Algorithmen und Programmierung

(Studienjahr 2005/2006)

Doz.Dr.habil.Werner Vogt

Technische Universität Ilmenau
Fakultät für Mathematik und Naturwissenschaften
98684 Ilmenau, Germany

e-mail: werner.vogt@tu-ilmenau.de

Teil 1/3

Literaturempfehlung (Auswahl)

Allgemeine Grundlagen

1. **Rechenberg,P.; Pomberger,G.(Hrsg.):**
   Informatik-Handbuch. 3. erw. Auflage, 1189 Seiten. Hanser Verlag 2002.
2. **Gumm, H.-P.; Sommer, M.:**
   Einführung in die Informatik. 4. Aufl., Oldenbourg-Verlag, München 2000.
3. **Levi, P.; Rembold, U.:**
   Einführung in die Informatik für Naturwissenschaftler und Ingenieure. 4. Aufl., Hanser Verlag, München 2003.

Algorithmen

1. **Heun, V.:**
   Grundlegende Algorithmen. Einführung in den Entwurf und die Analyse effizienter Algorithmen. 2. Aufl., Vieweg-Verlag, Wiesbaden 2003.
2. **Sedgewick, R.:**
   Algorithmen. 2. Aufl. (Nachdruck), Addison-Wesley, München 2003.
3. **Wirth, N.:**
   Algorithmen und Datenstrukturen. 5. Aufl., Teubner, Stuttgart 2000.
4. **Roller, D.:**
   Informatik-Grundlagen. Springer-Lehrbuch, 1994.

Programmieren in PASCAL

1. **Cooper, D.; Clancy, M.:**
   PASCAL – Lehrbuch für das strukturierte Programmieren. 6. Aufl., Vieweg-Verlag Braunschweig 2003.
2. **Van Canneyt, M.; Klämpfl, F.:**
   Free Pascal. Programmieren für Linux, DOS, Windows und OS/2. Computer und Literaturverlag, Nürnberg 2000
3. **Van der Heijden, J.J. u.a.:**
   The GNU Pascal Manual. Free Software Foundation, Inc. 1996-2001.
4. **Raymans, H.-G.; Schlabach, T.:**
   Windows-Programmierung mit Borland-Pascal 7.0. BHV-Verlag 1993.
5. **Starke, M. (Hrsg.):**
   Borland Pascal 7.0 - Das Buch. Te-wi Verlag, 1993.

Programmieren in MAPLE

1. **Heck, A.:**
   Introduction to Maple. 3rd Edition. Springer, New York 2003.
2. **Walz, A.:**
   Maple 7 : Rechnen und Programmieren. 2. vollst. überarb. Aufl., Oldenbourg Verlag, München 2002.
3. **Blachman, N.; Mossinghoff, M.J.:**
   Maple griffbereit. Vieweg Verlag, Braunschweig 1996.

4. **Krawietz, A.:**
Maple V für das Ingenieurstudium. Springer-Verlag, Berlin 1997.
5. **Heal, K.M. u.a.:** *Handbuch, Waterloo Maple*
Maple V – Learning Guide. Springer, New York 1998.
6. **Monagan, M.B. u.a.:** *Handbuch, Waterloo Maple*
Maple V – Programming Guide. Springer, New York 1998.

Programmieren in C++

1. **Stroustrup, B.:**
Die C++ - Programmiersprache. 4. aktualisierte u. erw. Aufl., 1068 Seiten. Addison-Wesley, München 2002.
2. **Erlenkötter, H.:**
C++ : Objektorientiertes Programmieren von Anfang an. 2. Aufl., Rowohlt Taschenbuch Verlag, Hamburg 2000.
3. **Wolff von Gudenberg, J.:**
Objektorientiert programmieren von Anfang an. 2. Aufl., Spektrum Hochschul-Taschenbuch, Mannheim, 1996.
4. **Davis, S.R.:**
C++ für Dummies - gegen den täglichen Frust mit C++. 2. Aufl., IWT Thomson Publishing, 1997.
5. **Claussen, U.:**
Objektorientiertes Programmieren mit Beispielen und Übungen in C++. Springer-Verlag, Berlin 1998.
6. **Flowers, B.H.:**
An Introduction to Numerical Methods in C++. Clarendon Press, Oxford 1995.

Numerische Verfahren und Wissenschaftliches Rechnen

1. **Hoffmann, A.; Marx, B.; Vogt, W.:**
Mathematik für Ingenieure. Band 1. Pearson Studium, München 2005.
2. **Hanke-Bourgeois, M.:**
Grundlagen der Numerischen Mathematik und des Wissenschaftlichen Rechnens. B.G. Teubner-Verlag, Stuttgart 2002.
3. **Quarteroni, A.; Sacco, R.; Saleri, F.:**
Numerische Mathematik. Band 1 und Band 2, Springer-Verlag, Berlin 2002.
4. **Schwarz, H.R.:**
Numerische Mathematik. 4.erw.Auflage, Teubner-Verlag, Stuttgart 1997.
5. **Hermann, M.:**
Numerische Mathematik. Oldenbourg Verlag, München 2001.
6. **Stoer, J.; Bulirsch, R.:**
Einführung in die Numerische Mathematik. Band 1 und 2 , 7. Aufl., Springer–Verlag Berlin 1994 .
7. **Roos, H.-G.; Schwetlick, H.:**
Numerische Mathematik - das Grundwissen für jedermann. B.G. Teubner, Stuttgart 1999.
8. **Plato, R.:**
Numerische Mathematik kompakt. Grundlagenwissen für Studium und Praxis. Vieweg–Verlag, Braunschweig 2000.

Inhaltsverzeichnis

Kapitel 1

Algorithmusbegriff und algorithmische Sprachen

1.1 Algorithmen und ihre Darstellung

Wesentliche Voraussetzungen für eine erfolgreiche Problemlösung:
– eine exakte Aufgabenstellung
– eine Algorithmierung des Lösungsweges
– eine fehlerfreie Formulierung des Lösungsverfahrens als Programm

1.1.1 Anschaulicher Algorithmusbegriff

Beispiel 1 Quadratische Gleichung $x^2 + ax + b = 0$.

1. Vorbetrachtung:
$$x_{1,2} = (-\frac{a}{2}) \pm \sqrt{(-\frac{a}{2})^2 - b}$$
Falls $D = (-\frac{a}{2})^2 - b < 0$, so sind x_1 und x_2 konjugiert komplex mit $x_{1,2} = Re \pm Im * i$.

2. Algorithmierung:
   – Gegebene Größen (Eingabegrößen)
   – Gesuchte Größen (Ausgabegrößen, Lösungen)
   – Umformungsregeln (Regelsystem)

Algorithmus in Schrittform:

S1 Gib a und b ein!

S2 Setze $c = -a/2$

S3 Setze $D = c * c - b$

S4 Setze $w = \sqrt{|D|}$

S5 Falls $D < 0$ ist, so gehe zu **S9**, sonst zu **S6**

S6 Berechne $x_1 = c + w\ ,\ x_2 = c - w$

S7 Gib x_1 und x_2 und "Lösungen reell!" aus

S8 Gehe zu **S11**

S9 Setze $Re = c\ ,\ Im = w$

S10 Gib Re, Im und "Lösungen komplex!" aus

S11 STOP

Definition 1 : Intuitiver Algorithmusbegriff

Sei P eine Problemklasse und $\mathcal{A}$ die Menge der Daten, die P beschreiben. Man bezeichnet mit

$\mathcal{A}$ – Menge der Eingabegrößen (Problemdaten, Eingabeinformationen)
$\mathcal{B}$ – Menge der Ausgabegrößen (Lösungsmenge, Ausgabeinformationen).

Ein Algorithmus $A_P : \mathcal{A} \rightarrow \mathcal{B}$

- ist ein determiniertes, schrittweises Verfahren,
- das für jedes $a \in \mathcal{A}$ nach endlich vielen Schritten anhält und
- eine Lösung $b \in \mathcal{B}$ liefert.

Eine Funktion $f : \mathcal{A} \rightarrow \mathcal{B}$ heißt berechenbar, Falls ein Algorithmus A_P mit $A_P(w) = f(w)\ \forall w \in \mathcal{A}$ existiert.

Darstellungsformen für Algorithmen:

– Schrittalgorithmus
– Programmablaufplan (PAP)
– Struktogramm
– Computerprogramm

Schrittalgorithmus

Der Schrittalgorithmus ist eine 1-dimensionale Form des Algorithmus mit verbaler rechnerunabhängiger Darstellung der Algorithmusschritte.

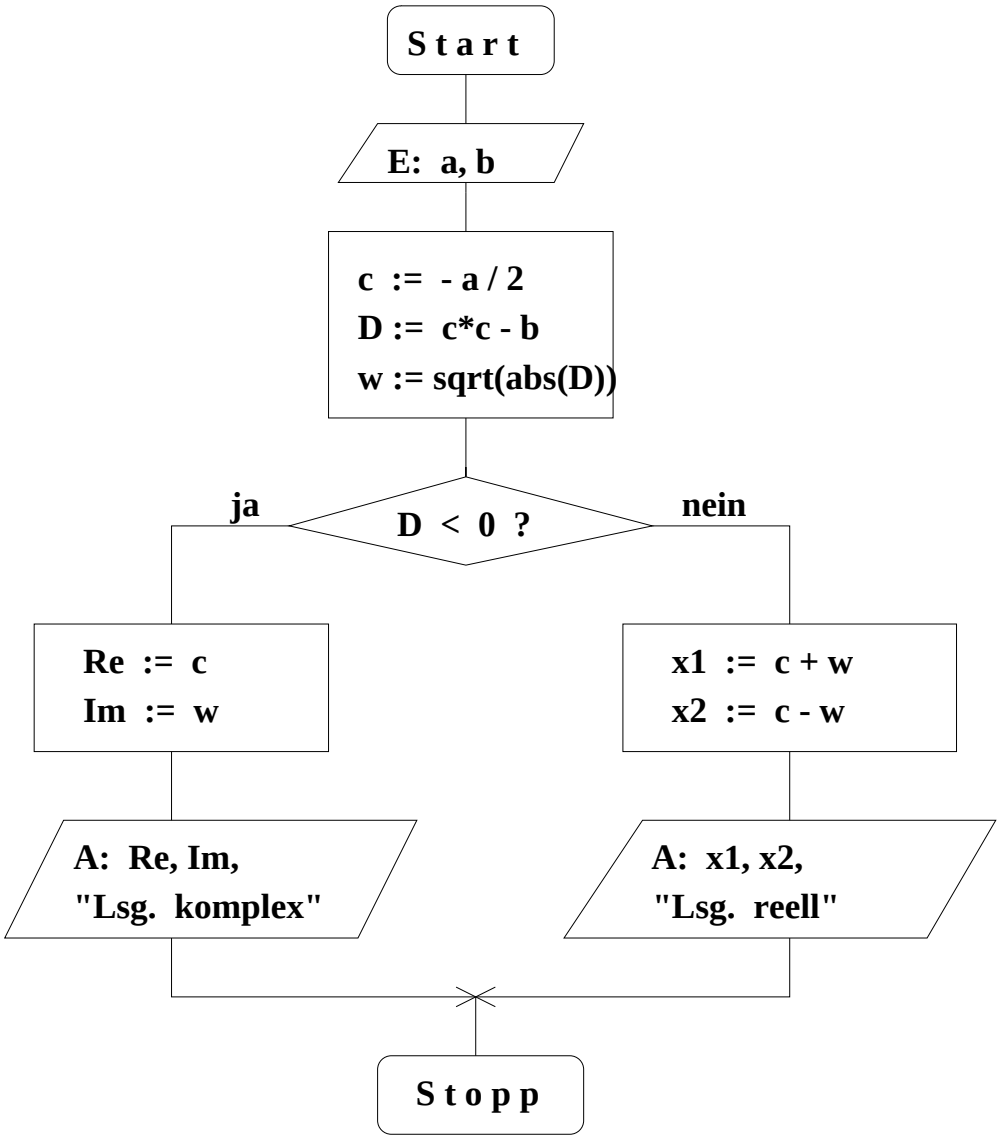
Beispiel 1 Quadratische Gleichung $x^2 + ax + b = 0$.

S1 Gib a und b ein!
S2 Setze $c = -a/2$
S3 Setze $D = c * c - b$
S4 Setze $w = \sqrt{|D|}$
S5 Falls $D < 0$ ist, so gehe zu **S9**, sonst zu **S6**
S6 Berechne $x_1 = c + w$, $x_2 = c - w$
S7 Gib x_1 und x_2 und "Lösungen reell!" aus
S8 Gehe zu **S11**
S9 Setze $Re = c$, $Im = w$
S10 Gib Re, Im und "Lösungen komplex!" aus
S11 STOP

Programmablaufplan (PAP)

Der Programmablaufplan ist eine weitgehend rechnerunabhängige 2-dimensionale grafische Algorithmusdarstellung, bei der die Schritte durch Kästen definiert und durch Linien miteinander verbunden werden.

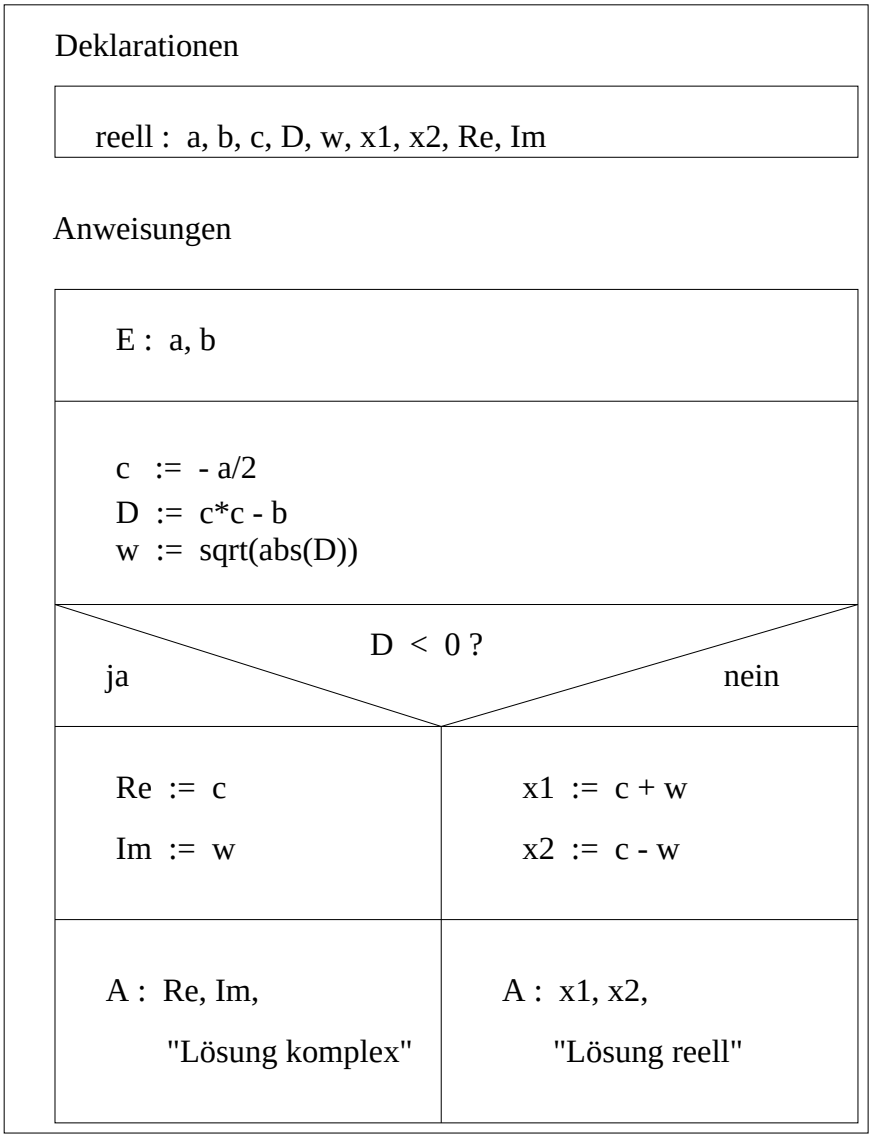
Beispiel 1 Quadratische Gleichung $x^2 + ax + b = 0$.



Struktogramm

Das Struktogramm ist eine weitgehend rechnerunabhängige 2-dimensionale grafische Algorithmusdarstellung innerhalb eines fest vorgegebenen Rahmens. In diesen Rahmen können beliebig komplizierte Strukturblöcke "kantendeckend" eingefügt werden.

Beispiel 1 Quadratische Gleichung $x^2 + ax + b = 0$.



(Computer)-Programm

- Ein Programm ist eine geordnete oder strukturierte Menge von Anweisungen zur Abarbeitung eines Algorithmus.
- Ein Computerprogramm ist die Formulierung eines Algorithmus in maschinell abarbeitbarer Form.

Beispiel 1 Quadratische Gleichung $\boldsymbol{x^2 + ax + b = 0}$ **.**

```
Program Quadratische_Gleichung;
var
  a, b, c, D, w, x1, x2, re, im : real;
begin
  readln (a, b);
  c := -a / 2.0;
  D := c * c - b;
  w := sqrt (abs (D));
  if D < 0 then
  begin
    re := c; im := w;
    writeln(re, im, 'Loesungen komplex')
  end
  else
  begin
    x1 := c + w; x2 := c - w;
    writeln (x1, x2, 'Loesungen reell')
  end
end.
```

1.1.2 Exakter Algorithmusbegriff

Frage: Existiert eine (technische) Aufgabe, die ein Computer jetzt und zukünftig nicht erfüllen kann – d.h irgendeine Aufgabe, für die kein Algorithmus existiert?

Geschichte der Berechenbarkeit

- Glaube der klassischen Mathematik:
  "Wenn irgendein Problem präzise dargelegt werden kann, so gibt es am Ende stets eine Lösung oder es kann bewiesen werden, daß keine Lösung existiert."

- Gottfried Wilhelm Leibniz (1646-1716):
  "Es ist ein Algorithmus zu konstruieren, der es gestattet, jede beliebige mathematische Aufgabe zu lösen."

- David Hilbert (1862-1943): Entscheidungsproblem
  "Zu finden ist ein Algorithmus, der zu einer beliebigen gegebenen Aussage in einem Axiomensystem entscheiden kann, ob die Aussage wahr oder falsch ist."

- Kurt Gödel (1906-78): Beweis des Unvollständigkeitssatzes, 1931
  "Es gibt keinen Algorithmus, der als Eingabe irgendeine Aussage über die natürlichen Zahlen enthält, und dessen Ausgabe feststellt, ob diese Aussage wahr oder falsch ist."

- Entscheidungsproblem der mathematischen Logik:
  "Von je 2 Formeln $\boldsymbol{R}$ und $\boldsymbol{S}$ eines Logikkalküls ist zu entscheiden, ob eine deduktive Kette existiert, die von $\boldsymbol{R}$ nach $\boldsymbol{S}$ führt oder nicht."

  Satz von Church, 1936 : "Das Entscheidungsproblem der mathematischen Logik ist algorithmisch unlösbar."

- Algorithmische Unlösbarkeit von Problemen der Informatik:

  1. Halteproblem: Es existiert kein Algorithmus $\boldsymbol{A}$, der zu einem beliebigen Programm $\boldsymbol{P}$ und beliebigen Eingabedaten $\boldsymbol{D}$ mitteilt, ob $\boldsymbol{P}$ anhält oder eine Endlosschleife enthält.
  2. Äquivalenzproblem: Es existiert kein Algorithmus $\boldsymbol{A}$, der zu 2 beliebigen Programmen $\boldsymbol{P_1}$ und $\boldsymbol{P_2}$ feststellen kann, ob sie die gleiche Aufgabe erfüllen, d.h. bei gleicher Eingabe stets dieselbe Ausgabe erzeugen.

- Algorithmusdefinitionen :

  – Kurt Gödel: Algorithmus als Folge rekursiver Funktionen
  – Alonso Church: Lambda-Kalkül
  – Alan Turing: Algorithmus als Funktionsschema einer hypothetischen Maschine

  Beweis der Äquivalenz dieser Algorithmusdefinitionen $\Rightarrow$ eine Definition ist hinreichend!

- Church–Turing–These :
  "Alle vernünftigen Definitionen von Algorithmus sind gleichwertig und gleichbedeutend."

