dietz 621

Full-PASCAL

Benutzeranleitung

dietz 621

Full-PASCAL

Benutzeranleitung

Heinrich Dietz
Solinger Straße 9
4330 Mülheim-Ruhr
Tel.: (0208) 4434-1
Telex 856770

DIETZ Computer SYSTEME

2-8011-01-181 Schutzgebühr DM 72,50

DIETZ Computer SYSTEME

1.	Einleitung	2
1.1.	Syntax-Diagramme	4
2.	Lexikalische Symbole	7
2.1.	Zeichenvorrat (character set)	7
2.1.1.	Schlüsselworte (key-words)	8
2.1.2.	Namen (identifier)	9
2.1.3.	Zahlen	10
2.2.	Zeichenketten (strings)	14
2.3.	Zwischenräume, Kommentare, Zeilenende	15
3.	Vereinbarungsteil	16
3.1.	Blockkonzept	16
3.1.1.	Gültigkeitsbereiche von Namen und Marken	17
3.1.2.	Aufbau des Vereinbarungsteils	18
3.2.	Marken - Vereinbarungen	19
3.3.	Konstanten-Definitionen	20
3.4.	Typ-Definitionen	23
3.4.1.	Einfache Typen (simple type)	24
3.4.1.1.	Standardtypen (standard simple-types)	24
3.4.1.2.	Aufzählungstyp (Enumerated-Types)	27
3.4.1.3.	Teilbereichstypen (subrange-type)	28
3.4.2.	Strukturierte Typen (structured type)	29
3.4.2.1.	Bereichs-Typen (array type) 1*	29
3.4.2.2.	Satz-Typ (record-type)	32
3.4.2.3.	Mengentyp (set type)	37
3.4.2.4.	Datei-Typ (file type)	40
3.4.3.	Zeiger-Typ (pointer-type)	42
3.4.4.	Zusammenfassendes Typen-Diagramm	43
3.4.5.	Zuordnungsverträglichkeit (assignment - compability)	44
3.5.	Deklaration von Variablen	46
3.5.1.	Einheitsvariable (entire variable)	47
3.5.2.	Komponentenvariable (component-variable)	49
3.5.2.1.	Indizierte Variable (indexed variable)	50
3.5.2.2.	Satzfeldbezeichner (field-designator)	51
3.5.2.3.	Puffervariable (file-buffer)	52
3.5.3.	Zeiger-Variable (pointer variable) und Referenzierte Variable (referenced variable)	53
3.6.	Prozedur- und Funktions-Deklarationen	57
3.6.1.	Prozedur-Deklarationen	57
3.6.2.	Funktions-Deklarationen	60
3.6.3.	Parameter	61
3.6.3.1.	Wertparameter (value-parameter)	62
3.6.3.2.	Variablenparameter (variable parameter)	64
3.6.3.3.	Prozedur-Parameter	65
3.6.3.4.	Funktions-Parameter	67
3.6.3.5.	Gegenüberstellung der Parameterübergabe-Formen	69
3.6.4.	Standard-Prozeduren	70
3.6.4.1.	Prozeduren zur Dateiverarbeitung (file - handling - procedures)	70
3.6.4.2.	Prozeduren zur Erzeugung dynamischer Variablen	78
3.6.4.3.	Standardfunktionen	85
3.6.4.4.	Arithmetische Funktionen	85
3.6.4.5.	Funktionen für Ganzzahlen	86
3.6.4.6.	Prädikate (predicates)	89
3.6.4.7.	Initialisierungsprozeduren	94

4.	Ausführungsteil (statement part)	96
4.1.	Ausdrücke (expression)	97
4.1.1.	Operanden	97
4.1.2.	Operatoren	100
4.1.2.1.	Logische Umkehrung (NOT)	101
4.1.2.2.	Multiplikationsoperatoren (multiplying operator)	101
4.1.2.3.	Additionsoperatoren (adding-operator)	103
4.1.2.4.	Vergleichsoperatoren (Relationale Operatoren)	106
4.1.2.5.	Präzedenzklassen der Operatoren	109
4.1.3.	Bildung von Ausdrücken	110
4.1.3.1.	Multiplikationsausdruck (Term)	110
4.1.3.2.	Additionsausdruck (einfacher Ausdruck)	111
4.1.3.3.	Ausdruck und Ausdrucksverbindungen (Element, Menge)	112
4.1.3.4.	Faktor	115
4.2.	Anweisungen (statement)	119
4.2.1.	Einfache Anweisungen (simple statement)	121
4.2.1.1.	Wertzuweisung oder Ergibtanweisung (assignment statement)	122
4.2.1.2.	Prozeduranweisung (procedure statement)	123
4.2.1.3.	Sprunganweisung (goto statement)	124
4.2.2.	Strukturierte Anweisungen (structured statements)	126
4.2.2.1.	Verbundanweisung (compound statement)	127
4.2.2.2.	Bedingte Anweisung (conditional statement)	128
4.2.2.3.	Wiederholungs- oder Zyklusanweisung (repetitive statement)	134
4.2.2.3.1.	WHILE-Anweisung	135
4.2.2.3.2.	REPEAT-Anweisung	137
4.2.2.3.3.	FOR-Anweisung	138
4.2.2.3.4.	LOOP-Anweisung	141
4.2.2.4.	WITH-Anweisung	144
5.	Externe Daten und Prozeduren	145
5.1.	Allgemeines	145
5.2.	Schnittstellenbeschreibung	146
5.3.	Externe Module	150
6.	Ein-Ausgabe-Prozeduren (Input - Output - Procedures)	153
6.1.	Eingabe-Prozedur (Input-Procedure)	153
6.2.	Ausgabe-Prozedur (Output-Procedure)	157

1. Einleitung

Die folgende Beschreibung definiert den Sprachumfang des DIETZ 621 - PASCAL - Compilers. Als Vorlage diente ein ANSI - Normvorschlag 1*. Der implementierte Sprachumfang entspricht der Definition der überarbeiteten Auflage (Revised Report) 2*. Auf Erweiterungen und Einschränkungen dieser im folgenden als Standard-Pascal bezeichneten Version wird im Text verwiesen.

Professor Niklaus Wirth hat PASCAL an der eidgenössischen Technischen Hochschule in Zürich auf Basis von ALGOL 60 entwickelt.

Bei PASCAL handelt es sich um eine Programmiersprache, deren Syntax und Semantik formal definiert sind. Hierbei wurde auf kurzgehaltene und präzise Ausdrucksformen Wert gelegt, ohne daß die Leistungsfähigkeit im Hinblick auf den Programmablauf leidet.

Im einzelnen trägt PASCAL folgenden Zielen Rechnung:

Klarheit und Eindeutigkeit

Die Anlehnung an mathematische Ausdrucksmittel präzisiert die Darstellung und Bedeutung von Vereinbarungen, Sprachanweisungen und Verknüpfungsregeln. Das betrifft sowohl die Folgerichtigkeit des Aufbaus (Logik) wie die bedeutungserhaltende Knappheit der Darstellung (geringe Redundanz).

1* ANSI (American National Standard Institute)
Working Draft Specification for PASCAL, Report No. 97/5 N462
New York, February 1979

2* Jensen, K.; Wirth, N.
PASCAL User Manual an Report
Springer Lecture Notes in Computer Science, Vol. 18,
Berlin, Heidelberg, New York 1974

Anwendungsbreite

Die Problemorientierung von PASCAL leitet sich nicht aus der Anlehnung an methodischen Problemkomplexen (z.B. rein kommerziell, technisch u.a.) sondern aus der Anlehnung an die Arbeitsweise aktueller Datenverarbeitungsanlagen ab.

Leistungsfähigkeit

Es wurde weitgehend auf Belastungen verzichtet, die während der Programmlaufzeit entstehen können. Das betrifft die Einführung bestimmter zu verarbeitender Daten und den Aufruf von Unterprogrammen und Funktionen, die während des Ablaufs zu interpretieren wären. Das vermeidet ablaufinterne Wartezeiten und steigert die zeitbezogene Wirksamkeit des Programmes.

Zuverlässigkeit

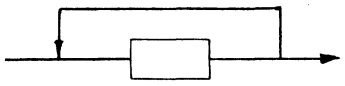
Mit der formalen Eindeutigkeit und den strengen Übersetzungsanforderungen vermeidet man Ausnahmen, die durch mißverständliche Sprachregelungen Fehlerquellen in sich bergen können. Der Zwang zu einem strukturierten Programmaufbau erfordert eine aufwendigere Vorbereitung, erhöht aber die Zuverlässigkeit beim späteren Programmeinsatz.

Lehr-, lernmethodische Eignung

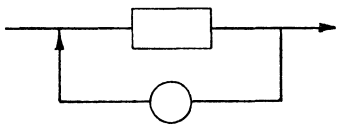
PASCAL gewinnt als didaktisches Hilfsmittel in Ausbildungsinstitutionen große Bedeutung, weil die Notwendigkeit in methodisch exaktem Vorgehen den Lehrenden und den Lernenden zu einer konsequenteren Denkweise zwingt. Die Darstellung in Struktogrammen ist dabei ein Mittel, die Programmplanung im Sinne "strukturierter Programmierung" zu unterstützen.

1.1. Syntax-Diagramme

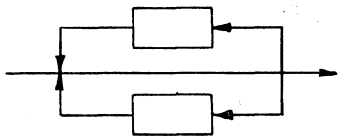
Folgende Regeln sind für das Verständnis festzuhalten:



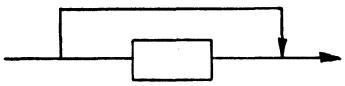
Zurückweisende Pfeile bedeuten, daß mehrere der jeweiligen Rechteckbezeichnung entsprechende Ausdrücke nacheinander aufgeführt werden können.



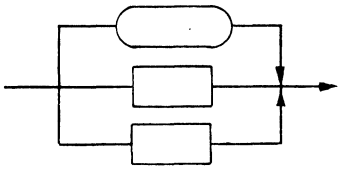
Solche Wiederholungen sind ggf. durch die im Kreis vorgeschriebenen Zeichen (Komma, Semikolon, Vorzeichen usw.) zu trennen.



In ähnlicher Weise lassen sich unterschiedliche Zeichen- und Symbolfolgen veranschaulichen.

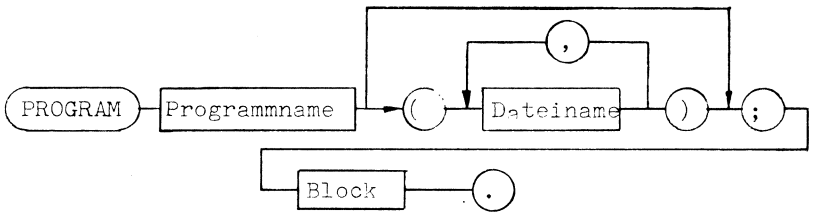


Überspringende Pfeile bedeuten, daß der im Rechteck genannte Ausdruck entfallen kann.



Weiterhin lassen sich damit alternative Ausdrücke darstellen.

Der Aufbau von PASCAL-Programmen wird hier durch die nachfolgende Form von Syntax - Diagrammen beschrieben.



Alle Worte bzw. Zeichen in den Kreisen und Ovalen sind in der angegebenen Form endgültig. Die Angaben in den Rechtecken müssen vom Programmierer festgelegt werden.
Dazu gehören "Programmname", "Dateiname" und "Block".

Im Gegensatz zu Namen (vgl. 2.1.2) im allgemeinen darf der Programmname nur aus Buchstaben bestehen.

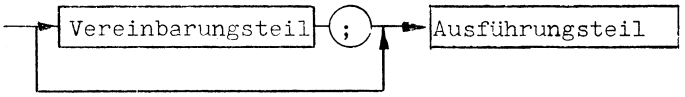
Unsere Implementierung gestattet das Fehlen von Dateinamen. (Bei Standard-PASCAL muß mindestens ein Dateiname vorhanden sein). Der zurückweisende Pfeil bedeutet die Möglichkeit, mehrere Dateinamen - jeweils durch "Komma" getrennt - zu nennen.

An dieser Stelle muß bei Verwendung der Standardprozedur für die Ausgabe (WRITE) der Dateiname OUTPUT; bzw. bei Verwendung der Standardprozedur für die Eingabe (READ) der Dateiname INPUT eingesetzt werden. Außerdem können vom Benutzer definierte Dateinamen eingesetzt werden.

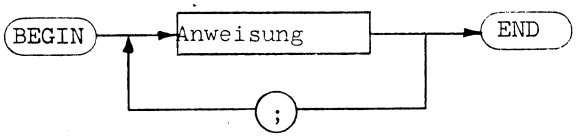
Ein Block besteht aus einem Vereinbarungsteil und einem Ausführungsteil. Im Vereinbarungsteil müssen alle Objekte mit Ausnahme von Konstanten (Marken, Typen, Variable, Funktionen und Prozeduren) dem Compiler bekanntgegeben werden.

Der Ausführungsteil beginnt mit dem Schlüsselwort BEGIN und enthält Anweisungen, die zur Laufzeit des Programms ausgeführt werden. Der Ausführungsteil endet mit dem Schlüsselwort END. Das gesamte Programm endet mit einem Punkt.

´Block´



'Ausführungsteil'



Aus dem Syntax-Diagramm erkennt man, daß der Vereinbarungsteil fehlen darf (überspringender Pfeil). Im Ausführungsteil können dann bei allen Anweisungen nur konstante Objekte verwendet werden.

In der Folge werden alle wesentlichen Programmbestandteile zur besseren Übersicht als Syntaxdiagramme bzw. Sprachelemente (vgl. 2 lexikalische Symbole) dargestellt. Das betrifft u.a. die Begriffe BEGIN und END (2.1.1 Schlüsselworte) genauso wie Programmname und Dateiname (2.1.2 Namen).

2. Lexikalische Symbole

Alle in PASCAL-Programmen verwendbaren Symbole bezeichnet man als lexikalische Symbole. Alternative Symbole werden im folgenden durch "¦" getrennt.

2.1. Zeichenvorrat (character set)

Für unterschiedliche Symbole gibt es einen zur Darstellung zugelassenen Zeichenvorrat.

Buchstaben: = (letter)

A¦ B¦ C¦ D¦ E¦ F¦ G¦ H¦ I¦ J¦ K¦
L¦ M¦ N¦ O¦ P¦ Q¦ R¦ S¦ T¦ U¦ V¦
W¦ X¦ Y¦ Z
a¦ b¦ c¦ d¦ e¦ f¦ g¦ h¦ i¦ j¦ k¦
l¦ m¦ n¦ o¦ p¦ q¦ r¦ s¦ t¦ u¦ v¦
w¦ x¦ y¦ z

Kleinbuchstaben in Namen werden von den Großbuchstaben nicht unterschieden. Der Compiler unterscheidet jedoch nicht zwischen kleinen und großen Buchstaben.

Ziffern: = (digit)

1¦ 2¦ 3¦ 4¦ 5¦ 6¦ 7¦ 8¦ 9¦ 0

Sonderzeichen: =

+¦ -¦ *¦ /¦ =¦ <¦ >¦ ¦ ¦ (¦) ¦
{¦ } ¦
.¦ ,¦ :¦ ;¦ ^¦ '¦ <space>
!¦ "¦ #¦ $¦ %¦ &¦ \

2.1.1. Schlüsselworte (key-words)

Schlüsselworte sind vorgeschriebene Buchstabenfolgen, die eine bestimmte Bedeutung für den Programmablauf haben (Indikatoren). Zusammen mit den syntaktischen Regeln bestimmt ihr semantischer Gehalt das Leistungsvermögen von PASCAL.

Folgende Schlüsselworte sind definiert:

Schlüsselworte: =

AND, ARRAY, BEGIN, CASE, CONST, DIV,
DOWNTO, DO, ELSE, END, FILE, FOR, FORWARD,
FUNCTION, GOTO, IF, IN, LABEL, MOD, NIL,
NOT, OF, OR, PACKED, PROCEDURE, PROGRAM,
RECORD, REPEAT, SET, THEN, TO, TYPE,
UNTIL, VAR, WHILE, WITH

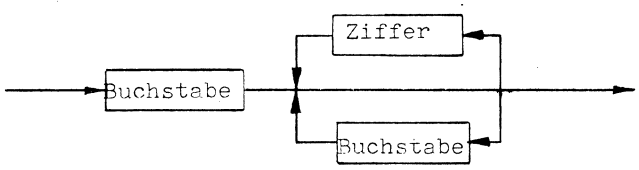
Folgende Bezeichnungen gehören noch zusätzlich zum Sprachumfang von Full-PASCAL:

EXPORTS, EXTERN, IMPORTS,
MODULE,
LOOP, EXIT, OTHERS, INITPROCEDURE

2.1.2. Namen (identifier)

Namen bezeichnen Konstanten, Typen, Variable, Prozeduren, Funktionen und Programme.

Namen sind auf 128 Zeichen begrenzt. Zur Unterscheidung reicht die Abweichung in einem dieser Zeichen aus.



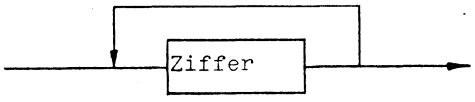
Alle Namen müssen mit einem Buchstaben beginnen, danach können Ziffern oder Buchstaben in beliebiger Reihenfolge angefügt werden.

2.1.3. Zahlen

Für Zahlen wird die Dezimalschreibweise verwendet.

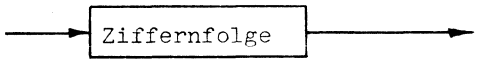
Man unterscheidet folgende Darstellungen:

- Ziffernfolgen (digit sequence)



Beispiele: 1|34

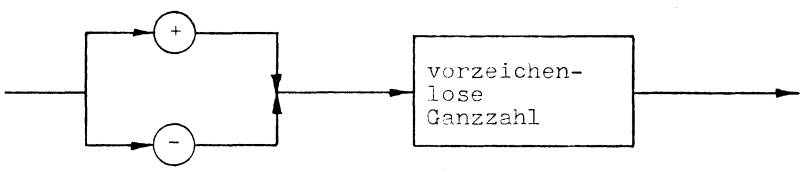
- Vorzeichenlose Ganzzahl (unsigned integer)



Beispiele: 7|50

Die Ziffernfolge entspricht einer vorzeichenlosen Ganzzahl; ihre gesonderte Darstellung ist aus formalen Gründen erforderlich. So wäre es falsch, eine Ziffernfolge ohne Zusammenhang als "vorzeichenlose Ganzzahl" darzustellen.

- Ganzzahl mit Vorzeichen (signed integer)

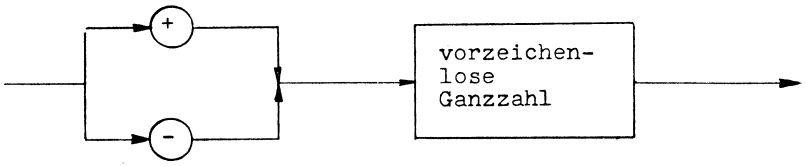


Beispiele: +3|-31

- Skalarfaktor (scale-factor)

Skalarfaktor (auch Skalierungsfaktor genannt) ist ein ganzzahliger Anhang einer Zahl. Er stellt den Exponenten einer Zehnerpotenz dar, der mit der Zahl multipliziert den Zahlenwert ergibt. Positive Skalarfaktoren stellen Zehnerpotenzen dar, die größer oder gleich 1 sind, negative solche die kleiner als 1 sind.

Sie entsprechen einer "Ganzzahl mit Vorzeichen" (signed integer).

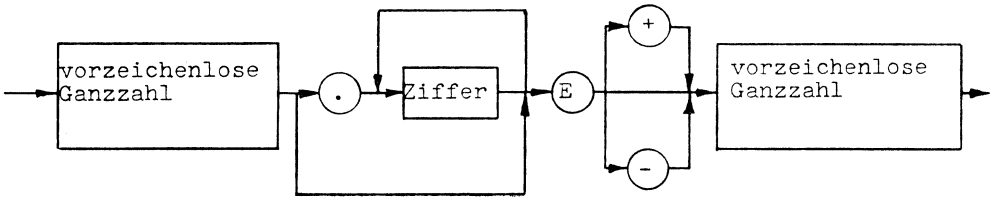


Beispiel: siehe "vorzeichenlose Realzahl"

- Vorzeichenlose Realzahl (unsigned real)

Realzahlen (real) umfassen den Bereich der reellen 1* Zahlen, die neben den Ganzzahlen auch die gebrochenen Dezimalzahlen beinhalten.

Bei der Darstellung von Realzahlen mit Skalarfaktor wird der Buchstabe E zwischen Zahl und Skalarfaktor gesetzt.



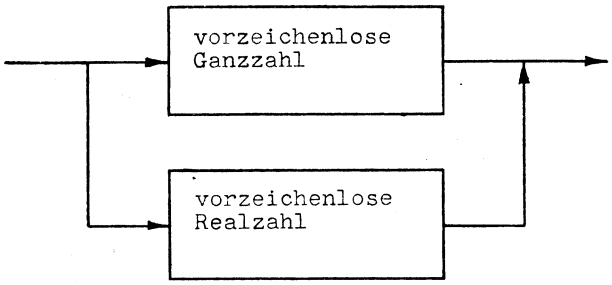
Beispiele: $6E\text{-}3 = 6x10^{-3} = 6x\frac{1}{1000} = 0.006$

$43.35E+5 = 43.35x10^{+5} = 43.35 \times 10000 = 433500$

1* Gegenüber dem mathematischen Begriff der reellen Zahlen ergibt sich bei Rechnern die Einschränkung auf Zahlen mit begrenzter Stellenanzahl.

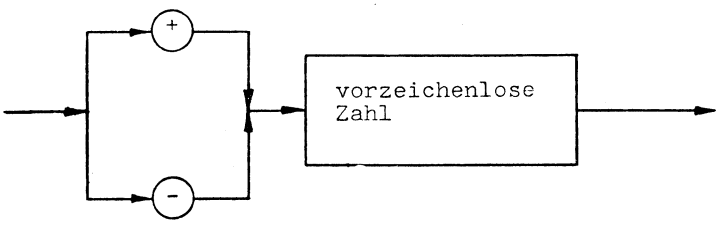
- vorzeichenlose Zahl (unsigned number)

Der Begriff "Zahl" umfaßt den Begriff der "Real-Zahlen". Die syntaktische Darstellung für "Zahl" ist dann geeignet, wenn nicht vorher bekannt ist, ob Ganzzahlen gesondert auftreten oder nicht. Ganzzahlen lassen sich Realzahlen zuweisen.



Beispiele: 4; 0.5; 3.6

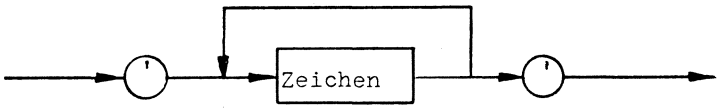
- Zahl mit Vorzeichen (signed number)



Beispiele: +16; -5; +0.8; -4.9

2.2. Zeichenketten (strings)

Es handelt sich um Zeichenfolgen (character strings), die bei der Darstellung in Hochkommata eingeschlossen werden.

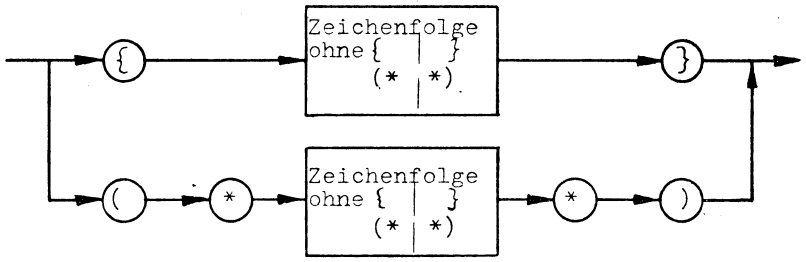


Beispiele: 'B': '*': 'ZEICHENKETTE'

Zeichenketten können zu Kennzeichnungen und Erläuterungen (u.a. Texte) eingesetzt werden. Wenn Hochkommata als Bestandteil einer Zeichenkette auftauchen, sind sie als zwei Hochkommata darzustellen (z.B. 'DER BEGRIFF 'STRING' ENTSPRICHT EINER ZEICHENKETTE.').

2.3. Zwischenräume, Kommentare, Zeilenende

Im PASCAL-Programm dürfen beliebig viele Leerstellen oder -zeilen eingefügt werden. Das gleiche gilt für erläuternde Texte (Kommentare), die allerdings an folgende Form gebunden sind:

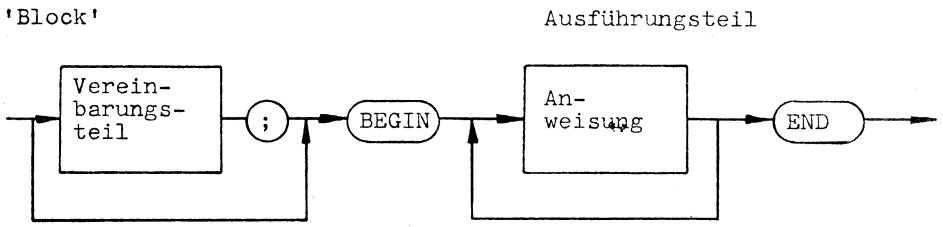


Je zwei aufeinanderfolgende vorzeichenlose Zahlen, Namen und Schlüsselworte müssen mindestens durch eine Leerstelle oder Kommentar oder Zeilenende getrennt werden. Innerhalb dieser Begriffe darf aber keine Unterbrechung durch Leerstellen, -zeilen, Trennzeichen und Zeilenende erfolgen.

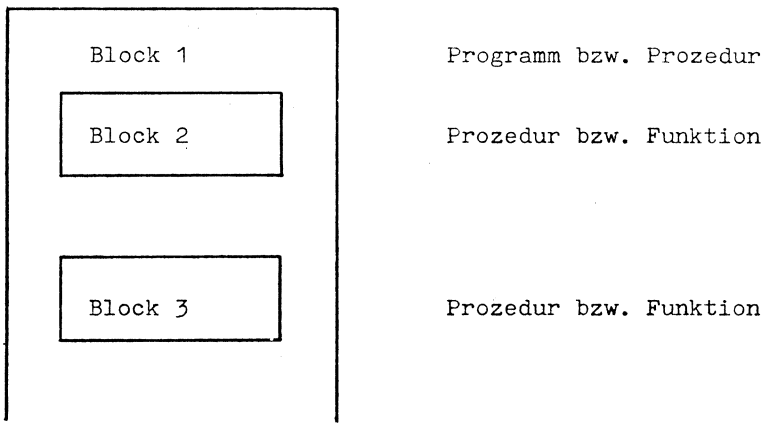
3. Vereinbarungsteil

3.1. Blockkonzept

Ein Block besteht aus einem Vereinbarungsteil (Definitionen und Deklarationen) und einem Ausführungsteil (Anweisungen), die zusammen Teil einer Prozedur, einer Funktion oder eines PASCAL-Programmes sind.



Da man innerhalb einer Prozedur bzw. Funktion wiederum Prozeduren bzw. Funktionen einrichten kann, ergeben sich folgende Blockschachtelungsmöglichkeiten:



Der Block 1 ist den Blöcken 2 und 3 übergeordnet. Block 2 und 3 sind parallel angelegt. Innerhalb der Blöcke 2 und 3 sind mehrere untergeordnete Blöcke zugelassen.

In der vorliegenden Implementierung dürfen max. 127 Prozeduren und Funktionen je Programm vorkommen. Die Schachtelungstiefe beträgt max. 7.

3.1.1. Gültigkeitsbereiche von Namen und Marken

Die in einem Block definierten Größen nennt man lokal bezüglich des Blockes. Global heißen diejenigen Vereinbarungen, die in einem übergeordneten Block festgelegt wurden.

