Rückblick

auf die

problemorientierte
Programmiersprache PL/1

für

ehemalige Nutzer

Programmiersprache PL/1 F - OS/360

Merkblätter PL/1 - V.7.3. Datum: 18. September 2015

Bearbeiter: Bernd Hartwich
Diplom Mathematik - Physiklehrer

Vorwort

Die problemorientierte Programmiersprache **PL/1-F** für das **OS/360 Operating System** wurde zur effektiven Lösung von Problemen wissenschaftlich-technischer und kommerzieller Aufgabenstellungen sowie zur Textverarbeitung vorrangig auf Großrechnern der dritten Computergeneration eingesetzt. Diese Sprache PL/1, oft auch als PL/I, PL1 oder PLI abgekürzt, ist eine Programmiersprache, die Mitte der 1960er Jahre von der Firma IBM in Zusammenarbeit mit den Organisationen SHARE und GUIDE entwickelt wurde. Es wurde versucht, die Vorteile aller bis dahin bestehenden Hochsprachen - insbesondere ALGOL 60, FORTRAN und COBOL - zu vereinigen.

PL/1 wurde zumeist auf IBM Großrechnern eingesetzt; es existieren jetzt aber auch Dialekte für Windows, AIX, OS/2 und andere Unix-Varianten. Neuere PL/1-Compiler entsprechen im Sprachumfang dem ISO-Standard 6160 und den 1976 erschienenen PL/1-Normen von ANSI und DIN 66255 von 1980.

In diesem Skript wird die Syntax sowie die Semantik der Sprache PL/1 F entsprechend eines von mir erarbeiteten Schulungskonzeptes auf Basis von Compilern, die auf IBM-Großrechnern installiert wurden, vermittelt. Weiterführende Informationen, insbesondere Details der Ein- und Ausgabe sowie sämtliche Regeln und Besonderheiten zu den angeführten Anweisungen sind bitte der Fachliteratur bzw. der maschinenabhängigen Systemliteratur zu entnehmen. Ebenso können hier die gesamten Sonder- und Spezialfälle zu den einzelnen Konstrukten, vor allem bei der Ein - und Ausgabe, die in den Seminaren einen sehr großen Raum eingenommen haben, nicht näher erläutert werden.

Die vorliegende Gliederung hat sich in ca. 70 Kursen, die ich in der Regel vor Berufseinsteigern nach ihrem Studium gehalten habe, als methodisch nützlich erwiesen und ist aus heutiger Sicht vor allem für ehemalige Nutzer zu Auffrischung der Kenntnisse und gegen das Vergessen gedacht. Sie ist als Erinnerung an eine sehr schöne und interessante Programmiersprache zu verstehen, die sicher viele Anwender jahrzehntelang begleitet hat. Es soll auch ein kleiner Beitrag sein, diese Programmiersprache bei ehemaligen Nutzern in schöner Erinnerung zu behalten.

Ich hoffe, dass die Leser die gleiche Freude beim Durcharbeiten dieses Skripts haben, wie ich bei dessen Erstellung.

Inhaltsverzeichnis **Seite**

1. Grundlagen

1.1. Zur Entwicklung höherer Programmiersprachen

Kurzer geschichtlicher Abriss

Die Entwicklung geht von den anfänglich echten Maschinen-Sprachen über die maschinenorientierten Sprachen zu den **problemorientierten Sprachen** wie **ALGOL 60**, **FORTRAN, COBOL** und **PL/1**. Echte **maschinenorientierte Sprachen**, wie der **Assembler,** kennen nur feste, numerische Werte für die Adressierung von Daten. Mittlere und zunehmend die höheren Sprachen verwenden hingegen symbolische, relative Adressen, d.h. der Programmierer braucht die absoluten Adressen nicht zu kennen. Schon bald nach der Entwicklung von FORTRAN - Formula Translation -, das für technisch-wissenschaftliche Anwendungen konzipiert wurde, und COBOL - Common Business oriented Language -, das für kommerziellen Einsatz gedacht war, stellte man fest, dass eine strenge Trennung der beiden Sprachen wenig zweckmäßig ist. Einerseits kommt es oft vor, dass Benutzer von FORTRAN Probleme mit einer großen Menge Eingabedaten programmieren müssen. Die Programmierung von Ein- und Ausgabe musste den neuen Verhältnissen angepasst werden. Andererseits genügten die beschränkten Verarbeitungsmöglichkeiten des COBOL den kommerziellen Benutzern dieser Sprache nicht mehr. Mit anderen Worten, eine neue, den aktuellen Bedürfnissen angepasste Sprache musste gefunden werden. Zusammen mit SHARE, der Organisation der wissenschaftlichen Computerbenutzer und GUIDE, der Organisation der kommerziellen Computerbenutzer, hat IBM deshalb diese universelle Sprache PL/1 - Programming Language 1 – geschaffen.

1.2. Der Formalismus der Sprachbeschreibung

Zur Beschreibung der Syntax der Sprache begibt man sich außerhalb dieser zu beschreibenden Sprache. Eine sinnvolle Variante zur Beschreibung der Syntax von PL/1 ergibt sich mit der Meta-Sprache von Backus-Naur, die sog. Backus-Naur-Form. Erstmalig wurde die Syntax der Sprache ALGOL 60 vollständig mit dieser Methode beschrieben. In der Beschreibung der Syntax von PL1 wird nur vereinzelt auf diese Notation zurückgegriffen, da mehrere Elemente der Sprache selbst mit den Elementen der Metasprache übereinstimmen und somit eine konsequente Anwendung der **Backus-Naur-Form** verbietet. Doch solange es der Übersichtlichkeit dient, wird von dieser Methode Gebrauch gemacht. Darüber hinaus würde bei einer konsequenten Anwendung der Metasprache für alle möglichen Konstrukte der Umfang dieses Skripts erheblich gesprengt werden; ebenso würde man dem methodisch didaktischen Aufbau der Sprachgerüstes nicht gerecht werden.

Es gibt 5 metalinguistische Konnektoren, die zur Beschreibung der Syntax dienen.

::=	Ergibtzeichen
|	Oder-Zeichen
< >	Beschreibungskontext
{ }	Auswahlkontext
[ ]	eingefügter Kontext findet wahlweise Verwendung

Beispiele:
< ziffer > ::= 0|1|2|3|4|5|6|7|8|9

Lesart:
Eine Ziffer in PL1 ist eine der lateinischen Zahlen, der Kardinalzahlen von 0-9

Allgemeine Form:
<ganze zahl> ::= [{±}] <ziffer> | <ganze zahl> <ziffer>

Lesart:
Man bedient sich der rekursiven Schreibweise. Eine ganze Zahl kann eine vorzeichenbehaftete Ziffer oder eine vorzeichenbehaftete Ziffer und eine weitere Ziffer, somit eine Folge von Ziffern mit wahlweisem Vorzeichen sein.

1.3 PL/1-Zeichenvorrat

PL/1 verfügt über einen Zeichenvorrat von 60 Zeichen.

Allgemeine Form:
1. Gruppe der Alphabetzeichen
<alphabetzeichen> ::= 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|$|@|#
2. Gruppe der Ziffern
< ziffer > ::= 0|1|2|3|4|5|6|7|8|9
3. Gruppe der Sonderzeichen
*<sonderzeichen> ::= blank | . | < | (| + | | | & | $ | * |) | ; | not* | _ | / | , |*
% | - | > | ? |: | # | @ | ' | =

* not steht hier und im folgendem synonym für das z.T. nicht darstellbare bzw. nicht druckbare Negationszeichen (siehe entspr. Systemliteratur). Im Weiteren wird das Negationszeichen generell durch not* dargestellt.

Sortierfolge aufsteigend:

1. Sonderzeichen interne hexadezimale Darstellung ab '40'
2. Alphabetzeichen interne hexadezimale Darstellung ab 'C1'
3. Ziffern interne hexadezimale Darstellung ab 'F0'

Um Zeichenketten- bzw. Bitkettenoperationen durchführen zu können - wie z.B. Sortierungen von Zeichen, Überlagerungen von Zeichen verschiedener Formen, Darstellungen und Veränderungen von Zeichen in einer Bitkette, usw. - kann es sehr hilfreich sein, die Bitstruktur des PL/1-Zeichensatzes zu kennen.

Tabelle der zulässigen PL/1 Zeichen im 60-Zeichensatz

Zeichen	Interner EBCDI-Code dual	hexadezima	Zeichen	Interner EBCDI-Code dual	hexadezimal
blank	0100 0000	40	G	1100 0111	C7
.	0100 1011	4B	H	1100 1000	C8
<	0100 1100	4C	I	1100 1001	C9
(	0100 1101	4D	J	1101 0001	D1
+	0100 1110	4E	K	1101 0010	D2
&	0101 0000	50	L	1101 0011	D3
$	0101 1011	5B	M	1101 0100	D4
*	0101 1100	5C	N	1101 0101	D5
)	0101 1101	5D	O	1101 0110	D6
;	0101 1110	5E	P	1101 0111	D7
not *	0101 1111	5F	Q	1101 1000	D8
_	0110 0000	60	R	1101 1001	D9
/	0110 0001	61	S	1110 0010	E2
|	0110 1010	6A	T	1110 0011	E3
,	0110 1011	6B	U	1110 0100	E4
%	0110 1100	6C	V	1110 0101	E5
-	0110 1101	6D	W	1110 0110	E6
>	0110 1110	6E	X	1110 0111	E7
?	0110 1111	6F	Y	1110 1000	E8
:	0111 1010	7A	Z	1110 1001	E9
#	0111 1011	7B	0	1111 0000	F0
@	0111 1100	7C	1	1111 0001	F1
'	0111 1101	7D	2	1111 0010	F2
=	0111 1110	7E	3	1111 0011	F3
A	1100 0001	C1	4	1111 0100	F4
B	1100 0010	C2	5	1111 0101	F5
C	1100 0011	C3	6	1111 0110	F6
D	1100 0100	C4	7	1111 0111	F7
E	1100 0101	C5	8	1111 1000	F8
F	1100 0110	C6	9	1111 1001	F9

1.4. Bezeichner

Ein Bezeichner ist eine Kette von maximal 31 Alphabetzeichen, Ziffern und Unterstreichungszeichen und ist frei wählbar, mit Ausnahme der Schlüsselwörter (1.5). Der Name des Bezeichners sollte allerdings wegen der besseren Lesbarkeit des Quelltextes im Kontext zu seiner vorgesehenen Verwendung stehen. Er muss mit einem Alphabetzeichen beginnen.

Allgemeine Form:
<bezeichner> ::= <alphabetzeichen> | *<bezeichner><alphabetzeichen>*|
<bezeichner><ziffer> | *<bezeichner><_>*

Bezeichner werden für folgende Zwecke verwendet:

- einfacher Variablennamen	DCL WERT FIXED(5,2);
- Bereichsnamen	DCL MATRIX(10,10) FIXED(5,2);
- Strukturnamen	DCL 1 STRUKTUR, 2 WERT_1 FIXED(5,2); 2 WERT_2 FIXED(5,2);
- Eingangsnamen	HAUPT_PROG: PROC OPTIONS (MAIN);
- Dateinamen	DCL AUSGABE FILE PRINT ENV(F(80));
- Bedingungsnamen	ON CONDITION(TEST) GOTO PRUEF;
- Markennamen	DCL MARKE LABEL (M1,M2,M3);

1.5. Schlüsselwörter

Ein Schlüsselwort legt fest, welche Art einer Vereinbarung oder Anweisung ausgeführt werden soll. Es ist ein Bezeichner, der Bestandteil der Sprache PL/1 ist. Sie sind keine reservierten Namen, sie können frei verwendet werden.

Schlüsselwörter können folgendermaßen klassifiziert werden:

- Schlüsselwörter für Anweisungen:
  READ, OPEN, GOTO, GO TO, IF, THEN, ELSE,...
- Schlüsselwörter für Attribute:
  FIXED, FLOAT, CHARACTER,....
- Schlüsselwörter für Formatelemente:
  SKIP, PAGE, LINE,...
- Schlüsselwörter für Optionen:
  FILE, PRINT, PAGE,...
- Schlüsselwörter für Funktionen:
  DATE, SQRT, TIME,...
- Schlüsselwörter für ON-Bedingungen:
  ON ZERODIVIDE, ON ENDFILE(DATEI), ON ENDPAGE(DRUCK),...
- Schlüsselwörter für Bedingungspräfixe:
  (SIZE): WERT = 123.45;
  (NOZERODIVIDE): WERT = DIVIDEND / DIVISOR;

1.6. Verwendung von Leerzeichen

Der Quelltext ist an keine Restriktionen gebunden, Leerzeichen können als Trennzeichen an allen Stellen eingefügt werden. Der Einsatz des Trennzeichens lässt eine gute und übersichtliche Gestaltung zu. Die Verwendung von Leerzeichen innerhalb von Bezeichnern, Konstanten, zusammengesetzten Operatoren und Schlüsselwörtern (außer GO TO) ist nicht erlaubt.

1.7. Ergibtanweisung

Die am häufigsten verwendete Anweisung für den hauptspeicherinternen Transport von Daten sowie die Berechnung von Ausdrücken ist die Ergibtanweisung. Sie ist die einfachste Form einer Wertzuweisung. Der Compiler arbeitet diesen Anweisungstyp grundsätzlich von rechts nach links unter Berücksichtigung der Priorität der Operationen ab.

Allgemeine Form:
<ergibtanweisung> ::= <bezeichner> = <arithmetischer ausdruck> | *< logischer ausdruck>* |
<vergleichsausdruck> | *<verkettungsausdruck>*

Die verschiedenen Konstrukte der Ausdrucksarten werden im folgendem definiert.

Beispiele:

```
VAR_1 = 25;                    /* DER DEZIMALWERT 25 WIRD VAR_1 ZUGEWIESEN */
INDEX = INDEX + 1;             /* REKURSIVE ANWEISUNG; INDEX WIRD UM 1 ERHOEHT */
                               /* UND INDEX ZUGEWIESEN */
BOOL = WERT_1 = WERT_2;    /* WERT_1 U. WERT_2 WERDEN AUF GLEICHHEIT */
                           /* GEPRUEFT  ERGEBNIS ALS BOOLSCHER WERT */
                                    /*BOOL ZUGEWIESEN */
KETTE = KETTE_1 || KETTE_2;        /* KETTE_1 WIRD MIT KETTE_2  VERKETTET  */                    /*
          UND GESAMTKETTE KETTE ZUGEWIESEN */
```

1.8. Kommentare

Kommentare dienen der Dokumentation der Quellcodes. Sie haben keinen Einfluss auf die Ausführung eines Programms. Sie dürfen eingefügt werden, wo die Verwendung von Leerzeichen erlaubt ist.

Allgemeine Form:
<kommentar> ::= /* <folge von alphabetzeichen, ziffern, sonderzeichen> */

Beispiele:

```
DCL WERT_1  FIXED(5,2);          /*DAS IST EIN KOMMENTAR*/
MATRIX(2,3) = 25;                /* DAS ELEMENT DER 2. ZEILE UND DER 3. SPALTE */
                                 /* DER MATRIX WIRD AUF DEN DEZIMALWERT 25 GESETZT  */
```

2. Datentypen und Attribute

Damit die Compilerroutinen Adressketten aufbauen können, muss der benötigte Speicherplatz explizit, implizit oder textabhängig vereinbart werden. Mit Hilfe der entsprechenden DECLARE-Anweisung können Bezeichner mit den dazugehörigen Attributen explizit vereinbart werden. Die Attribute legen die Darstellungsform eines Wertes im Hauptspeicher fest. Bei einer Ergibtanweisung eines Wertes erfolgt die interne Speicherung dieses Wertes stets entsprechend der vereinbarten Attribute.

Allgemeine Form:
<declare-anweisung>::= { DELCLARE / DCL } <bezeichner> [<attribute>] |
(<liste von bezeichnern>) [<attribute>];

Beispiele:
```
DCL VAR_1 FIXED (7,2);                    /* DEZIMALE FESTKOMMAGRÖSSE*/
DCL KETTE CHAR(100);                      /*ZEICHENKETTE ZUR AUFNAHME VON */
                                  /*ALPHABETZEICHEN, ZIFFERN UND  */
                              /* SONDERZEICHEN */
DCL (VAR_1, VAR_2, VAR_ 3) FIXED (7,2);       /* ALLE GLEICHE ATTRIBUTE */
```

2.1 Übersicht über Datentypen und Attribute

Datentyp	Typattribut
Problemdaten	
- arithmetische Daten	
Zahlenart: reell | komplex	REAL | COMPLEX
Basis: dezimal | binär(dual)	DECIMAL | BINARY
Kommaart: Festkomma | Gleitkomma	FIXED | FLOAT
- numerische Kettendaten	
reell	PICTURE 'numerische abbildungskette'
komplex	COMPLEX PICTURE 'num.abbildungsk.'
- Zeichenkettendaten	CHARACTER (länge der kette)
-Bitkettendaten	BIT (länge der bitkette)
Programmsteuerdaten	
- AREA-Daten	AREA-attribut
- BUILTIN-Daten	BUILTIN
- CONDITION-Daten	CONDITION
- ENTRY-Daten	ENTRY-attribut
- EVENT-Daten	EVENT
- FILE-Daten	FILE
- GENERIC-Daten	GENERIC-attribut
- LABEL-Daten	LABEL-attribut
- TASK-Daten	TASK
- Zeigerdaten	POINTER | OFFSET-attribut

2.2 Arithmetische Daten

Vier Attributtypen dienen der Beschreibung arithmetischer Daten:

1. <basisattribut > *::= [DECIMAL | BINARY] | [DEC | BIN]*
2. <skalierungsattribut> *::= [FIXED | FLOAT]*
3. <modusattribut> *::= [REAL | COMPLEX] | [REAL | CPLX]*
4. <genauigkeitsattribut> *::= [(<genauigkeit>)]*

2.2.1. Dezimale Festkommadaten

Darstellung der Konstanten: durch eine oder mehrere Ziffern mit wahlweisen Dezimalpunkt

Beispiele:
71 entspricht 71
.71 entspricht 0,71
1.7 entspricht 1,7

Anwendung: Dezimale Festkommaoperationen

Allgemeine Form:
<dezimale festkommadate>::= DCL <bezeichner> [REAL] [DEC] FIXED [(p[,q])];

Genauigkeitsattribut: p = Anzahl der Gesamtziffern 1<=p <= 15
q = Anzahl der Ziffern nach Dezimalpunkt -128 <= q <= 127
Standardattribut: wird kein Genauigkeitsattribut angegeben, dann p = 5, q = 0

Speicherung: Die Speicherung erfolgt gepackt und beginnt an der Bytegrenze. Die Stellung des Dezimalpunktes wird im Steuerblock DED vermerkt. Die Bytegröße errechnet sich aus der Beziehung: Byteanzahl = (p + 1) / 2

Beispiele:

```
DCL FESTWERT FIXED (5,2);
FESTWERT = -123.45;                    /* GESPEICHERT WIRD -123,45 */
FESTWERT = -123.456;                   /* GESPEICHERT WIRD -123,45 */
FESTWERT = -1234.5;                    /* FIXEDOVERFLOW */
```

Dritte Wertzuweisung erzeugt einen Festkommaüberlauf; das Programm bricht mit Programmfehler ab. Durch Programmierung geeigneter CONDITION-Routinen ist diese Ausnahmebedingung handelbar.

```
DCL FESTWERT_1 FIXED (2,6);
DCL FESTWERT_2 FIXED (2,-6)
FESTWERT_1 = 12;                       /* GESPEICHERT WIRD 12 000 000 */
FESTWERT_2 = 12;                       /* GESPEICHERT WIRD 0,000 012 */
```

Die Schreibweise p < q eignet sich für sehr große bzw. sehr kleine Werte. Der eingegebene Wert wird intern mit 10 hoch ±q multipliziert.

2.2.2. Dezimale Gleitkommadaten

Darstellung der Konstanten: Mantisse und Exponent zur Basis 10
Beispiele:
71E2 entspricht 71 * 10 hoch 2 = 7100
-.71E-2 entspricht -0,71 * 10 hoch -2 = -0,0071
-78 <= Exponent <= 75

Anwendung: Dezimale arithmetische Operationen, insbesondere Divisionen

Allgemeine Form:
<dezimale gleitkommadate> ::= DCL <bezeichner> [REAL] [DEC] FLOAT [(p)];

Erklärung:
Genauigkeitsattribut: p = Anzahl der Gesamtziffern der Mantisse
1<=p <= 16 (implementierungsabhänig)
Standardattribut: wird kein Genauigkeitsattribut angegeben, dann p = 6,
Speicherung: Die Speicherung als Gleitkommazahl mit normalisierter Mantisse und beginnt an der Wort- bzw. Doppelwortgrenze.
1 <= p <= 6 Ausrichtung Wortgrenze
6 < p <= 16 Ausrichtung Doppelwortgrenze

Beispiele:
```
DCL GLEITWERT_1 FLOAT;                    /* Standardmantisse p = 6 */
DCL GLEITWERT_2 FLOAT (12);
GLEITWERT_1 = -7.1 E6;                    /* GESPEICHERT WIRD -7,1*10h6 */
GLEITWERT_2 = -123.456;                   /* GESPEICHERT WIRD -123,456 */
```

2.2.3. Binäre Festkommadaten

Darstellung der Konstanten: durch eine oder mehrere Dualziffern mit wahlweisen Dualpunkt und abschließendem B

Beispiele:
1111B entspricht 1111 zur Basis 2 = 15
111.1B entspricht 11,1 zur Basis 2 = 7,5
111.11B entspricht 111,11 zur Basis 2 = 7,75

Anwendung: Duale Festkommaoperationen, insbesondere interne Zähler und Indizes.

Wertzuweisungen mit gebrochenen Dualzahlen können zu starken Ungenauigkeiten führen, da nur die Elemente der Dualfolge darstellbar sind

Genauigkeitsattribut: p = Anzahl der Gesamtdualziffern 1<=p <= 31
q = Anzahl der Ziffern nach Dezimalpunkt -128 <= q <= 127
Standardattribut: wird kein Genauigkeitsattribut angegeben, dann p = 15

Allgemeine Form:
<binäre festkommadate>::= DCL <bezeichner> [REAL] BIN FIXED [(p[,q])];

Speicherung: Die Speicherung beginnt an der Halbwort- bzw. Wortgrenze.

1 <= p <= 15 Ausrichtung Halbwortgrenze
15 < p <= 31 Ausrichtung Wortgrenze

Beispiele:

```
DCL DUALWERT BIN FIXED;                 /* STANDARD P=15, */
                                            /* ENTSPRICHT 2↑15 -1 = 32777 */
DUALWERT =  1111B;                          /* GESPEICHERT WIRD 15 */
DUALWERT =  1111;                           /* GESPEICHERT WIRD 1111 */
                                            /* INTERNE UMRECHNUNG */
DUALWERT =  32768;                          /* FIXEDOVERFLOW, MAX 32767 */
```

Dritte Wertzuweisung erzeugt einen Festkommaüberlauf; das Programm bricht mit Programmfehler ab. Durch Programmierung geeigneter CONDITION-Routinen sind diese Ausnahmebedingungen handelbar.

2.2.4. Binäre Gleitkommadaten

Darstellung d. Konstanten: Mantisse und Exponent zur Basis 2 und abschließendem B

Beispiele:

11E2B entspricht 11B * 2 hoch 2 = 1100B = 12
1111E2B entspricht 1111B *2 hoch 2 = 111100B = 60

Anwendung:

Duale Gleitkommaoperationen, insbesondere Divisionen
-260 <= Exponent <= 252 (implementierungsabhängig)

Allgemeine Form:

<binäre gleitkommadate> ::= DCL<bezeichner> [REAL] BIN FLOAT [(p)];

Genauigkeitsattribut: p = Anzahl der Gesamtziffern der Mantisse
1<=p <= 53 (implementierungsabhängig)
Standardattribut: wird kein Genauigkeitsattribut angegeben, dann p = 21

Speicherung: Die Speicherung als Gleitkommazahl mit normalisierter Mantisse und beginnt an der Wort- bzw. Doppelwortgrenze.
1 <= p <= 21 Ausrichtung Wortgrenze
21 < p <= 53 Ausrichtung Doppelwortgrenze

Beispiele:

```
DCL GLEITWERT  BIN FLOAT;       /* STANDARD P=21, */
GLEITWERT =  11E2B;             /* GESPEICHERT WIRD 12 */
GLEITWERT =  1100.1E4B;         /* GESPEICHERT WIRD  12,5 * 2 hoch 4 */
```

2.3. Nichtdeklarierte Bezeichner - implizite DCL's

Stößt der Compiler auf Bezeichner, die nicht explizit vereinbart wurden, werden Standardanahmen angenommen.
Ist das erste Zeichen des nichtdeklarierten Bezeichners ein
I | J | K | L | M | N dann wird BINARY FIXED (15,0) generiert, wenn
A bis H und O bis Z dann wird FLOAT DECIMAL (7) generiert.

Beispiel:

```
ILAUF = ILAUF + 1;              /* ILAUF NICHT DEKLARIERT, BIN FIXED(15,0) */
K_INDEX = K_INDEX + 1;          /* INDIZES EINER BEREICHSGRENZE ITERATIV ERHOEHT */
```

2.4. Kettendaten

Kettendaten - Zeichenketten und Bitketten - sind in PL/1 eine sehr gute Alternative zur arithmetischen Verarbeitung von Daten. Kettendaten werden vorrangig zur Textverarbeitung eingesetzt, Bitketten zur Auswertung von logischen Ausdrücken.

2.4.1. Zeichenketten

Darstellung der Konstanten: in Hochkomma Zeichen entsprechend PL/1 Zeichenvorrat

Beispiele:

```
' '                 /* LEERKETTE */
''                  /* NULLKETTE */
'BERLIN'
'_____'
(156)'_'            /* WIEDERHOLUNGSFAKTOR 156 _ ZEICHEN*/
```

Die maximale Länge einer konstanten Zeichenkette beträgt 1007 Byte. (implementierungsabhängig)

Anwendung: Einsatz in der Textverarbeitung

Allgemeine Form:
<zeichenkette> ::= DCL < bezeichner > {CHARACTER | CHAR} (p);

Genauigkeitsattribut: p = Gesamtanzahl der zu vereinbarten Zeichen
1<=p <= 32767 (implementierungsabhänig)
Speicherung:
Die Speicherung erfolgt pro Zeichen ein Byte linksbündig und wird bei Bedarf rechts mit Leerzeichen aufgefüllt.

Beispiele:

```
DCL HAUPTSTAEDTE CHAR (6);
HAUPTSTAEDTE = 'BERLIN';   /*SPEICHERUNG 'BERLIN'*/
HAUPTSTAEDTE = 'ROM';              /*SPEICHERUNG 'ROM   ' AUFFUELLUNG 3 LEERZ:*/
HAUPTSTAEDTE = 'WARSCHAU';     /*SPEICHERUNG 'WARSCH'  */
                                        /*BEDINGUNG STRINGSIZE WIRD GESETZT */
```

```
DCL WERT FIXED, KETTE CHAR(5);
WERT = 12345;                    /*SPEICHERUNG GEPACKT 3 BYTE*/
KETTE = '12345';                 /*SPEICHERUNG  5 BYTE */
```

2.4.2. Bitketten

Darstellung der Konstanten: in Hochkomma eingeschlossene Bitzeichen '0' und '1' mit folgendem B. Bitketten werden vorrangig bei logischen Operationen eingesetzt, wobei '0'B für falsch und '1'B für wahr steht.

Beispiele:

```
'00000000'B          /*SPEICHERUNG 1 BYTE   */
(8)'0'B
 '11000001'B         /* SPEICHERUNG HEX 'C1' =  'A'  */
 '11000010'B         /* SPEICHERUNG HEX 'C2' =  'B'  */
```

Die maximale Länge einer konstanten Bitkette beträgt 4088 Bits.

Anwendung:
1. Möglichkeit, in die Bitstruktur eines Zeichens einzugreifen
2. Einsatz in logischen Operationen

Neben ihren logischen Wert besitzen Bitketten auch numerische Werte 0 und 1

Beispiel:

```
ERG = ( W1 < W2 ) + ( W3 < W4);     ERG kann Werte zwischen 0 und 2 annehmen
```

Allgemeine Form:

<bitkette > ::= DCL <bezeichner> BIT (p);

Genauigkeitsattribut: p = Gesamtanzahl der zu vereinbarten Bits
1<=p <= 32767 (implementierungsabhängig)

Speicherung: Die Speicherung erfolgt linksbündig und wird bei Bedarf rechts mit binären Nullen aufgefüllt.

Beispiele:

```
DCL BITKETT_1 BIT (8);
BITKETT_1 = '11000001'B;          /*SPEICHERUNG 'A'   */
BITKETT_2 = '11000010'B;          /*SPEICHERUNG 'B'    */
```

2.4.3. Variable Ketten - VARYING-Attribut

Zeichen- und Bitketten können mit variabler Länge vereinbart werden. Die aktuelle Länge hängt von der letzten Wertzuweisung ab. Der aktuelle Längenschlüssel wird in einem gesonderten, zwei Byte langen Steuerfeld abgelegt und kann durch die Standardfunktion LENGTH bereitgestellt werden.

Allgemeine Form:

<variable kette> ::= DCL <bezeichner> {CHAR |BIT} (p) {VARYING | VAR};

Genauigkeitsattribut:

p = Gesamtanzahl der zu vereinbarten BIT oder CHARACTER - Zeichen

Speicherung:
Die Speicherung erfolgt linksbündig. Der Längenschlüssel wird aktualisiert. Der restliche Speicherbereich bleibt unberücksichtigt.
Anwendung: Verarbeitung sehr großer variabler Ketten, insbesondere bei Parameterübergabe

Beispiele:

```
DCL HAUPTSTAEDTE CHAR (20) VAR;
HAUPTSTAEDTE = 'BERLIN';          /*SPEICHERUNG 'BERLIN' */
HAUPTSTAEDTE = 'ROM';             /*SPEICHERUNG 'ROM'*/
HAUPTSTAEDTE = 'WARSCHAU';        /*SPEICHERUNG 'WARSCHAU'*/
```

Hinweis:
Das Steuerfeld muss bei Überlagerungen (z.B. DEFINED-Attribut) berücksichtigt werden.

2.5. Abbildungsketten

Im Unterschied zu den Zeichenketten der Form CHAR(p) wird bei den Abbildungsketten nicht nur die Anzahl der Zeichen für die aufzunehmenden Daten festgelegt, sondern außerdem die Art der Zeichen auf bestimmten Positionen definiert.

Allgemeine Form:
<abbildungskette> ::= DCL <bezeichner> {PICTURE | PIC} ' <abbildungsspezifikationen>';
<abbildungsspezifikationen> ::= { X | A | 9}

X – beliebige Zeichen, Wertevorrat wird im Folgenden definiert
A – alphabetisches Zeichen oder Leerzeichen
9 – numerisches Zeichen oder Leerzeichen

Die maximale Anzahl der Abbildungszeichen beträgt 32 Zeichen. Jedes Zeichen, außer dem Abbildungszeichen V, belegt ein Byte Speicherplatz. Durch Wahl geeigneter Abbildungsspezifikationen ist man in der Lage, jede beliebige Ausgabeschablone zu entwickeln, die jeder Anforderung gerecht wird.

2.5.1. Zeichenabbildungsketten

Der Vergleich zu Zeichenketten drängt sich unmittelbar auf. Durch die Wahl geeigneter Abbildungsspezifikationen können in einer Abbildungskette an bestimmten Positionen numerische Daten erzwungen werden. Im Fall des Auftretens eines für eine Position nicht zulässigen Zeichens wird die programmierbare Ausnahmebedingung CONVERATION gesetzt.

Beispiel:

```
DCL KET_1 CHAR (5),
KET_2 PIC 'XXXXX',
KET_3 PIC '(5)A',
KET_3 PIC 'XX99A';
```

KET_1 und KET_2 können jede beliebige Zeichenketten aufnehmen, KET_3 darf nur ein alphabetisches Zeichen oder ein Leerzeichen enthalten. In KET_4 wird zusätzlich festgelegt, dass auf der 3. und 4.Position nur numerische bzw. Leerzeichen und auf der letzten Position nur ein alphabetisches Zeichen oder ein Leerzeichen stehen darf.

2.5.2. Numerische Abbildungsketten

Numerische Kettendaten haben einen arithmetischen und einen Zeichenkettenwert. Intern werden sie ungepackter Form gespeichert. Bei arithmetischen Operationen und bei Ergibtanweisungen mit arithmetischen Zielvariablen erfolgt stets eine Typumwandlung vom numerischen Kettentyp in den arithmetischen Typ.
Ein großes Einsatzgebiet ergibt sich in der Druckbildaufbereitung.

2.5.2.1. Abbildungszeichen Ziffer und Dezimalpunkt

9 – Position der Ziffer innerhalb der Kette
V – angenommene Stellung des Dezimalpunktes

Beipiele:

```
DCL NUMKETT_1 PIC '999',     /* WERTEVORRAT = 0...+999 */
NUMKETT_2 PIC '(5)9',        /* WERTEVORRAT = 0...+99999 */
NUMKETT_3 PIC '999V99',      /* WERTEVORRAT = 0...+999,99 */
NUMKETT_4 PIC 'V99999';      /* WERTEVORRAT = 0...+0,99999 */

NUMKETT_1 = 375;       /* SPEICHERUNG 3 BYTE */
NUMKETT_2 = 100000;  /* BEDINGUNG SIZE */
NUMKETT_3 = 123.45;   /* SPEICHERUNG 5 BYTE */
NUMKETT_4 = 0.1;       /* SPEICHERUNG 5 BYTE */
NUMKETT_5 = 0,1;       /* BEDINGUNG CONVERTION */
```

2.5.2.2. Abbildungszeichen zur Vornullenunterdrückung

Z – führende Nullen werden durch Leerzeichen ersetzt
* – führende Nullen werden durch * ersetzt

Beispiele:

```
DCL DRUCKKETT_1 PIC 'ZZ999',     /* WERTEVORRAT = 0...+99999 */
DRUCKKETT_2 PIC '(3)Z99',        /* WERTEVORRAT = 0...+99999 */
DRUCKKETT_3 PIC 'ZZ9V99',        /* WERTEVORRAT = 0...+999,99 */
DRUCKKETT_4 PIC '**9V99';        /* WERTEVORRAT = 0...+999,99 */
DRUCKKETT_1 = 5;                 /* DARSTELLUNG   '005' */
DRUCKKETT_2 = 5;                 /* DARSTELLUNG   '05'*/
DRUCKKETT_3 = 0.5;               /* DARSTELLUNG   '0.50'*/
DRUCKKETT_4 = 1.5;               /* DARSTELLUNG   '**1.50 */
```

2.5.2.3. Abbildungszeichen zur Druckaufbereitung

Sie beschreiben keine Zifferpositionen; die entsprechenden Abbildungszeichen werden zwischen den Ziffern eingefügt.

