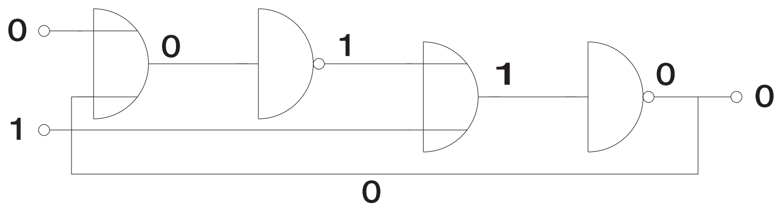
Stabiler Zustand $U = 0$.



Stabiler Zustand $U = 1$.

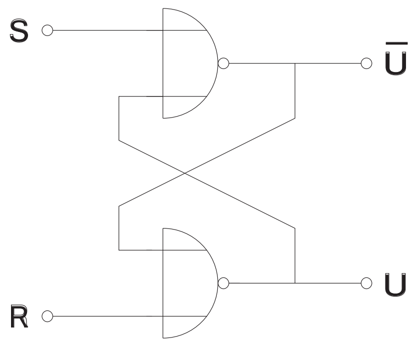


Setzen des Schalters (Versetzen der Schaltung in Zustand 1)

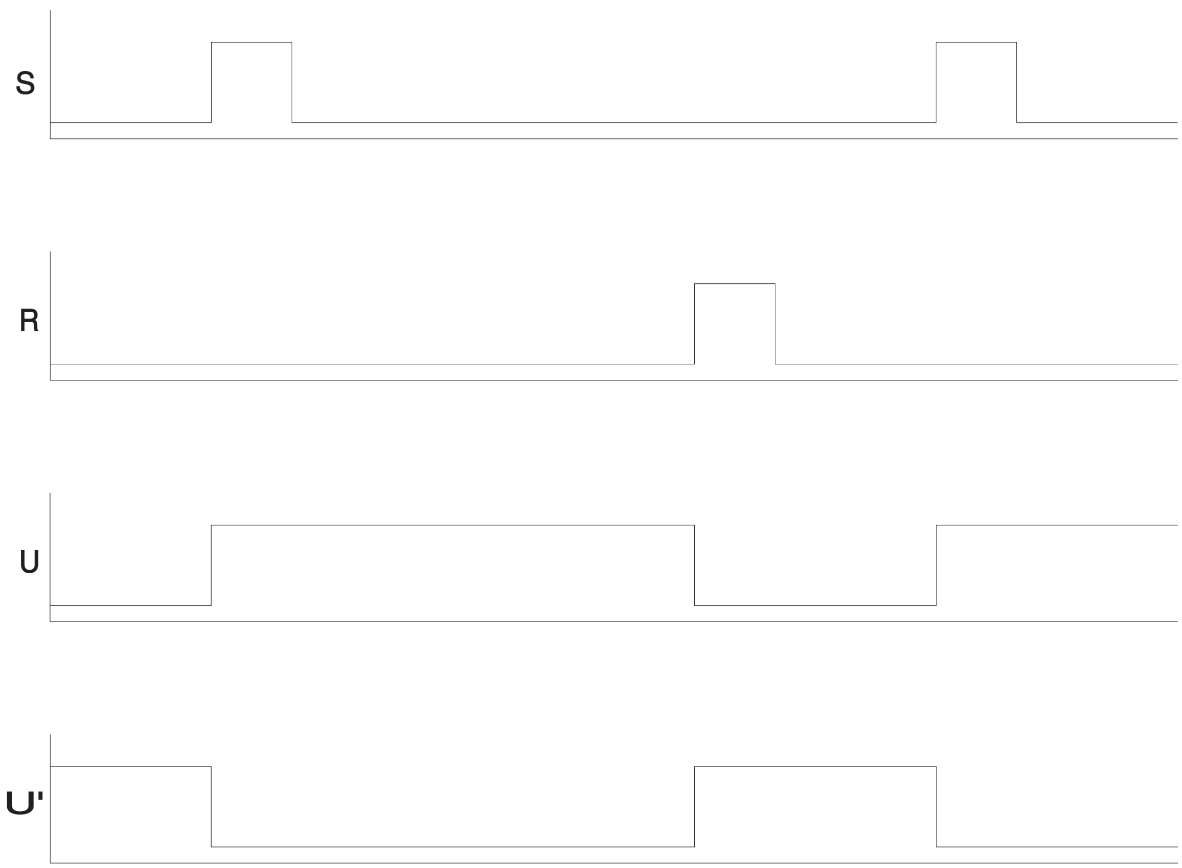


Rücksetzen des Schalters (Versetzen der Schaltung in Zustand 0)