Die Turingmaschine, 1936 (Alan M.Turing,1912-54)

1. Aufbau der Turingmaschine

	Der Mensch besitzt...	Die Turingmaschine besitzt...
1	Alphabet (mit Ziffern u. Sonderz.)	Bandalphabet B (verarbeitbare Zeichen)
2	Bogen Papier (äußerer Speicher) mit geg. Problem	Rechenband mit Eingabealphabet $\boldsymbol{E \subset B}$ (Symbole auf dem Band)
3	Gedächtnis (innerer Speicher)	Innere Zustände (Zustandsmenge S)
4	Auge und Bleistift	Lesevorrichtung (LV) Schreibvorrichtung (SV)
5	Steuerung von Auge und Bleistift	Verschiebevorrichtung (VV) (mit 3 Verschiebungen aus V)
6	Fähigkeit zu elementaren Rechenoperationen	Überführungsfunktion $\boldsymbol{\delta}$ mit: – Zustandsfunktion – Ausgabefunktion – Verschiebefunktion

$\boldsymbol{E = \{e_1, e_2, ..., e_r\}}$ Eingabealphabet (geg. Symbole)
$\boldsymbol{B = \{b_1, b_2, ..., b_m\}}$ Bandalphabet mit $\boldsymbol{E \subset B}$
$\boldsymbol{S = \{s_1, s_2, ..., s_n\}}$ Zustandsmenge
$\boldsymbol{V = \{L, M, R\}}$ Menge der 3 Verschiebungen
Das Spezialsymbol $*$ kennzeichnet oft ein Feld als leer.

2. Funktionsschema der Turingmaschine (Turing-Tafel)

Beispiel $\quad S = \{s_1, s_2, s_3, s_4, s_5\}\,, \quad B = \{\lambda, \heartsuit, \alpha, \beta\}$

S / B	s_1	s_2	s_3	s_4	s_5
λ	$\lambda R s_4$	$\lambda L s_3$	$\lambda R s_1$	$\lambda M s_5$	$\lambda M s_5$
$\heartsuit$	$\alpha M s_2$	$\beta M s_1$	$\heartsuit R s_1$	$\heartsuit L s_1$	$\heartsuit M s_5$
α	$\alpha L s_1$	$\alpha R s_2$	$\heartsuit L s_3$	$\lambda R s_4$	$\alpha M s_5$
β	$\beta L s_1$	$\beta R s_2$	$\lambda L s_3$	$\heartsuit R s_4$	$\beta M s_5$

Anfangszustand (0. Konfiguration):

		♡	♡	♡	♡	♡	♡	♡	♡	♡	♡			
					s_1									

1. Konfiguration:

		♡	♡	♡	α	♡	♡	♡	♡	♡	♡			
					s_2									

2. Konfiguration:

		♡	♡	♡	α	♡	♡	♡	♡	♡	♡			
					s_2									

3. Konfiguration:

		♡	♡	♡	α	β	♡	♡	♡	♡	♡			
						s_1								

usw. usw.

"Letzte" Konfiguration:

										♡	♡			
														s_5

EUKLIDischer Algorithmus – vereinfachte Darstellung der Turing-Tafel

S / B	s_1	s_2	s_3	s_4
λ	Rs_4	Ls_3	Rs_1	!
$\heartsuit$	αs_2	βs_1	Rs_1	Ls_1
α	L	R	$\heartsuit L$	λR
β	L	R	λL	$\heartsuit R$

Definition 2 : Turing–Maschine

1. $T = (E, B, S, \delta, s_0, F)$ ist eine Turing-Maschine, wenn
   - die 3 nichtleeren Mengen
     $E = \{e_1, e_2, ..., e_r\}$ Eingabealphabet
     $B = \{b_1, b_2, ..., b_m\}$ Bandalphabet mit $E \subset B$
     $S = \{s_1, s_2, ..., s_n\}$ Zustandsmenge
   - die 2 Zustände
     $s_0 \in S$ Anfangszustand
     $F \subset S$ Menge der möglichen Endzustände
   - die Überführungsfunktion $\delta : S \times B \longrightarrow S \times V \times B$

   definiert sind.
2. Existiert zu einer Funktion $f(x_1, x_2, ..., x_n)$ eine Turing-Maschine, die nach endlich vielen Schritten anhält, so heißt sie Turing-berechenbar.
3. Church-These: Die Klasse der intuitiv berechenbaren Funktionen ist gleich der Klasse der Turing-berechenbaren Funktionen.

1.2 Algorithmische Sprachen

Die Notwendigkeit der Betrachtung (formaler) Sprachen resultiert aus dem Bedürfnis, Mitteilungen an den Computer zu übergeben. Mitteilungen komplizierter Art sind jedoch an eine Sprache gebunden.

Formale Sprachen (z.B. Programmiersprachen)

- Syntaktische Definition der Sprache: Festlegung darüber, welche Zeichenkette ein Programm der Sprache ist und welche nicht!
- Notwendigkeit der exakten formalisierten Beschreibung:
  - Programmierer und Computer müssen die gleichen Programme als richtig bzw. falsch ansehen!
  - Programme sollen auf möglichst vielen Computern unter möglichst vielen Betriebssystemen laufen!
  - Programme sollen der Dokumentation von Algorithmen dienen!

1.2.1 Generative Grammatiken

Elemente formaler Sprachen

- Alphabet $\boldsymbol{T}$: Menge aller Grundsymbole (Terminalsymbole, terminals) der Sprache
- Sätze φ der Sprache: Aneinanderreihung von Zeichen zu Zeichenketten (Sätzen) der Sprache:

$$\varphi = z_1 z_2 z_3 ... z_n \ , \quad z_i \in \boldsymbol{T}$$

  Länge $|\varphi|$ der Zeichenkette = Anzahl der Zeichen von φ
- Zeichenkettenmenge $\boldsymbol{T}^*$: Menge aller möglichen Zeichenketten endlicher Länge über dem Alphabet $\boldsymbol{T}$

- Leere Zeichenkette ε :
  Sie gehört zu jeder Zeichenkettenmenge $\boldsymbol{T}^*$ und hat die Länge 0.
- Formale Sprache $\boldsymbol{L}$:
  Eine formale Sprache ist eine Teilmenge $\boldsymbol{L} \subset \boldsymbol{T}^*$.
- Syntax $\boldsymbol{P}$ der Sprache $\boldsymbol{L}$:
  Die Regeln, die entscheiden, welche Elemente aus $\boldsymbol{T}^*$ zur Sprache $\boldsymbol{L}$ gehören, bilden die Syntax von $\boldsymbol{L}$. Sie beschreiben, auf welche Weise die Zeichen aus $\boldsymbol{T}$ zu kombinieren sind, um zu einem Satz φ der Sprache $\boldsymbol{L}$ zu gelangen ("Produktionsregeln").
- Semantik der Sprache $\boldsymbol{L}$:
  Sie beschreibt die Bedeutung der Zeichen bzw. Zeichenkombinationen, d.h. sie ist ein Regelwerk zur inhaltlichen Gestaltung der Sätze.

Beispiel 1 Alphabet sei $\boldsymbol{T} = \{\boldsymbol{a}, +\}$

1. Zeichenkettenmenge

$$\boldsymbol{T}^* = \{\varepsilon, \boldsymbol{a}, +, \boldsymbol{aa}, \boldsymbol{a}+, +\boldsymbol{a}, ++, \boldsymbol{aa}+, ...\}$$

2. Syntax einer Sprache SUMME

   "Zur Sprache SUMME gehören alle die Sätze, die aus einem A, gefolgt von $\boldsymbol{n}$ $(\boldsymbol{n} \geq \boldsymbol{0})$ Zeichenketten $+$a bestehen."

   Damit ist

   SUMME $= \{\boldsymbol{a}, \boldsymbol{a}+\boldsymbol{a}, \boldsymbol{a}+\boldsymbol{a}+\boldsymbol{a}, \boldsymbol{a}+\boldsymbol{a}+\boldsymbol{a}+\boldsymbol{a}, ...\} \subset \boldsymbol{T}^*$

3. Semantik der Sprache SUMME:
   a ist als Variable und $+$ als Additionsoperator zu interpretieren.

Nachteil: Explizite Aufzählung aller Sätze der Sprache ist i.allg. nicht möglich!

Generative Grammatiken G

- Bedeutung:
  Sie liefern eine Konstruktionsvorschrift zur Erzeugung derjenigen Zeichenketten aus T^* , die zu einer Sprache L mit dem Alphabet T gehören $\Longrightarrow$ prinzipiell sind sämtliche Sätze der Sprache angebbar.

- Metasymbole:
  Das Alphabet T der Grundsymbole (terminals) wird durch Metasymbole (non-terminals) zu einem Vokabular V erweitert. Dann ist $V - T$ die Menge der Metasymbole.

- Syntaktische Regeln P (Produktionen):
  Sie werden als geordnetes Paar $(u, v) \in P$ dargestellt. Dabei ist

$$u \in V^* - T^* \quad \text{und} \quad v \in V^* - \{\varepsilon\}$$

  Wirkung: Stehen u und v in der Relation (u, v), so darf u in jeder Zeichenkette durch v substituiert werden.

Definition 3 : Erzeugung (Ableitung)

(i) Für $\varphi, \psi \in V^*$ heißt $\varphi \rightarrow \psi$ einfache Erzeugung (Ableitung) von ψ aus φ, wenn es Zeichenketten $\varphi_1, \varphi_2, u, v \in V^*$ gibt, so daß gilt

$$\varphi = \varphi_1 u \varphi_2 \,, \quad \psi = \varphi_1 v \varphi_2 \quad \text{mit} \quad (u, v) \in P \,.$$

(ii) Für $\varphi, \psi \in V^*$ heißt $\varphi \xrightarrow{*} \psi$ eine Erzeugung (Ableitungsfolge) von ψ aus φ, wenn es Zeichenketten $\varphi_i \in V^*$ für $i = 0, 1, 2, ..., n$ gibt, so daß gilt

$$\varphi_0 = \varphi \,, \; \varphi_n = \psi \quad \text{und} \quad \varphi_i \rightarrow \varphi_{i+1} \quad \text{mit} \quad i = 0(1)n-1 \,.$$

Beispiel 2

- Alphabet: $\boldsymbol{T = \{a, b, c\}}$
- Vokabular: $\boldsymbol{V = \{A, B, C, D\} \cup T}$
- Syntaktische Regeln:

$$\begin{aligned} \boldsymbol{P} \;=\; & \boldsymbol{\{(A, aD), (A, aAD), (D, BC), (CB, BC),} \\ & \boldsymbol{(aB, ab), (bB, bb), (bC, bc), (cC, cc)\}} \end{aligned}$$

Beispiele für Erzeugungen:

$$\begin{aligned} \boldsymbol{aBB} &\overset{(5)}{\rightarrow} \boldsymbol{abB} \overset{(6)}{\rightarrow} \boldsymbol{abb} \\ \boldsymbol{A} &\overset{(1)}{\rightarrow} \boldsymbol{aD} \overset{(3)}{\rightarrow} \boldsymbol{aBC} \overset{(5)}{\rightarrow} \boldsymbol{abC} \overset{(7)}{\rightarrow} \boldsymbol{abc}\,. \end{aligned}$$

Ursprungsketten sind $\boldsymbol{aBB}$ und $\boldsymbol{A}$.

Definition 4 : Grammatik **(A.N.Chomsky, 1959)**

Eine Grammatik (des Types 0) ist ein Quadrupel
$\boldsymbol{G = (V, T, \sigma, P)}$ mit

- $\boldsymbol{V}$ endliche nichtleere Menge (Vokabular)
- $\boldsymbol{T}$ echte nichtleere Teilmenge von $\boldsymbol{V}$ (Alphabet, Grundsymbole)
- $\boldsymbol{\sigma}$ ein Element aus $\boldsymbol{V - T}$ (Satzsymbol, Startsymbol)
- $\boldsymbol{P}$ endliche nichtleere Teilmenge von $\boldsymbol{(V^* - T^*) \times (V^* - \{\varepsilon\})}$ (Produktionsregeln).

$\boldsymbol{V - T}$ sind die Metasymbole (non-terminals), $\boldsymbol{T}$ die Grundsymbole (terminals). Die Ursprungskette ist auf ein einziges Metasymbol $\boldsymbol{\sigma}$ reduziert!

Beispiel 2 (s.o.) $G = (V, T, \sigma, P)$ mit

- Alphabet $T = \{a, b, c\}$
- Vokabular $V = \{A, B, C, D, a, b, c\}$
- Satzsymbol $\sigma = A$
- Syntaktische Regeln

$$P = \{(A, aD), (A, aAD), (D, BC), (CB, BC), (aB, ab), (bB, bb), (bC, bc), (cC, cc)\}$$

Welche Ableitungen aus dem Satzsymbol A sind möglich?

$$A \overset{(*)}{\rightarrow} abc\,, \quad A \overset{(*)}{\rightarrow} aabbcc\,, \quad A \overset{(*)}{\rightarrow} aaabbbccc$$

bzw. allgemein ist die Menge aller aus A ableitbaren Zeichenketten (Sätze)

$$L(G) = \{x \mid x = a^n b^n c^n\ ,\ n \in \mathbb{N}\}$$

Definition 5 : Sprache

Eine durch eine Grammatik $G = (V, T, \sigma, P)$ definierte (formale) Sprache $L(G)$ ist die Menge

$$L(G) = \{x \mid x \in T^* \quad \text{mit} \quad \sigma \overset{(*)}{\rightarrow} x\}\,.$$

Eine Sprache ist also die Menge aller Zeichenketten aus T^* , die aus dem Satzsymbol (Startsymbol) σ erzeugbar (ableitbar) sind!

Beispiel 3

Sprache "Natuerliche Zahl" $G = (V, T, \sigma, P)$ mit

- $T = \{0, 1, 2, 3, 4, 5, 6, 7, 8, 9\}$
- $V = \{\langle\text{Ziffer}\rangle, \langle\text{Natuerliche Zahl}\rangle\} \cup T$
- $\sigma = \langle\text{Natuerliche Zahl}\rangle$
- Syntaktische Regeln:

$$\begin{aligned} P \;=\; \{&(\langle\text{Ziffer}\rangle, 0), (\langle\text{Ziffer}\rangle, 1), (\langle\text{Ziffer}\rangle, 2), ..., (\langle\text{Ziffer}\rangle, 9),\\ &(\langle\text{Natuerliche Zahl}\rangle, \langle\text{Ziffer}\rangle),\\ &(\langle\text{Natuerliche Zahl}\rangle, \langle\text{Natuerliche Zahl}\rangle\langle\text{Ziffer}\rangle)\} \end{aligned}$$

Einige Erzeugungen aus dem Satzsymbol:

$$\begin{aligned} \sigma &\rightarrow \langle\text{Ziffer}\rangle \rightarrow 1\,,\\ \sigma &\rightarrow \langle\text{Natuerliche Zahl}\rangle\langle\text{Ziffer}\rangle \rightarrow \langle\text{Ziffer}\rangle\langle\text{Ziffer}\rangle \rightarrow 27\,,\\ \sigma &\overset{(*)}{\rightarrow} 4711\,,\\ \sigma &\overset{(*)}{\rightarrow} 007\,,\\ \sigma &\overset{(*)}{\rightarrow} 0 \text{ usw.} \end{aligned}$$

Definition 6 : Kontextfreie Grammatik

Eine Grammatik $G = (V, T, \sigma, P)$ heißt kontextfrei (oder: Typ-2-Grammatik), wenn alle Produktionen die Form

$$(A, v) \quad \text{mit} \quad A \in V - T\,,\; v \in V^* - \{\varepsilon\}$$

besitzen.

1.2.2 Backus-Naur-Form und Syntaxdiagramm

Idee der Backus-Naur-Form:

Bei kontextfreien Sprachen sind die Regeln (A, v) mit $A \in V - T$ in der Form von Definitionsgleichungen $A ::= v$ darstellbar.

Erweiterte Backus-Naur-Form (EBNF):

- Bildung syntaktischer Variabler:
  Darstellung der Metasymbole durch Zeichenketten (Mnemotechnik), z.B.

  $$\langle\text{Ziffer}\rangle,\ \langle\text{Konstante}\rangle,\ \langle\text{Bedingte Anweisung}\rangle,\ \langle\text{Programm}\rangle$$

- Definitionssymbol $::=$
  Die Regel (A, v) wird als Definitionsgleichung $A ::= v$ dargestellt, z.B.

  $$\begin{aligned}\langle\text{Ziffer}\rangle &::= \mathbf{3}\\ \langle\text{Vorzeichen}\rangle &::= -\\ \langle\text{Vorzeichen}\rangle &::= +\\ \langle\text{Ganze Zahl}\rangle &::= \langle\text{Vorzeichen}\rangle\langle\text{Ziffernfolge}\rangle\end{aligned}$$

- Alternativsymbol $|$
  Zusammenfassung mehrerer Regeln $(A, v_1), (A, v_2), ..., (A, v_n)$ zu einer Definitionsgleichung

  $$A ::= v_1 \mid v_2 \mid v_3 \mid ... \mid v_n \mid$$

  z.B.

  $$\begin{aligned}\langle\text{Vorzeichen}\rangle &::= + \mid -\\ \langle\text{Ziffer}\rangle &::= \mathbf{0}|\mathbf{1}|\mathbf{2}|\mathbf{3}|\mathbf{4}|\mathbf{5}|\mathbf{6}|\mathbf{7}|\mathbf{8}|\mathbf{9}\\ \langle\text{Ziffernfolge}\rangle &::= \langle\text{Ziffer}\rangle \mid \langle\text{Ziffernfolge}\rangle\langle\text{Ziffer}\rangle\end{aligned}$$

- Optionales Wiederholungssymbol $\{\ \}$
  Eine durch $\{\ \}$ geklammerte syntaktische Konstruktion kann beliebig oft stehen, muß jedoch nicht vorkommen, z.B.

$$\langle\text{Ziffernfolge}\rangle ::= \langle\text{Ziffer}\rangle\ \{\langle\text{Ziffer}\rangle\}$$

Beispiel 4 **Sprache "Bezeichner"** $G = (V, T, \sigma, P)$

- T : $A, B, C, ..., Y, Z, a, b, c, ..., z, 0, 1, 2, ..., 8, 9, \_$
- $V-T$: $\langle$Ziffer$\rangle$, $\langle$Buchstabe$\rangle$, $\langle$Buchstabe oder Ziffer$\rangle$, $\langle$Bezeichner$\rangle$
- σ : $\langle$Bezeichner$\rangle$
- Syntax P :

$$\begin{aligned}
\langle\text{Ziffer}\rangle &::= 0|1|2|3|4|5|6|7|8|9 \\
\langle\text{Buchstabe}\rangle &::= A|B|C|...|Z|a|b|c|...|y|z \\
\langle\text{Buchstabe oder Ziffer}\rangle &::= \langle\text{Buchstabe}\rangle|\langle\text{Ziffer}\rangle|\ _ \\
\langle\text{Bezeichner}\rangle &::= \langle\text{Buchstabe}\rangle\{\langle\text{Buchstabe oder Ziffer}\rangle\}
\end{aligned}$$

Syntaxdiagramm (N.Wirth)

Das Syntaxdiagramm ist eine 2-dimensionale Darstellung der Syntax mit folgenden Eigenschaften:

- Die Abarbeitung erfolgt von links nach rechts.
- Definierte Grundsymbole (terminals) erscheinen in ovalen Kästchen.
- Durch Syntaxdiagramme definierte syntaktische Variablen (non-terminals) erscheinen in rechteckigen Kästchen.
- Alternativen | werden durch Verzweigungen realisiert.
- Wiederholungen $\{\ \}$ werden durch Rückführungen realisiert.

Kapitel 2

Strukturierte Programmierung

Pascal ist eine allgemein anwendbare problemorientierte (imperative) Programmiersprache, die ursprünglich von Niklaus Wirth entwickelt wurde und nach Blaise Pascal, dem berühmten Philosophen und Mathematiker des 17. Jahrhunderts benannt wurde.