, – entsprechende Stelle wird ein Komma eingefügt
. – entsprechende Stelle wird ein Dezimalpunkt eingefügt
B – entsprechende Stelle wird ein Leerzeichen eingefügt

Beispiele:

```
DCL DRUCKKETT_1 PIC '999,99',          /* WERTEVORRAT = 0...+99999 */
      DRUCKKETT_2 PIC '9B999V,99',     /* WERTEVORRAT = 0...+9999,99 */
      DRUCKKETT_3 PIC 'ZZ9V,99',       /* WERTEVORRAT = 0...+999,99 */
      DRUCKKETT_4 PIC '**9V,99';       /* WERTEVORRAT = 0...+999,99 */

DRUCKKETT_1 = 12345;                   /* DARSTELLUNG   '123,45' */
DRUCKKETT_2 = 123456;                  /* DARSTELLUNG   '1 234,56'*/
DRUCKKETT_3 = 0.5;                     /* DARSTELLUNG   '0,50'*/
DRUCKKETT_4 = 1.5;                     /* DARSTELLUNG   '**1,50 */
```

2.5.2.4. Abbildungszeichen für Vorzeichen

Vorzeichen können statisch und driftend auftreten.

S – Wert des Feldes >= 0, dann wird + Zeichen eingefügt
    Wert des Feldes < 0, dann wird - Zeichen eingefügt
+ – Wert des Feldes >= 0, dann wird + Zeichen eingefügt
    Wert des Feldes < 0, dann wird Leerzeichen eingefügt
- – Wert des Feldes >= 0, dann wird Leerzeichen eingefügt
    Wert des Feldes < 0, dann wird - Zeichen eingefügt

Beispiele:

```
DCL DRUCKKETT_1 PIC 'S999V,99',        /* WERTEVORRAT = 0...+999,99 */
      DRUCKKETT_2 PIC '9B999V,99S',    /* WERTEVORRAT = 0...+9999,99 */
      DRUCKKETT_3 PIC 'SS999V,99',     /* WERTEVORRAT = 0...+999,99 */
      DRUCKKETT_4 PIC 'S999V,99';      /* WERTEVORRAT = 0...+999,99 */

DRUCKKETT_1 = 123.45;                  /* DARSTELLUNG   '+123,45' */
DRUCKKETT_2 = 123456;                  /* DARSTELLUNG   '1 234,56+'*/
DRUCKKETT_3 = -1.23;                   /* DARSTELLUNG   '  -1,23' */
DRUCKKETT_4 =  1.5;                    /* DARSTELLUNG   ' +1,50 */
```

2.6. Anfangsinitialisierung

Alle Datentypen können auf einen definierten Anfangswert gesetzt werden. Der Einsatz ist besonders sinnvoll für Zählgrößen sowie für statische, unveränderbare Daten.

Allgemeine Form:

<initialisierung> ::= DCL <typvereinbarung> {INITIAL | INIT} (folge von zeichen);

Beispiele:

```
DCL BLATTZAEL BIN FIXED (15) INIT (0);
besser
DCL IBLATTZAEL INIT (0),                              /*  ANFANGSWERT = 0 */
      PHI FIXED (3,2) INIT (3.14),                    /*  ANFANGSWERT = 3,14 */
      DRUCKFELD PIC '*999,99S' INIT (0),
      AUSGABEKETTE CHAR (125) VAR INIT ((125)' ');    /* LOESCHEN */


mit Mehrfachzuweisung
DCL (W_1, W_2, W_3) FIXED (5,2) INIT ((3)0);          /*  ALLE WERTE = 0 */
DCL (KET_1, KET_2) CHAR (10) INIT ((2)(10) ' ');      /* ALLE KETTEN LEER */
```

2.7. Konvertierungsregeln

Unter Konvertierung versteht man die Umwandlung eines Datenelements von einer Darstellungsform in eine andere. Konvertierungen treten vor allem bei der Berechnung von arithmetischen Ausdrücken mit Operanden unterschiedlichen Typs auf. Die notwendigen Konvertierungsroutinen werden vom Compiler generiert und sind für die erzielten Ergebnisse verantwortlich. Bei möglichen Ungenauigkeiten der Ergebnisse ist die Kenntnis der Regeln der Konvertierung für den Programmierer unverzichtbar.
Die Problemdatenumwandlungen werden eingeteilt in Typumwandlungen und arithmetische Umwandlungen.
Typumwandlungen finden zwischen den verschiedenen Datentypen statt:

- arithmetischer Typ
- numerischer Kettentyp
- Zeichenkettentyp
- Bitkettentyp.

Arithmetische Umwandlungen können eine Änderung der

- Zahlenart,
- Zahlenbasis,
- Kommaart,
- Genauigkeit

zur Folge haben.

Voraussetzung für eine erfolgreiche Umwandlung ist die Verträglichkeit der Daten, ansonsten findet eine Programmausnahme mit Setzen der entsprechenden Ausnahmebedingung statt.
Eine vollständige Aufzählung aller Fälle, in denen Problemdatenumwandlungen stattfinden, kann hier nicht behandelt werden. Die einschlägigen Konvertierungsregeln sind der vorhandenen Systemliteratur zu entnehmen.
Beispiele werden im Abschnitt "Arithmetische Operationen mit gemischten Charakteristiken" gezeigt.

3. Datenverknüpfungen

Für die Abarbeitung eines Programms sind neben der Ein- und Ausgabe von Daten zahlreiche interne Operationen notwendig, die bestimmte arithmetische Berechnungen, logische Verknüpfungen sowie die Durchführung von Vergleichen aller Schattierungen veranlassen können. Darüber hinaus müssen Operationen definiert werden, die die Möglichkeiten der Textverarbeitung in Zeichen- bzw. Bitketten sowie deren Verknüpfungen mit allen anderen möglichen Operationen realisieren.

3.1. Verwendung bestimmter Zeichen als Operatoren

Zur Ausführung arithmetischer Operationen mit den jeweils angegebenen Operanden stehen folgende Operatoren zur Verfügung:

Allgemeine Form:
<arithmetischer operator> ::= ** | * | / | + | -*
<vergleichsoperator> ::= < | = | <= | = | = | >= | > | >
*<logischer operator> ::= not** | | | *&*
< kettenoperator > ::= ||

3.2. Reihenfolge der Abarbeitung der Operatoren

Zur eindeutigen Umsetzung der Operationen durch den Compiler in Maschineninstruktionen sind Prioritäten der Auflösung dieser Anweisungen unumgänglich, die der Programmierer beherrschen muss. Die alte Regel, Punktrechnung geht vor Strichrechnung, gilt in Erweiterung auf alle Operationen uneingeschränkt.

Prioritätsliste der Abarbeitung:

1.)	Klammer	Klammernrechnung
2.)	not* | Vorzeichen + | Vorzeichen - | **	Negation, Präfixoperationen, Potenzierung
3.)	* | /	Multiplikation, Division
4.)	+ | -	Addition, Subtraktion
5.)	II	Verkettungsoperator
6.)	< | = | <= | = | = | >= | > | >	Vergleichsoperator
7.)	&	Konjunktion
8.)	|	Disjunktion

3.3. Einfache Ausdrücke

Ein Ausdruck ist ein Algorithmus, der zur Errechnung eines bzw. mehrerer Werte verwendet wird. Ausdrücke werden in drei Typen unterschieden:

1.) - einfacher Ausdruck - einzelner Wert
2.) - Bereichsausdruck - eine Gruppierung von Werten gleichen Typs
3.) - Strukturausdruck - eine Gruppierung von Werten unterschiedlichen Typs

Der Typ des Ergebnisses ist gleich dem Typ der Operanden. Sind beide Operanden von unterschiedlichem Typ, so findet eine Datenkonvertierung nach festen Regeln statt. Ausdrücke lassen sich nach der Art der Operation in vier Gesichtspunkte einteilen:

1.) - arithmetische Ausdrücke
2.) - logische Ausdrücke
3.) - Vergleichsausdrücke
4.) - Verkettungsausdrücke

3.3.1 Arithmetische Ausdrücke

Ein arithmetischer Ausdruck jeden Umfangs setzt sich aus einer Menge elementarer Operationen zusammen. Arithmetische Operationen werden nur mit Daten in verschlüsselter Form durchgeführt.

Allgemeine Form:
*<arithmetischer ausdruck> ::= { + | - } <bezeichner> | <bezeichner> { ** | * | / | + | - } <bezeichner>*

Beispiele:
```
WERT = -WERT;   /* DURCH PREAFIX – WIRD D. AKTUELLE WERT MIT -1 MULTIPLIZIERT */
```

Folgende Formeln sind zu programmieren:
u = 2π r v = 1/3 π r^2 h v = 1/6 π d^3 x = (a + bc)d + (a+b)e x = a / -b

```
U = 2 * 3.14 * R;
V = 1 / 3 * 3.14 * R **2 * H;
V = 1 / 6 * 3.14 * D **3;
X = (A + B * C) * D + (A + B) * E;
X = A / (-B);
```

Regeln zur Bildung arithmetischer Ausdrücke:

1.) Arithmetische Operatoren dürfen niemals aufeinander folgen. Im Bedarfsfall sind Klammern zu setzen.

Beispiele:
```
WERT = W1 /- W2;                    /* SEVERE ERROR COMPILATION */
WERT = W1 / (-W2);
WERT = W1 **-1;                     /* SEVERE ERROR COMPILATION  */
WERT = W1 ** (-1);
```

2.) Arithmetische Klammerausdrücke dürfen niemals aufeinander folgen.

Beispiele:
```
C = (A +B ) (A - B);                /* SEVERE ERROR COMPILATION  */
C = (A + B) * (A - B);
```

3.3.1.1 Gemischte Charakteristiken bei einfachen Ausdrücken

Die beiden Operanden können sich unterscheiden in

1. der Form (FIXED und PIC)
2. der Basis (DEC und BIN)
3. der Skala (FIXED und FLOAT)
4. dem Modus (REAL und CPLX)
5. der Genauigkeit (p und q)

Zu 1. Unterscheidet sich die Form, wird das Ergebnis in verschlüsselter Form (FIXED) bereitgestellt.

Beispiel:

```
DCL (W1, W2) FIXED,
      W3 PIC '(5)9';
      W1 = W2 + W3;    intern wird eine Pseudovariable (FIXED(5,0)) generiert
```

Zu 2. Unterscheiden sich die Basen, wird der dezimale Operand in den dualen umgewandelt

Beispiel:

```
Quelle: DEC FIXED (p1,q1)
Ziel: BIN FIXED (p2, q2) mit
                  p2 = MIN (31,1 + CEIL(p1 * 3.32))
                  q2 = CEIL (q1 * 3.32)
DCL  W1 FIXED (5,2),
      W2 BIN FIXED,....
      ERG = W1 + W2;     W1 wird intern zu BIN FIXED konvertiert
      p2 = MIN(31, 1 + CEIL(5 * 3.32))
            = MIN(31, 18)
      q2 = CEIL( 2 * 3.32)
         = 6
Die interne Umwandlung, die Größe der Pseudovariablen, hat die Attribute BIN FIXED (18,6)
```

Es ist zu beachten, dass bei diesen internen Umwandlungen Ungenauigkeiten entstehen können, die unerwartete Ergebnisse zur Folge haben können.

Beispiel:

```
DCL IZAEL INIT(1B),
      IZAEL = IZAEL + 0.1;
```

Bei Anwendung der obigen Formel ergibt die interne Darstellung der dezimalen 0,1 eine Pseudovariable BIN FIXED(5,4), d.h. 0.0001B = 1/2^4 = 0,0625

Zu 3. Unterscheiden sich die Skalen, so wird Festkomma in Gleitkomma umgewandelt.

Beispiel:

```
DCL VAR1 FIXED (7,2) INIT(17.77),
      VAR2 FLOAT INIT;
      VAR2 = VAR1 * VAR2;     Der Exponent stellt den gebrochenen Anteil der Quelle dar
                              (z.B. 17.77  ergibt intern .1777E2)
```

Da die Regel gilt, die Größe der Mantisse entspricht dem p des Quellfeldes, so wird die Genauigkeit der Pseudovariablen mit FLOAT(6) generiert.

Zu 4. Unterscheidet sich der Modus, so wird der reelle in einen komplexen Operanden umgewandelt, wobei sein imaginärer Anteil als Null angenommen wird.

Zu.5. Bei unterschiedlichen Genauigkeiten findet keine interne Umwandlung statt; das Ergebnis hängt von der Art der Attribute ab (siehe 3.3.1.2).

3.3.1.2 Grenzfälle arithmetischer Operationen

Bei der Festkommaarithmetik werden die Pseudovariablen, falls Zwischenergebnisse vom Compiler generiert werden müssen, nach folgenden Regeln definiert:

- Addition, Subtraktion

1. Operand FIXED DEC (p1,q1)
2. Operand FIXED DEC (p2,q2)
Ergebnis: p = 1 + MAX(p1-q1, p2-q2) + MAX(q1, q2)
q = MAX(q1,q2)

Beispiel:

```
DCL W1 FIXED (5,2),
    W2 FIXED (7,3),
    ERG FIXED (8,3);    Wie groß muss die Genauigkeit p und q des Ergebnisfeldes sein?
ERG = W1 + W2;
  p = 1 + MAX(3,4) + MAX(2,3) = 8
  q = MAX(2,3) = 3
```

- Multiplikation

1. Operand FIXED DEC (p1,q1)
2. Operand FIXED DEC (p2,q2)
Ergebnis: p = 1 + p1 + p2
q = q1 + q2

Beispiel:

```
DCL W1 FIXED (5,2),
    W2 FIXED (7,3),
    ERG FIXED (13,5);    Wie groß muss die Genauigkeit p und q des Ergebnisfeldes sein?
ERG = W1 * W2;
          p = 1 + 5 + 7 = 13,  q = 2 + 3 = 5
```

- Division

1. Operand FIXED DEC (p1,q1)
2. Operand FIXED DEC (p2,q2)
Ergebnis: p = 15
q = 15 - ((p1 - q1) + q2)

Beispiel:

```
DCL W1 FIXED (5,2),
    W2 FIXED (7,3),
    ERG FIXED (15,9);    Wie groß muss die Genauigkeit p und q des Ergebnisfeldes sein?
```

ERG = W1 / W2;
p = 15
q = 15 - 6 = 9

- Potenzierung
  1. Operand FIXED DEC (p1,q1)
  2. Operand Exponent, vorzeichenlose ganze Dezimalzahl n
  Ergebnis: p = (p1 +1) * (n -1)
  q = q1 * n

Beispiel:
DCL W1 FIXED (12,6),
ERG FIXED (15,9);
Wie groß muss die Genauigkeit p und q des Ergebnisfeldes sein?
ERG = W1 **3; p = 6 * 2 = 12 q = 2 * 3 = 6

3.3.2 Logische Operationen, Bitkettenoperationen

Da logische Operationen nur auf Bitketten anwendbar sind, werden sie auch als Bitkettenoperationen bezeichnet.

Allgemeine Form:
<bitkettenoperation> ::=
<bezeichner> = not <bezeichner>* | *<bezeichner> & <bezeichner>*| *<bezeichner>* | *<bezeichner>*

Das Ergebnis einer Bitkettenoperation ist eine Bitkette. Die Operationen werden Bit für Bit von links nach rechts nach folgenden Regeln durchgeführt:

		Negation	Konjunktion	Disjunktion
a	b	not* a	a & b	a | b
1	1	0	1	1
1	0	0	0	1
0	1	1	0	1
0	0	1	0	0

Beispiel:
```
DCL (BK1, BK2, BK3) BIT (8);
BK1 = '11110000'B;
BK2 =  not* BK1;
BK3 =  BK1 & BK2;                    /* BK3 = '00000000'B */
BK3 =  not* (BK1 | BK2);             /* BK3 = '00000000'B */
```

Sind die Operanden von unterschiedlicher Länge, so wird der kürzere auf der rechten Seite mit binären Nullen bis zur Länge des längeren Operanden erweitert. Ist das Ergebnisfeld kürzer vereinbart, so werden die entsprechenden Stellen rechts gestrichen.
Sind in einer Ergibtanweisung mehrere Bitkettenoperationen vorhanden, so gelten bei der Abarbeitung folgende **Prioritätsregeln:**

1. Negation not*
2. Konjunktion &
3. Disjunktion |

Beispiel:

```
DCL (BK_1, BK_2, BK_3) BIT (8);
      BK_1 = '11110000'B;
      BK_2 = '00001111'B;
      BK_3 = '00000000'B;
      BK_3 = not* (BK_1 & BK_2 | BK_3);          /* BK_3 = '11111111'B */
```

3.3.3. Kettenoperationen

Verkettungen dienen zum Zusammenfügen einzelner Ketten. Kettenoperationen sind auf Zeichen- und Bitketten anwendbar.

Allgemeine Form:

<kettenoperation> ::= <bezeichner> = <bezeichner> || <bezeichner>

Sind die Operanden vom unterschiedlichen Typ, so finden folgende Umwandlungen statt:

1. ein Operand vom Typ CHAR, der andere vom Typ DEC oder PIC, so findet eine Umwandlung in CHAR statt und das Ergebnis ist vom Typ CHAR
2. ein Operand vom Typ BIT, der andere vom Typ CHAR, so findet eine Umwandlung in CHAR statt und das Ergebnis ist vom Typ CHAR
3. ein Operand vom Typ BIT, der andere vom Typ BIN, so findet eine Umwandlung in BIT statt und das Ergebnis ist vom Typ BIT

Beispiele:

```
DCL ZK_1 CHAR (4) INIT ((4)'A'),
ZK_2 CHAR (7) INIT ((7)'B'),
BK_1 BIT (8) INIT ((4) '10'B),
BK_2 BIT (2) INIT ((2) '01'B),
BF BIN FIXED (8) INIT (2),
DF DEC FIXED (1) INIT (8);
      .......
ZIELK1 = ZK_1 || ZK_2;                          /* 'AAAABBBB' */
ZIELK2 = BK_1 || BK_2;                          /* '1010101001'B */
ZIELK3 = ZK_1 || BK_2;                          /* 'AAAA01' */
ZIELK4 = ZK_1 || BK_1;                          /* 'AAAA10101010' */
ZIELK5 = BK_2 || BF;                            /* '0100000010'B */
ZIELK6 = BF || BF;                              /* '0000001000000010'B */
ZIELK7 = BF || DF;                              /* '.....2...8' */   Punkt steht für Leerzeichen
ZIELK8 = 'PREIS = ' || 25 || '.000' || ' EUR';  /* 'PREIS= 25.000EUR' */
```

3.3.4 . Vergleichsoperationen

Ein Vergleichsausdruck stellt die Verknüpfung zweier Operanden durch einen Vergleichsoperator dar.

Allgemeine Form:

<vergleichsoperation> ::= <bezeichner> = <bezeichner> <vergleichsoperator> <bezeichner>

Vergleichsausdrücke werden vorrangig in entsprechenden Vergleichsoperationen eingesetzt und dienen der Programmsteuerung. Es gibt drei Typen von Vergleichen:

1. Algebraischer Vergleich

Vergleiche numerischer Werte mit Vorzeichen werden in verschlüsselter Form durchgeführt. Numerische Datenfelder werden umgewandelt.
Das Ergebnis eines Vergleiches ist eine Bitkette der Länge 1 mit den Werten '0'B für falsche Aussage und '1'B für wahre Aussage.

Beispiel:

```
DCL (W1, W2) FIXED (5,2),
ERG BIT (1);
 .......
W1 = -345.43;
W2 = 500;
ERG = W1 > W2;                          /* ERG = '0'B */
ERG = ( W1 *( -1)) > ( 0.5 * W2);       /* ERG = '1'B */
ERG = W1 = W2;                          /* ERG = '0'B */
.....
W2 = W1 + (W1 > W2) * (W1 - W2);        /* W2 = -345.43 */
```

2. Vergleich von Zeichen

Vergleiche von Zeichen werden von links nach rechts Stelle für Stelle durchgeführt. Wird Ungleichheit festgestellt, so wird der Bedingungsschlüssel gesetzt, mit dem Vergleichsoperator verglichen und die Ergebnisbitkette der Länge 1 gesetzt. Das Vergleichsergebnis hängt von der Sortierfolge der zu vergleichenden Zeichen ab. Haben die Operanden unterschiedliche Länge, so wird der kürzere rechts mit Leerzeichen bis zur Länge des längeren aufgefüllt.

Beispiel:

```
DCL   KET1 CHAR(10) INIT ('BER LIN'),
KET2 CHAR(10) INIT ('BERLIN'),
ERG  BIT (1);
 .......
ERG = KET1 = KET2;                      /* ERG = '0'B */
```

3. Vergleich von Bits

Der bitweise Vergleich von Ergebnisketten logischer Operationen erfolgt von links nach rechts Stelle für Stelle durchgeführt. Haben die Operanden unterschiedliche Länge, so wird der kürzere rechts mit binären Ziffern bis zur Länge des längeren aufgefüllt.

Beispiele:

```
DCL  BK1 BIT(8),
BK2 BIT(4);
.......
BK1 = '1010000'B;
BK2 = '1010'B,
ERG =  BK1 = BK2;                       /* ERG = '1'B */
ERG =  BK1 > BK2;                       /* ERG = '1'B */
ERG =  BK1 = BK2;                       /* ERG = '0'B */
```

Sind die Operanden vom unterschiedlichen Typ, so finden wird der Operand mit dem niederen Typ in den Typ des höheren umgewandelt, wobei die folgende **Rangfolge** gilt:

1. arithmetisch (höchster Rang)
2. Zeichenkette
3. Bitkette

Beispiele:

```
DCL  FW FIXED (5) INIT(-15),
ZK CHAR (5) INIT ('YXZ12'),
BK BIT(4) INIT ('1101'B),
ERG BIT (1);
.......
ERG = ZK > BK;                    /* ERG = '0'B */
ERG = FW < BK;                    /* '1101'B  --> '1101' --> 1101   ERG = -15 < 1101 =
'1'B */
ERG = FW < ZK;                    /* CONVERSION-ERROR   */
```

4. Datenorganisation I

Bis jetzt wurden nur Möglichkeiten besprochen, einzelne Daten eines bestimmten Typs zu vereinbaren. Für eine moderne Programmiersprache muss es jedoch möglich sein, Datenkolonnen gleichen Typs und Datenkolonnen verschiedenen Typs mit einer DCL-Anweisung zu beschreiben.

4.1. Bereiche, Felder

Ein Bereich ist eine geordnete Menge von einfachen Variablen gleicher Skala, Basis, Modus und Genauigkeit, die alle über einen gemeinsamen Bereichsnamen identifiziert werden.

Allgemeine Form:
< bereich > ::= DCL < bezeichner > (< indexgrenzenpaar >) < attribute >;
< indexgrenzenpaar > ::= [< untere grenze > :] < obere grenze >
< grenze > ::= < ausdruck >

Ist die untere Grenze = 1 gilt die Standardannahme.
Maximal 32 Indexgrenzenpaare sind möglich (implementierungsabhängig).
ausdruck muss bei Bezugnahme zur Ausführungszeit eine einfache Größe sein.

Typische Fälle für ein- und zweidimensionale Bereiche sind vor allem Vektoren und Matrizen. Ihre einzelnen Elemente werden über ihre Indizes identifiziert.
Die interne Speicherung eines Bereichs erfolgt zeilenweise; die einzelnen Bereichselemente werden lexikografisch - d.h. von links nach rechts in aufsteigender Reihenfolge ihrer Indizes - abgespeichert. Für die Überlagerung von Bereichen muss dem Anwender diese interne Speicherungsanordnung der Daten bekannt sein.

Beispiele:
```
DCL  VEKTOR (1 : 100) FIXED (5,2);
VEKTOR (1)  = 0;              /* DAS 1.ELEMENT WIRD AUF 0 GESETZT  */
VEKTOR (100) = 99;            /* DAS LETZTE ELEMENT WIRD AUF 99 GESETZT */
VEKTOR (INDEX) = 25;          /* INDEX DARF NUR WERTE ZWISCHEN 1 UND  */
                              /* 100 ANNEHMEN  */
```

Um einen gesamten Bereich auf einen definierten Anfangswert zu setzten, wird er mittels INIT initialisiert. Es ist eine sinnvolle Variante, beim Ausdrucken eines gesamten Bereichs unvorhersehbare Ergebnisse zu vermeiden.

Beispiele:
```
DCL  VEKTOR  (100) FIXED (5,2) INIT ((100) 0);   /* 100 VARIABLE WERDEN AUF 0 GESETZT */
DCL MATRIX (5, 10) FIXED (5,2) INIT ((50) 0);     /*  5 ZEILEN, 10 SPALTEN   AUF 0 GESETZT */
DCL KETTE (5,10) CHAR (100) INIT ((50) (1) ' ');  /* 50 LEERKETTEN WERDEN INITALISIERT  */
DCL E_MATRIX (3,3) FIXED (1) INIT (1, (3)0,1, (3)0, 1);
```

Das letzte Beispiel erzeugt im Speicher eine Einheitsmatrix mit drei Zeilen und drei Spalten. Ihre Hauptdiagonale wird auf 1 gesetzt, der Rest ist 0.
In der Matrixschreibweise ergibt sich für die Werte der Elemente von E_MAT

nach der Initialisierung

1	0	0
0	1	0
0	0	1

Hinweise:

- Wird versucht, einen nicht initialisierten Bereich zu bearbeiten, können Konvertierungsfehler entstehen, die Bedingung CONVERSION wird gesetzt
- Wird eine Bezugnahme außerhalb der vereinbarten Grenzen versucht, wird die Bedingung SUBSCRIPTRANGE gesetzt

4.2. Bereichsoperationen

4.2.1. Verknüpfung mit einfachen Größen

Bei dieser Form wird jedes Bereichselement operativ mit der einfachen Größe verknüpft. Der Compiler generiert entsprechend der Anzahl der Indexgrenzenpaare eine bestimmte Anzahl an Einzelanweisungen.

Beispiele:

```
DCL  MAT  (3,3) FIXED (5,2);
```

Sollen im Laufe der Abarbeitung alle Elemente auf 1 gelöscht werden, so gilt folgende Anweisung:

```
MAT = 1;                    /* 9 ELEMENTE WERDEN 1 GESETZT */
MAT = MAT + MAT (1,1);
```

Das Element der 1. Zeile und 1. Spalte wird auf 2 gesetzt, alle anderen auf 3.
In der Matrixschreibweise ergibt sich für die Werte der Elemente von MAT

vor der Zuweisung			nach der Zuweisung		
1	1	1	2	3	3
1	1	1	3	3	3
1	1	1	3	3	3

4.2.2. Verknüpfung mit Bereichen

Unter gewissen Voraussetzungen können Bereiche ohne Angabe von Indizes benutzt werden, um Bereichsausdrücke zu bilden. Alle Bereiche müssen die gleiche Anzahl von Indexgrenzenpaaren besitzen.

Beispiele:

```
DCL MAT (3,3) FIXED (1) INIT (2, -5, -4, 7, 8, -2, -4, 0, -9);
    MAT = - MAT;
```

In der Matrixschreibweise ergibt sich für die Werte der Elemente von MAT

vor der Zuweisung			nach der Zuweisung		
2	-5	-4	-2	5	4
7	8	-2	-7	-8	2
-4	0	-9	4	0	9

```
DCL  (BER_1 (2,3), BER_2 (2,3)) FIXED INIT ((2) (6) 0);       /* ODER ((12)0) ODER (0)  */
BER_1 = BER_1 + BER_2;            /* ELEMENTE MIT GLEICHEN INIZES WERDEN ADDIERT */
BER_1 = BER_1 *  BER_2;           /* ELEMENTE MIT GLEICHEN INIZES WERDEN MULTIPLIZIERT */
```

Es wird darauf verwiesen, dass die letzte Anweisung keine Matrizenmultiplikation im mathematischen Sinne darstellt. Eine Matrizenmultiplikation oder eine Transponierung kann mit einer DO-Anweisung realisiert werden.
Bereiche können nicht miteinander verglichen werden. Es gibt keine Vergleichsoperatoren zwischen Bereichen. Ein Vergleich zweier Bereiche kann z.B. mit einer DO-Anweisung realisiert werden.
Bereiche können miteinander verkettet werden.

Beispiele:

```
DCL KET_1 CHAR (3) INIT ( 'BER', 'LON', 'MOS' ),
      KET_2 CHAR (3) INIT ( 'LIN', 'DON', 'KAU' ),
      KET_3 CHAR (6);
KET_3 = KET_1 || KET_2;               /* KET_3 (1) =  'BERLIN';……*/
```

4.3. Unterbereiche

Ein Unterbereich ist dadurch gekennzeichnet, dass zumindest ein Index durch einen Stern gekennzeichnet ist. Der Stern durchläuft das gesamte Indexgrenzenpaar; also von der unteren Grenze bis zur oberen. Die Anzahl der Sterne und Indexgrenzen entspricht der Gesamtanzahl der Indexgrenzenpaare. Sind alle Indexgrenzen durch Sterne dargestellt, so wird eine Bezugnahme auf den gesamten Bereich vollzogen.

Beispiele:

Die erste Zeile einer Matrix ist auf Null zu löschen.

```
DCL MAT (3,3) FIXED(1);
      MAT (1,*) = 0 ;
```

Die zweite Spalte einer Matrix ist auf Null zu löschen

```
      MAT (*,2) = 0;
```

Die Zeilensumme einer Matrix ist zu bilden

```
      DCL MAT (3,3) FIXED(5,2) INIT (0),
SPALTSUM (3) FIXED (7,2);
…..........
SPALTSUM (*) = MAT(*,1) + MAT (*,2 ) + MAT (*,3);
```

In der Matrixschreibweise ergibt sich für die Werte der Elemente von MAT und SPALTSUM

Matrix (angenommene Daten)			Spaltenvektor
2.2	45.5	-24.6	23.1
-7.0	8.45	27.23	28.68
4.34	-34.55	9.44	-20.77

Hinweis: Beim Einsatz von entsprechenden iterativen DO-Gruppen reduziert sich die Anweisung auf folgende Laufanweisung:

```
DO ISPALT = 1 TO 3;
SPALTSUM  = MAT (*, ISPALT);
END;
```

5. Überblick über die PL/1 Programmstruktur

Eine Folge von Anweisungen und Klauseln, die in ihrer Gesamtheit zur Lösung einer Aufgabe ausreichend sind, ist ein Programm. Das Programm besteht aus einem oder mehreren Programmteilen, den Blöcken. Jeder Block besitzt wiederum eine oder mehrere Gruppen.
In diesem Punkt wird nur ein Überblick auf die Blockstruktur vermittelt; Details zu den Blöcken und Gruppen werden in den folgenden Kapiteln beschrieben.

5.1. Blöcke und ihre Klauseln

Ein Block ist eine Folge von Anweisungen und Klauseln. Am Anfang und am Ende eines Blockes steht jeweils eine Klausel zur Kennzeichnung.

{PROCEDURE | PROC} - Klausel
- bestimmt den Anfang eines Blockes
- bestimmt die Eigenschaft eines Blockes
- Hauptprocedur OPTIONS (MAIN)
- wahlweise Aufnahme einer Parameterliste PROC (parm)

ENTRY-Klausel
- bestimmt einen sekundären Eingangspunkt eines Blockes

BEGIN-Klausel
- bestimmt den Anfang eines BEGIN-Blockes

END-Klausel
- bestimmt das Ende eines PROC - oder BEGIN - Blockes

Weiteres zu Procedur-Blöcken wird im Punkt 8. behandelt.

5.2. Gruppen und ihre Klauseln

Eine Gruppe ist eine Folge von Anweisungen und wird von einer DO- und END-Klausel begrenzt.

DO - Klausel
- bestimmt den Anfang einer DO - Gruppe

SELECT - Klausel
- bestimmt den Anfang einer SELECT - Gruppe

END - Klausel
- bestimmt das Ende einer DO - oder SELECT - Gruppe

In Abhängigkeit von der Art der DO - Klausel kann eine DO - Gruppe für einmalige oder für mehrfache, für iterative Ausführungen vorgesehen sein.
Deshalb gilt folgende Unterteilung:

Einfache DO - Gruppe

- Zusammenfassung von Anweisungen, Gruppen und Blöcken
- bei Programmausführung Behandlung wie einzelne Anweisung
- die LEAVE - Option dient zum vorzeitigen Verlassen der Gruppe, weitere Abarbeitung mit der Anweisung, die auf END- Klausel folgt

Iterative DO - Gruppen

- wird von einer iterativen DO - und END - Klausel begrenzt
- werden gemäß den in der DO - Klausel formulierten Bedingungen wiederholt ausgeführt
- weitere Möglichkeiten zum vorzeitigen Verlassen der DO - Gruppe sind die Optionen WHILE, UNTIL, REPEAT und LEAVE

SELECT - Gruppen

- wird von einer SELECT - und END - Klausel begrenzt
- Verlassen durch WHEN - oder OTHERWISE - Klausel
- eine WHEN - oder OTHERWISE - Einheit kann ein
  ein BEGIN - Block
  eine DO - Gruppe
  oder eine ausführbare Anweisung
  sein.