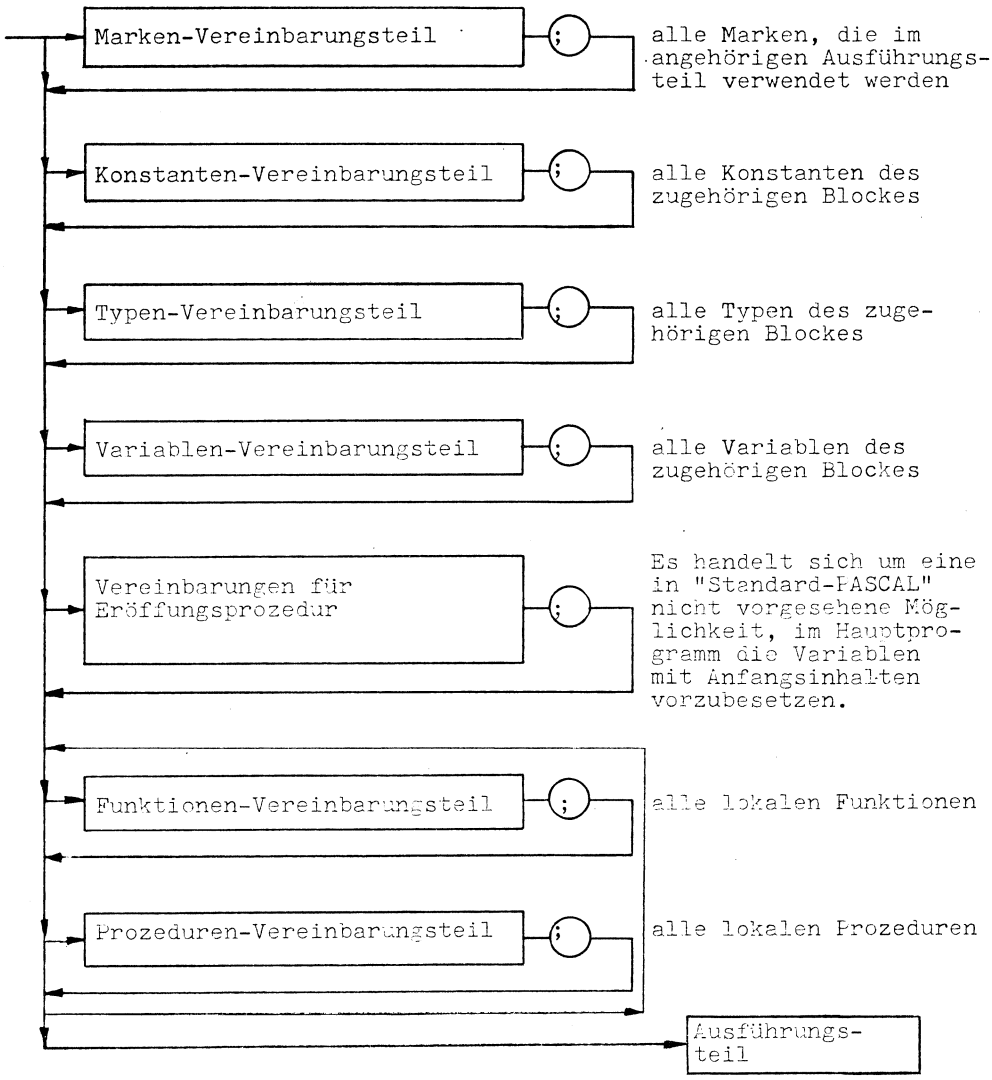
- Ein Name oder eine Marke darf innerhalb eines Blockes nicht mehr als einmal definiert werden.
- Alle Namen und Marken sind demzufolge "lokal" bezogen auf einen Block. Sie gelten auch "global" für Programmteile, die diesem Block untergeordnet sind (untergeordnete Blöcke).
- Wenn "lokale" und "globale" Vereinbarungen den gleichen Namen tragen, dann werden die "globalen" im "lokalen" Bereich nicht wirksam. Das kann aus Gründen sachlicher Vergleichbarkeit sinnvoll sein.

Entsprechend dieser genannten Bedingungen müssen alle Bestandteile der Vereinbarungsteile festgelegt werden.

3.1.2. Aufbau des Vereinbarungsteils

Im Vereinbarungsteil werden in bestimmter Reihenfolge Marken, Konstanten, Typen, Variable, Funktionen und Prozeduren beschrieben. Die Reihenfolge der Vereinbarungen ergibt sich aus folgendem Syntaxdiagramm.

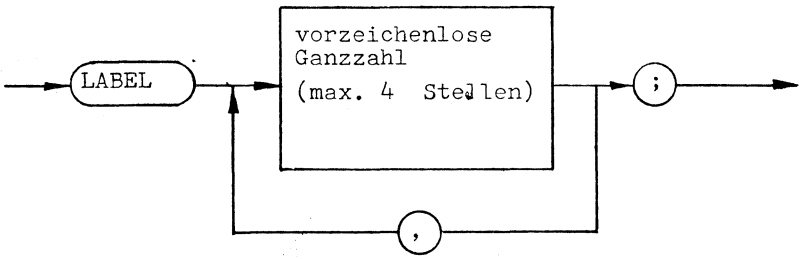
'Block'



3.2. Marken - Vereinbarungen

Marken (label) sind Ziele, zu denen im Programmablauf gesprungen werden kann (Ansprung mit GOTO). Im Vereinbarungsteil (Markendeklarationsteil oder "label-declaration-part") muß LABEL als Schlüsselwort ausdrücklich genannt werden.

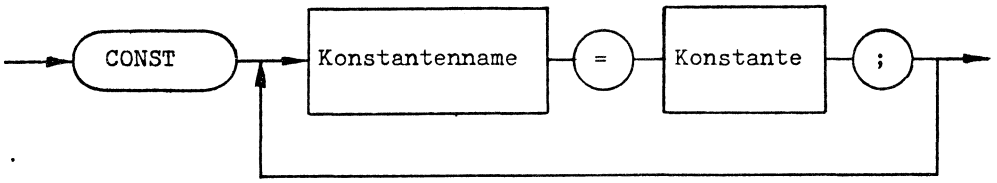
Marken werden als vorzeichenlose Ganzzahlen (unsigned integer) gebildet und dürfen nicht mehr als vier Stellen groß sein. Jede Marke darf nur einmal in ihrem Gültigkeitsbereich auftreten.



Beispiel: LABEL 1|23|4

3.3. Konstanten-Definitionen

Die Vereinbarung von Konstanten erlaubt es, einem Namen einen festen Wert zuzuweisen. Ein Konstantenname darf nicht auf der linken Seite einer Zuweisung verwendet werden.

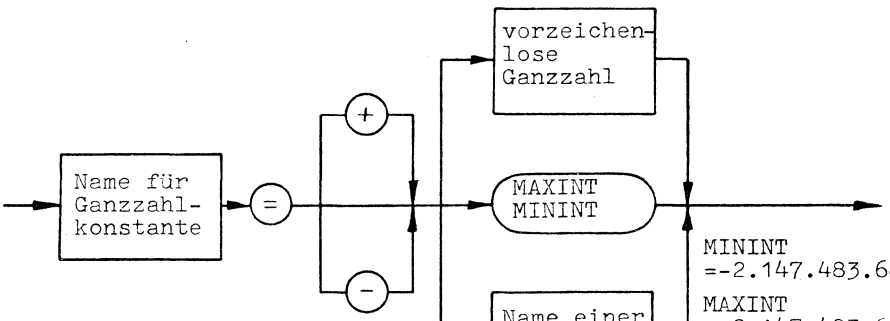


"Ganzzahl"- oder "Integer"-Konstante

Ein vereinbarter Konstantenname darf in einer danach folgenden Konstantenvereinbarung verwendet werden. Eine Konstante ist während des ganzen Programms unveränderbar.

Der Typ der Konstante wird nach dem Gleichheitszeichen ausgedrückt.
Es gibt folgende Möglichkeiten:

"Ganzzahl"- oder "Integer"-Konstante



MININT
=-2.147.483.647

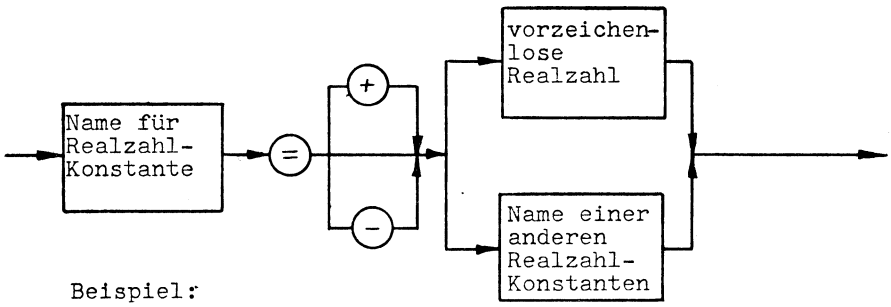
MAXINT
= 2.147.483.647

(standardmäßig die kleinste und größte Zahl, die auf das jeweilige System zugelassen sind)

Beispiel:

```
CONST KONSTANTE1=1980;
      KONSTANTE2=KONSTANTE1;
      KONSTANTE3=MAXINT;
```

Realzahl- oder "Real"-Konst.

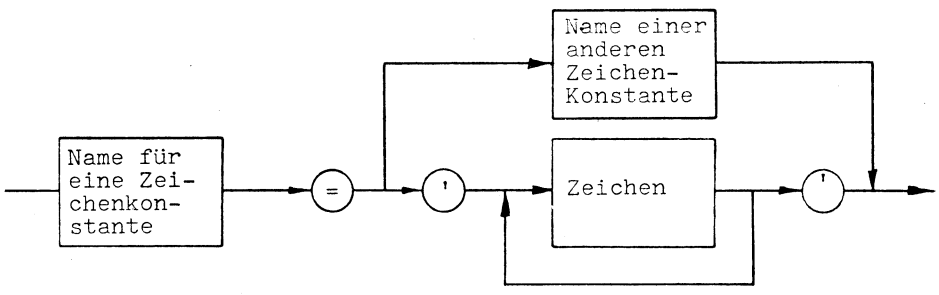


Beispiel:

```
CONST PI: 3.14159;
CONST A :-0.753E2;
```

Zeichen- oder "Character"-Konstante

Bei mehreren Zeichen spricht man von Zeichenketten- oder "String"-Konstanten

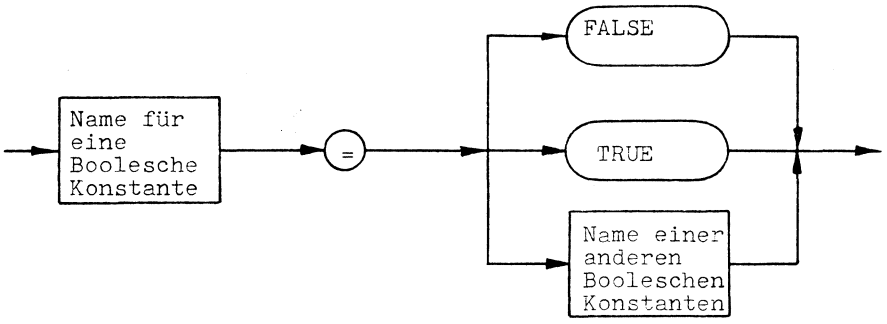


Beispiel:

```
CONST KONSTANTE-1 =  ... ;
      KONSTANTE-2 = 'DIES IST EINE ZEICHENKONSTANTE';
```

Boolesche Konstanten (boolean constant)

Hierbei werden Aussagen der Booleschen Algebra zugrundegelegt, die auf den Aussagen "falsch" (=FALSE) und "richtig" (=TRUE) beruhen. Die formelmäßige Verbindung (symbolische Logik) erlaubt die Darstellung und Lösung von Problemen, die ursprünglich nur als sprachliche Folgerungen erfaßt werden konnten. Damit wurde es möglich, schlüssige Ergebnisse für komplizierte Aufgabenstellungen (Mengenlehre, Schaltalgebra, Operations Research, Kybernetik, Informatik usw.) abzuleiten.



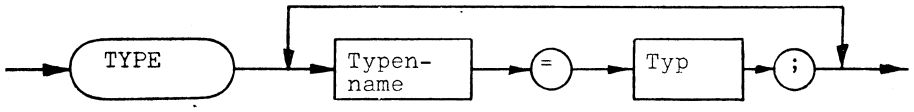
Beispiel:

```
CONST WAHR  =  TRUE;
      FALSCH =  FALSE;
```

3.4. Typ-Definitionen

Ein Typ bestimmt die Werte und die Struktur, die Variablen dieses Typs annehmen können und die Operationen, die mit diesen Variablen ausgeführt werden können.

Prinzipiell ist folgende Darstellung vorgeschrieben:



Durch die Typen-Definition wird ein Typ mit einem Namen versehen.

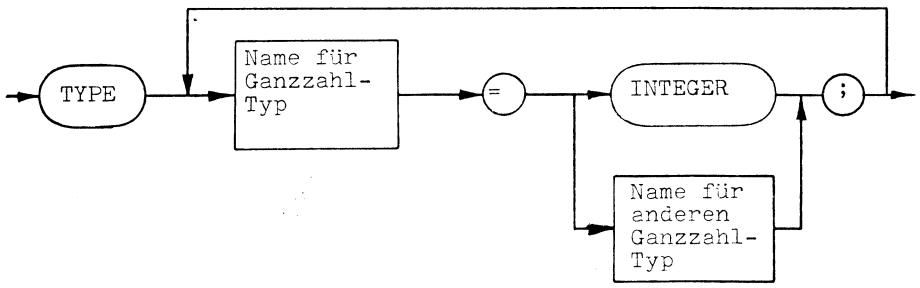
Grundsätzlich darf der vereinbarte Typ nur definierte Typnamen enthalten. Ausnahmen hierzu sind bei der Verwendung von Zeigertypen (vgl. 3.4.3) zugelassen. Weiterhin sind die Standardtypen (INTEGER, BOOLEAN, CHARACTER, REAL) als "von vornherein bestimmt" von dieser Regel ausgenommen.

3.4.1. Einfache Typen (simple type)

Alle einfachen Typen definieren geordnete Mengen und Werte.

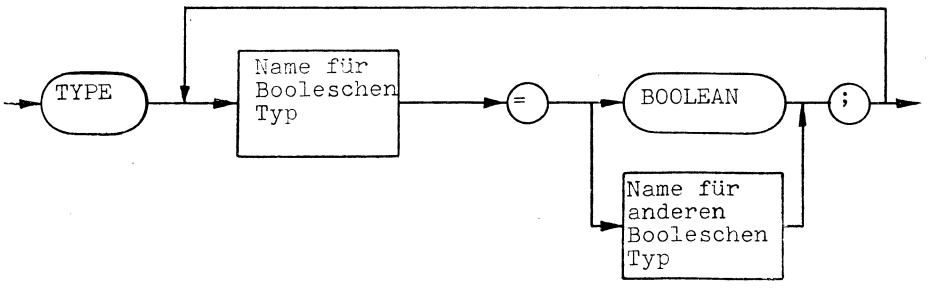
3.4.1.1. Standardtypen (standard simple-types)

- Ganzzahl-Typen (integer)



Die einsetzbaren Ganzzahlen werden intern 4 Byte lang dargestellt. Der Wertebereich geht von MININT 2.147.483.647) bis MAXINT 2.148.483.647).

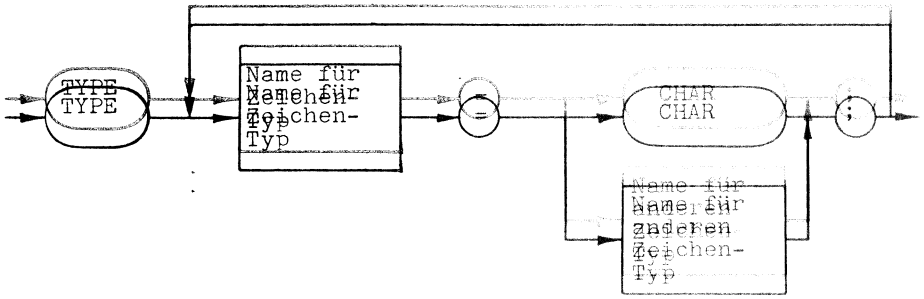
- Boolesche Typen (boolean)



Diese Werte sind Wahrheitswerte (vgl. 3.2 Boolesche Konstanten), die durch die Aussagen "falsch" (=FALSE) oder "wahr" (=TRUE) dargestellt werden. Der Boolesche Typ wird maschinenintern mit der binären Darstellung 0 (=FALSE) und 1 (=TRUE) dargestellt (FALSE < TRUE).

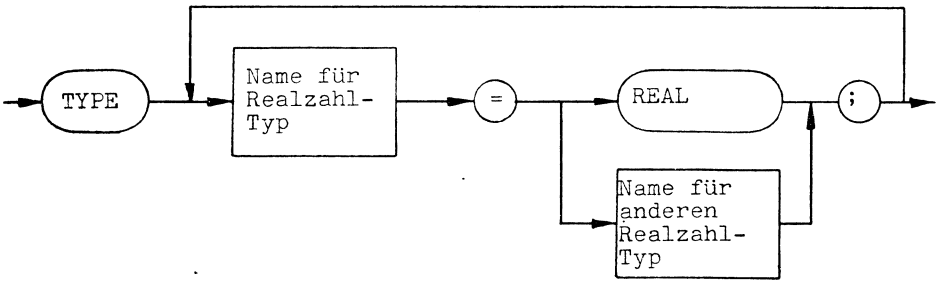
- Zeichen-Typen (character type)

Es dürfen alle Zeichen des Zeichenvorrates auftreten (vgl. 2:1).



Die Typen INTEGER, BOOLEAN und CHAR werden auch Ordinalstypen (ordinal type) genannt, weil sie standardmäßig zählbar sind. (1<2<3 ...), (FALSE<TRUE), (A<B<C ...<Z).
Im Gegensatz dazu handelt es sich bei Aufzählungstypen (vgl. 3.4.1.2) um frei vereinbarte Namen.

- Realzahl-Typen (real)



Die einsetzbaren realen Zahlen werden intern durch eine 7 Byte lange Mantisse und einen 1 Byte Exponenten dargestellt.

Der Zahlenbereich liegt bei 10^{-38} bis 10^{+54}, wobei sich durch Umrechnungsverschiebungen zwischen den Zahlensystemen und durch die Beschränkung der Dezimaldarstellung (auf 15-16 Stellen) kleinere Spannen ergeben können.

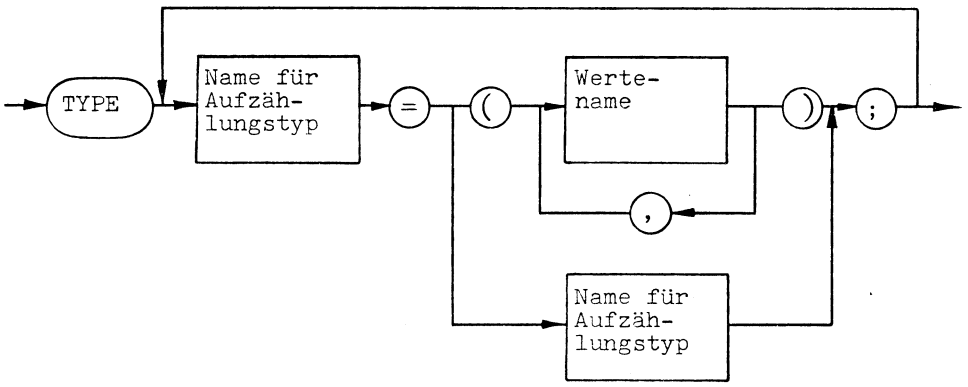
3.4.1.2. Aufzählungstyp (Enumerated-Types)

Aus Gründen der Verständlichkeit wurde hier der von Professor Wirth geprägte Begriff "Skalarer Typ" (scalar type) durch "Aufzählungstyp" ersetzt.

Durch die Vereinbarung wird einem Namen (Typname) eine Wertmenge bestehend aus Namen (= Werte-Namen) zugewiesen.

Der Reihenfolge der Werte-Namen werden maschinenintern Stellenwerte zugeordnet, nach denen die Namen gezählt und verglichen werden können.

In der vorliegenden Implementierung dürfen "Aufzählungstypen" maximal 256 Elemente haben.

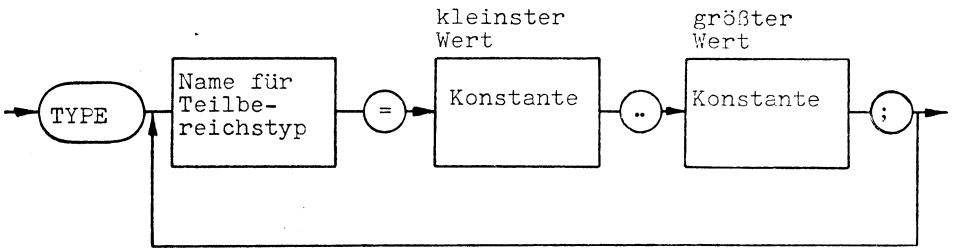


Beispiel:

```
TYPE WOCHE = (MONTAG, DIENSTAG, MITTWOCH, DONNERSTAG,
FREITAG, SAMSTAG, SONNTAG);
```

3.4.1.3. Teilbereichstypen (subrange-type)

Es ist möglich, mit Hilfe der Ordnungs-Typen (INTEGER, BOOLEAN, CHAR) und der Aufzählungstypen neue Typen mit eingeschränkter Wertemenge (Wertbereich) zu vereinbaren. Das geschieht durch Angabe des kleinsten und größten Wertes als Konstanten. Teilbereichstypen haben alle Eigenschaften des Basistyps mit der Beschränkung auf das vereinbarte Intervall.



Beispiel:

```
TYPE WOCHE    = (MONTAG,DIENSTAG,MITTWOCH,DONNERSTAG,
                 FREITAG,SAMSTAG,SONNTAG);

     WERKTAG = MONTAG .. FREITAG;
```

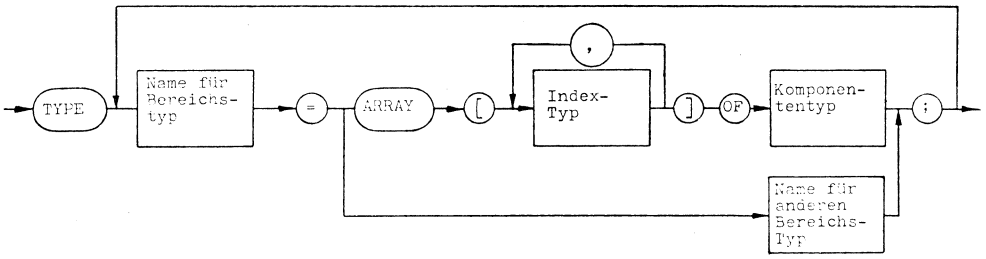
3.4.2. Strukturierte Typen (structured type)

Es handelt sich um eine Zusammensetzung von Komponenten nach bestimmten Ordnungsmerkmalen. Die Komponenten werden aus einem oder mehreren anderen Typen gebildet.

3.4.2.1. Bereichs-Typen (array type) 1*

Ein Bereichstyp besteht aus einer festgelegten Anzahl von Komponenten desselben Typs (Komponenten-Typ oder "component- type"). Die Komponenten des Bereichs können über Indizes angesprochen werden (Index-Typ oder "index-type").

1* Der Begriff "array" wird hier als "Bereich" und nicht als "Feld" (field) übersetzt (vgl. 3.4.2.2).



Der Indextyp wird mit Ordinaltypen oder Aufzählungstypen gebildet (vgl. 3.4.1.3), deren Bereich (Intervall aus Ober- und Untergrenze) definiert ist. Beim Komponententyp ist mit Ausnahme des Dateityps (file type - vgl. 3.4.2.4) jede andere Typ-Angabe möglich.

Beispiele

	Indextyp	Komponententyp	Formulierung
a)	BOOLEAN	INTEGER	TYPE AFALL = ARRAY [FALSE..TRUE] OF INTEGER;
b)	CHAR	BOOLEAN	TYPE BFALL = ARRAY ['A'..'Z'] OF BOOLEAN;
c)	CHAR	REAL	TYPE CFALL = ARRAY [' '.. '/'] OF REAL;
d)	INTEGER	ARRAY REAL	TYPE DFALL = ARRAY [-255 .. 255] OF ARRAY [1 .. 100] OF REAL;
e)	Aufzähl-lungstyp	REAL	TYPE ZAEHLER = (ANTON, BERTA, CAESAR); TYPE EFALL = ARRAY [ANTON .. CAESAR] OF REAL; Mit Hilfe eines Aufzählungstyps (ZAEHLER) können Indextypen gebildet werden.
f)	BOOLEAN	INTEGER	TYPE FFALL = ARRAY [FALSE .. FALSE] OF INTEGER; Wenn Unter- und Obergrenze übereinstimmen, besteht der Komponententyp aus nur einem Element.

Jede Definition kann außer den bestimmten Typen weitere zusammengesetzte Typdefinitionen (z.B. Bereichstyp) heranziehen. Dazu kommen auch die Anwendungsmöglichkeiten des Satztyps (record type - vgl. 3.4.2.2).

3.4.2.2. Satz-Typ (record-type)

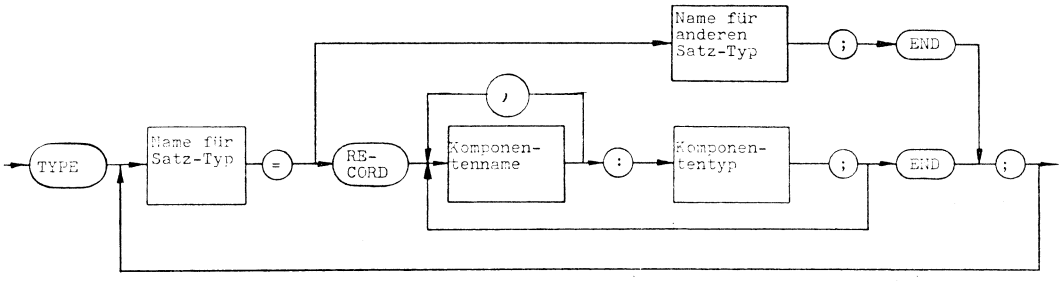
- Einfacher Satztyp

Im Vergleich zum Bereichstyp (ARRAY) gibt es folgende wesentlichen Unterschiede:

Die Komponenten des Satz-Typs (Satzfeld oder record field) bestehen aus gleichen oder verschiedenen Typen. 1*

Die Satzfelder (record field) werden nicht über Indizes angesprochen sondern haben einen eigenen festen Namen (Komponentennamen oder field identifier).

Dem Komponentenname folgt der Komponententyp (component-type).



Die Konstruktion aus Komponentenname und Komponenten-Typ wird Komponentenliste (field list) genannt.

1* Im Zusammenhang mit PASCAL sind array (Bereich) und field (Feld) zu unterscheiden. Die Satzfelder (record field) haben im Gegensatz zu den Elementen eines Bereichstyps (array type) eigene Namen und können von unterschiedlichem Typ sein.

Hier ist ein Satz-Typ mit drei verschiedenen Typen gebildet worden: Zählbarer Typ, Teilbereichstyp bei Ganzzahlen und Ganzzahltyp.

Beispiel:

```
TYPE DATUM = RECORD MONAT: (JAN,FEB,MAER,APR,
                            MAI,JUN,JUL,AUG,SEP,
                            OKT,NOV,DEZ);
                    TAG:   1 .. 31;
                    JAHR:  INTEGER;
             END;
```

- Varianten-Satztyp

Hier kann ein Satzfeld (record field) bei gleicher Feldbelegung mehrere unterschiedliche Typvarianten darstellen (Redefinition).

Der Variantenteil (variant part) beginnt mit dem Schlüsselwort "CASE" und dem Namen für die Auswahlmarke (tag field).

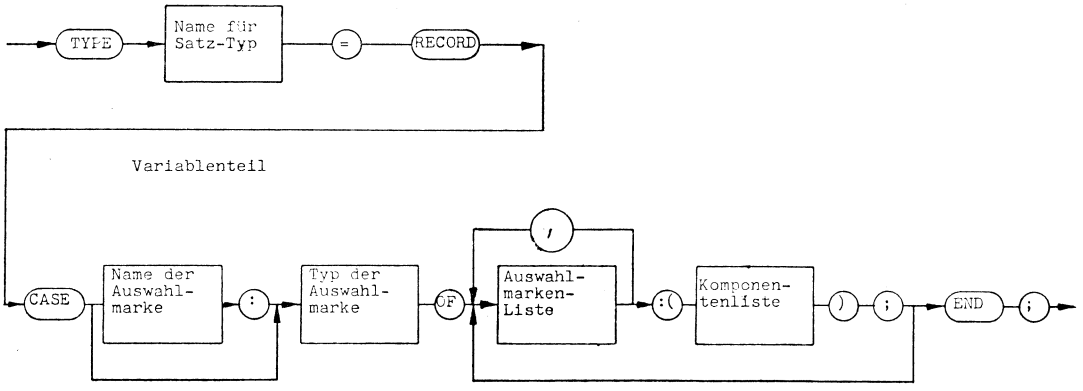
Danach folgt der Typ für die Auswahlmarke (tag type) und die Auswahlmarkenliste (case-constant-list), die durch das Schlüsselwort "OF" eingeleitet wird.

Die Auswahlmarkenliste (case-constant-list) gibt die Konstanten an, die die Auswahlmarke annehmen darf. Die den Konstanten zugeordnete Komponentenliste bestimmt die Art der Definition. Sie erfolgt mit Ordinaltypen oder zählbaren Typen.

Bei den hier zugelassenen Typen ist es nicht gestattet, daß als Komponenten des Satztyps Dateitypen (file type - vgl. 3.4.2.4) auftreten.

Alle Auswahlmarken in der Liste (case-constant-list) müssen voneinander verschieden sein. Auswahlmarke und Konstante müssen typidentisch sein.

Innerhalb der zugeordneten Komponentenliste (field list) wird die Variante durch Komponentenname (field identifier) und Komponententyp (component type) bestimmt.



