

Informationstechnik
Programmiersprache PEARL
PEARL 90

DIN
66253-2

ICS 35.060

Ersatz für
Ausgabe 1982-10

Deskriptoren: Programmiersprache, PEARL, Informationstechnik

Information technology – Programming language PEARL – PEARL 90

Technologies de l'information – Langage de programmation PEARL – PEARL 90

Inhalt

	Seite
Vorwort	3
1 Anwendungsbereich	3
2 Normative Verweisungen	3
3 Bezeichnung	3
4 Regeln zum Aufbau von PEARL-Sprachformen	4
4.1 Zeichensatz	4
4.2 Grundelemente	5
4.2.1 Bezeichner	6
4.2.2 Konstanten	6
4.2.3 Kommentare	6
4.3 Aufbau von Sprachformen	7
5 Programm	9
5.1 Module	9
5.2 Vereinbarungen	12
5.2.1 Deklaration	12
5.2.2 Spezifikation (SPC) und Identifikation (SPC IDENT)	14
5.3 Blöcke	15
5.4 Bezüge zwischen Modulen	18
6 Variablen und Konstanten	20
6.1 Vereinbarung von Variablen (DCL)	20
6.2 Ganze Zahlen (FIXED)	21
6.3 Gleitpunktzahlen (FLOAT)	23
6.4 Bitketten (BIT)	24
6.5 Zeichenketten (CHARACTER)	26
6.6 Die Längenvereinbarung	28
6.7 Uhrzeiten (CLOCK)	29
6.8 Zeitdauern (DURATION)	30
6.9 Referenzen (REF)	31
6.10 Felder	35
6.11 Strukturen	38
6.12 Typvereinbarung (TYPE)	42
6.13 Das Initialisierungsattribut (INITIAL)	45
6.14 Zuweisungsschutz (INV)	47
6.15 Konstante Ausdrücke	49

Fortsetzung Seite 2 bis 182

Normenausschuß Informationstechnik (NI) im DIN Deutsches Institut für Normung e. V.

7	Ausdrücke und Zuweisungen	51
7.1	Ausdrücke	51
7.2	Operatorenvereinbarung (OPERATOR)	62
7.3	Zuweisungen	64
7.3.1	Zuweisungen für skalare Variablen	64
7.3.2	Zuweisungen für Strukturen	65
7.4	Überlagerung von Datenstrukturen	65
7.4.1	Der "BY TYPE"-Operator	66
7.4.2	Der "VOID"-Datentyp	67
8	Anweisungen zur Steuerung des sequentiellen Ablaufs	69
8.1	Bedingte Anweisung (IF)	70
8.2	Anweisungsauswahl (CASE) und Leeraanweisung	71
8.3	Wiederholung (FOR - REPEAT)	74
8.4	Sprung-Anweisung (GOTO)	77
8.5	Exit-Anweisung (EXIT)	78
9	Prozeduren	79
9.1	Vereinbarung von Prozeduren (PROC)	79
9.2	Aufruf von Prozeduren (CALL)	83
9.3	Referenzen auf Prozeduren (REF PROC)	86
10	Parallele Aktivitäten	88
10.1	Vereinbarung von Tasks (TASK)	88
10.2	Anweisungen zur Steuerung von Tasks	90
10.2.1	Starten einer Task (ACTIVATE)	90
10.2.2	Beenden einer Task (TERMINATE)	93
10.2.3	Anhalten einer Task (SUSPEND)	94
10.2.4	Fortsetzen einer Task (CONTINUE)	94
10.2.5	Verzögern einer Task (RESUME)	96
10.2.6	Ausplanen einer Task (PREVENT)	97
10.3	Synchronisierung von Tasks	98
10.3.1	Semaphor-Variablen (SEMA) und -Anweisungen (REQUEST, RELEASE, TRY)	98
10.3.2	Bolt-Variablen (BOLT) und -Anweisungen (ENTER, LEAVE, RESERVE, FREE)	104
10.4	Interrupts und Interrupt-Anweisungen	107
10.4.1	Vereinbarung von Interrupts und Software-Interrupts	107
10.4.2	Interrupt-Anweisungen (TRIGGER, ENABLE, DISABLE)	109
10.5	Referenzen auf Tasks (REF TASK)	111
11	Eingabe und Ausgabe	113
11.1	Systemteil	113
11.2	Vereinbarung von Datenstationen (DATION) im Problemteil	116
11.2.1	System-Datenstationen	116
11.2.2	Benutzerdefinierte Datenstationen	117
11.3	Öffnen und Schließen von Datenstationen (OPEN, CLOSE)	122
11.4	Die Read- und die Write-Anweisung (READ, WRITE)	125
11.5	Die Get- und Put-Anweisung (GET, PUT)	132
11.5.1	Das Fixed-Format (F)	135
11.5.2	Das Float-Format (E)	137
11.5.3	Die Zeichenketten-Formate (A) und (S)	139
11.5.4	Das Bit-Format (B)	141
11.5.5	Das Zeit-Format (T)	143
11.5.6	Das Dauer-Format (D)	144
11.5.7	Das List-Format (LIST)	145
11.5.8	Das R-Format (R)	146
11.6	Die Convert-Anweisung (CONVERT)	147
11.7	Fehlerbehandlung in E/A-Anweisungen (RST)	148
11.8	Die Take- und die Send-Anweisung	149
12	Signale	150
12.1	Vereinbarung von Signalen	150
12.2	Signal-Einplanung (ON), Signal-Anweisungen (INDUCE)	152

Anhang A (normativ)	Vordefinierte Funktionen	156
Anhang B (normativ)	Syntax-Zusammenfassung	160
Anhang C (normativ)	Liste der Schlüsselwörter mit Kurzformen	174
Anhang D (normativ)	Sonstige Wortsymbole in PEARL	176
Anhang E (informativ)	Allgemeines zu PEARL	178
Anhang F (informativ)	Stichwortverzeichnis	181

Vorwort

Auf Basis der Erfahrungen aus Hunderten von PEARL-Projekten erarbeitete in den Jahren 1989 und 1990 ein Gremium aus PEARL-Anwendern und -Implementierern die Definition von PEARL 90 (vgl. Stieger, K: PEARL 90 – Die Weiterentwicklung von PEARL. In: Informatik Fachberichte 231, PEARL '89 Workshop über Realzeitsysteme, Springer 1989).

Der Sprachreport wurde dem zuständigen Arbeitsausschuß NI-22 "Programmiersprachen" als Normvorschlag unterbreitet und nach formaler Umgestaltung in der nun vorliegenden Form veröffentlicht.

Zur Normen-Reihe DIN 66253 gehören die folgenden Teile:

- die vorliegende Norm (PEARL 90);
- DIN 66253-3 (Mehrrechner-PEARL).

Die Vornorm DIN 66253-1 : 1981-07 (Basic PEARL) wurde im März 1986 zurückgezogen.

Änderungen

Gegenüber der Ausgabe Oktober 1982 wurden folgende Änderungen vorgenommen:

- PEARL 90 entspricht Full PEARL, wobei einige in der Praxis nicht benötigte Sprachelemente entfallen sind. Ferner enthält PEARL 90 fortschrittliche Erweiterungen.

Frühere Ausgaben

DIN 66253-2: 1982-10

1 Anwendungsbereich

PEARL steht für **P**rocess and **E**xperiment **A**utomation **R**ealtime **L**anguage und ist eine höhere Programmiersprache, die eine komfortable, sichere und weitgehend rechnerunabhängige Programmierung von Multitasking- und Echtzeit-Aufgaben erlaubt.

Die Norm beschreibt den Sprachumfang von PEARL 90. Sie stellt damit die Grundlage für das Herstellen von Übersetzern und das Ausarbeiten von Anleitungen (Handbücher, Lehrbücher) zur Programmierung in PEARL 90 dar.

2 Normative Verweisungen

Diese Norm enthält durch datierte oder undatierte Verweisungen Festlegungen aus anderen Publikationen. Diese normativen Verweisungen sind an den jeweiligen Stellen im Text zitiert, und die Publikationen sind nachstehend aufgeführt. Bei datierten Verweisungen gehören spätere Änderungen oder Überarbeitungen dieser Publikationen nur zu dieser Norm, falls sie durch Änderung oder Überarbeitung eingearbeitet sind. Bei undatierten Verweisungen gilt die letzte Ausgabe der in Bezug genommenen Publikation.

DIN 66003

Informationsverarbeitung – 7-Bit-Code

DIN 66253-3

Informationsverarbeitung – Programmiersprache PEARL – Mehrrechner-PEARL

ISO/IEC 646 : 1991

Information technology – ISO 7-bit coded character set for information interchange

3 Bezeichnung

Bezeichnung der Programmiersprache PEARL 90, die den Bedingungen dieser Norm entspricht:

Programmiersprache DIN 66253 – PEARL 90

4 Regeln zum Aufbau von PEARL-Sprachformen

Ein PEARL-Programm kann formatfrei geschrieben werden; es muß insbesondere nicht darauf geachtet werden, daß eine Anweisung in einer bestimmten Spalte beginnt.

Alle Elemente eines PEARL-Programms werden aus Zeichen des Zeichensatzes nach 4.1 erstellt. Allerdings dürfen Zeichenkettenkonstanten und Kommentare jedes Zeichen enthalten, das ISO/IEC 646:1991 (siehe DIN 66003) und nationale Varianten von ISO/IEC 646 zulassen.

4.1 Zeichensatz

Der Zeichensatz von PEARL ist eine Teilmenge des ISO 7-Bit-Zeichensatzes nach ISO/IEC 646:1991 (siehe DIN 66003). Er enthält

- a) die Großbuchstaben A bis Z
- b) die Kleinbuchstaben a bis z
- c) die Ziffern 0 bis 9 und
- d) die Sonderzeichen
 - Unterstrich
 - Zwischenraum (zur Verdeutlichung wird manchmal auch der Unterstrich _ benutzt)
 - ! Ausrufungszeichen (Beginn des Zeilenkommentars)
 - ' Apostroph
 - (runde Klammer auf
 -) runde Klammer zu
 - , Komma
 - . Punkt
 - ; Semikolon
 - : Doppelpunkt
 - + Pluszeichen (z.B. für Addition, Vorzeichen)
 - Minuszeichen (z.B. für Subtraktion, Vorzeichen)
 - * Stern (z.B. für Multiplikation)
 - / Schrägstrich (z.B. für Division)
 - = Gleichheitszeichen (z.B. für Zuweisung)
 - < Kleiner-Zeichen
 - > Größer-Zeichen
 - [Strukturklammerungszeichen auf
 -] Strukturklammerungszeichen zu
 - \ Rückschrägstrich (für Steuerzeichen in Zeichenketten)

Die folgenden Zeichenkombinationen werden jeweils als eine Einheit (zusammengesetztes Symbol) interpretiert:

<code>:=</code>	Zuweisungssymbol
<code>**</code>	Exponentiationssymbol
<code>/*</code>	Kommentar-Beginn
<code>*/</code>	Kommentar-Ende
<code>//</code>	Symbol für ganzzahlige Division
<code>==</code>	Gleich-Symbol
<code>/=</code>	Ungleich Symbol
<code><=</code>	Kleiner-Gleich-Symbol
<code>>=</code>	Größer-Gleich-Symbol
<code><></code>	Cyclic-Shift-Symbol
<code>><</code>	Verkettungssymbol
<code>'\</code>	Einleitung einer Steuerzeichensequenz in Zeichenkettenkonstanten
<code>'</code>	Ende einer Steuerzeichensequenz in Zeichenkettenkonstanten

Falls auf dem Gerät für die Programmniederschrift nicht alle Symbole zur Verfügung stehen, können folgende Zeichenfolgen alternativ benutzt werden:

LT	für <
GT	für >
EQ	für ==
NE	für /=
LE	für <=
GE	für >=
CSHIFT	für <>
CAT	für ><
(/	für [
/)	für]

4.2 Grundelemente

Ein PEARL-Programm wird aus folgenden Grundelementen aufgebaut:

- Bezeichner
- Konstanten
- Trennzeichen (das sind Sonderzeichen und zusammengesetzte Symbole)
und
- Kommentare

Auf die Zeichenfolge für Bezeichner und Konstanten müssen Trennzeichen oder Kommentare folgen.