5.3. Unterschied zwischen BEGIN - und PROC – Blöcken

PROZEDURE-Block	BEGIN-Block
Aktivierung, wenn Programmablaufsteuerung durch Prozeduraufruf an PROC-Block übergeben wird	Aktivierung, wenn Programmablaufsteuerung die BEGIN-Klausel beim sequentiellen Programmablauf erreicht
Datenaustausch über Argument-Parameterbeziehung möglich	Datenaustausch wegen fehlender Argument-Parameterbeziehung nicht möglich
Rückgabe von Daten an die rufende Prozedur möglich	Rückgabe von Daten bei Blockbeendigung nicht möglich
PROC-Block kann von verschiedenen Eingangspunkten ausgeführt werden	BEGIN-Block kann nur an der BEGIN-Klausel ausgeführt werden
PROC-Block kann als erster Block in einer Task aufgerufen werden; vom Betriebssystem Aufrufbar	BEGIN-Block kann nur in der bestehenden Task ausgeführt werden; vom Betriebssystem nicht erreichbar

6. Steuerung des Programmablaufs

Ein typisches Merkmal der inhaltlich-logischen Bearbeitung eines Problems besteht darin, dass mehrfache Wiederholungen bestimmter Abläufe, Verzweigungen und andere von der sequentiellen Ablauffolge abweichende Programmausführungen entsprechend der Aufgabenstellung auftreten. Deshalb gibt es Anweisungen, die den gesamten Programmablauf steuern, die sog. Steuerungsbefehle

6.1. Sprunganweisungen

Mittels dieser Anweisungen kann die sequentielle Programmabarbeitung unterbrochen werden und mit einer entsprechend markierten Anweisung fortgesetzt werden.

6.1.1. Marken

Marken stehen in Form von Markenpräfixen am Anfang von Anweisungen und am Anfang von BEGIN-, DO- und END-Klauseln. Sie muss angegeben werden, wenn die Anweisung bzw. Klausel das Ziel einer Sprunganweisung sein soll.

Allgemeine Form der Markenkonstanten:
< markenpräfix > ::= < bezeichner > :

Markenkonstanten werden nicht vereinbart.

Beispiele:

```
…….
   MARKE1 : WERT = WERT1 / WERT2;
   EINGABE: READ FILE  (dateibezeichnung) INTO (satzvariable);
   DRUCK: PUT EDIT (datenliste) (formarliste);
```

Neben den Markenkonstanten gibt es Markenvariable, die verschiedene Sprungadressen, also Markenpräfixe, aufnehmen kann. Sie muss mit LABEL vereinbart werden.

Allgemeine Form der Markenkonstanten:
< markenvariable > ::= DCL < bezeichner > LABEL [(< markenpräfix 1 >,…,> markenpräfix n >)];

Beispiel:

```
   DCL MARKVAR LABEL;
   ……
   wenn   W1 > W2  dann MARKVAR = M1;
   sonst MARKVAR = M2;
   GOTO MARKVAR;     /* MARKVAR ENTHAELT ADRESSE M1 ODER M2 */
   M1: W1 = W1 – W2;
   ……..
   M2: W1 = W2 – W1;
            ……
```

Durch Angabe eines Dimensionsattributes ist auch die Vereinbarung eines Feldes von Markenvariablen möglich. In der Liste sind nur die Markenpräfixe angegeben, die im Laufe der Programmabarbeitung tatsächlich dieser Variablen zugeordnet werden können.

Beispiele:

```
(1)        DCL MV1 LABEL,
(2)              MV2 LABEL INIT(MAR10),
(3)              MV3 LABEL (MAR1, MAR2, MAR3),
(4)              MV4 (3) LABEL (MAR4, MAR5, MAR6);
(5)     GOTO MV_4(IND);
```

(1) MV1 und MV2 dürfen im Verlauf der Programmabarbeitung alle Markenpräfixe annehmen, die im Programm auftreten können.
(2) Für MV2 wird als Anfangswert zu Beginn der Programmabarbeitung der Marken- präfix MAR10 festgelegt.
(3) Der Markenvariablen MV3 dürfen nur die Markenpräfixe MAR1, MAR2 und MAR3 zugeordnet werden.
(4) Der Bezeichner für die Markenvariable kann indiziert sein.
(5) In Abhängigkeit von IND wird zu MAR4, MAR5 oder MAR6 verzweigt.

6.1.2. Die unbedingte Sprunganweisung

Um in die normale Programmablaufsteuerung eingreifen zu können, wird in der einfachsten Form die unbedingte Sprunganweisung verwendet.

Allgemeine Form:
< unbedingtes sprunganweisung > ::= { GOTO } | {GO TO } < markenpräfix [(<ausdruck >)];

Markenpräfix verweist auf die markierte Anweisung, die als nächste abgearbeitet werden soll.

Beispiele:

```
    GOTO MARKVAR;
…
    GO TO MARKE(IND);
…
    GOTO BLOCK;
…
BLOCK: BEGIN;
….
END BLOCK;
```

Hinweis: Man sollte vorsichtig im Umgang mit GOTO sein. Die sog. GOTO-freie Programmierung erlaubt wesentlich elegantere Lösungen.

6.1.3. Die bedingte Sprunganweisung

Im Gegensatz zur unbedingten Sprunganweisung wird bei der bedingten Sprunganweisung, bei der bedingten Verzweigung, die sequentielle Programmablaufsteuerung auf Grund einer Bedingung verlassen.

Allgemeine Form:

< bedingte sprunganweisung > ::= IF < einfacher ausdruck > THEN < einheit_1 > ;
[ELSE < einheit_2 > ;]

Der einfache Ausdruck wird berechnet, in eine Bitkette der Länge 1 umgewandelt; ist sie ‘1‘B , also „wahr“, wird der THEN mit einheit_1, andernfalls wird der ELSE-Zweig mit einheit_2 abgearbeitet. Ist kein ELSE-Zweig vorhanden, wird mit der Anweisung fortgefahren, die der IF-Anweisung folgt.
einheit kann eine DO-Gruppe, ein BEGIN-Block oder eine beliebige Anweisung sein.

Beispiele:

```
(1)     IF A > B THEN GOTO M1;
          ELSE GOTO M2;
(2)     IF W1 < W2 & W3 > W4 THEN ERG = ‘1’B;        /*LOGISCHE VERKNÜPFUNG */
ELSE ERG = ‘0’B;
```

IF-Anweisungen können verschachtelt werden. Die Zuordnung der THEN- und ELSE- Klauseln erfolgt immer aus Sicht des Compilers von innen nach außen, unter Berücksichtigung, dass das ELSE dem THEN zugeordnet wird, welches ihm physisch am nächsten steht.

(3) Wenn W1 bis W6 gleich sind, soll das Ergebnisfeld auf "wahr" gesetzt werden, in allen anderen Fällen auf "falsch".

```
IF (W1 = W2 ) & (W3 = W4 ) & (W5 = W6) THEN  ERG = ‘1’B;
oder verschachtelt:
IF W1 = W2 THEN
          IF W3 = W4 THEN
       IF W5 = W6 THEN ERG = ‘1’B;
```

(4) Es sollen jeweils drei ungleiche einfache Werte eingelesen werden, die größte ermittelt und ausgedruckt werden

```
        SORT: PROC OPTION (MAIN);
  DCL (W1, W2, W3, MAX) FIXED;
…….
 LIES: GET LIST (W1, W2, W3);              /* STANDARDEINGABE */
  IF W1 > W2 THEN
                    IF W1 > W3 THEN MAX = W1;
                                ELSE MAX = W3;
                     ELSE
                   IF W2 > W3 THEN MAX = W2;
                   ELSE MAX = W3;
    PUT LIST (MAX);
    GOTO LIES;
              END SORT;
```

6.2. DO - Gruppen

Für die Programmierung von Befehlsschleifen werden DO-Gruppen mit verschiedenen Einsatzgebieten angeboten.

6.2.1. Einfache DO – Gruppen

Sie dient vor allem zur Gruppierung von Anweisungen, weiteren DO-Gruppen und Blöcken, die aus programmtechnischen Gründen als eine Einheit betrachtet werden müssen, wie z.B. im THEN- und ELSE-Zweig einer IF-Anweisung.

Allgemeine Form:

```
< einfache DO-gruppe >::= [ < markenpräfix > ] DO < folge von anweisungen >
                                                    END [ < markenpräfix > ];
```

Beispiel:

```
IF (W1 + W2) < (W3 – W4) THEN  DO; IZAEL = IZAEL + 1;
                PUT LIST (IZAEL);       /* STANDARDDRUCK */
                GOTO EINGABE;
                                  END;
                           ELSE DO; IZAEL = IZAEL - 1;
                           PUT LIST (IZAEL);  /* STANDARDDRUCK */
                                    GOTO FEHL_1;
                                 END;
```

6.2.2. Iterative DO – Gruppen

Soll die Folge von Anweisungen in einer DO-Gruppe mehrmals durchlaufen werden, bietet sich die iterative DO-Gruppe an.

Allgemeine Form:

```
< iterative DO-gruppe >::=  [ < markenpräfix > ] DO laufvariable = spezifikationen;
                            < folge von anweisungen >;
                            END [ < markenpräfix > ];
```

Der einfachen Laufvariablen wird je nach Spezifikation ein arithmetischer Wert zugeordnet. Spezifikation spezifiziert die verschiedenen Werte, die die Laufvariable annehmen soll. Ein vorzeitiges Verlassen der DO-Gruppe ist jederzeit durch die LEAVE-Option möglich.
Es gibt fünf mögliche Grundformen der Spezifikation:

1. Form der Spezifikation

Allgemeine Form:

```
< spezifikation_1 > ::= < folge von ausdrücken >
```

Die Laufvariable nimmt der Reihe nach die durch Komma getrennten Werte an, die durch die Ausdrücke bestimmt sind. Ist die Laufliste erschöpft, wird die der DO-Gruppe folgende Anweisung abgearbeitet.

2. Form der Spezifikation

Allgemeine Form:

< spezifikation_2 > ::= < ausdruck_1 > [BY < ausdruck_2 >] TO < ausdruck_3 >

Die Laufvariable nimmt der Reihe nach die Werte an, die in spezifikation spezifiziert sind.
ausdruck_1 ist der Anfangswert der Laufvariablen,
ausdruck_2 ist die Schrittweite, d.h. Erhöhung des Wertes der Laufvariablen um den Wert ausdruck_2, standardmäßig 1,
ausdruck_3 ist der Endwert der Laufvariablen.
Ist die Laufliste erschöpft, wird die der DO-Gruppe folgende Anweisung abgearbeitet.

3. Form der Spezifikation

Allgemeine Form:

< spezifikation_3 > ::= [< ausdruck_1 >] [BY < ausdruck_2 >] WHILE (< ausdruck_3 >)

Die Laufanweisung wird solange ausgeführt, solange die Bedingung ausdruck_3 wahr ist. Die Bedingung wird **vor** jedem Schleifendurchgang neu berechnet.

4. Form der Spezifikation

Allgemeine Form:

< spezifikation_3 > ::= [< ausdruck_1 >][BY < ausdruck_2 >] UNTIL (< ausdruck_3 >)

Die Laufanweisung wird solange ausgeführt, solange die Bedingung ausdruck_3 nicht wahr ist. Die Bedingung wird **nach** jedem Schleifendurchgang neu berechnet. Ist sie wahr, wird die DO-Anweisung sofort verlassen.

5. Form der Spezifikation

Allgemeine Form:

< spezifikation_3 > ::= < ausdruck_1 > REPEAT < ausdruck_2 > {UNTIL | WHILE} (< ausdruck_3 >)

REPEAT wird anstelle von BY und TO benutzt. Die Bedingung wird vor dem 2. und jedem weiteren Schleifendurchgang neu berechnet; dadurch ist eine nichtlineare Änderung der Laufvariablen möglich.

6. Kombinationen der Spezifikationen

Selbstverständlich sind Kombinationen aller Spezifikationen möglich. Ebenso ist eine Verschachtelung für Bereiche mit mehreren Indexgrenzenpaaren möglich. Grundsätzlich werden verschachtelte DO-Gruppen von Innen nach Außen abgearbeitet.

6.2.3. Beispiele zu iterativen DO – Gruppen

Beispiel zu Spezifikation 1:

```
DCL FELD (100) FIXED;
   LOESCH: DO I = 1,10, 50, 100;
                FELD (I) = 0;      /* BESTIMMTE INDIZIERTE WERTE VON FELD AUF 0
LOESCHEN */
                END LOESCH;
```

Beispiele zu Spezifikation 2:

(1) Die Elemente eines Feldes sollen mit den Werten ihrer Indizes gefüllt werden.

```
DCL FELD (1000) FIXED;
      LAUF:DO IND = 1 BY 1 TO 1000;
             FELD(IND) = IND;
             END LAUF;
```

(2) Negative Zahlen eines Bereichs sollen ermittelt werden, untereinander ausgedruckt und durch ihre positiven ersetzt werden:

```
DCL BEREICH (-500 : 500) FIXED (7,2);
   SUCH:  DO  IND = -500  TO 500;
            IF BEREICH(IND) >= 0 THEN;            /* LEERE THEN-KLAUSEL   */
 /* < 0 */                ELSE DO; PUT LIST SKIP ('INDEX = ', IND, BEREICH(IND));
                BEREICH(IND) = BEREICH(IND) * (-1);
                                END;
END SUCH;
```

(3) Von 5000 dezimalen Werten, die in einen Bereich eingelesen werden, ist der Wert der Größten zu suchen und mit ihrem Index auszudrucken.

```
DCL BEREICH (5000) FIXED (7,2),
      MAX  FIXED (7,2);
      GET LIST (BEREICH);     /* 5000 WERTE VON STANDARD-EINGABE LESEN */
      MAX = BEREICH (1);
  SUCH: DO IND = 1 TO 5000;
          IF BEREICH (IND) > MAX THEN DO;
          MAX = BEREICH (IND);
          IMAX = IND;
       END;
 PUT LIST SKIP ('INDEX = ', IMAX, BEREICH(IND));
END SUCH;
```

Beispiel zu Spezifikation 3:

(1)

```
DCL ABBRUCH BIT (1) INIT ('1'B);
   DO WHILE(ABBRUCH);
   .....              /* ABARBEITUNG, SOLANGE ABBRUCH = '1'B IST */
END;
```

(2)

```
DO WHILE ('1'B);
   ...
```

```
      IF W1 < W2 THEN LEAVE;    /* VERLASSEN NUR UEBER LEAVE MOEGLICH */
        .....
     END;
```

(3) Die erste negative Zahl eines Bereichs soll ermittelt und ausgedruckt werden:

```
    DCL BEREICH (-500 : 500) FIXED (7,2);
    DCL STOPBIT BIT(1) INIT ('1'B);
       SUCH:  DO  IND = -500  WHILE (STOPBIT);
                  IF BEREICH(IND) < 0 THEN DO; PUT LIST SKIP ('INDEX = ', IND, BEREICH(IND));
                    STOPBIT = '0'B;
                                    END;
   END SUCH;
```

Beispiel zu Spezifikation 4:

```
   DO ILAUF = -10 BY K * 2  UNTIL ( ILAUF > 100);
          .....    /* SCHRITTWEITE IMMER UM DAS DOPPELTE VON K */
       END;
```

Beispiel zu Spezifikation 5:

```
   DO IWERT = 1 REPEAT IWERT * 2 UNTIL ( IWERT = 8192);
            BER(IWERT) = IWERT;
            .....   /* IWERT NIMMT DIE WERTE 1,2,4,8,16,32,...,8192 AN */
    END;
```

Beispiele zu Kombinationen:

(1) Folgende Summe soll gelöst werden:

$$SUM = \sum_{k=-r}^{r} \frac{1}{k^2 - 1} \qquad \text{für } k \neq \pm 1$$

(k hoch 2 -1)

```
      SUM = 0;
      DO K  = -R  TO -2 ,0, 2 TO R;
           SUM = SUM + 1 / (K**2 - 1);
      END;
```

(2) DO - Schleifen können beliebig verschachtelt werden. Es soll die Summe der Werte eines zweidimensionalen Bereichs - einer Matrix - berechnet werden.

```
   DCL MAT (3,4) FIXED (7,2),     /* MATRIX 3 ZEILEN, 4 SPALTEN */
          SUM FIXED (9,2);
          .....
           SUM = 0;
     M_1: DO I = 1 TO 3;     /* ABARBEITUNG ZEILENWEISE */
                DO K = 1 TO 4;
                    SUM = SUM + MAT (I,K);
                END;
             END M_1;
```

Die Abarbeitung der Indizes erfolgt lexikografisch; d.h.

```
MAT(1,1)   MAT(1,2)   MAT(1,3)   MAT(1,4)
MAT(2,1)   MAT(2,2)   MAT(2,3)   MAT(2,4)
MAT(3,1)   MAT(3,2)   MAT(3,3)   MAT(3,4)
```

(3) In einer Matrix sind alle Spaltensummen zu bilden und zu speichern.

```
DCL MAT (3,4) FIXED (7,2),      /* MATRIX 3 ZEILEN, 4 SPALTEN */
      SSUM(4)  FIXED (9,2),
      TEMPSUM  FIXED(9,2);
       .....
            TEMPSUM = 0;
       M_1:DO I = 1 TO 3;/* ABARBEITUNG ZEILENWEISE */
                 DO K = 1 TO 4;
                      TEMPSUM = TEMPSUM + MAT (I,K);
                  END;
             SSUM (K) = TEMPSUM;
      END;
```

(4) Zwei Matrizen sollen nach mathematischen Regeln multiplikativ verknüpft werden, d.h. die Zeilen der 1. Matrix werden mit den Spalten der 2. Matrix multipliziert, d.h. wiederum, die Zeilenzahl der 1. Matrix muss mit der Spaltenzahl der 2. Matrix übereinstimmen; sie müssen verkettet sein.

$$C_{i,k} = \sum_{j=1}^{n} a_{i,j} * b_{j,k}$$

```
DCL (MATA (3,4), MATB (4,3)) FIXED (7,2),
      MATSUM (3,4) FIXED (9,2),
     TEMPSUM  FIXED(9,2);
      .....
      M_1:DO I = 1 TO 3;     /* ABARBEITUNG ZEILENWEISE */
               DO K = 1 TO 4;
                        TEMPSUM = 0;
                  DO J = 1 TO 4;
                           TEMPSUM = TEMPSUM + MATA (I,J) * MATB(J,K) ;
                   END;
                        MATSUM (I,K) = TEMPSUM;
                END;
           END M_1;
```

6.3 . SELECT - Gruppen

Für die Programmierung von Mehrfachauswertungen von Ausdrücken bzw. von Entscheidungstabellen, stehen SELECT-Gruppen zur Verfügung.

Allgemeine Form:

```
< SELECT-gruppe> ::=  SELECT [( < ausdruck_0 >) ];
                          WHEN ( < ausdruck_1 >) < einheit_1 >;
                          WHEN ( < ausdruck_2 >) < einheit_2 >;
                          ...
                          WHEN ( < ausdruck_49 >) < einheit_49 >;
                        [ OTHERWISE < einheit > ;]
                      END;
```

Der Wert von ausdruck_0 wird berechnet und gespeichert. ausdruck_1 bis ausdruck_n werden in der Reihenfolge ihres Auftretend berechnet und mit ausdruck_0

verglichen. ausdruck muss in eine Bitkette überführbar sein. Ist die Bitkette '1'B, wird die entsprechende einheit_n ausgeführt und die SELECT-Gruppe verlassen, ansonsten wird der nächste ausdruck in der WHEN-Klausel ausgewertet. Kann keine WHEN-Klausel ausgeführt werden, wird die wahlweise einheit in der OTHERWISE-Klausel ausgeführt.

Beispiel:

```
DCL SEL CHAR (1);
      .........
      SELECT (SEL);
          WHEN ('A') DO;......; END;
          WHEN ('B') BEGIN;......; END;
          WHEN ('C') GOTO MARKE;
          WHEN ('D') CALL UP1;
          WHEN ('E') I = I + 1;
       OTHERWISE PUT LIST (SEL);
       END;
```

Diese SELECT - Gruppe bewirkt die Ausführung
- der DO-Gruppe, wenn SEL = 'A',
- des BEGIN-Blocks, wenn SEL = 'B',
- der GOTO-Anweisung, wenn SEL = 'C',
- der CALL-Anweisung, wenn SEL = 'D',
- der Ergibtanweisung, wenn SEL = 'E',
- der PUT-Anweisung, wenn die obigen Bedingungen nicht zutreffen.

6.4. BEGIN - Blöcke

6.4.1. Allgemeine Erklärung

Ein BEGIN-Block ist ein interner Block, der aus einer von einer BEGIN- und einer END-Klausel begrenzten Folge von Anweisungen besteht.
BEGIN-Blöcke werden vorrangig bei der Maßnahmebehandlung von Ausnahmebedingungen - den ON-Einheiten - verwendet.

Allgemeine Form:

```
< BEGIN-block >::= [ < markenpräfix > ] BEGIN;
                   [< folge von vereinbarungen > ]
                    < folge von anweisungen >
                      END [ < markenpräfix > ];
```

Beispiel:

```
BLOCK_1: BEGIN;
         DCL  folge von vereinbarungen;
         folge von anweisungen;
         gruppen;
         blöcke;
              END BLOCK_1;
```

6.4.2. Prolog und Epilog

Beim Aktivieren und Beenden eines BEGIN- oder eines PROC-Blockes müssen bestimmte organisatorische Maßnahmen getroffen werden. Sie werden beim Aktivieren als Prolog und beim Verlassen als Epilog bezeichnet.
Die wichtigsten Funktionen des Prologs:

- Festlegung von Dimensionsgrenzen, Kettenlängen und deren Initialisierung
- Berechnung von Anfangswerten und Wiederholungsfaktoren bei INIT-Angaben
- Zuordnung von Speicherplatz für alle explizit und implizit vereinbarte Variablen
- Zuordnung von Speicherplatz für Zwischenergebnisse, für Pseudoveriable

Die wichtigsten Funktionen des Epilogs:

- Wiederherstellung des Zustandes, der vor der Blockaktivierung existierte
- Freigabe des Speicherplatzes, der durch den Prolog zugeordnet wurde

6.4.3. Aktivierung und Beendigung von BEGIN-Blöcken

Ein BEGIN-Block wird aktiviert, wenn seine BEGIN-Klausel im Programmablauf erreicht wird. Ein BEGIN-Block in einer aktiven ON-Einheit wird nur durch das Auftreten des Ausnahmezustandes aktiviert.
Ein BEGIN-Block wird normal beendet, wenn die Programmablaufsteuerung auf die END-Klausel trifft. Auch Möglichkeiten, BEGIN-Blöcke bei der Benutzung von internen Blöcken durch die Aufführungen von GOTO-Anweisungen, STOP-Anweisungen, EXIT-Anweisungen und RETURN-Anweisungen anormal zu beenden, bestehen.

7. Datenorganisation II

Bis jetzt wurden zwei Möglichkeiten der Vereinbarung von Daten vorgestellt:
- Einzeldaten durch Basis, Skala, Modus und Genauigkeit definiert
- Felder von Einzeldaten mit gleicher Basis, Skala, Modus und Genauigkeit

Die Möglichkeit, beide Formen in einer Organisationsform darzustellen, bieten die Strukturen.

7.1. Strukturen

Eine Struktur ist eine nach Stufen geordnete Menge von Variablen, Feldern und Unterstrukturen.
Folgende Stufen sind definiert:
- Hauptstruktur
- Unterstruktur
- Strukturelement

Hauptstruktur:
- Name der gesamten Datengruppierung
- Stufennummer 1

Unterstruktur:
- der Hauptstruktur untergeordnet
- zeigt den Beginn einer logischen Gruppierung an
- Stufennummer > 1, maximale Größe der Stufennummer = 255
- maximale Tiefe der Datengruppierung = 8

Strukturelement:
- kleinste Gruppierung einer Struktur
- Stufennummer des Strukturelements muss größer als die Stufennummer der übergeordneten Unterstruktur sein

Aufgabe: Es soll ein Datensatz vereinbart werden, der in der Lage ist, Daten zum Nachweis des Nettolohnes eines Mitarbeiters aufzunehmen:
Nettonachweis:

- Mitarbeiter	- Pers.Nr.	
	- Name	- Nachname
		- Vorname
- Zeit	- Monat	
	- Jahr	
- Arbeitsstunden		
- Verdienst	- Brutto	
	- Abzugspflichtig	- Steuerpflichtig
		- Versicherungspflichtig
- Abzüge	- Sozialversicherung	- Krankenversicherung
		- Rentenversicherung

		- Arbeitslosengeld
	- Steuern	- Lohnsteuer
		- Kirchensteuer
- Netto		

Zur Bearbeitung derartiger hierarchisch gegliederter Datenanordnungen bieten sich Lösungen mittels Strukturen an:

```
DCL 1 NETTONACHWEIS,
      2 MITARBEITER,
         3 PERS_NUMMER PIC '(8)9',
         3 NAME,
            4 NACHNAME CHAR (20),
            4 VORNAME   CHAR (15),
      2 ZEIT,
         3 MONAT CHAR (8),
         3 JAHR PIC '(4)',
      2 ARBEITSSTUNDEN FIXED (5,2),
      2 VERDIENST,
         3 BRUTTO              FIXED (7,2),
         3 ABZUGSPLICHTIG,
            4 STEUER              FIXED (6,2),
            4 VERSICHERUNGS FIXED (6,2),
      2 ABZUEGE,
         3 SOZIALVERSICHERUNG,
            4 KRANKENVERS         FIXED (5,2),
            4 RENTENVERS          FIXED (5,2),
            4 ARBEITSLOSENVERS FIXED (5,2),
         3 STEUERN,
            4 LOHNSTEUER          FIXED (5,2),
            4 KIRCHENSTEUER       FIXED (5,2),
      2 NETTO FIXED (6,2);
```

Der Anwender hat die Möglichkeit, auf die ganze Struktur Bezug zu nehmen. Eine sinnvolle Einsatzvariante stellt die Ein- und Ausgabe dar, wo große Datenmengen transportiert werden.

Beispiel:

```
READ FILE (MITARB_DATEI) INTO ( NETTONACHWEIS);
```

Ein Datensatz wird von der Eingabedatei "MITARB_DATEI" in den strukturierten Eingabebereich gelesen. Soll nun ein bestimmtes Strukturelement angesprochen werden, so muss eine Qualifizierung der Bezeichnung in der Weise vorgenommen werden, dass vor den eigentlichen Bezeichner noch die Unterstruktur- bzw. Hauptstrukturnamen geschrieben werden. Der Name ist vollqualifiziert, wenn alle seine übergeordneten Strukturnamen, getrennt durch einen Punkt, angegeben werden. Zur Eindeutigkeit bzgl. des umgebenden Blockes muss die Bezugname zumindest qualifiziert, d.h. eindeutig vorgenommen werden.

Beispiel: Dem Mitarbeiter Krause wird sein Bruttogehalt in der obigen Struktur zugewiesen.

```
 (1)  IF NETTONACHWEIS. MITARBEITER. NAME. NACHNAME = 'KRAUSE'
        THEN DO;
                  NETTONACHWEIS. VERDIENST. BRUTTO = 2345.24;
                  ..........
                 END;    /* BEZUGNAME VOLLQUALIFIZIERT */

 (2)  IF NETTONACHWEIS. NACHNAME = 'KRAUSE'
        THEN DO;
                  NETTONACHWEIS. BRUTTO = 2345.24;
                  ..........
                 END;    /* BEZUGNAME QUALIFIZIERT */
```

Weitere Möglichkeiten einer qualifizierten Bezugnahme sind denkbar.

7.2. Bereiche von Strukturen

Soll nun der Datensatz zum Nachweis des Nettolohnes aller 1000 Mitarbeiter eines Unternehmens analog 7.1. aufgebaut werden, so ist es sicher sehr müßig, 1000 Einzelstrukturen zu deklarieren. Analog Datenorganisation I im Punkt 4. lassen sich ebenso bei den Strukturen Bereiche definieren, wobei einzelne Strukturelemente über indizierte Namen eindeutig gekennzeichnet sind.

Beispiel: Dem Mitarbeiter Krause wird sein Bruttogehalt zugewiesen.

```
DCL 1 NETTONACHWEIS (1000),
         2 MITARBEITER,
            3 PERS_NUMMER PIC '(8)9',
            3 NAME,
               4 NACHNAME CHAR (20),
               4 VORNAME   CHAR (15),
         2 ZEIT,
            3 MONAT CHAR (8),
            3 JAHR PIC '(4)',
         2 ARBEITSSTUNDEN FIXED (5,2),
         2 VERDIENST,
            3 BRUTTO              FIXED (7,2),
              .....
SUCH: DO  I = 1 TO 1000;
         IF NETTONACHWEIS (I). NACHNAME = 'KRAUSE'
           THEN DO;
                  NETTONACHWEIS (I). BRUTTO = 2345.24;
                 ..........
                   END;
              .....
         END SUCH;
```

Die qualifizierte Bezugnahme auf einzelne Strukturelemente muss indiziert erfolgen.

Die Indizierung kann, die Eindeutigkeit vorausgesetzt, in vielfältiger Weise erfolgen.

Beispiel:

```
DCL 1 STR (10,15),
        2 USTR (5),
          3 FELD (10) FIXED (7,2),
          3 WERT FLOAT,
          .......
```

Es sollen in einer DO-Gruppe bestimmte Elemente des Bereichs FELD auf 0 gesetzt werden.

```
(1) STR(I, J). USTR_1 (K). FELD(L) = 0;
(2) STR(I, J, K, L). USTR. FELD = 0;
(3) STR. USTR. FELD(I, J, K, L) = 0;
```

Weitere indizierte Bezugnahmen sind möglich.

7.3. Strukturausdrücke, die Option BY NAME

Mit Strukturen können ähnlich wie bei einfachen Variablen Verknüpfungen vorgenommen werden. Allerdings können Strukturen nur in drei verschiedenen Arten miteinander verknüpft werden:

1. Struktur mit einer einfachen Variablen verknüpft
2. Strukturen mit gleichem Aufbau werden verknüpft
3. Strukturen mit unterschiedlichen Aufbau werden verknüpft, die Option BY NAME

Zu 1. Struktur mit einer einfachen Variablen verknüpft

Strukturergibtanweisungen werden vom Compiler in eine Folge einfacher Ergibtanweisungen aufgelöst.

Beispiel:

```
DCL 1 STR_1,
          2 BLUE FIXED,
          2 RED FIXED (5,2),
          2 GREEN BIN FIXED;
 STR_1 = 0;
 ......
 STR_1 = STR_1 + 5;
```

Aus dieser Strukturanweisung werden folgende Einzellanweisungen generiert

```
 STR_1.BLUE =  STR_1.BLUE  + 5;
 STR_1.RED =  STR_1.RED + 5;
 STR_1.GREEN =  STR_1.GREEN + 5 ;
```

Zu 2. Strukturen mit gleichem Aufbau werden verknüpft:

Strukturergibtanweisungen mit zwei Strukturen gleichen Aufbaus und unterschiedlichen Namen werden vom Compiler in eine Folge einfacher Ergibtanweisungen aufgelöst.

Beispiel:

```
DCL 1 STR_1,
        2 BLUE FIXED,
        2 RED FIXED (5,2),
        2 GREEN BIN FIXED,
DCL 1 STR_2,
        2 BLACK FIXED,
        2 WHITE FIXED (5,2),
        2 YELLOW BIN FIXED;
        .....
        /* DATENEINGABE */
STR_1 = STR_2 + 25;
```

Aus dieser Strukturanweisung werden folgende Einzellanweisungen generiert:

```
STR_1.BLUE =  STR_2.BLACK  + 25;
STR_1.RED =  STR_2.WHITE + 25;
STR_1.GREEN =  STR_2.YELLOW + 25 ;
```

Zu 3. Strukturen mit unterschiedlichen Aufbau werden verknüpft, die Option BY NAME

Strukturergibtanweisungen mit zwei Strukturen unterschiedlichen Aufbaus und gleichen Namen werden bei Angabe der BY NAME Option dem Namen nach verknüpft. Sie werden vom Compiler in eine Folge einfacher Ergibtanweisungen aufgelöst. Diese Form ist sehr praxisrelevant.

Beispiel:

```
DCL 1 STR_1,
        2 BLUE CHAR (5),
        2 RED FIXED (5,2),
        2 GREEN BIN FIXED,
DCL 1 STR_2,
        2 RED FIXED,
        2 WHITE BIT(8),
        2 GREEN BIN FIXED;

STR_1 = STR_1 + STR_2, BY NAME;
```

Aus dieser Strukturanweisung werden folgende Einzellanweisungen generiert:

```
STR_1.RED =  STR_1.RED  + STR_2;
STR_1.GREEN =  STR_1.GREEN + STR_2.GREEN;
```

7.4. Das DEFINED-Attribut

Daten können überlagert werden; d.h. verschiedene Bezeichner benutzen die gleichen Speicherplätze. Wertmäßige Veränderungen einer Variablen ziehen Veränderungen in Basisvariable und allen übergeordneten Variablen nach sich. Basisbezeichner und übergeordneter Bezeichner müssen verträglich sein.
Es bestehen drei Möglichkeiten, Daten mit dem DEFINED-Attribut statisch zu überlagern:

1. einfaches Überlagern
2. kettenüberlagerndes Definieren
3. korrespondierendes Überlagern

Zu 1. Das einfache Überlagern

Allgemeine Form:

```
< einfaches überlagern > ::= { DEFINED | DEF }  < basisbezeichner >
```

Der Basisbezeichner muss eine Variable der Stufe 1 sein. Der übergeordnete Bezeichner kann von anderem Typ sein. Basisbezeichner und übergeordneter Bezeichner müssen verträglich sein.