Beispiel:

```
TYPE FIGUR = (DREIECK,QUADRAT,RECHTECK);
     GEOMETRIE = RECORD CASE FLAECHE : FIGUR OF
                 DREIECK : (GRUNDSEITE,HOEHE:REAL);
                 QUADRAT : (SEITE:REAL);
                 RECHTECK: (LAENGE,BREITE:REAL);
                 END;
```

Im Verarbeitungszusammenhang ändert sich die Struktur der Satztypkomponente (Ausdruck ab CASE), wenn der Auswahlmarke ein neuer Wert zugewiesen wird. Dann ist der bis dahin geführte Inhalt nicht mehr definiert.

Bei allen Formen des Satztyps können außer Standardtypen weitere zusammengesetzte Typdefinitionen verwendet werden, die zudem beliebige Schachtelungsmöglichkeiten bieten.

3.4.2.3. Mengentyp (set type)

Der hier verwendete Begriff "Mengentyp" wird in dieser Darstellung nur für den "set-type" gebraucht. Der Mengentyp definiert einen Wertebereich, der die Potenzmenge seines Basistyps umfaßt.

Hinter Potenzmenge steht ein eindeutig beschriebener Begriff aus der mathematischen Mengenlehre, der alle Kombinationsmöglichkeiten einer beschriebenen Grundmenge umfaßt. Dabei werden die Grundmenge und die "leere Menge" = { } dazugerechnet.

Beispiele	Darstellung Grundmenge M	Anzahl Elemente	Darstellung Potenzmenge	Anzahl Kombinationen
a)	{0}	1	{ } {0}	$2=2^1$
b)	{0,1}	2	{ } {0} {1} {0,1}	$4=2^2$
c)	{0,1,2}	3	{ } {0} {1} {2} {0,1} {0,2} {1,2} {0,1,2}	$8=2^3$
d)	{1,2,3,4}	4	{ } {1} {2} {3} {4} {1,2} {1,3} {1,4} {2,3} {2,4} {3,4} {1,2,3} {1,2,4} {1,3,4} {2,3,4} {1,2,3,4}	$16=2^4$

Die Untersuchung einer fortgesetzten Reihe führt zu der Erkenntnis, daß die Anzahl der Kombinationen (Mächtigkeit der Potenzmenge) Zweierpotenzen in Abhängigkeit von der Anzahl der Elemente der Grundmenge sind:

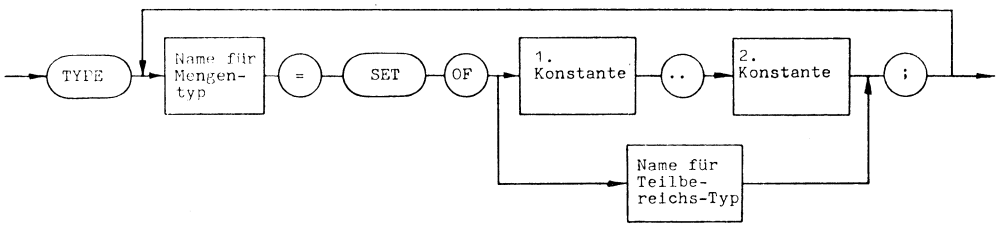
$$P(M) = 2^{M}$$

Da die Anzahl der Elementengrundmenge als Exponent zur Basis 2 auftritt, hat man den Begriff "Potenzmenge" geprägt.

Bei der Typdefinition entspricht die Grundmenge dem Basistyp und der Inhaltsbereich der Potenzmenge dem Mengentyp.

In dieser Implementierung sind als Basistypen Teilbereichstypen aus Ordinaltypen und "Aufzählungs - Typen" bis 255 Elemente (0-255) erlaubt.

Der Teilbereich kann im Mengentyp eingeschlossen sein oder als gesonderter Teilbereichstyp formuliert werden.

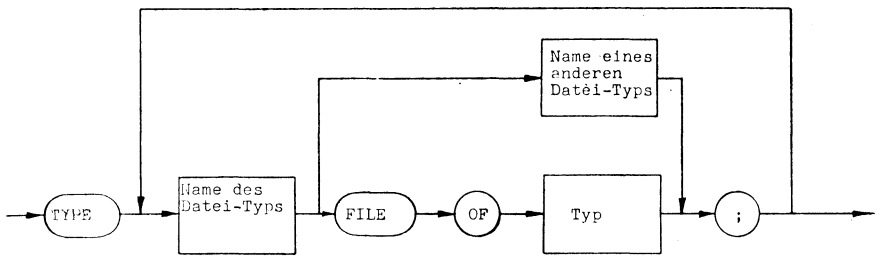


Beispiele:

	Basistyp	Formulierung
e)	INTEGER (eingeschlossen)	TYPE AINT=SET OF 0..9;
	(gesondert)	TYPE TEILB = 0..9; TYPE BINT=SET OF TEILB;
f)	BOOLEAN (eingeschlossen)	TYPE CBOOL=SET OF FALSE..TRUE;
g)	CHAR (eingeschlossen)	TYPE DCHAR=SET OF 'A'..'B';
h)	Aufzählungstyp gesondert	TYPE SKAT = (KREUZ,PIK,HERZ, KARO); TYPE BUBEN=SET OF SKAT; Mit dieser Definition werden alle Bubenkonstellationen beim Skatspiel definiert (vgl. Beispiel d).

3.4.2.4. Datei-Typ (file type)

Beim Dateityp handelt es sich um eine Folge von Komponenten gleichen Typs. Die Anzahl der Komponenten ist zum Zeitpunkt der Definition unbestimmt; ebenso der Inhalt der Datei.



Datei-Typen sind in dieser Implementierung als Komponenten nicht erlaubt.

Mit dem Datei-Typ lassen sich sequentielle Dateien aufbauen.

Dateinamen (z.B. ADATEI) müssen wie die Standarddateien (INPUT, OUTPUT) hinter dem Programmnamen (vgl. 1.1) aufgeführt sein.

Beispiel:

```
TYPE ADRESSE = RECORD NAME:ARRAY [1 .. 20] OF CHAR;
                      VORNAME:ARRAY [1 .. 15] OF CHAR;
                      STRASSE:ARRAY [1 .. 30] OF CHAR;
                      HAUSNUMMER: INTEGER;
                      POSTLEITZAHL: INTEGER;
                      ORT: ARRAY [1 .. 25] OF CHAR;
                END;
     PERSONEN=FILE OF ADRESSE;
```

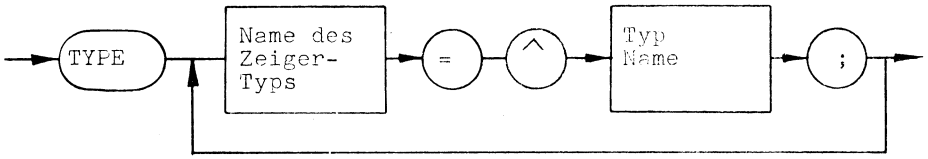
In diesem Zusammenhang ist darauf hinzuweisen, daß eine Typdefinition noch keine Datei begründen kann (vgl. 3.4). Dazu sind erst entsprechende Variablen-Vereinbarungen erforderlich (vgl. 3.5.2.3).

Im Standardprolog 1* ist ein vordefinierter Datei-Typ mit dem Namen "TEXT" enthalten. Variable dieses Typs heißen Text-Dateien und sind vom Zeichentyp (CHAR).

1* Unter Prolog werden in diesem Zusammenhang vom Übersetzer vorbesetzte Programmvereinbarungen verstanden, die für jedes mit diesem Compiler umgewandelte Programm gelten und bei der Programmierung genutzt werden können.

3.4.3. Zeiger-Typ (pointer-type)

Die Vereinbarung eines Zeiger-Typs entspricht der üblichen Typ-Definition:



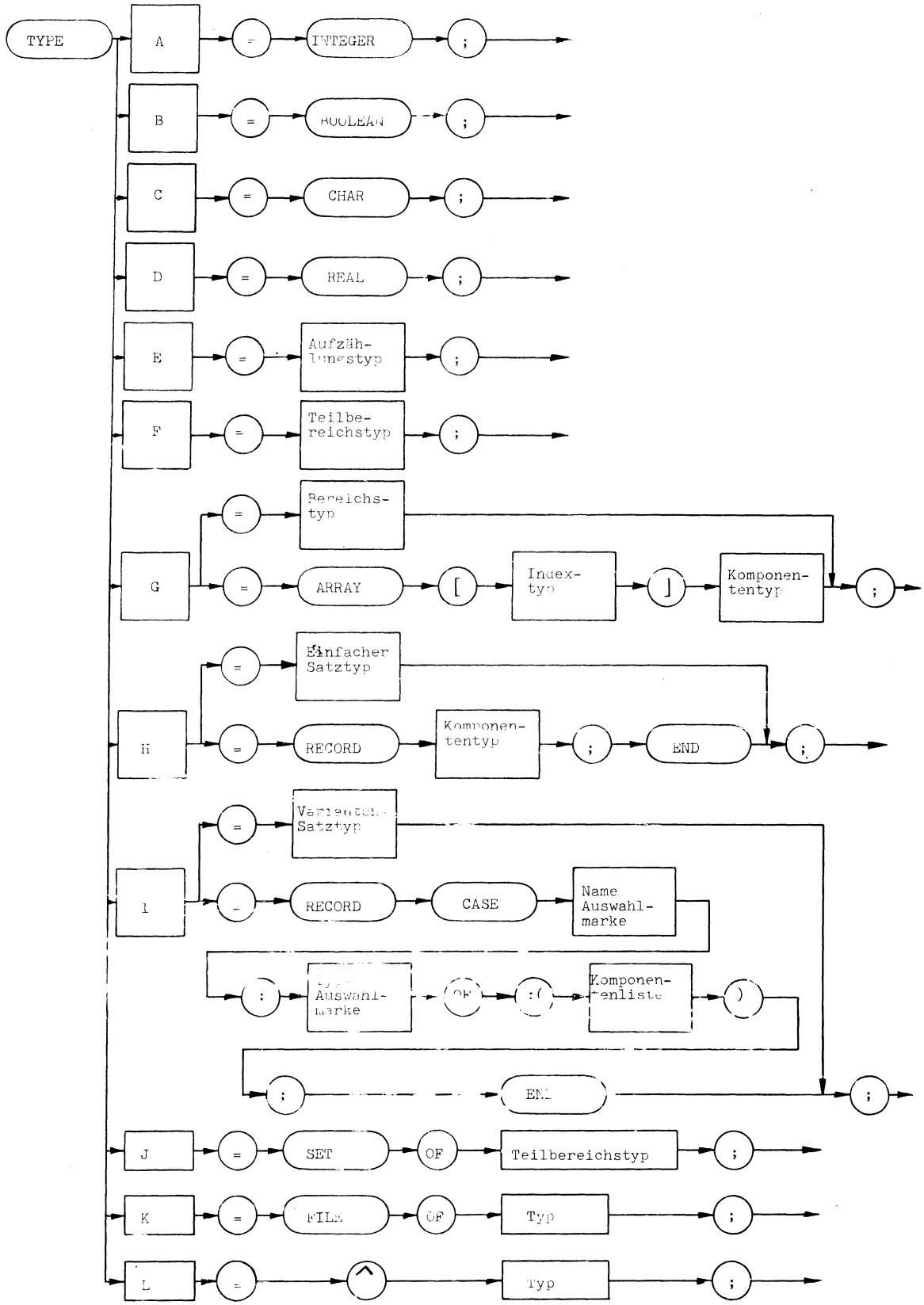
Der Zeiger-Typ erlaubt die Darstellung von Hauptspeicheradressen (je 2 byte) zur Adressierung von Variablen. Die eigentlichen Anwendungsmöglichkeiten des Zeiger-Typs lassen sich dabei nur in Verbindung mit Variablen sinnvoll beschreiben (vgl. 3.5.3).

Folgende Typ-Vereinbarung stellt ein Beispiel für den Zeiger-Typ dar:

```
TYPE A = RECORD
            BZEIGER :   ^B;
         END;
     B = RECORD
            AZEIGER :   ^A;
         END;
```

Bei jeder anderen Definition ist es zwingend, den Namen einer Komponenten vorher zu bestimmen. Der Zeiger-Typ gestattet die gegenseitige Bezugnahme (= Aufeinanderzeigen) in Abweichung von dieser Regel.

3.4.4. Zusammenfassendes Typen-Diagramm



3.4.5. Zuordnungsverträglichkeit (assignment - compability)

Wenn Inhalte unterschiedlichen Typs (z.B. Variable) verarbeitungstechnisch zusammengeführt werden sollen (z.B. bei einer Wertzuweisung), muß der Zuweisungstyp mit dem Empfangstyp zuordnungsverträglich sein. So ist es beispielsweise möglich, eine Ganzzahl einer Variablen vom Typ "REAL" zuzuweisen, was umgekehrt ausgeschlossen ist, nämlich eine Realzahl einer Variablen vom Typ "INTEGER" zu übergeben.

Die Voraussetzungen der Zuordnungsverträglichkeit sind in folgenden Fällen gegeben:

Beispiele

Wir beschränken uns auf die Typ-Darstellung; tatsächlich sind andere Konstruktionen (z.B. Zuweisungen an Variable) erforderlich.

	Zuweisungstyp	Empfangstyp
a) Typidentität Das gilt außer für Datei-Typen bei allen anderen Typen	REAL	REAL
b) vorgegebene Untermenge Diese Zuweisungsmöglichkeiten gelten im Bereich der Ordnungstypen (CHAR,INTEGER, BOOLEAN).	INTEGER	REAL

	Zuweisungstyp	Empfangstyp
c) vereinbarte Untermenge		
(1) Teilbereichs-Typen	TYPE INTTEIL = 5..10;	INTEGER
Auch hier sind ähnliche Zuweisungsmöglichkeiten wie bei b).	TYPE ALPHTEIL= J..Q;	CHAR
	TYPE WERKTAG = MO-FR; WOCHE =(MO,DI,MI,DO,FR, SA,SO);	
(2) Mengen-Typen	TYPE ZEHN=SET OF 1..10;	TYPE HUNDERT= SET OF 1..100;
(3) Zeichenfolgen	TYPE ABC=ARRAY[0..2] OF CHAR;	TYPE ABCDEF=ARRAY[0..5] OF CHAR;

Eine Zuordnungsverträglichkeit ist hierbei gegeben, wenn bei der Zuweisung bestimmt wird, an welcher Stelle die kürzere Zeichenfolge aufgenommen werden soll (vgl. Indizierung / 3.5).

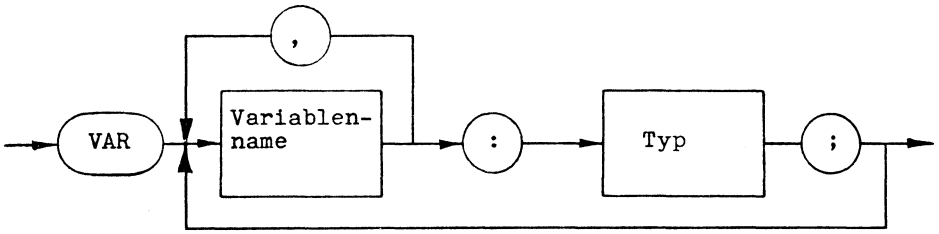
Bei Teilbereichstypen und bei eingeschalteter Kontrollfunktion (check option) werden Laufzeitfehler gemeldet, wenn die zugrundegelegte Zuordnungsverträglichkeit nicht mehr gegeben ist (Überschreitungen der vorgegebenen Intervalle).

Bei allen anderen Zuordnungsunverträglichkeiten werden die Fehler zur Übersetzungszeit gemeldet.

3.5. Deklaration von Variablen

Variable sind im Vergleich zu Konstanten inhaltlich änderbare Objekte, die aber ebenfalls durch bestimmte Namen (Variablennamen - "variable-identifier") oder Selektoren 1* angesprochen werden können.

Die Definition des Typs einer Variablen kann bei der Variablen-Deklaration erfolgen (Standard-Typen) oder durch Nennung eines vorher definierten Typennamens (Benutzertyp).



Der Geltungsbereich einer Variablendeklaration richtet sich nach der Blockstruktur (vgl. 3.1.0, 3.1.1).

1* Unter Selektion werden Auswahlmechanismen verstanden, mit denen man die Komponenten von Variablen ansprechen kann. Man unterscheidet in Anlehnung an den Aufbau der strukturierten Datentypen für die Variablenselektion:

- Bereichsselektoren (Indizierung)
  z.B. VAR A : ARRAY [1..3] OF CHAR;
  Indizierung der 2. Komponente A [2]

- Satzselektoren (Namensverknüpfung)
  z.B. VAR A : RECORD B,C,D:CHAR;END;
  Namensverknüpfung f.Komponente C A.C

3.5.1. Einheitsvariable (entire variable)

Hierbei handelt es sich um eine Variable, die innerhalb ihrer Bestimmung keine weiteren "Untervariablen" aufführt. Das unterscheidet sie von der Komponentenvariablen (vgl.3.5.2).

Beispiele:

a) mit Standardtypen

```
VAR NUMMER    :   INTEGER; (*BEZEICHNUNG Z.B.GANZZAHL-
                             VARIABLE*)
    AUSSAGE   :   BOOLEAN;
    BUCHSTABE:    CHAR;

    DIVISION,DIVIDEND,DIVISOR : REAL;
```

b) mit Aufzählungstypen (Aufzählungs-Variable)

```
VAR JAHRESZEIT   : (FRUEHLING,SOMMER,HERBST,WINTER);
```

c) mit Teilbereichstyp (Teilbereichs-Variable)

```
VAR TAGNUMMER    : 1..7;
```

d) mit Bereichstyp (Bereichs-Variable)

```
VAR BUCHSTABEN : ARRAY [A..Z] OF CHAR;
```

Während die Variable "Buchstaben" eine Einheitsvariable darstellt, deklariert Array bereits die Komponenten A bis Z. Um eine reine Einheitsvariable auszudrücken, kann man einen Typennamen ansprechen.

```
TYPE BUCHSTABEN        = ARRAY [A..Z] OF CHAR;
VAR VOKALE,KONSONANTEN : BUCHSTABEN;
```

e) mit Satztyp (Satz-Variable)

(entspricht den Darstellungsmöglichkeiten bei d))

```
TYPE EHESTATUS = RECORD. LEDIG,VERHEIRATET,
                 GESCHIEDEN,VERWITWET:BOOLEAN;
                 END;
VAR FAMILIENSTAND:EHESTATUS;
```

f) mit Mengentyp (Mengen-Variable)

```
VAR VERHEIRATET  :  SET OF FALSE..TRUE;
```

g) mit Dateityp (Datei-Variable)

```
VAR GANZZAHL     :  FILE OF INTEGER;
```

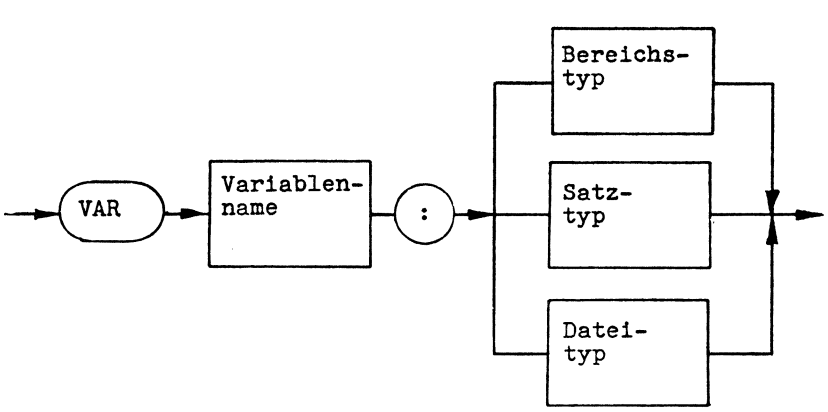
h) mit Zeigertypen (Zeiger-Variable)

Ähnlich wie d) und e) ist eine vorgelagerte TYP-Definition erforderlich, um eine Einheitsvariable zu deklarieren.

```
TYPE BUCHUNG = RECORD CASE KONTO  : INTEGER OF
                      1 : (SOLL   : REAL);
                      2 : (HABEN  : REAL);
               END;
VAR KASSE, BANK:^BUCHUNG;
```

3.5.2. Komponentenvariable (component-variable)

Komponentenvariable werden mit Einheitsvariablen eingeleitet. Die Untergliederungen einer Einheitsvariablen sind Komponentenvariablen.



Komponentenvariable

Indizierte Variable (vgl. 3.5.2.1)

Satzfeldbezeichner (vgl. 3.5.2.2)

Puffervariable (vgl. 3.5.2.3)

Auf die Komponenten einer Variablen kann über ihren Namen, auf den ein Selektor folgen muß, zugegriffen werden. Die Art des Selektors hängt von dem Typ der vorgelagerten Einheitsvariablen ab.

3.5.2.1. Indizierte Variable (indexed variable)

Wenn mit der Einheitsvariablen ein Bereichstyp festgelegt wird, ergibt sich mit Vereinbarung des Indextyps eine ansprechbare Anzahl von Komponenten.

Beispiel:

```
VAR RECHENFELD : ARRAY [1..10] OF REAL;
```

Es wird ein Rechenfeld vom Typ REAL angelegt, bei dem jede Komponente mit einem Index aus dem Bereich 1 bis 10 angesprochen werden kann.

Um einer bestimmten Komponente (z.B. 4) einen Wert (z.B. 3.5) zuzuordnen, ist folgende Zuweisungsform (vgl. 4.3.1) möglich:

```
RECHENFELD [4]:= 3.5;
```

Der Indexausdruck innerhalb der Zuweisung muß mit der Variablendeklaration verträglich sein (Zuweisungsverträglichkeit = assignment compatibility"), d.h der angesprochene Index muß in dem Bereich des Indextyps enthalten sein.

3.5.2.2. Satzfeldbezeichner (field-designator)

Die Komponenten eines Satztyps werden Satzfelder (record field) genannt. Im Unterschied zum Bereichstyp werden sie nicht durch Indizierung sondern über den Komponentennamen (field identifier) angesprochen.

Im Zusammenhang mit Variablendeklarationen nennt man die Komponenten Satzfeldbezeichner (field designator).

Beispiele:

a) Einfacher Satztyp

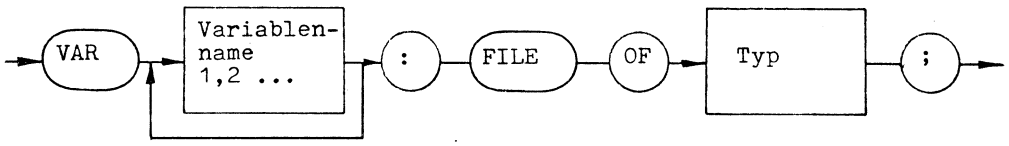
```
VAR DREIECKFLAECHE : RECORD GRUNDSEITE : REAL;
                            HOEHE      : REAL;
                     END;
Wertzuweisung:
DREIECKFLAECHE.GRUNDSEITE := 5.5;
```

b) Varianten-Satztyp

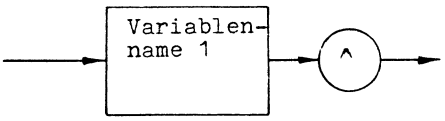
```
VAR RECHTECK : RECORD CASE FLAECHE : CHAR OF
                        'A' : (LAENGE: REAL);
                        'B' : (BREITE: REAL);
               END;
Wertzuweisung:
RECHTECK.FLAECHE := 'A';
RECHTECK.LAENGE  := 6.5;
```

3.5.2.3. Puffervariable (file-buffer)

Mit der Vereinbarung einer Datei-Variablen ist automatisch für deren Komponenten eine Puffervariable definiert.



Die Variablennamen werden als Puffervariablen bezeichnet, wenn sie die Komponenten einer sequentiellen Datei bezeichnen sollen. Sie werden mit symbolischen Namen folgender Form angesprochen:



Diese Puffervariable ist wie eine einfache Variable des Komponententyps zu behandeln. Sie dient dazu, Komponenten in eine Datei zu schreiben oder sie zu lesen.

Beispiel:

```
VAR PRIMZAHL,ZWILLINGSPRIMZAHL : FILE OF INTEGER;
```

Der Zugriff auf die Komponenten (PRIMZAHL, ZWILLINGSPRIMZAHL) erfolgt mit Hilfe der Puffervariablen, die mit den nachgestellten Zeigerzeichen bezeichnet werden (vgl. Referenzierte Variable 3.5.3).

Das Öffnen einer Datei mit "REWRITE" wird hier nicht dargestellt (vgl. 3.6.4.1).

Z.B. bei Wertzuweisung

```
PRIMZAHL^ := 3;
ZWILLINGSPRIMZAHL^ := PRIMZAHL +2;
```

Man kann jeweils nur die durch die aktuelle Datei-Position bestimmte Komponente direkt ansprechen (aktuelle Komponente).

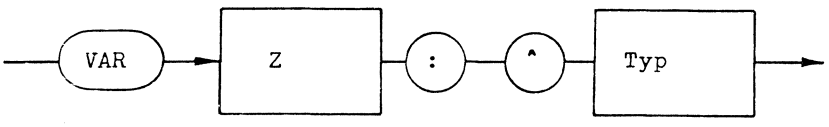
3.5.3. Zeiger-Variable (pointer variable) und Referenzierte Variable (referenced variable)

Mit Hilfe von Zeiger-Variablen können Speicherbereiche dynamisch eingeführt und freigegeben werden.

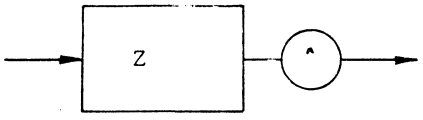
Zeiger-Variable und referenzierte Variable werden in folgender Weise dargestellt:

Die Kennzeichnung einer Variablen mit einem nachgestellten Zeigerzeichen nennt man Dereferenzierung, weil sie als "Referenzierte Variable" tatsächlich von der Zeigervariablen - also in umgekehrter Richtung - bezeichnet wird. Die Dereferenzierung einer Zeigervariablen führt demzufolge zu einer referenzierten Variablen.

Zeigervariable (Z = Name der Zeigervariablen)



Referenzierte Variable (Z = Name der Zeigervariablen)



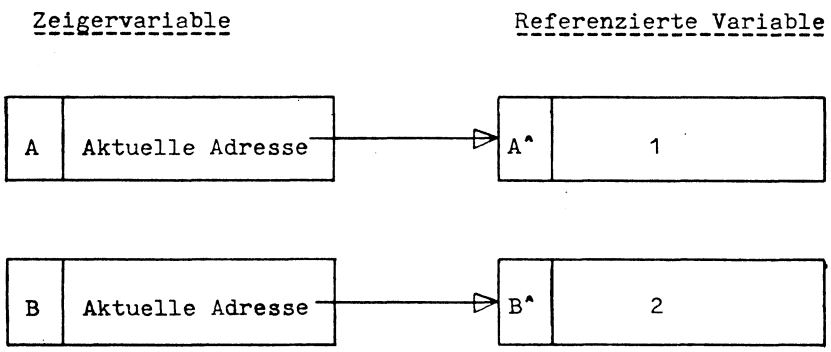
Beispiel a

```
TYPE ELEMENT = INTEGER;
VAR A,B  : ^ELEMENT;(*Zeigervariablen A und B*);
```

Das Besetzen der Zeigervariablen mit der Anweisung "NEW" wird hier nicht dargestellt (vgl. Beispiel b).

```
A^  : = 1;              (* "Referenzierte Variable" *)
B^  : = 2;              (* mit Wertzuweisung *)
```

A und B sind an den Typ ELEMENT gebundene Zeiger-Variablen, während A^ und B^ (= Referenzierte Variable) für Variable vom Typ ELEMENT stehen, auf die A und B zeigen.



Beispiel b)
Standardprozedur "NEW"

Mit der Prozedur "NEW" kann man die Anlage von Variablen im Programmablauf vornehmen.

```
TYPE     ELEMENTZEIGER =^LISTENELEMENT;
         LISTENELEMENT =RECORD OF;
                         NAME : ARRAY [0..4] OF CHAR;
                         VERKETTUNG: ELEMENTZEIGER;
                         END;
VAR      ANKER         :ELEMENTZEIGER;
BEGIN
     NEW (ANKER);
     ANKER^.NAME:='ANTON';(*ERSTNAME*)
     NEW (ANKER^.VERKETTUNG);
     ANKER^.VERKETTUNG^.NAME:='BERTA'(*ZWEITNAME*)
     NEW (ANKER^.VERKETTUNG^.VERKETTUNG);
     ANKER^.VERKETTUNG^.VERKETTUNG^.NAME:='COSTA'
                                          (*DRITTNAME*)

     END.
```

Das hier gezeigte Verkettungsprinzip beruht auf der Verbindung der Namenverknüpfung (Satzselektion) und Dereferenzierung. Damit ist es im Beispiel b) möglich, Speicherplätze für Variableninhalte während der Programmausführung zu belegen, mit ihnen zu arbeiten und sie anschließend wieder aufzufinden.