4.2.1 Bezeichner

Bezeichner werden zur Bildung von Namen für Objekte (z.B. Zahlenvariablen, Prozeduren) benutzt. Sie bestehen aus einer Folge von Buchstaben (groß und klein), dem Unterstrich und/oder Ziffern; diese Folge muß mit einem Buchstaben beginnen. PEARL unterscheidet bei Bezeichnern zwischen großen und kleinen Buchstaben, d.h. ventil, VENTIL und Ventil bezeichnen verschiedene Objekte.

Beispiel: Zaehler_1, DISPO, warten

Einige Wörter haben an festgelegten Stellen im PEARL-Programm eine spezifische Bedeutung; diese Wörter werden Schlüsselwörter genannt. Z.B. sind die Wörter BIT oder GOTO solche Schlüsselwörter. Der Anhang C enthält eine Liste aller Schlüsselwörter. Sie dürfen nicht als Bezeichner benutzt werden und müssen immer in Großbuchstaben geschrieben sein.

4.2.2 Konstanten

Konstanten sind ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern. Sie werden zusammen mit den entsprechenden Variablen in Abschnitt 6 beschrieben.

4.2.3 Kommentare

Kommentare dienen zur Erläuterung des Programms und sind ohne Bedeutung für den Programmablauf. Es gibt zwei Arten von Kommentaren: Die eine Art kann sich über mehrere Zeilen erstrecken und wird durch die zusammengesetzten Symbole `/*` und `*/` eingeklammert. Innerhalb dieser Klammern können beliebige Zeichen erscheinen mit Ausnahme des zusammengesetzten Symbols `*/` für Kommentar-ende.

Die andere Art, der Zeilenkommentar, beginnt mit einem Ausrufungszeichen `!` und wird durch das Zeilenende beendet.

Kommentare dürfen überall eingefügt werden, wo Leerzeichen zulässig sind.

Beispiel:

<code>/*</code>	Dieser Kommentar wird durch das Zeilenende nicht beendet <code>*/</code>
<code>!</code>	Dieser Kommentar ist auf 1 Zeile begrenzt

4.3 Aufbau von Sprachformen

In den nachfolgenden Abschnitten werden die in PEARL zulässigen Sprachformen beschrieben. Damit diese Beschreibungen genau und möglichst kompakt sind, werden über die verbale Formulierung hinaus einige formale Möglichkeiten benötigt:

Jede Sprachform hat einen Namen, unter dem sie mit Hilfe des (Meta-)Symbols “::=” definiert wird:

Name-der-Sprachform ::=
Definition-der-Sprachform

Beispiel 1:

Großbuchstabe ::=
A oder B oder . . . oder Z

Ziffer ::=
0 oder 1 oder . . . oder 9

Wie dieses Beispiel zeigt, kann die Definition einer Sprachform Elemente enthalten, die beim Aufbau der Sprachform alternativ zueinander angegeben werden. Zur Abkürzung werden die alternativ möglichen Angaben künftig durch das Symbol “|” getrennt:

Beispiel 2:

Buchstabe ::=
A | B | . . . | Z | a | b | . . . | z

Ziffer ::=
0 | 1 | 2 | . . . | 9

Soll ein Element beliebig oft, aber mindestens einmal, auftreten dürfen, wird es oben mit drei Punkten versehen.

Beispiel 3:

einfache-ganze-Zahl ::=
Ziffer ***

Um auszudrücken, daß ein Element beim Aufbau der Sprachform optional ist, wird es in eckigen Klammern “[...]” angegeben (Im Zweifelsfall werden die Strukturklammerungszeichen fett gedruckt).

Beispiel 4:

```
Bezeichner ::=  
    Buchstabe [ { Buchstabe | Ziffer | _ } ... ]
```

Hier wurden bereits zwei weitere Regeln benutzt: Die Definition einer Sprachform darf wieder Namen von Sprachformen enthalten; außerdem werden die geschweiften und eckigen Klammern auch dazu benutzt, Elemente zu neuen Elementen zusammenzufassen. So ist das letzte Beispiel dem folgenden äquivalent:

Beispiel 5:

```
Bezeichner ::=  
    Buchstabe [ Buchstabe | Ziffer | _ ] ...
```

Zur Beschreibung von Listen, deren Elemente durch ein bestimmtes Symbol getrennt sind, werden bei der Definition der entsprechenden Sprachform das Listenelement und das Trennsymbol in der Form

```
Listenelement [ Trennsymbol Listenelement ] ...
```

angegeben.

Beispiel 6:

```
Bezeichnerliste ::=  
    Bezeichner [ , Bezeichner ] ...
```

Zum besseren Verständnis oder zur genaueren Beschreibung der Definition einer Sprachform werden häufig Elemente mit einem erläuternden oder einschränkenden Kommentar versehen, der von dem Element durch das Symbol “\$” getrennt wird.

Beispiel 7:

```
Geräteliste ::=  
    Bezeichner$Gerät [ , Bezeichner$Gerät ] ...
```

5 Programm

Syntax:

Die allgemeine Form eines PEARL-Programms lautet:

```
PEARL-Programm ::=  
    Modul ...
```

Semantik:

Ein PEARL-Programm setzt sich aus einem oder mehreren Teilen, sogenannte Module, zusammen, die unabhängig übersetzt werden.

5.1 Module

Syntax:

```
Modul ::=  
    MODULE [ (Bezeichner§des-Moduls) ] ;  
        { Systemteil [ Problemteil ] } | Problemteil  
    MODEND ;  
  
Systemteil ::=  
    SYSTEM ; [ Benutzernamen-Vereinbarung ... ]  
  
Problemteil ::=  
    PROBLEM ; [ { Deklaration | Spezifikation | Identifikation } ... ]
```

Semantik:

Jeder Modul besteht aus einem Systemteil und/oder einem Problemteil.

Erläuterungen:

Im Systemteil werden die Verbindungen des projektierten Rechners mit den Elementen des technischen Prozesses (Meßwertgeber, Stellglieder, usw.) und der Standardperipherie (Tastatur, Monitor, Drucker, Platten, Bänder usw.) beschrieben. Dabei kann der Programmierer z. B. den Eingängen eines Interruptwerks und den im Problemteil in E/A-Anweisungen angesprochenen Peripherieelementen frei wählbare Namen zuordnen, um diese rechnerunabhängigen Namen im Problemteil verwenden zu können.

Im Problemteil wird der Algorithmus zur Lösung der gestellten Automationsaufgabe beschrieben. Hierzu kann der Programmierer folgende Objekte durch Deklaration, Spezifikation oder Identifikation vereinbaren:

- Variablen und Konstanten für ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Zeitdauern, Uhrzeiten, Referenzen
- Sprungmarken
- Prozeduren für mehrfach auftretende Teilaufgaben
- Tasks für die zeitliche parallele Abarbeitung von Aufgaben
- Blöcke für die Strukturierung von Prozeduren und Tasks
- Interrupts
- Signale
- Synchronisierungsvariablen (Sema- und Bolt-Variablen) sowie
- Datenstationen und Formate für die Ein/Ausgabe

Die erforderlichen Anweisungen werden in den Prozeduren und Tasks zusammen mit weiteren "lokalen" Vereinbarungen, die nur dort benötigt werden, angegeben. Dabei gilt allgemein, daß Objekte erst dann (in Anweisungen) benutzt werden dürfen, wenn sie vereinbart worden sind.

Objekte werden auf Modulebene, d.h. außerhalb von Prozeduren und Tasks, oder auf Prozedur-, Task- oder Blockebene vereinbart. Auf Modulebene vereinbarte Objekte sind im ganzen Modul bekannt und können von jeder Task und Prozedur des Moduls unter Angabe des Bezeichners benutzt oder gegebenenfalls geändert werden. Ein auf Task-, Prozedur- oder Blockebene vereinbartes Objekt ist nur in der betreffenden Task, Prozedur bzw. dem betreffenden Block bekannt und kann nur dort benutzt oder gegebenenfalls geändert werden.

Beispiel (Modul-Struktur):

MODULE ;

SYSTEM ;

Beschreibung der Verbindungen und Vereinbarung von
Namen für die Peripherieelemente

PROBLEM ;

Vereinbarung von Konstanten und Variablen
Vereinbarung von Interrupts
Vereinbarung von Datenstationen

Vereinbarung einer Task

Deklaration lokaler Konstanten und Variablen
Deklaration lokaler Prozeduren
Anweisungen

Vereinbarung einer Prozedur

Deklaration lokaler Konstanten und Variablen
Deklaration lokaler Prozeduren
Anweisungen

...

MODEND ;

Nach dem Laden eines PEARL-Programms startet das PEARL-Laufzeitsystem automatisch alle Tasks, die mit dem Attribut MAIN gekennzeichnet sind (sie werden lauffähig). Alle mit MAIN versehenen Tasks müssen im selben Modul deklariert sein.

Beispiel (Attribut MAIN):

MODULE (Main) ;

SYSTEM ; ...

PROBLEM ;

Start : **TASK MAIN ;**
! Einplanungen anderer Tasks
END ; ! Start

Messen : **TASK PRIO 1 ;**

! Taskkörper
END ; ! Messen

...

MODEND ;

Nach dem Laden wird die Task Start automatisch gestartet.