Beispiel:

```
DCL BASIS CHAR (100),
      UEBER CHAR (25) DEF BASIS;
UEBER = 'DAS IST EIN DEXT';
```

Der Text wird in den ersten Stellen beider Zeichenketten abgelegt.

Beispiel:

```
DCL BASIS CHAR (100),
      UEBER (100) CHAR (1) DEF BASIS;
 UEBER (13) = 'T';
```

Damit besteht die Möglichkeit, einzelne Zeichen einer Kette anzusprechen.

Beispiel: Alle nichtnumerischen Zeichen eines Eingabebereiches sollen Null gesetzt werden.

```
DCL EINGABE CHAR(100),
       SPALTE (100) CHAR (1) DEF EINGABE:
DO I = 1 TO 100;
        IF SPAKTE(1) < '0' | SPALTE (I) > '9' THEN SPALTE (I) = '0';
END;
```

Zu 2. Kettenüberlagerndes Definieren

Allgemeine Form:

```
< kettenüberlagerndes definieren > ::= { DEFINED | DEF } < basisbezeichner > { POSITION(n) |
POS(n)}
```

Basisbezeichner und übergeordneter Bezeichner müssen beide vom Typ CHAR oder BIT sein. POS(n) beschreibt die Position des Zeichen oder Bits, ab dem die Überlagerung beginnen soll.

Beispiel:

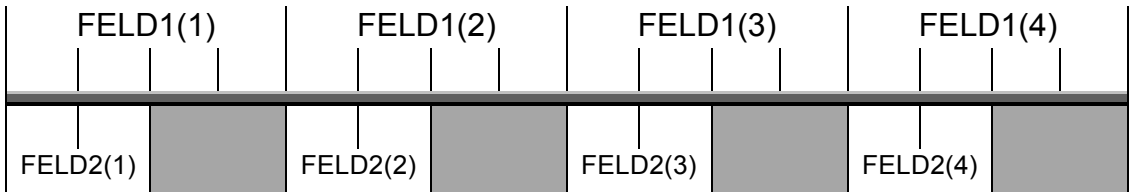
```
DCL BASIS CHAR (100),
      UEBER CHAR (4) DEF BASIS POS (97);
 UEBER  = ' ENDE';
```

Man beachte den Unterschied in der Speicherung der übergeordneten Daten:

Beispiel 1:

```
DCL FELD1 (4) CHAR (4),
      FELD2 (4) CHAR (2) DEF FELD1;
```

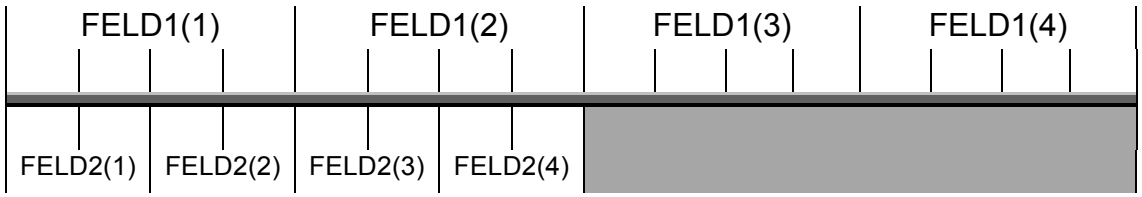
Speicherung:



Beispiel 2:

```
DCL FELD1 (4) CHAR (4),
      FELD2 (4) CHAR (2) DEF FELD1 POS(1);
```

Speicherung:



Zu 3. Korrespondierendes Überlagern

Das korrespondierendes Überlagern liegt vor, wenn in der Indexliste des übergeordneten Bezeichners mindest eine nSUB Variable enthalten ist. Eine nSUB Variable ist eine Bezugnahme in der Indexliste des Basisfeldes auf die n.Dimension des DEF-Feldes.

Allgemeine Form:

```
< korrespondierendes überlagern > ::= { DEFINED | DEF }  < basisbezeichner > [ ( indexliste ) ]
```

Beispiel 1:

```
DCL MATRIX (5,10) CHAR (1),
      ZEILVEKTOR (10) CHAR(1) DEF MATRIX (3, 1SUB),
      SPALTVEKTOR (5) CHAR(1) DEF MATRIX (1SUB, 5);
```

ZEILVEKTOR: Der Zeilenvektor wird auf der 3.Zeile ab 1.Spalte der Matrix definiert

SPALTVEKTOR: Der Spaltenvektor wird ab 1.Zeile auf der 5.Spalte der Matrix definiert.

Beispiel 2:

```
DCL MATRIX (5,10) CHAR (1),
      TEILMATRIX (2,3) CHAR(1) DEF MATRIX (3SUB, 8SUB);
```

TEILMATRIX: Die Teilmatrix wird ab der 3.Zeile und ab der 8.Spalte definiert.

7.5. Das CELL-Attribut

Bei Nutzung des DEFINED - Attributes ermöglichen verschiedene Adressen Bezugnahmen auf gleiche Daten - Datengleichheit.
Die Bezugnahme auf einzelne Variable in der CELL - Liste erfolgt alternativ. Es liegt eine Speichergleichheit vor. Daraus resultieren folgende Regeln:

- Eine Verträglichkeit der einzelnen Daten ist nicht erforderlich, da bei Bezugnahme unter Nutzung des CELL-Attributes die Daten alternativ genutzt werden
- Nur eine Alternative kann zu einem Zeitpunkt aktiv sein
- Nur eine Alternative kann zu einem Zeitpunkt einen aktuellen Wert enthalten
- Eine Zuweisung eines Wertes zu einer anderen Alternative hebt die letzte Wertzuweisung auf
- Nur eine Alternative kann das INIT-Attribut besitzen
- Jede Zellenbezeichner -Dimensionsangabe geht auf die Alternativen dieser Zelle über
- Die Anwendung des CELL-Attributes ist nur bei riesigen Datenmengen sinnvoll

Allgemeine Form:

```
< alternatives überlagern > ::= CELL   < alternativliste >
```

Beispiel:

```
DCL 1 A,
      2  B(10,10) FIXED (7,2),
      2 C CELL,       /* ZELLE C HAT 3 ALTERNATIVEN, C1,C2,C3 */
        3 C1 FIXED,
        3 C2 CHAR (3),
        3 C3 BIT (24),
      2  D (20) CELL,              /* Z.B. D(10).D1 UND D1(10) SIND GLEICHE BEZUGNAHMEN
*/
        3 D1 FIXED(7,2),
        3 D2 CHAR (4);
```

8. Subroutinen, Funktionsprozeduren, Standardfunktionen

8.1. PROCEDURE-Blöcke

In jedem PL/1 Programm muss eine Procedur als Hauptprocedur bzw. als externe Procedur gekennzeichnet werden, die dem Betriebssystem als Startprocedur mitgeteilt wird.
PROCEDURE-Blöcke können miteinander verschachtelt werden. Interne Blöcke in verschachtelten Blöcken sind interne Proceduren.

Beispiel zur Schachtelungen von Blöcken:

```
  HAUPT: PROC OPTIONS (MAIN);
                anweisung 1;
        UP1: PROC;
anweisung 2;
anweisung 3;
        END UP1;
        UP2: PROC;
anweisung 4;
              BL1: BEGIN;
                     anweisung 5;
               END BL1;
 anweisung 6;
        X: ENTRY;
anweisung 7;
UP3: PROC;
      anweisung 8;
 END UP3;
 anweisung 9;
        END UP2;
 anweisung 10;
  END HAUPT;
```

Beim Ablauf der PL/1 Anweisungsfolge ist die Procedur sequentiell nicht erreicht; sie ist nur durch einen speziellen Aufruf aktivierbar. Deshalb müssen alle Procedurblöcke mit einem Namen, einer Marke, versehen sein.

8.2. Gültigkeitsbereich von Namen

Ein Name, ein Bezeichner, hat nur Gültigkeit innerhalb eines Blockes, in dem er vereinbart wurde. Sein Gültigkeitsbereich kann nicht nach außen erweitert werden, in den internen Blöcken ist der Name bekannt.
Regel 1: Bei externen Proceduren sind die Namen der einzelnen Proceduren gegenüber unbekannt. Der Gültigkeitsbereich bezieht sich nur auf die Procedur, in dem er deklariert wurde.

Beispiel:

```
HAUPT: PROC OPTION (MAIN);
          DCL VAR_1 FIXED;
          .....
          VAR_1 = 234;
    UP_1: PROC;                          /* VAR_1 BEKANNT, EXTERNER NAME     */
            DCL VAR_2 FLOAT;
              VAR_2 = 12345;             /* VAR_2 INTERER NAME BZGL. UP_1      */
             ........
    END UP_1;
    UP_2: PROC;                          /* VAR_1 BEKANNT, EXTERNER NAME     */
           DCL VAR_3 FIXED;
           VAR_3 = 0;                                    /* VAR_3 INTERER NAME BZGL.
UP_2      */
            .....
    END UP_2;
          .....          /* VAR_2 UND VAR_3 UNBEKANNT          */

END HAUPT;
```

Regel 2: Bei geschachtelten Proceduren kann man sich nur auf Namen beziehen, die in übergeordneten Proceduren vereinbart wurden, soweit diese Namen in der internen Procedur nicht neu vereinbart wurden.

Beispiel:

```
HAUPT: PROC OPTION (MAIN);
          DCL VAR_1 FIXED;
          folge von anweisungen;
          VAR_1 = 234;
    UP_1: PROC;                          /* VAR_1 BEKANNT, EXTERNER NAME    */
            DCL VAR_2 FLOAT;             /* VAR_3 UNBEKANNT, INTERNER NAME */
              VAR_2 = 12345;
              folge von anweisungen;
        UP_2: PROC;                                   /* VAR_1 BEKANNT, EXTERNER
NAME     */
              DCL VAR_3 FIXED;           /* VAR_2 UNKANNT, INTERNER NAME      */
              VAR_2 0;
              folge von anweisungen;
       END UP_2;
    END UP_1;
END HAUPT;
```

8.3. Aktivierung und Aufruf von Proceduren

Eine Prozedur heißt aktiviert, wenn sie an einem ihrer Eingangspunkte aufgerufen wird. Eine Prozedur wird beendet, wenn die Programmablaufsteuerung auf die END-Klausel oder auf die RETURN-Klausel stößt. Die END-Klausel ist zwingend erforderlich.

Es gibt zwei Möglichkeiten, Prozeduren zu aktivieren:
1. durch CALL-Anweisung bei Subroutinen
2. durch Funktionsaufruf bei Funktionsprozeduren

Wird als Rückgabewert genau ein Wert erwartet, so handelt es sich um eine Funktionsprozedur. Wird kein Wert oder mehr als ein Wert als Rückgabewert der Prozedur erwartet, so benutzt man die Subroutinen.

Je nach Art der Parameteranwendung und der Art des Aufrufes unterscheidet man:

Subroutine
- Aufruf durch CALL
- Anzahl der Argumente ist von 0 bis 64 möglich

Funktionsprozedur
- Aufruf über den Funktionsnamen
- Rückgabe genau ein einfaches Element

Eingefügte Funktion - BUILTIN - Funktionen
- fester Bestandteil des PL/1 Sprachumfanges
- Aufruf über den Funktionsnamen
- Anzahl der Parameter und Argumente je nach Einsatzgebiet

8.4. Subroutinen

8.4.1. Aktivierung und Aufruf von Subroutinen

Allgemeine Form:

<call-anweisung> ::=CALL <subroutine> [(< liste von argumenten>)];
< liste von argumenten> ::= maximal 74 argumente

Als argument gilt:
- einfacher ausdruck
- markenkonstante
- markenvariable
- eingangsname
- dateiname
- bereichsname
- strukturname
- zeiger-name
- zeiger-ausdruck

Als Parameter wird die Größe bezeichnet, die in der Prozedur das Argument aufnimmt. Die Parameter geben grundsätzlich an, wie die Argumente beim Prozeduraufruf verwendet werden.
Argumente können Prozeduren Werte übergeben und auch übernehmen.
Die Attribute der Argumente und die der Parameter müssen verträglich sein, ansonsten muss das ENRTY-Attribut benutzt werden.

Beispiel:

```
HAUPT: PROC OPTIONS (MAIN);
```

```
        folge von anweisungen;
    UNTERPROG: PROC( PARM_1, PARM_2);
        DCL (PARM_1, PARM_2) FIXED;
        folge von anweisungen;
        IF ARG_1 = ARG_2 THEN RETURN;    /* WENN BEDINGUNG ERFUELLT,
RUECKSPRUNG */
                                          /* ANWEISUNG HINTER CALL */
    END UNTERPROG;
      folge von anweisungen;
      CALL UNTERPROGR ( ARG_1, ARG_2);
      folge von anweisungen;
END HAUPT;
```

8.4.2. Übergabe von Bereichen

Sollen abwechselnd unterschiedliche Bereiche an eine Subroutine übergeben werden, bedient man sich bei der Parameter-Argumentübergabe der sog. Sternschreibweise. Bei dieser Methode wird kein neuer Speicherplatz belegt. Zur Ermittlung der aktuellen grenzen werden die eingefügten Funktionen HBOUND( ) und LBOUND( ) verwendet.

LBOUND(bereich, i) stellt die gegenwärtig untere Grenze der i.Dimension von bereich dar
HBOUND(bereich, i) stellt die gegenwärtig obere Grenze der i.Dimension von bereich dar

Beispiel:

```
HP: PROC OPTIONS (MAIN);
DCL BER1 (50, 20) CHAR(10),
    BER2 (50, 25) CHAR(20),
    BER3 (50, 30) CHAR(15);

EING: GET LIST (BER1, BER2, BER3);
    folge von anweisungen;
    CALL UPDRUCK (BER1);
    CALL UPDRUCK (BER2);
    CALL UPDRUCK (BER3);

END HP;

UPDRUCK: PROC (BEREICH);
    DCL BEREICH (*,*) CHAR (*);          /* KEIN NEUER SPEICHERPLATZ */

    IU  = LBOUND(BEREICH,1);             /* UNTERE GRENZE 1.INDEX    */
    IO  = HBOUND(BEREICH,1);             /* OBERE GRENZE 1.INDEX     */
    KU = LBOUND(BEREICH,2);              /* UNTERE GRENZE 2.INDEX    */
    KO = HBOUND(BEREICH,2);              /* OBERE GRENZE 2.INDEX     */

    DO I = IU TO IO;
      DO K = KU TO KO;
      /* DRUCKAUFBEREITUNG UND DRUCK DES AKTUELLEN BEREICHES */
      END;
```

```
        END;
    END UPDRUCK;
```

8.5. Funktionsprozeduren

Soll in einem Unterprogramm genau ein skalarer Wert ermittelt und über den Namen der Procedur an die Aufrufstelle zurückgegeben werden, so wird die Funktionsprozudur eingesetzt.

Allgemeine Form:

<funktionsprozedur> ::= PROC (parameterliste) attribut des rückgabewertes;

Der errechnete Funktionswert wird an die Aufrufstelle zurückgegeben und falls erforderlich konvertiert. Die Ausführung einer Funktionsprozedur muss mindestens durch eine Klausel der Form

RETURN (funktionswert);

beendet werden. Der errechnete Funktionswert wird der Aufrufstelle als skalarer Wert zurückgegeben. Die notwendige END-Klausel dient nur als Mitteilung für des Ende der zugehörigen Befehlsfolge an den Compiler.

Beispiel 1:

```
HAUPTPROC: PROC OPTIONS (MAIN);

                folge von anweisungen;

/* (1) */       ERG = W1 * W2 / VOLUMEN (ARG1, ARG2, ARG3);  /* UEBERGABE ARGUMENTE
*/
                PUT LIST (ERG);
                folge von anweisungen;
END HAUPTPROC;

VOLUMEN:  PROC ( LAENGE, BREITE, HOEHE) FIXED (9,2);    /*  RUECKGABEWERT FIXED
(9,2)  */
                DCL (LAENGE, BREITE, HOEHE) FIXED (5,2);
                RETURN (LAENGE * BREITE * HOEHE);  /* ERGEBNIS WIRD IN (1) EINGEFUEGT
*/
END VOLUMEN;
```

Ablauf: Wird in der Anweisung (1) der Funktionsaufruf VOLUMEN in der sequentiellen Abarbeitungsfolge erreicht, so wird die entsprechende Funktionsprozedur aktiviert und der Funktionswert mit den aktuellen Argumenten ausgeführt. Die RETURN-Klausel veranlasst den Rücksprung an die Aufrufstelle und die Bearbeitung der Anweisung (1) wird fortgeführt.

Beispiel 2:

Es soll eine quadratische Gleichung der Form

Ax *x + Bx + C = 0 mit den Lösungen $x_{1,2} = -p/2 \pm \sqrt{(p*p/4 - q)}$

programmiert werden, wobei die Beziehungen p = B/A und q = C/A gelten.

```
QUADR_GLEICH: PROC OPTIONS (MAIN),
DCL (A,B,C,P,Q,X1,X2) FIXED(9,2);
ON ENDFILE (SYSIN) GOTO ENDE;
LIES: GET LIST (A,B,C);
        IF A = 0 THEN DO; PUT LIST (' KOEFFIZIENT A = 0') SKIP;  /* VERMEIDUNG DIV. DURCH
      0 */
                              GOTO LIES;
                        END;
        P = B/A;
        Q = C/A;
        X1 = WURZEL_1 (P,Q);
        X2 = WURZEL_2 (P,Q);
        PUT LIST ('X1 = ', X1, 'X2 = ', X2) SKIP;
        GOTO LIES;

WURZEL_1: PROC (X,Y) RETURNS (FIXED (9,2));     /* FUNKTIONSWERT IST FIXED (9,2) */
  DCL (X,Y) FIXED (9,2);
  RETURN (- X / 2 + SQRT (X * X / 4 - Y);    /* ERGEBNISWERT WIRD X1 ZUGEORDNET */
END WURZEL_1;

WURZEL_2: PROC (X,Y) RETURNS (FIXED (9,2));
  DCL (X,Y) FIXED (9,2);
  RETURN (- X / 2 - SQRT (X * X / 4 - Y);     /* ERGEBNISWERT WIRD X2 ZUGEORDNET */
END WURZEL_2;

ENDE: PUT LIST ('ENDE DES PROGRAMMS') SKIP;
END QUAD_GLEICH;
```

8.6. Wichtige Spezifikationen für Subroutinen und Funktionsprozeduren

Spezifikationen beinhalten verschiedene Aufgaben zur Kennzeichnung spezieller Eigenschaften des jeweiligen Unterprogramms.

8.6.1. OPTIONS(MAIN) Spezifikation

Diese Spezifikation ist für solche Programme anzugeben, die als Hauptprogramm verwendet werden soll. In jedem Programm muss folglich mindestens ein Teilprogramm mit dieser Spezifikation versehen sein. Das Betriebssystem erhält auf diese Weise die Mitteilung, welches Programm die Steuerung erhalten und damit mit der gesamten Abarbeitung beginnen soll.

8.6.2. Die RETURN-Klausel

Die Beendigung einer Subroutine wird normalerweise dadurch bewirkt, dass die Befehlsablaufsteuerung die END-Klausel erreicht. Es ist aber auch möglich, über die RETURN-Klausel die Steuerung an das aufrufende Programm zu übergeben.

Beispiel:

```
UNTER_PROG:PROC ( LAENGE, BREITE, HOEHE);
folge von anweisungen;
IF LAENGE = 0 | BREITE = 0 | HOEHE = 0 THEN RETURN;
folge von anweisungen;
END UNTER_PROG;
```

Im Unterschied zu den Subroutinen muss bei den Funktionsprozeduren stets der Funktionswert an das aufrufende Programm zurückgegeben werden. Die END-Klausel dient hier nur als Mitteilung an den Compiler über das Ende der vorliegenden Befehlsfolge. Eine Funktionsprozedur muss deshalb zur Berechnung des Funktionswertes, der an die rufende Prozedur zurückgeben wird, mindestens eine RETURN-Klausel der Form

RETURN(ausdruck);

enthalten. Nach der Berechnung von (ausdruck) steht der Funktionswert unmittelbar mit den Attributen des Eingangsnamens (siehe 2.3) zur Verfügung.

Beispiel:

```
WURZEL_1: PROC (X,Y) RETURNS (FIXED (9,2));     /* FUNKTIONSWERT IST FIXED (9,2) */
  DCL (X,Y) FIXED (9,2);
  RETURN (- X / 2 + SQRT (X * X / 4 - Y);
END WURZEL_1;
```

8.6.3. Das RETURNS Attribut

Soll der Funktionswert einer Funktionsprozedur nicht die Standardattribute erhalten, die durch den Anfangsbuchstaben des Eingangsnames definiert sind (siehe 2.3), muss das Attribut des Rückgabewertes explizit angegeben werden.

Allgemeine Form:

<RETURNS-attribut> ::= {PROC | ENTRY} (parameterliste) RETURNS (attribute des funktionswertes);

Beispiel:

Ein Funktionswert soll als Gleitkommazahl mit verlängerter Mantisse zurückgegeben werden.

```
WURZEL: PROC (X) RETURNS (FLOAT(16));
RETURN (X * X + SQRT(1E0 + 2E0 * X + 3E0 * X * X));
END WURZEL;
```

8.6.4. Die ENTRY-Klausel

Man kann bei Subroutinen und Funktionsprozeduren sekundäre Einspringpunkte festlegen. Die ENTRY-Klausel gibt einen solchen sekundären Eingangspunkt eines

PROC-Blockes an. Sie kann ebenfalls eine Parameterliste, die RETURNS-Option zur Vereinbarung der Attribute des Rückgabewertes und andere Optionen enthalten.

Allgemeine Form:

<ENTRY-klausel> ::= ENTRY [(parameterliste)] [(datenattribute)];

Abschließendes Beispiel siehe 8.6.5 Beispiel 2

8.6.5 Das ENTRY-Attribut

Stimmen die Attribute der Argumente mit denen der Parameter nicht überein, so wird das ENTRY-Attribut verwendet. Die Attribute der Argumente werden denen der Parameter angepasst. Diese Vorgehensweise ist notwendig, wenn man Subroutinen oder Funktionsprozeduren benutzt, ohne deren inneren Aufbau und somit die Art der Attribute zu kennen.

Allgemeine Form:

<ENTRY-attribut> ::= DCL <name der procedur> ENTRY (<liste der attribute der parameter>);

Beispiel 1:

```
UNTERPROG: PROC (ARG1,ARG2);
                DCL ARG1 FIXED(9,2),
                    ARG2 FLOAT;
                folge von anweisungen;
END UNTERPROG;

HAUPTPROG: PROC OPTIONS(MAIN);
                DCL UNTERPROG ENTRY (FIXED(10,2), FIXED (7,2));
                CALL UNTERPROG (PAR1,PAR2);  /* INTERNE UMWANDLUNGEN DER
ATTRIBUTE */
                folge von anweisungen;
   END HAUPTPROG;
```

Beispiel 2:

```
HAUPT_PROG: PROC OPTIONS (MAIN);
   DCL UP1 ENTRY (FIXED BIN) RETURNS (FIXED DEC (10,5));
   DCL ARG BIN FIXED,
       ERG DEC FIXED (10,5);
   ERG = UP1 (ARG);
   folge von anweisungen;

   UP1: PROC (PARM) RETURNS (FIXED DEC (10,5));
   DCL PARM BIN FIXED,
       PIE FIXED (5,4) STATIC INIT (3.1415);
   IF PARM = 0 THEN RETURN (0);
   RETURN (ARG * PIE);
   END UP1;
   folge von anweisungen;
END HAUPT_PROG;
```

8.7. Standardfunktionen

Standardfunktionen, auch eingefügte Funktionen oder BUILTIN-Funktionen, sind Bestandteile des Sprachumfangs. Sie werden nicht explizit vereinbart; abgesehen davon, dass bei Verwendung des Namens einer Standardfunktion für andere Objekte dann die Standardfunktion mit BUILTIN vereinbart werden muss.

Allgemeine Form des Standardfunktionsaufrufs:

```
< aufruf standardfunktion> ::= < name der standardfunktion > [ ( < liste von argumenten >) ];
<vereinbarung standardfunktion> ::= BUILTIN < name der standardfunktion > ;
```

Beispiel:

```
HP: PROC OPTIONS (MAIN);
    folge von anweisungen;
    UP: PROC;
        DCL (WERTE(10), SUM(10)) FIXED (7,2);       /* SUM WIRD ALS FELD BENUTZT */
        DO I = 1 TO 10;
        SUM (I) = SUM (I) + WERTE(I);
        END;
        BL1: BEGIN;
             DCL SUM BUILTIN;     /* SUM SOLL ALS STANDARDFUNKTION BENUTZT WERDEN
             */
             DCL SUMME FLOAT;
             SUMME = SUM(liste von elementen);    /* BILDUNG DER SUMME ALLER PARAMETER
             */
             folge von anweisungen;
        END BL1;
        folge von anweisungen;
    END UP;
    folge von anweisungen;
END HP;
```

Bei der weiteren Abarbeitung des Ausdrucks wird an der Stelle dieses Aufrufs der jeweilige Funktionswert gesetzt und anschließend wie ein gewöhnlicher Operand verwendet.

8.7.1. Überblick über Standardfunktionen

Die Standardfunktionen lassen sich nach ihrer Verwendung in Gruppen unterteilen:

Gruppe	Name der Funktion	Bedeutung, Funktionswert

Funktionen für mathematische Grundfunktionen	ATAN, ATAND ATANH SIN, SIND, COS, COSD TAN, TAND SINH, COSH, TANH EXP ERF, ERFC LOG, LOG10, LOG2 SQRT	Arcustangens in Bogenmaß und Altgrad Arcustangens Hyperbolicus Sinus, Cosinus Tangens Hyperbolische Funktionen Exponentialfunktion Gaußsche Fehlerfunktionen Logarithmische Funktionen Wurzelfuktionen
Funktionen für Datenumwandlungen	BIT, CHAR BINARY, DECIMAL FIXED, FLOAT PRECISION	Konvertierungen in Bitkette, Zeichenkette Konvertierungen in Dual- bzw. Decimaldarstellung Konvertierungen in Fest- bzw. Gleitkommadarst. Veränderung des Genauigkeitsattributes
Funktionen für die Behandlung komplexer Daten	COMPLEX CONJG IMAG, REAL	Komplexer Wert mit Real- und Imaginärteil konjugiert komplexen Wert a - bi zum Wert a + bi reeller Wert, der den Imaginär- bzw. den Realteil des Arguments darstellt
Funktionen für arithmetische Funktionen	ADD MULTIPLY DIVIDE	Addition mit vorgegebener Genauigkeit Multiplikation mit vorgegebener Genauigkeit Division mit vorgegebener Genauigkeit
Funktionen für die Behandlungen mehrerer Wert	MOD MAX MIN	Rest einer Division Maximum aller angegebenen Argumente Minimum aller angegebenen Argumente
Funktionen für die Behandlung arithmetischer Werte	ABS SIGN ROUND CEIL FLOOR TRUNC	Absolutbetrag Bestimmung des Vorzeichens Rundung kleinste ganze Zahl, die nicht kleiner als Argument größte ganze Zahl, die nicht größer als Argument Wert von Argument ohne gebrochenem Anteil
Funktionen zur Behandlung von Bit- und Zeichenketten	INDEX, LENGTH, REPEAT SUBSTR, TRANSLATE	wird im Punkt 8.6.2. behandelt
Funktionen zur Erzeugung und Behandlung von Bit- und Zeichenkettenketten	HIGH, LOW STRING, VERIFY,	Bildung und Bearbeitung von Ketten
Funktionen für logische Operationen	BOOL ALL ANY	bitweise logische Verknüpfung der angegeben Argumente Erzeugung Bitkette durch Verknüpfung aller Feldelemente mit Operator & Erzeugung Bitkette durch Verknüpfung aller Feldelemente mit Operator |
Funktionen für die Behandlung von Feldwerten	DIM LBOUND HBOUND	Ermittlung Dimensionsanzahl Ermittlung untere Grenze Ermittlung obere Grenze
Funktionen für Felder	SUM PROD	Erzeugung Summe aller Elemente Erzeugung Produkt aller Elemente
Funktionen für die Anwendung mehrgliedriger Ausdrücke	POLY	Wert eines Polynoms aus den beiden Argumenten
Funktion zur Feststellung der Auftrittsursachen bei Ausnahmezuständen	DATAFIELD ONCHAR ONCODE ONFILE ONKEY ONLOC ONSOURCE	fehlerhaftes Datenfeld mit DATA-Option fehlerhaftes Zeichen Erzeugung Fehlercode fehlerhafte Datei fehlerhafter Schlüssel fehlerhafte Prozedur fehlerhaftes Datenfeld
Funktionen für die Arbeit mit AREA Variablen	ADDR EMPTY NULL, NULLO	Die Funktionswerte sind spezielle Adressen entsprechend der Verwendung
Funktionen für Multiaufgabenbetrieb und EVENT Variablen	COMPLETION PRIOTRITY STATUS	Beendigung einer EVENT-Variablen Priorität einer Aufgabe Zustandswert einer EVENT-Variablen
Funktionen zur Kontrolle der Ein - und Ausgabe	COUNT LINENO	Anzahl der ein- oder ausgegebenen Elemente Aktuelle Zeilennummer beim Druck
Funktionen für Datum und Uhrzeit	DATE TIME	Bereitstellung des aktuellen Datums Bereitstellung der aktuellen Uhrzeit zum Zeitpunkt des Funktionsaufrufes
Funktionen für Speicherplatzzuordnung für CONTROLLED - Variable	ALLOCATION	Gibt an, ob Speicherplatz zugeordnet wurde

Funktionen für die Ermittlung des binären Aufbaus von Daten	UNSPEC	Bitkette der internen Darstellungsform

8.7.2. Bearbeitung von Kettenoperationen

Die Standardkettenfunktionen sind auf Zeichen- und Bitketten anwendbar.

1. Bildung einer Teilkette : SUBSTR (kette, p [l])

   kette - Die Kette, aus der die Teilkette gebildet werden soll
   p - Bildung der Teilkette ab p-ter Position, 1 <= p <= kette
   l - Länge der Teilkette, 1 <= l <= kette

Beispiel:

```
DCL KET CHAR (8) INIT ('COMPUTER'),
      TKETA(8) CHAR(1),
      TKETB(8) CHAR(8),
      TKETC(8) CHAR(8) VARYING;
    DO IND = 1 TO 8;
       TKETA(IND) = SUBSTR (KET, 1, IND);      /* TKETA(1) = 'C';      TKET(8) = 'R';  */
       TKETB(IND) = SUBSTR (KET, 1, IND);      /* TKETB(1) = 'C               ' ;  */
                                               /* TKETB(2) = 'CO              ';   */
                                               /* TKETB(8) = 'COMPUTER';   */
       TKETC(IND) = SUBSTR (KET, IND, 1);      /* TKETC(1) = 'C';                  */
                                                /* TKETC(2) = 'CO';                */
                                                /* TKETC(8) = 'COMPUTER';   */
    END;
```

SUBSTR kann auch als Pseudovariable benutzt werden.

Beispiel:

```
DCL KET CHAR (8) INIT ('COMPUDER');
      SUBSTR (KET,6,1) = 'T';
```

2. Kettenfunktion INDEX (kette1, kette2)

   Man kann feststellen, ob kette2 in kette1 enthalten ist.
   Der Funktionswert ist eine binäre ganze Zahl, die die Position
   der Fundstelle mitteilt. Er ist 0B, wenn kette2 nicht in kette1 enthalten ist

Beispiel:

```
DCL KET CHAR (8) INIT ('COMPUTER'),
  I = INDEX (KET, 'T');       /* I = 6 */
```

3. Ermittlung der aktuellen Länge LENGTH(kette)

   Man kann die aktuelle Länge einer VARING-Kette ermitteln.

Beispiel:

```
DCL KET CHAR (100) VAR;
                KET = 'COMPUTER';
                DO IND = 1 TO LENGTH (KET);    /* I ND LAEUFT VON 1 BIS 8 */
                  folge von anweisungen;
                END;
```

4. Umwandlung von Zeichen- und Bitketten TRANSLATE (kette1, kette2 [, kette3])
   In kette1 sollen Zeichen geändert werden, kette2 enthält alle
   einzusetzenden Zeichen, kette3 zeigt die zu übersetzenden Zeichen aus kette1 an

Beispiel:

```
DCL (KETEIN, KETAUS) CHAR (100),
                ( ZEICHALT, ZEICHNEU) CHAR (2) INIT ('.$', ', €');
                GET LIST (KETEIN);
                 KETAUS = TRANSLATE( KETEIN, ZEICHNEU, KETALT);
                 folge von anweisungen;
                PUT LIST (KETAUS);
```

5. Wiederholung von Zeichen oder Bits REPEAT (kette, zahl)
   zahl gibt an, wie oft kette wiederholt werden soll

Beispiel:

```
KET = REPEAT ('ABC', 3);     /* KET = 'ABCABCABC' */
```

8.8. Dynamisches Laden und Freigeben von externen Prozeduren

Es besteht auch die Möglichkeit, externe Lademoduln während der Ausführung dynamisch zu laden und wieder freizugeben. Diese Methode ist sehr nützlich bei der Programmierung
riesiger Projekte, beim Einsatz vorgefertigter Programmpakete, bei der Verbindung mit anderen Sprachen wie COBOL oder FORTRAN bzw. zur Realisierung der sehr wichtigen PL1-Assembler Verbindungen.