Bei den "Prozeduren für die Erzeugung dynamischer Variablen" (vgl. 3.6.4.2) gibt es einige Anwendungsmöglichkeiten mehr, die insgesamt gesehen besonders für schwierige und beliebig verzweigte Datenstrukturen infrage kommen, die im Verarbeitungszusammenhang gewartet, geändert und erweitert werden.

Um die Einsatzmöglichkeiten der Zeiger-Variablen erschöpfend darzustellen, wäre die Vorwegnahme von "Prozeduren für die Erzeugung dynamischer Variablen" (vgl. 3.6.4.2) erforderlich.

3.6. Prozedur- und Funktions-Deklarationen

Prozeduren und Funktionen sind Unterprogramme, deren Anweisungsfolge mit einem vereinbarten Namen (Prozedur- bzw. Funktionsname) angesprochen werden kann. Funktionen unterscheiden sich von Prozeduren dadurch, daß ihr Aufruf die Bildung eines einzigen Wertes 1* zur Folge hat und sie überall dort auftreten dürfen, wo Bezug auf sie genommen wird (Platzhalter 2*).

3.6.1. Prozedur-Deklarationen

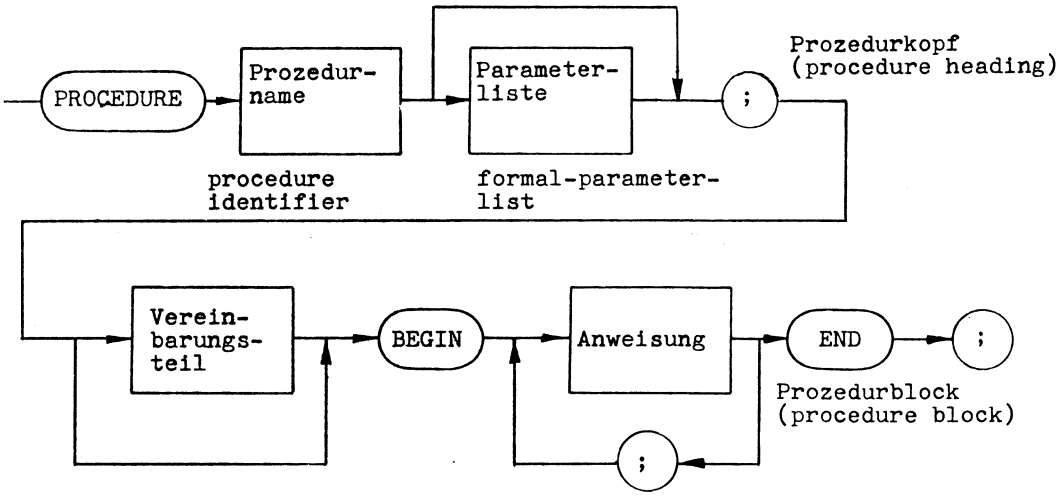
Ein Unterprogramm, das mit seinen Anweisungen nicht auf die Abgabe eines Wertes beschränkt ist sondern Wirkungen auf den Programmlauf ausüben kann, nennt man Prozedur. Eine Prozedur kann nicht wie eine Funktion als Platzhalter für einen Wert gestellt werden.

Eine Prozedurdeklaration erklärt ein Unterprogramm zur Prozedur und ordnet ihm einen Namen zu. Dieser Name ermöglicht den wiederholbaren Aufruf der Prozedur.

Prozeduren sind wie Programme aufgebaut (vgl. 1.1). Man unterscheidet einen Prozedurkopf (procedure-heading) und einen Prozedurblock (procedure-block).

1* Der Begriff "Wert" bezieht sich hier auf den Bereich der Standard- und Zeigertypen.

2* Im Zusammenhang mit der Funktion ist hier gemeint, daß der Funktionsname an den Platz eines Wertes gestellt werden kann.



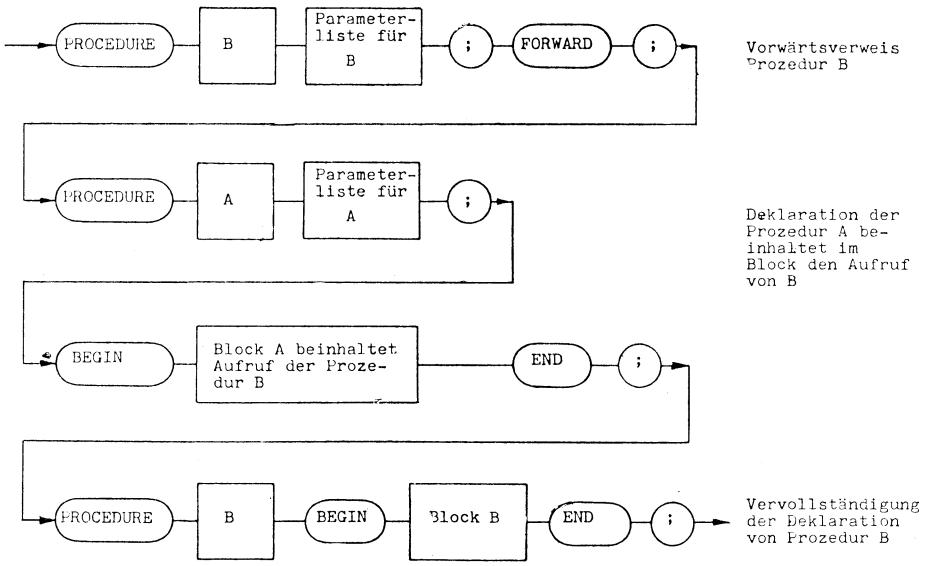
In der Parameterliste wird festgelegt, ob ein formales Argument 1* eine Variable, eine Prozedur oder eine Funktion darstellt (vgl. 3.6.3). Ein "formales Argument" bezeichnet einen symbolischen Platzhalter, für den beim Aufruf der Prozedur ein "aktuelles Argument" 1* eingesetzt wird. Daher muß die Parameterliste mit dem Prozedurnamen zusammen benannt werden.

Innerhalb des Prozedurblocks werden Verarbeitungsalgorithmen aus Anweisungsfolgen gebildet, die beim Prozeduraufruf aktiviert werden und unter Einsatz der aktuellen Argumente vorgesehene Aufgabenstellungen lösen.

1* Der hier verwendete Begriff "Argument" lehnt sich an den mathematischen Sprachgebrauch bei Funktionen an. Bei einer Funktion y=f(x) stellt x ein formales Argument (unabhängige Veränderliche) dar, von dem y abhängt. Nimmt x einen konkreten Wert an, spricht man von aktuellem Argument.

Dabei ist es möglich, daß eine Prozedur sich selbst aufruft (rekursiver Aufruf) und innerhalb ihrer eigenen Ausführung nochmals abläuft.

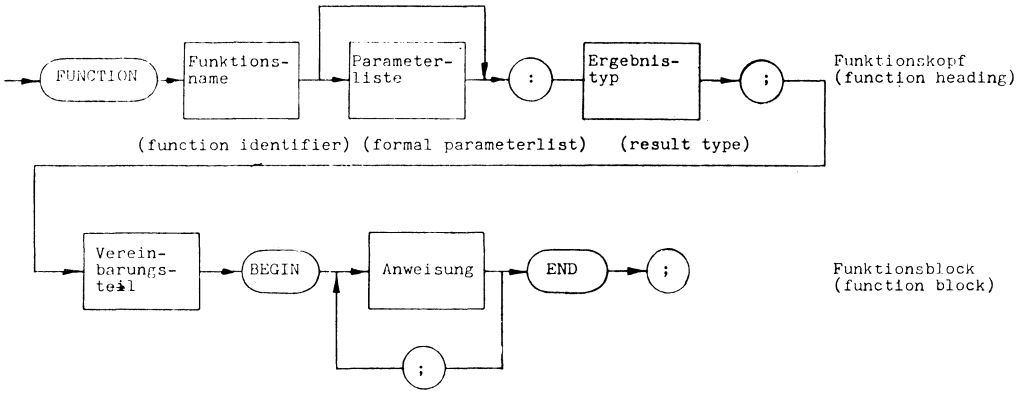
Eine Variante von Prozedurvereinbarungen ist die FORWARD - Deklaration. Hierbei wird innerhalb einer Prozedur A eine Prozedur B verwendet, die mit FORWARD angekündigt und später erst vervollständigt wird. Dieser Vorwärtsverweis muß die Parameterliste für die Prozedur B beinhalten; sie darf nicht in einem tiefer geschachtelten Block stehen.



Der Vorwärtsverweis und die spätere Vervollständigung einer solchen Prozedur müssen im gleichen Block stehen. FORWARD - Deklarationen sind für solche Fälle geeignet, in denen Prozeduren sich gegenseitig aufrufen sollen. Diese Verschachtelungsmöglichkeiten vermeiden erheblichen Programmieraufwand, weil ohne den Verweis auf eine andere Prozedur der entsprechende Algorithmus nochmals wiederholt werden müßte.

3.6.2. Funktions-Deklarationen

Funktionen bezeichnen Unterprogramme, die einen Wert berechnen und als Platzhalter dafür eingesetzt werden können. Sinngemäß entspricht der Aufbau einer Funktion der Prozedur; abweichend ist die Benennung eines Ergebnistyps (result types), der angibt, von welchem Typ der abzugebende Wert ist. Zugelassen sind einfache Typen (simple type) oder Zeigertypen (pointer type).



Innerhalb des Funktionsblocks muß dem Funktionsnamen der Funktionsinhalt (Wert, Algorithmus) zugewiesen werden; hierbei bewirkt die Benennung des Funktionsnamens keinen Funktionsaufruf.

Z.B.: Wenn CONST PI=3,14159; und VAR R:REAL; sind,
lautet die Zuweisung für die Funktion KREISFLAECHE:
KREISFLAECHE := R*R*PI;

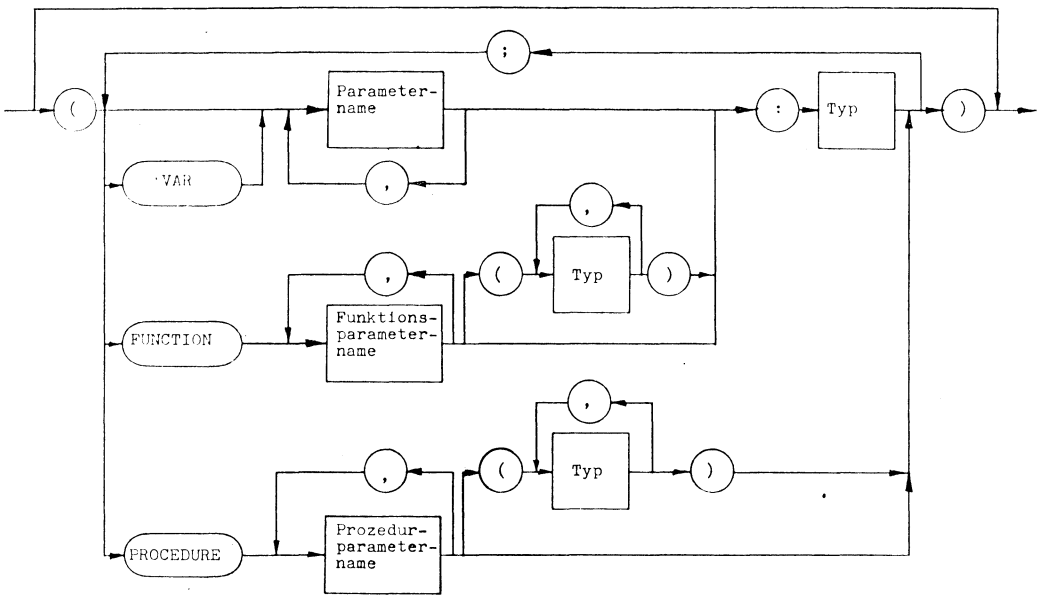
Vor dieser Zuweisung gilt der Funktionswert als unbestimmt.

Auch bei Funktionen ist die "Rekursion" möglich. Das gilt ebenso für FORWARD-Deklarationen; bei der nachfolgenden Funktionsvervollständigung darf der Ergebnistyp nicht mehr aufgeführt werden.

3.6.3. Parameter

Bei der Deklaration von Prozeduren und Funktionen werden zunächst die "formalen Argumente" benannt. Im Hinblick auf die Parameterliste unterscheidet man daher die "Formale Parameterliste" (formal-parameter-list) und "Aktuelle Parameter" (actual parameter), die den jeweils im Verarbeitungszusammenhang tatsächlich zugewiesenen Inhalten entsprechen.

Die formale Parameterliste (formal-parameter-list) hat folgenden grundlegenden Aufbau:



In der Darstellung sind die vier Arten von Parametern dargestellt:

3.6.3.1. Wertparameter (value-parameter)

Eine Parametergruppe (parameter group) ohne vorangehendes Schlüsselwort (z.B. VAR), entspricht einer Liste von Wertparametern, die aus Parameternamen mit Typ-Definitionen bestehen. Der formale Parameter stellt eine lokale Vereinbarung für den Block der dazugehörigen Prozedur oder Funktion dar.

Der aktuelle Parameter muß ein Ausdruck (expression) sein (vgl. 4.1). Ausdrücke stellen mathematische Aufgabenstellungen (z.B. Formeln) dar, die aus Operanden (Konstanten, Variablen und Funktionen) sowie Operatoren (Zeichen zur Darstellung und Auslösung mathematischer Regeln und Operationen) bestehen und neue Werte des bestimmten Typs erzeugen.

Beispiel:

```
PROGRAM TESTA;
      VAR A,B : INTEGER;
  .
  .
  PROCEDURE X (Y:REAL);(*FORMALER PARAMETER*)
  .
  .
  BEGIN
        .
        .
        A:=1;
        B:=Y+1;
        X(A/B);    (*AKTUELLER PARAMETER*)
        .
        .
  END;
BEGIN
  .
  .
END.
```

Bei Aufruf einer Prozedur oder Funktion (im Beispiel: rekursiv) wird der aktuelle Wert des Ausdrucks zugewiesen. Er muß dem formalen Parameter angepaßt sein, d.h. im Sinne der Typendeklaration zuordnungsverträglich (vgl. 3.3.5) sein. Anzahl und Reihenfolge der aktuellen Parameter entsprechen den formalen.

3.6.3.2. Variablenparameter (variable parameter)

Hierunter wird eine Parametergruppe mit dem Schlüsselwort "VAR" verstanden; sie stellt eine lokale Vereinbarung für den Block der dazugehörigen Prozedur oder Funktion dar.

Der aktuelle Parameter muß eine Variable sein. Jede Operation, die für den formalen Parameter vorgesehen ist, erfolgt im Verarbeitungszusammenhang mit dem aktuellen Parameter. Hinsichtlich der Zuordnungsverträglichkeit (vgl. 3.4.5) muß Typidentität gegeben sein. Anzahl und Reihenfolge der aktuellen Parameter entsprechen den formalen.

Beispiel:

```
PROGRAM TESTB (INPUT, OUTPUT);
        VAR X0,X2:REAL;X1:INTEGER;
        FUNKTION Z(VAR C:INTEGER;VAR D:REAL):REAL;
        (*FORMALE PARAMETER*)
        .
        .
                 BEGIN
                 .
                 .
                 END;
        BEGIN     (*HAUPTPROGRAMM*)
        X0 := 1;
        .
        .
        X1 := 3;
        X2 : X0 * 2;
        X0 := Z (X1,X2);  (*AKTUELLE PARAMETER*)
        .
        .
        END.
```

3.6.3.3. Prozedur-Parameter

Die formalen Parameter bezeichnen eine oder mehrere Prozeduren mit Namen, die von den aktuellen Prozeduren während der Aktivierung des aufzurufenden Blocks repräsentiert werden.

Abweichend vom Standard-PASCAL ist es in der hier beschriebenen Implementierung erforderlich, hinter dem Prozedurparameternamen Typen (Typenliste) aufzuführen, die für formale Parameter (formale formale Parameter) der einzusetzenden Prozeduren stehen. So ist bereits zur Übersetzungszeit eine vollständige Typprüfung der Parameter möglich.

Prozeduren, die Parameter von anderen Prozeduren und Funktionen sind, dürfen nur Wertparameter beinhalten. Hinsichtlich der Zuordnungsverträglichkeit (vgl. 3.3.5) formaler und aktueller Parameter muß Typidentität gegeben sein. Anzahl und Reihenfolge der aktuellen Parameter entsprechen den formalen.

Beispiel:

```
PROGRAM TESTC;
  TYPE T = 0..9;
.
.
  PROCEDURE P1 (X:T; Y:REAL); (*WERTPARAMETER:
                               TYPIDENTITAET MIT DER
                               TYPENLISTE VON
                               PROZEDUR PX*)
    BEGIN
    .
    .
    END;
  PROCEDUR P2 (XX:T; YY:REAL); (*WERTPARAMTER:
    .                           TYPIDENTITAET MIT DER
    .                           TYPENLISTE VON
                                PROZEDUR PX*)
    BEGIN
    .
    .
    END;
  PROCEDURE P3 (PROCEDURE PX (T,REAL)); (*FORMALE
                                          PROZEDUR PX*)
               (*DIE TYPEN T UND REAL HINTER DER
               FORMALEN PROZEDUR PX STELLEN DIE
               TYPENLISTE FUER DIE "FORMALEN FORMALEN
               PARAMETER" DER EINZUSETZENDEN
               PROZEDUREN P1 und P2 DAR*)
  VAR V1:T; V2:REAL;
  BEGIN
  .
  .
  V1 := 2;
  V2 := 1.5;
  PX (V1, V2);
  .
  .
  END;
  BEGIN (*HAUPTPROGRAMM*)
  .
  .
  P3 (P1);                  (*AKTUELLE PROZEDUR P1*)
  P3 (P2);                  (*AKTUELLE PROZEDUR P2*)
  .
  .
  END.
```

3.6.3.4. Funktions-Parameter

Die formalen Parameter bezeichnen eine oder mehrere Funktionen mit Namen, die von den aktuellen Funktionen während der Aktivierung des aufzurufenden Blocks repräsentiert werden.

Abweichend von Standard-PASCAL ist es in der hier beschriebenen Implementierung möglich, hinter dem Funktionsparameternamen Typen (Typenliste) aufzuführen,die für formale Parameter(formale formale Parameter)der einzusetzenden Funktionen stehen.
So ist bereits zur Übersetzungszeit eine vollständige Überprüfung der Parameterübergabe möglich.

Funktionen, die Parameter von anderen Prozeduren und Funktionen sind, dürfen nur Wertparameter beinhalten. Der aktuelle Parameter muß ein Funktionsname sein.

Aktuelle und formale Funktion müssen identische Parameter- und Ergebnis-Typen haben (vgl. 3.4.5).

Beispiel

```
(*ES GELTEN ENTSPRECHENDE ANFORDERUNGEN WIE IM
PROZEDURBEISPIEL.*)

PROGRAM TESTD;
.
.
FUNCTION F1 (X,Y: INTEGER) : REAL;
.
.
     BEGIN
     .
     .
     END;
FUNCTION F2 (FUNCTION FX(INTEGER,INTEGER):REAL;
            (*FORMALE FUNKTION FX*)
     VAR V10,V11:REAL;
     BEGIN
         V10      :=  1;
         V11      :=  3;
         FX (V10,V11);
         .
         .
     END;
     BEGIN (*HAUPTPROGRAMM*)
        .
        .
        F2 (F1);        (*AKTUELLE FUNKTION F1*)
        .
        .
     END.
```

3.6.3.5. Gegenüberstellung der Parameterübergabe-Formen

Wertübergabe (call by value)

Daten können nur aus dem Hauptprogramm an die Prozedur oder Funktion abgegeben werden (vgl. 3.6.3.1).

Bei Parametern strukturierten Typs (z.B. Satz-Typ) wächst die Übergabezeit mit dem Umfang der Speicherbelegung, da das gesamte Datenfeld in die Funktion bzw. in die Prozedur kopiert wird. Das wirkt sich nachteilig auf die Laufzeit und die Speicherplatzbelegung aus.

Adressübergabe (call by reference)

Bei Variablenparametern (vgl. 3.6.3.2) werden demgegenüber keine Speicherinhalte übergeben sondern die Adressen der aktuellen Parameter.
Die Daten können sowohl vom aufrufenden zum gerufenen Programm wie umgekehrt übergeben werden (bidirektionale Übergabe). Die Veränderung globaler Daten ist zu vermeiden.

Namensübergabe (call by name)

Es ist möglich, beliebige Funktionen als aktuelle Parameter an die Stelle des formalen Funktionsnamens einzusetzen, sofern die Zuordnungsverträglichkeit (identische Parameter- und Ergebnistypen) gewährleistet ist (vgl. 3.6.3.4).

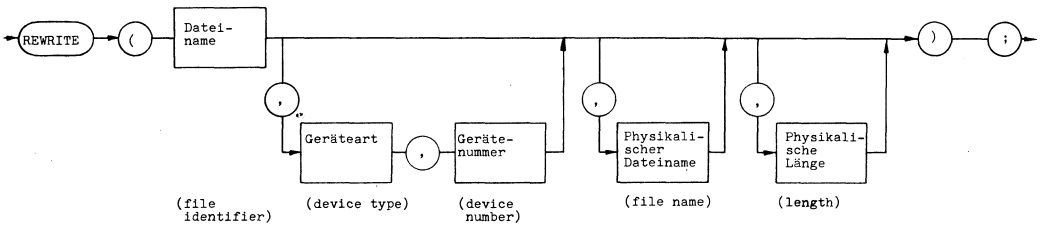
3.6.4. Standard-Prozeduren

Bestimmte, häufiger einsetzbare Anweisungsfolgen sind in PASCAL standardmäßig vorgesehen; sie gelten in jedem Programm als deklarierte Prozeduren. Standard-Prozeduren dürfen nicht als aktuelle Prozedur-Parameter übergeben werden.

3.6.4.1. Prozeduren zur Dateiverarbeitung (file - handling - procedures)

Die Prozeduren zur Dateiverarbeitung lassen sich von ihrer Funktionsweise her in zwei Bereiche gliedern:

- Dateiverwaltung
- Dateizugriff



Prozeduren zur Dateiverwaltung:

a) Öffnen einer Datei für Schreibzugriff

Beim Geräteeinsatz unterscheidet man

für Geräteart (device type):	
0	SLI-Gerät (seriell-line-interface) z.B. Drucker)
1	Platte
2	Schließen Datei
Keine Angabe (Default)	Es wird "1" (= Platte) eingesetzt

für Gerätenummer (device number):
Es wird jeweils die logische Gerätenummer (Device-Nr.) angeben. Wenn die Angabe fehlt, (default) wird das Gerät "0" eingesetzt.

Wenn andere Geräte als die Platte "0" verwendet werden sollen, dann müssen Geräteart- und nummer angegeben werden.

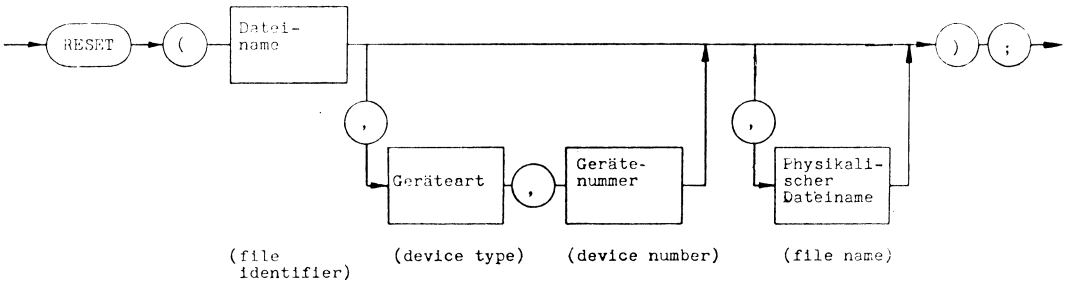
Die Prozedur REWRITE öffnet eine Datei für den Schreibzugriff und muß vor Beschreiben ausgeführt werden. Ist die Datei bereits vorhanden, wird sie gelöscht und neu angelegt; anderenfalls wird die Datei nur angelegt.

Nach Ausführung von REWRITE hat die Standard-Funktion EOF (Dateiname) (vgl. 3.6.4.3) den Wert TRUE. Die Prozedur REWRITE darf nicht auf die Standard-Dateien INPUT und OUTPUT angewendet werden (vgl. 6).

Die physikalische Länge wird in Sektoren (1 Sektor = 128 Byte) angegeben. Fehlt diese Angabe, wird eine Datei mit 512 Sektoren physikalischer Länge angelegt. Maximal sind 65535 Sektoren möglich. Der physikalische Dateiname muß 6 Zeichen lang sein; er kann als Variable vereinbart werden oder unmittelbar als Konstante (in Hochkommata) eingesetzt werden.

Die Parameter (Geräteart, Gerätenummer, Physikalischer Dateiname, Physikalische Länge) werden vom Compiler darauf überprüft, ob es sich um "Ordnungstypen" bzw. beim physikalischen Dateinamen um eine "6-Byte lange Zeichenkette" handelt. Darüberhinaus muß der Programmierer auf die Zulässigkeit der Ziffern und Zeichen achten.

b) Öffnen einer Datei für Lesezugriff



Mit der Prozedur RESET (Dateiname) wird die aktuelle Lese- Position auf Dateianfang gesetzt und der Wert der ersten Dateikomponente der Puffervariablen zugewiesen. Im Gegensatz zu REWRITE (Dateiname) wird sofort implizit gelesen (Ausnahme Textdateien, vgl. 6); beim ersten Zugriff ist eine gesonderte Leseprozedur somit nicht erforderlich.

Im gleichen Zusammenhang wird die Standardfunktion EOF (Dateiname) (vgl. 3.6.4.6 Prädikate) im booleschen Sinn auf unwahr (=FALSE) gesetzt. Bei leerer Datei bleibt die Puffervariable undefiniert und EOF ist wahr (=TRUE).

Für die Parameter (Geräteart, Gerätenummer, Physikalischer Dateiname) gelten die gleichen Vereinbarungen wie beim REWRITE. Die Angabe der physikalischen Länge (length) ist bei "RESET" nicht vorgesehen.

Bei Textdateien (vgl. 6) ist der Lesezugriff abweichend implementiert. Die Puffervariable erhält keine automatische Zuweisung. RESET (Dateiname) setzt die aktuelle Position auf den Dateianfang und SOR (Dateiname) auf "TRUE" (vgl. 3.6.4.6 Prädikate).
Die Anwendung der Prozedur RESET ist bei den Standard-Dateien INPUT und OUTPUT verboten.

Prozeduren zum Dateizugriff:
---

a) Schreibzugriff

Mit der Prozedur PUT (Dateiname) wird der unmittelbare Schreibvorgang ausgelöst, sofern er mit REWRITE eingeleitet wurde (zwingende Voraussetzung).

Wenn vor der Ausführung von PUT die Puffervariable am Dateiende "Prädikat EOF (Dateiname) ist wahr" (vgl. 3.6.4.6) steht, so wird der Wert der Puffervariablen hinzugefügt.

Beispiel:

```
PROGRAM SCHREIBEN (INPUT,BEWERBER);
TYPE AUSBILDUNG = RECORD SCHULBILDUNG:ARRAY [1..50]
                  OF CHAR;
                        BERUFSBILDUNG:ARRAY [1..50]
                  OF CHAR;
                END;
VAR BEWERBER : FILE OF AUSBILDUNG;
    PERSON   : AUSBILDUNG;
BEGIN
   .
   .
   REWRITE (BEWERBER);
   BEWERBER ^ := PERSON;
   PUT (BEWERBER);
.
.
END.
```

Der Inhalt der Puffervariablen BEWERBER steht jetzt am Ende der Datei BEWERBER. Nach Durchführung dieser Standardprozedur ist der Wert der Puffervariablen unbestimmt, während das Prädikat EOF (Dateiname) für die erweiterte Datei wiederum wahr ist.

Falls EOF (Dateiname) vor der Ausführung falsch ist, erfolgt das Setzen des Prädikats.

IOERROR (Dateiname) auf TRUE (vgl. 3.6.4.6)

Hiermit werden Eingabe-, Ausgabefehler gekennzeichnet (input - output - error), die vom Programmierer abgefragt werden können.

Bei Textdateien (vgl. 6) wird mit der Prozedur BREAK (Dateiname) eine unmittelbare Ausgabe des Puffers bewirkt. Das gilt sowohl bei Magnetplatten wie bei SLI-Geräten. Diese Möglichkeit ist bei Standard-PASCAL nicht vorgesehen.

b) Lesezugriff

Mit der Prozedur GET (Dateiname) wird der unmittelbare Lesevorgang (Puffervariable erhält Zuweisung) ausgelöst, sofern ein "RESET" vorher erfolgt ist (zwingende Voraussetzung).