Historie:

- 1971 – erste Vorstellung der Sprache durch N. Wirth (TU Zürich)
- 1974 – Veröffentlichung des Sprachenreports (K. Jensen, N. Wirth)
- 1975 – Concurrent Pascal: für "Großrechner" mit Multiprogramming
- 1983 – Turbo Pascal (TP): von BORLAND INT. für PC konzipiert und mit effektiver Entwicklungsumgebung versehen
- 1986 – Object Pascal: kleine Erweiterung von Pascal um wesentliche objektorientierte Sprachelemente
- 1992 – Borland Pascal (**BP**): Weiterentwicklung von Turbo Pascal
- 1991 – Pascal-XSC (PXSC): Pascal eXtensions for Scientific Computation (U.Kulisch u.a., Uni Karlsruhe)
- 1990 – Standard: ISO Standard Pascal (ISO 7158)
- 1991 – Standard: ISO Extended Pascal Standard (ISO 10206)
- 1997 – GNU Pascal (**GPC**): GNU Project Pascal Compiler (unter ständiger Weiterentwicklung)
- 2002 – Free Pascal (**FP**): Version 1.0.6 (unter Entwicklung)

Zielsetzungen:

- Systematisches Erlernen des strukturierten Programmierens
- Entwicklung von Implementationen, die zuverlässig und effektiv sind
- Universelle Sprache für viele Einsatzgebiete und Rechnertypen

2.1 Sprachelemente und Datentypen

2.1.1 Grundlegende Sprachelemente

Alphabet (Grundsymbole) von BP, FP, GPC :

$$\begin{aligned}
\langle\text{Buchstabe}\rangle &::= \boldsymbol{A|B|C|...|Z|a|b|c|...|y|z} \\
\langle\text{Ziffer}\rangle &::= \mathbf{0|1|2|3|4|5|6|7|8|9} \\
\langle\text{Sonderzeichen}\rangle &::= +\,|\,-\,|\,*\,|\,=\,|\,<\,|\,>\,|\,\ldots\,|\langle\text{Reserviertes Wort}\rangle \\
\langle\text{Grundsymbol}\rangle &::= \langle\text{Buchstabe}\rangle|\langle\text{Ziffer}\rangle|\langle\text{Sonderzeichen}\rangle
\end{aligned}$$

Merke: Pascal unterscheidet nicht zwischen Groß- und Kleinbuchstaben, es ist nicht "case sensitiv"!

$$\begin{aligned}
\langle\text{Reserviertes Wort}\rangle ::= {} & \boldsymbol{div|mod|or|and|not|xor|} \\
& \boldsymbol{if|then|else|for|do|while|} \\
& \boldsymbol{begin|end|const|var|type|...|} \\
& \boldsymbol{function|procedure|unit|program|}
\end{aligned}$$

Merke: Möglich sind z.B. `program, Program, PROGRAM`

Begrenzer von BP, FP, GPC:

- Sie dienen der Trennung der Sprachelemente. Zwei Sprachelemente müssen wenigstens von einem Begrenzer unterbrochen werden!
- Begrenzer sind (a) Leerzeichen, (b) Neue Zeile (CR), (c) Sonderzeichen außer reservierten Wörtern.
- Wo 1 Leerzeichen stehen kann, dürfen beliebig viele stehen!

2.1.2 Skalare Standardtypen

Ein **Datentyp** definiert die Art der Werte, die eine Variable annehmen kann. Jede Variable in einem Programm muß genau einem Datentyp zugeordnet werden. Das geschieht durch eine Deklaration.
BP unterscheidet skalare Datentypen und strukturierte Datentypen. Letztere werden aus den skalaren Typen aufgebaut.

Standard-Datentypen von BP, FP und GPC:

1. Natürliche Zahlen (unsigned-Typen):
   `byte, word, [cardinal, Qword, shortword, longword]`
2. Ganze Zahlen (signed-Typen):
   `shortint, integer, [smallint, longint, int64, byteint]`
3. Reelle Zahlen (float-Typen):
   `real, single, double, extended`
4. Logische Wahrheitswerte (Boolean-Typen):
   `boolean` mit Werten `false, true` (1 Byte)
5. Alphanumerische Zeichen (Character-Typen):
   `char` mit Werten gemäß ASCII-Code (1 Byte)

Float-Standardtypen für reelle Zahlen (in BP, FP, GPC):

Typ	Wertebereich	Bytes	Stellen
`single`	$1.5 \cdot 10^{-45}$ .. $3.4 \cdot 10^{38}$	4	7 – 8
`real`	$2.9 \cdot 10^{-39}$ .. $1.7 \cdot 10^{38}$	6	11 – 12
`double`	$5.0 \cdot 10^{-324}$ .. $1.7 \cdot 10^{308}$	8	15 – 16
`extended`	$1.9 \cdot 10^{-4951}$ .. $1.1 \cdot 10^{4932}$	10	19 – 20

ANSI/IEEE - Standard 754 zur Gleitpunktarithmetik

Zahlenformate :	**single**	single extended	**double**	double extended
Wortbreite	32 Bit 4 Byte	$\geq$ 43 Bit $\geq$ 6 Byte	64 Bit 8 Byte	$\geq$ 79 Bit $\geq$ 10 Byte
Mantissenbreite	24 Bit	$\geq$ 32 Bit	53 Bit	$\geq$64 Bit
Exponentenbreite	8 Bit	$\geq$ 11 Bit	11 Bit	$\geq$15 Bit
hidden bit	ja	undefiniert	ja	undefiniert
bias	127	undefiniert	1023	undefiniert
max. Exponent	$\mathbf{+127}$	$\mathbf{\geq +1023}$	$\mathbf{+1023}$	$\mathbf{\geq +16383}$
min. Exponent	$\mathbf{-126}$	$\mathbf{\leq -1022}$	$\mathbf{-1022}$	$\mathbf{\leq -16382}$
Dezimalstellen	$\mathbf{6.9 - 7.2}$	$\mathbf{\geq 9.3 - 9.6}$	$\mathbf{15.6 - 15.9}$	$\mathbf{\geq 18.9 - 19.2}$
größte Zahl	$\mathbf{3.4 * 10^{+38}}$	$\mathbf{\geq 1.7 * 10^{+308}}$	$\mathbf{1.7 * 10^{+308}}$	$\mathbf{\geq 1.1 * 10^{+4932}}$
kleinste norm. Zahl	$\mathbf{1.2 * 10^{-38}}$	$\mathbf{\leq 2.3 * 10^{-308}}$	$\mathbf{2.3 * 10^{-308}}$	$\mathbf{\leq 3.4 * 10^{-4932}}$
kl. denorm. Zahl	$\mathbf{1.5 * 10^{-45}}$	$\mathbf{\leq 1.7 * 10^{-317}}$	$\mathbf{5.0 * 10^{-324}}$	$\mathbf{\leq 3.7 * 10^{-4951}}$

Literatur :

1. ANSI: American National Standards Institute / IEEE: Institute of Electrical & Electronic Engineers, New York. *A Standard for Binary Floating-Point Arithmetic*, 1985. ANSI/IEEE Standard 754-1985.

2. ANSI: American National Standards Institute / IEEE: Institute of Electrical & Electronic Engineers, New York. *A Standard for Radix-Independent Floating-Point Arithmetic*, 1987. ANSI/IEEE Standard 854-1987.

Integer-Standardtypen (in BP, FP und GPC):

signed	B	Wertebereich	Borland-Pascal	GNU-Pascal	Free-Pascal
nein	1	$0 \,..\, 2^8 - 1$	Byte	Byte	Byte
nein	2	$0 \,..\, 2^{16} - 1$	Word	ShortWord	Word
nein	4	$0 \,..\, 2^{32} - 1$	–	Word	Cardinal
nein	8	$0 \,..\, 2^{64} - 1$	–	LongWord	QWord
ja	1	$-2^7 \,..\, 2^7 - 1$	ShortInt	ByteInt	ShortInt
ja	2	$-2^{15} \,..\, 2^{15} - 1$	Integer	ShortInt	SmallInt
ja	4	$-2^{31} \,..\, 2^{31} - 1$	LongInt	Integer	Integer
ja	8	$-2^{63} \,..\, 2^{63} - 1$	Comp	LongInt	Int64

Bereichsgrenzen:

$$
\begin{aligned}
2^7 &= 128 \\
2^8 - 1 &= 255 \\
2^{15} &= 32768 \\
2^{16} - 1 &= 65535 \\
2^{31} &= 2\,147\,483\,648 \\
2^{32} - 1 &= 4\,294\,967\,295 \\
2^{63} &= 9\,223\,372\,036\,854\,775\,808 \\
2^{64} - 1 &= 18\,446\,744\,073\,709\,551\,615
\end{aligned}
$$

Äquivalente Bezeichnungen in GPC:

```
Byte     =  ByteCard
ShortWord  =  ShortCard
Word     =  Cardinal
LongWord   =  LongCard
```

2.2 Programmaufbau

Jedes Programm besteht aus dem Programmkopf, gefolgt vom Programmblock. Der (Programm-)Block setzt sich aus dem Deklarationsteil und dem Anweisungsteil zusammen.

Beispiel 1 **Kugelberechnung**

Gegeben ist der Radius $r = \mathbf{5.12007}$ einer Kugel; gesucht sind Volumen und Oberfläche:

$$ofl = 4\pi r^2\,, \qquad vol = \frac{4}{3}\pi r^3 = \frac{1}{3} ofl \cdot r$$

Programm (1.Version):

```
Program Kugel;
  { Bestimmung von Oberflaeche und
    Volumen einer Kugel mit Radius r }

var
  r, ofl, vol : real;

begin
  r   := 5.12007;
  ofl := 4.0 * pi * r * r;
  vol := ofl / 3.0 * r

end.
```

Kommentare: Zur Dokumentation dürfen Kommentare überall im Programm geschrieben werden, wo ein Begrenzer erlaubt ist. Sie sind in Klammern { und } zu setzen!

2.2.1 Programmkopf

Syntax:

⟨Programmkopf⟩ ::= Program ⟨Bezeichner⟩ (⟨Programmparameter⟩);

⟨Programmparameter⟩ ::= ⟨Bezeichner⟩{, ⟨Bezeichner⟩}

Semantik:

- Der Programmname hinter `Program` sollte im Programmblock nicht benutzt werden.
- Über Programmparameter kann das Programm mit seiner Umgebung kommunizieren. In ISO-Pascal sind sie obligatorisch; ggf. sind die Standarddateien `Input` und `Output` zu benutzen.
- In BP können Programmparameter fehlen; dann werden `Input` und `Output` gesetzt.

Beispiele:

```
Program Kugel;
Program Loesung_einer_Gleichung;
Program Sortieren (Input,Output);
Program Kopiere (Quellendatei, Zieldatei);
```

2.2.2 Deklarationsteil

Bedeutung der Deklaration:

Alle in einem Programm auftretenden Objekte – außer Standard-Objekten – müssen zuerst deklariert (bekanntgemacht) werden, bevor sie im anschließenden Anweisungsteil verarbeitet werden können.

Objekt	Deklarationsart	Res. Wort
Variable	Variablendeklaration	`Var`
Konstante	Konstantendeklaration	`Const`
Datentyp	Typdeklaration	`Type`
Funktion	Funktionsdeklaration	`Function`
Verfahren	Prozedurdeklaration	`Procedure`
Sprungmarke	Markendeklaration	`Label`

Variablendeklaration:

Jede Variable muß vor ihrer Verwendung genau einmal deklariert werden. Die Deklaration legt fest, zu welchem Datentyp die möglichen Werte der deklarierten Variablen gehören. Die Deklaration dient der Bereitstellung von Speicherplatz entsprechend dem angegebenen Typ.

Beispiele:

```
Var
  i,j,index,k2      : integer;
  a, radius_1, x,z  : double;
  p, q              : Boolean;
  zeichen           : char;
```

Syntax (vorläufig):

⟨Var-Deklaration⟩ **::=** ⟨Bezeichner⟩**{,** ⟨Bezeichner⟩**}** **:** ⟨Typ⟩

⟨Var-Deklarationsteil⟩ **::=** Var ⟨Var-Deklaration⟩**{;** ⟨Var-Deklaration⟩**};**

Konstantendeklaration:

Im Konstantendeklarationsteil führt der Programmierer nach eigenem Ermessen Namen (d.h. Synonyme) für Konstanten ein. Jeder auftretende Konstantenbezeichner muß vor seiner Verwendung deklariert werden. Ausnahmen sind Standardkonstanten (z.B. `pi`, `maxint`, `true`, `false`). Man unterscheidet untypisierte Konstanten und typisierte Konstanten.

Beispiele (untypisierte Konstanten):

```
Const
  n = 2048;              { Mathemat. Konstanten }
  e = 2.718281828459;
  g = 9.81;              { Physikal. Konstanten }
  Boltzmann = 1.3804E-23;

Const                    { Weitere Konstanten }
  PLZ      =  98693;
  Kennwort =  'SESAM';
  leer10   =  '          ';
```

Syntax (vorläufig):

⟨Const-Definition⟩ ::= ⟨Bezeichner⟩ = ⟨Konstante⟩|
⟨Bezeichner⟩ : ⟨Typ⟩ = ⟨Konstante⟩

⟨Const-Deklarationsteil⟩ ::= Const ⟨Const-Definition⟩
{; ⟨Const-Definition⟩};

⟨Konstante⟩ ::= ⟨vorzeichenlose Zahl⟩|
⟨Vorzeichen⟩⟨vorzeichenlose Zahl⟩|
⟨Konstantenbezeichner⟩|
⟨Vorzeichen⟩⟨Konstantenbezeichner⟩|
⟨Zeichenkette⟩

BP läßt anstelle der Konstanten auch konstante Ausdrücke zu, die während der Compilierung jedoch auswertbar sein müssen. Typisierte Konstanten werden genau einmal beim Start des Programmes initialisiert.

Beispiele (typisierte Konstanten):

```
Const
  n: Integer       = 2048;         { Mathemat. Konstante }
  Boltz: Double    = 1.3804E-23;   { Physikal. Konstante }
  Messg: String[20]= 'Hello girls';  { Zeichenkettentyp }
  n2: Integer      =  n*n-1;       { Konstanter Ausdruck }
```

Die Deklaration von Typen, Funktionen und Prozeduren wird später behandelt. Auf Marken wird im Interesse des strukturierten Programmierens bewußt verzichtet.

2.2.3 Anweisungsteil

Der Anweisungsteil eines Programmes legt den Algorithmus fest, der nach dem Programmstart abzuarbeiten ist. Er hat die Form einer Verbundanweisung und folgt unmittelbar dem Deklarationsteil.

Syntax:

⟨Anweisungsteil⟩ **::=** ⟨Verbundanweisung⟩

⟨Verbundanweisung⟩ **::=** Begin ⟨Anwweisung⟩ **{,** ⟨Anweisung⟩**}**End**.**

Merke: Das Semikolon dient der Trennung von Anweisungen, nicht aber deren Abschluß!

Beispiel (s.o.):

```
begin
  r   := 5.12007;
  ofl := 4.0 * pi * r * r;
  vol := ofl / 3.0 * r
end.
```

Zuweisungs-Anweisung (Ergibtanweisung, Wertzuweisung)

Als grundlegendste Anweisung dient sie dazu, den berechneten Wert eines Ausdruckes einer Variablen zuzuweisen. In Pascal fungiert := als Zuweisungsoperator.

Beispiele:

```
var
  i,k           : integer;
  r, ofl, vol   : real;
  x, y, z       : double;
  p, q          : Boolean;
begin
  {integer-Zuweisungen}
    i := 1;
    k := k+2;
  {float-Zuweisungen}
    ofl := 4.0 * pi * r * r;
    vol := ofl / 3.0 * r;
    z   :=  x / y / z;
  {Boolean-Zuweisung}
    p   :=  p and q or p and not q;
  {Gemischte Zuweisungen}
    x   :=  i mod k;
    z   :=  x / r / y;
    x   := (4 + x) * (x*x - r) - 1;
end.
```

Merke: Typ der Variablen und Typ des Ausdruckes müssen übereinstimmen. Ausnahmen bilden "miteinander verträgliche Typen", z.B. wenn der Typ des Ausdruckes ein Teilbereich des Variablentyps ist.

2.2.4 Ein- und Ausgabe, Geradeausprogramm

Ein- und Ausgabe über Standarddateien

- Eingabe über die Standarddatei `Input` mit den Prozeduren
  - `read` – Eingabe von Zeichen,
  - `readln` – Eingabe einer Zeile.
- Ausgabe über die Standarddatei `Output` mit den Prozeduren
  - `write` – Ausgabe von Zeichen,
  - `writeln` – Ausgabe einer Zeile.

Beispiel 1 **Kugelberechnung (2.Version)**

```
Program Kugel;
  { Bestimmung von Oberflaeche und
    Volumen einer Kugel mit Radius r }
var
  r, ofl, vol : real;
begin
  writeln ('Kugelberechnung');
  writeln ('===============');
  writeln;
  write ('Radius  r = ');
  readln (r);
  ofl := 4.0 * pi * r * r;
  vol := ofl / 3.0 * r;
  writeln ('Oberflaeche  ofl = ', ofl);
  writeln ('Volumen      vol = ',vol);
  readln
end.
```

- Eingabeanweisung read:

  Die allgemeine Form lautet

  ```
  read ( var_1, var_2, ... , var_n );
  ```

  mit Variablenbezeichnern `var_1, var_2, ... , var_n` vom Integer-Typ, Float-Typ, Char-Typ oder String-Typ. Die Anweisung wird wie die Folge

  ```
  read (var_1); read(var_2); ... ; read(var_n );
  ```

  ausgeführt.

- Eingabeanweisung readln:

  Die allgemeine Form

  ```
  readln ( var_1, var_2, ... , var_n ); oder readln;
  ```

  ist äquivalent der Folge

  ```
  read (var_1); ... ; read(var_n ); readln;
  ```

  Allgemeine Wirkung von `readln` : Übergang zu einer neuen Zeile in der Eingabedatei. Bei Tastatureingabe wartet der Computer auf die Eingabe eines Zeileabschlusses (End-of-Line, CR/LF), was durch ENTER bewirkt wird.

  Beispiele:
  ```
  readln (Name, Konto-Nr, Guthaben);
  read (r, Dichte);
  readln;
  ```

- Ausgabeanweisungen write und writeln:

  Die allgemeine Form lautet

  ```
  write   ( par_1, par_2, ... , par_n );   bzw.
  writeln ( par_1, par_2, ... , par_n );
  ```

  mit Ausgabeparametern `par_1, par_2, ... , par_n` vom Integer-Typ, Float-Typ, Char-Typ, Boolean-Typ oder String-Typ.

Die Anweisungen werden wie die Folgen

```
write (par_1); ... ; write(par_n );   bzw.
write (par_1); ... ; write(par_n ); writeln;
```

ausgeführt.

Allgemeine Wirkung von `writeln` : Schreiben eines Zeilenende-Kennzeichens (EOLn, End-of-Line) in die Ausgabedatei. Bei Bildschirm- oder Druckerausgabe erfolgt ein Zeilenwechsel mit CR/LF.

- Ausgabeparameter: Die Syntax lautet:

  ⟨Ausgabeparameter⟩ ::= ⟨Ausdruck1⟩|
  ⟨Ausdruck1⟩ : ⟨Ausdruck2⟩|
  ⟨Ausdruck1⟩ : ⟨Ausdruck2⟩ : ⟨Ausdruck3⟩

  Semantik:

  - ⟨Ausdruck1⟩ stellt den auszugebenden Wert dar. Steht er ohne weitere Parameter, so wird die Information in einer standardisierten Form (ohne Informationsverlust) ausgegeben.
  - Die Datenfeld-Längenparameter ⟨Ausdruck2⟩ und ⟨Ausdruck3⟩ dienen der Formatierung; sie müssen vom Integer-Typ sein.
  - Format ⟨Ausdruck2⟩ gibt die gesamte Datenfeldlänge an. Der Eintrag erfolgt rechtsbündig.
  - ⟨Ausdruck3⟩ dient ausschließlich bei Float-Typen der (rechtsbündigen) Festpunktdarstellung mit Rundung zur Angabe der gebrochenen Nachpunkt-Stellen.