Allgemeine Form:

<dynamisches laden> ::= FETCH <lademodul>;
<freigeben> ::= RELEASE <lademodul>;

Beispiel:

```
HAUPT: PROC OPTION (MAIN);
          DCL  UP_1  ENTRY EXTERNAL,
               UP_2  ENTRY ((CHAR(100)) RETURNS ((CHAR(10))  EXTERNAL;
          .....;
          FETCH UP_1;                    /* DYN. LADEN AUS LADEMODULBIBL. IN HS  */
          CALL UP_1;                     /* AKTIVIEREN UP_1 */
```

```
        .....;
        RELEASE UP_1;                   /* IM HS FREIGEBEN  */
        .......;
        KETTE = UP_2 (ARG);             /* DYNAM. LADEN UND AKTIVIEREN     */
                                        /* KETTE LAENGE 100 UEBERGEBEN     */
                                        /* RÜCKGABEWERT KETTE LAENGE 10 */
        RELEASE UP_2;                   /* IM HS FREIGEBEN  */
END HAUPT;
```

Mit dieser Methode sind auch die wichtigen PL/1-Assembler Verbindungen zu realisieren.

Beispiel:

```
HAUPT:
        folge von vereinbarungen und anweisungen;

            DCL UPASS ENTRY OPTIONS(ASSEMBLER);
            DCL ARG CHAR (100) EXTERNAL;

          folge von anweisungen;
          FETCH  UPASS;
          CALL  UPASS(ARG);   /*AUFRUF UND ABARBEITUNG ASSEMBLERMODUL */
```

9. Dateibeschreibung PL/1

Der PL/1 Compiler übt eine nur sehr geringe Kontrolle über die Durchführung der Ein- und Ausgabeoperationen aus. Es müssen verschiedene Angaben über die Organisation und Verwendung der Datei dem IOCS, dem E/A-Steuersystem des Betriebssystems zugeführt werden, ehe die E/A-Operation korrekt ausgeführt werden kann.
Die notwendigen Steuerinformationen werden folgenden Quellen entnommen:
- aus dem Betriebssystem
- aus der DCL-Vereinbarung
- aus dem OPEN-Statement
- aus der E/A-Anweisung

9.1. Dateiattribute

Dateiattribute sind Bestandteil der DCL-Vereinbarungen zur Definition der zu benutzten Dateien.

9.1.1. Das FILE-Attribut

Das FILE-Attribut dient zur Vereinbarung des Dateinamens.

Allgemeine Form:

< FILE-attribut> ::= DCL < name > FILE [<attribute>];

Das FILE-Attribut bewirkt, dass name mit einer externen Datenmenge auf einem externen Speichermedium verknüpft wird, die über eine bestimmte Dateneinheit mit den angegeben Attributen ein- oder ausgegeben werden soll.

Beispiel:

```
DCL  EING_DATEI FILE;
```

EING_DATEI ist eine Datei, die in weiteren Anweisungen noch näher spezifiziert werden muss.

9.1.2. Die alternativen Attribute

Die alternativen Attribute beschreiben in der DCL-Anweisung die Art der zu definierten Datei.

9.1.2.1. Die FILE-Verwendungsattribute

Allgemeine Form:

<FILE-verwendungsattribut> ::= STREAM | RECORD

STREAM beschreibt die reihenweise E/A, RECORD die satzweise E/A. Wird kein Verwendungsattribut angegeben, wird STREAM als Standardannahme angenommen.
Die reihenweise E/A darf nur in Verbindung mit den Anweisungen GET und PUT angewendet werden.
Die satzweise E/A darf nur in Verbindung mit den Anweisungen READ, WRITE, LOCATE und REWRITE angewendet werden.

Beispiel:

```
DCL  EING_DATEI FILE RECORD;
```

EING_DATEI ist eine Datei mit satzweiser Verarbeitung, die in weiteren Anweisungen noch näher spezifiziert werden kann.

9.1.2.2. Die FILE-Funktionsattribute

Allgemeine Form:

<FILE-funktionsattribut> ::= INPUT | OUTPUT | UPDATE

INPUT beschreibt eine Datei, die Daten von einem externen Medium über ein Programm in einen bestimmten Hauptspeicherbereich einliest.
OUTPUT beschreibt eine Datei, die Daten von einem bestimmten Hauptspeicherbereich über ein Programm in ein externes Medium schreibt.
UPDATE beschreibt eine Datei, die Daten von einem externen Medium über ein Programm in einen bestimmten Hauptspeicherbereich zum Verändern einliest und wieder auf das externe Medium zurückschreibt.

Beispiel:

```
DCL  EING_DATEI FILE RECORD OUTPUT;
```

9.1.2.3. Die FILE-Zugriffsattribute

Allgemeine Form:

<FILE-zugriffsattribut> ::= SEQUENTIAL | DIRECT

Die Zugriffsattribute haben keinen Einfluss auf die Organisationsform der Daten.
SEQUENTIAL beschreibt eine Datei, in der die Datensätze in der Reihenfolge verarbeitet werden, wie sie in der Datei organisiert sind.

DIRECT beschreibt eine Datei, in der auf die Datensätze über einen Schlüssel oder eine Adresse zugegriffen werden kann.

Beispiel:

```
DCL  EING_DATEI FILE RECORD OUTPUT DIRECT;
```

9.1.2.4. Die FILE-Pufferungsattribute

Allgemeine Form:

<FILE-pufferungsattribut> ::= BUFFERED | UNBUFFERED

Diese Attribute geben an, ob die Übertragung der Sätze zwischen Datei und E/A-Bereiche gepuffert oder ungepuffert erfolgen soll; ob die Sätze während der Übertragung einen Zwischenspeicher (BUFFERED) passieren müssen oder nicht. Standardmäßig werden zwei Pufferbereiche angelegt. Sie dürfen nicht für STREAM und DIRECT angegeben werden.

9.1.3. Die additiven Attribute

Zusätzlich zu den alternativen Attribute müssen bei bestimmten Verarbeitungsformen weitere Attribute angegeben werden.

9.1.3.1. PRINT-Attribut

gilt nur für Druckdateien, OUTPUT STREAM wird impiziert. Das Druckersteuerzeichen wird über entsprechende Anweisungen innerhalb der OPEN- oder PUT-Anweisung an die erste Stelle des zu druckenden Streams entsprechend der benutzen Drucker automatisch eingetragen.

Beispiel:

```
DCL DRUCK FILE PRINT;
```

9.1.3.2. BACKWARDS-Attribut

legt fest, dass die entsprechende Datei rückwärts gelesen werden soll.

Beispiel:

```
DCL BAND_DATEI FILE RECORD INPUT SEQUENTIAL;
```

Die Datensätze werden in umgekehrter Reihenfolge gelesen, d.h. die Eingabe beginnt mit dem letzten Satz der Datei. Durch die INTO-Option in der READ-

Anweisung werden die Daten innerhalb des Datensatzes in der richtigen Reihenfolge zur Verfügung gestellt; durch die SET-Option in der READ-Anweisung wird dagegen der Zeiger auf das Ende des Datensatzes gesetzt (siehe Punkt 11.).

9.1.3.3. KEYED-Attribut

gibt an, dass die Datei unter Verwendung eines Schlüssels verarbeitet werden soll.

Beispiel:

```
DCL PLATTE FILE RECORD OUTPUT DIRECT KEYED;
```

KEYED muss für jede Datei angegeben werden, die Schlüssel enthält, auch wenn die Datensätze sequentiell verarbeitet werden sollen.

9.1.3.4. EXCLUSIVE-Attribut

gibt an, dass bei der Datei DIRECT und UPDATE verhindert werden soll, einen Datensatz zu lesen, löschen oder zurückzuschreiben, während eine andere Task diesen Satz liest, löscht oder zurückschreibt.

9.1.3.5. ENVIRONMENT-Attribut

beschreibt die Organisationsform der Datei..

Allgemeine Form:

<ENVIRONMENT-attribut> *::= { ENVIRONMENT | ENV } (<optionsliste>)*

< optionsliste> ::= CONSECUTIVE | INDEXED | REGIONAL(1) | REGIONAL(2) | REGIONAL(3)

Optionsliste	Organisationsform	Zugriffsform
CONSECUTIVE	starr fortlaufend	sequentiell
INDEXED	logisch fortlaufend	sequentiell oder direkt
REGIONAL(1)	direkte Satzadressierung	gestreut, sequentiell oder direkt
REGIONAL(2)	relative Satzadressierung	gestreut, sequentiell oder direkt
REGIONAL(3)	relative Spuradressierung	gestreut, sequentiell oder direkt

Die einzelnen Organisationsformen werden bei der reihenweisen und satzweisen Ein- und Ausgabe behandelt.

Weitere optionale Angaben:

Optionsliste	Angabe	Bedeutung
Satzformat	F(blocklänge [,satzlänge])	Sätze fester Länge
	V(max. blocklänge [max.satzlänge])	Sätze variabler Länge
	U(max. blöcklänge)	Sätze undefinierter Länge
Angabe Puffer	BUFFERS(n)	n - Anzahl der Puffer
Schlüssellänge	KEYLENGTH(n)	n - Schlüssellänge
Schlüsselposition	KEYLOC(n)	n - Position des Schlüssels

Beispiel:

```
DCL MBEIN FILE RECORD INPUT SEQUENTIAL ENVIRONMENT( BUFFERS(4) U(256));
```

Unter der Bezeichnung MBEIN wird eine starr fortlaufende Eingabedatei zur satzweisen Datenübertragung vereinbart, deren Sätze undefinierte Länge von maximal 256 Bytes haben und unter Verwendung von 4 Eingabepuffer übertragen werden.

9.2. Eröffnung und Abschluss von Dateien

Die Vereinbarung von Dateien hat zunächst nur symbolischen Charakter, es ist damit in keiner Weise eine Bezugnahme auf einen Datenbestand verbunden. Die Herstellung der Verbindung zwischen der im Programm vereinbarten Datei und dem externen Datenbestand erfolgt im Normalfall zum Zeitpunkt der Dateieröffnung.

9.2.1. Die OPEN-Anweisung

Durch die OPEN-Anweisung wird die interne Datenmenge mit dem Datenträger verknüpft. Die OPEN-Anweisung hat folgende Aufgaben:

1. dem FILE-Namen eine Datei zuzuordnen
2. den FILE eine vollständige Liste von Attributen zuzuordnen
3. Aktivieren von Routinen zur Bearbeitung von Standardkennsätzen
4. eventuell vorhandene Benutzerkennsätze zur Prüfung zur Verfügung zu stellen oder
   Benutzerkennsätze zu erstellen

Allgemeine Form:

< dateieröffnung > ::= OPEN FILE (<dateiname>) [< optionslguppen >]
[, FILE (< dateiname)) [< optionsgruppen >];

< optionsgruppen> ::= [SEQUENTIAL | DIRECT]
[BUFFERED | UNBUFFERED]
[STREAM | RECORD]
[INPUT | OUTPUT | UPDATE]
[KEYED]
[PRINT]
[TITLE (ausdruck)]
[IDENT (zeichenkettenvariable)]
[LINESIZE (elementausdruck)]
[PAGESIZE (elementausdruck)]

TITLE (ausdruck) ermöglicht, aufgrund bestimmter Bedingungen eine Datei mit unterschiedlichen Attributen zu eröffnen.

Beispiel:

```
DCL DATEI_1 FILE RECORD UPDATE,
      DATEI_2 FILE RECORD OUTPUT;

IF SEL = '1' THEN OPEN FILE (DATEI_1);
             ELSE OPEN FILE (DATEI_2) TITLE (DATEI_1);
```

IDENT (zeichenkettenausdruck) dient zur Bearbeitung von Benutzerkennsätzen. Wird eine Datei mit IDENT (zeichenkettenausdruck) eröffnet, so wird zeichenkettenausdruck als Vorsatz in die Datei eingetragen.

LINESIZE (ausdruck) legt die Länge einer Druckzeile fest.

Beispiel:

```
OPEN FILE (DRUCK) PRINT LINESIZE (120);
```

PAGESIZE (ausdruck) legt die Anzahl der Zeilen pro Seite fest.

Beispiel:

```
OPEN FILE (DRUCK) PRINT LINESIZE (120) PAGESIZE (60);
```

9.2.2. Die CLOSE-Anweisung

Die CLOSE-Anweisung hebt die zur Eröffnungszeit hergestellte Verbindung zwischen FILE-Konstante und Datei und veranlasst gegebenenfalls eine Dateiendebehandlung. Wird eine IDENT-Option angegeben, so werden beim Schließen der Datei Benutzerkennsätze geschrieben. Für Magnetbanddateien kann die beim Schließen durchzuführende Positionierung des Datenträgers durch die ENVIRONMENT-Angaben LEAVE und REREAD angegeben werden.

Allgemeine Form:

< dateiabschluss > ::= CLOSE FILE (<dateiname>) [IDENT (zeichenkettenausdruck)]
[ENVIRONMENT ({ LEAVE | REREAD})];

Beispiel:

```
DCL MBEIN FILE RECORD SEQUENTIAL ENVIRONMENT( BUFFERS(4) U(256));
OPEN FILE (MBEIN) INPUT;
Folge von anweisungen;
CLOSE FILE (MBEIN);
```

10. Reihenweise Ein- und Ausgabe

10.1. Arbeit mit Standarddateien

Vom Betriebssystem werden zwei Standarddateien zur Verfügung gestellt, die für einfache E/A-Operationen sehr hilfreich sind. Das zieht folgende Erleichterungen nach sich:

- keine Vereinbarungen, also keine DCL's, die notwendigen Dateiattribute werden durch
  das Betriebssystem zur Verfügung gestellt
- das Eröffnen und Schließen der Dateien wird ebenfalls durch des Betriebssystem realisiert; als keine OPEN- und CLOSE-Anweisungen bei Standardattributen
- in den E/A-Anweisungen selbst wird keine Datei benannt, also keine FILE-Option

Die Benutzung der **Standardeingabedatei** - SYSIN - impliziert folgende Dateiattribute:

- Funktionsattribut Eingabe, also INPUT
- reihenweise Verwendung, also STREAM
- sequentielle Zugriffsform, also SEQUENTIAL
- starr fortlaufende Organisationsform , also CONSECUTIVE

```
[ DCL SYSIN FILE  CONSECUTIVE STREAM SEQL ; ]
[ OPEN FILE (SYSIN) INPUT; ]
```

Die Eingabeanforderung wird durch GET erreicht. Die Angabe der Standarddatei FILE(SYSIN) ist ebenfalls nicht erforderlich.

```
GET [ FILE (SYSIN) ] LIST (eingabebereich);
```

Die Benutzung der **Standardausgabedatei** - SYSPRINT - impliziert folgende Dateiattribute:

- Funktionsattribut Ausgabe, also OUTPUT
- reihenweise Verwendung, also STREAM
- sequentielle Zugriffsform, also SEQUENTIAL
- starr fortlaufende Organisationsform , also CONSECUTIVE
- Druck-Attribut PRINT

```
[ DCL SYSPRINT FILE  CONSECUTIVE STREAM SEQL ; ]
[ OPEN FILE (SYSPRINT) PRINT OUTPUT; ]
```

Die Eingabeanforderung wird durch PUT erreicht. Die Angabe der Standarddatei FILE(SYSPRINT) ist ebenfalls nicht erforderlich.

```
PUT [ FILE (SYSPRINT)] LIST (druckbereich);
```

Einzelheiten werden im folgenden besprochen.

10.2. Allgemeine Betrachtungen zur reihenweisen Ein- und Ausgabe

Merkmale:

- Daten der reihenweisen E/A werden als "endlose" Kette, als ein kontinuierlicher Strom
  unbestimmter Länge von Zeichen mit dem Attribut CHARACTER behandelt
- Datenumwandlungen sind bei Eingabe und Ausgabe möglich;
  Datenkonvertierungen sind eine wichtige Eigenschaft der reihenweisen Datenverarbeitung
- es werden nur Sätze fester Länge im ungeblockten Format behandelt
- Datenkonvertierungen können durch die E/A-Routinen veranlasst werden
- nur sequentielle Verarbeitung ist möglich
- für die Eingabe gilt die GET- Anweisung
- für die Ausgabe gilt die PUT- Anweisung

10.2.1. GET- Anweisung

Durch die GET-Anweisung können Teilstücke beliebiger Länge für die Verarbeitung aus dem Eingabepuffer linksbündig abgenommen werden.

Allgemeine Form der reihenweisen Eingabe:

```
GET [{ FILE ( dateiname ) | STRING ( zeichenkettenvariable ) }]
  { EDIT ( datenliste ) ( formatliste ) | LIST ( datenliste ) | DATA ( datenliste )} [COPY];
```

Beispiel:

```
GET EDIT (EING_BEREICH) (A(100)) COPY;
```

Von der Standarddatei SYSIN werden die nächsten 100 Zeichen in den Eingabebereich EING_BEREICH gelesen und sofort über die Standardausgabe SYSPRINT kopiert.

10.2.2. PUT- Anweisung

Durch die PUT-Anweisung wird in die Ausgabepuffer der Ausgabedatei eine "endlose" Kette von Zeichen zusammengestellt.Die Länge der Kette wird durch die Formatliste der PUT- Anweisung bestimmt

Allgemeine Form der reihenweisen Ausgabe:

```
PUT [{ FILE ( dateiname ) | STRING ( zeichenkettenvariable ) }]
      { EDIT ( datenliste ) ( formatliste ) | LIST ( datenliste ) | DATA ( datenliste )}
```

```
[SKIP [( skalarer ausdruck )] ]  [ PAGE ]  [ LINE (skalarer ausdruck )];
```

Beispiel:

```
PUT EDIT (DRUCK_BEREICH) (A(100)) PAGE SKIP;
```

In die Standarddatei SYSPRINT werden die nächsten 100 Zeichen zum Druck bereitgestellt. Der Druck beginnt auf einer neuen Seite mit einem einzeiligen Vorschub.
Im Punkt 10.5.2. werden die Optionen bzw. Formatelemente PAGE, SKIP und LINE ausführlich besprochen.

10.3. Die LIST- gesteuerte E/A

Die LIST- gesteuerte E/A ist die einfachste Form der reihenweisen E/A. Man hat kaum Gestaltungsmöglichkeiten, vor allem beim Druck ist es nur bedingt möglich, bei der Druckbildgestaltung auf individuelle Wünsche einzugehen. Sie dient vornehmlich für wissenschaftlich-technische Zwecke, für den kommerziellen Einsatz ist sehr nur sehr bedingt einsetzbar.

Die LIST- gesteuerte Übertragung betrachtet die Eingabedatei als eine Reihe von Zeichen, wobei die zu jeder Eingabevariablen gehörende Folge von Zeichen von der nächstfolgen-den durch einen Zwischenraum oder durch ein Komma getrennt sind.
Bei der LIST- gesteuerten Ausgabe wird die Datenmenge in der Ausgabedatei in der Form so angeordnet, dass sie wieder über die LIST- gesteuerte Eingabe mit den notwendigen Trennungen einlesen werden kann. Das Ausgabefeld wird links durch die Ausgaberoutine um drei Byte erweitert.

Beispiel:

```
DCL  W_1 FIXED (5),
       W_2 FIXED (3),
       W_3 CHAR (6);
GET LIST ( W_1, W_2, K_1);        /*  MOEGLICHER DATENSTROM: 12345, 123, 'BERLIN' */
```

Es werden aus dem Eingabepuffer drei Ketten gelesen, zwei werden in das Festkomma-format konvertiert und den entsprechenden Adressen W_1, W_2 zugewiesen, die letzten sechs Zeichen werden K_1 zugewiesen.

Beispiel:

```
DCL  MATRIX (4,7) FIXED (5);

DO I = 1 TO 4;
   DO K = 1 TO 7;
GET LIST ( MATRIX (I,K) );   /* ES WERDEN 28 ZEICHENKETTEN GELESEN UND
KONVERTIERT */
    END;
END;
```

besser

```
GET LIST ( (( MATRIX (I,K) DO K = 1 TO 7) DO I = 1 TO 4) );
```

Die Klammerung entspricht den END- Klauseln.

noch besser

```
GET LIST (MATRIX);
```

Es werden 28 Zeichenketten eingelesen und entsprechend den angegebenen Attributen konvertiert.

Beispiel:

```
DCL WERT FIXED(2);
WERT = 78;
PUT LIST (WERT);                              /* AUSGABE: 'bbb78' */
```

Beispiel:

```
DCL  DRUCKBER CHAR (120);
folge von Anweisungen;
OPEN FILE (DRUCKLIST) PRINT LINESIZE(120) PAGESIZE (60);
PUT FILE (DRUCKLIST) LIST (DRUCKBER) PAGE SKIP(3);
```

Die Kette DRUCKBER wird auf einer neuen Seite mit 3 Zeilen Vorschub vollständig ausgegeben.

10.4. Die DATA- gesteuerte E/A

Die DATA- gesteuerte E/A ist ausschließlich nur für Testzwecke geeignet. Die Daten werden im Ein- und Ausgabestrom als vollständige Ergibtanweisungen abgebildet. Der Abschluss des Lesens wird durch ein Semikolon angezeigt.

Beispiel 1:

```
Eingabestrom für Testzwecke:  W1=+2.1E2   W3='1001'   W2=100B;

DCL W1 FLOAT,
      W2 CHAR(4),
      W3 BIN FIXED;
GET DATA (W1, W2, W3);  /* IN DER REIHENFOLGE DER DATENLISTE WERDEN DIE WERTE
*/
                                /* ZUGEORDNET */
```

Beispiel 2:

```
DCL KETTE CHAR (3),
      BER BIT(1),
      WERT FIXED;

KETTE = 'TEST';
BER(1), BER(2) = 0B;
BER(3) = 1B;
WERT = 15;
PUT DATA (KETTE, BER, WERT);
```

```
Ausgabestrom: KETTE='TES'   BER(1)='0'B   BER(2)='0'B   BER(3)='1'B   WERT=   15;
```

10.5. Die EDIT- gesteuerte E/A

Die EDIT- gesteuerte E/A ist ausschließlich für kommerzielle Einsätze gedacht. Durch Angabe geeigneter Formatelemente kann jede Form der Eingabe als auch der Ausgabe, vor allem im Bereich der Drucklistengestaltung programmiert werden.

Allgemeine Form:

```
{ GET | PUT } [{ FILE ( dateiname ) }]  EDIT ( datenliste ) ( formatliste ) ;
```

Es werden zwei Formen der Formatelemente benutzt:

```
< datenformatelemente >          ::= F | A | B | E | C | P
< steuerungsformatelemente > ::= X | PAGE | SKIP |  LINE | COL
```

10.5.1. Festkommaformatelement F(w,p,q)

Das Festkommaformatelement F konvertiert Daten in die sog. gepackte Form. Sie ist für die Dezimalarithmetik die optimale Basis.

Drei Formen sind möglich:

1. F(w) w - Gesamtanzahl der zu konvertierenden Zeichen
   w -> [b...b][{+|-}]dezimalzahl[b...b]
2. F(w, p) p - Anzahl der Zeichen nach dem Dezimalpunkt
3. F(w, p, q) q - Bei der Eingabe wird der Wert mit 10 hoch q multipliziert,
   konvertiert und zugewiesen
   Bei der Ausgabe wird der interne Wert 10 hoch q fach ausgegeben

Beispiel zu 1:

Eingabestrom: 12345bbb b Blank, Leerzeichen

```
DCL WERT FIXED (5);
GET EDIT (WERT) (F(5));    /* LIES DIE NAECHSTEN 5 ZEICHEN UND KONVERTIERE */
                           /* IN FESTKOMMA   */
PUT EDIT (WERT) F(8));     /* STELLE 6 DRUCKWERKSTELLEN ZU VERFUEGUNG UND */
                           /* KONVERTIERE IN CHARACTER-FORMAT */
```

Ausgabe: bbb12345

Beispiel zu 2:

```
DCL WERT FIXED (5,2);

GET EDIT (WERT) (F(5,2)); /* LIES DIE NAECHSTEN 5 ZEICHEN UND KONVERTIERE */
                          /* IN FESTKOMMA MIT 2 ZEICHEN NACH DEZIMALPUNKT  */
PUT EDIT (WERT) F(8,2));  /* DER ZU DRUCKENDE DEZ.PUNKT MUSS MIT  */
```

```
                                    /* BERUECKSICHTIGT WERDEN */
```

Ausgabe: bb123.45

Beispiel zu 3:

```
DCL WERT FIXED (5);

GET EDIT (WERT) (F(5));    /* LIES DIE NAECHSTEN 5 ZEICHEN UND KONVERTIERE */
                           /* IN FESTKOMMA   */

PUT EDIT (WERT) F(9,0,3));
```

Ausgabe: b12345000

Fallunterscheidungen F(w)

Eingabe	Zeichen der Eingabedatei	Formatelement	Ziel: FIXED(3), interne Darstellung mit gedachten Dezimalpunkt
	123	F(3)	123C
	-12	F(3)	012D
	-1b	F(3)	001D
	1bb	F(3)	001C
	bbb	F(3)	000C
	b1b	F(3)	001C
	-b1	F(3)	CONVERSION
	12345	F(3)	123C
	123.45	F(3)	123C
	b12.34	F(6)	12.34C
Ausgabe	**Interner Wert**	**Formatelement**	**Zeichen der Ausgabedatei**
	123	F(5)	bb123
	-12	F(5)	bb-12
	-12	F(2)	SIZE
	-12	F(6,0,3)	-12000

Fallunterscheidungen F(w,p)

Eingabe	Zeichen der Eingabedatei	Formatelement	Ziel: FIXED(5,2), interne Darstellung mit gedachten Dezimalpunkt
	12345	F(5,2)	123.45C
	123bb	F(5,2)	001.23C
	12345	F(3,1)	12.3C
	12345	F(5,1)	1234.5C
	12.34	F(5,2)	012.34C
Ausgabe	**Interner Wert**	**Formatelement**	**Zeichen der Ausgabedatei**
	12.345	F(7,2)	b12.345
	-0.001	F(7,2)	bb0.001
	-12.567	F(4)	b-13
	-12.345	F(9,2)	bb-12.345b

10.5.2. Zeichenkettenformatelement A [(w)]

Das Zeichenkettenformatelement überträgt Zeichenketten der Länge w. Wird keine Feldbreite w in der Ausgabe angegeben, wird die Zeichenkette der gesamten Länge übertragen. Eine Umwandlung findet nicht statt.

Beispiel:

```
DCL KETTE CHAR (6) INIT ('BERLIN');
PUT EDIT (KETTE) (A(3));                /* 'BER'  */
PUT EDIT (KETTE) (A(3));                /*  'LIN'   */

oder
PUT EDIT (KETTE) (A);                  /*  'BERLIN'  */

oder
PUT EDIT (KETTE) (A(10));              /*  'BERLINbbbb' */
```

Die letzte Anwendung ist vor allem bei VARYING-Ketten sinnvoll, da die aktuelle Kettenlänge u.U. unbekannt ist.

Beispiel:

```
DCL  KETTE CHAR(100) VAR;

Verarbeitung der Kette;
PUT EDIT (KETTE) (A);            /*  AUSGABE DER KETTE MIT DER AKTUELLEN LAENGE  */
```

10.5.3. Bitkettenformatelement B [(w)]

Das Bitkettenformatelement B [(w)] überträgt Bitketten der Länge w. Wird keine Feldbreite w in der Ausgabe angegeben, wird die Bitkette der gesamten Länge übertragen. Eine Umwandlung findet nicht statt. Enthält das w-Feld Zeichen ungleich '0' und '1', wird die Bedingung CONVERSION gesetzt.

Beispiel:

```
DCL  BIT_KET BIT(4),
       ZEICH_KET CHAR (4);
    BIT_KET = '0011'B;
    ZEICH_KET = 'TEST';

   PUT EDIT ('ZEICHENKETTE = ', ZEICH_KET, 'BITKETTE =  ', BIT_KET) (A, A(6), A, B);
```

Druckzeile: ZEICHENKETTE = TEST BITKETTE = 0011

10.5.4. Gleitkommaformatelement E (w, p [,q])

Das Gleitkommaformatelement E (w, p [, q]) konvertiert alle möglichen Daten bei der Eingabe intern in Gleitkommazahlen und bei der Ausgabe in Gleitkommadarstellung.

Das Formatelement hat zwei Formen:

1. E(w,p) Enthält das Dateneingabefeld w keinen Dezimalpunkt, dann wird durch p die Anzahl der Stellen nach dem Dezimalpunkt angegeben.
Bei der Ausgabe belegt der Exponent vier Stellen: E ± nn, p gibt die Stellen
nach dem Dezimalpunkt an. Vor dem Dezimalpunkt wird immer eine Ziffer ausgegeben.

2. E(w,p,q) Bei der Eingabe bleibt q unberücksichtigt. Bei der Ausgabe wird durch q angegeben, wieviel Ziffern ingesamt für die Mantisse ausgegeben werden
sollen. Wird q nicht angegeben, so gilt q = p + 1
Das Ausgabefeld stellt sich wie folgt dar:
[-] q-p ziffern . p-ziffern E ± exponent

Beispiel:

```
DCL  (W_1, W_2, W_3) FLOAT;
W_1 = 9.81;
W_2 = -981;
W_3 = 98;
PUT EDIT ( 'W_1 =', W_1,'  W_2 =', W_2, '  W_3 =', W_3) ( A(6), E(10,2, A(6), E (10,2), A(6),
E(10,0,2));

Ausgabefeld: W_1 =bbb9.81E+00bbW_2 =bb-9.81E+02  W_3 =bbbb98.E+00
```

Fallunterscheidungen E(w,p,q)

Ausgabe	Interner Wert	Formatelement	Zeichen der Ausgabedatei
	-123	E(8,0)	bb-1E+00
	-123	E(10,2)	b-1.23E+02
	-123	E(12,4)	b-1.2300E+02
	0	E(8,0)	bbb0E+00
	0	E(10,2)	bb0.00E+00
	0	E(12,4)	bb0.0000E+00
	1.23456789	E(14,6)	bb1.234567E+00
	1.23456789	E(14,6,9)	123.456789E-02
	1.23456789	E(14,6,6)	bbb.123456+01

10.5.5. Komplexzahlformatelement C

Das Komplexzahlformatelement C (reelles formatelement [,reelles formatelement]) setzt sich aus dem Real- und Imaginärteil zusammen. Beide reelle Formatelemente dürfen nur die Formate F oder E beinhalten.

10.5.6. Abbildungsformatelement P

Das Abbildungsformatelement PICTURE beschreibt jede Stelle des Mediums, auf das sich der Ein- oder Ausgabevorgang bezieht. Mit dieser Methode ist es möglich, jede Form von Ausgabeschablonen darzustellen. Mit dieser Methode sind nur Festkommadaten und Zeichenketten darstellbar.

Allgemeine Form:

```
< abbildungsformatelement > ::= { P' numerische abbildungsspezifikation' |
                                  P' zeichenketten abbildungsspezifikation' }
```

10.5.6.1. Numerische Abbildungsformatelemente

Jedes Zeichen des externen Mediums wird unmittelbar durch Abbildungsformatelemente dargestellt.

Allgemeine Form:

```
< numerisches abbildungsformatelement > ::= 9 | V | Z | * | Y | B | ' | / | . | S | + | - | T | I | R
```

- Abbildungsformatelement 9 für Dezimalziffer

Beispiel:

```
DCL  W_1 FIXED (5) INIT (123);
PUT EDIT ( W_1) ( P'99999');          /* AUSGABE: 000123 */
```

- Abbildungsformatelement V für gedachten Dezimalpunkt
  V bewirkt nicht den Druck eines Dezimalpunktes.

Beispiel:

```
DCL  W_1 FIXED (5,2) INIT (123.45);
PUT EDIT ( W_1) ( P'999V99');         /* AUSGABE: 12345 */
```

- Abbildungsformatelement Z für Dezimalziffer mit Vornullenunterdrückung

Beispiel:

```
DCL  W_1 FIXED (5) INIT (123);
PUT EDIT ( W_1) ( P'ZZZZ9');          /* AUSGABE: bbb123 */
```

- Abbildungsformatelement * für Dezimalziffer
  Führende Nullen werden durch * ersetzt

Beispiel:

```
DCL  W_1 FIXED (5) INIT (123);
PUT EDIT ( W_1) ( P'****9');          /* AUSGABE: ***123 */
```

- Abbildungsformatelement Y für Dezimalziffer

Alle Nullen werden durch Leerzeichen ersetzt

Beispiel:

```
DCL  W_1 FIXED (5) INIT (1203);
PUT EDIT ( W_1) ( P'YYYYY');          /* AUSGABE: b12b3 */
```

- Aufbereitungszeichen B

  Es wird ein Leerzeichen eingesetzt

Beispiel:

```
DCL  W_1 FIXED (5) INIT (1203);
PUT EDIT ( W_1) ( P'9B999');          /* AUSGABE: 1b203 */
```

- Aufbereitungszeichen ,

  Es wird ein Komma eingesetzt

Beispiel:

```
DCL  W_1 FIXED (5,2) INIT (123.45);
PUT EDIT ( W_1) ( P'ZZ9V,99');        /* AUSGABE: 123,45 */
```

- Aufbereitungszeichen .

  Es wird ein Punkt eingesetzt

Beispiel:

```
DCL  W_1 FIXED (7,2) INIT (12345.67);
PUT EDIT ( W_1) ( P'Z.999V,99');      /* AUSGABE: 12.345,67 */
```

- Aufbereitungszeichen /

  Es wird ein Schrägstrich eingesetzt

Beispiel:

```
DCL  W_1 FIXED (7,2) INIT (12345.67);
PUT EDIT ( W_1) ( P'Z/999V,99');      /* AUSGABE: 12/345,67 */
```

Punkt und Komma werden nur dann erzeugt, wenn rechts neben Komma oder Punkt Datenelemente stehen.

- Vorzeichenspezifikationen S + -

Die Vorzeichen können statisch oder driftend angegeben werden.