Da der erste Lesevorgang im "RESET" eingeschlossen ist, wird die aktuelle Datei-Position auf die nächste Komponente gesetzt und der Wert dieser Komponente der Puffer-Variablen zugewiesen. Voraussetzung ist, daß EOF (Dateiname) vor der Ausführung von "GET" unwahr (=FALSE) ist (vgl. 3.5.5.3 Prädikate).

Falls EOF (Dateiname) = TRUE ist, erfolgt das Setzen des Prädikats.

IOERROR (Dateiname) auf TRUE (vgl. 3.6.4.6)

Hiermit werden Eingabe-, Ausgabefehler gekennzeichnet (input-, output-error), die vom Programmierer abgefragt werden können. Am Dateiende wird damit der Lesevorgang abgeschlossen.

Bei Textdateien, die zum Lesezugriff eröffnet werden, muß die Zuweisung der ersten Komponente an die Puffervariable mit einem "GET" erfolgen, das dann auch SOR (F) zurücksetzt (=FALSE) (vgl. 3.6.4.6).

Beispiel:

In diesem Beispiel wird angedeutet, wie man mit den Dateiverwaltungsprozeduren arbeiten kann.

Es geht darum, daß eine Datei alphabetisch in eine andere sortiert wird. Folgende Dateien liegen auf der Platte vor und haben die physikalischen Dateinamen KUNDEN, LIEFER und PERSON. Die Dateien KUNDEN und LIEFER sollen in vorgesehener Sortierung nach PERSON gebracht werden.

Innerhalb des Sortierprogrammes TSORT soll eine frei vereinbarte Prozedur SORTIER diese Aufgabe erfüllen. Sie wird hier nicht näher beschrieben. Sie verwendet folgende Dateien:

	logischer Dateiname	physikalischer Dateiname
Quelldatei	QUELLE	ABGABE
Zieldatei	ZIEL	ERHALT

Der Einfachheit halber wird angenommen, daß die Besonderheiten der Behandlung von Textdateien innerhalb der Prozedur SORTIER enthalten seien. Das betrifft die Abfrage SOR (DATEINAME) = TRUE und das zusätzlich erforderliche GET (DATEINAMEN).

```
PROGRAM TESTSORT (QUELLE, ZIEL);
.
.
.
   TYPE ADRESSE = RECORD NAMEN : ARRAY [1..40] OF CHAR;
                         ANSCHRIFT: ARRAY [1..60] OF CHAR;
                  END;
   VAR QUELLE,ZIEL : FILE OF ADRESSE;

       BEGIN
       .
       .
       .
          RESET (QUELLE,1,0,'KUNDEN');
          REWRITE (ZIEL,1,0,'PERSON');
          SORTIER (QUELLE,ZIEL);

(* HIERBEI WIRD DEUTLICH, WIE MAN DEN PARAMETER
 "PHYSIKALISCHER DATEINAME" FUER BESTIMMTE
 VERARBEITUNGEN NUTZEN KANN! *)

          RESET (QUELLE,1,0,'LIEFER');
          REWRITE (ZIEL,1,0,'PERSON');
          SORTIER (QUELLE,ZIEL);
       .
       .
       .
      END.
```

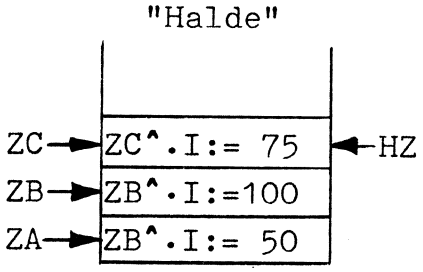
3.6.4.2. Prozeduren zur Erzeugung dynamischer Variablen

Bei der Besprechung von Zeiger-Variablen und Referenzierten Variablen (vgl. 3.4.3) wurde die Anlage von dynamischen Variablen bereits dargestellt.

a) Anlage von dynamischen Variablen

Die Standardprozedur NEW (Zeigervariable) legt auf der Halde (heap) Platz für eine "Referenzierte Variable" an und weist der dazugehörigen Zeigervariablen deren Adresse zu.

Unter Halde (heap) ist ein Speicherbereich für die Ablage von Variablen zu verstehen, deren Lebensdauer nur dynamisch bestimmt ist.



Die "Referenzierten Variablen" (ZA^,ZB^,ZC^) werden in der Reihenfolge ihrer Erzeugung abgelegt. Beliebiger Zugriff ist über im Programm vereinbarte Zeigervariablen (ZA,ZB,ZC) möglich.

Darüberhinaus gibt es einen "Haldenzeiger" (HZ), der vom PASCAL-Laufzeitsystem verwaltet wird und jeweils auf die zuletzt angelegte Variable zeigt.

Beispiel:

```
            .
            .
            .
TYPE T = RECORD I : INTEGER;
         END;
VAR ZA,ZB,ZC : ^T;

NEW (ZA);  (* ADRESSE FUER
           "REFERENZIERTE VARIABLE"
           WIRD ZUGEWIESEN UND ALS PLATZ AUF DER
           "HALDE" ANGELEGT. *)

ZA^.I : = 50; (* ZUWEISUNG AN DIE "REFERENZIERTE
                 VARIABLE" *)

NEW (ZB); ZB^.I := 100;
NEW (ZC); ZC^.I :=  75;
          .
          .
          .
```

Anlage von Varianten-Satztyp-Variablen

Weiterhin ist es möglich, Satztypvarianten (vgl. 3.4.2.2) zu erzeugen. Die Auswahlmarken müssen in der Reihenfolge ihrer Definition aufgeführt werden. Sie können von hinten nach vorn weggelassen werden und erhalten dann den Wert "NIL". Dieses Schlüsselwort besagt, daß die Zeigervariable auf keine Variable gerichtet ist und stellt insofern eine Standardkonstante dar.

Beispiel:

```
TYPE ST   = RECORD CASE BO : BOOLEAN OF
                   TRUE : (A,B,C : CHAR);
                   FALSE: (X,Y,Z : INTEGER);
            END;
VAR VSZ   : ST;
.
.
NEW (VSZ, TRUE);
VSZ^.A := 'A';
.
.
```

Anlage von Bereichstyp-Variablen

Wenn die "Referenzierte Variable" ein Bereichstyp (vgl. 3.4.2.1) ist, gilt folgende Schreibweise (in Standard-PASCAL nicht vorgesehen):

```
  .
  .
  TYPE BT = ARRAY [0..99] OF INTEGER;
  VAR VBZ = ^BT;
  .
  .
  NEW (VBZ,50); (*MIT DIESER DARSTELLUNG WIRD
                  DER URSPRUENGLICH DEFINIERTE
                  BEREICH EINGESCHRAENKT *)
  .
  .
  VBZ^ [45] := 250; (*ZUWEISUNG AN DER STELLE 45*)
  .
  .
```

Analog dazu ist die Konstruktion von mehrdimensionalen Bereichs-Variablen:

```
.
.
TYPE MBT = ARRAY [0..99, 0..49] OF INTEGER;
           (*ZWEIDIMENSIONALER BEREICHSTYP*)
VAR VTAB : ^MBT;
           (*ENTSPRICHT EINER TABELLE MIT
            100 ZEILEN UND 50 SPALTEN BZW.
            50 ZEILEN UND 100 SPALTEN.*)
BEGIN
NEW (VTAB);
VTAB^ [1,5] := 1600; (*ZUWEISUNG IN ZEILE 1,
                       SPALTE 5*)
.
.
```

Es ist darauf hinzuweisen, daß die mit "NEW" erzeugten "Referenzierten Variablen" vom Varianten-Satztyp und vom Bereichstyp nicht als aktuelle Parameter übergeben werden dürfen. Auch können sie nicht als Variable in einer Zuweisung oder als Operand in einem Ausdruck stehen. Im Übersetzer kann keine Prüfung erfolgen, die feststellt, mit welchen Komponenten die Variable erzeugt wurde.

Wohlgemerkt kann ein durch Dereferenzierung, Indizierung und Selektion bestimmter Teil einer erzeugten Variablen sehr wohl diese Anforderungen erfüllen - nur die Variable insgesamt nicht.

BEISPIEL FÜR DIE ERZEUGUNG DYNAMISCHER VARIABLEN

Unter Zeigervariablen (vgl. 3.5.3) haben wir als Beispiel b) für die Erzeugung dynamischer Variablen "Mehrfachdeklarationen" gebracht. Umständlich ist dort die langwierige Verkettung durch Namensverknüpfung (Satzselektoren) und Derefernzierung. Im folgenden wird das gleiche Beispiel mit einer verkürzten Konstruktion gebracht:

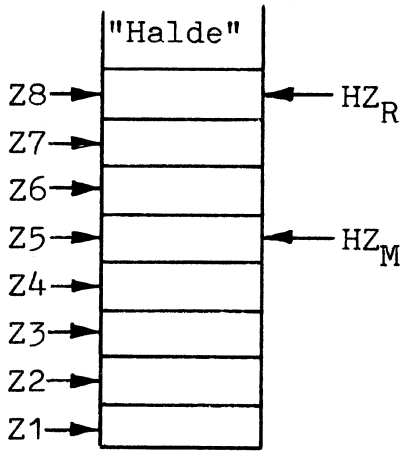
```
PROGRAM MEHRFACHDEKLARATION;
TYPE ELEMENTZEIGER = ^LISTENELEMENT;
     LISTENELEMENT = RECORD
                       NAME:ARRAY [0..4]OF CHAR;
                       VERKETTUNG:ELEMENTZEIGER;
                       END;
VAR ANKER : ELEMENTZEIGER;
    HILFA : ELEMENTZEIGER;
    HILFB : ELEMENTZEIGER;
            (* DIE VARIABLEN "HILFA" UND "HILFB"
             DIENEN ALS ZWISCHENSPEICHER FUER
             EINE PERMANENTE VERKETTUNG. *)
BEGIN
     NEW (ANKER);
     ANKER^.NAME:= 'ANTON';
     NEW (HILFA);
     HILFA^.NAME:= 'BERTA';
     ANKER^.VERKETTUNG:=HILFA; (*VERKETTUNG*)
     NEW (HILFB);
     HILFB^.NAME:='COSTA';
     HILFA^.VERKETTUNG:=HILFB; (*VERKETTUNG*)
     HILFA:= HILFB; (*UMBESETZUNG*)
     NEW (HILFB);
     HILFB.NAME:='DORIS';
     HILFA^.VERKETTUNG:=HILFB; (*VERKETTUNG*)
                        (* MIT DER UMBESETZUNG KANN
                         DIE EINFUEHRUNG NEUER
                         VARIABLEN BELIEBIG WIEDER-
                         HOLT WERDEN *)
     HILFB^.VERKETTUNG:= NIL; (*ENDE-BEDINGUNG*)
     .
     .
     END;
```

Die im Beispiel dargestellte Erzeugung von Variablen ist insofern unüblich, als im wirklichen Anwendungsfall die Namen eingelesen werden und der Erzeugungsvorgang mit einer Schleife erfolgt. Es kam hierauf an, das Prinzipielle bei dem Einsatz der Standardprozedur "NEW" zu verdeutlichen.

b) Haldenverwaltung

Die ständige Generierung neuer Variablen kann zu einem Überlauf führen, weil angelegte Variablen auch dann weiterbestehen, wenn sie nicht mehr gebraucht werden.

Um diesen Platz im Programm mit anderen Variablen belegen zu können, bedarf es einer Speicherbereinigung. Der hier gewählte Weg legt bereits bei der Deklaration einer Variablen fest, ob und wann der Speicherplatz einer Variablen freizugeben ist.



MARK (Z5);

Diese Prozedur weist den Stand des Haldenzeigers (HZM) der Zeigervariablen (Z5) die vom Typ Ganzzahl (INTEGER) sein muß, zu. D.h., von dieser Adresse an können die "Referenzierten Variablen" Z5 bis Z8 freigegeben werden.

RELEASE (Z8);

Diese Prozedur gibt die Adressen Z5 bis Z8 frei; sie werden bei der Erzeugung neuer Variablen wieder vergeben, die Inhalte der ursprünglich "Referenzierten Variablen" sind dann nicht mehr verfügbar.

Geht einem "RELEASE" kein "MARK" voraus, werden alle Adressen der Halde freigegeben. Der Übersetzer prüft nicht, ob ein "MARK" vorangegangen ist.

Diese Prozeduren wurden anstelle der in Standard-PASCAL vorgesehenen Prozedur DISPOSE implementiert. Da die Programmlogik oftmals nicht ohne weiteres bis ins Letzte einsichtig wird, ist Vorsicht geboten. Es muß exakt überprüft werden, ob angelegte Variable tatsächlich nicht mehr im Programmablauf benötigt werden.

3.6.4.3. Standardfunktionen

Bestimmte, häufiger einsetzbare Funktionen sind in PASCAL standardmäßig vorgesehen; sie gelten in jedem Programm als deklarierte Standardfunktionen und dürfen nicht als aktuelle Funktions-Parameter übergeben werden.

3.6.4.4. Arithmetische Funktionen

Es handelt sich um Standardfunktionen, die mathematische Funktionen lösen oder Werte eines Typs in Werte eines anderen Typs umwandeln. Folgende "arithmetische Funktionen" sind vorgesehen:

Funktion	Parameter-typ	Ergebnis-typ	Bemerkung
ABS(X)	INTEGER REAL	INTEGER REAL	Absolutwert von X
SQR(X)	INTEGER REAL	INTEGER REAL	X^2
SIN(X)	INTEGER REAL	REAL	X im Bogenmaß
COS(X)	INTEGER REAL	REAL	X im Bogenmaß
EXP(X)	INTEGER REAL	REAL	e^x = (2,718281828 ^{x})
LN(X)	INTEGER REAL	REAL	Logarithmus zur Basis e bei x > 0
SQRT(X)	INTEGER REAL	REAL	Quadratwurzel ($\sqrt[2]{X}$) bei x > 0
ARCTAN(X)	INTEGER REAL	REAL	Arcustangens von X

Der Benutzer kann diese Standardfunktionen mit ihrem Namen aufrufen und einsetzen, ohne daß zusätzliche Vereinbarungen getroffen werden müssen.

3.6.4.5. Funktionen für Ganzzahlen

Diesen Standardfunktionen ist gemeinsam, daß ihr Ergebnistyp mit einer Ausnahme immer "INTEGER" ist. Bei ORD(X) muß der Parametertyp "INTEGER" sein.

Funktion	Parametertyp	Bemerkung
TRUNC(X)	REAL (INTEGER ist möglich, aber nicht sinnvoll)	Es wird der ganzzahlige Anteil von X geliefert/z.B.TRUNC (6.3) =6. Bei eingeschalteter Kontrollfunktion (check option) erfolgt eine Fehlermeldung, wenn der Bereich der darstellbaren Ganzzahlen (MININT bis MAXINT) überschritten wird.
ROUND(X)	REAL (INTEGER ist möglich, aber nicht sinnvoll)	Es wird im mathematischen Sinne auf- bzw. abgerundet. Z.B.: ROUND (3.6) = 4 ROUND (3.4) = 3 Tatsächlich setzt der Compiler den Aufruf in eine "TRUNC"-Funktion um: Für X > = 0 gilt ROUND (X) = TRUNC (X+0.5) z.B.: X = 3.6 TRUNC (3.6 + 0.5) = 4 X = 3.4 TRUNC (3.4 + 0.5) = 3 Für X < 0 gilt ROUND (X) = TRUNC (X-0.5) z.B.: X = -4.2 TRUNC (-4.2 - 0.5) =-4

Funktion	Parametertyp	Bemerkung
ORD(X)		X muß ein Ausdruck von einem Ordnungstyp sein. INTEGER ist möglich, aber nicht sinnvoll.
	BOOLEAN	Für FALSE wird 0, für TRUE 1 ausgedrückt. z.B. ORD(FALSE) = 0
	CHAR	Das Ergebnis ist der dezimale Ganzzahlwert vom ASCII-CODE. z.B. ORD('A') = 65
	Aufzählungs-typen	Man erhält Ganzzahlen, die sich aus der Reihenfolge der Vereinbarung ausgehend von 0 positiv zählend ergeben.

z.B.:

```
          .
          .
VAR FAMILIENSTAND : (LEDIG,VERHEIRATET
                     GESCHIEDEN,VERWITWET);
    EHESTAND      : 0..3;
BEGIN
    .
    .
    FAMILIENSTAND := GESCHIEDEN;
EHESTAND := ORD (FAMILIENSTAND);

(* DIE GANZZAHL 2 WURDE DER VARIABLEN
   EHESTAND ZUGEWIESEN *)
.
.
END.
```

Funktion	Parametertyp	Bemerkung
CHR(X)	INTEGER	Es wird ein ganzzahliger Wert von 0 bis 255 erwartet. Bei eingeschalteter Kontrollfunktion (check option) erfolgt eine Fehlermeldung, wenn der Parameter nicht in diesem Bereich liegt. Als Ergebnis wird das entsprechende ASCII-Code-Zeichen bereitgestellt. z.B. CHR(65) ='A' Insoweit entspricht diese Funktion der Umkehrung von ORD(X) beim Parametertyp "CHAR".
SUCC(X)	INTEGER BOOLEAN CHAR Aufzählungs-typen	Der eingegebene Ausdruck vom bezeichneten Ordnungstyp wird um 1 erhöht und ergibt so die Nachfolgezahl in der Zählreihe (Nachfolger/successor). z.B. SUCC('A') = 'B'
PRED(X)	INTEGER BOOLEAN CHAR Aufzählungs-typen	Der eingegebene Ausdruck vom bezeichneten Ordnungstyp wird um 1 reduziert und ergibt so den Vorgänger (predecessor) in der Zählreihe. . z.B. PRED(TRUE) = FALSE

3.6.4.6. Prädikate (predicates)

Es handelt sich um Standardfunktionen, die eine Aussage (Prädikat) im Sinne der zugrundeliegenden Logik liefern (Prädikatenlogik). Sie werden daher auch "logische Standardfunktionen" genannt. Dabei werden in einem Vergleich bestimmte Bedingungen geprüft und mit dem "booleschen Prädikat" "TRUE" angezeigt, wenn sie zutreffen; anderenfalls erhalten sie den Wert "FALSE".

Funktion	logischer Wert	Bemerkungen
ODD(X)	TRUE	Der Wert des Ganzzahlausdrucks X ist ungerade.
	FALSE	Der Wert des Ganzzahlausdrucks X ist gerade.

Beispiel:

Die Funktion soll innerhalb einer Abfrage dazu eingesetzt werden, um eingelesene Informationen bestimmten Kategorien zuzuweisen.

```
.
.
VAR FAHRZEUG:(KRAD,FAHRRAD,PKW,ANHÄNGER,LKW,
              PFERDEFUHRWERK);
BEGIN
(* BEI EINER VERKEHRSZÄHLUNG WERDEN FAHRZEUG-
 ARTEN EINGEGEBEN. MIT EINER IF-ABFRAGE WIRD
 NACH MOTORFAHRZEUGEN UND NICHTMOTORFAHRZEUGEN
 UNTERSCHIEDEN *)

    IF ODD (ORD (FAHRZEUG)) THEN

(* "ORD" WEIST GANZZAHLEN ZU. DIE DEKLARATION
 WAR SO ANGELEGT, DASS GERADE ZAHLEN
 MOTORISIERTE FAHRZEUGE KENNZEICHNEN UND
 UNGERADE NICHTMOTORISIERTE. WENN ODD
 (FAHRZEUG)=TRUE, DANN ERFÜLLT SICH DER
 PROGRAMMTEIL NACH "THEN", ANSONSTEN DERJENIGE
 NACH "ELSE". *)

    BEGIN
    .
    .
    .
    END
ELSE
    BEGIN
    .
    .
    .
    END
END.
```

Funktion	logischer Wert	Bemerkungen
EOF (Dateiname)	TRUE	Die Puffervariable ist am Dateiende positioniert.
	FALSE	Die Puffervariable ist nicht am Dateiende positioniert. Wenn die Parameterliste fehlt, wird die Funktion auf die Standarddatei "INPUT" bezogen und entspricht EOF (INPUT). Hinsichtlich der Anwendungsmöglichkeiten ist auf die Prozeduren zur Dateiverarbeitung (vgl. 3.5.4.1) zu verweisen.

Beispiel:

Ebenso läßt sich "EOF" als Abfrage sinnvoll im Programm einsetzen.

```
.
.
VAR A : FILE OF INTEGER;
BEGIN
   IF EOF (A) THEN
   (* PROGRAMMTEIL FUER EOF (A) = TRUE*)
      BEGIN
      .
      . (*Z.B. SCHREIBPROZEDUR MIT
      .  "REWRITE" UND "PUT"*)
      END
ELSE
   (*PROGRAMMTEIL FUER EOF (A)=FALSE*)
      BEGIN
      .
      . (*Z.B. LESEPROZEDUR MIT "RESET"
      .  UND "GET"*)
      END
END.
```

Funktion	logischer Wert	Bemerkungen
EOLN (Datei-name)		Diese Funktion ist ausschließlich auf Textdateien anzuwenden (FILE OF CHAR).
	TRUE	Die Puffervariable hat das Ende einer Zeile erreicht.
	FALSE	Die Puffervariable hat das Ende einer Zeile nicht erreicht. Wenn die Parameterliste fehlt, wird die Funktion auf die Standard-daten "INPUT" bezogen und entspricht "EOLN (INPUT)".

Beispiel:

Die Abfrage "EOLN (DATEINAME)" ist u.a. dann sinnvoll, wenn beim zeichenweisen Lesen oder Schreiben das Zeilenende erreicht wird und danach eine spezifische Verarbeitung stattfinden soll. So kann z.B. nach jeder geschriebenen Zeile eine Leerzeile vorgesehen werden:

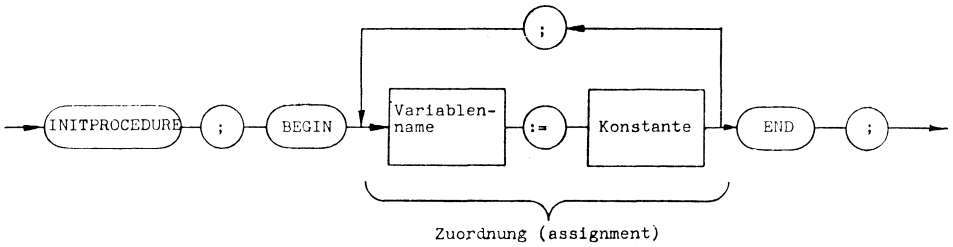
```
      .
      .
VAR DATEI : FILE OF CHAR;
      .
      .
BEGIN
    REWRITE (DATEI);
      .
      .
    IF EOLN (DATEI) THEN WRITELN (DATEI)
    (* MIT "WRITELN (DATEI)" OHNE WEITERE
     ERGÄNZUNG WIRD EINE LEERZEILE
     EINGEFÜGT. *)
ELSE (* SCHREIBEN IN DIE DATEI *)
      .
      .
END.
```

Die beiden folgenden Funktionen IOERROR (Dateiname) und SOR (Dateiname) betreffen ebenfalls die Dateiverarbeitung. Sie gehören nicht zum Sprachumfang von Standard-PASCAL.

Funktion	logischer Wert	Bemerkungen
IOERROR (Dateiname)	TRUE	Es wird ein Ein-/Ausgabefehler angezeigt.
	FALSE	Es liegt kein Eingabe- oder Ausgabefehler vor. Wird auf "IOERROR" abgefragt, lassen sich Eingabe-, Ausgabefehler im Dialog erfassen (vgl. 3.6.4.1).
SOR (Dateiname)		Diese Standardfunktion ist lediglich im Zusammenhang mit dem Lesen von Textdateien zu verwenden (vgl. 3.6.4.1).
	TRUE	Die Puffervariable hat keinen definierten Wert.
	FALSE	Die Puffervariable hat einen definierten Wert. "SOR" ohne Dateinamen bezieht sich auf die Standarddatei "INPUT" und entspricht somit "SOR (INPUT)".

3.6.4.7. Initialisierungsprozeduren

Diese Prozeduren sind nicht in Standard-PASCAL enthalten. Es geht darum, Daten des Hauptprogrammes bzw. eines Moduls (selbständige Programmteile, die noch nicht zu einem ablauffähigen Programm gebunden sind) zur Übersetzungszeit vorzubesetzen (Initialisierung), um Speicherplatz zu sparen.



Beispiel:

```
      .
      .
      VAR FELD1, FELD2 : INTEGER;
      INITPROCEDURE;
      BEGIN
            FELD1 := 0;   (* BESETZUNG MIT 0 *)
            FELD2 := 1;   (* BESETZUNG MIT 1 *)
      END;
      .
      .
```

Die Zuweisung auf die Variable darf nur konstante Selektoren enthalten, d.h. variable Selektoren sind nicht zugelassen.

Beispiel: Bereichsselektion (Indizierung)

```
          .
          .
TYPE TEXT       = ARRAY [0..8]  OF CHAR;
VAR JAHRESZEIT : ARRAY [1..4]  OF TEXT;
INITPROCEDURE;
   BEGIN
       JAHRESZEIT [1] := 'FRUEHLING';
       JAHRESZEIT [2] := 'SOMMER';
       JAHRESZEIT [3] := 'HERBST';
       JAHRESZEIT [4] := 'WINTER';
   END;

(* Bei buchstabenweiser Übertragung wird
   folgendermaßen selektiert:

   JAHRESZEIT [1,0] := 'F'; *)
```

Für Satzselektion (Namensverknüpfung) und Verbindungen beider Selektionsarten gilt entsprechendes.

Auch Variablen vom Mengentyp (set type) kann man mit Konstanten vorbesetzen.

Beispiel:

```
          .
          .
TYPE STATUS = (BESTAND, NEUANLAGE, AENDERUNG,
               SPERRUNG, LOESCHUNG);
VAR SATZZUSTAND : SET OF STATUS;
    INITPROCEDURE
    BEGIN
      SATZZUSTAND :=  [BESTAND .. LOESCHUNG];
    END;
       .
       .
```

4. Ausführungsteil (statement part)

Unter 3. wurde der Vereinbarungsteil beschrieben. Ein Programm oder Unterprogramm (Prozedur, Funktion) besteht meistens aus zwei Teilen, nämlich dem Vereinbarungsteil und dem Ausführungsteil (Anweisungsteil/statement part), wobei der Vereinbarungsteil fehlen darf.

Der Ausführungsteil bestimmt mit seinen Anweisungen algorithmische Aktionen, mit denen die im Vereinbarungsteil bezeichneten Daten verarbeitet werden. Man unterscheidet Operanden und Ausdrücke (vgl. 4.1), die zusammen mit Befehlen die eigentlichen Anweisungen bilden.
Die Abgrenzung der Begriffe ist fließend; daher werden für diese Beschreibung folgende Klärungen zugrundegelegt:

Deutsch/Englisch	Bedeutung
Operand (operand)	Aktuelle Inhalte im Vereinbarungsteil bestimmter Daten (Konstanten, Variable und Funktionen)
Operator (operator)	Zeichen zur Darstellung und Auslösung von Operationen nach mathematischen Regeln
Ausdrücke (expression)	Mathematische Aufgabenstellungen, die mit Operanden und Operatoren Werte eines bestimmten Typs erzeugen
Befehl,Befehlswort (statement, statement key word)	Befehle (i.e.S. Befehlsworte), die Ausdrücke in den Rahmen bestimmter Operationen (Zuweisungen, Aktivieren von Prozeduren, Verzweigungen, Abfrage- und Bedingungsfolgen, Schleifenbildungen) stellen
Anweisung (statement)	Zusammensetzung von Befehlen und Ausdrücken zu Befehlsfolgen. Da eine aus einem Befehl bestehende Anweisung schon als "einfache Befehlsfolge" aufgefaßt wird, ist in vielen anglo-amerikanischen Texten der Begriff "statement" sowohl für Befehle wie für Anweisungen gebräuchlich.

Anmerkung Beispiele:

Während im Vereinbarungsteil die Beispiele inhaltlich so eingeschränkt wurden, daß nur ein ganz konkreter Sachverhalt vermittelt wurde, werden wir im Rahmen des Ausführungsteils möglichst vollständige Beispiele angeführt.

Das geschieht zum Teil mit Sprachkonstruktionen, die bis dahin nicht gebraucht wurden. Diese Methode wird deshalb gewählt, um eine praxisnahe Vermittlung zu ermöglichen und um den Anwender zur aktiveren Arbeit (z.B. mit gezielten Vorgriffen) mit der Benutzeranleitung anzuregen.

4.1. Ausdrücke (expression)

Ausdrücke stellen mathematische Aufgaben dar, die mit Operanden und Operatoren gebildet werden und neue Werte eines bestimmten Typs erzeugen.