Anweisung	Ausgabe
write (1E0);	1.00000000000000E+0000
write (-pi);	-3.14159265358979E+0000
write (-pi:20);	-3.14159265359E+0000
write (-pi:17);	-3.14159265E+0000
write (-pi:11);	-3.14E+0000
write (-pi:10:6);	-3.141593

Beispiel 2 Transformation von Kugelkoordinaten

Die Untersuchung kugelförmiger Bauteile wird oft durch sog. Kugelkoordinaten erleichtert. Jeder Punkt $\boldsymbol{P \neq 0}$ des 3D-Raumes wird eindeutig durch folgende 3 Koordinaten beschrieben:

1. Radius $\boldsymbol{r \geq 0}$ für den Abstand zum Nullpunkt
2. Winkel $\boldsymbol{\varphi}$ zur positiven x-Achse mit $\boldsymbol{-\pi < \varphi \leq \pi}$
3. Winkel $\boldsymbol{\theta}$ zur positiven z-Achse mit $\boldsymbol{0 \leq \theta \leq \pi}$

Ein einzugebender Punkt $\boldsymbol{P = (r, \varphi, \theta)}$ in Gradmaß ist in cartesische Koordinaten $\boldsymbol{P = (x, y, z)}$ zu transformieren. Zusätzlich sollen die Winkel $\boldsymbol{\theta_x}$ und $\boldsymbol{\theta_y}$ des Radiusvektors mit der positiven x- bzw. y-Achse bestimmt werden.

Berechnungsformeln:

$$\begin{aligned} x &= r\cos\varphi\sin\theta \\ y &= r\sin\varphi\sin\theta \\ z &= r\cos\theta \end{aligned} \qquad \begin{aligned} \theta_x &= \frac{\pi}{2} - \arctan\frac{x}{\sqrt{r^2 - x^2}} \\ \theta_y &= \frac{\pi}{2} - \arctan\frac{y}{\sqrt{r^2 - y^2}} \end{aligned}$$

```
Program Kugelkoordinaten;
uses crt;    { Einbinden der Bildschirmprozeduren }
var r, phi, theta, sig, x, y, z, thx, thy : double;
begin
  clrscr;     { Bildschirmloeschen }
  writeln ('Winkel und (x,y,z)-Koordinaten');
  writeln ('============================');
  write   ('Radius (in cm)  = '); readln (r);
  write   ('PHI (in Grad)   = '); readln (phi);
  write   ('THETA (in Grad) = '); readln (theta);
    { Umrechnung in Bogenmass                    }
  phi := pi * phi / 180.0;
  theta := pi * theta / 180.0;
    { Winkelberechnung                           }
  sig := sin (theta);
  x   := sig * cos (phi);
  y   := sig * sin (phi);
  thx := pi / 2.0 - arctan (x / sqrt (1.0 - sqr (x)));
  thy := pi / 2.0 - arctan (y / sqrt (1.0 - sqr (y)));
  writeln ('THETAX = ', thx * 180.0 / pi:7:2, ' Grad');
  writeln ('THETAY = ', thy * 180.0 / pi:7:2, ' Grad');
    { Kartesische Koordinaten (x,y,z)             }
  x := r * x;
  y := r * y;
  z := r * cos (theta);
  writeln ('Kartesische Koordinaten:');
  writeln ('x = ':15, x);
  writeln ('y = ':15, y);
  writeln ('z = ':15, z); readln
end.
```

```
#include <iostream.h>
#include <math.h>

int main() {
  double r, phi, theta, sig, x, y, z, thx, thy;
  cout << "\nKugel- und (x,y,z)-Koordinaten" ;
  cout << "\n==============================" << endl;
  cout << "Radius (in cm)  = ";   cin >> r;
  cout << "PHI (in Grad)   = ";   cin >> phi;
  cout << "THETA (in Grad) = ";   cin >> theta;
    /* Umrechnung in Bogenmass                        */
  phi = M_PI * phi / 180.0;
  theta = M_PI * theta / 180.0;
    /* Winkelberechnung                               */
  sig = sin(theta);
  x = sig * cos(phi);
  y = sig * sin(phi);
  thx = M_PI / 2.0 - atan(x / sqrt(1.0 - x*x));
  thy = M_PI / 2.0 - atan(y / sqrt(1.0 - y*y));
  cout << "THETAX = " << thx * 180.0 / M_PI << endl;
  cout << "THETAY = " << thy * 180.0 / M_PI << endl;
    /* Kartesische Koordinaten (x,y,z)                */
  x = r * x;
  y = r * y;
  z = r * cos(theta);
  cout << "\nKartesische Koordinaten:" << endl;
  cout << "  x = " << x << endl;
  cout << "  y = " << y << endl;
  cout << "  z = " << z << endl;
  return 0;
}
```

Standardfunktionen (Auswahl in BP, FP, GPC):

Aufruf	Bedeutung	Param.-typ	Funkt.-typ
abs(x)	$\|x\|$	int, float	int, float
sqr(x)	x^2	int, float	int, float
sqrt(x)	$\sqrt{x},\ x \geq 0$	int, float	float
exp(x)	e^x	int, float	float
ln(x)	$\ln x,\ x > 0$	int, float	float
sin(x)	$\sin x$, BM	int, float	float
cos(x)	$\cos x$, BM	int, float	float
arctan(x)	$\arctan x$, BM	int, float	float
int(x)	ganzer Anteil	float	float
frac(x)	gebrochener Anteil	float	float
round(x)	Rundung auf ganz	float	integer
trunc(x)	Abschneiden	float	integer
ord(x)	ordinaler Wert	Skalar-Typ	integer
chr(x)	Zeichen mit Ordinalwert x	integer	char
pred(x)	Wert des Vorgängers	Skalar-Typ	Arg.-typ
succ(x)	Wert des Nachfolgers	Skalar-Typ	Arg.-typ
odd(x)	Ist x ungerade ?	int	Boolean
random(x)	gleichverteilte Zufallszahl mit	int	int
random	$0 \leq z < x$ bzw. $0 \leq z < 1$	–	real
upcase(x)	Großgeschriebenes Äquivalent	char	char
keypressed	Taste gedrückt ?	–	Boolean
sizeof(x)	Speicherplatz für x (Bytes)	beliebig	integer

Abkürzungen: BM - Bogenmaß, int - integer-Typ, float - float-Typ

2.3 Kontrollstrukturen

Pascal besitzt folgende strukturierte Anweisungen:

- Verbundanweisung
- Bedingte Anweisungen (if-Anweisung, case-Anweisung)
- Zyklusanweisungen (while-, repeat-, for-Anweisung)

2.3.1 Verbundanweisung

`Begin` und `end` fungieren als Anweisungsklammern. Sie fassen eine Sequenz zu einer einzigen Anweisung zusammen. Die Verbundanweisung wird dort gebraucht, wo mehrere Anweisungen ausgeführt werden sollen, die Syntax jedoch nur eine Anweisung gestattet.

Beispiel:

Zyklische Vertauschung dreier Größen x, y, z

```
begin
  hilf := z;
  z    := y;
  y    := x;
  x    := hilf
end;
```

2.3.2 Bedingte Anweisungen

If–Anweisung: Es existieren 2 Formen

- Verkürzte if-Anweisung – Ausführung einer Anweisung, falls eine Bedingung erfüllt ist
- Vollständige if-Anweisung – Ausführung der Anweisung 1, falls eine Bedingung erfüllt ist, andernfalls Ausführung der Anweisung 2

Syntax:

⟨if -Anweisung⟩ ::= ***if*** ⟨Bedingung⟩ ***then*** ⟨Anweisung⟩|
if ⟨Bedingung⟩ ***then*** ⟨Anweisung 1⟩
else ⟨Anweisung 2⟩

Beispiel 1 Quadratische Gleichung $\boldsymbol{x^2 + ax + b = 0}$ **.**

```
Program Quadratische_Gleichung;
var
  a, b, c, D, w, x1, x2, re, im : real;
begin
  readln (a, b);   c := -a / 2.0;
  D := c * c - b;  w := sqrt (abs (D));

  if D < 0 then

  begin
    re := c; im := w;
    writeln(re, im, 'Loesungen komplex')
  end

  else

  begin
    x1 := c + w; x2 := c - w;
    writeln (x1, x2, 'Loesungen reell')
  end

end.
```

Geschachtelte if–Anweisung:

Im then-Zweig oder im else-Zweig einer if-Anweisungen treten häufig weitere if-Anweisungen auf.
Merke: Eine else-Klausel gehört stets zu dem nach links nächsten (innersten) `if`, das noch keine else-Klausel hat ("else-Regel").

Beispiel:

```
if a > b then if a > c then M := a else M := c;
```

Richtige Interpretation:

```
if a > b then
    if a > c then M := a
             else M := c;
```

Falsche Interpretation:

```
if a > b then
    if a > c then M := a
         else
   M := c;
```

Beispiel 2 Maximum $\max(a, b, c)$ dreier Zahlen

```
Program Maxi;
var
  a, b, c, max : double;
begin
  readln(a);  readln(b);  readln(c);

  if (a > b) and (a>c) then  max := a
                       else
    if c > b then  max := c
             else  max := b ;
  writeln ('MAX = ', max); readln
end.
```

Case–Anweisung:

Sie legt fest, daß aus endlich vielen Anweisungen genau eine auszuführen ist. In Abhängigkeit vom Wert eines Selektors wird entschieden,
– ob eine der aufgelisteten Anweisungen bzw.
– welche dieser Anweisungen
ausgeführt wird.

Struktur der case-Anweisung:

```
case ⟨Selektor⟩ of
     ⟨Auswahlmarken 1⟩ : ⟨Anweisung 1⟩;
     ⟨Auswahlmarken 2⟩ : ⟨Anweisung 2⟩;
     ......
     ⟨Auswahlmarken n⟩ : ⟨Anweisung n⟩
else
     ⟨Anweisung n+1⟩;
     ⟨Anweisung n+2⟩;
     ......
     ⟨Anweisung n+m⟩
end
```

Beispiel 3 Auswahl aus 4 Funktionen

In Abhängigkeit von $nr \in \{1, 2, 3, 4\}$ sind die Funktionswerte

$$\begin{aligned} f_1(x) &= \sin x \\ f_2(x) &= \cos x \\ f_3(x) &= \sinh x = (e^x - e^{-x})/2 \\ f_4(x) &= \cosh x = (e^x + e^{-x})/2 \end{aligned}$$

zu berechnen, wenn x eingegeben wird.

Beispiel 3 Auswahl aus 4 Funktionen

In Abhängigkeit von $nr \in \{1, 2, 3, 4\}$ sind die Funktionswerte

$$\begin{aligned} f_1(x) &= \sin x \\ f_2(x) &= \cos x \\ f_3(x) &= \sinh x = (e^x - e^{-x})/2 \\ f_4(x) &= \cosh x = (e^x + e^{-x})/2 \end{aligned}$$

zu berechnen, wenn x eingegeben wird.

```
Program Funktionen;
var
  nr  : integer;
  x,y : double;
begin
  readln(nr); readln(x);
  case nr of
    1 : y := sin (x);
    2 : y := cos (x);
    3 : begin
          y := exp (x);
          y := (y - 1.0 / y) / 2.0
        end;
    4 : begin
          y := exp (x);
          y := (y + 1.0 / y) / 2.0
        end
  end;
  writeln(x,y); readln
end.
```

```
// Version des Beispiels 3 in C++
#include <iostream.h>
#include <math.h>

int main() {
  long nr;
  double x, y = 1.0E20;

  /*   Eingabe nr und x   */
  cout << "\nFunktionsauswahl" ;
  cout << "\n================" << endl;
  cout << "Funktions - Nr = ";   cin >> nr;
  cout << "Argument     x = ";   cin >> x;

  switch (nr)
  {
   case 1:
     y = sin(x);  break;
   case 2:
     y = cos(x);  break;
   case 3:
     y = exp(x);
     y = (y - 1.0 / y) / 2.0;  break;
   case 4:
     y = exp(x);
     y = (y + 1.0 / y) / 2.0;  break;
  }
  cout << "Funktionswert y = " << y << endl;
  return 0;
}
```

2.3.3 Zyklusanweisungen

While–Anweisung:

Die while-Anweisung wirkt abweisend, d.h. zuerst wird eine Bedingung ausgewertet. Ist deren Wert `true` , so wird der Schleifenkörper (Anweisung) ausgeführt und anschließend erneut die Bedingung getestet. Im anderen Falle wird sofort zur nächsten Programmanweisung übergegangen.

Syntax:

⟨while-Anweisung⟩ **::=** ***while*** ⟨Bedingung⟩
do ⟨Anweisung⟩

Beispiele:

```
while  x > 0.1  do  x := 0.5*x;

while  not q  do
  begin
    readln(x);
    q := abs(x-1.0) < epsilon
  end;
```

PAP: Struktogramm:

Beispiel 4 Bisektionsverfahren für $f(x) = 0$

Gegeben sind eine Funktion $f(x) = \sin x - e^{-x}$ und ein Intervall $[a, b]$ mit $a < b$ und $f(a) \cdot f(b) < 0$. Gesucht ist eine Nullstelle $x \in [a, b]$ mit einer relativen Genauigkeit (Toleranz) $\varepsilon > 0$.

```
Program Bisektion;
  { Funktion f(x) = sin(x) - exp(-x)  }
var
  a, b, epsi, x, y : double;
begin
  writeln ('Bisektionsverfahren');
  readln (a); readln (b); readln (epsi);
  y := sin (a) - exp (-a);
  if y >= 0.0 then
    begin
      x := a; a:= b; b := x
    end;  { Nun ist f(a)<0 und f(b)>0  }

  while abs (a - b) > epsi * abs (a) do

    begin
      x := (a +b) / 2.0;
      y := sin (x) - exp (-x);
      if y < 0.0 then a := x else b := x
    end;  { while-Zyklus }

  writeln ('Nullstelle x = ',x); readln
end.
```

```
// Version des Beispiels 4 in C++
#include <iostream.h>
#include <math.h>

int main() {
  /* Funktion f(x) = sin(x) - exp(-x)  */
  double a, b, epsi, x, y;
  cout << "\nBisektionsverfahren" << endl;
  cout << "====================" << endl;
  cout << "Linkes Intervallende  a   = ";  cin >> a;
  cout << "Rechtes Intervallende b   = ";  cin >> b;
  cout << "Genauigkeitsschranke epsi = ";  cin >> epsi;
  y = sin(a) - exp(-a);
  if (y >= 0.0)
  {
    x = a;  a = b;   b = x;
  }   // Nun ist f(a)<0 und f(b)>0

  while (fabs(a - b) > epsi * (fabs(a)+ 1.0))
  {
    x = (a + b) * 0.5;
    cout << x << endl;
    y = sin(x) - exp(-x);
    if (y < 0.0)
      a = x;
    else
      b = x;
  }  /*   while - Zyklus  */
  cout << "\nNullstelle x = " << x << endl;
  return 0;
}
```

Repeat–Anweisung:

Die repeat-Anweisung ist nicht abweisend, d.h. zuerst wird der Schleifenkörper (Anweisungs-Sequenz) mindestens einmal ausgeführt und anschließend die Bedingung getestet. Ist sie erfüllt, so wird die repeat-Anweisung beendet, andernfalls wird der Schleifenkörper erneut abgearbeitet.

Syntax:

⟨repeat-Anweisung⟩ **::=** ***repeat*** ⟨Anweisung⟩**{;** ⟨Anweisung⟩**}**
until ⟨Bedingung⟩

Beispiele:

```
repeat  x := 0.5*x
until   x < 1.0;

repeat
   readln(x);
   q := abs(x-1.0) < epsilon
until  q;
```

PAP:

Struktogramm:

Beispiel 5 Quadratwurzelbestimmung $\sqrt{x}\,, x > 0$

Mit dem Newton–Verfahren kann man $y = \sqrt{x}$ durch die quadratisch konvergente Iterationsvorschrift

$$y_{n+1} = \frac{1}{2}(y_n + \frac{x}{y_n})\,, \quad n = 0, 1, 2, ...$$

bis auf eine relative Genauigkeit ε bestimmen. Nach jedem Schritt wird geprüft, ob

$$|y_{n+1} - y_n| < \varepsilon |y_{n+1}|$$

gilt.

```
Program Quadratwurzel;
 { sqrt (x) fuer x > 0 }
var
  x, eps, y, ya : double;
begin
  write ('x   = '); readln (x);
  write ('eps = '); readln (eps);
  y := x;

  repeat
    ya := y;
    y  := (ya + x / ya) / 2.0
  until  abs (y - ya) < eps * abs (y);

  writeln ('y = ', y); readln
end.
```

For–Anweisung:

Die for-Anweisung (Laufanweisung) nutzt man für Zyklen, die durch eine Laufvariable (Kontrollvariable) lv gesteuert werden. Diese Variable durchläuft einen Laufbereich von einem Anfangswert aw bis zu einem Endwert ew.

Syntax:

⟨for-Anweisung⟩ ::= ***for*** ⟨Laufvariable⟩ := ⟨Laufliste⟩
do ⟨Anweisung⟩

⟨Laufliste⟩ ::= ⟨Anfangswert⟩ ***to*** ⟨Endwert⟩|
⟨Anfangswert⟩ ***downto*** ⟨Endwert⟩

Beispiele:

```
for  n := -3  to  3  do  x := x+2;
for  n := 3  downto  -1 do  y := y+1;
for  k := n-1  to  2*n+1  do  z := k*z;
for  c := 'a'  to  'z'  do  write(c);
```

Merke:

- lv, aw und ew müssen vom selben Typ sein; dieser einfache Typ darf kein float-Typ sein.
- Vor Abarbeitung der for-Anweisung werden die Werte von aw und ew berechnet.
- Der Wert von lv darf nicht durch die Anweisung zusätzlich verändert werden.
- Die for-Anweisung ist abweisend, d.h. zuerst wird die Zulässigkeit der Laufvariablen geprüft und anschließend ggf. der Schleifenkörper ausgeführt.

Beispiel 6 Arithmetisches Mittel von Meßwerten

Gegeben sind n reelle Meßwerte $a_1, a_2, \ldots, a_n$ einer technischen Größe a . Beim Einlesen ist zu prüfen, ob $a_i > 50$ ist; derartige "Ausreißer" sind bei der Mittelbildung nicht zu berücksichtigen:

$$s := \frac{1}{m} \sum_{\substack{i=1 \\ |a_i| \leq 50}}^{n} a_i \ , \quad m > 0$$

```
Program Mittel;
var
  i, n, m : integer;
  s, a    : double;
begin
  write ('Anzahl n = '); readln (n); writeln;
  m := 0; s := 0.0;
  for i := 1 to n do
    begin
      write ('a(',i,') = '); readln (a);
      if abs (a) <= 50.0 then
        begin
          s := s + a;
          m := m + 1
        end
    end;
    writeln;
    if m = 0 then
      writeln ('Das Mittel s ist nicht bestimmbar!')
    else begin
      s := s / m;  writeln ('Mittel s = ',s)
    end;  readln
end.
```

Geschachtelte Zyklusanweisungen:

Eine Zyklusanweisung kann komplett im Schleifenkörper einer anderen Zyklusanweisung liegen (geschachtelte Zyklen). Die Schleifenkörper dürfen dann einander nicht überschneiden. Geschachtelte for-Anweisungen dürfen nicht ein- und dieselbe Laufvariable besitzen.