5.2 Vereinbarungen

5.2.1 Deklaration

Syntax:

```
Deklaration ::=
    Längen-Vereinbarung
  | Typ-Vereinbarung
  | Variablen-Deklaration
  | Format-Deklaration
  | Prozedur-Deklaration
  | Task-Deklaration
  | Operator-Vereinbarung
  | Rang-Vereinbarung
```

Semantik:

Die Deklaration dient dazu, ein Objekt und seinen Namen einzuführen, d.h. bei Auswertung der Deklaration wird Speicherplatz für das Objekt angelegt, und es kann ab diesem Zeitpunkt unter dem in der Deklaration angegebenen Namen darauf zugegriffen werden.

Auf Modul-, Prozedur- oder Taskebene darf ein Objekt nur einmal deklariert werden. Ist ein Bezeichner X sowohl auf Modulebene als auch auf Prozedur-, Task- oder Blockebene als Objekt deklariert, so sind damit zwei verschiedene Objekte eingeführt: In der betreffenden Prozedur, Task oder dem betreffenden Block bezieht man sich unter dem Bezeichner X auf das dort (lokal) deklarierte Objekt, außerhalb der Prozedur, Task oder dem Block auf das auf Modulebene deklarierte Objekt.

Innerhalb von Prozeduren und Tasks dürfen

- Tasks
- Synchronisierungsvariablen
- Datenstationen
- Formate

nicht deklariert werden.

Tabelle 5-1: Zulässigkeit von Deklarationen

Objekt	Deklaration möglich auf			
	Modulebene	Taskebene	Prozedurebene	Blockebene
Variablen, Konstanten	x	x	x	x
Sprungmarken	-	x	x	x
Prozeduren	x	x	x	-
Task	x	-	-	-
Block	-	x	x	x
Sema-, Bolt-Variablen	x	-	-	-
Datenstationen, Formate	x	-	-	-
Typen	x	x	x	x

Erläuterungen:

Beispiel (Zulässigkeit von Deklarationen):

PROBLEM ;

DECLARE x FLOAT; ! 1. Deklaration auf Modulebene

DECLARE x FIXED; ! 2. Deklaration auf Modulebene (falsch)

P : PROCEDURE ;

DECLARE x FIXED; ! Deklaration in der Prozedur P (zulässig)

...

x := 3; ! Zuweisung an die lokale Variable x

...

END ; ! P

T : TASK ;

...

x := 5; ! Zuweisung an die auf Modulebene deklarierte Variable x

...

END ; ! T

...

5.2.2 Spezifikation (SPC) und Identifikation (SPC IDENT)

Syntax:

```
Spezifikation ::=  
    { SPC | SPECIFY } Spezifikations-Angabe [ , Spezifikations-Angabe ] ... ;  
  
Identifikation ::=  
    { SPECIFY | SPC } Bezeichner [ Zuweisungsschutz ] Typ Identifikationsattribut ;  
  
Identifikationsattribut ::=  
    IDENT (Name$Objekt)
```

Semantik:

Mit einer Spezifikation bezieht man sich auf ein bereits deklariertes Objekt. Dies ist sinnvoll für Objekte, die auf Modulebene deklariert und in anderen Modulen benutzt werden sollen (siehe Kap. 5.4 „Bezüge zwischen Modulen“). Mit einer Identifikation können zusätzliche Namen für bereits vereinbarte Objekte eingeführt werden.

Der angegebene Typ bei der Identifikation muß dem Typ des benannten Objekts entsprechen.

Erläuterungen:

Beispiel (Identifikation):

Das Objekt x vom Typ FIXED (15) soll xx als zweiten Namen erhalten, d. h. das Objekt x mit dem Namen xx *identifiziert* werden können:

PROBLEM ;

```
...  
DECLARE x FIXED ;  
...  
SPECIFY xx FIXED IDENT (x) ;  
...  
xx := 7 ;           ! Zuweisung an das Objekt x
```

5.3 Blöcke

Syntax:

```
Block ::=
    BEGIN
    [ {Deklaration| Identifikation} ... ]
    [ Anweisung ... ]
    END [ Bezeichner$Block ] ;
```

Semantik:

Ein Block ist eine Zusammenfassung von Deklarationen, Identifikationen und Anweisungen:

Blöcke dürfen nur innerhalb von Tasks und Prozeduren auftreten, dort allerdings auch ineinander geschachtelt. Der Eintritt in einen Block erfolgt mit der Ausführung von BEGIN. Verlassen wird ein Block durch Überlaufen des zugehörigen END oder durch eine Verzweigung zu einer Anweisung außerhalb des Blocks, z.B. durch eine Exit-Anweisung. Sprünge in einen Block hinein sind nicht erlaubt.

Auf Blockebene dürfen keine Prozeduren deklariert werden!

Erläuterungen:

Den auf Blockebene deklarierten ("lokalen") Objekten wird erst bei Eintritt in den Block Speicherplatz zugewiesen; dieser wird bei Verlassen des Blocks wieder freigegeben. Wie Tasks, Prozeduren und Wiederholungen können Blöcke also Objekte dynamisch einführen und entfernen und bieten somit die Möglichkeit, den zur Verfügung stehenden Speicherplatz mehrfach zu nutzen.

Die Lebensdauer eines Objekts ist die (Ausführungs-) Zeit zwischen der Auswertung seiner Deklaration und der Ausführung des Endes des Blocks (bzw. der Wiederholung, der Prozedur, der Task oder des Moduls), in dem (bzw. in der) die Deklaration erfolgt.

Als Gültigkeitsbereich eines Objekts werden alle Teile des Programms bezeichnet, in denen das Objekt benutzt werden kann. Folgende Regeln sind zu beachten:

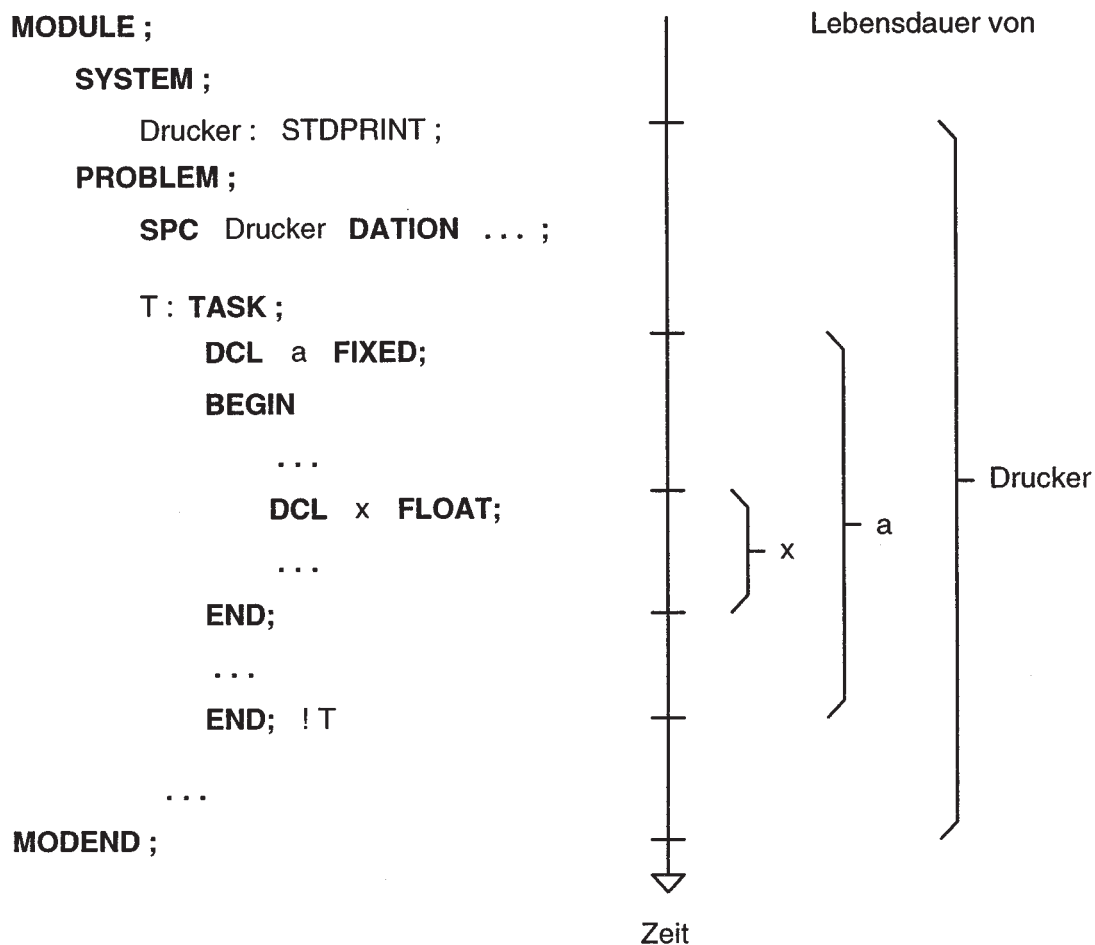
- Ein auf Modulebene vereinbartes Objekt ist auf Modulebene und in allen Tasks und Prozeduren dieses Moduls benutzbar sowie in allen eingeschachtelten Prozeduren, Blöcken und Wiederholungen mit folgender Ausnahme: Der Gültigkeitsbereich wird eingeschränkt, wenn auf Task- oder Prozedurebene ein anderes Objekt unter demselben Namen deklariert wird.

- Ein in einer Task, Prozedur, Wiederholung oder einem Block vereinbartes Objekt ist in dieser Task, Prozedur, Wiederholung oder diesem Block und allen darin eingeschachtelten Prozeduren, Wiederholungen und Blöcken benutzbar mit folgender Ausnahme: Der Gültigkeitsbereich wird eingeschränkt, wenn in einer der eingeschachtelten Prozeduren, Wiederholungen oder Blöcke ein anderes Objekt unter demselben Namen deklariert wird.

Blöcke werden dazu benutzt, Task- oder Prozedurkörper zu strukturieren und den Gültigkeitsbereich und die Lebensdauer von PEARL-Objekten zu beeinflussen.

Blöcke können nach dem END mit Bezeichnern versehen werden. Dadurch können geschachtelte Blöcke mittels der Exit-Anweisung gezielt verlassen werden.