4.1.1. Operanden

Operanden sind Variable, Konstanten und Funktionsbezeichner, deren Inhalte mit den in Ausdrücken enthaltenen Algorithmen (Operatoren, Befehle) verarbeitet werden. Der "Operanden"-Begriff wird durch den Faktorenbegriff erweitert (vgl. Multiplikationsoperatoren 4.1.2.2 und Faktoren 4.1.3.4).

a) Konstanten (vgl. 3.3)

An dieser Stelle ist darauf hinzuweisen, daß Konstante den Wert "NIL" (Standardkonstante) annehmen können (vgl. 3.6.4.2). Ergänzend zu den Standardkonstanten MAXINT (2.147.483.647) und MININT (-2.147.483.647) gilt folgendes:

Alle ganzzahligen Werte im Intervall zwischen MININT..MAXINT werden als Integer-Typen dargestellt.

Wenn das Ergebnis einer binären Ganzzahloperation mit zwei Operanden, die im darstellbaren Bereich (MININT..MAXINT) liegen, außerhalb dieses Bereichs liegt, so wird ein Fehler gemeldet.

b) Variable (vgl. 3.5)

Es geht im wesentlichen um die aktuellen Inhalte von Variablen (vgl. auch Parameter Prozedur- und Funktionsdeklarationen - 3.6), die diese im Verarbeitungszusammenhang annehmen können. Dabei ist darauf zu achten, daß Variable vor ihrer ersten Verwendung mit einem Wert besetzt sind (Zuweisung, Initialisierungsprozeduren). Es wird sonst mit zufällig gespeicherten Inhalten gearbeitet, was zu Programmfehlern führen kann, die weder vom Übersetzer erkannt werden noch als Laufzeitfehler sicher festzustellen sind.

Hinsichtlich der Zuordnungsverträglichkeit (assignment compability - vgl. 3.4.5) ist folgendes zu berücksichtigen:

- Ein Operand, dessen Typ den Teilbereich eines anderen ausmacht, wird behandelt wie ein Operand des übergeordneten Typs.

- Das gilt gleichermaßen für Mengentypen.

- Die Operanden von Mengen-Operatoren müssen verträgliche Basistypen haben; sie liefern ein Ergebnis vom Typ des Operanden.

Beispiele

```
TYPE UEBER = (WINZIG,KLEIN,MITTEL,GROSS,GEWALTIG);
VAR UNTER  : KLEIN..GROSS; (*FUER DIE VARIABLEN
                             "UNTERMENGE" BZW.
                             "UNTER" GELTEN MIT
                             AUSNAHME DER TEIL-
                             BEREICHSEINSCHRAENKUNG
                             DIE GLEICHEN ANFOR-
                             DERUNGEN WIE BEI DEN
                             TYPEN "UEBER" BZW.
                             MENGE *)
TYPE MENGE = SET OF 0..100;
VAR UNTERMENGE: SET OF 26..75;
```

c) Funktionsbezeichner (vgl. 3.6.2 und 3.6.3)

Ein Funktionsbezeichner (function-designator) muß neben dem Funktionsnamen die aktuellen Parameter enthalten, damit die Funktion den für den formellen Parameter vorgesehenen Algorithmus ausführen kann (Aktivierung der Funktion). Die Liste der aktuellen Parameter muß nach Anzahl und Reihenfolge der Deklaration entsprechen; die Auswertungsreihenfolge ist damit nicht vorgegeben.

Beispiele

Funktionsdeklaration	Aktuelle Parameter	Funktionsbezeichner (-aufruf)
FUNCTION X (Y:REAL):REAL;	VAR A,B:INTEGER; . . A:=Y; B:=A+2;	X(A/B);
FUNCTION Y (VAR C:INTEGER; (VAR D:REAL): REAL;	VAR X1:INTEGER; X2:REAL . . X1:=3; X2:=X1+0.5;	Y(X1,X2);
FUNCTION Z (FUNCTION FX (INTEGER, INTEGER):REAL);	FUNCTION F1 (X,Y: INTEGER):REAL; . . VAR V10,V11:INTEGER; . . V10 := 5; V11 := V10 * 2;	F2(F1);

4.1.2. Operatoren

Operatoren und Zeichen zur Darstellung und Auslösung mathematischer Regeln und Operationen; sie wirken auf Operanden (vgl. 4.1.1).

Mengen- und Boolesche Operatoren werden als Multiplikations-, Additions- und relationale Operatoren dargestellt.

4.1.2.1. Logische Umkehrung (NOT)

Der Operator "NOT" kann nur auf Operanden vom Typ "BOOLEAN" angewendet werden. Er drückt eine Negation aus, die logisch die Bedeutung des Operanden umkehrt.

Beispiel

```
.
.
VAR BOOL, UMKEHR:BOOLEAN;
.
.
BOOL:=TRUE;
.
.
UMKEHR := NOT BOOL;

(* IN UMKEHR STEHT NACH DER
 NOT-OPERATION "FALSE". *)
.
.
```

4.1.2.2. Multiplikationsoperatoren (multiplying operator)

Unter Multiplikationsoperatoren sind alle Symbole für Rechnerschritte gemeint, die sich auf die Multiplikation zurückführen lassen.

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele
*	generelle Multiplik. oder Schnittmenge	REAL, INTEGER Mengentyp	REAL, INTEGER Mengentyp	6*3|3.5*2.9|5*1.6|A*B Wenn M1 alle geraden Zahlen und M2 alle durch 3 teilbaren Zahlen wären, dann enthielte die Schnittmenge (M1 M2) die Zahlen 6,12,18 usw. Darstellung: M1*M2
/	generelle Division	REAL, INTEGER	REAL	4/2|6.4/0.8|2.5/5|C/D
DIV	ganzzahlige Division	INTEGER	INTEGER	I DIV J, 14 DIV 8 (Ergebnis 1) Bei I>=J > 0 gilt: I DIV J=TRUNC(I/J) -I DIV J=I DIV-J =-(I DIV J) -I DIV-J=I DIV J Für J=0 ist DIV undef.
MOD	Rest bei ganzzahliger Division (abgeleitet von Modulus oder modulo)	INTEGER	INTEGER	I MOD J, 14 MOD 8 (Rest 6) Bei I>=J > 0 gilt: I MOD J=I-(I DIV J) Wenn I DIV J nicht definiert ist, erhält auch I MOD J keinen Wert.
AND	logisches "Und" (auch Konjunktion o. logisches Produkt genannt)	BOOLEAN	BOOLEAN	hintereinander geschaltete Schalter

Fall	X1	X2	X1*X2
I	0	0	0
II	0	1	0
III	1	0	0
IV	1	1	1

Ausgehend davon, daß TRUE=1 und FALSE=0 sind, ergibt die Schaltungslogik Ergebnisse, die der Multiplikation entsprechen. Darstellung X1 AND X2

4.1.2.3. Additionsoperatoren (adding-operator)

Unter Additionsoperatoren sind alle Symbole für Rechnerschritte gemeint, die sich auf die Addition zurückführen lassen.

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele
+	Addition (Identität bzw. Vorzeichenidentität)	REAL, INTEGER	REAL, INTEGER	4+5 | 2.6+3.1 | 2+1.4 | E+F Man spricht von einem Identitätsoperator, weil das Vorzeichen eines Operanden nicht geändert wird und insofern seine Identität erhalten bleibt: +(-4) = -4
	oder Vereinigungsmenge	Mengentyp	Mengentyp	Wenn M1 alle geraden Zahlen, M2 alle durch 4 teilbaren Zahlen wären und M3 der Menge M1 ohne M2 entspricht, dann gilt: M1 = M2 U M3. Darstellung: M1 := M2 + M3

-	Subtrak-tion (Um-kehrung Inversion bzw. Vor-zeichen-umkehrung)	INTEGER, REAL	INTEGER, REAL	5-4 | 6.9-3.5 | 4-1.5 | 6-H Man spricht von einem Umkehrungsoperator, weil das Vorzeichen eines Operanden um-gestellt wird: - (-4) = + 4 - (+4) = - 4
	Unitäres Minus ("vereinigtes Minus")			Das im obigen Beispiel in Klammern gesetzte Minus (-4) bildet eine Einheit mit der dazu-gehörigen Zahl und ist vorrangig vor jedem anderen Operator zu berücksichtigen.
	oder Differenz-menge			Es gilt z.B.: M3 = M1 M2 Darstellung: M3 := M1-M2

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele
OR	logisches "Oder" (auch Disjunktion, Alternative oder logische Addition genannt)			parallele Schalter

Fall	X1	X2	X1+X2
I	0	0	0
II	0	1	1
III	1	0	1
IV	1	1	1

Ausgehend davon, daß TRUE=1 und FALSE=0, ergibt die Schaltungslogik Ergebnisse, die der Addition entsprechen: (X1 OR X2).
Im Fall IV wird tatsächlich nur der Überlauf 1 von 10 registriert.

Die vordefinierten Konstanten MAXINT und MININT liefern bei linearen Operatoren (+ und -) Ergebnisse nach der Ganzzahlarithmetik.

```
MININT = - (- 2.147.483.647) = + 2.147.483.647
MAXINT =                     + 2.147.483.647
```

4.1.2.4. Vergleichsoperatoren (Relationale Operatoren)

Unter Vergleichsoperatoren (relationale operator) sind alle Symbole zu verstehen, die zwei Werte miteinander vergleichen und die Vergleichsbeurteilung als boolesches Ergebnis (TRUE,FALSE) ausweisen.

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele	Ergebnis
=	Klärung, ob ein Operand mit einem anderen gleich ist (Identität)	außer FILE alle Typen	BOOLEAN	4 = 4 TRUE = FALSE 6.5 = 2*3+0.5 usw.	TRUE FALSE TRUE
<>	Klärung, ob ein Operand von einem anderen ungleich ist	außer FILE alle Typen	BOOLEAN	Umkehrung der Beispiele unter "=". 4 <> 4 6.5 <> 5 usw.	FALSE TRUE
<	Klärung, ob ein Operand kleiner als der andere ist	einfache Typen		4 < 4 FALSE< TRUE J < I 4.2 < 5.1 usw.	FALSE TRUE FALSE TRUE
		Zeichentyp (Zeichenkette)		'ABC'< 'AB'	FALSE
>	Klärung, ob ein Operand größer als ein anderer ist	Einfache Typen, Zeichentypen (Zeichenketten)		Umkehrung der Beispiele unter "<". 4 > 4 4.2> 5.1 usw.	FALSE FALSE

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele	Ergebnis
< =	Klärung, ob ein Operand gegenüber einem anderen kleiner oder gleich ist	Einfache Typen	BOOLEAN	4 < = 4 TRUE< = FALSE usw.	TRUE FALSE
		Zeichenketten		'ABC'<='ABCD' usw. Zeichenketten werden beispielsweise nach ihrer lexographischen Ordnung (Sprachbedeutungsordnung) verglichen, um die "lexikalischen Symbole" (vgl. 2.0) im Hinblick auf unterschiedliche Aufgaben im Programm unterscheiden zu können	TRUE
		Mengentypen		Wenn M1 alle geraden und M2 alle durch 4 teilbaren Zahlen sind, dann gilt: M2 < = M1 usw.	TRUE
> =	Klärung, ob ein Operand gegenüber einem anderen gleich oder größer ist	Einfache Typen, Zeichenketten Mengentypen	BOOLEAN	Beispiele analog zu " < = "	

Operator	Operation	Typ des Operanden	Typ des Ergebnisses	Beispiele Ergebnis
IN	Klärung, ob der linke Operand in dem rechten enthalten ist	linker Operand: Ordn.-Typ rechter Operand: Mengentyp mit typenverträglichem Basistyp	BOOLEAN	. . VAR LINKOPD:CHAR; RECHTOPD:SET OF CHAR; . . (*IM KONKRETEN FALL GILT NACH FUELLUNG DER OPERANDEN "LINKOPD" UND "RECHTOPD": LINKOPD IN RECHTOPD = TRUE *)

Die Operanden von Vergleichsoperatoren müssen zuordnungsverträgliche Typen sein (vgl. 3.4.5).

Da die Vergleichsoperatoren auf die Subtraktion zurückgeführt werden, können sich bei Überläufen falsche Aussagen ergeben. Bei eingeschalteter Kontrollfunktion (check option) wird in einem solchen Fall ein Fehler gemeldet.

4.1.2.5. Präzedenzklassen der Operatoren

Mit den vier Präzedenzklassen wird der Vorrang (precedence) der Operatoren für das Zusammenfügen festgelegt:

Präzedenzklasse	Operatoren
1.	NOT | Unitäres Minus
2.	* | / | DIV | MOD | AND
3.	+ | - | OR
4.	= | <> | < | > | <= | >= | IN

"NOT" hat die höchste Priorität, die Vergleichsoperatoren die niedrigste; Folgen von gleichrangigen Operatorn werden von links nach rechts abgearbeitet.

Bei Klammerausdrücken werden die Operatoren innerhalb der Klammern vor denen außerhalb der Klammer aufgelöst.

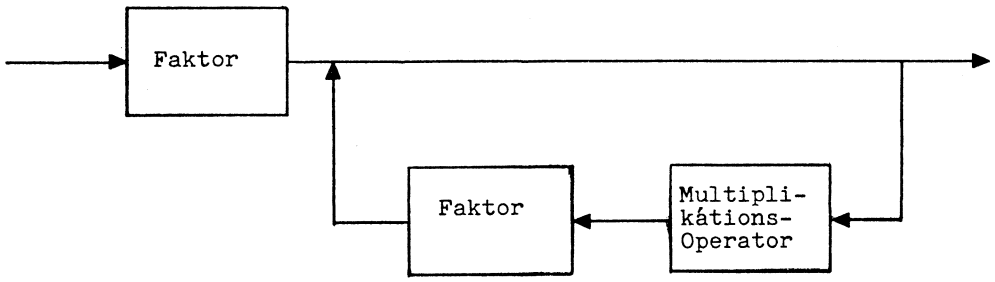
Beispiel:

```
            4 + 2 * 8     =  20
            * vor +

ABER:       (4 + 2) * 8   =  48
            ( + )  vor *
```

4.1.3. Bildung von Ausdrücken

4.1.3.1. Multiplikationsausdruck (Term)

Ein Term ist eine im mathematischen Sinne begrenzte (terminierte) Einheit, die formale Verbindungen von Faktoren (vgl. 4.1.3.4) durch Multiplikationsoperatoren (vgl. 4.1.2.2) beschreibt.

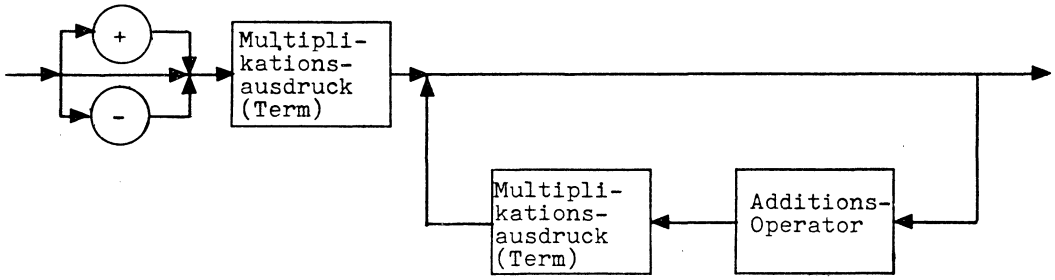


Beispiele:

```
A * B
C / (D-E)
usw.
```

4.1.3.2. Additionsausdruck (einfacher Ausdruck)

Ein "einfacher Ausdruck" (simple expression) ist eine mathematische Verbindung von Multiplikationsausdrücken (vgl. 4.1.3.1) durch Additionsoperatoren (vgl. 4.1.2.3).



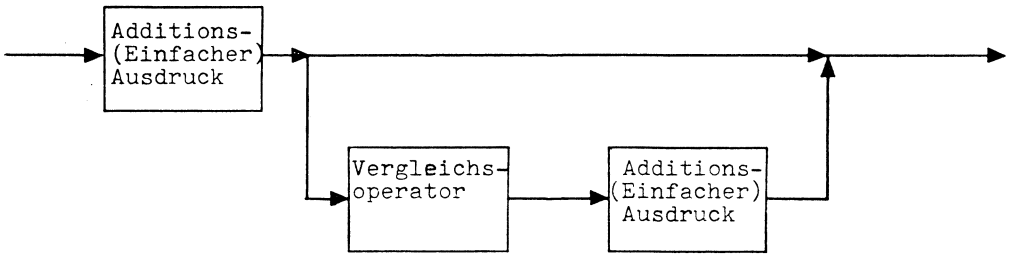
Beispiele:

```
F + G
- H
(I * J) - (K/L)
usw.
```

4.1.3.3. Ausdruck und Ausdrucksverbindungen (Element, Menge)

a) Ausdruck (= Einheitsausdruck)

Ein "Ausdruck" (expression) ist eine mathematische Verbindung von "einfachen Ausdrücken" (vgl. 4.1.3.2) durch Vergleichsoperatoren (vgl. 4.1.2.4).



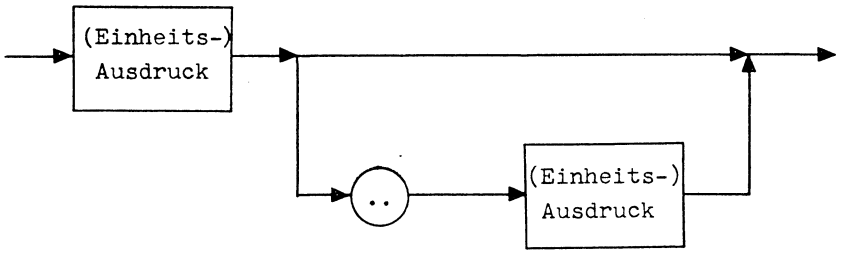
Beispiele:

```
- A + B                    ("einfacher Ausdruck"
                            = Additionsausdruck)

H = 6.4
* (M-N) = P/(Q-R+S)        (Vergleichsausdruck)
T + U > V - W

usw.
```

Für einen Ausdruck, der aus einem einzigen Operanden von einem Teilbereichstyp besteht, gelten abgesehen von der Teilbereichseinschränkung die Bedingungen des übergeordneten Typs. Das gilt gleichermaßen für Mengentypen (vgl. 4.1.1 / Operanden).

b) Element (Ausdrucksverbindungen)

Ein "Element" (element) ist ein Ausdruck oder eine Verbindung von zwei "Ausdrücken" (vgl. a) Einheitsausdruck), um eine Mengenkonstellation darzustellen.

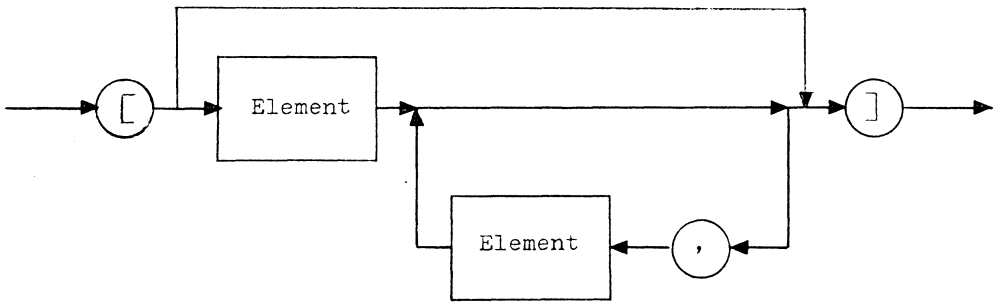


Ein Ausdruck ohne die Verbindung mit einem weiteren entspricht inhaltlich dem (Einheits-) Ausdruck (vgl. 4.1.3.3), während die andere Konstruktion einen Mengenausdruck (set expression) darstellt, wie er u.a. bei der Bildung von Mengenvariablen auftaucht (vgl. 3.4.2.3 und 3.5).

Ausdrücke, die Elemente einer Menge sind (Mengenausdruck) müssen identische Typen haben (Basistyp der Menge).

c) Menge (Elementverbindungen)

Eine "Menge" (set) ist eine Verknüpfung mehrerer Elemente, um Mengen detailliert beschreiben und Mengenselektionen vornehmen zu können.



Ausdrücke, die Elemente einer Menge sind, müssen identische Typen haben (Basistyp der Menge).

Eckige Klammern ohne Inhalt "( )" kennzeichnen eine leere Menge; sie ist Bestandteil jeder Menge bzw. jeden Mengentyps. Eine leere Menge kann auch dadurch gekennzeichnet sein, daß die erste Grenzmenge größer als die zweite ist (z.B. 6..5).

Beispiele:

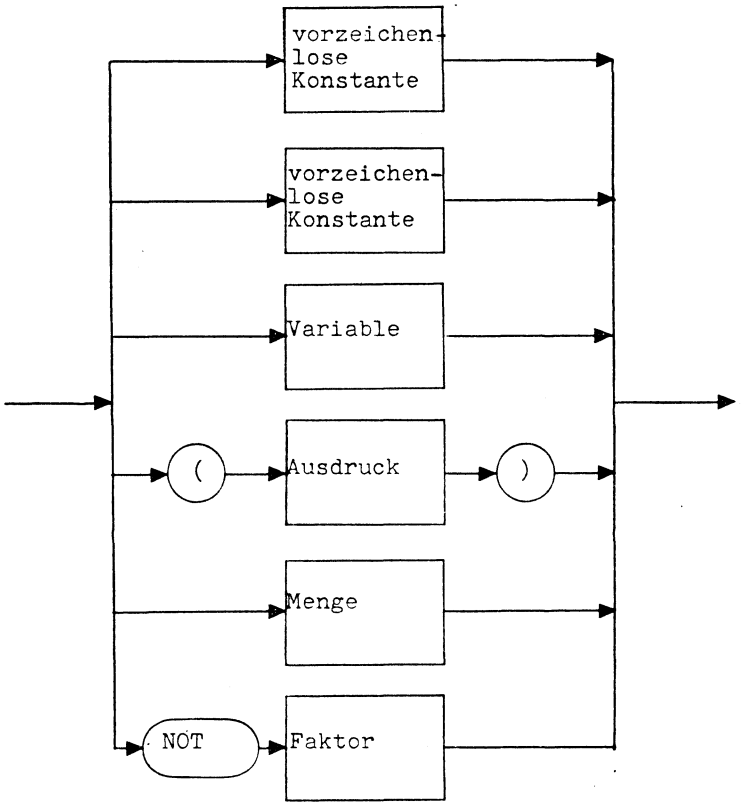
```
[1..3, 5, 7..10]

[0,18,20,22..24,27,30,33,35,36,40,44,45,46,48,
 50,55,60,69,92 .. 200]
 (* REIZZAHLEN BEIM SKAT, OHNE DIE MOEGLICH-
 KEITEN UEBER 92 ZU SPEZIFIZIEREN *)

['A' .. 'D', 'B', 'L .. T', Z]
```

usw.

Bei Ganzzahlmenge (SET OF INTEGER) wird bei eingeschalteter Kontrollfunktion (check option) ein Laufzeitfehler gemeldet, wenn ein Mengenelement außerhalb der Begrenzungen (0..255) liegt.

4.1.3.4. Faktor

Faktoren (factor) sind die Operanden eines Multiplikationsausdrucks (vgl. 4.1.3.1). Während die Operanden (vgl.4.1.1) sich üblicherweise auf Konstanten, Variable und Funktionsbezeichner beschränken, beinhalten Faktoren darüberhinaus Ausdrücke und Mengen (vgl. 4.1.3.3).



Die logische Umkehrung "NOT FAKTOR" (vgl. 4.1.2.1) gilt im Hinblick auf eine vorgegebene Definition, die u.a. bei Abfragen (IF NOT FAKTOR THEN ...) eingesetzt wird. Unter Faktor ist der Zusammenhang des aktuellen Inhalts einer der alternativen Möglichkeiten zu verstehen.

Beispiel für "Menge":

```
     .
     .
VAR A,B,C      : INTEGER;
     .
     .
BEGIN
     .
     .
C : = A * B
IF NOT (C IN  [0 .. 100]  ) THEN
     .
     .
(* DER IN RUNDEN KLAMMER GESETZTE BOOLESCHE
AUSDRUCK UEBERPRUEFT, OB C EINEN AKTUELLEN WERT
ANGENOMMEN HAT, DER ZWISCHEN 0 UND 100
LIEGT. MIT "NOT" WIRD BEWIRKT, DASS DER
PROGRAMMTEIL NACH "THEN" FOLGT, WENN DIESE
BEDINGUNG NICHT ERFUELLT IST. *)
```

Beispiele für Faktoren

a) vorzeichenlose Konstante

Typ	
INTEGER	5
REAL	12.631
CHAR	'A'
(string)	'ZEICHENFOLGE'

Name einer anderen Konst.

```
CONST A = 4;
      B = A;
```

Standardkonstanten

```
FALSE TRUE MININT MAXINT NIL
```

b) Variable

Typ	Zuweisung
VAR V1,V2: INTEGER;	V1:= 6;
V3,V4:REAL;	V3:= 7.5;

c) Funktionsbezeichner

Standardfunktion

```
SIN (V1+V2);
```

Deklarierte Funktion:

```
PROGRAMM SATZDESPYTAGORAS
   .
   .
   VAR KATHETEA,KATHETEB,HYPOTENUSE:REAL;
   FUNCTION KATHETENQUADRATE (KA,KB:REAL):REAL;
        BEGIN
        .
        .
              KATHETENQUADRATE:=KA*KA + KB*KB;
        END;
   BEGIN
        .
        .
        IF HYPOTENUSE * HYPOTENUSE
        = KATHETENQUADRATE (KATHETEA,KATHETEB)
          (*FUNKTIONSAUFRUF MIT FUNKTIONS-
          BEZEICHNER*)
        THEN ...
        ELSE ...
   END.
```

d) Ausdruck (Einheitsausdruck)

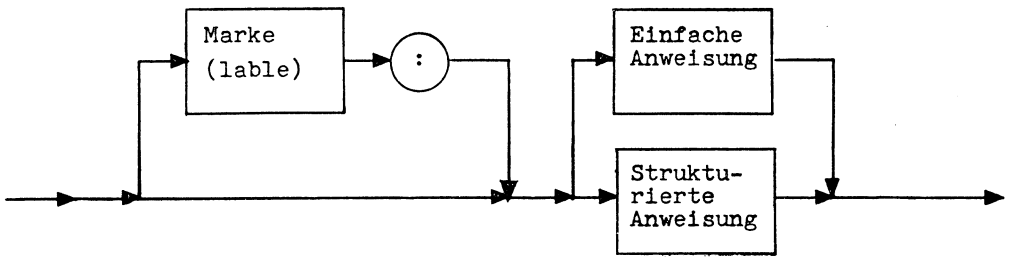
als "Einfacher Ausdruck" (Additionsausdruck)	(- X) (Y + Z) ((A*B) + C - (D/E)) (F DIV J) (K MOD L) (M AND N)
als "Vergleichsausdruck"	(D-P = Q +(R*S)) ((T*U)-V>(W/X) +4) E DIV J = TRUNC (I/J)
(Zeichenketten)	('ABC' < 'ABCD')
(logische Ausdrücke)	((E>F) AND (F>F)) ((G OR Q) AND NOT (R<10))

e) Mengen

als "Einfacher Ausdruck"	[0..9,15,20,51..99]
als "Vergleichsausdruck"	((E AND F) IN ([FALSE.. (G OR H)]))

4.2. Anweisungen (statement)

Anweisungen beschreiben die "algorithmischen Aktionen", mit denen die aktuellen Inhalte der im Vereinbarungsteil bezeichneten Daten verarbeitet werden. Anweisungen bestehen aus Befehlen (statement keyword) und Ausdrücken (vgl. 4.1).



Vor einer Anweisung darf eine Marke stehen (vgl. 3.2), zu der im Programmablauf verzweigt werden kann (Ansprung mit GOTO).