S ist das auszugebende Datenelement >= 0, dann wird + gedruckt
ist das auszugebende Datenelement < 0, dann wird - gedruckt

\+ ist das auszugebende Datenelement >= 0, dann wird + gedruckt
ist das auszugebende Datenelement < 0, dann wird blank gedruckt

\- ist das auszugebende Datenelement >= 0, dann wird blank gedruckt
ist das auszugebende Datenelement < 0, dann wird - gedruckt

Beispiel:

```
DCL  W_1 FIXED (7,2) INIT (12345.67);
PUT EDIT ( W_1) ( P'BZ.999V,99');      /* AUSGABE: +12.345,67 */
PUT EDIT ( W_1) ( P'Z.999V,99S');      /* AUSGABE: 12.345,67+ */
PUT EDIT ( W_1) ( P'Z.999V,99-');      /* AUSGABE: 12.345,67b */
PUT EDIT ( W_1) ( P'Z.999V,99+');      /* AUSGABE: 12.345,67+ */

DCL  W_2 FIXED (5) INIT (-12);
PUT EDIT ( W_2) ( P'SZZZZ9');          /* AUSGABE: -bbb12  VORZEICHEN STATISCH*/
```

Bei Angabe einer Driftkette, SSS...S, +++...+ oder ---...- driftet das Vorzeichen an die erste signifikante Ziffer heran. Das erste Driftzeichen legt das zu abzubildende Vorzeichen fest, alle anderen Driftzeichen sind beliebige Ziffernstellen. Driftketten dürfen nicht unterein-ander gemischt werden, allerdings können sie mit , . / V B vermischt werden.

Beispiel:

```
DCL  W_1 FIXED (7,2) INIT (-12345.67),
       W_2 FIXED (5,2) INIT (-1.23);
PUT EDIT ( W_1) ( P'+++.+++V,99');    /* AUSGABE: +12.345,67 */
PUT EDIT ( W_2) ( P'----V,99');       /* AUSGABE: bbb-1,23  VORZEICHEN DRIFTEND*/
```

10.5.6.2. Zeichenketten - Abbildungsformatelemente

Jedes Zeichen des externen Mediums wird unmittelbar durch Abbildungsformatelemente dargestellt.

Allgemeine Form:

```
< zeichenketten abbildungsformatelement > ::= 9 | A | X |
```

- Abbildungsformatelement 9 für Dezimalziffer oder Leerzeichen

Beispiel:

```
DCL  KET CHAR (5) INIT ('12345');
PUT EDIT ( KET) ( P'99999');          /* AUSGABE: 12345 */
```

- Abbildungsformatelement A für Alphabetzeichen oder Leerzeichen

Beispiel:

```
DCL  KET CHAR (10) INIT ('ABCDE12345');
PUT EDIT ( KET) ( P'AAAAA99999');    /* AUSGABE: ABCDE12345 */
```

- Abbildungsformatelement X für beliebige Zeichen

Beispiel:

```
DCL  KET CHAR (10) INIT ('ABCDE12345');
PUT EDIT ( KET) ( P'AAAAAXXX99999');                    /* AUSGABE:
ABCDEbbb12345 */
```

10.5.6.3. Abschließendes Beispiel

Beispiel:

```
DCL  KET CHAR (8) INIT ('DRUCK: '),
       GLEIT FLOAT INIT (1.23),
       FEST FIXED INIT (-6.50);

PUT EDIT (KET, GLEIT, FEST)
            (A, P'9V.99', P'SS9V,9');          /* DRUCK:b1.23-b6,5 */
PUT EDIT (KET, GLEIT, FEST)
            (A, P'999V.9', P'ZZZV.99');        /* DRUCK:b0001.2bbb6.50 */
PUT EDIT (KET, GLEIT, FEST)
            (A, P'--9V.99', P'+++V.99');       /* DRUCK:b0001.2bbb6.50 */
PUT EDIT (KET, GLEIT, FEST)
            (P‘(10)A‘, (2) P’SSS9V.999‘);                  /* DRUCK:bbbbbb+1.230bb-6.500 */
PUT EDIT (KET, GLEIT, FEST)
            (P‘(5)A‘, F(4,2),  P’SSS9V/9 ‘);   /* DRUCK1.23bb-6/5 */
```

10.6. Steuerungsformatelemente

Sie dienen dazu, die Ein - und Ausgabemöglichkeiten den Praxisforderungen noch besser entgegenzukommen.

10.6.1. Zwischenraumformatelement X(w)

Der Ausdruck w gibt bei der Eingabe an, wie viele w Stellen überlesen werden sollen. Bei der Ausgabe werden w Leerstellen in der Druckzeile eingefügt.

Beispiel:

```
DCL  (W_1, W_2) FIXED (5);
GET EDIT ( W_1, W_2) (X(4), F(5), X(1), F(5));
PUT EDIT ( W_1, W_2) (X(5), F(5), (X(3), F(5));
```

Eingabedatenstrom: TEST45678;12345
Druckzeile: bbbbb45678bbb12345

10.6.2. Druckformatformatelemente

Die Druckformatelemente dienen der optimalen Druckersteuerung zur Listenausgabe.

10.6.2.1. Zeilenvorschubformatelement SKIP

Das Vorschubformatelement SKIP [(w)] gibt an, wie viele w - Vorschübe (Wagenrücklauf - Zeilenvorschub) eingefügt werden sollen.

Ist der Ausdruck w > 0, so rückt der Drucker w-Zeilen vorwärts, er fügt w-1 Leerzeilen ein
Ist der Ausdruck w <= 0, so wird kein Vorschub eingefügt; der Drucker beginnt wieder in der gleichen Zeile ab Spalte 1 zu drucken.

10.6.2.2. Blattwechselformatelemente PAGE

Das Blattwechselformatelemente PAGE führt beim Druck einen sofortigen Blattwechsel aus.

10.6.2.3. Zeilendruckformatelemente LINE (w)

Das Zeilendruckformatelemente LINE (w) führt den Druck auf der w. Zeile aus. Ist w kleiner als die aktuelle Zeile, wird ein Blattwechsel impliziert.

10.6.2.4. Spaltendruckformatelemente COLUMN (w)

Das Spaltendruckformatelemente COLUMN (w) führt den Druck in der w. Spalte aus. Ist w kleiner als die aktuelle Spalte, wird ein Zeilenvorschub impliziert.

Beispiel:

```
PUT EDIT ( 'PREISLISTE', (10) '_', 'SUMME: _______,__ EUR')
          (PAGE, SKIP (3), COLUMN(7), A(10), SKIP (0), X(6), A(10),  LINE (50), X(6), A );
```

Druckseite:

```
        Spalte
          |
Zeile  1

              PREISLISTE


Zeile 50      SUMME: ______,__EUR
```

10.6.3. Remote Formatelemente R

Das Remote Formatelemente R findet vor allem Einsatz bei sich wiederholenden Formatlisten.

Allgemeine Form:

```
R(marke)
marke: FORMAT (f1, ...,fn);
```

Beispiel:

```
PUT EDIT (DATLIST_1) (SKIP, COLUMN(20), F(7,2));
  folge von anweisungen;
```

```
PUT EDIT (DATLIST_2) (SKIP, COLUMN(20), F(7,2));

besser;

PUT EDIT (DATLIST_1) (R(FORMATLISTE));
  folge von anweisungen;
PUT EDIT (DATLIST_2) (R(FORMATLISTE));
  folge von anweisungen;
FORMATLISTE: FORMAT (SKIP, COLUMN(20), F(7,2));
  folge von anweisungen;
PUT EDIT (DATLIST_3, DATLIST_4) (R(FORMATLISTE), PAGE, X(3), A);
```

10.7. Wiederholung von Formatelementen

Die Wiederholungsspezifikation wird in Klammern angegeben. Die Spezifikation kann ein Ausdruck sein.

Beispiel:

```
PUT EDIT (BEREICH) (SKIP, COLUMN(20),  (3) F(7,2), (5) (X(2), F(4)));
PUT EDIT (BEREICH) (SKIP, COLUMN(20),  (I + 1) ( X(2), F(4)));
```

10.8. Wechselspiel zwischen Daten- und Formatlisten

Die Beziehung zwischen Daten- und Formatliste ist vor allem bei der Ausgabe von Bereichen und Strukturen sehr wichtig. Es gilt der Grundsatz: jedes Element, was ein- oder
auszugeben ist, muss in der Formatliste durch ein Datenformatelement spezifiziert sein.

Beispiel:

```
PUT EDIT (          W1,      W2 ,     W3,          W4,      W5
)
        ( PAGE, LINE(15), X(5), F(7,2), X(3), F(7,2), X(6), F(5,2), SKIP(3), X(5), F(7,2), X(6),
F(5,2));
```

Folgende Sonderfälle sind möglich:
1. Datenliste enthält mehr skalare Elemente als die Formatliste Datenformatelemente
2. Datenliste enthält weniger skalare Elemente als die Formatliste Datenformatelemente
3. Datenliste enthält einen Bereich
4. Datenliste enthält eine Struktur

Zu 1.: Ist die Formatliste erschöpft, wird sie von vorn wieder abgearbeitet

Beispiel:

```
PUT EDIT (W1, W2, W3) (SKIP(2), COLUMN(10), F(7,2));
```

Alle Daten werden untereinander ab Spalte 10 gedruckt.

Zu 2.: Ist die Datenliste erschöpft, wird der Rest der Formatliste nicht berücksichtigt. Die Datenliste ist primär.

Beispiel:

```
PUT EDIT (W1, W2) (SKIP(2), COLUMN(10), F(7,2), x(5), F(5,2), SKIP(2));
```

Das Steuerformatelement SKIP(2) wird nicht mehr berücksichtigt.

Zu.3: Die Formatliste wird solange bzw. sooft abgearbeitet, solange Datenelemente in der Datenliste vorhanden sind.

Beispiel 1:

```
DCL BER(5) FIXED (2);
folge von anweisungen;
PUT EDIT (BER) (F(4));                    Druckbild:  bbZZbbZZbbZZbbZZbbZZ
PUT EDIT (BER) (SKIP, F(4));                                  Druckbild:  bbZZ
                                                                  bbZZ
                                                                  bbZZ
                                                                  bbZZ
```

Beispiel 2:

```
DCL BER(2,2,) FIXED (5,2);
DO J = 1 TO 2;
PUT EDIT ( 'ZEILE:', J, (BER (J, K) DO K = 1 TO 2))
        ( SKIP, A(7), F(2), SKIP, X(6), (2)(X(3), F(7)));
END;
                                        Druckbild:   ZEILE:bb1
                                                          ZZZZZZbbbZZZZZZ
                                                     ZEILE:bb2
                                                          ZZZZZZbbbZZZZZZ
```

Beispiel 2a: Beide DO-Schleifen in der Datenliste

```
DCL BER(2,2,) FIXED (5,2);
PUT EDIT ( 'ZEILE:', J, ((BER (J, K) DO K = 1 TO 2) DO J = 1 TO 2))
        ( SKIP, A(7), F(2), SKIP, X(6), (2)(X(3), F(7)));
```

Zu.4: Die Formatliste wird solange bzw. sooft abgearbeitet, solange Datenelemente in der Datenliste vorhanden sind.

Beispiel :

```
DCL 1 STR,
       2 KET CHAR(3),
       2 BER (4) FIXED (2),
       2 U_STR(5),
         3 W1 FIXED (3,2),
         3 U_BER(3) BIT (1);
PUT EDIT (STR) ( A(5), SKIP(2), (4) (F(3), X(2)), SKIP(2),
                                (5) (F(6,2), X(2), (3) ( B, X(2)) ) );
```

10.9. Die STRING – Option

Eine gute Möglichkeit, Daten im Hauptspeicher zu transportieren, ist mit der STRING – Option alternativ zur FILE – Option möglich.

Allgemeine Form:

```
< STRING-option > ::=  { GET | PUT } STRING (name einer zeichenkette)
                       { EDIT ( datenliste ) ( formatliste ) | LIST ( datenliste ) | DATA ( datenliste )};
```

Durch GET mit der STRING-Option werden intern als Zeichenketten gespeicherte Daten den Variablen nach den Regeln der externen Eingabe zugeordnet, die in Datenliste erscheinen. PUT mit der STRING-Option richtet sich nach Regeln der externen Ausgabe. Nur werden die Daten nicht dem externen Datenträger zugewiesen, sondern sie werden als Zeichenkette einer Zeichenkettenvariablen zugeordnet.

Beispiel:

Ein sinnvoller Einsatz ergibt sich, wenn Daten zur weiteren Bearbeitung identifiziert werden sollen; im folgenden Fall zwischen Kundendaten und Rechnungsdaten. Später werden wesentlich elegantere Möglichkeiten besprochen. Ebenso könnten die unbedingten Sprünge über DO-Gruppen effizienter programmiert werden.

```
LIES: GET FILE ( EINGANG ) EDIT ( KENN, ZWISCH_SPEICHER) ( A(1), A(79));
folge von anweisungen – datenprüfungen;

    IF KENN = 'K' THEN GOTO KUND;
                  ELSE GOTO RECH;
KUND: GET STRING ( ZWISCH_SPEICHER ) EDIT (KUND-STRUKTUR) ( folge von
formatelementen);
          /* DIE IN ZWISCH_SPEICHER GESPEICHERTEN DATEN WERDEN DER  */
          /* KUNDENSTRUKTUR  ZUGEORDNET */
    folge von anweisungen;
    GOTO LIES;
RECH: GET STRING ( ZWISCH_SPEICHER ) EDIT (RECH-STRUKTUR) ( folge von
formatelementen);
          /* DIE IN ZWISCH_SPEICHER GESPEICHERTEN DATEN WERDEN DER  */
          /* RECHNUNGSSTRUKTUR ZUGEORDNET */
    folge von anweisungen;
    GOTO LIES;
```

11. Speicherklassenattribute

Die Zuordnung der Speicherplätze für alle variablen Größen eines Programms kann auf unterschiedliche Weise erfolgen und ist abhängig vom Speicherklassenattribut des betreffenden Datenelements. Im Normalfall hat eine Größe standardmäßig das Speicher-klassenattribut AUTOMATIC; sie erhält ihren Speicherplatz automatisch mit Beginn der Abarbeitung und mit Beendigung des Programms wird der entsprechende Speicherplatz wieder freigegeben. Der Anwender hat allerdings auch die Möglichkeit, diese Speicher-plätze mit entsprechenden Speicherklassenattributen selbst zu verwalten.

Allgemeine Form:

< speicherklassenattribut > ::= AUTOMATIC | STATIC | CONTROLLED | BASED (zeigervariable)

11.1. Speicherklasse AUTOMATIC

Die Zuordnung von Speicherplatz geschieht bei Aktivierung des Blockes durch den Prolog; das Lösen der Zuordnung von Speicherplatz wird durch das Beenden des Blockes, durch den Epilog, ausgelöst. Dimensionen bei Bereichen und Kettenlängen können Ausdrücke sein, sie werden in jedem Prolog neu berechnet (siehe 6.4.2.). Die Gültigkeit dieser AUTOMATIC - Variablen erstreckt sich über den aktuellen Block als auch auf alle internen Blöcke. Bei Unterprogrammen, die sehr oft aufgerufen werden, ist es sicher nicht ratsam, in diesem Unterprogramm AUTOMATIC - Variable zu vereinbaren, da die Verwaltung derartiger Variablen mit einem großen organisatorischen Aufwand verbunden sind.

Beispiel:

```
HAUPT: PROC OTIONS (MAIN);
    DCL WERT FIXED;              /* AUTOMATIC */
    folge von anweisungen;
    CALL UP1;
    folge von anweisungen;
UP1: PROC;
      DCL WERT2 FIXED;          /* AUTOMATIC, WIRD MIT JEDEM AUFRUF NEU
ZUGEORDNET */
      folge von anweisungen;
END UP1;
END HAUPT;
```

11.2. Speicherklasse STATIC

Die Zuordnung von Speicherplatz geschieht einmalig vor der Aktivierung des Programms durch den Prolog; das Lösen der Zuordnung von Speicherplatz wird durch das Beenden des Programms, durch den Epilog, ausgelöst. Dimensionen bei Bereichen und Ketten-längen können Ausdrücke sein, sie werden einmalig berechnet.

Die Gültigkeit dieser STATIC - Variablen erstreckt sich über das gesamte Programm. Bei Unterprogrammen, die sehr oft aufgerufen werden, ist es sehr ratsam, in diesem Unterprogramm STATIC - Variable zu vereinbaren.

Beispiel:

```
HAUPT: PROC OTIONS (MAIN);
   DCL WERT FIXED;                /* AUTOMATIC */
   folge von anweisungen;
   CALL UP1;
   folge von anweisungen;
UP1: PROC;
      DCL WERT2 FIXED STATIC,     /* SPEICHPLATZZUORDNUNG EINMALIG IM PROLOG  */
            IAUFRUF INIT (0) STATIC;
      IAUFRUF = IAUFRUF + 1;  /* AUFRUFZAEHLER, MIT CONTROLLED -  IAUFRUF = 1 */
      folge von anweisungen;
END UP1;
END HAUPT;
```

11.3. Speicherklasse CONTROLLED

Eine gute Möglichkeit, Speicherplatz an jeder beliebigen Stelle des Programms anzufordern und wieder freizugeben, ist mit der Speicherklasse CONTOLLED bzw. CTL zu erreichen.

11.3.1. Die Anweisungen ALLOCATE und FREE

Die Zuordnung von Speicherplatz wird durch die ALLOCATE - Anweisung erreicht; die Freigabe dieses Speicherplatzes geschieht durch die FREE - Anweisung. Bei Mehrfach-anwendung der ALLOCATE - Anweisung werden immer neue Generationen von Speicherplatz zugeordnet; die 1. bis zur n-1. Generation werden in den Kellerstapel "gestapelt". Die Einsatzmöglichkeiten werden in den folgenden Beispielen gezeigt:

Beispiel 1- einfache Anwendung:

```
DCL MAT (10,20) CHAR (50) CONTROLLED;  /* NOCH KEIN SPEICHER ZUGEORDNET*/
ALLOCATE MAT;       /* ZUORDNUNG VON 10.000 BYTE */
folge von anweisungen;
FREE MAT;    /* FREIGABE DER ZULETZT ZUGEORDNETEN 10.000 BYTE */
```

Die ALLOCATE - Anweisung kann folgende Attribute aufnehmen:

- Dimensionsattrubut
- Kettenattribut
- AREA Attribut
- INITIAL - Attribut

Beispiel 2 - Sternschreibweise:

```
DCL MAT (*,*) CHAR (50) CONTROLLED;     /* DIM-GRENZEN ERST ZUR AUSFÜHRUNG */
folge von anweisungen;
ALLOCATE MAT (DIM1,DIM2) CHAR (10);
folge von anweisungen;
ALLOCATE MAT (7,5) INIT ((35) (1) ' ');
```

Beispiel 3 - Sternschreibweise:

```
DCL MAT (*,*) CHAR (*) CONTROLLED;     /* DIM-GRENZEN UND KETTENLAENGE  */
                                       /* ERST ZUR AUSFÜHRUNG */
folge von anweisungen;
ALLOCATE MAT (10,20,) CHAR (50);
```

Mit Hilfe von DO-Anweisungen kann der Speicherplatz dynamisch verwaltet werden.

Beispiel 4 - Kellerstapel */:

```
DCL MAT (10,20) CHAR (50) CONTROLLED;  /* NOCH KEIN SPEICHER ZUGEORDNET */
      DO I = 1 TO N;
      ALLOCATE MAT;       /* DYNAMISCHE ZUORDNUNG VON JE 10.000 BYTE */
      folge von anweisungen;    /* NUR DIE LETZTE GENEATION VON MAT IST VERFUEGBAR */
      END;

FREE MAT;    /* FREIGABE DER LETZTEN GENERATION VON 10.000 BYTE */
```

Abschließendes Beispiel:

```
HAUPT: PROC OPTIONS (MAIN);
   DCL AKET CHAR(5) CTL,
         BKET (*) BIT(1) CTL;
   folge von anweisungen;
   ALLOCATE AKET, BKET (3) INIT ('1'B, '0'B, '1'B);     /* 1. GENERATION VON AKET UND BKET
   */
   AKET = '1.GEN:';
      ALLOCATE AKET INIT ('2.GEN:');         /* 2. GENERATION VON AKET   */
      PUT LIST (AKET, BKET) SKIP;            /* DRUCKBILD : 2.GEN:,1B,0B,1B   */
      FREE AKET;                             /* FREIGABE 2. GENERATION VON AKET   */
      PUT LIST (AKET, BKET) SKIP;            /* DRUCKBILD : 1.GEN:,1B,0B,1B   */
   FREE AKET, BKET;                    /* FREIGABE 1. GENERATION VON AKET UND BKET */
END HAUPT;
```

11.3.2. Die Standardfunktion ALLOCATION (Bezeichner)

Mit Hilfe dieser Funktion kann bestimmt werden, ob zum Zeitpunkt des Funktionsaufrufes Speicherplatz für Bezeichner durch CONTROLLED zugeordnet ist oder nicht. Der Funktionswert gibt die Anzahl der existierenden Generationen zurück, während der Wert 0 besagt, dass keine Zuordnung mehr von Speicher existiert.

Beispiel:

```
DCL BEREICH (100) CHAR (80) CTL;
folge von anweisungen;
ON ENDFILE (SYSIN) GOTO FREIGABE;     /* DATEIENDEBEHANDLUNG VON SYSIN */
```

```
ALLOC: ALLOCATE BEREICH;
        DO I = 1 TO 100;
        GET LIST (BEREICH (I));
        END;
        GOTO ALLOC;
FREIGABE: DO WHILE ('1'B);
            verarbeitung der letzten generation von BEREICH;
            IF ALLOCATION (BEREICH) > 0 THEN FREE BEREICH;
                                        ELSE LEAVE FREIGABE;
            END FREIGABE;
/* KEINE WEITERE GENERATION VON BEREICH VORHANDEN */
```

11.4. BASED - Speicher

Die Vereinbarung einer BASED - Variablen gibt nur den geforderten Speicherplatzbedarf an. Die Lokalisierung dieses Speicherplatzes erfolgt über eine Zeiger - Variable. Da eine BASED - Variable mit mehreren Zeigervariablen verbunden sein kann, können BASED - Variable mehrere Generationen von Daten, die jederzeit alle direkt verfügbar sind, besitzen. Darin unterscheiden sich BASED - Variable von allen anderen Variablen.

Allgemeine Form:

```
< basisbezogene variable > ::= < bezeichner > < attribute > BASED [( < zeigervariable > )];
```

zeigervariable ist eine Variable, deren Wert die jeweilige Hauptspeicheradresse darstellt.

Es gibt 5 Möglichkeiten, einem BASED - Speicher, einer Zeigervariablen, eine Hauptspeicheradresse zuzuordnen:

1. READ - Anweisung mit der SET-Option
2. LOCATE - Anweisung mit der SET-Option
3. die eingefügten Funktionen ADDR und NULL
4. durch eine POINTER - Zuweisung
5. ALLOCATE mit der SET - Option

Zu 1. und 2.
Die READ - und LOCATE - Anweisungen werden im Punkt 12. "Satzweise Ein - und Ausgabe" 12.3. und 12.4.1. besprochen

Zu 3.
Die ADDR - Funktion liefert als Funktionswert die absolute Adresse des angegebenen Arguments, die der BASED - Variablen und der mit ihr verbundenen Zeigervariablen zugewiesen werden kann. Als Argument einer ADDR Funktion sind möglich:

- Elementvariable
- Bereich
- Struktur
- AREA

- Bereichselement
- Unterstruktur oder Element einer Struktur

Beispiel:

```
DCL KET_1 CHAR(11),
      KET_2 CHAR (6) BASED (P);
      folge von anweisungen;
      KET_1 = 'ANFANGSWERT';
      P = ADDR ( KET_1);          /* KET_1 UND KET_2 HABEN DIE GLEICHEN ADRESSEN */
```

Beispiel zur Verwendung des PARM-Parameters:

```
HAUPT: PROC (PARM) OPTIONS (MAIN);
   DCL PARM CHAR (100) VARYING;
   DCL 1 PARM_STR BASED (P),
           2 KENN CHAR (1),
           2 DATEN CHAR (25),
           2 .....;
    DCL P POINTER;      /* NICHT ZWINGEND, TEXTABHAENGIGE DEKLARATION */
    folge von anweisungen;
    P = ADDR ( PARM);   /* PARM_STR WIRD ADRESSE VOM PARM-PARAMETER
ZUGEORDNET */
    IF PARM_STR.KENN = 'A' THEN ....
```

Die Funktion NULL gibt einen absoluten Adresswert an, der in keiner Beziehung zu einem Speicherplatz steht.

Beispiel:

```
P = NULL;      /* AKTUELLE SPEICHERZUORDNUNG WIRD ZERSTOERT */
folge von anweisungen;
IF P = NULL  THEN   /* P ENTHAELT KEINEN REELLEN SPEICHER */
```

Zu 4.

Die POINTER - Zuweisung ist im einfachsten Fall nur eine vereinfachte Darstellung einer Adresszuweisung.

Beispiel 1:

```
DCL KET_1 CHAR(11),
      KET_2 CHAR (6) BASED (P);
      folge von anweisungen;
      KET_1 = 'ANFANGSWERT';
      P  ->  KET_1;          /* KET_1 UND KET_2 HABEN DIE GLEICHEN ADRESSEN */
                             /* P ZEIGT AUF KET_1 */
```

Beispiel 2:

```
DCL  KETTE CHAR(4) BASED (P1),
       WERT PIC '(4)9 BASED (P2),
       BITKET(4)  BIT (8) BASED,
       (P1,P2) POINTER;
ALLOCATE KETTE, WERT;
folge von anweisungen;
P1 -> BITKET = '0'B;
folge von anweisungen;
```

```
P2 -> BITKET = '1'B;
```

Die Variablen KETTE und WERT werden durch die explizite Vereinbarung mit dem BASED-Attribut mit POINTER-Variablen verbunden, die zur Lokalisierung des den Variablen zuzuordnenden Speicherplatzes dienen können. Die Verbindung mit den POINTER-Variablen erfolgte also implizit. Eine explizite Verbindung mit zwei POINTER-Variablen erfolgte mittels POINTER-Zuweisung für die Bitkette BITKET. Jede POINTER-Variable dient der Identifizierung einer Generation von BITKET.

Zu 5.
Die Zuordnung von Speicherplatz einer BASED-Variablen wird durch die ALLOCATE - Anweisung erreicht; durch Nutzung der SET-Option wird eine weitere Generation der betreffenden Variablen definiert.

Beispiel:

```
DCL  WERT FIXED (7,2) BASED (P);
DCL (P,Q) PIONTER;
ALLOCATE WERT;                  /*SPEICHERPLATZ WIRD ZUR VERFUEGUNG GESTELLT
*/
                                 /* PIONTER P ERHAELT AKTUELLE ADRESSE */
folge von anweisungen;
ALLOCATE WERT SET (Q);          /* 2.GENERATION MIT ANDERER ADRESSE POINTER Q */

WERT = 12345.67;                /* 1. GENERATION */
Q -> WERT = -12345.67;          /* 2. GENERATION */
Q -> WERT = Q -> WERT + 1;
PUT LIST (Q -> WERT) SKIP;

FREE  Q -> WERT;                /* FREIGABE 2. GENERATION */
FREE WERT;                      /* FREIGABE 1. GENERATION */
```

Beispiel:

```
DCL  MAT (3,3) FIXED BASED (P);
folge von anweisungen;
ALLOCATE MAT SET (P);
ALLOCATE MAT SET (Q);
P -> MAT = P -> MAT + Q -> MAT;     /* DATEN BEIDER GENERATIONEN VERKNUEPFT */
```

Durch Nutzung der REFER-Option in BASED-Strukturen können dynamische Dimensionsgrenzen und Kettenlängen vereinbart werden. Die entsprechenden Dimensionsgrenzen und Kettenlängen werden durch Variable definiert, die innerhalb der gleichen Struktur zur Abarbeitungszeit definiert sein müssen.

Beispiel:

```
DCL 1 STR BASED (P),
        2  N BIN FIXED;
        2 KETTE CHAR(K REFER(N));
folge von anweisungen;
K = 10;
ALLOCATE STR SET (P);
```

```
KETTE = 'BERLIN';                /*  SPEICHERINHALT: BERLINbbbb */
```

Die aktuelle Länge von KETTE ist 10 Byte und wird N als Wert zugewiesen.

11.5. AREA- und OFFSET - Variable

Die Speicherplatzzuweisungen sind im Allgemeinen für BASED-Variable über den ganzen Speicher verstreut. Möchte man einen zusammenhängenden Hauptspeicherbereich verwalten, so ist die Gebietsvariable AREA einzusetzen. AREA-Variable haben eine Standardlänge von 1000 Byte.

Beispiel:

```
DCL GEBIET AREA (5000);
DCL BEREICH (1000) CHAR (1) BASED (P),
      (P,Q) POINTER;
ALLOCATE BEREICH SET (P) IN (GEBIET);
folge von anweisungen;
ALLOCATE BEREICH SET (Q) IN (GEBIET);        /* 2000 VON 5000 BYTE BELEGT */
```

OFFSET-Variable sind Zeigervariable, die nur im Zusammenhang mit AREA-Variablen verwendet werden. Sie werden durch Angabe des OFFSET-Attributes in einer DECLARE-Anweisung explizit vereinbart. Der Vorteil der Verbindung einer OFFSET-Variablen mit einer AREA-Variablen besteht darin, dass jede Bezugnahme auf die OFFSET-Variable die Bezugnahme auf die entsprechende AREA-Variable impliziert.

Beispiel:

```
DCL VAR1 BASED (P),
      VAR2 BASED (OFFS),
      OFFS OFFSET (GEBIET),
      GEBIET AREA (1000);                    /* OFFS WIRD MIT GEBIET VERBUNDEN
*/

ALLOCATE VAR1 SET (P);                       /*   ZUWEISUNG VON SPEICHERPLATZ  */
ALLOCATE VAR2 SET (OFFS) IN (GEBIET);
```

Die erste ALLOCATE-Anweisung ordnet der BASED-Variablen einen Speicherplatz zu, auf dessen Anfang die in der SET-Option angegebene POINTER-Variable hinweist. Die zweite ALLOCATE-Anweisung ordnet VAR2 einen Speicherplatz in GEBIET zu, auf dessen Anfang die OFFSET-Variable OFFS hinweist.

12. Satzweise Ein- und Ausgabe

12.1. Allgemeine Betrachtungen

Entsprechend den unterschiedlichen Anforderungen einer praxisnahen Arbeit enthält die Programmiersprache PL/1 zwei grundsätzliche Grundformen für die Durchführung von Ein - und Ausgabeoperationen:

- die Datenübertragung mit Aufbereitung der Daten, die reihenweise Ein - und Ausgabe
- die Datenübertragung ohne Aufbereitung der Daten, die satzweise Ein - und Ausgabe

Die reihenweise Ein - und Ausgabe wurde im Punkt 10. mit ihren ganzen Möglichkeiten der Datenaufbereitung umfangreich behandelt, wobei nochmals darauf verwiesen wird, dass nicht alle möglichen Konstrukte besprochen werden können. Das gilt natürlich erst recht für die satzweise Ein -und Ausgabe; vor allem die Vielfältigkeit der Dateiorganisationsformen, die in 9.1.3.5. ENVIRONMENT-Attribut schon kurz erläutert wurden, können in ihrer Vielfalt nur im Grundsätzlichen angesprochen werden.

Programmseitig ist bei der Verwendung der satzweisen Datenübertragung vor allem von Bedeutung, dass sich jede Ein- oder Ausgabeanweisung grundsätzlich auf genau einen vollständigen Satz bezieht. Während die Anweisungen der reihenweisen Datenüber-tragung nur zulassen, Dateien aus im Hauptspeicher befindlichen Daten aufzubauen oder Daten von vorhandenen Dateien in den Hauptspeicher einzulesen, besteht bei der satz-weisen Datenübertragung darüber hinaus auch die Möglichkeit, Sätze einer Datei in ihrem Inhalt zu verändern, zu ersetzen, zu löschen sowie neue Sätze in einen vorliegenden Datenbestand sequentiell als auch im wahlfreiem Zugriff einzufügen.

12.2. Satzweise Ein - und Ausgabeanweisungen

Bei der satzweisen Verarbeitung drängt sich der Vergleich auf, dass unter Eingabe die naturgetreue Abbildung eines vollständigen Satzes von einem externen Speichermedium in den Hauptspeicher verstanden wird, während die Ausgabe die Kopie eines Satzes vom Hauptspeicher auf ein externes Medium beschreibt. Daraus ergeben sich gewisse Konsequenzen, die bei der Nutzung beachtet werden müssen:

- keine Datenkonvertierungen
- keine Angaben von Formaten
- notwendige Datenkonvertierungen müssen vor der Eingabe in den Hauptspeicher
  bzw. vor der Ausgabe aus dem Hauptspeicher an den Daten vorgenommen werden

Allgemeine Form der Ein - und Ausgabe:

```
READ FILE (dateiname) { INTO ( bezeichner ) | SET ( zeiger-variable ) |
                        IGNORE ( skalarer ausdruck) }
                        [ KEY ( skalarer ausdruck )]
                        [ KEYTO (  zeichenkettenvariable ) ];

WRITE FILE ( dateiname ) FROM ( bezeichner) [ KEYFROM ( ( skalarer ausdruck )];
LOCATE  bezeichner FILE ( dateiname ) SET ( zeiger-variable ) [ KEYFROM ( ( skalarer aus-
druck )];
DELETE ( dateiname)
REWRITE ( dateiname)
UNLOCK ( dateiname)
```

Als Satzvariable < bezeichner > können alle Variablen außer den folgenden verwendet werden:

- Variable, die nicht auf der Stufe 1 stehen
- Parameter
- DEFINED-Variable
- Struktur, die VARYING-Ketten enthält
- Variable, die nicht auf der Stufe 1 stehen als Satzvariable der LOCATE-Anweisung
- Felder und Strukturen, die keinen zusammenhängenden Speicherplatz belegen, Feld- und Strukturparameter müssen das Attribut CONNECTED besitzen
- BASED-, DEFINED-Variable, Feld- oder Strukturelemente, die UNALIGNED-Bitketten fester Länge sind.
- Hauptstrukturen, die Parameter, BASED- oder DEFINED-Variablen sind und deren erstes oder letztes Element eine UNALIGNED-Bitkette fester Länge ist.