Beispiel 7 Tabelle der Binomialkoeffizienten

Binomialkoeffizienten werden insbesondere bei statistischen Untersuchungen benötigt. Zu einer natürlichen Zahl n_{max} sind alle Binomialkoeffizienten

$$\binom{n}{k} = \frac{n}{k!(n-k)!} = \frac{n}{1} \cdot \frac{n-1}{2} \cdot \frac{n-2}{3} \cdots \frac{n-k+1}{k}.$$

für $k = 0(1)n,\ n = 0(1)n_{max}$ zu berechnen und in einer Tabelle auszugeben:

```
n = 0   k = 0   bk = 1

n = 1   k = 0   bk = 1
        k = 1   bk = 1

n = 2   k = 0   bk = 1
        k = 1   bk = 2
        k = 2   bk = 1

n = 3   k = 0   bk = 1
        k = 1   bk = 3
        k = 2   bk = 3
        k = 3   bk = 1
usw.
```

```
Program Binomialkoeffizienten;
var
  nmax, n, k, i, kk : integer;
  bk : double;
begin
  write ('nmax = '); readln (nmax); writeln;

  for n := 0 to nmax do
    begin

      for k := 0 to n do
        begin
          if k > n - k then kk := n - k
                       else kk := k;
          bk := 1.0;

          for i := 1 to kk do
            bk := bk * (n - i + 1) / i;

          if k = 0 then write ('n=':3, n:3)
                   else write (' ':6);
          writeln ('k=':8,k:3,'bk=':9,bk:20)
        end;  {   k - Zyklus  }

      writeln
    end  {  n - Zyklus   }

end.
```

```
/* Output from p2c, the Pascal-to-C translator */
/* From input file "binkoef2.pas" */
#include <iostream.h>

int main()
{
  long nmax, n, k, i, kk;
  double bk;
  cout << endl << "nmax = ";  cin  >> nmax;

  for (n = 0; n <= nmax; n++)
  {
    for (k = 0; k <= n; k++)
    {
      if (k > n - k)
        kk = n - k;
      else
        kk = k;
      bk = 1.0;
      for (i = 1; i <= kk; i++)
        bk = bk * (n - i + 1) / i;
      if (k == 0)
        cout << "n = " << n;
      else
        cout << "     ";
      cout << " k = " << k << "   bk = " << bk << endl;
    }           //  k - Zyklus
    cout << endl;
  }            //  n - Zyklus
  return 0;
}
```

2.4 Prinzipien der Programmentwicklung

Bisher wurden einfache Probleme betrachtet, deren Algorithmierung, Programmierung, Implementierung und Erprobung durch 1 Person ausgeführt werden konnte.
Bei umfangreichen Projekten der Ingenieurpraxis ist jedoch eine Korrespondenz zwischen Auftraggeber, Programmentwickler und Programmnutzer erforderlich, um

– die Struktur der Ein- und Ausgabedaten
– die Struktur des Programmes und seiner Teile

endgültig festzulegen.

Forderung = Systematisierung des Entwicklungsprozesses:

(1) Anforderungsdefinition: Festlegung der Aufgabenstellung

(2) Problemanalyse: Einarbeitung in das Problem, Algorithmierung wesentlicher Teilschritte

(3) Festlegung der logischen Programmstruktur: Struktogramme, Programmablaufplan o.a. Schemata

(4) Entwicklung der physischen Programmstruktur: Programm, Module und Unterprogramme

(5) Programmerprobung: Verifikation, Optimierung

(6) Programmanalyse: Rechnung von Alternativen und Aufwandsvergleiche

Es empfiehlt sich - auch bei kleineren Problemen - bereits diese Schritte zu berücksichtigen.

Beispiel 7 Bessel-Funktion $J_0(x)$

Die Schwingungen einer kreisförmigen, am Rand eingespannten Membran mit Radius a werden durch eine partielle Differentialgleichung beschrieben, deren Lösung mit Hilfe der Bessel-Funktionen 1.Art $J_k(x)$, $k = 0, 1, 2, ...$ notiert werden können. Diese höheren transzendenten Funktionen lassen sich durch Potenzreihen definieren:

$$J_k(x) \;=\; \sum_{n=0}^{\infty} \frac{(-1)^n \cdot x^{2n+k}}{2^{2n+k} \cdot n! \cdot (k+n)!}\,, \quad k = 0, 1, 2, 3, ...$$

Aufgabe: Man erzeuge zu einzugebender Schrittweite $h > 0$ eine Tabelle der Bessel-Funktion $J_0(x)$ für $x = 0(h)6.0$.

Problemanalyse:

- Darstellung von $J_0(x)$:

$$J_0(x) \;=\; 1 - \frac{1}{2^2 \cdot 1}x^2 + \frac{1}{2^4 \cdot 2^2}x^4 - \frac{1}{2^6 \cdot 6^2}x^6 + - \cdots$$

- Frage: Wieviele Glieder sind zu berücksichtigen?

$$\text{mit} \quad a_n := \frac{(-1)^n \cdot x^{2n}}{2^{2n} \cdot (n!)^2} \quad \text{gilt} \quad \lim_{n\to\infty} a_n = 0$$

  Abbruch der Summation, falls $|a_n| < 10^{-10}$ ist.

- Effektive Berechnung der Summanden a_n :

$$a_0 = 1\,, \quad a_{n+1} = -\left[\frac{x}{2(n+1)}\right]^2 \cdot a_n\,,\; n = 0, 1, 2, ...$$

Logische Programmstruktur: Struktogramm

Physische Programmstruktur: 1. Version des Programmes

```
{ Programm Besselfunktion_J0          }
   { Deklarationen                    }
   { Eingabe h, Ausgabe Tabellenkopf  }
   x := 0; xende := 6.0 + 0.1 * h;
   while x <= xende do begin
      {  Berechnung von y = J0(x)     }
      {  Ausgabe von x,y              }
      x := x + h
   end  {  while x                    }
{ Programmende Besselfunktion_J0      }
```

Physische Programmstruktur: 2. Version des Programmes

```
Program Besselfunktion_J0;
  { Deklarationen                     }
  { Eingabe h, Ausgabe Tabellenkopf   }
  x := 0; xende := 6.0 + 0.1 * h;
  while x <= xende do begin
      { Berechnung von y = J0(x)      }
      a:=1.0; n := 0; s := 1.0;
      while abs (a) >= 1e-10 do  begin
          n := n+1;
          a := - sqr (x / 2.0 / n) * a;
          s := s + a
      end;  {  while a                }
      y := s;
      { Ausgabe von x,y               }
      x := x + h
  end  {  while x                     }
end. { Programmende Besselfunktion_J0 }
```

Physische Programmstruktur: Endversion des Programmes

```
Program Besselfunktion_J0;
var   { Deklarationen                    }
  h, x, xende, a, s, y :real;
  n : integer;
begin
  { Eingabe h, Ausgabe Tabellenkopf   }
  writeln ('Tabelle der Besselfunktion J0(x)');
  writeln ('================================');
  write('Schrittweite h = '); readln(h); writeln;
  writeln ('x':10, 'J0(x)':8);
  x := 0; xende := 6.0 + 0.1 * h;
  while x <= xende do
  begin
      { Berechnung von y = J0(x)      }
      a:=1.0; n := 0; s := 1.0;
      while abs (a) >= 1e-10 do
      begin
          n := n+1;
          a := - sqr (x / 2.0 / n) * a;
          s := s + a
      end;  {  while a                }
      y := s;
      { Ausgabe von x,y               }
      writeln (x, y);  x := x + h
  end  {  while x                     }
end. { Programmende Besselfunktion_J0 }
```

Hinweis: Für komplexere Projekte hat sich das **Entwurfsprinzip der schrittweisen Verfeinerung** (Top–down–Entwurf) bewährt.

Endversion des Programmes in C++

```
#include <iostream.h>
#include <math.h>

int main() {
  double h, x, xende, a, s, y;
  long n;

  /*   Eingabe h, Ausgabe Tabellenkopf   */
  cout << "Tabelle der Besselfunktion J0(x)" << endl;
  cout << "================================" << endl;
  cout << "Schrittweite h = ";  cin >> h;
  cout << endl << "x      J0(x)" << endl;
  x = 0.0;  xende = 6.0 + 0.1 * h;
  while (x <= xende)
  {
    /*  Berechnung von y = J0(x)  */
    a = 1.0; n = 0; s = 1.0;
    while (abs(a) >= 1e-10)
    {
      n++;
      a = - (x / 2.0 / n) * (x / 2.0 / n) * a;
      s += a;
    }
    y = s;
    cout << x << "     " << y << endl;
    x += h;
  }
  return 0;
}
```

Kapitel 3

Prozedurale Programmierung

Um Programme zu entwickeln, die den Prinzipien der Universalität (Nutzung allgemeiner Grundprinzipien, leichte Anwendbarkeit) und der Unabhängigkeit (Lauffähigkeit auf verschiedenen Computern, Einsetzbarkeit in andere Programme) genügen, wurde das Konzept der Prozedur entwickelt.

Definition 1 : Prozedur [1]

Eine Prozedur ist eine Folge von Anweisungen (Aktionen), die einen in sich abgeschlossenen Programmteil darstellt, der von einem anderen Programmteil aus aufgerufen, d.h. ausgeführt werden kann. Eine Prozedur hat eine Schnittstelle, die ihren Namen, die zulässigen Eingabeobjekte (auch Argumente genannt) und das Resultat, d.h. die nach der Ausführung vorliegenden Resultatobjekte, spezifiziert.

Das Konstrukt "Prozedur" ist in der **prozedurorientierten (prozeduralen) Programmierung** von zentraler Bedeutung.

Pascal unterscheidet folgende 2 Arten von Unterprogrammen (Routinen):

- Verfahrens-Unterprogramme $\Rightarrow$ Prozeduren
- Funktions-Unterprogramme $\Rightarrow$ Funktionen

[1] Rechenberg,P.; Pomberger,G.(Hrsg.): Informatik-Handbuch. 2. Auflage. Hanser Verlag 1999, S.518

3.1 Prozeduren in PASCAL

3.1.1 Prozedur–Deklaration

Die Prozedur-Deklaration legt (a) die Eingangs- und Ausgangsparameter und (b) den Inhalt des Algorithmus fest. Damit dient sie zugleich der Prozedur-Definition. Jede Prozedur besteht - wie ein Programm - aus dem Prozedurkopf, gefolgt vom Prozedurblock. Der (Prozedur-)Block setzt sich aus dem Deklarationsteil und dem Anweisungsteil zusammen.

(a) Prozedurkopf

Syntax (vorläufig):

⟨Prozedurkopf⟩ ::= Procedure ⟨Prozedurname⟩
(⟨Prozedurparameter⟩);
⟨Prozedurparameter⟩ ::= ⟨Par-Abschnitt⟩{; ⟨Par-Abschnitt⟩}
⟨Par-Abschnitt⟩ ::= Var ⟨Variable⟩{, ⟨Variable⟩} : ⟨einfacher Typ⟩

Semantik:

- Über Prozedurparameter kann die Prozedur mit ihrer Umgebung kommunizieren. Man unterscheidet Eingangs- und Ausgangsparameter.
- Der Prozedurname hinter `Procedure` darf im nachfolgenden Prozedurblock benutzt werden ⇒ rekursive Prozedur.
- In BP können Prozedurparameter fehlen ⇒ parameterlose Prozedur.

Beispiele:

```
Procedure zeile (var n : integer; var ch : char);
  { n, ch - Eingangsparameter; keine Ausgangsparameter }
Procedure Bisektions_Verfahren (var a,b,epsi,x : double);
  { a,b,epsi - Eingangsparameter; x - Ausgangsparameter }
Procedure tausche (var a,b : single);
  { a,b - Eingangsparameter; a,b - Ausgangsparameter }
```

(b) Prozedurblock

Er besteht - wie ein Programmblock - aus Deklarationsteil und Anweisungsteil. Falls nichts zu deklarieren ist, kann der Deklarationsteil fehlen. Merke: Auch für Prozeduren gilt, daß alle dort auftretenden Objekte – außer Standard-Objekten und Prozedurparametern – zuerst deklariert (bekanntgemacht) werden müssen, bevor sie im anschließenden Anweisungsteil verarbeitet werden können!

Beispiel 1 Ausgabe von n Zeichen

Es sind n identische Zeichen der Art ch hintereinander in einer Zeile auszugeben, z.B. '**********' oder ' ' (5 Leerzeichen).

```
Procedure zeile (var n : integer; var ch : char);
  { n, ch - Eingangsparameter }
var
  i : integer;  { lokale "Hilfsvariable" }
begin
  for i:=1 to n do
    write(ch)
end;
```

Beispiel 2 (s.o.) Bisektionsverfahren für $f(x) = 0$

Gegeben sind eine Funktion $f(x) = \sin x - e^{-x}$ und ein Intervall $[a, b]$ mit $a < b$ und $f(a) \cdot f(b) < 0$. Gesucht ist eine Nullstelle $x \in [a, b]$ mit einer relativen Genauigkeit (Toleranz) $\varepsilon > 0$.

```
Procedure Bisektions_Verfahren
            (var a, b, epsi, x : double);
  { a,b,epsi - Eingangsparameter;
           x - Ausgangsparameter    }
var
  y : double;   { lokaler Parameter }
begin
  y := sin (a) - exp (-a);
  if y >= 0.0 then
  begin
     x := a; a:= b; b := x
  end;          { Nun ist f(a)<0 und f(b)>0  }
  while abs (a - b) > epsi * abs (a) do
  begin
     x := (a +b) / 2.0;
     y := sin (x) - exp (-x);
     if y < 0.0 then a := x else b := x
  end           { Nullstelle ist x }
end;
```

Auftretende Parameterarten:

- Prozedurparameter = formale Parameter (`n`, `ch` bzw. `a`, `b`, `epsi`, `x`): Sie dienen der Übergabe von Eingangs- und Ausgangswerten zum aufrufenden Programm. Da die konkreten (aktuellen) Werte zum Zeitpunkt der Deklaration noch nicht bekannt sind, dienen sie als "Platzhalter" (deshalb: formale Parameter).
- Hilfsparameter = lokale Parameter (`i` bzw. `y`):
  Sie werden im Deklarationsteil der Prozedur vereinbart und sind deshalb auch nur im Innern der Prozedur gültig und verfügbar (deshalb: lokale Parameter).

3.1.2 Prozedur–Aufruf

Prozeduren werden im Deklarationsteil eines Programmes (oder einer Prozedur) vereinbart. Bei BP kann dies an beliebiger Stelle des Deklarationsteil geschehen. Der Prozedur-Aufruf im Anweisungsteil aktiviert den Prozedurblock und überträgt ihm die Steuerung des Programmablaufes. Dabei werden die formalen Parameter durch aktuelle Parameter (Argumente) ersetzt.

Beispiele für Prozedur-Aufrufe:

```
zeile (m, zeichen);
zeile (n, ch);

Bisektions_Verfahren (a, b, Toleranz, Loesung);
Bisektions_Verfahren (u_Grenze, o_Grenze, epsilon, x0);
```

Merke:

- Der Aufruf enthält den Prozedurnamen und - in Klammern - die Liste der aktuellen Parameter (der Argumente). Vorläufige Form der aktuellen Parameter = Variablen.
- Beim Aufruf wird jeder aktuelle Parameter dem entsprechenden Formalparameter der Reihenfolge nach zugeordnet.
- Folge: Anzahl und Typen der einander zugeordneten aktuellen und formalen Parameter müssen übereinstimmen. Die entsprechenden Bezeichner hingegen können sich voneinander unterscheiden.

Beispiel 1 (s.o.) Ausgabe von n Zeichen

Es sind n identische Zeichen der Art ch hintereinander in einer Zeile auszugeben, z.B. '**********' bzw. ' ' (Leerzeichen).

```
Program Zeichen_Ausgabe;
var
  m, n        : integer;
  ch, zeichen : char;

  Procedure zeile (var n: integer;var ch: char);
    { n, ch - Eingangsparameter }
  var
    i : integer;  { lokaler Parameter }
  begin
    for i:=1 to n do  write(ch)
  end; { of Procedure zeile }

begin
  write('Anzahl n = '); readln(n);
  ch := '*';
  zeile (n,ch); writeln;

  m := 54; zeichen := ' '; {Leerzeichen }
  zeile (n,ch);
  zeile (m,zeichen);
  zeile (n,ch); readln
end.
```

Beispiel 2 (s.o.) Bisektionsverfahren für $f(x) = 0$

Einzugeben ist ein Intervall $[a, b]$ mit $a < b$ und eine Genauigkeitsschranke (Toleranz) tol > 0. Die Nullstelle $x \in [a, b]$ ist auszugeben.

```
Program Bisektion;
  { Funktion f(x) = sin(x) - exp(-x)  }
var
  e, f, tol, loesung : double;

  Procedure Bisektions_Verfahren
            (var a,b,epsi,x : double);
  var
    y : double;   { lokaler Parameter }
  begin
    { Prozedur-Anweisungen siehe oben
      . . . . . . . . . . . . . . . . }
  end;

begin  { Hauptprogramm }
  writeln ('Bisektionsverfahren');
  write('Intervallgrenze e = '); readln(e);
  write('Intervallgrenze f = '); readln(f);
  write('Toleranz     tol = '); readln(tol);

  Bisektions_Verfahren (e, f, tol, loesung);

  writeln ('Nullstelle = ',loesung); readln
end.
```

```
Bisektionsverfahren
===================
Intervallgrenze  e = 0
Intervallgrenze  f = 2
Toleranz       tol = 1e-5
Nullstelle  =  5.885 353 088 379E-1

Intervallgrenze  e = 0
Intervallgrenze  f = 2
Toleranz       tol = 1e-10
Nullstelle  =  5.885 327 440 337E-1

Intervallgrenze  e = 0
Intervallgrenze  f = 2
Toleranz       tol = 1e-15
Nullstelle  =  5.885 327 439 819E-1
```

3.1.3 Parameterübergabe

Parametervermittlung an obigem Beispiel:

```
Procedure
  Bisektions_Verfahren(var a, b, epsi, x :double);
                           |  |   |    |
                          1| 2|  3|   4|
                           |  |   |    |
  Bisektions_Verfahren(    e, f, tol, loesung);
```

Eine Variable kann benutzt werden, wenn sie (a) einen Namen, (b) eine Adresse und (c) einen Wert besitzt. Formale Parameter besitzen anfangs jedoch nur Namen und Adresse (z.B. a, b, epsi).

2 Mechanismen der Wertvermittlung in PASCAL:

– Variablenaufruf (Namensaufruf, Referenzaufruf)
– Wertaufruf

(a) Variablenaufruf (Referenzaufruf, call by name)

- Kennzeichen: Reserviertes Wort `var` vor dem Formalparameter a
- Beispiele (s.o.):

```
Procedure Bisektions_Verfahren
                (var a, b, epsi, x : double);

Procedure zeile (var n: integer;var ch: char);
```

- Realisierung: Übertragung der Adresse des aktuellen Parameters e in den Werteteil des formalen Parameters a $\Rightarrow$ Zugriff zum Wert von a stets (indirekt) über die Adresse, d.h. über eine Referenz
- Bezeichnung: Variablenparameter, Referenzparameter
- Konsequenzen:
  - Aktueller Parameter e muß eine Variable sein
  - Jede Veränderung des formalen Parameters a verändert auch den aktuellen Parameter e
  - Anwendung bei Ergebnisparametern obligatorisch
  - Nutzung um Speicherplatz zu sparen (große Vektoren und Matrizen)
- Schematische Darstellung:

(b) Wertaufruf (call by value)

- Kennzeichen: Das reservierte Wort `var` fehlt vor dem Formalparameter a
- Beispiele (s.o.):

```
Procedure Bisektions_Verfahren
  ( a, b, epsi : double; var x : double);
Procedure zeile ( n: integer; ch: char);
```

- Realisierung:
  - Automatische Bildung einer lokalen Hilfsvariablen (no name)
  - Berechnung des Wertes des aktuellen Parameters e und Zuweisung zur Hilfsvariablen
  - Direkte Verwendung der Hilfsvariablen (entspricht a) und ihres Wertes im Prozedurblock
- Bezeichnung: Wertparameter (hier: a, b, epsi, n, ch)
- Konsequenzen:
  - Aktueller Parameter e kann eine Variable, Konstante oder ein Ausdruck sein
  - Variablen des aufrufenden Programmes behalten ihren Wert; es findet keine wertmäßige Veränderung statt
  - Anwendung nur bei Eingangsparametern sinnvoll

Beispiele für Wertaufrufe:

```
zeile (25, '*');
readln(n); ch := ' ';  zeile(2*n+1, ch);

readln(e); readln(f);
Bisektions_Verfahren (e+3.0*f, e+exp(f), 1E-8, Loesung);
```

Beispiel 3 Zyklische Vertauschung

Die Inhalte der 3 Variablen x, y, z sind zyklisch zu vertauschen:
$x \longrightarrow y \longrightarrow z \longrightarrow x$.

```
Program Zyklisches_Vertauschen;
var
  x, y, z : double;

  Procedure Tausche (var a, b, c : double);
    { a,b,c - Eingangs- und Ausgangsparameter }
  var
    hilf : double;        { lokaler Parameter }
  begin
    hilf := c;
    c    := b;
    b    := a;
    a    := hilf
  end;

begin
  readln(x); readln(y); readln(z);
  Tausche (x,y,z);          { Variablenaufruf }
  writeln (x,y,z); readln
end.
```

Frage: Was bewirkt die Prozedur Tausche, falls a,b,c Wertparameter sind?