Der symmetrische Aufbau eines Flip-Flops sieht folgendermaßen aus:



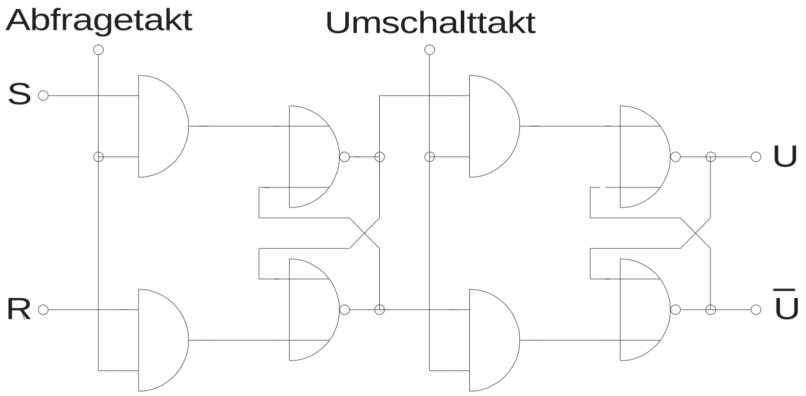
S bezeichnet den Setzeingang.
R bezeichnet den Rücksetzeingang.



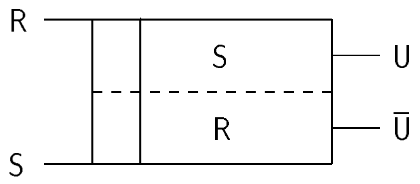
Eine Funktionstabelle für ein Flip-Flop sieht folgendermaßen aus:

S	R	U_{neu}
0	0	U_{alt}
0	1	0
1	0	1
1	1	verboten

Um sicherzustellen, daß sich die Ausgangssignale einschwingen und stabilisieren können, wird in der Praxis mit *getakteten Flip-Flops* gearbeitet. Diese bestehen aus zwei ungetakteten Flip-Flops, von denen eins durch den Abfragetakt und das andere durch den Umschalttakt gesteuert wird:



Achtung: Die Eingangssignale wirken nicht sofort, sondern erst einen Takt später. Deshalb werden getaktete Flip-Flops auch als Verzögerungsglieder bezeichnet. Symbolisch werden getaktete Flip-Flops folgendermaßen dargestellt:



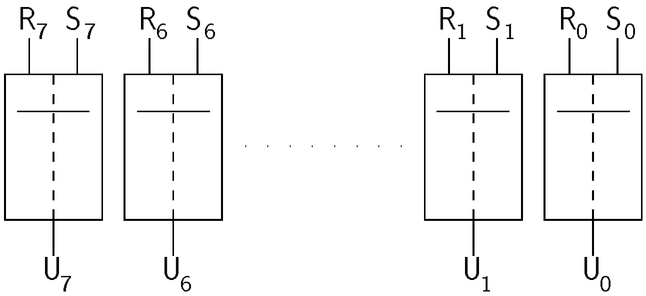
Schaltwerke sind also Schaltungen, die neben Gattern auch Speicherelemente wie z.B. Flip-Flops enthalten. Sie werden auch als *sequentielle Schaltungen* bezeichnet.

3.4 Das Rechenwerk

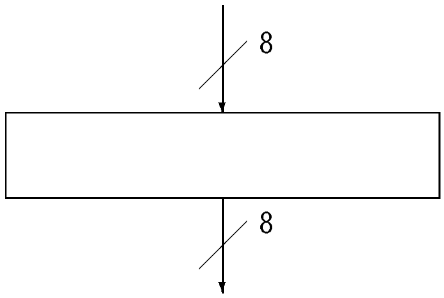
Aus Gattern und Flip-Flops sind alle wichtigen Schaltungen aufgebaut, aus denen ein Computer bestehen.