12.3. Verarbeitungsmodi

Bei der satzweisen Ein- und Ausgabe gibt es zwei verschiedene Verarbeitungsmodi für die Daten:

- Transportmodus

- Zeigermodus

Beim Transportmodus werden die Daten vom externen Speicher, gegebenenfalls über Puffer, in die Satzvariable übertragen oder umgekehrt. Die Verarbeitung der Daten erfolgt in dem der Variablen zugewiesenen Speicherbereich.

Für diesen Verarbeitungsmodus werden die READ - Anweisung ohne SET - Option, die WRITE - und die REWRITE - Anweisung benutzt.

Beim Zeigermodus werden die Daten vom externen Speicher in einen oder mehrere Puffer übertragen oder umgekehrt. Die Verarbeitung der Daten erfolgt im Puffer.

Das erfordert die Benutzung einer oder mehrerer BASED - Variablen, durch die die Sätze beschrieben werden.

Der Zugriff zu einem Satz im Puffer wird durch Zuweisung eines entsprechenden Wertes zu der POINTER - Variablen ermöglicht, die durch die BASED - Vereinbarung oder durch die SET - Option der READ- oder LOCATE - Anweisung angegeben wird.

Für den Zeigermodus können nur die READ - Anweisung mit SET - Option, und die LOCATE - Anweisung verwendet werden. Er hat den Vorteil, dass man mit geringerem Speicherplatzbedarf arbeitet und eine geringere Datenbewegung erforderlich ist als beim Transportmodus. Letzterer ist jedoch einfacher für den Programmierer zu handhaben, da nicht die Speicherungsform der Daten im Puffer, d.h. die Ausrichtung der Datenelemente, zu beachten ist.

12.4. Dateiorganisationsformen

In PL/1 wird die Organisationsform der Daten auf externen Datenträgern durch die Angabe der entsprechenden ENVIRONMENT - Option beschrieben (siehe 9.3.1.5.). Bei der satzweisen Ein- und Ausgabe können starr fortlaufende (CONSECUTIVE), regionale (REGIONAL(1), REGIONAL(2) oder REGIONAL(3)), logisch fortlaufende (INDEXED) Dateiorganisationsformen benutzt werden. Weitere Organisationsformen wie, Datenfernverarbeitung oder VSAM-Dateiorganisationsformen werden hier nicht besprochen.

12.4.1. Verarbeitung von CONSECUTIVE - FILE

1. Die Datei kann nur sequentiell aufgebaut, gelesen, aktualisiert und erweitert werden. Die Sätze besitzen keine Schlüssel.
2. Das Aufbauen der Datei erfolgt mit SEQUENTIAL OUTPUT. Die Sätze werden in der Reihenfolge der Ausgabe auf den Datenträger geschrieben.
3. Eine bereits existierende Datei kann zu ihrer Verarbeitung eröffnet werden mit:
   - SEQUENTIAL INPUT
     Die Sätze werden in der Reihenfolge der physischen Anordnung auf dem Datenträger gelesen. Ein Rückwärtslesen (Attribut BACKWARDS) von Magnetbanddateien mit Sätzen fester oder undefinierter Länge ist möglich.

   - SEQUENTIAL OUTPUT
     Die Sätze werden in der Reihenfolge der Ausgabe auf den Datenträger geschrieben. In Abhängigkeit der Disposition wird die Datei erweitert (bei DISP = MOD) oder vom Anfang an überschrieben.

- SEQUENTIAL UPDATE
  Lesen wie bei SEQUENTIAL INPUT, aber kein Rückwärtslesen möglich. Sätze können ersetzt werden, wenn sie vorher gelesen wurden. Es wird stets der zuletzt gelesene Satz ersetzt.

4. CONSECUTIVE - Dateien können, unabhängig von der Art der Datenübertragung (STREAM oder RECORD) beim Aufbauen, danach sowohl mit STREAM- als auch mit RECORD - Dateibeschreibungen eröffnet werden. Allerdings können mit STREAM eröffnete Dateien nicht aktualisiert oder rückwärts gelesen werden und dürfen nicht das Satzformat VS oder VBS haben.

Beispiel 1:

Es soll eine Datei aufgebaut werden, die Informationen über diejenigen ihrer Schuldner enthält, deren Schuldsumme größer als 10.000,00 € ist. Zur Eingabe gelangt eine Datei, die die Daten aller ihrer Schuldner erhält. Das Mahnbyte wird bei der Ausgabe um 1 erhöht.

```
PROG_1: PROC OPTIONS (MAIN);
/* ERSTELLEN EINES CONSECUTIVE-FILES IM TRANSPORTMODUS */
DCL DAT_SCHULD FILE RECORD, DAT_GRSCHULD FILE RECORD;
DCL 1 SCHULDNER,
          2 NAME,
            3 IDENT_NR PIC '(8)9',
             3 NACHNAME CHAR (15),
             3 VORNAME CHAR (10),
           2 ADRESSE,
             3 PLZ PIC '(5)9',
             3 ORT CHAR (15),
             3 STRASSE CHAR (20),
             3 NUMMER CHAR (4),
           2 KONTO_NUMMER CHAR (8),
           2 SCHULDSUM FIXED (9,2),
           2 MAHNBYTE FIXED (2);

ON ENDFILE  (DAT_SCHULD) GOTO ENDE;
OPEN FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD) OUTPUT;

LIES: READ FILE (DAT_SCHULD) INTO (SCHULDNER);
        IF SCHULDNER.SCHULDSUM > 10000 THEN
        DO;
        SCHULDNER.MAHNBYTE = SCHULDNER.MAHNBYTE  + 1;
        WRITE FILE ( DAT_GRSCHULD ) FROM (SCHULDNER);
        END;
        GOTO LIES;

ENDE: PUT LIST ( 'ALLE DATEN VERARBEITET') PAGE SKIP;
CLOSE FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD);
END PROG_1;
```

In diesem Programm werden die Daten vom Eingabepuffer in den Hauptspeicherbereich und anschließend werden die auszugebenen Daten in den Ausgabepuffer übertragen; diesen Verarbeitungsmodus der Ein - und Ausgabe nennt man den Transportmodus (siehe 12.3.).
Im Gegensatz dazu werden im Zeigermodus die Puffer durch das Betriebssystem mittels eines Zeigers, eines POINTER, verwaltet. Der Datenstruktur muss die Anfangsadresse des Eingabepuffers mittels Zeiger mitgeteilt werden. Dazu darf die Struktur im Prolog keine reale Hauptspeicheradresse erhalten; sie wird basisbezogen auf einen Zeiger definiert. Erst zur Eingabe wird der basisbezogenen Struktur die Adresse des aktuellen Satzes durch die SET - Option mitgeteilt. Die nächste READ - Anweisung setzt den Zeiger entsprechend der Satzlänge im Eingabepuffer auf die Anfangsadresse des nächsten Satzes.

```
DCL 1 SCHULDNER BASED (P),
        2 NAME,
        .......
        2 SCHULDSUM FIXED (9,2);
DCL P POINTER;

LIES: READ FILE (DAT_SCHULD) SET (P);
```

Die Verwaltung des Ausgabepuffers wird über die SET - Option in der LOCATE - Anweisung realisiert. Durch die LOCATE - Anweisung werden die Daten im Ausgabepuffer maskiert; sie werden nicht ausgegeben. Die nächste LOCATE - Anweisung "schiebt " die Daten gewissermaßen auf die entsprechende Datei.

```
LOCATE SCHULDNER FILE (DAT_SCHULD) SET (P);
```

Abschließendes Beispiel:

```
PROG_2: PROC OPTIONS (MAIN);
/* ERSTELLEN EINES CONSECUTIVE-FILES IM ZEIGERMODUS */
DCL DAT_SCHULD FILE RECORD, DAT_GRSCHULD FILE RECORD;
DCL 1 SCHULDNER BASED (P),
        2 NAME,
          3 IDENT_NR PIC '(8)9',
          3 NACHNAME CHAR (15),
          3 VORNAME CHAR (10),
        2 ADRESSE,
          3 PLZ PIC '(5)9',
          3 ORT CHAR (15),
          3 STRASSE CHAR (20),
          3 NUMMER CHAR (4),
        2 KONTO_NUMMER CHAR (8),
        2 SCHULDSUM FIXED (9,2);

ON ENDFILE  (DAT_SCHULD) GOTO ENDE;
OPEN FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD) OUTPUT;

LIES: READ FILE (DAT_SCHULD) SET (P);
      IF SCHULDNER.SCHULDSUM > 10000 THEN
```

```
        DO;
        LOCATE SCHULDNER FILE ( DAT_GRSCHULD ) SET (P);    /* DATENBEREITSTELLG
*/
        SCHULDNER.MAHNBYTE = SCHULDNER.MAHNBYTE  + 1;      /*VERARBEITUNG */
        END;
        GOTO LIES;

ENDE: PUT LIST ( 'ALLE DATEN VERARBEITET') PAGE SKIP;
CLOSE FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD);
END PROG_1;
```

12.4.2. Verarbeitung von INDEXED - FILE

1. Die Datei ist logisch fortlaufend organisiert. Die Sätze haben das Satzformat F, FB, V oder VB und besitzen aufgezeichnete Zeichenkettenschlüssel. In der Datei können sowohl signifikante als auch Pseudosätze enthalten sein.

2. Pseudosätze sind durch hexadezimal FF gekennzeichnet. Sie können durch signifikante Sätze, die die gleichen Schlüssel wie die Pseudosätze haben müssen, ersetzt werden.

3. Die ersten n-Zeichen des Schlüssels, des Zeichenkettenwertes des in der KEY - oder KEYFROM - Option der E/A - Anweisung angegebenen Ausdrucks, liefern den aufgezeichneten Schlüssel. n ist die in der KEYLENGTH - Option angegebene
   Schlüssellänge. Ist der Schlüssel kürzer als n, so wird er rechts mit Leerzeichen auf die angegebene Länge erweitert. Der aufgezeichnete Schlüssel kann maximal 255 Zeichen lang sein. Er ist als eingefügter Schlüssel in der Satzvariablen an der durch die KEYLOC - Option beim Aufbauen der Datei festgelegten Position enthalten.
   Unabhängig davon, ob der aufgezeichnete Schlüssel im Satz enthalten ist oder nicht, steht der aufgezeichnete Schlüssel des letzten Satzes (d. h. des einzigen Satzes im Falle ungeblockter Sätze) vor jedem Block in der Datei.
   Bei der Ausführung einer WRITE - Anweisung wird, falls eingefügte Schlüssel vorhanden sind, der in der KEYFROM - Option angegebene Schlüssel der Position des eingefügten Schlüssels in der Satzvariablen zugewiesen. Falls die Satzvariable eine Struktur ist, darf der eingefügte Schlüssel kein Strukturelement sein, dass mit dem VARYING - Attribut vereinbart ist.
   Bei der Verwendung der LOCATE - Anweisung wird, falls eingefügte Schlüssel vorhanden sind, der in der KEYFROM - Option angegebene Schlüssel beim Ausführen der nächsten E/A - Operation auf die Position des eingefügten Schlüssels in der Satzvariablen übertragen.
   Bei Verwendung der WRITE - oder LOCATE - Anweisung wird, falls eingefügte Schlüssel vorhanden sind, der in der KEYFROM - Option angegebene

Schlüssel mit dem eingefügten Schlüssel verglichen. Bei Ungleichheit tritt der Ausnahmezustand KEY auf.

4. Das Aufbauen der Datei erfolgt mit:
   - SEQUENTIAL OUTPUT
     Die Sätze müssen nach aufsteigenden Schlüsseln geordnet ausgegeben werden. Fehler in der Reihenfolge führen zum Auftreten des Ausnahmezustandes KEY.

5. Eine bereits existierende Datei kann wieder eröffnet werden mit:
   - SEQUENTIAL INPUT
     Die Sätze werden in aufsteigender Reihenfolge ihrer Schlüssel gelesen (READ - Anweisung ohne KEY - Option). Es ist möglich, einen bestimmten Satz durch Angabe seines Schlüssels oder den ersten Satz einer Schlüsselklasse (siehe GENKEY - Option) durch Angabe des Klassenschlüssels in der KEY - Option der READ - Anweisung zu lesen und damit auf einen bestimmten Satz vorwärts oder rückwärts zu positionieren. Eine READ - Anweisung ohne KEY- Option liest den Satz mit dem nächst höheren Schlüssel.

   - SEQUENTIAL OUTPUT
     Die Datei muss F - Formatsätze enthalten. Die zu schreibenden Sätze werden zu der bestehenden Datei hinzugefügt. Sie müssen in aufsteigender Reihenfolge ihrer Schlüssel geschrieben werden. Ihre Schlüssel müssen größer sein als die Schlüssel der in der Datei schon enthaltenen Sätze.

   - SEQUENTIAL UPDATE
     Lesen wie bei SEQUENTIAL INPUT, Ersetzen wie bei CONSECUTIVE - Dateien. Ein eingefügter Schlüssel darf nicht geändert werden. Löschen von Sätzen mit der DELETE - Anweisung ist möglich. Gelöscht wird der zuletzt gelesene Satz. Geblockte Sätze mit KEYLOC (1) können nicht gelöscht werden, da sonst der aufgezeichnete Schlüssel zerstört werden würde.

   - DIRECT INPUT
     Wahlfreies Lesen beliebiger Sätze

   - DIRECT UPDATE
     Lesen wie bei DIRECT INPUT.
     Hinzufügen von Sätzen, für deren Schlüssel in der Datei noch keine signifikanten Sätze existieren: Ist ein Schlüssel der eines Pseudosatzes, so wird dieser überschrieben, ist der Schlüssel der eines gültigen Satzes, so tritt der Ausnahmezustand KEY auf.
     Ersetzen gültiger Sätze: Enthält die Datei V - oder VB - Formatsätze und die Länge des neuen Satzes ist ungleich der des alten, so wird der Rest des Spurinhaltes der Datei verschoben. Es können Sätze in einen Überlaufbereich ausgelagert werden.

Löschen von Sätzen ist möglich, wenn beim Aufbauen der Datei entsprechende Voraussetzungen geschaffen wurden. Geblockte Sätze mit KEYLOC (1) können nicht gelöscht werden, da sonst der aufgezeichnete Schlüssel zerstört werden würde.

Beispiel 2 zum sequentiellen Erstellen eines INDEXED - FILE:

Es soll eine indexsequentielle Datei aufgebaut werden, die Informationen über diejenigen ihrer Schuldner enthält, deren Schuldsumme größer als 10.000,00 € ist. Zur Eingabe gelangt eine Datei, die die Daten aller ihrer Schuldner erhält. Als Schlüssel wird die IDENT_NR erstellt. Das Mahnbyte wird bei der Ausgabe um 1 erhöht.

```
PROG_2: PROC OPTIONS (MAIN);
/* ERSTELLEN EINES INDEXED-FILES IM TRANSPORTMODUS */
DCL DAT_SCHULD FILE RECORD,
      DAT_GRSCHULD FILE RECORD KEYED ENVIRONMENT ( INDEXED);
DCL 1 SCHULDNER,
         2 NAME,
           3 IDENT_NR PIC '(8)9',
           3 NACHNAME CHAR (15),
           3 VORNAME CHAR (10),
         2 ADRESSE,
           3 PLZ PIC '(5)9',
           3 ORT CHAR (15),
           3 STRASSE CHAR (20),
           3 NUMMER CHAR (4),
         2 KONTO_NUMMER CHAR (8),
         2 SCHULDSUM FIXED (9,2),
         2 MAHNBYTE FIXED (2);

ON ENDFILE  (DAT_SCHULD) GOTO ENDE;
OPEN FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD) OUTPUT SEQUENTIAL;

LIES: READ FILE (DAT_SCHULD) INTO (SCHULDNER);
       IF SCHULDNER.SCHULDSUM > 10000 THEN
       DO;
       SCHULDNER.MAHNBYTE = SCHULDNER.MAHNBYTE  + 1;
       WRITE FILE ( DAT_GRSCHULD ) FROM (SCHULDNER) KEYFROM (IDENT_NR);
       END;
       GOTO LIES;

ENDE: PUT LIST ( 'ALLE DATEN VERARBEITET') PAGE SKIP;
CLOSE FILE ( DAT_SCHULD) , FILE (DAT_GRSCHULD);
END PROG_2;
```

Beispiel 3 zum sequentiellen Update eines INDEXED - FILE:

Die im Beispiel 2 aufgebaute indexsequentielle Datei soll dahingehend verändert werden, dass alle Schuldner, deren Schuldsumme inzwischen durch Ausgleich kleiner als 10.000 € ist, aus dieser Datei gelöscht werden. Zur Eingabe gelangt

eine Datei, die alle zu bearbeitenden Schuldner beinhaltet. Als Schlüssel wird weiterhin die IDENT_NR benutzt. Es werden alle Sätze zurückgeschrieben, deren Schuldsumme immer noch höher als 10.000 € beträgt. Das Mahnbyte wird um 1 erhöht.

```
PROG_3: PROC OPTIONS (MAIN);
/* DIRECTES UPDATE EINES INDEXED-FILES */
DCL  DAT_GRSCHULD FILE RECORD KEYED ENVIRONMENT ( INDEXED),
       DAT_KLSCHULDNER FILE RECORD;
DCL 1 SCHULDNER,
         2 NAME,
           3 IDENT_NR PIC '(8)9',
           3 NACHNAME CHAR (15),
           3 VORNAME CHAR (10),
         2 ADRESSE,
           3 PLZ PIC '(5)9',
           3 ORT CHAR (15),
           3 STRASSE CHAR (20),
           3 NUMMER CHAR (4),
         2 KONTO_NUMMER CHAR (8),
         2 SCHULDSUM FIXED (9,2),
         2 MAHNBYTE FIXED (2);
DCL TEMP_SCHULD FIXED (9,2);

ON ENDFILE  (DAT_KLSCHULD) GOTO ENDE;
OPEN FILE (DAT_GRSCHULD) OUTPUT UPDATE,
         FILE (DAT_KLSCHULD) INPUT;

LIES: READ FILE (DAT_KLSCHULD) INTO (SCHULDNER);
       IF SCHULDNER.SCHULDSUM >= 10000 THEN
       DO;
          TEMP_SCHULD = SCHULDNER_SCHULDSUM;
          READ FILE ( DAT_GRSCHULD) INTO (SCHULDNER) KEY ( IDENT_NR );
          SCHULDNER.MAHNBYTE = SCHULDNER.MAHNBYTE  + 1;
          SCHULDNER_SCHULDSUM = TEMP_SCHULD;
          REWRITE FILE ( DAT_GRSCHULD ) FROM (SCHULDNER) KEY (IDENT_NR);
       END;
       ELSE  DELETE FILE ( DAT_GRSCHULD ) KEY (IDENT_NR);
GOTO LIES;

ENDE: PUT LIST ( 'ALLE DATEN VERARBEITET') PAGE SKIP;
CLOSE FILE ( DAT_KLSCHULD) , FILE (DAT_GRSCHULD);
END PROG_2;
```

12.4.3. Verarbeitung von REGIONAL(1) FILE - die direkte Satzadressierung

1. Die Datei muss sich auf einem Direktzugriffsdatenträger befinden. Sie ist in Regionen eingeteilt. Eine Region enthält genau einen Satz vom Format F. Die Regionen sind mit Null beginnend nummeriert. Die größtmöglichte Regionsnummer ist 16.777.215. In der Datei können neben signifikanten Sätzen auch

Pseudosätze enthalten sein. Letztere sind durch hexadezimal FF im ersten Byte gekennzeichnet.

2. Der Zugriff auf die Sätze kann direkt durch die Angabe der Regionsnummer oder sequentiell erfolgen. Die Sätze besitzen keine aufgezeichneten Schlüssel.

Die Regionsnummer wird aus dem Schlüssel, dem Zeichenkettenwert des Ausdrucks in der KEY- oder KEYFROM-Option der E/A-Anweisung, bestimmt. Der Schlüssel darf nur aus Ziffern und Leerzeichen bestehen. Ist der Schlüssel kürzer als 8 Zeichen, so wird links mit Leerzeichen auf die Länge 8 erweitert. Die letzten 8 Zeichen des Schlüssels werden als Dezimalzahl interpretiert und aus ihr modulo 16.777.216 die Regionsnummer berechnet.

Führende Leerzeichen werden als Nullen interpretiert. Tritt ein Leerzeichen nach einer oder mehreren Ziffern auf, so werden nur die davor stehenden Ziffern zur Berechnung der Regionsnummer verwendet.

3. Das Aufbauen der Datei kann erfolgen mit:

- SEQUENTIAL OUTPUT

Die Sätze müssen nach aufsteigenden Regionsnummern geordnet ausgegeben werden; die Regionen, die ausgelassen werden, werden mit Pseudosätzen belegt. Bei einem Fehler in der Reihenfolge oder bei mehrfacher Verwendung einer Regionsnummer tritt der Ausnahmezustand KEY auf. Wird die Datei geschlossen, so wird der noch nicht belegte Speicherplatz des der Datei zugeordneten Bereiches mit Pseudosätzen aufgefüllt.

- DIRECT OUTPUT

Beim Eröffnen der Datei wird der ihr zugeordneter Bereich mit Pseudosätzen aufgefüllt. Entsprechend den angegebenen Regionsnummern werden dann die Sätze in die Regionen geschrieben.

4. Eine bereits existierende Datei kann zu ihrer Verarbeitung eröffnet werden mit:

- SEQUENTIAL INPUT

Die Sätze werden in der Reihenfolge der Regionsnummern gelesen.
Die Regionsnummer des gerade gelesenen Satzes kann man mit Hilfe der KEYTO Option erhalten, wenn die Dateibeschreibung das Attribut KEYED besitzt.

- SEQUENTIAL UPDATE

Lesen wie bei SEQUENTIAL INPUT, Ersetzen wie bei CONSECUTIVE-Datei

- DIRECT INPUT

Lesen beliebiger Sätze

- DIRECT UPDATE

Lesen, Ersetzen und Löschen beliebiger Sätze ist möglich. Gelöschte Sätze sind als Pseudosätze gekennzeichnet. Das Schreiben von Sätzen mit der WRITE - Anweisung ist äquivalent dem Ersetzen mit der REWRITE - Anweisung

5. Pseudosätze werden bei der Verarbeitung der Datei nicht erkannt. Sie werden wie alle anderen Sätze behandelt. Das PL1 - Programm muss gegebenenfalls selbst für ihre Erkennung sorgen

6. Wird eine bereits bestehende REGIONAL(1) - Datei mit OUTPUT eröffnet, so wird die ganze existierende Datei überschrieben

12.4.4. Verarbeitung von REGIONAL(2) FILE - die indirekte Satzadressierung

1. Die Datei muss sich auf einem Direktzugriffsdatenträger befinden. Sie ist in Regionen eingeteilt. Eine Region enthält genau einen Satz vom Format F. Die Regionen sind mit Null beginnend nummeriert. Die größtmögliche Regionsnummer ist 16.777.215. In der Datei können sowohl signifikante als auch Pseudosätze enthalten sein. Letztere sind durch hexadezimal FF im ersten Byte des aufgezeichneten Schlüssels und durch die Folgenummer des Satzes auf der Spur im ersten Datenbyte gekennzeichnet.

2. Die Sätze besitzen aufgezeichnete Schlüssel. Auf die Sätze kann direkt durch Angabe eines Schlüssels oder sequentiell zugegriffen werden. Der Schlüssel ist der Zeichenkettenwert des in der KEY - oder KEYFROM - Option der E/A - Anweisung angegebenen Ausdrucks und enthält zwei Teile, den Vergleichsschlüssel und die Regionsnummer.
   Der Vergleichsschlüssel wird bei der Ausgabe als aufgezeichneter Schlüssel geschrieben und bei der Eingabe mit dem aufgezeichneten Schlüssel verglichen. Er besteht aus den ersten n Zeichen des Schlüssels, wobei n die Angabe aus der KEYLENGTH - Option ist. Ist der Schlüssel zu kurz, wird er rechts mit Leerzeichen aufgefüllt.

   Die letzten 8 Zeichen des Schlüssels dürfen nur Ziffern und Leerzeichen sein. Sie werden als Dezimalzahl interpretiert und aus ihr modulo 16.777.216 die Regionsnummer berechnet.

   Führende Leerzeichen werden als Nullen interpretiert. Tritt ein Leerzeichen nach einer oder mehreren Ziffern auf, so werden nur die davor stehenden Ziffern zur Berechnung der Regionsnummer verwendet.

3. Das Aufbauen der Datei kann erfolgen mit:
   - SEQUENTIAL OUTPUT
     Die Sätze müssen nach aufsteigenden Regionsnummern geordnet ausgegeben werden; ausgelassene Regionen werden mit Pseudosätzen aufgefüllt. Fehler in der Reihenfolge oder mehrfache Verwendung der gleichen Regionsnummer führen zum Auftreten des Ausnahmezustandes KEY. Wird die Datei abgeschlossen, so wird noch unbelegter Speicherplatz des der Datei zugeordneten Bereiches mit Pseudosätzen aufgefüllt.

- DIRECT OUTPUT
  Beim Eröffnen der Datei wird der ihr zugeordnete Bereich mit Pseudosätzen aufgefüllt. Die Sätze können in beliebiger Reihenfolge in die Datei geschrieben werden. Es wird stets der zuerst gefundene Pseudosatz durch den ausgegebenen Satz überschrieben.
  Die Suche nach einer Region mit einem Pseudosatz erfolgt sequentiell vom Anfang der Spur, auf der sich die angegebene Region befindet, bis zum Ende der Datei und wird dann vom Anfang der Datei fortgesetzt bis die ganze Datei durchsucht ist. Mehrfache Verwendung von Regionsnummern und aufgezeichneten Schlüsseln führen nicht zum Auftreten eines Ausnahmezustandes.

4. Eine bereits existierende Datei kann zu ihrer Verarbeitung eröffnet werden mit:
   - SEQUENTIAL INPUT
     Die Sätze werden in der Reihenfolge ihrer physischen Ordnung in der Datei gelesen. Den aufgezeichneten Schlüssel des gerade gelesenen Satzes kann man erhalten durch die KEYTO - Option, wenn die Dateibeschreibung das Attribut KEYED besitzt.

   - SEQUENTIAL UPDATE
     Lesen wie bei SEQUENTIAL INPUT, Ersetzen wie bei CONSECUTIVE - Datei

   - DIRECT INPUT
     Lesen beliebiger Sätze: Die Suche nach einem Satz erfolgt sequentiell vom Anfang der Spur an, die die angegebene Region enthält, bis ein Satz mit dem angegebenen Schlüssel gefunden wird. Die Suche wird bei Erreichen des Dateiendes vom Anfang
     der Datei an fortgesetzt, bis die ganze Datei durchsucht ist.

   - DIRECT UPDATE
     Lesen, Ersetzen gültiger Sätze, Hinzufügen (Ersetzen von Pseudosätzen wie bei DIRECT OUTPUT) und Löschen von beliebigen Sätzen möglich. Gelöschte Sätze sind als Pseudosätze gekennzeichnet. Die Suche nach einem Satz erfolgt wie bei DIRECT INPUT beschrieben.

5. Pseudosätze können nicht gelesen werden. Beim sequentiellen Lesen werden sie übergangen.

6. Eine bereits existierende Datei kann nicht als OUTPUT - Datei eröffnet werden.

12.4.5. Verarbeitung von REGIONAL(3) FILE - die indirekte Spuradressierung

1. Die Datei muss sich auf einem Direktzugriffsdatenträger befinden. Die Datei ist in Regionen eingeteilt. Eine Region umfasst eine Spur und enthält im Allgemeinen mehrere Sätze. Die Regionen sind mit Null beginnend nummeriert. Die größt-

mögliche Regionsnummer ist 32.767. In der Datei können sowohl signifikante als auch Pseudosätze enthalten sein.

2. Ein Pseudosatz in einer REGIONAL(3) - Datei mit Sätzen fester Länge ist identisch mit dem einer REGIONAL(2) - Datei.
   In REGIONAL(3) - Dateien mit Sätzen variabler oder undefinierter Länge können Pseudosätze nur durch Ausführung von DELETE - Anweisungen entstehen. Sie sind durch hexadezimal FF im ersten Byte des aufgezeichneten Schlüssels gekennzeichnet. Die Satzlängeninformation bei Pseudosätzen variabler Länge ist identisch mit der vor Ausführung der DELETE - Anweisung.
   Der Inhalt von Pseudosätzen variabler oder undefinierter Länge ist unbestimmt. Der von ihnen belegte Speicherplatz kann nicht wieder verwendet werden.

3. Die Sätze besitzen aufgezeichnete Schlüssel. Auf die Sätze kann direkt durch Angabe eines Schlüssels oder sequentiell zugegriffen werden. Die Schlüssel sind so wie bei REGIONAL(2) - Dateien aufgebaut. Die Regionsnummern werden modulo 32.768 berechnet.

4. Das Aufbauen der Datei kann erfolgen mit:
   - SEQUENTIAL OUTPUT
     Die Sätze müssen nach aufsteigenden Regionsnummern geordnet ausgegeben werden. Aufeinanderfolgende Sätze können die gleiche Regionsnummer haben. Fehler in der Reihenfolge führen zum Auftreten des Ausnahmezustandes KEY. Ist eine Region gefüllt und ein Satz mit der gleichen Regionsnummer soll geschrieben werden, so tritt der Ausnahmezustand KEY auf.

   - DIRECT OUTPUT
     Beim Eröffnen der Datei wird bei Sätzen fester Länge die ganze Datei mit Pseudosätzen aufgefüllt. Bei Sätzen variabler oder undefinierter Länge wird an den Spuranfang ein Kapazitätssatz eingetragen, der Informationen über den verfügbaren Speicherplatz der Region enthält. Die Sätze können in beliebiger Reihenfolge in die Datei übertragen werden. Mehrfache Verwendung voll aufgezeichneten Schlüsseln und Regionsnummern führt nicht zum Auftreten eines Ausnahmezustandes.
     Bei Sätzen fester Länge wird der jeweils erste Pseudosatz der angegebenen Region überschrieben. Sätze variabler oder undefinierter Länge werden nacheinander in die angegebene Region eingetragen. Ist in einer Region kein Pseudosatz oder kein verfügbarer Speicherplatz mehr vorhanden, so kann der Satz auf den nächsten verfügbaren Platz geschrieben werden. Die Suche danach erfolgt sequentiell vom Anfang der nächsten Spur an bis zum Ende der Datei und wird dann vom Anfang der Datei fortgesetzt, bis die ganze Datei durchsucht ist.

5. Eine bereits existierende Datei kann zu ihrer Verarbeitung eröffnet werden mit:

- SEQUENTIAL INPUT

  Die Sätze werden in der Reihenfolge ihrer physischen Ordnung in der Datei gelesen. Den aufgezeichneten Schlüssel des gerade gelesenen Satzes kann man durch die KEYTO - Option erhalten, wenn die Dateibeschreibung das Attribut KEYED besitzt.

- SEQUENTIAL UPDATE

  Lesen wie bei SEQUENTIAL - INPUT, Ersetzen wie bei CONSECUTIVE - FILE

- DIRECT INPUT

  Lesen beliebiger Sätze: Die Suche nach einem Satz erfolgt sequentiell vom Anfang der angegebenen Spur (Region) an, bis ein Satz mit dem angegebenen aufgezeichneten Schlüssel gefunden wird. Die Suche wird bei Erreichen des Dateiendes vom Anfang der Datei an fortgesetzt, bis die ganze Datei durchsucht ist.

- DIRECT UPDATE

  Lesen, Ersetzen gültiger Sätze, Hinzufügen (Eintragen von Sätze wie bei DIRECT OUTPUT) und Löschen beliebiger Sätze ist möglich. Die Suche nach einem Satz erfolgt wie bei DIRECT INPUT beschrieben. Gelöschte Sätze sind als Pseudosätze gekennzeichnet.

6. Pseudosätze können nicht gelesen werden. Beim sequentiellen Lesen werden sie übergangen.

7. Wird eine bereits bestehende Datei mit OUTPUT eröffnet, so wird die ganze existierende Datei überschrieben.

13. Bedingungen, Testhilfen

13.1. Allgemeine Betrachtungen

Während der Abarbeitung eines Programms wird das Auftreten von Ausnahme- und Fehlerbedingungen im Allgemeinen vom Betriebssystem registriert und es erfolgt eine Unterbrechung des normalen Programmablaufs. Standardmäßig werden Maßnahmen ausgeführt, die in der Regel den Abbruch des Programms zur Folge haben. Die Art des Abbruchs wird über ON-Codes dem Anwender mitgeteilt.
PL1 bietet dem Programmierer jedoch die Möglichkeit, für eine Reihe dieser Ausnahmebedingungen eigene Maßnahmen im Falle ihres Auftretens zu veranlassen, die die Fortsetzung oder einen definierten Abschluss des entsprechenden Programmteils gestatten.

Diese Bedingungen lassen sich wie folgt klassifizieren:
- Berechnungsbedingungen
- Ein- und Ausgabebedingungen
- Programmprüfungsbedingungen
- Systembedingungen

13.2. Berechnungsbedingungen

Sie können bei Berechnungen und Wertzuweisungen auftreten.