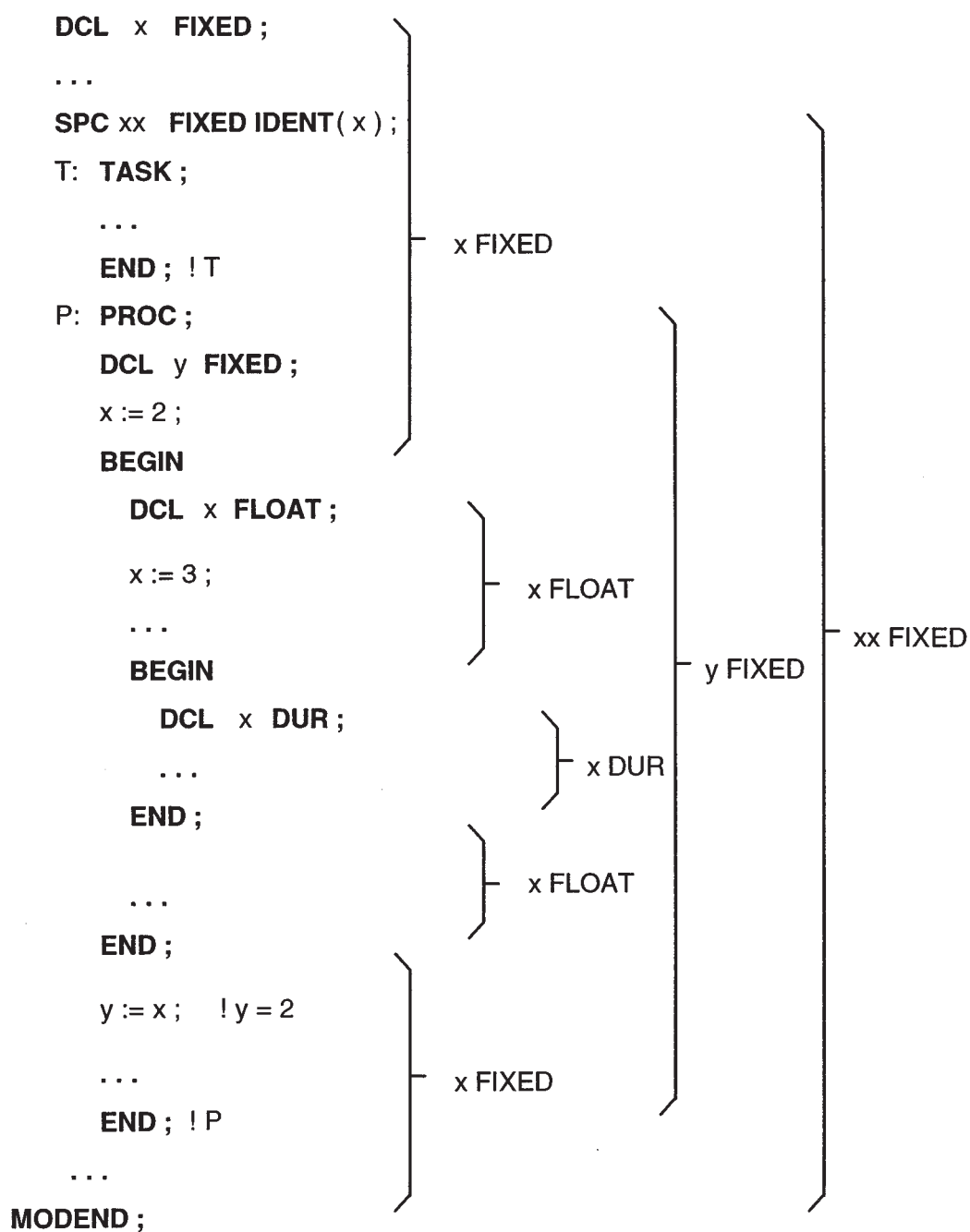
Beispiel (Lebensdauer von Objekten):



Beispiel (Gültigkeitsbereich von Objekten):

PROBLEM ;

Gültigkeitsbereich von



5.4 Bezüge zwischen Modulen

Syntax:

Die allgemeine Form des Global-Attributs lautet:

Global-Attribut ::=
GLOBAL [(BezeichnerSeines-Moduls)]

Semantik:

Die Gültigkeit von auf Modulebene deklarierten Objekten kann auf andere Module erweitert werden, indem diese Objekte mit dem GLOBAL-Attribut deklariert und in den anderen Modulen auf Modulebene mit dem GLOBAL-Attribut spezifiziert werden.

Bei der Spezifikation eines Objekts müssen alle Attribute aus seiner Deklaration mit Ausnahme einer angegebenen Priorität, Genauigkeit oder Länge übernommen werden. In letzterem Ausnahmefall wird für den Bereich des Programms, in dem die Spezifikation gilt, die in der entsprechenden Längenvereinbarung definierte Genauigkeit oder Länge eingesetzt, die Priorität wird aus der Deklaration übernommen.

Alle im Systemteil eines bestimmten Moduls aufgeführten Datenstationen, Interrupts und Signale gelten als (implizit) mit dem Global-Attribut deklariert. Deshalb werden sie in den Problemteilen des Programms nur noch spezifiziert; dabei kann das Global-Attribut im Problemteil desselben Moduls entfallen, in allen anderen Modulen muß es angegeben werden.

Erläuterungen:

Bezüge zu in anderen Implementierungssprachen geschriebenen Modulen sind unter Einhaltung der im Benutzerhandbuch angegebenen Regeln möglich.

Beispiel (Variablen):

```

MODULE ( a );
PROBLEM ;
...
DCL x FIXED GLOBAL ;
...
x := 2 ;
...
MODEND ;

```

```

MODULE ( b );
PROBLEM ;
...
SPC x FIXED GLOBAL ( a );
...
x := 3 ;
...
MODEND ;

```

Beispiel (Prozedur, Task):

```

MODULE ;
PROBLEM ;
    T : TASK PRIO 3 GLOBAL ;
        ! Taskkörper
    END ; ! T

    P : PROC ( A (,) FIXED IDENT )
        GLOBAL ;
        ! Prozedurkörper
    END ; ! P
...
MODEND ;

```

```

MODULE ;
PROBLEM ;
    SPC T TASK GLOBAL ,
        P ENTRY ( (,) FIXED IDENT )
        GLOBAL ;

    INIT : TASK ;
        DCL TAB (10, 20) FIXED ;
        ...
        CALL P ( TAB ) ;
        ...
        ACTIVATE T ;
    END ; ! INIT
...
MODEND ;

```

6 Variablen und Konstanten

6.1 Vereinbarung von Variablen (DCL)

Syntax:

Die allgemeine Form der Deklaration skalarer Variablen lautet wie folgt:

```
Skalare-Variablen-Deklaration ::=
    { DECLARE | DCL } Variablen-Angabe [ , Variablen-Angabe ] * ;

Variablen-Angabe ::=
    Bezeichner-Angabe [ Zuweisungsschutz ] Typ-Attribut [ Global-Attribut ]
    [ Initialisierungsattribut ]

Bezeichner-Angabe ::=
    Bezeichner | (Bezeichner [ , Bezeichner ] * )

Typ-Attribut ::=
    einfacher-Typ | Typ-Referenz | Bezeichner$für-Typ

einfacher-Typ ::=
    Typ-ganze-Zahl | Typ-Gleitpunktzahl |
    Typ-Bitkette | Typ-Zeichenkette |
    Typ-Uhrzeit | Typ-Dauer
```

Die allgemeine Form der Spezifikation skalarer Variablen lautet wie folgt:

```
Skalare-Variablen-Spezifikation ::=
    { SPECIFY | SPC } Variablen-Angabe-S [ , Variablen-Angabe-S ] * ;

Variablen-Angabe-S ::=
    Bezeichner-Angabe [ Zuweisungsschutz ] Typ-Attribut
    { Global-Attribut | Identifikationsattribut }
```

Semantik:

Die von einem PEARL-Programm bei seiner Ausführung benutzten Daten (ganze Zahlen, Gleitpunktzahlen, Bitketten, Zeichenketten, Uhrzeiten und Zeitdauern) treten in Form von Konstanten oder als Werte von Variablen auf. Konstanten identifizieren sich durch ihre Schreibweise und behalten ihren Wert während des gesamten Programmablaufs. Die Werte von Variablen bezeichnen Daten, die sich während des Programmablaufs ändern können.

Der Wertebereich einer Variablen ist im allgemeinen auf eine bestimmte Art von Daten, z. B. Bitketten, beschränkt, die den Typ der Variablen bestimmt. (Bitketten-Variablen haben z. B. nur Bitketten als Werte.) Dieser Typ muß bei der Deklaration und Spezifikation einer Variablen zusammen mit ihrem Bezeichner angegeben werden.

Bei der Deklaration einer Variablen muß ihr Typ in einem Typ-Attribut angegeben werden; sollen verschiedene Variablen mit gleichem Typ-Attribut vereinbart werden, so kann dies in Form einer Liste und einmaliger Angabe des Typ-Attributs erfolgen. Verschiedene Vereinbarungen können in einer Deklaration formuliert werden, indem sie durch Kommata getrennt werden.

Erläuterungen:

Beispiel (Angabe von Typen in Deklaration und Spezifikation):

```
DECLARE  Schalter  BIT ;    /* Deklaration einer Variablen Schalter vom Typ Bitkette
                                der Länge 1 */
```

```
SPECIFY Status  BIT (16) GLOBAL ;    /* Spezifikation einer globalen Variablen
                                Status vom Typ Bitkette der Länge 16 */
```

Beispiel (Liste von Variablen gleichen Typs):

```
DECLARE  (x, y, z) FLOAT ;
```

Beispiel (Deklaration von Variablen unterschiedlichen Typs):

```
DECLARE    x FLOAT,
            i FIXED ;
```

6.2 Ganze Zahlen (FIXED)

Syntax:

```
Typ-ganze-Zahl ::=
    FIXED [ (Genauigkeit) ]
```

```
Genauigkeit ::=
    ganze-Zahl-ohne-Genauigkeit$größer-Null
```

```
ganze-Zahl ::=
    ganze-Zahl-ohne-Genauigkeit [ ( Genauigkeit ) ]
```

```
ganze-Zahl-ohne-Genauigkeit ::=
    Ziffer ... | { 0 | 1 } ... B
```

Semantik:

Variablen vom Typ ganze Zahl werden mit dem Typ-Attribut **FIXED** deklariert.

Die Genauigkeit legt die Anzahl der Bitstellen (ohne Vorzeichen) fest, in denen der jeweilige Wert der Variablen dargestellt wird. Fehlt die Genauigkeitsangabe, so wird die in einer Längenvereinbarung definierte Genauigkeit eingesetzt.

Ganze Zahlen können in Dezimaldarstellung oder Dualdarstellung geschrieben werden. Die Dezimaldarstellung einer ganzen Zahl besteht aus einer Folge von Ziffern. Die Dualdarstellung einer ganzen Zahl besteht aus einer Folge der Ziffern 0 und 1, die mit dem Zeichen **B** abgeschlossen wird.

Zudem kann für eine Zahlenkonstante die Genauigkeit ihrer Darstellung definiert werden, indem anschließend an die Zahlenkonstante die Anzahl der Bitstellen (ohne Vorzeichen) in Klammern vermerkt wird, mit der sie rechnerintern dargestellt werden soll.

Erläuterungen:

Ist bei Zahlenkonstanten keine Genauigkeit angegeben, wird die Genauigkeit aus der Konstanten abgeleitet.