3.4.1 Register

Ein Register ist ein Speicher für eine Anzahl logisch zusammenhängender Bits. Die Länge eines Register entspricht meist der Länge eines Maschinenwortes. Register bestehen aus einer Aneinanderreihung von getakteten Flip-Flops:



Die symbolische Darstellung eines Registers sieht folgendermaßen aus:



3.4.2 Halbaddierer

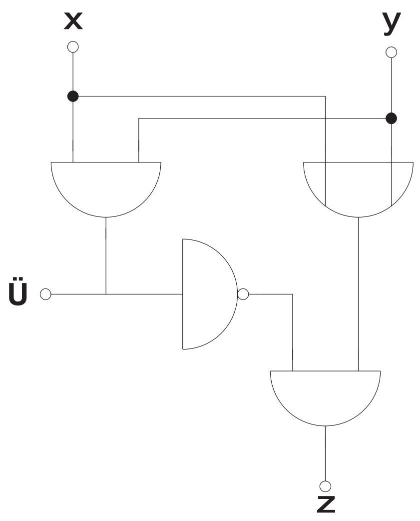
Ein Halbaddierer ist eine Komponente zur Addition einstelliger Dualzahlen, dessen Funktionstabelle so aussieht:

		Ergebnisbit	Übertrag
x	y	z	$ü$
0	0	0	0
0	1	1	0
1	0	1	0
1	1	0	1

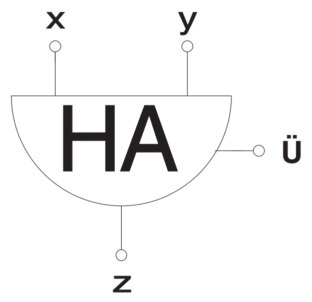
Für das Ergebnisbit z gilt: $z \; = \; \overline{(x \wedge y)} \wedge (x \vee y)$

Für den Übertrag $ü$ gilt: $ü \; = \; (x \wedge y)$

Dies läßt sich als Schaltnetz folgendermaßen realisieren:



Symbolische Darstellung:

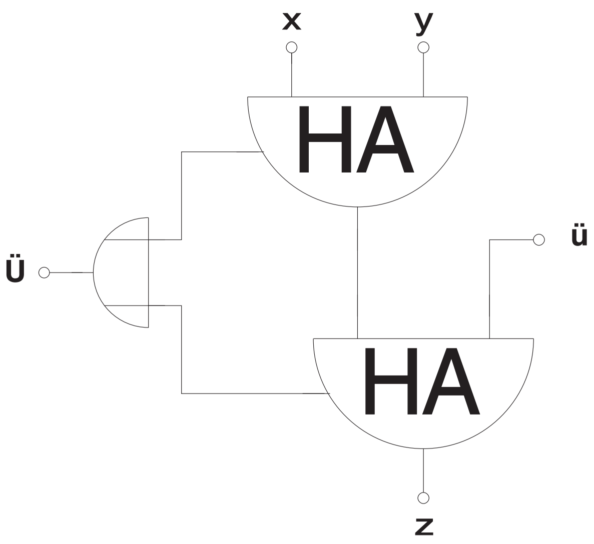


3.4.3 Volladdierer

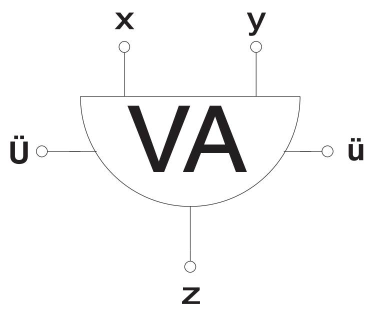
Ein Volladdierer dient zur Addition beliebiger mehrstelliger dualer Zahlen. Er muß neben den Eingabebits auch den Übertrag berücksichtigen:

x	y	$ü$	z	$Ü$
0	0	0	0	0
0	0	1	1	0
0	1	0	1	0
0	1	1	0	1
1	0	0	1	0
1	0	1	0	1
1	1	0	0	1
1	1	1	1	1

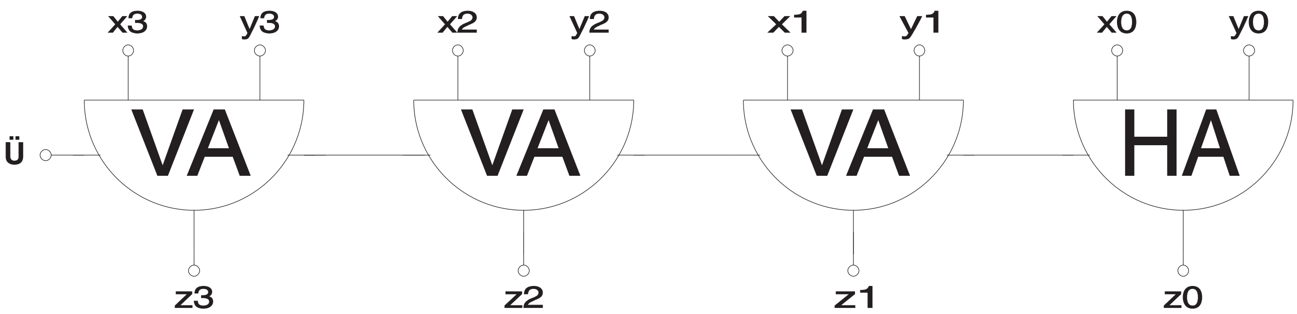
Ein Volladdierer läßt sich durch die Hintereinanderschaltung zweier Halbaddierer realisieren, deren Überträge durch ein Oder-Gatter verknüpft werden:



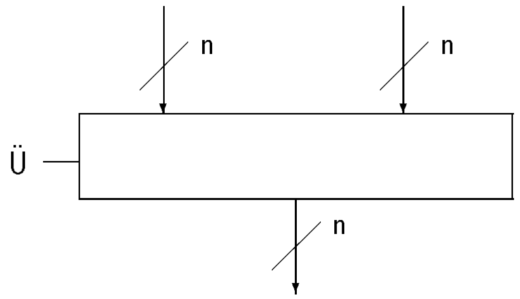
Symbolische Darstellung:



Mit Volladdierern können n–stellige Dualzahlen addiert werden:

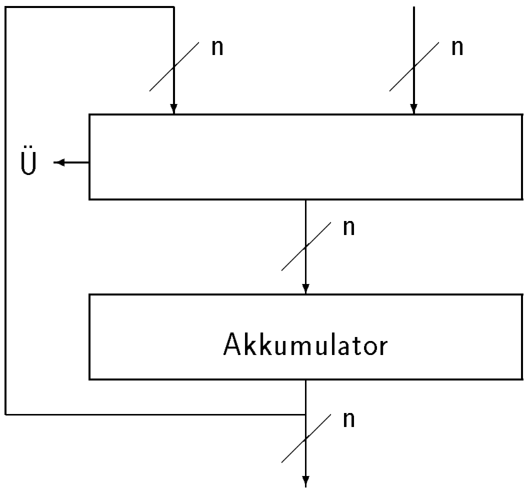


Symbolische Darstellung:



3.4.4 Akkumulator

Der Akkumulator ist ein Register innerhalb einer Additionsschaltung für n-stellige Dualzahlen:



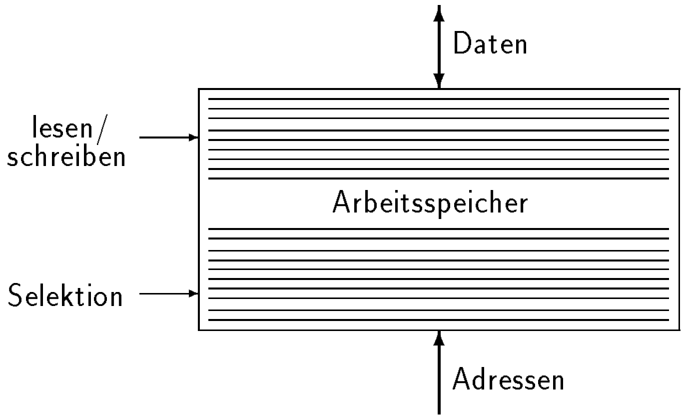
Zur Addition n-stelliger Dualzahlen sind folgende Schritte notwendig:

1. Lösche AC ($\Rightarrow AC = 0$)
2. Addiere (aus Speicherzelle geholtes) x_1 zum Inhalt von AC ($\Rightarrow AC = x_1$)
3. Addiere x_2 zum Inhalt von AC ($\Rightarrow AC = x_1 + x_2$)

Der Akkumulator sammelt also die Ergebnisse fortlaufender Additionen. Diese Additionsschaltung ist der Kern jedes Rechenwerks, da alle Operationen (mit ganzen Zahlen) auf eine Folge von Additionen zurückgeführt werden kann.

3.4.5 Der Arbeitsspeicher

Der Arbeitsspeicher ist eine lineare Anordnung von Speicherzellen:



Die Daten werden über n Datenleitungen zwischen Arbeitsspeicher und Prozessor übertragen, bei einer Wortlänge von 32 Bit sind z.B. 32 Leitungen notwendig. Es wird zwischen "lesen" und "schreiben" unterschieden. Die Selektion bestimmt den Zeitpunkt des Lese- oder Schreibvorgangs. Neben den Datenleitungen sind noch m Adressleitungen zur Adressierung notwendig, so können zum Beispiel mit $m = 24$ Leitungen $2^{24} = 16$ MByte adressiert werden.