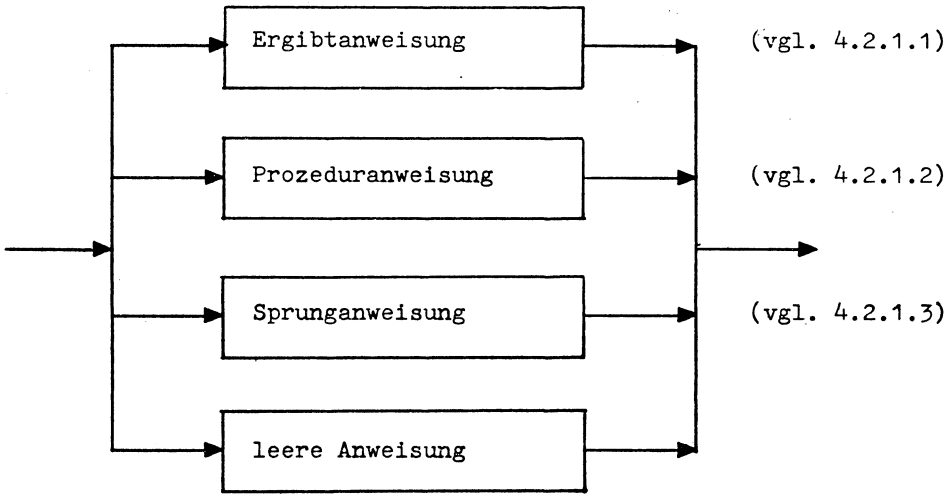
Beispiel:

```
 1  PROGRAM MARKEN (INPUT,OUTPUT);
 2  LABEL     (*DEKLARATION*)
 3  1,2;
 4  VAR
 5    I: INTEGER;
 6   BEGIN
 7    I:=0;
 8   1:       (*EINLEITUNG ANWEISUNGSFOLGE*)
 9    I:= SUCC (I);
10    WRITELN (I);
11    IF I > 5
12    THEN
13    GOTO 2  (*ANSPRUNG*)
14    ELSE
15    GOTO 1; (*ANSPRUNG*)
16   2:       (*EINLEITUNG ANWEISUNGSFOLGE*)
17    WRITELN ('I IST GROESSER 5');
18   END.
```

Der vorgegebene Programmtext bestimmt gleichzeitig die Ausführungsfolge. Sie kann nur durch den Ansprung von Marken unterbrochen werden. Im Regelfall wird man Wiederholungen durch Programmschleifen lösen, wobei aber aus unterschiedlichen Gründen (z.B. nachträgliche Änderungen) ein mit Marken eingeflochtenes Unterprogramm sinnvoll sein kann. Das geht zu Lasten einer "sauberen" Programmstruktur (Unübersichtlichkeit).

4.2.1. Einfache Anweisungen (simple statement)

Eine einfache Anweisung läßt sich nicht in andere Anweisungen unterteilen.

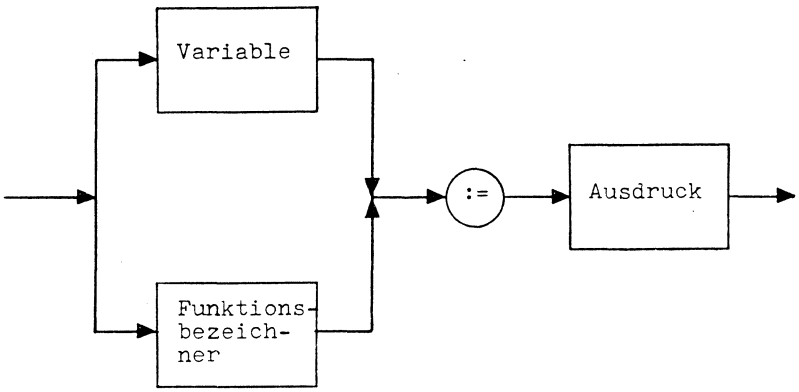


Leere Anweisungen sind durch "nichts" definiert. Aus syntaktischen Gründen werden sie eingeführt (vgl. auch 4.2.2). Ein Anwendungsfall besteht beispielsweise darin, daß man aus einer Anweisung heraus direkt an das Programmende springen will.

```
 1  PROGRAM SPRUNGENDE (INPUT,OUTPUT);
 2  LABEL 1,2;
 3  VAR A: INTEGER;
 4      BEGIN
 5      A:= 0;
 6  1:
 7    A:= SUCC (A);
 8    IF A > 5
 9    THEN GOTO 2
10    ELSE GOTO 1;
16  2:;         (*LEERE ANWEISUNG*)
17   END.
```

4.2.1.1. Wertzuweisung oder Ergibtanweisung (assignment statement)

Eine Zuweisung dient dazu, den aktuellen Wert einer Variablen oder einer Funktion (Funktionsbezeichner) durch den Wert eines Ausdrucks zu ersetzen.



Nach dem Ergibtzeichen ":=" wird die Wertzuweisung auch Ergibtanweisung genannt. Im Unterschied zum Gleichheitszeichen können nach dem aktuellen Status unterschiedliche Inhalte auf beiden Seiten der Zuweisung stehen.

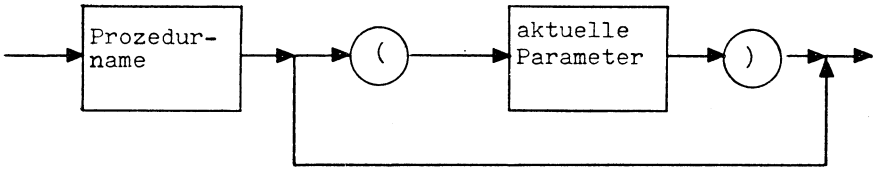
Beispiel: A:= A + B

"A=A+B" ist mathematisch falsch.

Der Ausdruck, der in einer Zuweisung übergeben wird, muß mit der Variablen oder der Funktion zuordnungsverträglich (vgl. 3.4.5) sein.

4.2.1.2. Prozeduranweisung (procedure statement)

Eine Prozeduranweisung beinhaltet den Prozedurnamen und, wenn formale Parameter vorgegeben sind, die entsprechenden aktuellen Parameter (vgl. 3.6.2 und 3.6.3).



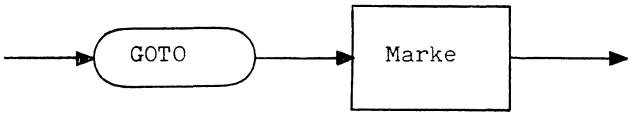
Mit der Prozeduranweisung wird eine Prozedur aufgerufen (Aktivierung); die aktuellen Parameter treten an die Stelle der formalen Parameter. Anzahl und Reihenfolge der aktuellen Parameter richten sich nach den formalen Parametern.

Beispiel:

```
 1 PROGRAM AUSWECHSELN (INPUT,OUTPUT);
 2       VAR A,B:INTEGER;
 3       PROCEDURE TAUSCHEN (VAR P1,P2:INTEGER);
 4                           (*FORMALE PARAMETER*)
 5          VAR P:INTEGER;
 6              BEGIN
 7                P:=P1;
 8                P1:=P2;
 9                P2:=P
10              END;
11        BEGIN
12              READ (A,B);
13              TAUSCHEN (A,B); (*AKTUELLE PARAMETER*)
14              WRITE (A,B)
15        END.
```

4.2.1.3. Sprunganweisung (goto statement)

Eine Sprunganweisung dient dazu, ein Programm zu unterbrechen und an anderer Stelle fortzuführen. Die Bezeichnung des aufgerufenen Programmteils geschieht mit Marken, die im Vereinbarungsteil zu deklarieren und bei den Sprunganweisungen und den auszuführenden Anweisungen anzugeben sind.



"GOTO" ist das Befehlswort, das die Sprunganweisung auslöst. Das Sprungziel (Marke) muß im gleichen oder einem übergeordneten Block stehen. In diesem Gültigkeitsbereich müssen alle Marken verschieden sein.

Ausdrücklich muß auf den Unterschied zwischen "Marken" und "Auswahlmarken" hingewiesen werden. Auswahlmarken bezeichnen die Varianten von Satztypdefinitionen bzw. Satzvariablendeklarationen (vgl. 3.4.2.2 und 3.5.2.2) und von Auswahlanweisungen (vgl. 4.2.2.2). Wenn ganzzahlige Auswahlmarken gewählt werden, ist eine Verwechslung möglich.

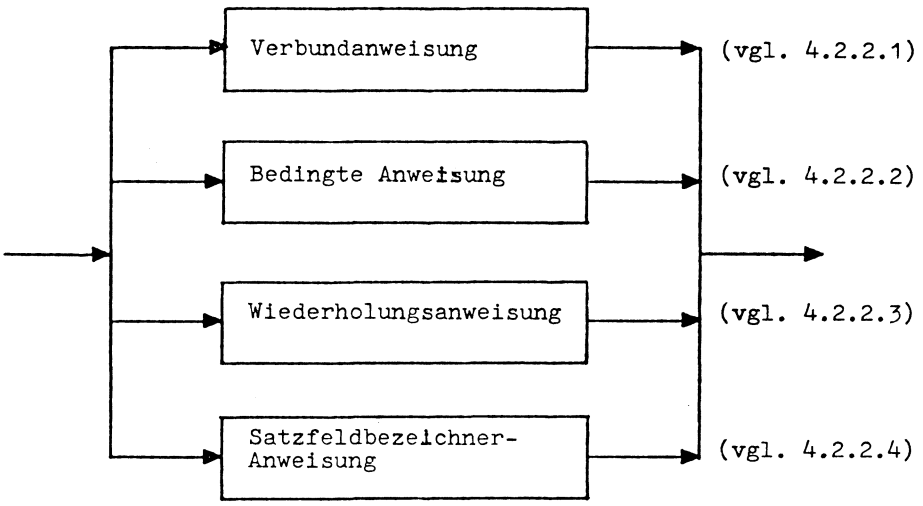
Beispiel:

```
 1  PROGRAM SPRUNG (INPUT,OUTPUT);
 2  LABEL 1,2;                    (*MARKEN*)
 3  VAR A: RECORD CASE B:INTEGER OF
 4                 1:(C:INTEGER);              (*AUSWAHL-
 5                 2:(D:ARRAY(1..4)OF CHAR); MARKEN*)
 6         END;
 7      V:INTEGER;
 8      BEGIN
 9         1: READ (V);
10            IF ODD(V) THEN GOTO 2     (*MARKE 1*)
11                      ELSE GOTO 1     (*MARKE 2*)
12         2: IF V > 1000 THEN
13                  BEGIN
14                      A.B := 1;
15                      A.C := V;
16                      WRITE (A.C)
17                  END
18              ELSE
19                  BEGIN
20                      A.B := 2;
21                      A.D := 'LEER';
22                      WRITE (A.D)
23                  END;
24      END.
```

Sprunganweisungen können im Programm dazu führen, daß der Programmtext nicht mehr mit seinem Ablauf übereinstimmt. Das Programm wird dadurch unübersichtlich und ist schwer zu verstehen. Dadurch ist seine Änderungsfähigkeit eingeschränkt. Neue Fehler können im Korrekturfall mangels Überschaubarkeit hinzukommen. Es gibt meistens Anweisungen (vgl. 3. Strukturierte Anweisungen), die Sprunganweisungen sachgerecht ersetzen. Daher sollte ihr Einsatz auf ungewöhnliche Fälle beschränkt sein.

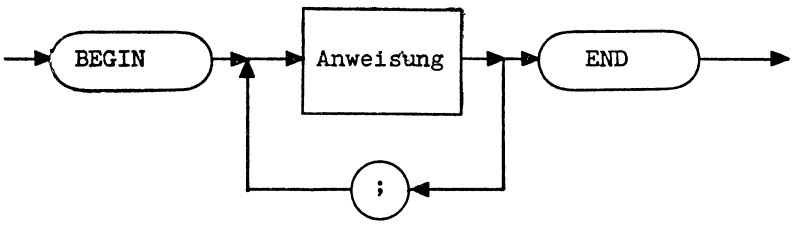
4.2.2. Strukturierte Anweisungen (structured statements)

Strukturierte Anweisungen verbinden mit bestimmten Befehlswörtern Anweisungen verschiedener Art. Entweder sind sie nacheinander auszuführen (Verbundanweisung), treten unter bestimmten Bedingungen ein (Bedingte Anweisung) oder wiederholen sich (Wiederholungs- oder Zyklusanweisung). In diesem Zusammenhang wird auch die Satzfeldbezeichner - Anweisung (With-Anweisung) behandelt.



4.2.2.1. Verbundanweisung (compound statement)

Verbundanweisungen werden in der Reihenfolge, in der sie niedergeschrieben wurden, ausgeführt. Die Befehlsworte "BEGIN" und "END" begrenzen eine Verbundanweisung. Der Ausführungsteil als Bestandteil von Programmen und Unterprogrammen (Prozeduren und Funktionen) entspricht einer Verbundanweisung.



Mit einer Verbundanweisung können darüberhinaus mehrere Anweisungen syntaktisch zu einer einzigen Anweisung zusammengefaßt werden. Überall dort, wo eine Anweisung gefordert ist, darf auch eine Verbundanweisung stehen.

Das Semikolon zwischen den Anweisungen ist ein Folgeoperator, der besagt, daß die nachfolgende Anweisung erst dann ausgeführt werden darf, wenn die vorangehende beendet ist.

4.2.2.2. Bedingte Anweisung (conditional statement)

Eine bedingte Anweisung geht von bestimmten Voraussetzungen aus, unter denen bestimmte Anweisungen ausgeführt werden sollen.

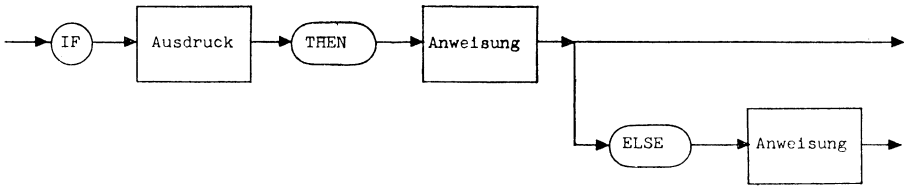
Man unterscheidet dabei

- die Abfrage von Bedingungen (Vergleichsausdrücke), deren boolesche Aussage (Wahrheitswert) über die Wahl einer von zwei Anweisungsalternativen entscheidet (vgl. 4.2.2.2 - a) Wenn-Anweisung oder IF-statement);

- und die Vorgabe von einer und mehrerer Alternativen, die nicht mit einer Abfrage verknüpft sind, sondern von einem Bedingungsschlüssel (Auswahlmarken) ausgehen, der vorher vereinbart wird (vgl. 4.2.2.2 - b) Auswahl-Anweisung oder case-statement).

a) Wenn-Anweisung (IF-statement)

In Abhängigkeit von einer Bedienungsabfrage (IF) wird entsprechend dem booleschen Wahrheitswert (TRUE oder FALSE) eine vorgesehene Anweisung (THEN) ausgeführt und im Abweisungsfall nicht ausgeführt. Das Programm wird mit der nächsten Anweisung fortgesetzt; eine spezifische Folgeanweisung (ELSE) kann eingesetzt werden.



Je nach Abfragestruktur können sich folgende Konstellationen ergeben:

Abfrage-form positiv	Eintritt der vor-gesehenen Anweisg. bei	Eintritt der nächsten Anweisung bzw. der Folgeanweisg./ELSE bei
z.B. IF EOF (Dateiname) THEN	TRUE	FALSE
negativ		
IF NOT EOF (Dateiname) THEN	FALSE	TRUE

Beispiel:

```
1  PROGRAM SCHREIBEN (INPUT,OUTPUT)
2  VAR A : INTEGER
3        BEGIN
4           READ (A);
5           IF ODD(A) THEN WRITE (UNGERADE´) (*TRUE*)
6           ELSE WRITE (´GERADE´);           (*FALSE*)
7        END.
```

Wenn eine Abfrage mehr als 2 Alternativen beinhaltet, kann man Wenn-Anweisungen schachteln. Das führt oft zu einem unübersichtlichen Programmtext.

```
z.B.: IF B1 THEN IF B2 THEN IF B3 THEN A1
      ELSE IF B4 THEN A2 ELSE A3 ELSE A4;
```

Die Lesbarkeit diees Beispiels läßt sich verbessern, wenn man mit Verbundanweisungen (BEGIN ...END) arbeitet:

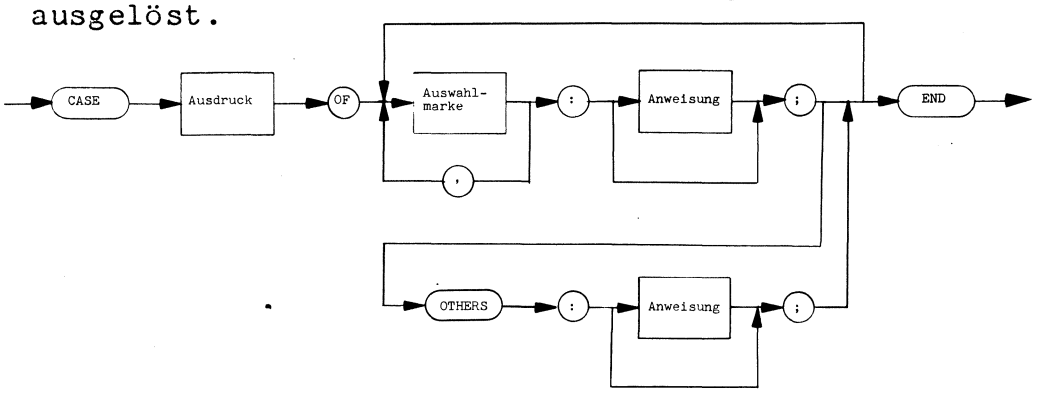
```
.
.
BEGIN
   IF B1 THEN
      BEGIN
       IF B2 THEN
         BEGIN
           IF B3 THEN A1
           ELSE
                BEGIN
                  IF B4 THEN A2
                  ELSE A3
                END
          END
        ELSE A4;
      END
END.
```

Während es bei der ursprünglichen Darstellung noch zu Unklarheiten darüber führen kann, zu welcher Wenn-Bedingung die Folgeanweisungen (ELSE) gehören, gibt es bei der durch Verbundanweisung geklammerten Schreibweise keine diesbezüglichen Schwierigkeiten.

Auch hier gilt, daß man die Verschachtelungsmöglichkeiten sinnvoll beschränken sollte, um die Lesbarkeit des Programms zu steigern und seine Wartungseignung damit zu gewährleisten.

b) Auswahl-Anweisung (CASE-statement)

In Abhängigkeit von einem Bedingungsschlüssel (Auswahlmarken) werden Anweisungsalternativen ausgelöst.



Die Auswahl-Anweisung wird von den Schlüsselwörtern CASE... OF... charakterisiert. Während unmittelbar nach CASE der aktuelle Ausdruck für die Auswahl einer Anweisung geführt wird, stehen nach OF die Anweisungsalternativen.

Anweisungsalternativen:

Eine Anweisungsalternative wird jeweils mit einer Auswahlmarke (case constant list) eingeleitet, an die nach ":" eine Anweisung geknüpft ist. Man kann mehrere unterschiedliche Auswahlmarken mit einer einzigen Anweisung verbinden.

Aktueller Ausdruck:

Der Ausdruck (expression) nach CASE muß vom gleichen Typ wie die Auswahlmarken sein. Er wird im Programmablauf bestimmt und wenn er einer Auswahlmarke entspricht, ist ein vorgesehener Fall (CASE) eingetreten; die dazugehörige Anweisung wird dann veranlaßt.

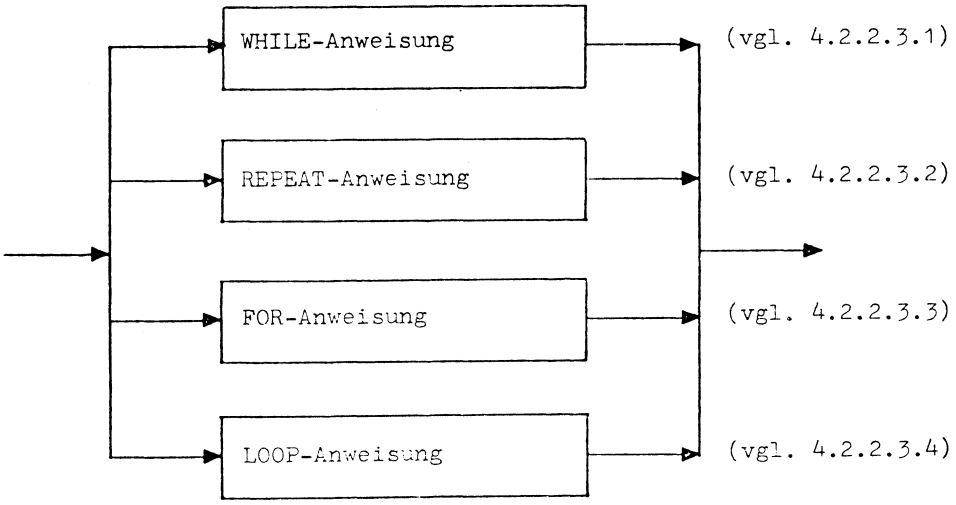
Als Spracherweiterung gegenüber Standard-PASCAL ist wahlweise die OTHERS-Anweisung vorgesehen, die dann ausgeführt wird, wenn der aktuelle Ausdruck einen Wert annimmit, der mit keiner der vorgesehenen Auswahlmarken übereinstimmt.

Beispiel:

```
 1  PROGRAMM WAEHRUNG (INPUT,OUTPUT);
 2  VAR KENNUNG : INTEGER;
 3      DM,VALUTA : REAL;
 4  BEGIN
 5  KENNUNG:=1;
 6  REPEAT
 7     READ (KENNUNG,VALUTA);
 8     CASE KENNUNG OF 0:;
 9                     1:   DM:=VALUTA * 1.90;
10                     2:   DM:=VALUTA * 3.50;
11     OTHERS : WRITELN ('WAEHRUNG WIRD NICHT
                          UMGERECHNET')
12     END;
13     IF (KENNUNG>0) AND (KENNUNG <3) THEN
       WRITELN (DM:9:2);
14     UNTIL (KENNUNG <1) OR (KENNUNG >9);
15     END.
```

4.2.2.3. Wiederholungs- oder Zyklusanweisung (repetitive statement)

Die Sprache PASCAL kennt 3 Wiederholungs- oder Zyklusanweisungen. Der hier zu beschreibende Sprachumfang ist um eine zusätzliche Wiederholungsanweisung erweitert worden.



Bei den Wiederholunsanweisungen lassen sich prinzipiell zwei verschiedene Fälle unterscheiden.

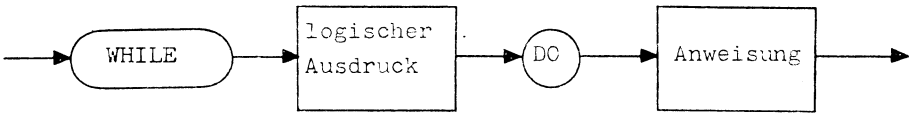
- Entweder kennt man bei der Programmerstellung die Anzahl der Schleifendurchläufe (determinierte Schleife);

- oder die Anzahl der Schleifendurchläufe ergibt sich zur Programmlaufzeit (iterative Schleife).

Für die iterative Schleifenverarbeitung stehen die REPEAT-Schleife, die WHILE-Schleife und die LOOP-Schleife zur Verfügung. Determinierte Schleifenverarbeitung wird durch die FOR-Schleife ermöglicht.

4.2.2.3.1. WHILE-Anweisung

Die WHILE -Anweisung ist eine Wiederholungsanweisung, bei der die Anzahl der Wiederholungen von einer Bedingung abhängig ist, die zur Programmlaufzeit erzeugt wird (iterative Schleife). Die Bedingung wird zu Beginn eines neuen Durchlaufs geprüft.



Die Anweisung hinter DO wird solange ausgeführt, wie der logische Ausdruck den Wert "TRUE" hat. Hinter DO darf nur eine Anweisung stehen. Falls mehrere Anweisungen ausgeführt werden sollen, so ist die Verbundanweisung BEGIN END (vgl. 4.2.2.1) einzusetzen. Die Verbundanweisung selbst gilt als eine Anweisung, kann jedoch mehrere Anweisungen enthalten.

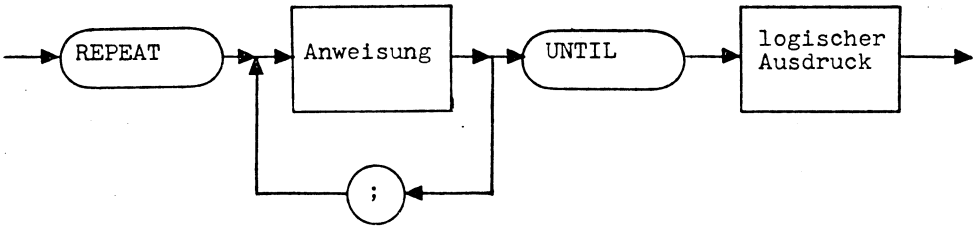
Operanden des logischen Ausdrucks müssen innerhalb der Schleife so verändert werden, daß die Endebedingung für die Programmschleife (logischer Ausdruck = FALSE) mindestens einmal zur Programmlaufzeit erfüllt ist. Hat der logische Ausdruck zu Beginn der Schleife den Wert FALSE, so wird die Programmschleife keinmal durchgeführt.

Beispiel: Es sollen Zahlen eingelesen und summiert werden, bis ein bestimmter Betrag erreicht ist.

```
PROGRAM SUMMIEREN (INPUT,OUTPUT);
CONST GRENZE = 531.75;
VAR ZAHL,SUMME : REAL;
BEGIN
 SUMME:=0;
 WHILE SUMME <= GRENZE DO
 BEGIN
 READ (ZAHL)
 SUMME := SUMME + ZAHL;
 END;
WRITE ('Summe =:' SUMME:8:2)
END.
```

4.2.2.3.2. REPEAT-Anweisung

Bei der REPEAT-Anweisung erfolgt der Test auf Abbruch der Schleife nach jedem Durchlauf.



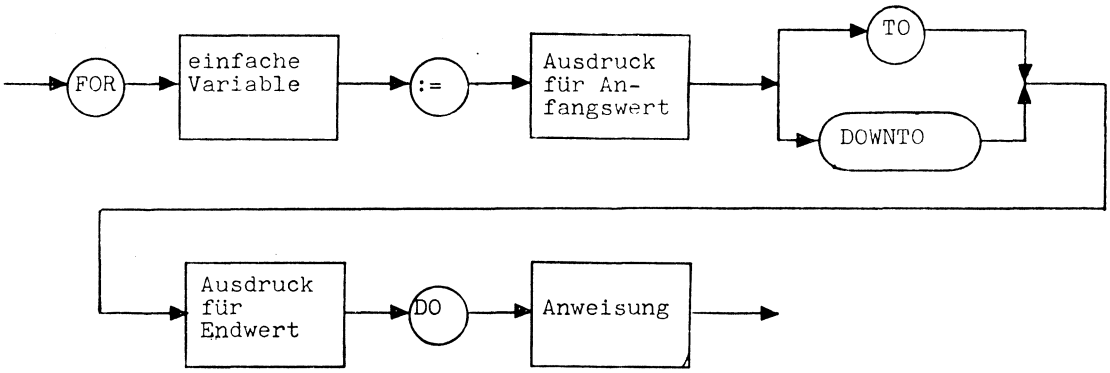
Die Anweisungen zwischen REPEAT und UNTIL werden solange wiederholt, wie der logische Ausdruck den Wert FALSE hat. Die Operanden des logischen Ausdrucks müssen innerhalb der Schleife so verändert werden, daß die Endebedingung (logischer Ausdruck = TRUE) zur Programmlaufzeit erfüllt wird. Hat der logische Ausdruck zu Beginn der Schleife den Wert TRUE, so wird die Programmschleife einmal ausgeführt.

Beispiel: Das Programmbeispiel aus Kap. 4.2.2.3.1 Einlesen und Summieren von Zahlen bis Erreichen eines Grenzwertes, soll mit Hilfe der Repeat-Anweisung formuliert werden.

```
PROGRAM SUMMIEREN (INPUT,OUTPUT);
CONST GRENZE = 531.75;
VAR ZAHL, SUMME:REAL;
BEGIN
SUMME:=0;
REPEAT
 READ (ZAHL)
 SUMME:=SUMME+ZAHL;
UNTIL SUMME > GRENZE;
WRITE ('Summe =:', SUMME:8:2)
END.
```

4.2.2.3.3. FOR-Anweisung

Die FOR-Anweisung dient zur Formulierung von Programmschleifen, bei denen die Anzahl der Durchläufe vor Eintritt in die Schleife festliegt.



Die einfache Variable hinter FOR wird auch Laufvariable oder Schleifenzähler genannt. Die Laufvariable muß vom Aufzählтyp (skalarer Typ) sein. Aufzählbare Typen sind die Standard-Typen INTEGER, CHAR, BOOLEAN sowie vom Benutzer definierte aufzählbare Typen (vgl. 3.4.1.2).

Diese Typen bilden eine geordnete Menge von Werten; zu jedem Wert gibt es genau einen Vorgänger (Predecessor) und einen Nachfolger (successor). Die Laufvariable muß zu demselben Block gehören wie die FOR-Schleife selbst. Außerdem darf die Laufvariable innerhalb der Schleife nicht verändert werden. Die Ausdrücke für Anfangswert und Endwert müssen vom gleichen Typ wie die Laufvariable sein.

Zu Beginn der FOR-Schleife wird der Laufvariablen der Wert des Ausdrucks für den Anfangswert zugewiesen. Die Anweisung hinter DO wird solange wiederholt, bis die Laufvariable den Wert des Ausdrucks für den Endwert überschritten hat.