Gängige Genauigkeiten sind 15 oder 31. Die angegebenen Genauigkeiten sind Mindest-Genauigkeiten. Ist die geforderte Genauigkeit nicht realisierbar, wird die nächsthöhere verwendet.

Beispiele (Zahlenkonstanten vom Typ **FIXED** in Dezimal- und Dualdarstellung):

Dezimaldarstellung	Dualdarstellung
6	110B
123	1111011B

Beispiel (**FIXED**-Zahlenkonstante mit Genauigkeit):

: 123(31) Die ganze Zahl 123 wird in 31 Bitstellen (ohne Vorzeichen) dargestellt.

Beispiel (**FIXED**-Variablen-Deklaration mit und ohne Genauigkeitsangabe):

```
DCL Zaehler FIXED (31) ,  
    (i, j, k) FIXED ;  
...  
i := 2 ;
```

6.3 Gleitpunktzahlen (FLOAT)

Syntax:

Typ-Gleitpunktzahl ::=
 Float [(Genauigkeit)]

Gleitpunktzahl ::=
 Gleitpunktzahl-ohne-Genauigkeit [(Genauigkeit)]

Gleitpunktzahl-ohne-Genauigkeit ::=
 { Ziffer [Ziffer ...] | . Ziffer ... } [Exponent]
 | Ziffer ... Exponent

Exponent ::=
 E [+ | -] Ziffer ...

Semantik:

Variablen vom Typ Gleitpunktzahl (mit ganzen Zahlen oder Gleitpunktzahlen als Werten) werden mit dem Typ-Attribut **Float** deklariert.

Konstanten für Gleitpunktzahlen können dargestellt werden als eine Folge von

- (i) einem Punkt, einer ganzen Zahl und eventuell einem Exponenten zur Basis 10, wobei ein Exponent aus einer Folge des Zeichens E, eventuell einem Plus- oder Minuszeichen und einer ganzen Zahl besteht.

Beispiel: .123 (entspricht 0,123)
 .123E2 (entspricht 12,3)
 .123E-1 (entspricht 0,0123)

- (ii) einer ganzen Zahl und der unter (i) angegebenen Folge.

Beispiel: 3.123E2 (entspricht 312,3)

- (iii) einer ganzen Zahl, einem Punkt und eventuell einem Exponenten.

Beispiel: 3. (entspricht 3,0)

- (iv) einer ganzen Zahl und einem Exponenten

Beispiel: 3E-2 (entspricht 0,03)

Analog zu ganzen Zahlen können auch Gleitpunktzahl-Konstanten mit Genauigkeit formuliert werden.

Erläuterungen:

Gängige Genauigkeiten sind 23 bzw. 53.

Beispiele (FLOAT-Variablen-Deklaration mit und ohne Genauigkeitsangabe):

```
DCL  (x, y, z)  FLOAT,  
      Koeff  FLOAT (53) ;  
...  
x := 3.5 ; y := 1 ; Koeff := 3.14 E-10 ;
```

6.4 Bitketten (BIT)

Syntax:

```
Typ-Bitkette ::=  
    BIT [ (Länge) ]  
  
Länge ::=  
    ganze-Zahl-ohne-Genauigkeit$größer-Null  
  
Bitkettenkonstante ::=  
    ' B1-Ziffer' '{ B1 }' | ' B2-Ziffer' ' B2' | ' B3-Ziffer' ' B3' | ' B4-Ziffer' ' B4'  
  
B1-Ziffer ::=  0 | 1  
  
B2-Ziffer ::=  0 | 1 | 2 | 3  
  
B3-Ziffer ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7  
  
B4-Ziffer ::=  0 | 1 | 2 | 3 | 4 | 5 | 6 | 7 | 8 | 9 | A | B | C | D | E | F
```

Semantik:

Variablen für Bitketten werden mit dem Typ-Attribut BIT deklariert.

Länge gibt die Anzahl der Elemente einer Bitkette an. Fehlt die Längenangabe, so muß sie entweder in einer Längenvereinbarung definiert worden sein, oder es wird als Länge 1 eingesetzt.

Eine Bitkettenkonstante kann binär (B1), in Form von Tetraden (B2), Oktaden (B3) oder in hexadezimaler Form (B4) angegeben werden.

Die Form B_i ($i = 1, \dots, 4$) besteht aus einem Apostroph, einer Folge

- der Ziffern 0 und 1 im Fall B1
- der Ziffern 0 bis 3 im Fall B2
- der Ziffern 0 bis 7 im Fall B3
- der Ziffern 0 bis 9 und Buchstaben A bis F im Fall B4

sowie einem Apostroph und der entsprechenden Angabe B1 oder B2 oder B3 oder B4.

Anstatt B1 kann kürzer B geschrieben werden.

Tabelle 6-1: Zuordnung zwischen der Binärform und den anderen Formen:

<u>B2</u>	<u>B1</u>	<u>B3</u>	<u>B1</u>	<u>B4</u>	<u>B1</u>
0	00	0	000	0	0000
1	01	1	001	1	0001
2	10	2	010	2	0010
3	11	3	011	3	0011
		4	100	4	0100
		5	101	5	0101
		6	110	6	0110
		7	111	7	0111
				8	1000
				9	1001
				A	1010
				B	1011
				C	1100
				E	1110
				F	1111

Erläuterungen:

Beispiel (Entsprechungen von BIT-Ketten-Konstanten)

'110010100111'B1	entspricht
'302213'B2	entspricht
'6247'B3	entspricht
'CA7'B4	

Beispiel (Längenangaben bei Variablen-Deklaration):

```
DCL  X_Koord  BIT (2) ,  
      Y_Koord  BIT (8) ;  
  
...  
X_Koord := '01'B ;  
Y_Koord := 'A9'B4 ;
```

6.5 Zeichenketten (CHARACTER)

Syntax:

```
Typ-Zeichenkette ::=  
    { CHARACTER | CHAR } [ (Länge) ]  
  
Zeichenkettenkonstante ::=  
    ' { Zeichen-außer-Apostroph | " | Kontrollzeichen-Sequenz } ... '  
  
Zeichen-außer-Apostroph ::=  
    Ziffer | Buchstabe | _ | + | - | * | / | \ | ( | ) | [ | ] | : | . | ; | , | = | < | > | !  
    | /* weitere druckbare Zeichen des Maschinensatzes */  
  
Kontrollzeichen-Sequenz ::=  
    '\ { B4-Ziffer B4-Ziffer } ... \'
```

Semantik:

Variablen für Zeichenketten werden mit dem Typ-Attribut CHARACTER deklariert.

Länge gibt die Anzahl Zeichen an. Fehlt die Längenangabe, so muß sie entweder in einer Längenvereinbarung (siehe 6.6) definiert worden sein oder es wird die Länge 1 eingesetzt.

Eine Zeichenkettenkonstante besteht aus einem Apostroph, einer Folge von beliebigen Zeichen (außer Apostroph) und einem Apostroph.

Soll jedoch die Zeichenkette einen Apostroph enthalten, so muß dieser durch zwei aufeinanderfolgende Apostrophe dargestellt werden.

Steuerzeichen können mit Hilfe der Umschaltssymbole “ \ ” und “ \ ” in Zeichenkettenkonstanten eingefügt werden. Die Steuerzeichen werden als Paare von Hexadezimal-Ziffern angegeben.

Erläuterungen:

Beispiel (Zeichenketten-Konstante ohne Apostroph):

```
'Stoerung Nr: '
```

Beispiel (Zeichenketten-Konstante mit Apostroph):

```
'Stoer' 'Nr: '
```

Beispiele (Zeichenketten-Konstanten mit Steuerzeichen):

- (1) 'Dieser String '\0D\0A\' enthält zwei Steuerzeichen'
 '\1B\' Steuerzeichen am String-Anfang'
 'Steuerzeichen am String-Ende'\00\
 "\00\" /* String bestehend aus einem einzigen Steuerzeichen */
- (2) 'Durch die Umschaltung auf die Steuerzeichensequenz'\ 20
 \'können sehr lange Zeichenkettenkonstanten'\ 20
 \'(unabhängig vom verwendeten Editor) erstellt werden.'

Beispiele (Variable vom Typ CHAR mit Längenangabe):

```
DCL Artikelbez CHAR (6) ;  
...  
Artikelbez := 'BCD/27' ;
```

6.6 Die Längenvereinbarung

Syntax:

```
Längenvereinbarung ::=  
    LENGTH      { { FIXED | FLOAT } (Genauigkeit)  
                | { BIT | CHARACTER | CHAR } (Länge) } ;
```

Semantik:

Mit der Längenvereinbarung werden die Genauigkeiten bzw. Längen für solche Zahlen- und Ketten-Objekte definiert, deren Genauigkeiten bzw. Längen nicht durch die Schreibweise (bei Konstanten) oder die Deklaration (bei Variablen) festgelegt ist.

Für eine Längenvereinbarung gelten die gleichen Gültigkeitsregeln wie für Variablendeklarationen.

Erläuterungen:

Beispiel (Gültigkeit von Längenvereinbarungen):

```
PROBLEM ;  
    LENGTH FIXED(15);  
    LENGTH FLOAT(53);  
    DCL   A  FIXED ,           /* A hat den Typ FIXED(15) */  
        X  FLOAT ,           /* X hat den Typ FLOAT(53) */  
        Y  FLOAT(23);        /* Y hat den Typ FLOAT(23) */  
    ...
```

Ist die vereinbarte Genauigkeit nicht realisierbar, wird die nächsthöhere verwendet.

6.7 Uhrzeiten (CLOCK)

Syntax:

Typ-Uhrzeit ::=
 CLOCK

Uhrzeitkonstante ::=
 Ziffer '...' : Ziffer '...' : Ziffer '...' [. Ziffer '...']

Semantik:

Variablen für Uhrzeiten werden mit dem Typ-Attribut **CLOCK** deklariert.

Eine Uhrzeitkonstante besteht aus einer ganzen Zahl für die Stundenangabe, einer ganzen Zahl zwischen 0 und 59 für die Minutenangabe und einer Gleitpunktzahl zwischen 0 und 59.999... für die Sekundenangabe - jeweils durch Doppelpunkt getrennt. Die Stundenangabe wird modulo 24 interpretiert.

Erläuterungen:

Beispiele (Uhrzeit-Konstanten):

11:30:00	bedeutet 11.30 Uhr
15:45:3.5	bedeutet 15.45 Uhr und 3,5 Sekunden
25:00:00	bedeutet 1 Uhr

Beispiel (Deklaration einer Uhrzeit-Variablen):

```
DCL Zeit CLOCK ;  
...  
Zeit := 12:30:00 ;
```

6.8 Zeitdauern (DURATION)

Syntax:

Typ-Dauer ::=
 DURATION | **DUR**

Dauerkonstante ::=
 Stundenangabe [Minutenangabe] [Sekundenangabe]
 | Minutenangabe [Sekundenangabe]
 | Sekundenangabe

Stundenangabe ::=
 ganze-Zahl-ohne-Genauigkeit **HRS**

Minutenangabe ::=
 ganze-Zahl-ohne-Genauigkeit **MIN**

Sekundenangabe ::=
 { ganze-Zahl-ohne-Genauigkeit | Gleitpunktzahl-ohne-Genauigkeit } **SEC**

Semantik:

Variablen für Zeitdauern werden mit dem Typ-Attribut DURATION deklariert.