3.5 Das Steuerwerk und die Programmausführung

Wir werden Steuerwerk und Programmausführung anhand des Beispielprogramms "Euklidischer Algorithmus" (vgl. S. 6) vereinfacht darstellen. Zur Erinnerung noch einmal der Algorithmus und das Programm zur Berechnung des größten gemeinsamen Teilers von p und q:

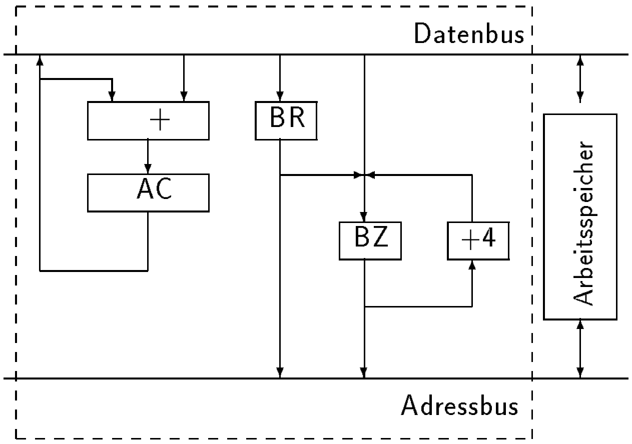
Schritt 1: Man dividiere p durch q und erhält einen Rest r.

Schritt 2: Ist $r = 0$, so ist q der gesuchte ggT.
Ist $r \neq 0$, dann benenne q in p und r in q um, wiederhole Schritt 1 und 2.

Programm:

Adresse	Befehle		Kommentar
			Speicherplatzreservierung:
00	–		Speicherplatz für p
04	–		Speicherplatz für q
08	–		Speicherplatz für r
			Schritt 1
12	LOAD	00	Lade AC mit p
16	MODULO	04	Berechne den Rest von AC/q in AC
20	STORE	08	Speichere AC in r
			Schritt 2
24	IFZERO	48	Wenn $AC = 0$ dann gehe nach 48
28	LOAD	04	Lade AC mit q
32	STORE	00	Speichere AC in p
36	LOAD	08	Lade AC mit r
40	STORE	04	Speichere AC in q
44	JUMP	12	Springe nach 12
48	END		Halte an, q enthält ggT

Das Programm steht in den Registern 12 bis 51, jedes der Register ist 4 Byte lang. Die Daten (p, q und r) werden vor dem Programm gespeichert.
Aus den bisher eingeführten elektronischen Bauelementen ist jeder *Mikroprozessor* aufgebaut:



BR ist das *Befehlsregister* und dient zur Zwischenspeicherung des gerade ausgeführten Befehls. BZ ist der *Befehlszähler* und enthält die Adresse des nächsten auszuführenden Befehls. (Befehlszählregister)

Die Arbeitsweise des Mikroprozessors:

Das Befehlsregister bekommt über die Datenleitung einen Befehl. Der Operationsteil des Befehls (links im Register) bestimmt die Operation, die als nächstes ausgeführt werden soll und veranlaßt dessen Ausführung. Der Adreßteil des Befehls (rechts im Register) geht auf den Adressbus und damit an den Arbeitsspeicher zum Laden oder Speichern von Operanden. Zur Ausführung von Sprungbefehlen (JUMP, IFZERO, ...) benutzt man die Verbindung zum Befehlszähler. Der Befehlszähler wird normalerweise in jedem Durchgang durch die Addition der Befehlslänge (im Beispiel 4 Byte) erhöht. Er liefert seinen Inhalt an den Adressbus und damit an den Arbeitsspeicher zum Lesen des nächsten Befehls.

Befehlszyklus:

Eine Programmausführung ist — vom Steuerwerk gesehen — ein zyklischer Vorgang:

Anfangszustand: Der Befehlszähler enthält die Adresse des ersten Befehls.

Schritt 1: Bringe den Befehl, dessen Adresse im Befehlszähler steht, aus dem Speicher in das Befehlsregister und erhöhe zugleich den Inhalt des Befehlszählers um die Länge des Befehls.

Schritt 2: Führe den im Befehlsregister stehenden Befehl aus und gehe wieder zu Schritt 1.

Index