Die Änderung der Laufvariablen erfolgt automatisch jeweils am Ende eines Schleifendurchlaufs. Bei Verwendung des Schlüsselworts TO wird der Laufvariablen jeweils ihr Nachfolger (SUCC (Laufvariable)) zugewiesen; eine Verwendung des Schlüsselwortes DOWNTO bedingt die Zuweisung des Vorgängers (PRED (Laufvariable)) auf die Laufvariable. Wird die FOR-Schleife regulär, d.h. nicht durch einen GOTO verlassen, so ist der Wert der Laufvariablen undefiniert. Ist bei Verwendung von TO (DOWNTO) der Anfangswert größer (kleiner) als der Endwert, so wird die Schleife keinmal ausgeführt.

Sollen mehrere Anweisungen wiederholt werden, so ist hinter DO die Verbundanweisung (BEGIN ... END) einzusetzen.

Beispiele:
---

a) Eine bestimmte Anzahl von Zahlen soll eingelesen und in einem eindimensionalen Feld gespeichert werden; anschließend soll die größte der eingegebenen Zahlen ermittelt und ausgegeben werden.

```
PROGRAM MAXIMUM  INPUT,OUTPUT ;
CONST MAXANZ=20;
TYPE ZAHLENFELD=ARRAY [1..MAXANZ] OF REAL;
VAR I:1 .. MAXANZ;
    MAXI:REAL;
    FELD:ZAHLENFELD;
BEGIN (*ZAHLEN EINLESEN*)
    FOR I:= 1 TO MAXANZ DO
      READ (FELD [I] );
         (*MAXIMUM SUCHEN*)
    MAXI:=FELD [1] ;
    FOR I:=2 TO MAXANZ DO
       IF FELD  [I] >  MAXI THEN
                           MAXI:=FELD [I] ;
         (*MAXIMUM AUSGEBEN*)
    WRITE ('Maximum = ',MAXI)
   END.
```

b) FOR-Schleife, bei der die Laufvariable einen vom Benutzer definierten zählbaren Typ hat.

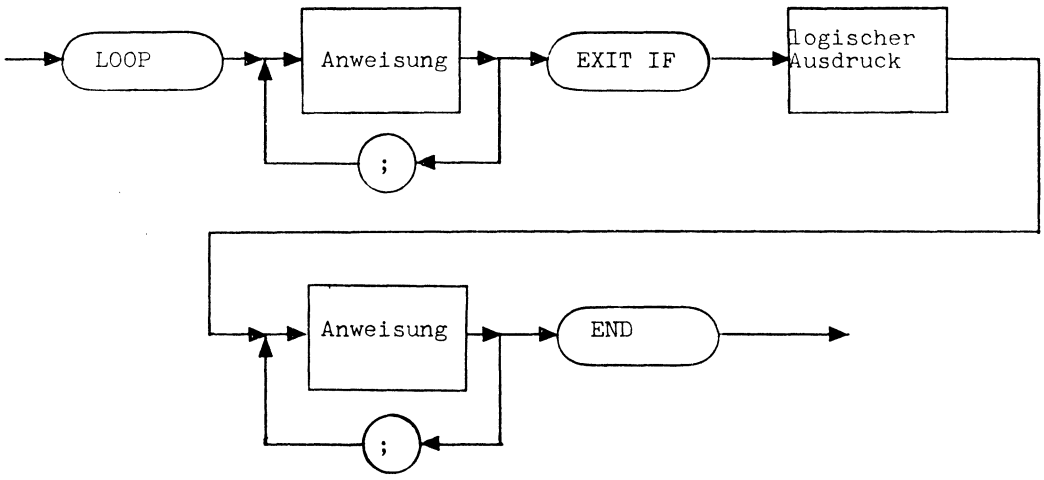
```
PROGRAM FARBEN (INPUT,OUTPUT);
TYPE FARBEN = (ROT,GELB,GRUEN,SCHWARZ);
VAR BUNT : ARRAY   ROT .. SCHWARZ  OF FARBEN;
       I : FARBEN;
BEGIN
   FOR I := ROT TO SCHWARZ DO
               BUNT  I  := I;
    .
    .
    .
    .
  END.
```

c) Ein zweidimensionales Zahlenfeld soll zur Programmlaufzeit auf einen Anfangswert gesetzt werden.

```
PROGRAM FELD (INPUT,OUTPUT);
CONST MAX = 20;
      WERT= 1;
VAR FELD:ARRAY   [1..MAX, 1..MAX]  OF REAL;
    I,J:1 .. MAX;
BEGIN
    FOR I:= 1 TO MAX DO
        FOR J:=1 TO MAX DO
              FELD   I,J    := WERT     .
END.
```

4.2.2.3.4. LOOP-Anweisung

Die LOOP-Anweisung ist in Standard-PASCAL nicht enthalten. Diese Anweisung bietet die Möglichkeit, eine Programmschleife zu formulieren, bei der das Beenden der Schleife durch Prüfen einer Bedingung an einer beliebigen Stelle innerhalb der Schleife möglich ist.



Die Anweisungsfolge hinter LOOP wird ausgeführt; danach wird eine Bedingung geprüft und falls diese wahr ist, die Schleife verlassen. Ist die Bedingung falsch, wird die Anweisungsfolge vor END ausgeführt und danach wieder zum Anfang der LOOP - Anweisung gesprungen.

Die Anwendung dieser Anweisung soll eine Gegenüberstellung einer Programmfolge, die einmal mit Hilfe einer While-Anweisung und zum anderen mit Hilfe der LOOP-Anweisung formuliert ist, zeigen.

a) Programmfolge mit While-Anweisung:

```
PROGRAM TEST (INPUT);
CONST ENDE = '*';
TYPE STRING = ARRAY   1..30   OF CHAR;
VAR AUTOR,TITEL:STRING;
    JAHR:INTEGER;
    X:BOOLEAN;
BEGIN
    X:=TRUE
    WHILE X = TRUE DO
    BEGIN
      READ (TITEL);
      X:= NOT (TITEL  1  = ENDE);
      IF X THEN
          BEGIN
             READ (AUTOR);
             READ (JAHR)
          END;
      END
END.
```

Die Variablen TITEL, AUTOR und JAHR sollen solange eingelesen werden, bis das erste Zeichen der Variablen TITEL das Endekennzeichen (hier '*') enthält. Danach soll die Schleife beendet sein; d.h. das Einlesen der Variablen AUTOR und JAHR soll entfallen. Das oben beschriebene Programm löst dieses Problem. Allerdings mit zusätzlichen Anweisungen.

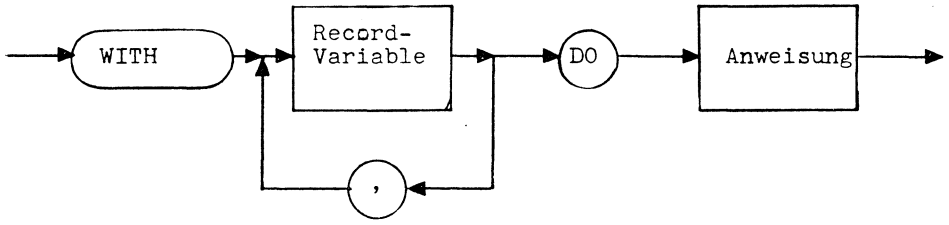
b) Programmfolge mit Hilfe der LOOP-Anweisung:

```
PROGRAM TEST (INPUT);
CONST ENDE = '*';
TYPE STRING=ARRAY [1 .. 30] OF CHAR;
VAR AUTOR,TITEL:STRING;
    JAHR:INTEGER;
BEGIN
    LOOP
       READ (TITEL);
       EXIT IF TITEL [1] = ENDE
       READ (AUTOR);
       READ (JAHR)
    END
END.
```

Dieses Programm entspricht dem unter Punkt a beschriebenen.

4.2.2.4. WITH-Anweisung

Die WITH-Anweisung dient dazu, innerhalb einer Anweisung, die mehrere Recordkomponenten enthält, die symbolischen Namen zu vereinfachen.



Beispiel:

```
PROGRAM ARCHIV (INPUT,OUTPUT);
TYPE STRING = ARRAY   [1 .. 30]  OF CHAR;
     BUCH = RECORD   TITEL:STRING;
                     AUTOR:STRING;
                     JAHR:INTEGER;
            END;
VAR ARCHIV:ARRAY   [1 .. 50]   OF BUCH;
  - I:1 .. 50;
BEGIN
    FOR I:=1 TO 50 DO
    WITH ARCHIV [I] DO
    BEGIN
     LOOP
       READ (TITEL);
       EXIT IF TITEL [1] =*;
       READ (AUTOR);
       READ (JAHR)
     END
    END
END.
```

Die WITH-Anweisung erspart dem Programmierer Schreibarbeit; ohne diese Anweisung müßte ein Einlesebefehl des Beispielprogramms folgendermaßen codiert werden:

```
READ (ARCHIV[I]. TITEL)
```

Außerdem erspart die Verwendung der WITH-Anweisung bei Zugriff auf Elemente eines Feldes (array) vom Typ Record Rechenzeit.

5. Externe Daten und Prozeduren

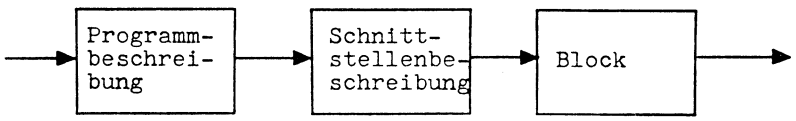
5.1. Allgemeines

Das Kapitel 5 beschreibt eine Erweiterung von Standard-PASCAL um ein Modul-Konzept. Damit wird dem Anwender die Möglichkeit gegeben, von PASCAL-Programmen auf externe Größen zuzugreifen. Das Modul-Konzept bietet somit die Möglichkeit, modular zu programmieren.

Als externe Größen können Daten, Prozeduren und Funktionen vereinbart werden. Die Schnittstelle zwischen Programm und Modul muß in einer Schnittstellenbeschreibung, die alle notwendigen Informationen für eine vollständige Konsistenzprüfung (Zusammenhangsprüfung) enthält, beschrieben werden.

Die Erweiterung der Sprache stellt folgendes Syntaxdiagramm dar.

Programm mit externen Größen:

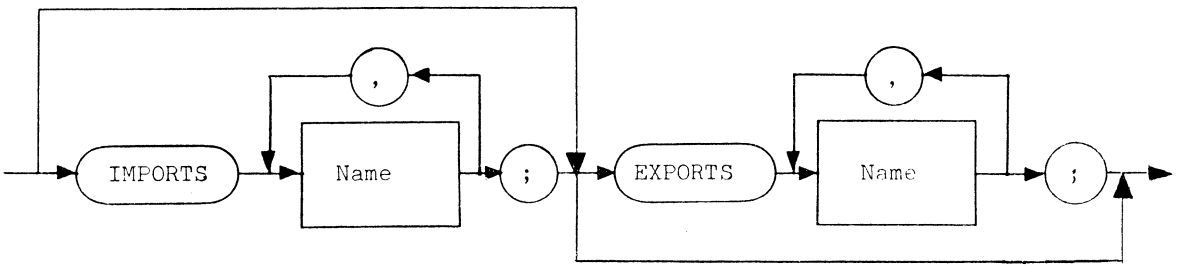


Das Programm, das externe Größen benutzt, wird hier Hauptprogramm genannt; das externe Programm heißt hier Modul.

5.2. Schnittstellenbeschreibung

Die Schnittstellenbeschreibung muß alle Namen, die im Hauptprogramm bzw. Modul deklariert wurden und nach außen zur Verfügung stehen sollen, enthalten.

Schnittstellenbeschreibung:



Die Namen, die hinter IMPORTS stehen, dürfen nicht hinter EXPORTS stehen. Alle Namen, die in der EXPORTS- oder IMPORTS-Liste stehen, müssen im Deklarationsteil definiert werden, damit der Übersetzer den entsprechenden Code erzeugen und Typprüfungen durchführen kann.

Steht ein Prozedur- oder Funktionsname in einer IMPORTS-Liste, so ist die dazugehörige Prozedur bzw. Funktion extern. Der zu diesen Prozeduren bzw. Funktionen gehörende 'Block' wird durch das Wort EXTERN ersetzt.

Werden für die Deklaration eines Namens, der in einer EXPORTS- oder IMPORTS-Liste steht, weitere Namen benutzt, so müssen alle verwendeten Namen exportiert bzw. importiert werden.

Beispiel:

```
PROGRAM PROG01;
EXPORTS V1, T1, C1;
IMPORTS V2;
CONST
  C1  =  255;
TYPE
  T1  =  0..C1;
VAR
  V1 : T1;
  V2 : REAL;
BEGIN
  (* Ausführungsteil *)
END.
```

In diesem Beispiel wurde für die Definition der Variablen V1 der Typ-Name T1 benutzt. Demnach muß, falls V1 exportiert wird, ebenfalls T1 exportiert werden. Bei der Typdefinition von T1 wurde die Konstante C1 verwendet, deshalb muß auch C1 exportiert werden.

Standard-Namen dürfen weder exportiert noch importiert werden; außer wenn sie umdefiniert wurden. In so einem Fall müssen diese Namen in der Liste stehen. Fehlt so eine Auflistung, erzeugt der Compiler eine Typecodierung für die Standardbedeutung, so daß der Binder eine falsche Typprüfung durchführt.

Beispiel:

```
PROGRAM PROG02;
IMPORTS INTEGER, V1;
TYPE
   INTEGER = 0..255;
VAR
   V1:INTEGER;
BEGIN
   (* Ausführungsteil *)
END.
```

Man kann sich Beispiele denken, bei denen die Bedeutung von Vereinbarungen von der Reihenfolge der Typdefinitionen abhängt:

```
PROGRAM PROG03;
EXPORTS INTEGER, T1, V1, V2;
TYPE
   T1 = INTEGER;
   INTEGER = 0..255;
VAR
   V1 : INTEGER;
   V2 : T1;
BEGIN
   (* Ausführungsteil *)
END.
```

Konstruktionen dieser Art sind verboten !

Der Übersetzer erzeugt in diesem Fall für alle exportierten Namen eine Typecodierung, die die Variablen V1 und V2 auf den Bereich 0..255 festlegt.

Ein Sonderfall bei der Schnittstellenbeschreibung bilden vom Benutzer definierte skalare Typen. Dies kann z.B. bei Variablen- Deklarationen der Fall sein:

```
VAR
    V1,V2 : ARRAY   BOOLEAN   OF (ROT,BLAU,GELB);
    V3    :   BLAU .. GELB;
```

Soll die Variable V3 exportiert werden, so müssen die Namen BLAU und GELB in der EXPORTS-Liste stehen. Für eine vollständige Typprüfung ist es notwendig, auch den Typ der Namen GELB und BLAU in die EXPORTS-Liste aufzunehmen. Deshalb muß die Variable, die unmittelbar vor dem Doppelpunkt der Deklaration des skalaren Typs steht, in die Liste aufgenommen werden. Im obigen Beispiel ist dies der Name V2.

Wird ein skalarer Typ innerhalb einer Typedefinition eingeführt,so muß der zugehörige Typname in die Liste aufgenommen werden.

Für die Kompatibilitätsprüfung von externen Größen wird eine identische Deklaration verlangt:

```
VAR
V1 : RECORD A1, B1 : INTEGER END;
V2 : RECORD B1, A1 : INTEGER END;
V3 : RECORD A1:INTEGER; B1:INTEGER END;
V4 : RECORD A1, B1: INTEGER END;
```

In diesem Beispiel sind nur die Namen V1 und V4 typekompatibel.

5.3. Externe Module

Externe Module sind wie PASCAL-Programme mit externen Größen aufgebaut.

Anstelle des Schlüsselwortes PROGRAM wird das Schlüsselwort MODULE eingesetzt. Der Anweisungsteil des Hauptprogrammblocks eines Moduls darf nur die leere Verbundanweisung BEGIN END enthalten.

Beispiel: Es soll ein externes Modul für die Steuerung von Bildschirmfunktionen für den Bildschirm 5423/5424 erstellt werden.

```
MODULE BILDSCHIRMFUNKTIONEN (OUTPUT);
EXPORTS CLEAR, POSI;
PROCEDURE CLEAR;
   (* AUSGABE VON '7E1C = BILDSCHIRM LOESCHEN *)
 BEGIN
  WRITE (CHR(126), CHR(28))
 END;
 PROCEDURE POSI (ZEILE, SPALTE:INTEGER);
   (* AUSGABE VON '7E11 = POSITIONIEREN *)
 BEGIN
  WRITE (CHR(126),CHR(17),CHR(SPALTE),CHR(ZEILE))
 END;
BEGIN
   (* LEERE VERBUNDANWEISUNG *)
END.
```

Der Deklarationsteil des Hauptprogramms, das diesen Modul benutzt, enthält folgende Vereinbarungen.

```
  PROCEDURE CLEAR;
  EXTERN;
  PROCEDURE POSI (ZEILE,SPALTE:INTEGER);
  EXTERN;
```

Das Hauptprogramm und der Modul werden getrennt übersetzt. Beim Binden wird mit Hilfe des Kommandos ENT (vgl. Bedienungsanleitung BINDER) die Zuordnung festgelegt.

Ein weiteres Beispiel soll eine etwas komplexere Schnittstellenbeschreibung erläutern.

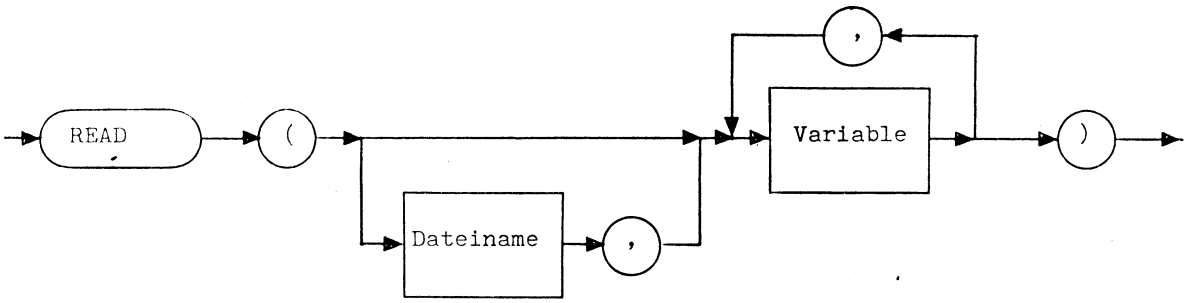
```
PROGRAM PROG04 (INPUT,OUTPUT);
EXPORTS C1, T1, PROZ0, V1;
IMPORTS T2, FCT01, V2, F1, V3, BLAU, ROT, T3;
CONST
  C1 = 10;
TYPE
  T1 = 0..C1;
  T2 = 0..255;
  T3 = ARRAY  CHAR  OF (BLAU,GELB,ROT,GRUEN);
VAR
  V1,V2 : T2;
  V3:BLAU..ROT;
  F1:FILE OF CHAR;
PROCEDURE PROZ0 (F1:T1);
  BEGIN
   (* Ausführungsteil *)
  END;
FUNCTION FCT01 (U:REAL) : INTEGER;
  EXTERN;
BEGIN
 .  (* Hauptprogramm *)
 .
END.
```

```
  MODULE M (F1);
  EXPORTS FCT01, F1, T2, V2, V3, BLAU, ROT, T3;
  TYPE
    T2 = 0..255;
    T3 = ARRAY [CHAR] OF (BLAU,GELB,ROT,GRUEN);
  VAR
    F1 : FILE OF CHAR;
    V2 : T2;
    V3 : BLAU..ROT;
  PROCEDURE LOCAL; BEGIN (* Ausführungsteil *) END;
  FUNCTION FCT01 (U:REAL) : INTEGER;
  BEGIN ... LOCAL ... END;
 BEGIN
     (* Ausführungsteil Modul: leere Anweisung BEGIN END*)
END.
```

6. Ein-Ausgabe-Prozeduren (Input - Output - Procedures)

Wie in Kapitel 1.1 angedeutet, erfolgt der Datentransport zwischen Anwenderprogramm und Peripherie-Geräten (z.B. Bildschirm-Terminal) über die beiden Standarddateien INPUT und OUTPUT, die als Dateinamen hinter den Programmnamen anzugeben sind. Die beiden Standarddateien INPUT und OUTPUT sind Textfiles, d.h.: Dateien, deren Komponente Zeichen (vgl.2.1) sind. Für den Zugriff auf diese beiden Standarddateien und für den Zugriff auf vom Benutzer definierte Textfiles gibt es die Standardprozeduren READ und WRITE.

6.1. Eingabe-Prozedur (Input-Procedure)



Die Prozedur READ darf nur auf Textfiles angewendet werden. Wird der Dateiname nicht angegeben, so erfolgt das Einlesen über die Datei INPUT.

Die Variable muß vom Type CHAR, INTEGER (bzw.Teilbereich von CHAR oder INTEGER), REAL oder Zeichenkette (ARRAY, OF CHAR) sein.

Es gelten folgende Konventionen:

a) Der Prozedur-Aufruf
READ (F,V1, ... Vn)
hat die gleiche Bedeutung wie
BEGIN READ (F,V1); ... READ (F,Vn) END.
(F = Dateiname, Vx = Variable).

b) Ist die Variable (Vx) vom Typ CHAR oder Teilbereich von CHAR, so ist der Prozeduraufruf
READ (F,V1)
gleichbedeutend mit

```
IF SOR (F) = TRUE THEN BEGIN
                       GET (F);
                       V:=F^;
                       GET (F)
                       END  (vgl. 3.6.4.1)
```

c) Ist die Variable vom Typ INTEGER (bzw. Teilbereich davon) oder REAL, so bewirkt der Prozeduraufruf

READ (F,V)

das Einlesen einer Zeichenfolge, die eine vorzeichenbehaftete Ganzzahl (signed integer, vgl. 2.1.3) oder eine vorzeichenbehaftete Realzahl (signed real, vgl. 2.1.3) darstellt. Bei Variablen vom Typ INTEGER muß die eingelesene Zahl zuweisungsverträglich (vgl.3.4.5) sein. Führende Zwischenräume (Blanks) oder Zeilenendezeichen (Carriage Return) werden überlesen.

Das Einlesen wird beendet, sobald ein Zeichen angetroffen wird, das nicht der Syntax einer vorzeichenbehafteten Ganzzahl oder einer vorzeichenbehafteten Realzahl entspricht.

d) Ist die Variable eine Zeichenkette (ARRAY OF CHAR), so wird solange eingelesen, bis die Variable gefüllt ist oder bis ein Zeilenendezeichen gefunden wird.

Eine Erweiterung der Prozedur READ stellt die Prozedur READLN (Read Line) dar. READLN nat die gleiche Parameterliste wie READ. Allerdings darf READLN auch ohne Parameterliste bzw. mit einer Parameterliste, die als einziges Element den Dateinamen enthält, benutzt werden. Die Prozedur READLN bewirkt, daß nach dem Einlesen auf das erste Zeichen nach dem nächsten Zeilenendezeichen geschaltet wird.

Beispiel:

Man nehme an, daß die Standarddatei INPUT folgende Zeichen enthält:

```
4.12  300 CR -3.6 -200
```

Carriage Return (Zeilenendezeichen)

Es soll folgender Vereinbarungsteil gelten:

```
VAR A,C,  :  REAL;
    B,D:INTEGER;
```

Der Prozeduraufruf

```
READ (A,B,C,D)
```

bewirkt, daß der Variablen

A der Wert 4.12,
B der Wert 300,
C der Wert -3.6 und
D der Wert -200 zugewiesen wird.

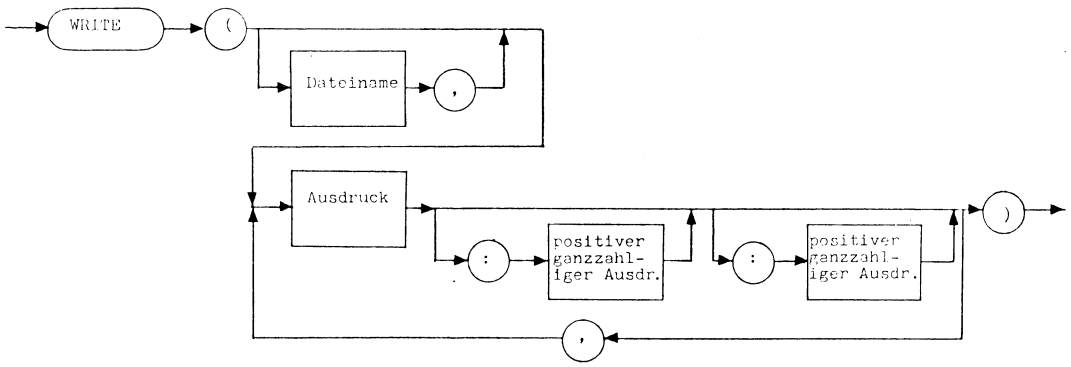
Die Anweisungsfolge
READ(A); READ(B); READ(C); READ(D) hätte die gleiche Wirkung.

Die Programmfolge
READLN(A); READ(C,D)
bewirkt folgende Wertzuweisungen:

```
A    4.12
C   -3.6
D   -200
```

Nach dem Einlesen der Variablen A wird das erste Zeichen nach dem nächsten Zeilenendezeichen, hier das Minuszeichen, mit der nächsten READ-Prozedur eingelesen.

6.2. Ausgabe-Prozedur (Output-Procedure)



Die Ausgabe-Prozedur ist die Umkehrung der Eingabe-Prozedur. Werte werden aus dem Programmspeicher in die Datei, die bei Dateiname eingesetzt ist, ausgegeben. Fehlt diese Angabe, erfolgt die Ausgabe über die Standarddatei OUTPUT.

Der Ausdruck muß vom Typ REAL, INTEGER, CHAR, BOOLEAN oder eine Zeichenkette (ARRAY OF CHAR) sein. Der Wert des Ausdrucks wird ausgegeben.

Die beiden positiven ganzzahligen Ausdrücke können benutzt werden, um die Ausgabe zu formatieren. Der erste ganzzahlige Ausdruck gibt die Feldweite, d.h. die gesamte Anzahl der auszugebenden Zeichen (evtl. einschließlich Vorzeichen, Dezimalpunkt, Exponentialzeichen) an. Der zweite positive ganzzahlige Ausdruck gibt die Anzahl der Nachkommastellen an. Die Anzahl der Nachkommastellen darf nur angegeben werden, wenn der auszugebende Wert vom Typ REAL oder vom Typ ARRAY OF CHAR ist. Bei Werten vom Typ ARRAY OF CHAR darf nur die Feldweite angegeben werden; dabei wird dann die Anzahl der zu verarbeitenden Elemente (Länge) bestimmt.

Entfällt die Formatangabe, erfolgt die Ausgabe im automatischen Format.

Eine Erweiterung der Prozedur WRITE stellt die Prozedur WRITELN (Write Line) dar. WRITELN hat die gleiche Parameterliste wie WRITE. Allerdings darf WRITELN auch ohne Parameterliste bzw. mit einer Parameterliste, die als einziges Element den Dateinamen enthält, benutzt werden. Die Prozedur WRITELN bewirkt, daß nach der Ausgabe zusätzlich ein Zeilenendezeichen ausgegeben wird.

Beispiel:

a) Die Variablen A, B, C und D seien vom Typ REAL und haben die Werte 4.12, 300, -3.6, -200; dann bewirkt die Anweisungsfolge

```
WRITELN (A:8:3); WRITELN (B:8:3);
WRITELN (C:8:3); WRITELN (D:8:3);
```

folgendes Druckbild:

```
   4.120
 300.000
  -3.600
-200.000
```

b) Eine auf einem Plattenspeicher vorhandene Textdatei, in der Briefe gespeichert sind, soll zeilenweise gelesen werden. In der gelesenen Zeile sollen alle Vokale (Selbstlaute) durch das Zeichen * (Stern) ersetzt und die gelesene Zeile auf ein Bildschirmgerät (Standarddatei OUTPUT) ausgegeben werden.

```
PROGRAM TEXTVERARBEITUNG (OUTPUT,BRIEFE);
CONST M=80;
VAR ZEILE : ARRAY  [1..M]  OF CHAR;
    PRUEF : SET OF CHAR;
    I : 1 .. 80;
    Q : 1 .. 80;
    BRIEFE : FILE OF CHAR;
BEGIN
 PRUEF := ['A','E','I','O','U']  ;
 RESET ('BRIEFE',1,1);
 REPEAT
  I := 1;
  REPEAT
   READ (BRIEFE,ZEILE  [I] );
   I := I+1
  UNTIL EOLN (BRIEFE) OR EOF (BRIEFE);
  IF EOLN (BRIEFE) THEN
     BEGIN
       FOR Q := 1 TO I DO
       BEGIN
       IF ZEICHEN [Q] IN PRUEF THEN ZEICHEN [Q] :='*';
       WRITE (ZEICHEN [Q] )
       END;
       WRITELN;
     END;
 UNTIL EOF (BRIEFE);
 RESET (BRIEFE,2)
END.
```

Heinrich Dietz
Solinger Straße 9
4330 Mülheim-Ruhr
Tel.: (0208) 4434-1
Telex 856770

DIETZ Computer SYSTEME