3.2 Funktionen in PASCAL

Eine Funktion ist in PASCAL eine Vorschrift zur Berechnung eines skalaren Wertes aus einer oder mehreren Eingangsgrößen. Der Funktionstyp muß also skalar sein!

3.2.1 Funktions–Deklaration

Die Funktions-Deklaration legt (a) die Eingangsparameter und deren Typ, (b) den Ergebnistyp und (c) den Inhalt des Algorithmus fest.

Aufbau der Funktions-Deklaration

Sie ist wie die Prozedur-Deklaration aufgebaut - mit folgenden 2 Abweichungen:

- Syntax des Funktionskopfes:

$$\begin{aligned}
\langle\text{Funktionskopf}\rangle &::= \text{function } \langle\text{Funktionsname}\rangle \\
&\quad (\langle\text{Funktionsparameter}\rangle) : \langle\text{einfacher Typ}\rangle; \\
\langle\text{Funktionsparameter}\rangle &::= \langle\text{Par-Abschnitt}\rangle \{; \langle\text{Par-Abschnitt}\rangle\} \\
\langle\text{Par-Abschnitt}\rangle &::= \text{Var } \langle\text{Variablenparameter}\rangle \mid \\
&\quad \langle\text{Wertparameter}\rangle
\end{aligned}$$

- Anweisungsteil:
  Mindestens 1 Anweisung muß dem Funktionsnamen einen Wert zuweisen, z.B. mittels Zuweisungs-Anweisung:

$$\langle\text{Funktionsname}\rangle ::= \langle\text{Ausdruck}\rangle;$$

  Der Funktionsname darf nicht wie eine übliche Variable benutzt werden; er darf nur links bei Zuweisungen stehen.

Beispiel 4 Dekadischer Logarithmus $\lg x$

Eine Funktion $\lg(x) = \ln(x)/\ln(10)$ ist zu definieren.

```
function lg ( x : double) : double;
  { x - Eingangsparameter }
begin
  lg := ln(x) / ln(10.0)
end;
```

Beispiel 5 Physikalische Kraft $F(s)$

Die Abhängigkeit einer Kraft F vom Weg s werde durch die folgende reelle Funktion beschrieben:

$$F(s) = \begin{cases} e^{-s^2} & \text{falls } |s| > 1 \\ s^2/e & \text{falls } 0 \le s \le 1 \\ s+1 & \text{falls } -1 \le s < 0 \end{cases}$$

```
function F ( s : double) : double;
  { s - Eingangsparameter }
const
  e:double = 2.718281828459;
begin
  if abs(s) > 1 then  F := exp(-sqr(s)) else
       if s >=0 then  F := sqr(s) / e
                else  F := s+1.0
end; { of F }
```

Merke: Eingangsparameter werden meist als Wertparameter behandelt. Auch Funktionen dürfen lokale Parameter enthalten. Im Deklarationsteil des Programmes stehen Funktionen wie Prozeduren.

3.2.2 Funktions–Aufruf

Standardfunktionen werden - anders als Prozeduren - innerhalb von Ausdrücken aufgerufen. Sie können als Operanden fungieren und liefern einen Wert. Dieser Mechanismus gilt auch für nutzerdefinierte Funktionen. Die Regeln für die Parameterübergabe (Variablenparameter, Wertparameter) bei Prozeduren treffen auch auf Funktionen zu.

Beispiele für Funktions-Aufrufe:

```
var
  t, x, y, s0, s1, s2, f1, f2 : double;
begin
  readln(t,s0,s1,s2);

  x := ln(t+1.0) - 2.0*lg(t+1.0);
  y := exp(2.0 * lg(t + lg(x+2.0)));
       { geschachtelter Aufruf }

  f1 := F(s1 - s0) - F(s2 - s1);
  f2 := 0.5*( F(s0) + 2.0*F(0.5(s0+s1)) + F(s1) );
```

Beispiel 6 Physikalische Arbeit $\boldsymbol{A}$

Die Abhängigkeit einer Kraft $\boldsymbol{F}$ vom Weg $\boldsymbol{s}$ werde durch die reelle Funktion $\boldsymbol{F(s)}$ aus Bsp.5 dargestellt. Zu berechnen ist die physikalische Arbeit für den Weg $[\boldsymbol{a}, \boldsymbol{b}]$ mit $\boldsymbol{a < b}$ gemäß

$$\boldsymbol{A = \int_a^b F(s)\, ds\ .}$$

Problemanalyse:

Das Integral muß i.allg. näherungsweise berechnet werden. $[a, b]$ werde in n Teilintervalle der Breite $h = (b-a)/n$ unterteilt. Die Teilintegrale $A_1, A_2, ..., A_n$ werden durch die Trapeze $T_1, T_2, ..., T_n$ angenähert. Resultat ist die zusammengesetzte Trapezformel

$$\begin{aligned} T &= T_1 + T_2 + T_3 + \cdots + T_n \\ &= h \cdot \left[\frac{1}{2}F(a) + F(a+h) + F(a+2*h) + \cdots + \frac{1}{2}F(b)\right] \end{aligned}$$

Logische Programmstruktur:

1. Hauptprogramm `'Bestimmtes_Integral'`:
   - Eingabe der Werte a, b, n
   - Aufruf der Prozedur `Trapez`
   - Ausgabe des Näherungswertes T des Integrals
   - Wiederholung der Rechnung mit neuem n
2. Prozedur `'Trapez'`:
   - Berechnung von T mit $y = F(x)$
3. Funktion `'F'`:
   - Berechnung von $F(s)$

Physische Programmstruktur:

```
Program Bestimmtes_Integral;
  uses Crt;
  var
    n       : integer;
    z       : char;
    a, b, T : double;
```

```
function F ( s : double) : double;
  { Deklaration des Integranden }
const
  e:double = 2.718281828459;
begin
  if abs(s) > 1 then  F := exp(-sqr(s))  else
       if s >=0 then  F := sqr(s) / e
                 else  F := s+1.0
end; { of F }


procedure Trapez (a, b : double; n : integer;
                  var T : double);
  { Bestimmtes Integral mit Trapezregel }
  var
    i   : integer;
    h,x,y0,y : double;
  begin
    h  := (b-a)/n; x := a;
    y0 := F(a);  y := F(b);
    T  := 0.5*(y0+y);
    for i:=1 to n-1 do
    begin
      x := a + i*h;
      y := F(x);
      T := T + y
    end;
    T := T*h
  end;  { of Trapez }
```

```
  begin   { Hauptprogramm }
    ClrScr;
    writeln('Bestimmtes Integral ueber F(s)');
    writeln('============================');
    writeln;
    write('Untere Grenze  a  = '); readln(a);
    write('Obere Grenze   b  = '); readln(b);
    repeat  writeln;
      write('Intervallzahl n = '); readln(n);

      Trapez (a,b,n,T);
      writeln('Integralwert  T  = ',T);
      write('Wiederholung (J/N)? '); readln(z)
    until  upcase(z) = 'N'
  end.
```

```
Bestimmtes Integral ueber F(s)
==============================
Untere Grenze   a = -2
Obere Grenze    b =  2

Intervallanzahl n = 500
Integralwert    T = 8.876804692E-1
Wiederholung (J/N)? j

Intervallanzahl n = 1000
Integralwert    T = 8.904079851E-1
Wiederholung (J/N)? n
```

Integral mittels Simpsonregel – Version in C++

```
#include <iostream.h>
#include <math.h>

double f(double x)  // Funktion 1
{
    return sin(x)+exp(-x);
}
double g(double x)  // Funktion 2
{
    return cos(2.0*x)-exp(-x);
}
// Berechnung von I aus der Simpsonregel zur Funktion f,
// dem Intervall [a,b] und der Schrittanzahl n
double simpson(double (f)(double), double a, double b,
          double tol=1E-8, int n=50, int k=10)
{
    if (a<=b && n>0 && !(n%2))  // Parametertest
    {
      double h, I, alt, neu = 1E15, abstand;
      int i;
      do
      {
        alt = neu;  h = (b-a)/n; I = f(a);
        for (i=1; i<n; i++)
        if (i%2)  // i ungerade
           I = I + 4*f(a+i*h);
        else      // i gerade
           I = I + 2*f(a+i*h);
        I = h/3*(I+f(b));
        neu = I;
```

```
        abstand = fabs(alt-neu);
        n += k;
      } while (abstand >= tol);
      return neu;
    }
    else
      return 0;
}

int main() {
    double a, b, tol;
    cout.precision(16);  // double !

    cout << endl << "Integrale mit Simpsonregel" << endl;
    cout << "==========================" << endl;
    cout << "Linke Intervallgrenze : ";    cin >> a;
    cout << "Rechte Intervallgrenze: ";    cin >> b;
    cout << "Genauigkeitsschranke  : ";    cin >> tol;
    cout <<  endl;

    cout << "Gegebene Toleranz : " << endl;
    cout << "Integral1 = " << simpson(f,a,b,tol) << endl;
    cout << "Integral2 = " << simpson(g,a,b,tol) << endl;
    cout << endl;

    cout << "Standard-Toleranz : " << endl;
    cout << "Integral1 = " << simpson(f,a,b) << endl;
    cout << "Integral2 = " << simpson(g,a,b) << endl;
    return 0;
}
```

3.3 Blockstruktur und globale Objekte

Ein Block ist gemäß Syntax der Hauptteil eines Programmes, einer Prozedur oder einer Funktion. In BP werden auch Units als selbständige Blöcke betrachtet.

- Externe Blöcke – selbständige, von anderen Blöcken weitgehend unabhängige Programmteile, die übersetzt werden können - und nicht in anderen Blöcken enthalten sind (BP: Program, Unit)
- Interne Blöcke – in übergeordneten (internen oder externen) Blöcken enthaltene Programmteile, die nicht selbständig übersetzt werden können (BP: Procedure, Function)

3.3.1 Blockniveau, Lebensdauer und Gültigkeitsbereich

PASCAL gestattet die Deklaration von Blöcken im Innern anderer Blöcke ("geschachtelte Prozeduren").

Blockniveau:
Es gibt die Anzahl der Programme, Units, Funktionen, Prozeduren an, in denen die jeweilige Prozedur-/Funktions-Deklaration enthalten ist, d.h. deren Schachtelungstiefe.
Blockniveaus sind 0 (externe Blöcke), 1, 2, 3,... (interne Blöcke).

Beispiel 7 **Blockniveaus** (vgl. nächste Seite)

- HP besitzt Blockniveau 0,
- UP1 und UP2 sind von Niveau 1 und
- UP2 besitzt Blockniveau 2.

```
Program  HP;
   var  j,y : integer; . . .

   Procedure UP1;
      var  i,j : integer; . . .

      Procedure UP2;
         var  y,j : integer; . . .
      begin
         j := 10;  y := 1; . . .
      end;    { of UP2 }

   begin
      UP2;
      i := 15;  j := 20;
      y := 2;   . . .
   end;   { of UP1 }

   Procedure UP3;
      var  i,j : integer; . . .
   begin
      j := 50; . . .
   end;   { of UP3 }

begin
   UP1;         { falsch: UP2; }
   j := 40;. . .
   UP3;
end.   { of HP }
```

- **Rationelle Speichernutzung:**

  Zielsetzung der Blockstrukturierung: Allen Programmobjekten (Konstanten, Variablen, Prozeduren, Funktionen etc.) ist nur solange Speicherplatz zuzuweisen, wie sie gebraucht werden!

- **Lebensdauer eines Objektes:**

  Sie definiert den Zeitraum, in dem deklarierten Bezeichnern im Speicher reale, physikalische Objekte zugewiesen sind. Bei den Compilierungszeit-Objekten unterscheidet man

  – statische (globale) Objekte: Zuweisung während der gesamten Programmdauer – in festen Datensegmenten; betrifft: Variablen, Konstanten auf Niveau 0, Funktionen und Prozeduren

  – automatische (lokale) Objekte: Zuweisung bei Eintritt in einen internen Block und ("automatischer") Verlust des Speicherplatzes bei Verlassen dieses Blockes – im Stack (Stapel, Kellerspeicher); betrifft: Variablen, Konstanten auf Niveau 1,2,3,...

  Beispiel 7 (s.o)

  – Statische (globale) Objekte: `j,y,UP1,UP2,UP3`

  – Automatische (lokale) Objekte: `i,j (in UP1);`
  `y,j (in UP2); i,j (in UP3)`

  Laufzeit-Objekte mit dynamischer Lebensdauer werden später behandelt.

- **Gültigkeitsbereich G:**

  Der Gültigkeitsbereich $\boldsymbol{G}$ eines Bezeichners ist derjenige Programmteil, in dem über den Bezeichner auf das Objekt zugegriffen werden kann. Für statische und automatische Objekte ist $\boldsymbol{G}$ genau derjenige Block $\boldsymbol{B}$, in dem der zugehörige Bezeichner deklariert ist - beginnend mit der Deklaration.

Beispiel 7 (s.o) Darstellung der Gültigkeitsbereiche:

- Statische (globale) Objekte: `j,y,UP1,UP2,P3`
- Automatische (lokale) Objekte: `i,j (in UP1); y,j (in UP2); i,j (in UP3)`

- **Sichtbarkeit:**
  Unter der Sichtbarkeit eines Bezeichners versteht man denjenigen Teil seines Gültigkeitsbereiches, von dem aus ein zulässiger Zugriff auf das Objekt möglich ist. Das Auftreten eines doppelten Bezeichners unterbricht die Sichtbarkeit, die erst wieder eintritt, wenn der Gültigkeitsbereich des doppelten Bezeichners endet.

  Beispiel 7 (s.o) Sichtbarkeit von `j,y,i`

Beispiel 7 (s.o.) Darstellung der

- Gültigkeits- und Sichtbarkeitsbereiche
- Speicherbelegung von Datensegment und Stack

3.3.2 Globale Programmobjekte

$\boldsymbol{B}$ sei der Block eines Programmes, einer Prozedur oder einer Funktion. Ein Programmobjekt $\boldsymbol{x}$ mit Gültigkeitsbereich $\boldsymbol{G}$ heißt

- lokal bezüglich $\boldsymbol{B}$, falls $\boldsymbol{G \subseteq B}$
- global bezüglich $\boldsymbol{B}$, falls $\boldsymbol{G \supset B}$

gilt. Im 2.Fall wird $\boldsymbol{x}$ außerhalb von Block $\boldsymbol{B}$ deklariert.

Beispiel 7 (s.o)

Variable y von HP ist global bzgl. UP1, UP2 und UP3 $\Rightarrow$ Anweisung `y := 2;` ist zulässig.

Empfehlungen:

- Die Benutzung globaler Objekte innerhalb von Prozeduren / Funktionen sollte in Interesse der Klarheit und Universalität weitgehend vermieden werden.
- Mitunter macht der - sparsame - Einsatz globaler Konstanten, Variablen oder Funktionen jedoch Sinn. Bei globalen Variablen sollte eine wertmäßige Veränderung im internen Block unbedingt unterbleiben.

Beispiel 8 **Masse eines inhomogenen Stabes**

Die Abhängigkeit der Dichte $\boldsymbol{\varrho(s)}$ von der Koordinate $\boldsymbol{s}$ werde durch

$$\boldsymbol{\varrho(s) = \begin{cases} \varrho_A \cdot e^{-(s-0.7)^2} & \text{falls} \quad 0 < s \leq 1.5 \\ \varrho_B + 0.11s & \text{falls} \quad 1.5 \leq s \leq 5.0 \end{cases}}$$

dargestellt.

Skizze:

Die Abhängigkeit der Dichte $\varrho(s)$ von der Koordinate s werde durch

$$\varrho(s) = \begin{cases} \varrho_A \cdot e^{-(s-0.7)^2} & \text{falls} \quad 0 < s \leq 1.5 \\ \varrho_B + 0.11s & \text{falls} \quad 1.5 \leq s \leq 5.0 \end{cases}$$

dargestellt. ϱ_A und ϱ_B sind gegebene konstante Dichten der 2 Stabkomponenten. Der Stabquerschnitt sei mit $A = 1$ als konstant angenommen.
Zu berechnen ist die Masse über $[0, L]$ mit Stablänge $L \leq 5.0$ gemäß

$$m = \int_0^L A\varrho(s)\, ds\ .$$

Problemanalyse:

1. Das Integral wird näherungsweise mittels der Trapezformel (Bsp. 6) mit n Teilintervallen berechnet.
2. ϱ_A und ϱ_B werden als globale Konstanten deklariert; die Stablänge L und die Zahl n der Teilintervalle sollen variabel sein.

Logische Programmstruktur:

1. Hauptprogramm `'Stabmasse'`:
   – Eingabe der Werte L, n
   – Aufruf der Prozedur `Trapez`
   – Ausgabe des Näherungswertes m des Integrals
   – Wiederholung der Rechnung mit neuen L, n
2. Prozedur `'Trapez'`:
   – Berechnung von T mit $y = F(x)$
3. Funktion `'F'`:
   – Berechnung der Dichtefunktion $F(s)$

Physische Programmstruktur:

```
Program Stabmasse;
  uses Crt;
  const        { Materialdichten }
    rho_A:double = 3.658;
    rho_B:double = 2.045;
  var
    n    : integer;
    z    : char;
    L, m : double;

  function F ( s : double) : double;
    { Deklaration des Integranden }
  begin
    if (0 < s) and (s< 1.5) then
      F := rho_A * exp(-sqr(s-0.7))  else
      F := rho_B + 0.11 * s
  end; { of F }

  procedure Trapez (a, b : double;
      n : integer; var T : double);
    { Bestimmtes Integral mit Trapezregel }
    { vgl. Beispiel 6                     }
      . . .
      y0 := F(a);  y := F(b);
      . . .
      y := F(x);
      . . .
  end;  { of Trapez }
```

```
begin   { Hauptprogramm }
  ClrScr;
  writeln('Masse m eines Stabes');
  writeln('====================');
  repeat  writeln;
    write  ('Stablaenge     L = '); readln(L);
    write  ('Intervallzahl  n = '); readln(n);
    Trapez (0,L,n,m);
    writeln('Gesamtmasse    m = ', m:12:6);
    write  ('Wiederholung (J/N)? '); readln(z)
  until  upcase(z) = 'N'
end.
```

```
Masse eines Stabes
==================
Stablaenge       L = 4.6
Intervallanzahl  n = 100
Gesamtmasse      m = 11.979221
Wiederholung (J/N)? j

Stablaenge       L = 4.6
Intervallanzahl  n = 200
Gesamtmasse      m = 11.978402
Wiederholung (J/N)? j

Stablaenge       L = 2.9
Intervallanzahl  n = 500
Gesamtmasse      m =  7.804488
Wiederholung (J/N)? n
```

Wertung:

(a) Die Konstanten `rho_A, rho_B` sind global bzgl. der Funktion `F`.

(b) Die Funktion `F` ist global bezüglich der Prozedur `Trapez`.

Kapitel 4

Grundlegende Datenstrukturen

Felder (array) und Zeichenketten (string) sind die häufigsten Datenstrukturen im Bereich mathematisch-naturwissenschaftlich-technischer Algorithmen. Charakteristisch ist die Festlegung der Strukur dieser Daten - einschließlich des Speicherbedarfs - während der Compilierungszeit (statische bzw. automatische Speicherklasse). Syntaktisch werden diese und andere Strukturen in PASCAL vermittels Datentypen definiert.

4.1 Typdefinitionen

Eine Datentyp-Definition legt die Menge der Werte fest, die die Variablen des entsprechenden Typs annehmen können und ordnet dem Typ einen Bezeichner (Typnamen) zu. Damit kann der Programmierer neben oder mit den eingeführten Standardtypen (integer, longint, real, double,...) neue eigene "abstrakte" Datentypen erzeugen. Dies geschieht im Deklarationsteil des Programmes, der Prozedur oder der Funktion.

Objekt	Deklarationsart	Res. Wort
Variable	Variablendeklaration	`Var`
Konstante	Konstantendeklaration	`Const`
Datentyp	Typdeklaration	`Type`
Funktion	Funktionsdeklaration	`Function`
Verfahren	Prozedurdeklaration	`Procedure`

Syntax:

⟨Typ-Deklaration⟩ ::= Type ⟨Typ-Definition⟩{; ⟨Typ-Definition⟩};

⟨Typ-Definition⟩ ::= ⟨Bezeichner⟩ = ⟨Typ⟩

⟨Typ⟩ ::= ⟨einfacher Typ⟩|⟨strukturierter Typ⟩|⟨Zeigertyp⟩

⟨einfacher Typ⟩ ::= ⟨Aufzählungstyp⟩|⟨Teilbereichstyp⟩|⟨Typbezeichner⟩

Aufzählungstypen:

Ein Aufzählungstyp definiert eine geordnete Menge von Werten durch Aufzählung der Bezeichner, die diese Werte ausdrücken. Die Definition gibt - in linearer Ordnung - alle möglichen Werte (d.h. die Konstanten) an.