1. CONVERSION (CONV) - Konvertierungsfehler
   tritt immer bei dem Versuch auf, mit Zeichenketten unerlaubte Umwandlungen vorzunehmen
   Standardmaßnahme: Kommentar und ERROR-Bedingung

   Beispiel:

```
DCL WERT FIXED (5);
GET LIST (WERT);
```

   Die Eingabekette darf nur numerische Zeichen einschließlich Dezimalpunkt, Vor- und Leerzeichen beinhalten. Zur vereinfachten Darstellung werden hier Konvertierungsfehler durch Wertzuweisungen dargestellt.

```
WERT = '-12345';
WERT = '- 12345';          /* CONV */
WERT = '+12.345';
WERT = '+ 12.345';
WERT = '-12,345;           /* CONV */
WERT = '  1    ';
WERT = ' 1  1';            /* CONV */
WERT = '       ';          /* CONV */
```

```
WERT = '-123.00EUR';        /* CONV */
WERT = '-123.O0';           /* CONV */
```

2. FIXEDOVERFLOW (FOFL) - Festkommaüberlauf
   tritt auf, wenn die maximale Datenfeldweite in der Festkommaarithmetik überschritten wird.
   Standardmaßnahme: Kommentar und ERROR-Bedingung

Beispiel:

```
DCL WERT FIXED INIT (999),
      WERT_BIN BIN FIXED;
WERT = WERT **100;        /* FOFL, WERT > ALS 15-STELLIGE DEZ.ZAHL  */
WERT_BIN = 32768;          /* FOFL, 32768 > 2**15-1 */
```

3. OVERFLOW (OFL) - Gleitkommaüberlauf
   tritt auf, wenn der Exponent die zulässige obere Grenze überschreitet
   Standardmaßnahme: Kommentar und ERROR-Bedingung

Beispiel:

```
DCL WERT FLOAT (6) INIT (999);
      WERT = WERT **1000;        /* OFL, WERT > ALS +75 EXPONENT */
```

4. SIZE - Wertebereichüberlauf
   tritt auf, wenn einem Festkommaelement ein Wert zugewiesen wird, der die Grenze der Vereinbarung übersteigt
   Standardmaßnahme: Kommentar und ERROR-Bedingung

Beispiel:

```
DCL WERT FIXED (3) ;
      WERT = 1000;        /* SIZE, 1000 > 3 ZIFFERN */
```

5. UNDERFLOW (OFL) - Gleitkommaunterlauf
   tritt auf, wenn der Exponent die zulässige untere Grenze unterschreitet
   Standardmaßnahme: Kommentar und ERROR-Bedingung

Beispiel:

```
DCL WERT FLOAT (6) INIT (999);
      WERT = WERT **-1000;        /* OFL, WERT < ALS -78 EXPONENT */
```

6. ZERODIVIDE (ZDIV) - Versuch Division durch Null
   tritt auf, wenn Division durch Null versucht wird
   Standardmaßnahme: Kommentar und ERROR-Bedingung

Beispiel:

```
DCL WERT FIXED INIT (999),
```

```
        WERT_BIN BIN FIXED;
    WERT = WERT / 0;            /* ZDIV
```

13.3. Ein - und Ausgabebedingungen

Sie gelten entsprechend ihres Einsatzes für die reihenweise wie für die satzweise Ein - und Ausgabe.
(Beispiele zu den Ein - und Ausgabebedingungen werden aus methodischen Gründen am Ende des Kapitels angegeben)

1. ENDFILE (dateiname)
   tritt bei einem Versuch auf, einen Satz nach dem letzten zu lesen, wenn bei der Eingabe keine weiteren Datensätze vorliegen
   Standardmaßnahme: Kommentar und ERROR-Bedingung

2. ENDPAGE (dateiname)
   tritt auf, wenn mit der PUT-Anweisung hinter die letzte vereinbarte Zeile gedruckt werden soll
   Standardmaßnahme: Vorschub auf eine neue Seite

3.TRANSMIT (dateiname)
   tritt bei einem permanenten E/A- Fehler auf
   Standardmaßnahme: Kommentar und ERROR-Bedingung

4. UNDEFINEDFILE (dateiname)
   tritt bei dem Versuch ein, eine Datei zu öffnen, die nicht verfügbar bzw. noch nicht geöffnet wurde
   Standardmaßnahme: Kommentar und ERROR-Bedingung

5. NAME (dateiname)
   tritt bei der DATA-gesteuerten GET-Anweisung auf bei dem Versuch einer Bezugnahme auf eine nicht vorhandene Variable
   Standardmaßnahme: keine Maßnahme

6. KEY (dateiname)
   tritt auf, wenn bei READ - oder REWRITE Anweisung der Schlüssel nicht gefunden wurde; bei einer WRITE - oder LOCATE Anweisung existiert der Schlüssel bereits
   Standardmaßnahme: Kommentar und ERROR-Bedingung

7. RECORD (dateiname)
   tritt auf, wenn der Umfang des angesprochenen Satzes bei einer READ- oder REWRITE Anweisung nicht mit dem Umfang der Strukturvariablen übereinstimmt
   Standardmaßnahme: Kommentar und ERROR-Bedingung

13.4. Programmprüfungsbedingungen

Sie dienen vorrangig dazu, die Fehlersuche zu erleichtern.

1. SUBSCRIPTRANGE (SUBRG) - Indexüberschreitung
   tritt bei dem Versuch auf, Indexgrenzen anzuspechen, die außerhalb der definierten Grenzen liegen
   Standardmaßnahme: Kommentar und ERROR-Bedingung

   Beispiel:

```
DCL BER (100) FIXED;
DO I = 1 TO 100;
folge von anweisungen;
END;
IF BER(I) = 0 THEN ....          /* SUBRG, WEIL I = 101 IST */
```

2. CHECK (liste von bezeichnern)
   Die Angabe von CHECK (bezeichner) gibt bei jedem Durchlauf nach der DATA-gesteuerten Ausgabe den jeweils aktuellen Wert des angegeben Bezeichners an

3. STRINGRANGE (STRG)
   tritt auf, wenn mittels SUBSTR Zeichen angesprochen werden, die außerhalb der vereinbarten Kette liegen
   Standardmaßnahme: Bildung einer Teilkette, die innerhalb der Begrenzung liegt

   Beispiel:

```
DCL KETTE CHAR(8) INIT ('WARSCHAU'),
      ZKETTE CHAR(10) VARYING;
I = 2;
K = 8;
ZKETTE = SUNSTR(KETTE, I, K);      /* STRG, ZKETTE = 'RSCHAU' */
```

13.5. Systembedingungen

Das Betriebssystem setzt diese Bedingungen bei o.g. Standardmaßnahmen.

1. ERROR - Fehlerbedingung
   es erfolgt eine Unterbrechung des normalen Programmablaufs

2. FINISH - Endebedingung
   es erfolgt die Rückkehr zum Supervisor, d.h. Programmabbruch

3. AREA - Bedingung

bei Zuordnung von Speicherplatz für BASED - Variable in einem Bereich (AREA) reicht der noch im Bereich zur Verfügung stehende Speicherplatz nicht aus
Standardmaßnahme: Kommentar und ERROR-Bedingung

13.6. Wirksamkeit von Bedingungen

Neun Bedingungen kann der Programmierer an beliebigen Stellen ein- und ausschalten, man kann somit deren Wirksamkeit beliebig verwalten. Anfänglich sind fünf Bedingungen wirksam, sie können aus programmtechnischen Gründen unwirksam gemacht werden; ebenso sind vier Bedingungen anfänglich unwirksam, sie können aus programmtechnischen Gründen wirksam gemacht werden. Alle nichtgenannten Bedingungen sind immer wirksam. Die Unwirksamkeit einer dieser neun Bedingungen wird durch das Präfix NO erzielt.

Anfänglich unwirksam:

```
SIZE
STRINGRANGE
SUBSCRIPTRANGE
CHECK
```

Anfänglich wirksam:

```
OVERFLOW
UNDERFLOW
ZERODIVIDE
CONVERSION
FIXEDOVERFLOW
```

Durch ein Bedingungspräfix kann der Programmierer die Wirksamkeit oder Unwirksamkeit dieser neun Bedingungen an jedem notwendigen Programmpunkt ein- und ausschalten. Der Wirkungsbereich des Bedingungspräfix ist durch seine Stellung determiniert.

Allgemeine Form:

```
< bedingungspräfix > ::= ( < bedingung >):
```

Die Stellung des Präfixes ist ausschlaggebend für den Wirkungsbereich.

Beispiele:

```
1. (SIZE): HAUPT: PROC OPTIONS (MAIN);  /* WIRKSAMKEIT FUER GES.PROGRAMM */
2. (CHECK): UNTER: PROC;          /* WIRKSAMKEIT FÜR GES. UNTERPROGRAMM */
3. (STRINGRANGE): BEGIN;                  /* WIRKSAMKEIT FÜR GES. BEGIN-BLOCK */
4. (NOZDIV): WERT = W_1 / W_2;  /* FALLS DIVISION DURCH NULL - KEIN
PROGRAMMABBRUCH */
```

13.7. Verarbeitung von Bedingungen

Der Programmierer hat drei Möglichkeiten, die Bedingungen zu verarbeiten bzw. anzuwenden.

1. ON - Anweisung
2. REVERT - Anweisung
3. SIGNAL - Anweisung

13.7.1. Die ON - Anweisung

Mit der ON-Anweisung wird eine Behandlungsmaßnahme für den Fall festgelegt, dass die entsprechende Bedingung vom Betriebssystem aktiviert wird.

Allgemeine Form:

< behandlungsmaßnahme > ::= ON < bedingung > [SNAP] < folge von anweisungen >;

Die ON-Anweisung ist sequentiell nicht erreichbar; sie ist nur durch das Eintreten des entsprechenden Ausnahmezustandes erreichbar. Die angegebene Behandlungsmaßnahme bestimmt den weiteren Ablauf des Programms. Die ON-Anweisung aktiviert eine eigene Prolog - und Epilog-Routine.

SNAP erzeugt einen Kommentar auf die Systemausgabedatei.

Beispiel einen SNAP - Ausgabe:
IHE 806i DATA INTERRUPT IN STATEMENT 756

Die Wirksamkeit einer ON - Anweisung ist solange aktiviert, solange sie durch die Programmablauffolge nicht durch eine neue ersetzt wird.

Es bestehen vier Möglichkeiten, Behandlungsmaßnahmen einzuleiten:

1. Leere Anweisung
   Die Programmablaufsteuerung übergibt bei Eintreffen des Ausnahmezustandes die Steuerung an die ON - Anweisung und springt an die Verursacherstelle zurück. Die Standardmaßnahme wird somit reaktiviert.

   Beispiel:

```
ON ENDPAGE(SYSPRINT) ;                    /* ERZEIGT ENDLOSDRUCK */
```

2. Einfache Anweisung
   Nach Eintreten des entsprechenden Ausnahmezustandes wird die Behandlungsmaßnahme aktiviert, die einfache Anweisung abgearbeitet und an die Verursacherstelle zurückgesprungen und die folgende Anweisung abgearbeitet. Im Falle einer Sprunganweisung steuert der Programmierer selbst das weitere Vorgehen.
   Beispiel:

```
ON ENDFILE (SYSIN)  GOTO VERARBEITUNG;
ON ENDPAGE (LISTE) PUT FILE (LISTE) EDIT (' KOPFZEILE') (SKPI (3), X(5), A(20)) PAGE;
```

3. BEGIN - Block
   Mit dem BEGIN - Block besteht die Möglichkeit, Anweisungsfolgen abzuarbeiten.

   Beispiel:

```
ON ENDFILE (EING_DAT) BEGIN;
                    folge von anweisungen;
                    END;
```

4. SYSTEM - Maßnahme
   Durch die Behandlungsmaßnahme SYSTEM wird die Standardmaßnahme beim Eintreten des entsprechenden Ausnahmezustandes durchgeführt.

   Beispiel:

```
ON FIXEDOVERFLOW BEGIN;
                  folge von anweisungen;
                  END;
folge von anweisungen;
ON FIXEDOVERFLOW SYSTEM;
```

13.7.2. Die REVERT - Anweisung

Die REVERT - Anweisung hebt die Wirkung von ON - Anweisungen auf bzw. setzt sie zurück auf die Maßnahme, die im gleichen Block durchlaufen wurde.

Beispiel:

```
BEGIN;
    ON ENDPAGE ( DRUCKLISTE) GOTO M1;
    PUT FILE (DRUCKLISTE)  EDIT (' ZEILE_I') (SKIP(3), X(5), A(20));
    folge von anweisungen;
    ON ENDPAGE ( DRUCKLISTE) GOTO M2;
    PUT FILE (DRUCKLISTE)  EDIT (' ZEILE_K') (SKIP (3), X(5), A(20));
    folge von anweisungen;
    REVERT ENDPAGE ( DRUCKLISTE) ;
```

```
        PUT FILE (DRUCKLISTE)  EDIT (' ZEILE_N1') (SKIP (3), X(5), A(20)) PAGE;
        folge von anweisungen;
        M1: PUT FILE (DRUCKLISTE) EDIT (' KOPFZEILE_1') (SKIP (3), X(5), A(20)) PAGE;
        folge von anweisungen;
        M2: PUT FILE (DRUCKLISTE) EDIT (' KOPFZEILE_2') (SKIPI(3), X(5), A(20)) PAGE;
    END;
```

13.7.3. Die SIGNAL - Anweisung

Die SIGNAL - Anweisung simuliert das Auftreten eines Ausnahmezustandes einer ON - Einheit für die angegeben Bedingung. Sie dient vorrangig zum Testen dieser Bedingung.

1. Beispiel zum Testen einer ENDFILE - Bedingung:

```
ON ENDFILE (EING_DAT) BEGIN;
        behandlungsmaßnahme;
                        END;
SIGNAL ENDFILE (EING_DAT);
```

2. Beispiel für den Druck der ersten Kopfzeile:

```
ON ENDPAGE (LISTE) BEGIN;
        PUT FILE (LISTE )  EDIT (' KOPFZEILE') (SKIP (3), X(5), A(20)) PAGE;
        folge von anweisungen;
        END;
SIGNAL ENDPAGE (LISTE);
```

13.8. Bedingungsfunktionen

Sie erlauben die Untersuchungen von Unterbrechungen, die durch wirksame ON - Bedingungen entstehen.
Folgende ON - Bedingungsfunktionen ohne Argumente bestehen: (siehe auch 8.7.1.)

1. ONCHAR	stellt das Zeichen dar, welches die Unterbrechung verursacht hat
2. ONSOURCE	stellt das Feld dar, das das fehlerhafte Zeichen beinhaltet
3. ONCODE	stellt den Unterbrechungscode für die Art der Unterbrechung dar
4. ONLOC	stellt den Eingangsnamen des Programms dar, in dem die Unterbrechung auftrat
5. ONCOUNT	gibt die Anzahl der Unterbrechungen an, die von der anormalen Unterbrechung Beendigung resultieren
6. ONFILE	stellt den Namen der Datei dar, für die die Unterbrechung gesetzt wurde
7. ONKEY	stellt den Wert des Schlüssels von dem Datensatz dar, der das Setzen der Unterbrechung verursacht hat
8. DATAFIELD	fehlerhaftes oder nicht vorhandenes Datenfeld bei der DATA-gesteuerten Ein-und Ausgabe

ONCHAR UND ONSOURCE können auch als Pseudofunktionen genutzt werden, um die Ursache der Unterbrechung zu "reparieren". Diese Methode wird vorrangig zum Austesten der Programme genutzt.

Beispiel :

```
   HAUPT: PROC OPTIONS (MAIN);
          DCL (W1, W2) FIXED;
          ON CONVERSION BEGIN;
                PUT LIST (' CONVERSION-ERROR')
                      (ONCHAR, ONSOURCE, ONCODE, ONLOC, ONFILE);
                ONCHAR = '0';
          END;
          GET FILE (EING_DATEI) EDIT (W1, W2) (2 F(4));
          folge von anweisungen;
END HAUPT;
```

Eingabefeld: 123&45EU

```
    W1 nach Lesevorgang:   1230
    W2 nach Lesevorgand:   4500
```

Druckbild:

```
    CONVERSION-ERROR   &   123&   603   HAUPT   EING_DATEI
    CONVERSION-ERROR   E   450U   603   HAUPT   EING_DATEI
    CONVERSION-ERROR   U   4500   603   HAUPT   EING_DATEI
```

Der Versuch, die nächsten vier Zeichen des Eingabefeldes zu lesen und zu konvertieren, führt wegen des nicht konvertierbaren Zeichens '&' zum Konvertierungsfehler und damit
zur ON-Einheit mit der entsprechenden Behandlungsmaßnahme. Nach dem Setzen des fehlerverursachenden Zeichens '&' auf '0' wird erneut konvertiert. Analog werden die nächsten Zeichen behandelt.

14. Übersicht zur Verwendung der PL1 - Makrosprache

14.1. Allgemeine Betrachtungen

Die PL1-Makrosprache ist eine in sich abgeschlossene Sprache zur Beschreibung spezieller Zeichenkettentransformationen. Die zu transformierenden Ketten bilden zusammen mit der Beschreibung der Transformationen das PL1-Makroprogramm. Der Makrointerpreter, auch Preprozessor, ist so konzipiert, dass diese Ausgabezeichenkette ein PL1-Quellmodul sein kann, der vom Compiler verarbeitet wird. Dieser Makrointerpreter der jeweiligen Komponente verarbeitet das PL1-Makroprogramm und liefert als Ergebnis eine Zeichenkette, die keine Elemente der Transformationsbeschreibungen mehr enthält. Die Eingabekette für einen Makrointerpreter, das Makroprogramm, besteht aus einer Folge von Zeichen, das zeichenweise durchmustert und sequentiell interpretiert wird.
Die Eingabekette wird, solange keine Transformationsregeln, die durch ein vorangehendes Prozentzeichen (%) gekennzeichnet sind, erkannt werden, in unveränderter Reihenfolge in die Ausgabekette gestellt.

Allgemeine Form:

< makrobefehl > ::= % [< markenveriable >] < klausel > | < anweisung >;

Das Prozentzeichen muss vor Makroanweisungen und Makroklauseln stehen, denn sie geben an, dass es sich um spezielle Elemente der Makrosprache handelt.

Mit Hilfe der Makroprogrammsätze kann der Makrotext geändert werden, indem

- ausgewählte Textteile durch andere ersetzt werden,
- Teile ausgewählt werden und
- er durch Teile aus Bibliotheken erweitert wird.

Die Verarbeitung durch den Makrointerpreter geschieht vor der eigentlichen PL1-Übersetzung in einer gesonderten Arbeitsstufe.

14.2. Möglichkeiten der Anwendung der PL/1 Makrosprache

Eine sinnvolle Anwendung der Makrosprache findet in der Modifizierung von PL/1 Quelltexten statt. Besteht die Notwendigkeit, Programmtexte auf Grund ständig veränderter Anwendungsanforderungen zu modifizieren bzw. neue Programmtexte einzufügen, bietet PL/1 die Nutzung der Makrosprache an. Durch Anwendung dieser Makrosprache können Teile des Programmtextes durch andere Zeichenfolgen

ersetzt und damit beispielsweise neue Anweisungen vollständig oder durch neue Anweisungen ergänzt werden.
Von größerer Bedeutung ist jedoch seine Fähigkeit, extern gespeicherte Quelltexte in ein Programm einfügen zu können. Es ergibt sich daraus die Möglichkeit, modulare Programmbausteine zusammenzusetzten, ohne das durch ständige Aktivierungen von Funktionen oder Unterprogrammen die Effektivität der Abarbeitung beeinträchtigt wird. Eine weitere, sehr nützliche Anwendung besteht in der Möglichkeit, auf der Grundlage der Makrosprache spezielle Anwendersprachen zu entwickeln, die auf fachgebietsspezifischen Problemen ausgerichtet sind oder zur Bearbeitung spezieller abgegrenzter Aufgabenkomplexe dienen. Allerdings sind im Rahmen dieses Skipts umfassende Darstellungen dieser Nutzungsmöglichkeiten der PL/1 Makrosprache nicht möglich; es soll nur auf diese Möglichkeiten der Nutzung hinweisen. In diesem Skript ist die hier vorliegende Beschreibung der Makrosprache nur als Übersicht zu verstehen.

14.3. Möglichkeiten für die Veränderung von PL/1 - Programmtexten

Grundlage für die Veränderung von Programmteilen bildet die Vereinbarung von Makrovariablen.

Allgemeine Form:

< makro-vereinbarung >::= %{ DELCLARE / DCL } < bezeichner > < attribute > ;

Für Bezeichner und Attribute der Makrovereinbarungen gelten alle gemachten Angaben in Punkt 2. mit der Einschränkung, dass für Attribute nur CHARACTER und FIXED ohne Genauigkeitsattribute zur Verfügung stehen. Zeichenketten sind immer Ketten variabler Länge, deren maximale Länge nicht vereinbart wird. Die dezimalen Festkommagrößen haben stets das Genauigkeitsattribut (5,0).

Beispiele:

```
%DCL KETTE CHARACTER;
%DCL (WERT_1; WERT_2, WERT_3) FIXED;
%DCL (KET_1, KET_2) CHAR;
```

Die Zuordnung eines konkreten Wertes zu einer Makrovariablen ist nur durch eine Makrozuordnungsanweisung möglich.

Allgemeine Form:

< makro - anweisung > ::= %< makro - variable> = < makro - ausdruck > ;

Als Operanden eines Makroausdruckes können Konstanten oder Makrovariable verwendet werden. Ihre Verknüpfung kann in sämtlichen in PL1 gültigen Operationen – außer Potenzieren – ausgeführt werden. Konstanten dürfen in Zeichen – bzw. Bitkettenkonstanten sowie in ganzzahligen dezimalen Festkommakonstanten mit maximal 5 Ziffern auftreten. Die Berechnung des Wertes für den angegebenen

Ausdruck erfolgt nach den üblichen Regeln; eventuelle Datenumwandlungen finden nach den allgemeinen Konvertierungsregeln statt (siehe 2.7. und 3.).
Vor der Ausführung von arithmetischen Operationen werden sämtliche Operanden in DECIMAL FIXED (5,0) umgewandelt. Die Umwandlung einer Festkommagröße in eine Zeichenkette erfolgt nach der LIST – Steuerung; die Ergebniskette hat somit die Gesamtlänge von 8 Byte. Ist die Makrovariable vom Typ CHARACTER, so wird ihre Bezeichnung durch eine Zeichenfolge ersetzt, die sich aus ihrem Zeichenkettenwert zuzüglich eines Leerzeichens am Beginn und am Ende des Textes ergibt.

Beispiele:

```
%WERT_1 = 0;
%WERT_2 = 99999;
%KET_1 = 'A + B';
```

Beispiel 1 zur Bearbeitung einer Befehlsfolge:

```
%DCL (KET1, KET2) CHAR;
%KET2 = '99+K';
%KET1 = 'KET2-99';
IND=MAX+KET1;
```

Durch den Makrointerpreter wird folgende Anweisung in den Ausgabetext generiert: Zunächst wird in der Ergibtanweisung IND die Bezeichnung KET1 durch den Zeichenkettenwert der gleichnamigen Makrovariablen ersetzt und die Zeichenfolge

```
IND=MAX+␣KET2– 99␣;
```

gebildet. Die darin enthaltene Bezeichnung KET2 wird anschließend ebenfalls ersetzt und in den Ausgabetext wird die Anweisung

```
IND=MAX+␣␣99 + K␣-99␣;
```

generiert.

Beispiel 2 zur Bearbeitung einer Befehlsfolge:

```
%DCL (ORG,DATEI) CHAR;
%ORG = 'REGIONAL(1)‘;
%DATEI = 'SCHULDNERDATEI‘;
DCL DATEI FILE RECORD INPUT ENV(ORG);
```

Durch den Makrointerpreter wird folgende Anweisung in den Ausgabetext generiert:

```
DCL  SCHULDNERDATEI  FILE RECORD INPUT ENV( REGIONAL(1));
```

Im weiteren Verlauf könnten andere Makroanweisungen diese DCL-Anweisung modifizieren. Sollte jedoch keine entsprechende Ersetzung mehr bewirkt werden, ist dafür die Makroanweisung

```
< deaktivierung > ::= % {DEACTIVE | DEACT} < liste von makrobezeichnern >;
```

anzugeben; sollte dagegen im weiteren Verlauf erneut die Modifizierung desTextes wirksam werden, so ist dafür die Makroanweisung

```
< aktivierung > ::= % {ACTIVE | ACT} < liste von makrobezeichnern >;
```

anzugeben.
Neben den Makrovariablen besteht auch die Möglichkeit, Funktionen in der Makrosprache zu vereinbaren, die nur während der Interpretation durch den Makrointerpreter angesprochen werden können.

Allgemeine Form:

```
< makro-funktion > ::= %DCL < funktionsname > ENTRY (< parameterliste >)
                                                RETURNS ( { FIXED | CHAR } );
```

Beispiel:

```
% DCL  FUNKT_1 ENTRY (CHAR, CHAR) RETURNS (CHAR),
       FUNKT_2 ENTRY (,,FIXED,) RETURNS (FIXED);
```

Es werden zwei Makrofunktionen vereinbart. FUNKT_1 enthält zwei Parameter vom Typ CHARACTER, der Funktionswert ist ebenfalls vom Typ CHARACTER. FUNKT_2 enthält vier Parameter, über deren ersten, zweiten und vierten Typ keine Aussagen gemacht werden, der dritte Parameter ist vom Typ FIXED, der Funktionswert ist eine dezimale Festkommagröße.

Das Makrofunktionsprogramm selbst hat folgende
Allgemeine Form:

```
< makro-funktionsprogramm > ::=
    %< markenveriable >:  PROC (< parameterliste >) RETURNS ( { FIXED | CHAR } );
    folge von makro-anweisungen;
    %END < markenveriable >;

< makro-aufruf > ::= RETURN (< makro-funktionsprogramm >);
```

Alle zwischen %PROC und %END befindlichen Anweisungen müssen Makroanweisungen sein; allerdings ohne Notation des %-Zeichens.
Im Unterschied zur Behandlung der sonstigen Makroanweisungen bewirkt das Auftreten eines Makrofunktionsprogramms während der Textbearbeitung durch den Makrointerpreter nicht dessen unmittelbare Ausführung. Dazu ist stets innerhalb einer Makroanweisung ein entsprechender Funktionsaufruf erforderlich. Auf Grund eines solchen Aufrufs werden die Anweisungen des Makrofunktionsprogramms ausgeführt und der Funktionswert dann in dem betreffenden Makroanweisung weiter verwendet. Als Argumente sind jeweils Makroausdrücke anzugeben.
Tritt die Bezeichnung eines Makrofunktionsprogramms innerhalb der Textteile auf, die keine Makroanweisungen darstellen, so wird ebenfalls ein Aufruf bewirkt, und der Funktionswert ersetzt diese Bezeichnung in der gleichen Weise wie eine Makrovariable. Die angegebenen Argumente werden jeweils einschließlich sämtlicher Leerzeichen als Zeichenkettenkonstanten betrachtet, sind aber nicht in Apostrophe

einzuschließen. Diese ersetzende Wirkung eines Makrofunktionsprogramms kann durch die Anweisung %DEAC aufgehoben und durch %ACT wieder aktiviert werden. Ein Aufruf der Makrofunktion innerhalb einer anderen Makroanweisung wird davon jedoch nicht berührt.
Beispiel siehe 14.4.

14.4. Möglichkeiten für die Zusammenstellung von PL/1 - Programmtexten

Die Interpretation durch den Makrointerpreter geschieht in der Regel am Anfang der vorliegenden Zeichenkette und wird sequentiell bis zum Ende fortgesetzt. Durch die Angabe verschiedener Makroanweisungen innerhalb der Zeichenkette besteht die Möglichkeit, extern gespeicherte Textfolgen einzufügen oder das sequentielle Interpretieren durch abweichende Abarbeitungsfolgen zu veranlassen.

14.4.1. Makroanweisungen zum Einfügen von PL/1 - Programmtexten

Der einzufügende Text muss Bestandteil einer untergliederten Datei, ein Member einer Bibliothek sein. Um einen als Teilbestand gespeicherten Text einfügen zu können, ist folgende Makroanweisung notwendig:

< textaufruf aus bibliothek> ::= %INCLUDE < bibliotheksname > (< liste von membernamen >);

Sollte der Member Bestanteil der Systembibliothek SYSLIB sein, ergibt sich folgende **Allgemeine Form:**

< textaufruf aus bibliothek> ::= %INCLUDE < liste von membernamen >;

Sollten mehrere, gesondert gespeicherte Texte eingefügt werden, können die einzelnen Membernamen, durch Komma getrennt, hintereinander angegeben werden.

Beispiel zum Einfügen einer umfangreichen Strukturdefinition:

```
PROGR: PROC OPTIONS (MAIN);
%DCL STRUKTUR CHAR;
%STRUKTUR = 'EINGABE_SATZ';
%INCLUDE BIBL_01 (STRUKT_1);
```

Der generierte Quelltext könnte sich folgt darstellen:

```
PROGR: PROC OPTIONS (MAIN);
DCL 1  EINGABE_SATZ,          /* STRUKTURDEFINITION IN DER BIBL_01(STRUKT_1) */
       2  UNTERSTR_1,
          3  KET_1 CHAR(25),
          3  WERT_1 FIXED(7,1),
       2  UNTERSTR_2,
          3  FELD_1(10,5) FIXED(5,2);
          usw;
```

14.4.2. Makro - Sprunganweisungen

Durch Angabe einer Sprunganweisung in der Zeichenkette setzt der Makrointerpreter mit der Bearbeitung an der angegebenen Marke seine Interpretation fort. Ein Makroprogramm darf nicht mittels einer GOTO-Anweisung verlassen werden.

Allgemeine Form:

```
< makro-sprunganweisung > ::= %GOTO < markenvariable >;
```

Beispiel:

```
W1 = W2 * W3;
%GOTO MARK1;
IF W1 <  0 THEN
      DO; folge von anweisungen; END;
%MARK1: ;                                   /* MAKRO- LEERANWEISUNG */
ERG = (W4 + W5) / W1;
```

Mit der Ausführung der Makro-Sprunganweisung %GOTO MARK1 durch den Makrointerpreter zu der mit der Marke MARK1 versehenen Makro-Leeranweisung wird bewirkt, dass die angegeben Behandlung für den Fall eines negativen Wertes von W1 nicht mit in den Ausgabetext übernommen wird; generiert wird dann lediglich die Anweisungsfolge:

```
W1 = W2 * W3;
ERG = (W4 + W5) / W1;
```

14.4.3. Makroanweisungen für bedingte Verzweigungen

Mittels bedingter Makro-Sprunganweisungen nimmt der Makrointerpreter die weitere Bearbeitung des Textes in Abhängigkeit vom Wert eines Makroausdrucks vor.
Die Regeln und die Möglichkeiten der Nutzung einer bedingten Sprunganweisung wurden in 6.1.3. ausführlich besprochen.

Allgemeine Form:

```
< bedingte makroverzweigung > ::= %IF < makroausdruck > %THEN < makroanweisung_1 >
                                                    [%ELSE < makroanweisung_2 >];
```

Selbstverständlich können die beiden Makroanweisungen im THEN- und ELSE-Zweig sich aus einer Folge von Anweisungen zusammensetzen; nur dann müssen %DO und %END benutzt werden.

Beispiel:

Das Beispiel in 14.4.1. soll so abgeändert werden, dass bei negativem Wert der Variablen W1 der unbedingt ausgelassenen Textteil nur dann in den Ausgabetext übertragen werden soll, wenn eine Makrovariable SCHALTER den Wert ‘1‘ hat.

```
W1 = W2 * W3;
%IF SCHALTER = '1'  %THEN %DO; IF W1 <  0 THEN
                                                DO; folge von anweisungen END;
                                %END;
ERG = (W4 + W5) / W1;
```

14.4.4. Makro DO-Anweisungen

Die DO-Klausel als Bestandteil einer Gruppe dient dem Zusammenfassen von Makroprogrammsätzen und Makrotext zwecks ein- oder mehrmaliger Ausführung. Durch Angabe einer Makro DO-Anweisung in einer Zeichenkette kann der Makrointerpreter den gleichen Text mehrfach in den Ausgabetext einfügen, wobei normalerweise eine Modifizierung in der Form stattfindet, dass bestimmte Textteile jeweils durch den Wert der Laufvariablen ersetzt werden.
Die Regeln und die Möglichkeiten der Nutzung einer DO-Anweisung wurden in 6.2. ausführlich besprochen.

Allgemeine Form:

```
< makro  do-anweisung > ::=
%DO < makrovariable > = < ausdruck_1 > TO < ausdruck_2 > BY < ausdruck_3 >;
folge von makro-anweisungen;
%END;
```

Beispiel:

```
%DCL (I, K) FIXED;
%K = 3;
%DO; I=1 TO K;
VEK1(I) = VEK2(I) + VEK3(I);
%END;
```

In den Ausgabetext werden folgende Anweisungen generiert:

```
VEK1( 1 ) = VEK2( 1 ) + VEK3( 1 );
VEK1( 2 ) = VEK2( 2 ) + VEK3( 2 );
VEK1( 3 ) = VEK2( 3 ) + VEK3( 3 );
```

15. Quellenangaben

1. Henning Schoch
   Programmieren in PL1
   Akademie-Verlag Berlin 1987

2. B.Higman
   Programiersprachen – eine vergleichende Studie
   BSB B.G. Teubner Verlagsgesellschaft

3. Robotron-Projekt Dresden
   PL1 Sprachbeschreibung
   Teil 1 und Teil 2

4. IBM Deutschland
   Einführung in die PL/1-Sprache – eine programmierte Unterweisung
   DP Schulungsentwicklung

5. IBM Deutschland
   Übungs- und Aufgabensammlung
   Schule für Datenverarbeitung

6. MR Leipzig
   Übungs- und Aufgabensammlung
   Schulungszentrum Leipzig

7. Günter Riedewald
   Formale Beschreibung von Programmiersprachen
   Eine Einführung in die Semantik

8. Meine persönlichen Schulungsunterlagen
   Merkblätter, Seminarunterlagen, Aufgabensammlungen