Eine Dauerkonstante setzt sich aus einer Stundenangabe, Minutenangabe und Sekundenangabe zusammen, wobei einzelne dieser Angaben fehlen können. Eine Stundenangabe besteht aus einer ganzen Zahl und der Zeichenfolge HRS, eine Minutenangabe aus einer ganzen Zahl und der Zeichenfolge MIN, eine Sekundenangabe aus einer Gleitpunktzahl und der Zeichenfolge SEC.

Die ganzen Zahlen und Gleitpunktzahlen in Dauerkonstanten dürfen keine Genauigkeitsangaben enthalten.

Erläuterungen:

Beispiele (Dauerkonstanten):

5 **MIN** 30 **SEC** bedeutet 5 Minuten und 30 Sekunden
.05 **SEC** bedeutet 50 Millisekunden
5 **HRS** 10 **SEC** bedeutet 5 Stunden und 10 Sekunden

Beispiel (Deklaration einer Zeit-Dauer-Variablen):

DCL Verzögerung **DUR** ;
...
Verzögerung := 0.1 **SEC** ;

6.9 Referenzen (REF)

Syntax:

```

Typ-Referenz ::=
    REF [ virt_Dimensionsliste ] [ Zuweisungsschutz ]
    { einfacher-Typ | Bezeichner$für-Typ | Typ-Struktur |
      Typ-Dation | SEMA | BOLT | INTERRUPT | IRPT | SIGNAL |
      Typ-Prozedur | TASK | Typ-VOID | CHAR( )
    }

virt-Dimensionsliste ::=
    ( [ , ' ' ] )

Dereferenzierung ::=
    CONT { Name$Referenz-Variable | Funktionsaufruf }

Nullzeigerkonstante ::= NIL

```

Semantik:

Referenz-Variablen haben als Werte die Namen von Objekten (sie zeigen auf Objekte). Analog zu den bisher eingeführten Variablen ist dabei der Wertebereich einer Referenz-Variablen auf genau einen Typ von Objekten beschränkt; dieser wird bei der Deklaration der Referenz-Variablen angegeben.

Auf den Wert der referierten Variablen bezieht man sich mittels des auf die Referenz-Variable angewandten monadischen Operators **CONT** (von "content"); dieser Vorgang wird "Dereferenzierung" genannt.

Anstelle von $k := \text{CONT } rk1$ kann auch einfacher $k := rk1$ geschrieben werden; d.h. der Operator **CONT** darf auf der rechten Seite einer Zuweisung weggelassen werden ("implizite Dereferenzierung").

Die Definition der Sprachform Typ-Referenz zeigt, welchen Typ eine Variable haben darf, auf die eine Referenz-Variable zeigen soll. Insbesondere dürfen Referenz-Variablen auch auf Felder und Strukturen zeigen. Die Anzahl n der Kommata in der virtuellen Dimensionsliste gibt an, daß die referierte Variable ein $(n + 1)$ -dimensionales Feld ist. Hingegen dürfen Referenz-Variablen nicht auf Referenz-Variablen zeigen; allerdings dürfen referierte Felder und Strukturen Referenz-Variablen als Elemente und Komponenten besitzen.

Zeigt eine Referenz-Variable R auf ein Feld F mit den Elementen $F(i, j, k, \dots)$ oder auf eine Struktur S mit den Komponenten $S.K_i$, so werden (ausgehend von R) die Elemente von F oder die Komponenten von S ohne Benutzung von **CONT** mit $R(i, j, k, \dots)$ oder $R.K_i$ angesprochen (implizite Dereferenzierung).

Eine Referenz-Variable wird außerdem implizit dereferenziert,

- wenn sie aktueller Parameter eines Prozeduraufrufs ist und der zugehörige formale Parameter keine Referenz-Variable ist.
- wenn sie als Operand eines dyadischen Operators benutzt wird, sofern dieser nicht genau für Werte von Referenz-Variablen definiert ist wie IS und ISNT.
- wenn sie für einen Prozeduraufruf, für eine Taskanweisung oder für einen Zugriff auf Strukturkomponenten verwendet wird.

Um beispielsweise das Ende einer Kette zu kennzeichnen, steht die Nullzeiger-Konstante NIL zur Verfügung.

Zum Vergleich der Werte von Referenz-Variablen untereinander und mit Objektnamen oder der Nullzeigerkonstanten können die dyadischen Operatoren IS oder ISNT benutzt werden; IS (ISNT) liefert das Ergebnis '1'B ('0'B), wenn die beiden angegebenen Referenz-Variablen denselben (verschiedene) Wert(e) besitzen, andernfalls das Ergebnis '0'B ('1'B).

Variable Zeichenkettenzeiger (REF CHAR ())

Variabel lange Zeichenketten können mit speziellen Zeiger-Objekten, den sogenannten variablen Zeichenkettenzeigern, leicht verarbeitet werden.

Ein Zeiger vom Typ "REF CHAR ()" enthält neben der Adresse einer Zeichenkette noch zwei Zähler, die die maximale Länge und die aktuell benutzte Länge der referenzierten Zeichenkettenvariable speichern. Bei der Zuweisung der Adresse einer Zeichenkettenvariable an einen variablen Zeichenkettenzeiger werden beide Zähler mit der Länge der referenzierten Zeichenkettenvariable initialisiert.

Damit unterscheidet sich der variable Zeiger zunächst nicht von einem Zeichenkettenzeiger mit einer konstanten Länge. Bei Zuweisungen an die Zeichenkettenvariable über diesen Zeiger wird jedoch die Länge der zugewiesenen Zeichenkette in dem Zähler für die aktuelle Länge gespeichert. Das bei Zeichenketten fester Länge übliche Auffüllen der Zeichenkette bis zur maximalen Länge mit Leerzeichen entfällt.

Bei lesenden Zugriffen auf die Zeichenkette über einen variablen Zeichenkettenzeiger wird die aktuelle Länge benutzt.

Ein variabler Zeichenkettenzeiger ist als Ergebniswert einer Funktion nicht erlaubt. Der Typ "CHAR ()" darf nur in Verbindung mit "REF" benutzt werden, d.h. der Parametertyp "CHAR () IDENT" ist nicht erlaubt.

Erläuterungen:

Für die indirekte Adressierung stehen in PEARL sogenannte Referenz-Variablen (Zeigervariablen, Pointer) zur Verfügung.

Beispiele (Benutzung von REF-Variablen):

```

DCL  (k, l) FIXED,
      x  FLOAT,
      (rk1, rk2) REF FIXED,      /* Fixed-Referenz-Variable */
      rx  REF FLOAT;            /* Float-Referenz-Variable */
rk1 := k;                       /* rk1 zeigt auf k */
rk1 := l;                       /* rk1 zeigt auf l */
rk2 := rk1;                     /* rk2 zeigt auf l, Zeigerzuweisung */
rx := x;                       /* rx zeigt auf x */
rx := k;                       /* falsch, Typ ungleich */
rx := rk1;                     /* falsch, Typ ungleich */
l := 2;
k := CONT rk1;                 /* k erhält den Wert 2 */
rk2 := 3;                     /* falsch, 3 ist keine Variable */
CONT rk2 := 3;                 /* l erhält den Wert 3 */
CONT rk2 := k;                 /* l erhält den Wert 2 */

```

Beispiel (implizite Dereferenzierung):

```

DCL  rk    REF FIXED,
      k     FIXED;
...
rk := k; k := 2;
k := rk + 1; /* äquivalent k := k + 1; */

```

Beispiel (Verwendung von IS):

```

DCL Nextauftrag REF Typ_Auftrag;
...
IF Nextauftrag IS NIL THEN
...
FIN;

```

Beispiel (Vereinbarungen variabler Zeichenkettenzeiger):

```

DCL str_ptr REF CHAR ();      /* Zeiger-Deklaration */
SPC print ENTRY ( REF CHAR () ) GLOBAL; /* Zeiger als Parameter */

```

Beispiel (Initialisierung eines variablen Zeichenkettenseigers):

```
DCL s1      CHAR(20);
DCL s2      CHAR(100);
DCL str_ptr REF CHAR ( );
str_ptr := s1;      /* Adr=s1, Länge=20, Max=20 */
str_ptr := s2;      /* Adr=s2, Länge=100, Max=100 */
```

Beispiel (Zuweisungen an eine referenzierte Variable):

```
str_ptr := s1;      /* Adr=s1, Länge=20, Max=20 */
CONT str_ptr := 'PEARL 90'; /* Adr=s1, Länge=8, Max=20 */

str_ptr := s2;      /* Adr=s2, Länge=100, Max=100 */
CONT str_ptr := 'Werum GmbH'; /* Adr=s2, Länge=10, Max=100 */
```

Beispiel (lesender Zugriff über variablen Zeichenkettenseiger):

```
DCL s1      CHAR(20);
DCL s2      CHAR(100);      /* platzspendende Variable */
DCL str_ptr REF CHAR ( );   /* variabler Zeiger */

str_ptr := s2;      /* Adr=s2, Länge=100, Max=100 */
CONT str_ptr := 'D21337 ' CAT 'Lueneburg'; /* Adr=s2, Länge=16, Max=100 */
```

Nach dieser Zuweisung enthält die Variable "s2" an den ersten 16 Stellen die Zeichen "D21337 Lueneburg", die restlichen Zeichen wurden durch die Zuweisung nicht verändert. Der Längenindex von "str_ptr" wurde auf die aktuelle Länge 16 eingestellt.

```
s1 := CONT str_ptr;
```

Bei dieser Zuweisung erzeugt "CONT str_ptr" den Zeichenkettenwert 'D21337 Lueneburg' (mit dem Typ "CHAR(16)"). Nach den Zuweisungsregeln von PEARL wird die Zeichenkette mit vier Leerzeichen erweitert, bevor dieser Wert der Variablen "s1" (mit dem Typ "CHAR(20)") zugewiesen wird.

Besonders nützlich erweisen sich variable Zeichenkettenseiger als formale Parameter von Prozeduren. Der Kompilierer übergibt an der Aufrufstelle neben der Adresse einer Zeichenkettenvariablen auch deren aktuelle Länge. So kann z.B. eine Fehleroutine unterschiedlich lange Fehlertexte ausgeben.