Syntax: ⟨Aufzählungstyp⟩ ::= (⟨Bezeichner⟩ {, ⟨Bezeichner⟩ })

Beispiele:

```
Type   Tag         = (MO,DI,MI,DO,FR,SA,SO);
       geschlecht  = (weiblich, maennlich);
       Richtung    = (Sueden, Westen, Norden, Osten);
       Marke       = (audi,bmw,citroen,ferrari,suzuki,vw);
       Studiengang = (Maschinenbau, Mechatronik);

Var    Feiertag, Pruefungstag : Tag;
       f,g                    : geschlecht;
       Neuwagen, Rostlaube    : Marke;
       Fachrichtung           : Studiengang;

Begin  Pruefungstag := MI;
       Fachrichtung := Mechatronik;
       Neuwagen     := ferrari;
       Rostlaube    := Neuwagen;
       if Pruefungstag < FR then Feiertag := FR;
       . . .
```

Merke: Bei Aufzählungstypen sind Wertzuweisungen und Vergleiche (entsprechend der Aufzählungs-Reihenfolge) möglich. Für die skalaren Standardtypen `Char` und `Boolean` gilt:

```
(a) ... < 'A' < 'B' < ... < 'Z' < 'a' < ... < 'z' < ...
(b) Type  Boolean = (false, true);  mit  false < true
```

Teilbereichstypen:

Ein Datentyp $\boldsymbol{T}$ ist Teilbereich eines anderen, wenn die Menge seiner möglichen Werte in der Menge der möglichen Werte des umfassenderen Typs enthalten ist. Man benutzt dazu i.allg. die Teilbereichs-Notation.

Syntax: ⟨Teilbereichstyp⟩ **::=** ⟨Konstante⟩ **..** ⟨Konstante⟩

Beispiele:

```
Type  Tag          = (MO,DI,MI,DO,FR,SA,SO);
      Werktag      = MO .. FR;
      Ziffer       =  0 .. 9;        { Grundtyp: }
      Winkel       = 0 .. 360;       {  integer  }
      Grossbuchstabe = 'A' .. 'Z'; { Grundtyp: }
      Kleinbuchstabe = 'a' .. 'z'; {     char  }

Var   Feiertag     : Tag;
      Pruefungstag : Werktag;
      alpha1, psi  : Winkel;
      . . .
```

Merke: Beide Konstanten in der Definition müssen in aufsteigender Anordnung im Grundtyp stehen. Der Grundtyp muß zuvor definiert worden sein oder ist ein Standardtyp. Von float-Typen sind keine Teilbereiche möglich!

Typbezeichner:

Hiermit findet lediglich eine Neubezeichnung eines Standardtyps oder eines bereits definierten Typs statt (Synonymgebung, Alias-Name).

Beispiele:

```
Type   ganz      = integer;
       logical   = Boolean;
       float     = double;
       int1      = byte;
       int4      = longint;
       character = Char;
Var    n,k       : ganz;
       p,q,r     : logical;
       z1, z2    : character;
```

Standardfunktionen für skalare Typen:

1. `succ(x)` bestimmt den Nachfolger in der Anordnung
2. `pred(x)` bestimmt den Vorgänger in der Anordnung
3. `ord(x)` bestimmt den Ordinalwert in der Anordnung

Merke: Das 1. Element der Aufzählung hat die Platznummer 0.

Beispiele:

```
Type    Tag  = (MO,DI,MI,DO,FR,SA,SO);

succ(DO) -> FR    succ(12)  -> 13    ord(false) -> 0
pred(MI) -> DI    pred(-3)  -> -4    ord(true)  -> 1
ord (DI) -> 1     succ('B') -> 'C'
```

Die strukturierten Typen array, string, record, file werden nun behandelt; Zeigertypen dagegen erst zum späteren Zeitpunkt in Kapitel 9.

4.2 Algorithmen auf Feldern (array-Typ)

Definition 1 : Feld (array)

- Ein Feld ist eine aus einer vorher festgelegten Anzahl von Komponenten gleichen Typs bestehende Datenstruktur.
- Zugriff auf eine Komponente des Feldes erfolgt durch Angabe von Indizes.
- Die Anzahl der Indizes zur Auswahl einer Komponente gibt die Dimension des Feldes an. Diese kann beliebig groß sein.

4.2.1 Felddeklaration und -speicherung

Sie erfolgt am besten in Form einer Typdefinition im Deklarationsteil. Der Feldtyp erhält dadurch einen Bezeichner zugeordnet, der in seinem Gültigkeitsbereich zur Deklaration von Variablen und Prozedurparametern benutzt werden kann.

Syntax:

$$\begin{aligned} \langle\text{Typ-Definition}\rangle \;::=\; & \langle\text{Feldtyp}\rangle \;=\; \text{array}[\langle\text{Indextyp}\rangle\{,\langle\text{Indextyp}\rangle\}] \\ & \text{of } \langle\text{Komponententyp}\rangle \\ \langle\text{Feldtyp}\rangle \;::=\; & \langle\text{Bezeichner}\rangle \end{aligned}$$

Semantik:

- Der Indextyp muß ein Aufzählungstyp sein.
- Die Zahl der Indextypen ergibt die Dimension n des Feldtyps.
- Der Komponententyp kann beliebig sein.

Beispiele:

```
{1) Numerische Felder }
type   Zahlen = array [1 .. 20] of double;
var    a, b : Zahlen;
{2) Nicht-numerische Felder }
type   Kette = array [1..80] of char;
       Tag   = (MO,DI,MI,DO,FR,SA,SO);
       Woche = array [1..7] of Tag;
var    Text1, Text2 : Kette;
       w  : Woche;
{3) Mehrdimensionale Felder }
const  n = 20;
       m = 30;
type   Matrix = array [0..n, 1..m] of double;
       Tafel  = array [Boolean,Boolean,Boolean]
                   of Boolean;
var    A, B  :  Matrix;
       Wert  :  Tafel;
```

Speicherung eindimensionaler Felder:

Eindimensionale Felder: werden linear in Reihenfolge der Komponenten abgespeichert, indem der Indextyp alle Werte des Indexbereichs durchläuft

Beispiel:

```
type   Tag   = (MO,DI,MI,DO,FR,SA,SO);
       Woche = array [1..7] of Tag;
var    w  : Woche;
```

DI	FR	FR		MO		
1	2	3	4	5	6	7

⟸ Komponenten

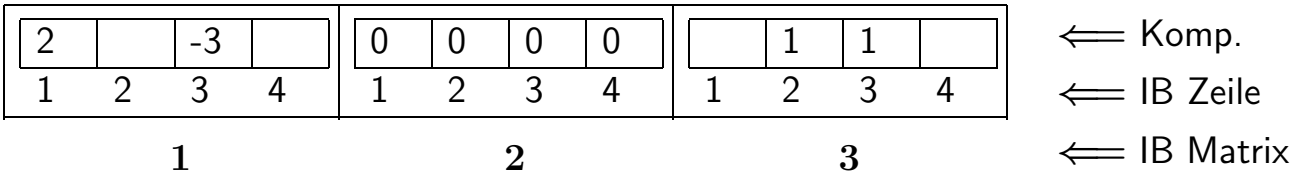
⟸ Indexbereich (IB)

Speicherung mehrdimensionaler Felder:

Ist der Komponententyp eines Feldtyps wiederum ein Feldtyp, so liegt ein mehrdimensionales Feld vor.

Beispiel:

```
type   Zeile  = array [1..4] of integer;
       Matrix = array [1..3] of Zeile;
var    M  : Matrix;
```



Äquivalent sind die Typdeklarationen

```
type  Matrix = array [1..3] of array [1..4] of integer;
type  Matrix = array [1..3, 1..4] of integer;
```

Mehrdimensionale Felder werden allgemein in lexikografischer Reihenfolge abgespeichert, d.h. die Komponenten sind linear so angeordnet, daß der letzte Index am schnellsten variiert, der erste Index dagegen am langsamsten.
Insbesondere werden Matrizen zeilenweise gespeichert!

Indizierte Variablen:

Der Zugriff auf eine Feldkomponente erfolgt durch Angabe der dafür nötigen Indizes.

Syntax: ⟨Feldvariable⟩ [⟨Indexausdruck⟩ **{,** ⟨Indexausdruck⟩**}**]

Beispiele:

```
const  n = 20; m = 30;
type   Zahlen = array [1 .. 20] of double;
       Matrix = array [0..n, 1..m] of double;
       Tensor = array [1..2, 1..2, 1..2] of single;
var    x, y   :  Zahlen;
       A, B   :  Matrix;
       V      :  Tensor;
       i,j,k  :  integer;
{ Indizierte Variablen }
  x[2],  A[1,4],  V[1,2,2],
  y[i+1],  B[i-1-k, 2*k+1],  V[i,j,k+i]
```

Äquivalent sind die Variablenzugriffe

```
(i)    V [i, j, k]
(ii)   V [i][j, k]
(iii)  V [i][j][k]
```

Felder als Prozedurparameter:

Strukturierte Datentypen dürfen nicht in Formalparameter-Abschnitten stehen. Ausweg: Deklaration eines Typbezeichners im aufrufenden Programm!

Beispiel 1 **Skalarprodukt zweier Vektoren**

Von gegebenen Vektoren $\boldsymbol{a}$ und $\boldsymbol{b}$ mit $\boldsymbol{n}$ Komponenten $(\boldsymbol{n \leq 200})$ ist das Skalarprodukt

$$\boldsymbol{s = skalar(a,b) = \sum_{k=1}^{n} a_k \cdot b_k\,, \quad a_k, b_k \in \mathbb{R}}$$

mit einer Funktion `skalar` zu berechnen.

```
Program main;
  type  Vektor = array [1..200] of double;

  function  skalar (n : integer;
              var a,b : Vektor) : double;
  var  i   : integer;
       sum : double;
  begin
    sum := 0;
    for i:= 1 to n do
      sum := sum + a[i] * b[i];
      skalar := sum
  end;  { of skalar }

  . . .
end.
```

4.2.2 Numerische Feldverarbeitung

Wegen der großen Bedeutung ein- und zweidimensionaler Zahlenfelder in der Technik werden 2 typische Grundalgorithmen - in Form von Prozeduren - betrachtet.

Beispiel 2 **Methode der kleinsten Quadrate**

Gegeben sind n Meßwerte (x_i, y_i), $i = 1(1)n$ (z.B. x - Zeit, y - Temperatur), zwischen denen ein linearer Zusammenhang vermutet wird:

$$y = f(x) = ax + b$$

Man bestimme die Koeffizienten a und b so, daß die Gesamtsumme der Fehler minimal wird!

Problemanalyse:

- Quadratsumme der Fehler

$$Q = Q(a,b) = \sum_{i=1}^{n} [f(x_i) - y_i]^2 \quad \longrightarrow \quad \text{Min!}$$

- Methode der kleinsten Quadrate (C.F.GAUSS, 1809):

$$\begin{aligned} \frac{\partial Q}{\partial b} &= 2\sum_{i=1}^{n} [ax_i + b - y_i] &= 0 \\ \frac{\partial Q}{\partial a} &= 2\sum_{i=1}^{n} [ax_i + b - y_i]x_i &= 0 \end{aligned}$$

- Normal(en)gleichungen:

$$\begin{aligned} n \cdot b + \left(\sum_{i=1}^{n} x_i\right) \cdot a &= \sum_{i=1}^{n} y_i \\ \left(\sum_{i=1}^{n} x_i\right) \cdot b + \left(\sum_{i=1}^{n} x_i^2\right) \cdot a &= \sum_{i=1}^{n} x_i y_i \end{aligned}$$

- Hilfsgrößen:

$$\begin{aligned} sx &:= \sum_{i=1}^{n} x_i, \quad sy := \sum_{i=1}^{n} y_i, \quad sxx := \sum_{i=1}^{n} x_i^2, \\ syy &:= \sum_{i=1}^{n} y_i^2, \quad sxy := \sum_{i=1}^{n} x_i y_i \end{aligned}$$

- Cramersche Regel:

$$\begin{aligned} D &:= n \cdot sxx - (sx)^2 \\ b &:= (sy \cdot sxx - sx \cdot sxy)/D \\ a &:= (n \cdot sxy - sx \cdot sy)/D \end{aligned}$$

```
{ Deklarationen im Hauptprogramm }

  const   nmax = 1000;     { max. Messwertezahl }
  type  Vektor = array [1 .. nmax] of double;

  procedure Least_squares (
                n    : integer;        { Anzahl }
            var x, y : Vektor;      { Messwerte }
            var a, b : double;  { Koeffizienten }
            var Q    : double );  {Quadratsumme }

  var  D,hx,hy,sx,sy,sxx,syy,sxy  : double;
       i                          : integer;
  begin
    sx :=0; sy :=0; sxx :=0; syy :=0; sxy :=0;
    for i:= 1 to n do
    begin
      hx := x[i];  hy := y[i];
      sx := sx + hx;
      sy := sy + hy;
      sxx := sxx +hx * hx;
      syy := syy +hy * hy;
      sxy := sxy +hx * hy
    end;
    D := n * sxx - sx * sx;
    b := (sy * sxx - sx * sxy) / D;
    a := (n * sxy  - sx * sy ) / D;
    Q := a*a*sxx + b*b*n + syy +
         (a*b*sx - a*sxy - b*sy) * 2.0
  end;  { of Least_squares }
```

Beispiel 3 Lineare Gleichungssysteme

Mechanische Schwingungsprobleme sowie die Analyse elektrischer Netzwerke erfordern die Lösung großer linearer Gleichungssysteme mit n Gleichungen für n Unbekannte $x_1, x_2, ..., x_n$.

- Allgemeine Darstellung:

$$\begin{array}{rcl} a_{11}x_1 + a_{12}x_2 + \cdots + a_{1n}x_n &=& a_1 \\ a_{21}x_1 + a_{22}x_2 + \cdots + a_{2n}x_n &=& a_2 \\ \vdots \qquad\quad \vdots \qquad\qquad\quad \vdots &\quad& \vdots \\ a_{n1}x_1 + a_{n2}x_2 + \cdots + a_{nn}x_n &=& a_n \end{array}$$

  bzw. in Matrixform $Ax = a$ mit

$$A = \begin{pmatrix} a_{11} & a_{12} & \cdots & a_{1n} \\ a_{21} & a_{22} & \cdots & a_{2n} \\ \cdots & \cdots & \cdots & \cdots \\ a_{n1} & a_{n2} & \cdots & a_{nn} \end{pmatrix}, \; a = \begin{pmatrix} a_1 \\ a_2 \\ \vdots \\ a_n \end{pmatrix}, \; x = \begin{pmatrix} x_1 \\ x_2 \\ \vdots \\ x_n \end{pmatrix}$$

- Verketteter Gaußscher Algorithmus:
  - Überführung des Systems in ein gestaffeltes Dreiecksystem mittels LR-Zerlegung in einem einzigen Arbeitsgang, der alle einzelnen Eliminationsschritte miteinander verknüpft ("verkettet")

$$\begin{array}{rcl} r_{11}x_1 + r_{12}x_2 + \cdots + r_{1n}x_n &=& r_1 \\ r_{22}x_2 + \cdots + r_{2n}x_n &=& r_2 \\ \cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots\cdots & & \\ r_{nn}x_n &=& r_n \end{array}$$

  - Falls erforderlich, erfolgen Zeilenvertauschungen. Ziel: $r_{ii} \neq 0$ für alle i (Falls A regulär ist, so ist das stets erreichbar!)
  - Rekursive Berechnung der Lösungen $x_n, x_{n-1}, ..., x_2, x_1$ aus dem gestaffelten System:

$$x_i = (r_i - x_n r_{in} - ... - x_{i+1} r_{i,i+1})/r_{ii}\,, \quad i = n(-1)1$$

Prozedurparameter:

n – Anzahl der Unbekannten $(n \leq nmax \leq 1000)$

a – Koeffizientenmatrix mit a_{ik} reell

b – Vektor der rechten Seiten b_i reell

x – Vektor der Lösungen x_i reell

```
{ Deklarationen des Hauptprogramms }

const maxn   = 100; { Max. Zahl der Unbekannten }
type  vektor = array [1..maxn] of double;
      matrix = array [1..maxn,1..maxn] of double;

procedure Gauss(
          n    : byte;   { Zahl der Unbekannten }
      var a    : matrix; { Koeffiizientenmatrix }
      var b    : vektor;        { Rechte Seiten }
      var x    : vektor;            { Loesungen }
      var sing : boolean);      { Singularitaet }

{ *** Loesung des linearen Gleichungssystems Ax=b
  mit dem verketteten GAUSSschen Algorithmus
  (LR-Zerlegung) mit Zeilenpivotisierung ****** }

label M1;         { Marke fuer singulaeren Fall }
const eps = 1e-15;         { Abbruchgenauigkeit }
var   max, s          : double;
      i, j, m, index : byte;
```

```
begin
  sing:=false;  {Annahme: Matrix A ist regulaer }

  for m:=1 to n do   { LR-Zerlegung }

  begin { Pivotsuche }
    max:=0;
    for i:=m to n do
    begin s:=a[i,m];
      for j:=1 to m-1 do s:=s+a[i,j]*a[j,m];
      a[i,m]:=s;
      if abs(s) > abs(max) then
        begin max:=s; index:=i end
    end;

    { Test auf Singularitaet}
    if abs(max) < eps then
      begin sing:=true; goto M1 end;

    if index > m then   { Zeilenvertauschung }
    begin
      for i:=1 to n do
      begin s:=a[m,i]; a[m,i]:=a[index,i];
            a[index,i]:=s
      end;
      s:=b[m]; b[m]:=b[index]; b[index]:=s
    end;

    { Bestimmung der Spaltenelemente }
    for i:=m+1 to n do a[i,m]:=-a[i,m]/max;
```

```
      { Bestimmung der Zeilenelemente }
      for i:=m+1 to n do
      begin s:=a[m,i];
        for j:=1 to m-1 do s:=s+a[m,j]*a[j,i];
        a[m,i]:=s
      end;
      s:=b[m];
      for j:=1 to m-1 do s:=s+a[m,j]*b[j];
      b[m]:=s
    end; { of LR-Zerlegung}

    { Rueckwaerts-Elimination }
    for i:=n downto 1 do
    begin s:=-b[i];
      for j:=n downto i+1 do s:=s+a[i,j]*x[j];
      x[i]:=-s/a[i,i]
    end; { of Rueckwaerts-Elimination }

    M1:
  end; { of Gauss }
```

Dynamische Felder:

Alle eingeführten arrays sind Compilierungszeit-Objekte; die Indexgrenzen müssen also bereits während der Complilierung berechenbar sein. Wird die Speicherfestlegung während der Laufzeit angestrebt, so sind dynamische Felder zu benutzen. Der Speicherplatz kann dann zur Laufzeit explizit reserviert und wieder freigegeben werden (Laufzeit-Objekte). Das eleganteste und bequemste Konzept bietet GPC mit den Schema-Typen (vgl. Kap. 4.4).

4.3 Algorithmen auf Zeichenketten (string-Typ)

Zeichenketten können als `array[ ] of char` mit fester Länge definiert werden. Um jedoch Zeichenketten variabler Länge verarbeiten zu können, existieren in BP

- Zeichenketten des Typs `string` variabler (dynamischer) Länge mit Maximallänge 255
- Null-terminierte Zeichenketten (theoretisch) beliebiger Länge.