Beispiel (Zeichenkettenseiger als Parameter):

```
SPC print_fehler ENTRY ( REF CHAR ( ) ) GLOBAL;
DCL s1 CHAR(20) INIT ( '...' );
DCL s2 CHAR(100) INIT ( '...' );

...
CALL print_fehler ( s1 );
CALL print_fehler ( s2 );
```

Es dürfen auch Zeichenkettenkonstanten übergeben werden, wenn der formale Parameter mit dem Typ "REF INV CHAR ()" definiert wurde.

Beispiel (Zeichenkettenkonstanten als Parameter):

```
SPC print_message ENTRY ( REF INV CHAR ( ) ) GLOBAL;
```

```
...  
CALL print_message ( 'Text 1' );  
CALL print_message ( 'laengerer Text 2' );
```

Ein variabler Zeichenkettenzeiger kann in allen Ausdrücken analog zu konstanten Zeichenkettenzeigern benutzt werden. Die zu der Typprüfung gehörende Längenprüfung erfolgt erst zur Laufzeit.

6.10 Felder

Syntax:

Die allgemeine Form der Deklaration eines Feldes lautet:

```
Feld-Deklaration ::=  
    { DECLARE | DCL } Feld-Angabe [ , Feld-Angabe ] ... ;  
  
Feld-Angabe ::=  
    Bezeichner-Angabe Dimensionsattribut [ Zuweisungsschutz ] Typ-Attribut  
    [ Global-Attribut ] [ Initialisierungsattribut ]  
  
Dimensions-Attribut ::=  
    ( Grenzangabe$erste-Dimension [ , Grenzangabe$weitere-Dimension ] ... )  
  
Grenzangabe ::=  
    [ Grenze$unten : ] Grenze$oben  
  
Grenze ::=  
    [ - ] ganze-Zahl-ohne-Genauigkeit | Bezeichner$benannte-Konstante |  
    Konstanter-Ausdruck
```

Die allgemeine Form der Spezifikation eines Feldes lautet:

```
Feld-Spezifikation ::=  
    { SPECIFY | SPC } Feld-Angabe-S [ , Feld-Angabe-S ] ... ;  
  
Feld-Angabe-S ::=  
    Bezeichner-Angabe virtuelles-Dimensionsattribut [ Zuweisungsschutz ]  
    Typ-Attribut { Global-Attribut | Identifikationsattribut }  
  
virtuelles-Dimensionsattribut ::=  
    ( [ , ... ] )
```

Semantik:

Skalare Variablen gleichen Typs können zu n-dimensionalen Feldern zusammengefaßt werden ($n = 1, 2, 3, 4$).

Dem Feld wird bei seiner Deklaration ein Bezeichner zugeordnet; die einzelnen Feldelemente (skalare Variablen) werden unter diesem Bezeichner und der Angabe ihrer Position innerhalb des Feldes (dem Index) angesprochen.

So wird z.B. mittels

DCL Koeff (1:2, 0:3) **FIXED** ;

ein 2-dimensionales Feld vereinbart, wobei die erste Dimension die untere Grenze 1 und die obere Grenze 2, d.h. die Länge 2 besitzt, während die zweite Dimension die untere Grenze 0 und die obere Grenze 3, d.h. die Länge 4 hat. Dies bedeutet, daß Koeff aus den 8 skalaren Fixed-Variablen

Koeff(1,0)	Koeff(1,1)	Koeff(1,2)	Koeff(1,3)
Koeff(2,0)	Koeff(2,1)	Koeff(2,2)	Koeff(2,3)

besteht.

Auch negative ganze Zahlen sind als Dimensionsgrenze zugelassen. Die obere Grenze einer Dimension muß jedoch immer größer oder gleich ihrer unteren Grenze sein.

Ist die untere Grenze nicht angegeben, so wird hierfür der Wert 1 angenommen. Unter einer benannten Konstante wird hier eine Variable vom Typ **FIXED** (15) verstanden, die mit Zuweisungsschutz (INV) und Initialwert (INIT) deklariert ist.

Felder können auch gemischt mit skalaren Variablen in einer Deklaration vereinbart werden.

Die Anzahl n der Kommata im virtuellen Dimensionsattribut gibt an, daß das spezifizierte Feld $n + 1$ Dimensionen besitzt.

Erläuterungen:

Nach Möglichkeit werden beim Programmieren gleichartige Objekte unter einem Bezeichner zusammengefaßt und die einzelnen Objekte indiziert angesprochen.

Beispiel:

Eine unterlagerte Steuerung steuert drei Geräte G(1), G(2) und G(3). Bei Ausgabe der Bitkette '0001'B an die unterlagerte Steuerung soll G(1) eingeschaltet werden, bei Ausgabe von '0010'B das Gerät G(2), bei Ausgabe von '0100'B das Gerät G(3). Hierzu bietet sich an, die drei Einschaltsignale unter einem Bezeichner, z.B. einschalten, zusammenzufassen und indiziert zu benutzen:

```
DCL  einschalten (1 : 3) BIT (4) ,  
      i FIXED ;  
einschalten (1) := '0001'B ;  
einschalten (2) := '0010'B ;  
einschalten (3) := '0100'B ;  
...
```

/* Übernahme des Wertes von i (Index des einzuschaltenden Geräts) von einem anderen Programmteil
Ausgabe von einschalten (i) an die unterlagerte Steuerung */

Beispiele (Felddeklarationen mit impliziter unterer Grenze):

```
DCL  einschalten (3) BIT (4) ,  
      Koeff (2, 0:3) FIXED ,  
      Meldung (20) CHAR (12) ;
```

Das eindimensionale Feld Meldung enthält z.B. 20 Fehlertexte, damit im Fehlerfall i ein Programm die Meldung (i) an eine Konsole ausgeben kann (i = 1, ... , 20).

Beispiel („Mischung“ von Felddeklarationen und Deklarationen skalarer Variabler):

```
DCL  Anz_Meld INV FIXED (15) INIT (20) ;  
...  
  
DCL  Meldung (Anz_Meld) CHAR (12) ,  
      (i, j, k) FIXED ,  
      (einschalten, ausschalten) (3) BIT (4) ;
```

6.11 Strukturen

Syntax:

Die allgemeine Form der Struktur-Deklaration lautet:

```
Struktur-Deklaration ::=  
    { DECLARE | DCL } Struktur-Angabe [ , Struktur-Angabe ] ... ;  
  
Struktur-Angabe ::=  
    Bezeichner-Angabe$Hauptstruktur [ Dimensionsattribut ] [ Zuweisungsschutz ]  
    Typ-Struktur [ Global-Attribut ] [ Initialisierungsattribut ]  
  
Typ-Struktur ::=  
    STRUCT [ Strukturkomponente [ , Strukturkomponente ] ... ]  
  
Strukturkomponente ::=  
    Bezeichner-Angabe Typ-Attribut-in-Struktur-Vereinbarung  
  
Typ-Attribut-in-Struktur-Vereinbarung ::=  
    [ Dimensionsattribut ]  
    { einfacher-Typ | Typ-Referenz | Bezeichner$für-Typ | Typ-Struktur }
```

Die allgemeine Form der Struktur-Spezifikation lautet:

```
Struktur-Spezifikation ::=  
    { SPECIFY | SPC } Struktur-Angabe-S [ , Struktur-Angabe-S ] ... ;  
  
Struktur-Angabe-S ::=  
    Bezeichner-Angabe$Hauptstruktur [ virtuelles-Dimensionsattribut ]  
    [ Zuweisungsschutz ] Typ-Struktur { Global-Attribut | Identifikationsattribut }
```

Eine Struktur wird in eckige Klammern eingeschlossen; diese sind hier fett gedruckt, um sie von den metasprachlichen Zeichen “[“ und “]“ zu unterscheiden.

Semantik:

Anders als Feldelemente werden die Komponenten einer Struktur nicht über den gemeinsamen Bezeichner und einen Index, sondern über den gemeinsamen Bezeichner und ihren in der Vereinbarung angegebenen Komponentenbezeichner angesprochen, wobei die beiden Bezeichner durch einen Punkt getrennt werden.

Komponenten von Strukturen dürfen Felder sein; die Feldelemente werden dann wie gewohnt mit ihrem indizierten Bezeichner angesprochen, dem der Strukturbezeichner vorangestellt ist.

Eine Strukturkomponente kann jedoch auch selbst wieder eine Struktur sein, wodurch nicht nur lineare, sondern auch hierarchische Datenstrukturen nachgebildet werden können. Unterstrukturen können über ihren Bezeichner und die vorangestellten Bezeichner der jeweils übergeordneten Strukturen angesprochen werden.

Der Typ einer Struktur wird durch die Anordnung ihrer Komponenten und deren Typ bestimmt. Strukturen gleichen Typs können zu Feldern zusammengefaßt werden. Allgemein wird deshalb eine beliebige Komponente einer Struktur unter ihrem (gegebenenfalls indizierten) Bezeichner und den vorangestellten (gegebenenfalls indizierten) Bezeichnern aller übergeordneten Strukturen angesprochen, wobei diese Namen jeweils durch einen Punkt getrennt werden.

Erläuterungen:

Bei vielen Automationsaufgaben, insbesondere bei solchen mit dispositivem Charakter, müssen Datenstrukturen beschrieben werden, deren Komponenten *unterschiedliche* Typen besitzen.

Beispiel:

Die Beiträge für die Fernseh-Nachrichtensendungen eines Tages sind auf Magnetaufzeichnungsgeräten (MAZ) gespeichert; die Beitragsfolge einer bestimmten Nachrichtensendung dieses Tages wird von Redakteuren mit Hilfe eines Rechners zusammengestellt, der die entsprechenden MAZ steuert. Dazu ist für jeden Beitrag ein Datensatz nötig, der z.B. folgende Struktur besitzt:

- Identitätskennzeichen des Beitrages
- Archivnummer
- Hinweise, ob der Beitrag bereits in der ersten, zweiten oder dritten Nachrichtensendung dieses Tages gesendet wurde
- Anfangsposition des Beitrages auf dem Band
- Endeposition des Beitrages auf dem Band
- Kennzeichen, ob Originalton vorhanden ist
- Länge des gegebenenfalls zu speichernden Texts
- Ein gegebenenfalls zu speichernder Text

Solche Datenstrukturen lassen sich problemorientiert als Struktur beschreiben; die obige Struktur "Beitrag" wird beispielsweise wie folgt vereinbart:

```
DCL Beitrag STRUCT  
  [ (Identnr, Archiv) FIXED,  
    schon_gesendet (3) BIT (1),  
    (Anfang, Ende) FIXED,  
    Originalton BIT (1),  
    Textlaenge FIXED,  
    Text CHAR (200) ] ;
```

Beispiel (Zuweisung an eine Strukturkomponente):

Durch die Anweisung

```
Beitrag.Anfang := 1027;
```

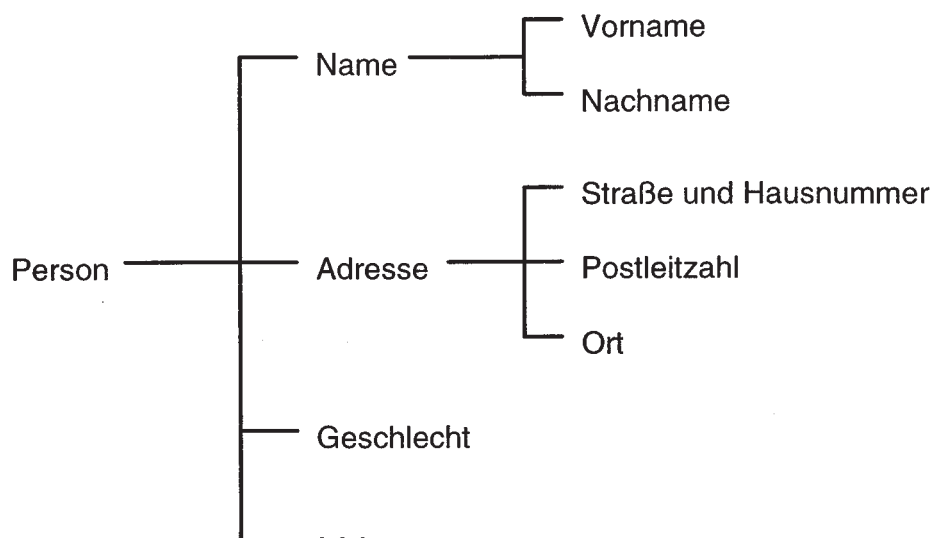
erhält die Komponente Anfang der Struktur Beitrag den Wert 1027 zugewiesen.