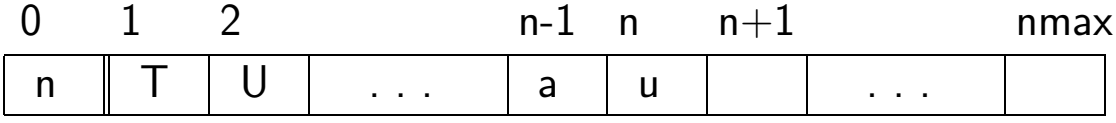
4.3.1 Zeichenketten-Definition und -Verarbeitung

Abspeicherung von Zeichenketten:

- Jede Zeichenkette besitzt eine vereinbarte maximale Länge (Zeichenzahl) $\boldsymbol{nmax}$ und eine aktuelle Länge $\boldsymbol{n}$, wobei $\boldsymbol{n}$ im Bereich $\boldsymbol{0 \leq n \leq nmax}$ variiert.
- Eine Zeichenkette belegt $\boldsymbol{nmax + 1}$ Bytes:

0. Byte	:	belegt durch aktuelle Länge (LI)
1. Byte	:	belegt durch 1. Zeichen
2. Byte	:	belegt durch 2. Zeichen
. . .	:	. . . . .
n-tes Byte	:	belegt durch n-tes Zeichen
(n+1)-tes Byte	:	unbestimmt
. . .	:	. . . . .
nmax-tes Byte	:	unbestimmt.

- Folgerung für $\boldsymbol{nmax}$: $\boldsymbol{1 \leq nmax \leq 255}$
- Skizze zur Abspeicherung:



Typ- und Variablendefinition

Syntax:

⟨Kettentyp⟩ ::= string | string [⟨Kettenlänge⟩]

⟨Kettenlänge⟩ ::= ⟨integer-Konstante⟩

Kettenkonstante ::= '' | '⟨Zeichen⟩{⟨Zeichen⟩}'

Semantik:

- Fehlt die Kettenlänge, so wird die Maximallänge 255 angenommen; andernfalls muß $\mathbf{1} \leq$ Kettenlänge $\leq \mathbf{255}$ gelten.
- In BP existiert auch die leere Zeichenkette.
- Variablendeklarationen erfolgen in üblicher Weise:

```
Var  name_1, name_2, ... , name_N : Kettentyp;
```

Beispiele:

```
{1) Vereinbarungen }
const  laenge = 80;

type   Namen  = string[25];
       infos  = string[20];
       zeile  = string[laenge];

var    Vorname, Zuname : Namen;
       Bildschirm      : zeile;
       uni, titel, ort : infos;

{2) Zuweisungen von Konstanten}
    Vorname := 'Harry';
    Zuname  := 'Potter';
    titel   := 'TU Ilmenau';
    uni     := titel;
    titel   := '';
```

Zugriff auf einzelne Zeichen:

Die Zeichen einer Kette sind mittels indizierter Variablen

⟨Name⟩[⟨Indexausdruck⟩]

erreichbar. Das Ergebnis kann einer Variablen des Typs `char` zugewiesen werden. Die Komponente 0 liefert den Längenindikator (LI) - als Zeichen des Typs `char`.

Beispiele:

```
var  ch : char;
     n : integer;
begin
  uni   := 'TU Ilmenau';
  titel := '';
  ch    :=  uni[2];          --> 'U'
  ch    :=  uni[0];          --> Steuerzeichen
  n     :=  ord(uni[0]);     --> 10
  n     :=  ord(titel[0]);   -->  0
```

Kettenausdrücke und - anweisungen

1. Kettenzuweisung:

   ⟨Kettenvariable⟩ := ⟨Kettenausdruck⟩

   Einfachste Ausdrücke sind Kettenkonstanten und -variablen. Ist der LI des Ausdruckes größer als die Maximallänge der Kettenvariablen, so wird rechts abgeschnitten.
2. Verkettung (Konkatenation):
   Durch den Operator + werden 2 Ketten aneinandergehängt (verkettet). Dabei sind die aktuellen Längen n_1, n_2 der beteiligten Ketten maßgeblich! $n = n_1 + n_2$ ist die Länge der neuen Zeichenkette.

Beispiele:

```
{1) Vereinbarungen }
type   string20 = string[20];
       string15 = string[15];
var    name, titel : string20;
       ort         : string15;
       adresse     : string;

{2) Zuweisungen von Konstanten}
    name  := 'TU Ilmenau, Ehrenberg';
    ort   := name;  { --> Informationsverlust! }

{3) Verkettung}
    name  := 'TU Ilmenau, Ehrenberg';
    ort   := '98684 Ilmenau';
    adresse := name + ort;    { --> Abstaende! }
    adresse := name + ', ' + ort;
```

Zeichenketten-Vergleiche:

- Die Priorität der Relationsoperatoren $=, <>, <, >, <=, >=$ ist geringer als die der Verkettung $+$.
- Ketten werden zeichenweise von links nach rechts gemäß ASCII-Code verglichen. Sie sind nur dann gleich, wenn Inhalt und aktuelle Länge übereinstimmen.
- Haben 2 Ketten verschiedene Länge, stimmen jedoch bis zum letzten zeichen der kürzeren Kette überein, so ist die kürzere Kette kleiner.
- Der Ergebniswert ist vom Typ `Boolean`.
- Beispiel: $<$ und $\leq$ gilt für die Anordnung

```
Mai < Maier < Mayer      < Meier     < Meier1  <
    < Meyer < Meyer-Hoff < MeyerHoff < Meyerhoff
```

Beispiel 4 Schnelles Sortieren von Namen

Nach kompletter Eingabe von maximal 1000 Namen sind diese lexikografisch zu sortieren und anschließend in sortierter Reihenfolge auszugeben.

```
Program Sortierung_von_Namen_durch_Partitionierung;
  uses  crt;
  const maxlaenge = 25;        { max. Namenslaenge }
        maxanzahl = 1000;      { max. Namensanzahl }
  type  element   = string[maxlaenge];
        vektor    = array[1..maxanzahl] of element;
  var   i,n : integer;   a : vektor;

  procedure QUICKSORT(var a:vektor; l,r:integer);
    { Sortieren der Elemente  a[l],a[l+1],...,a[r]
      in aufsteigender Anordnung }
    var  i,j : integer;   g : element;
    begin
      i := l;  j := r;  g := a[i];
      { Divide ! ...  -> Teilen des Feldes }
      repeat
        while (a[j] >= g) and (i < j) do  dec(j);
        a[i] := a[j];
        while (a[i] <= g) and (i < j) do  inc(i);
        a[j] := a[i]
      until i = j;
      a[i] := g;
      { ... et impera !  -> Sortieren der Teile }
      if l < i-1  then  QUICKSORT(a,l,i-1);
      if i+1 < r  then  QUICKSORT(a,i+1,r)
    end;          { of QUICKSORT }
```

```
begin          { Hauptprogramm }
  ClrScr;
  writeln('Sortieren mit Quicksort');
  writeln('======================');
  writeln;
  write('Anzahl  = '); readln(n); writeln;
  writeln('Unsortierte Folge :');
  for i := 1 to n do
  begin
    write('Name[',i,']  =  '); readln(a[i])
  end;
  writeln;
  QUICKSORT(a,1,n);
  writeln('Sortierte Folge :');
  for i := 1 to n do
    writeln('Name[',i,']  =  ',a[i]);
  readln
end.
```

4.3.2 Zeichenketten-Prozeduren

Zur Vereinfachung der Zeichenketten-Verarbeitung stehen in BP folgende Prozeduren zur Verfügung.

- Löschen in einer Zeichenkette

  Format : `Delete (St, Pos, Anzahl);`

  Parameter : St – Bezeichner einer Kettenvariablen
  Pos – integer-Ausdruck
  Anzahl – integer-Ausdruck

  Wirkung : Aus Kette St werden Anzahl Zeichen gelöscht, beginnend mit Position Pos.

- Einfügen in eine Zeichenkette

  Format : `Insert (Objekt, St, Pos);`

  Parameter : Objekt – Kettenausdruck
  St – Kettenvariable
  Pos – integer-Ausdruck

  Wirkung : Einfügen der Zeichenkette Objekt in die Kette St ab Position Pos.

- Umwandlung in eine Zeichenkette

  Format : `Str (Par, St);`

  Parameter : Par – Schreibparameter in 3 Formen:
  (a) Ausdruck
  (a) Ausdruck : Format1
  (a) Ausdruck : Format1 : Format2
  St – Kettenvariable

  Wirkung : Umwandlung des numerischen Wertes Par in eine Zeichenkette mit Bezeichner St. Str wirkt wie write, das Ergebnis wird St zugeordnet.

- Umwandlung in einen numerischen Wert

  Format : `Val (St, Var, Code);`

  Parameter : St – Zeichenkettenausdruck, der einen numerischen Wert ausdrückt
  Var – integer- oder real-Variable
  Code – integer-Variable

  Wirkung : Umwandlung des Kettenausdruckes St in einen real- oder integer-Wert und Zuordnung zu Var. Code $= \mathbf{0}$, falls die Umwandlung erfolgreich war, andernfalls gibt Code die Position des ersten fehlerhaften Zeichens aus St an.

Beispiel 5 **Zahleneingabe mit Fehlerkontrolle**

Beim Einlesen von double-Zahlen sind Eingabefehler (syntaktisch) festzustellen.

```
procedure double_input (var a : double);
   { Eingabe von double-Zahlen mit Syntaxtest }
  var  zk : string[40];
       c  : integer;
begin
  readln(zk);    { Eingabe als Zeichenkette }
  val (zk, a, c);  { Konvertierungs-Versuch }
  while  c <> 0  do        { Wiederholungen }
  begin
    write('Syntaxfehler! Neue Eingabe : ');
    readln(zk);
    val (zk, a, c);
  end;     { of while }
end;   { of double_input }
```

4.3.3 Zeichenketten-Funktionen

Wie auch Standardfunktionen dürfen sie – anders als die Zeichenketten-Prozeduren – direkt in Ausdrücken benutzt werden.

- Länge einer Zeichenkette

  Format : `Length (St)`
  Parameter : St – Zeichenkettenausdruck
  Wirkung : Aktuelle Länge des Kettenausdruckes St.
  Ergebnistyp : integer

- Erzeugung einer Teilkette

  Format : `Copy (St, Pos, Anzahl)`
  Parameter : St – Zeichenkettenausdruck
  Pos, Anzahl – integer-Ausdrücke
  Wirkung : Bildung einer Teilkette aus St mit Anzahl Zeichen, beginnend mit der Position Pos.
  Ergebnistyp : string-Typ

- Suchen einer Teilkette

  Format : `Pos (Objekt, St)`
  Parameter : Objekt, St – Zeichenkettenausdrücke
  Wirkung : Durchsuchen der Kette St nach dem ersten Vorkommen des Ausdruckes Objekt. Pos liefert diejenige Position in St, die das erste Zeichen von Objekt innehat (Andernfalls liefert Pos: 0).
  Ergebnistyp : integer

Beispiel 6 Textanalyse

Es ist festzustellen, wie oft eine bestimmte Zeichenfolge z der maximalen Länge 50 in einem gegebenen Text zk enthalten ist.

Beispiele:

```
zk := 'Pfeiffer';  z := 'f';   -->  3-mal
zk := 'Begin  x := 2.3; y := x*x;
        zk := ''Pfeiffer'';  z := ''f''
      End.';     z := ':=';  -->   4-mal
```

```
{ Deklarationen im aufrufenden Programm }
type  string50 = string[50];

function Anzahl ( zk : string; { Gegebene Kette }
                  z  : string50)    { Suchkette }
                     : integer;        { Anzahl }
var  a, p, l : integer;
     y       : string;
begin
  l := length (z) -1;              { Hilfsgroesse }
  a := 0;                  { Zaehler fuer Anzahl }
  y := zk;           { Kopie der gegebenen Kettte }
  p := Pos(z,y);         { 1.Position von z in y }
  while  p <> 0  do begin
    inc(a);
    delete(y,1,p+l);                { Abschneiden }
    p := Pos(z,y)        { 1.Position von z in y }
  end;  {of while }
  Anzahl := a
end;  {of Anzahl }
```

4.4 Schema-Typen in GNU Pascal (GPC)

Schema–Typen sind als Erweiterung im Extended Pascal (ISO 10206) auch in GNU Pascal implementiert. Sie sind in der Regel ein- und mehrdimensionale Felder, die von „Diskriminanten“ abhängen. Diesen Variablen können

- *während der Compilierung* oder
- *während der Laufzeit*

Werte zugeordnet werden.

Beispiele mit festen Diskriminantenwerten:

```
type
  vektor(n   : integer)  = array[1..n] of double;
  kette(maxi : integer)  = array[1..maxi] of char;
  matrix(m,n : integer)  = array[1..m,1..n] of double;
{ Diskriminanten sind n, maxi, m,n }

var  { 3 Schema-Variablen }
  x : vektor(450);
  a : matrix(1000,1000);
  W : kette(200000);
```

Man beachte:

- Schema-Typen sind dynamisch, d.h. sie werden im Freispeicher (Heap) angelegt. Die gesamte Verwaltung (Allozierung, Disallozierung, Zugriff über Zeiger etc.) erfolgt automatisiert.
- Gültigkeitsbereich der Objekte ist der aktuelle Block, beginnend bei der Definition.
- Die Dimensionen werden nur durch die Größe des verfügbaren Freispeichers begrenzt.

- Zugriff auf Schemakomponenten erfolgt wie bei arrays über die Indizes in [ ]
- Schema-Variablen „kennen“ ihre Diskriminanten; Zugriff kann über die Recordkomponente `<name>.<diskriminante>` erfolgen.

Beispiele (s.o.):

```
writeln (x.n, W.maxi);  { liefert 450 und 200000 }
readln ( W[123456] );   { Eingabe eines Zeichens }
for i:= 1 to a.m do  a[i,i] := 3.5*i;
```

Variable Diskriminantenwerte

Schema-Typen können in inneren Blöcken (Prozeduren, Funktionen) definiert werden. Dann dürfen die benötigten *variablen* Diskriminantenwerte als Parameter übergeben werden.

Beispiel 8 **(s.o.) Skalarprodukt zweier Vektoren**

Von gegebenen Vektoren $\boldsymbol{x}$ und $\boldsymbol{y}$ mit $\boldsymbol{n}$ Komponenten ist das Skalarprodukt $\boldsymbol{s = x \cdot y}$ zu berechnen. Die Größe von $\boldsymbol{n}$ soll beliebig sein und während des Laufes eingelesen werden!

```
Program schema1;  { Skalarprodukt mit Schema-Typ }
var   m : integer;

procedure main (n : integer);   { n - Dimension }
type
  vektor(n: integer) = array[1..n] of double;
var
  x,y : vektor(n);            { Schema-Variablen }
  s   : double;
  i   : integer;
```

```
begin
  for i:= 1 to n do
  begin
    x[i] := 1.0; y[i] := i;
  end;
  s := 0;
  for i:= 1 to n do
    s := s + x[i] * y[i];
  writeln('Ergebnis  s = ',s:20:0);
  readln
end;

begin
  writeln; writeln;
  writeln('Skalarprodukt zweier n-Vektoren');
  writeln('================================');
  write('Dimension n = '); readln(m);
  main(m);           { "Hauptprozedur" main }
end.
```

```
  Skalarprodukt zweier n-Vektoren
  ================================
  Dimension n = 100000
  Ergebnis  s = 5000050000

  Skalarprodukt zweier n-Vektoren
  ================================
  Dimension n = 1000000
  Ergebnis  s = 500000500000
```

Schema-Typen können mehrdimensional sein. Damit lassen sich großdimensionale Matrizen und Vektoren bequem verarbeiten. Die Größen der Dimensionen m, n dürfen während der Laufzeit eingelesen werden!

Beispiel 9 **Matrix * Vektor**

Das Produkt $x = A * b$ einer $(n, n)-$Matrix A mit einem $n-$Vektor b ist zu bilden und auszugeben. Die Größe von n soll beliebig sein und während des Laufes eingelesen werden!

```
Program schema2;  { Matrix*Vektor mit Schema-Typen }
var   n : integer;

procedure main (m,n : integer);
type
  matrix(m,n: integer) = array[1..m,1..n] of double;
  vektor(n: integer)   = array[1..n] of double;
var
  a   : matrix(n,n);
  b,x : vektor(n);
  s   : double;
  i,k : integer;

  procedure Eingabe(var a: matrix; var b:vektor);
  var i,k : integer;
  begin
    for i:= 1 to b.n do begin
      for k:=1 to b.n do
        a[i,k] := i+k;
      b[i] := i*i;
    end
  end;
```

```
  procedure Ausgabe(var x: vektor);
  var i : integer;
  begin
    for i:= 1 to x.n do
      writeln('x[',i,'] = ', x[i]);
    readln
  end;

  procedure Produkt(var a:matrix; var b:vektor;
                    var x:vektor);
  var
    i,k : integer;
    s   : double;
  begin
    for i:=1 to b.n do
    begin
      s  := 0;
      for k:=1 to b.n do
        s := s + a[i,k] * b[k];
      x[i] := s
    end
  end;   { of Produkt }

begin  { of main }
  writeln;
  Eingabe(a,b);
  Produkt(a,b,x);
  {  Ausgabe(x); }
  writeln('Dimension n = ',x.n); readln
end;
```

```
begin  { of Program }
  writeln; writeln;
  writeln('Matrix * Vektor mit Schema-Typen');
  writeln('================================');
  write('Dimension n = ');
  readln(n);
  main(n,n);
end.
```

Der Schema-Typ String

Der Typ `String` in GPC ist ein vordefinierter (built-in) Schema-Typ mit der inneren Struktur

```
String (Capacity : Cardinal) = record
  Length : 0 .. Capacity;
  Chars  : packed array [1 .. Capacity + 1] of Char
end;
```

GPC-Strings können praktisch *beliebige Länge* besitzen. Es gilt nicht die 255-Zeichen-Grenze. Man beachte insbesondere:

- Die maximale Länge einer String-Variablen z kann mit `z.Capacity` erhalten werden. Die aktuelle Länge von z kann mittels der Funktion `Length(z)` abgefragt werden.
- Die BP-Funktionen und -Prozeduren

  `Length, Pos, Str, Val, Copy, Insert, Delete`

  sind anwendbar.
- Variablen-Parameter des Types `String` ohne Diskriminante können für Schema-Typen benutzt werden. Ansonsten wird die BP-Typvereinbarung `String` als `String(255)` ausgewertet.

Beispiel 10 (s.o.) Textanalyse

Es ist festzustellen, wie oft eine bestimmte Zeichenfolge z der maximalen Länge 500 in einem gegebenen Text zk enthalten ist. Dessen Maximallänge `maxi` soll beliebig sein und erst während des Laufes eingelesen werden.
Hinweis: In GPC dürfen Deklarationen auch im Anweisungsteil eines Programmes stehen.

```
Program Anzahl1;           { String-Schema in GPC }
type
  string500 = string(500);           { Suchkette }
var
  maxi : Integer;
  z    : string500;

function Anzahl (var zk : string;    { Geg. Text }
                 var z  : string500) { Suchkette }
                        : integer;      { Anzahl }
var  a, p, l : integer;
begin
  l := length(z)-1;               { Hilfsgroesse }
  a := 0;                  { Zaehler fuer Anzahl }
  p := Pos(z,zk);       { 1.Position von z in zk }
  while  p <> 0  do begin
    inc(a);
    delete(zk,1,p+l);              { Abschneiden }
    p := Pos(z,zk)      { 1.Position von z in zk }
  end;  {of while }
  Anzahl := a
end;  {of Anzahl }
```

```
begin
  writeln;
  repeat
    write ('Maximallaenge des Textes : ');
    readln (maxi)
  until maxi >= 1;

  var  zk: string(maxi);  { ... in GPC zulaessig! }

  write('Eingabe des Textes zk    : '); readln(zk);
  write('Eingabe der Suchkette z  : '); readln(z);
  writeln('Laenge von zk = ', length(zk));
  writeln('Laenge von z  = ', length(z));
  writeln('Anzahl des Auftretens    : ',
          Anzahl(zk,z));
  readln
end.
```

Hinweis:

GPC ist der freie 32/64-bit Compiler der GNU Compiler-Sammlung.
Er unterstützt (Version 20010409)

- ISO-7185 Standard Pascal
- große Teile von ISO-10206 Extended Pascal
- Borland Pascal 7.0 (mit einigen Delphi-Erweiterungen)
- Teile von Pascal-SC (PXSC).

Vgl. *The GNU Pascal Manual, Free Software Foundation.* Alle Informationen und Downloads unter

www.gnu-pascal.de

********** Ende des Script-Teils 1 von 3 **********