Beispiel (Zugriff auf das Element eines Feldes, das Komponente einer Struktur ist):

```
IF Beitrag.schon_gesendet (i) THEN ... FIN ;
```

Beispiel (Strukturen als Komponenten einer Struktur):

Eine Personaldatei enthalte die Beschreibungen von Mitarbeitern; diese Beschreibungen haben jeweils folgende Struktur:



Die Komponenten Name und Adresse sind selbst Strukturen, nämlich Unterstrukturen der Hauptstruktur Person; diese kann so vereinbart werden:

```

DCL   Person
[
  STRUCT
  Name
  [
    STRUCT
    Vorname   CHAR (10),
    Nachname  CHAR (15)
  ],
  Adresse
  [
    STRUCT
    Str       CHAR (15),
    PLZ       FIXED,
    Ort       CHAR (15)
  ],
  Geschlecht CHAR (1),
  ...
];
...
  
```

Beispiel (Feld von Strukturen):

DCL Beitrag(20) **STRUCT** ... ;

Beispiel (Zugriffe auf Komponenten komplexer Strukturen):

```

Person.Name.Vorname := 'OTTO' ;
IF Beitrag(i).schon_gesendet (j) THEN ... FIN ;
  
```

6.12 Typvereinbarung (TYPE)

Syntax:

Ein neuer Datentyp kann wie folgt vereinbart werden:

```
Typ-Vereinbarung ::=  
    TYPE Bezeichner$für-Typ { einfacher-Typ | Typ-Struktur }
```

Semantik:

Die Vereinbarung eines neuen Datentyps muß vor seiner Benutzung erfolgen. Im übrigen können auf Modulebene vereinbarte Datentypen in allen Tasks und Prozeduren des Moduls benutzt werden; die Benutzbarkeit lokal vereinbarter Datentypen richtet sich nach den Regeln der Blockstruktur.

Im Gegensatz zu globalen Objekten können neu vereinbarte Datentypen nur in dem Modul benutzt werden, in dem sie vereinbart sind. Typvereinbarungen für globale Objekte müssen also in allen diese Objekte benutzenden Modulen wiederholt werden.

Erläuterungen:

Eine bestimmte Struktur kann beispielsweise als Parameter oder als Unterstruktur in anderen Strukturen oder als Typ der Übertragungsdaten einer Datenstation auftreten. Dabei muß jedesmal der Typ dieser Struktur angegeben werden, was bei komplexen Strukturen umfangreichen Schreibaufwand erfordern kann. Deshalb und zur Erhöhung der Lesbarkeit des Programms kann der Typ einer Struktur unter einem frei wählbaren Bezeichner als neuer Datentyp vereinbart und unter diesem Bezeichner z.B. dazu benutzt werden, Variablen dieses Datentyps zu vereinbaren.

Beispiele (Vereinbarung von Datentypen):

(1) PROBLEM ;

```

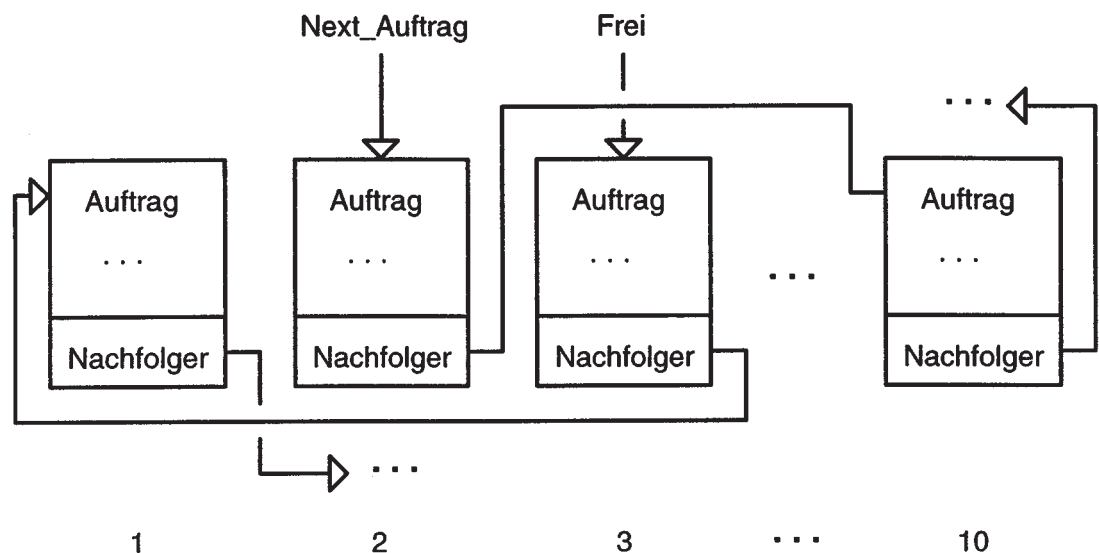
...
TYPE Nachricht STRUCT
  [ (Identnr, Archiv) FIXED,
    schon_gesendet (3) BIT (1),
    (Anfang, Ende) FIXED,
    Originalton BIT (1),
    Textlaenge FIXED,
    Text CHAR (200) ];

DCL Inhalt_MAZ DATION INOUT Nachricht ... ;

Koord: TASK ;
      DCL Beitrag Nachricht ;
      ...
      READ Beitrag FROM Inhalt_MAZ ;
      ...
      END ; ! Koord
...

```

- (2) Eine Task steuert bestimmte Geräte; die Steuerungsaufträge erhält sie von einer anderen Task. Weil zeitweise mehr als ein Auftrag vorliegen kann und die Aufträge unterschiedliche Dringlichkeit haben können, werden sie in Form einer Warteschlange gepuffert. Die Struktur der Aufträge sei vom (neu vereinbarten) Typ Auftragstyp; dieser enthalte auch das Dringlichkeitskriterium, nach dem neue Aufträge eingeordnet werden. Die Referenz-Variablen Next_Auftrag bzw. Frei zeigen auf den nächsten zu bearbeitenden Auftrag bzw. den nächsten freien Platz in der Warteschlange Auftragskette; ausgehend von Frei seien die freien Plätze ebenfalls verkettet. Die Warteschlange sei höchstens 10 Elemente lang.



Die Auftragskette kann folgendermaßen vereinbart werden:

```
...  
TYPE Kettenelement STRUCT  
    [ Auftrag Auftragstyp,  
      ...,  
      Nachf REF Kettenelement  
    ];  
DCL Auftragskette ( 10 ) Kettenelement,  
    (Next_Auftrag, Frei) REF Kettenelement ;  
...
```

Nach erfolgter Initialisierung und dem Einordnen verschiedener Aufträge kann ein fertig bearbeiteter Auftrag wie folgt aus der Warteschlange entfernt werden (Hilf sei dabei eine Variable vom Typ REF Kettenelement).

```
...  
Hilf := Next_Auftrag ;  
Next_Auftrag := Next_Auftrag.Nachf ;  
Hilf.Nachf := Frei ;  
Frei := Hilf ;  
...
```

Diese Anweisungen sind auch weitere Beispiele für implizites Deferenzieren.

Die Auftragskette kann so initialisiert werden:

```
FOR i TO 9  
    REPEAT  
        Auftragskette (i).Nachf := Auftragskette (i + 1) ;  
    END ;  
Auftragskette (10).Nachf := NIL ;
```

Beispiel (Verwendung eines vereinbarten Datentyps in verschiedenen Moduln):

```
MODULE (Modul A) ;  
PROBLEM ;  
  
    TYPE T STRUCT [...];  
    ...  
    DCL x T GLOBAL ;  
    ...  
MODEND ;
```

```
MODULE (Modul B) ;  
PROBLEM ;  
  
    TYPE T STRUCT [...];  
    ...  
    SPC x T GLOBAL ;  
    ...  
MODEND ;
```

6.13 Das Initialisierungsattribut (INITIAL)

Syntax:

Initialisierungsattribut ::=
 { **INITIAL** | **INIT** } (Init-Element [, Init-Element] ...)

Init-Element ::=
 Konstante
 | Bezeichner\$benannte-Konstante
 | konstanter-Ausdruck
 | Bezeichner\$Task, Prozedur, Struktur

Konstante ::=
 ganze-Zahl
 | Gleitpunktzahl
 | Bitkettenkonstante
 | Zeichenkettenkonstante
 | Uhrzeitenkonstante
 | Dauerkonstante
 | Nullzeigerkonstante

Semantik:

Das Initialisierungsattribut erlaubt, Variablen bei ihrer Deklaration Anfangswerte zuzuweisen. Für Variablen mit Zuweisungsschutz ist dies die einzige Möglichkeit der Wertzuweisung. Das Initialisierungsattribut wird bei der jeweiligen Deklaration als letztes Attribut angegeben.

Der Typ des angegebenen Init-Elementes muß mit dem Typ der deklarierten Variablen gemäß den Regeln für die Zuweisung verträglich sein.

Felder und Strukturen können ebenfalls an der Deklarationsstelle initialisiert werden. Für alle Strukturkomponenten muß jeweils ein Initialwert angegeben werden. Bei Feldern darf die Initialisierungsliste kürzer als die Anzahl der Feldelemente sein. In diesem Fall wird der letzte Initialwert für die restlichen Elemente wiederholt benutzt.

Die Initialisierungsliste kann außer Konstanten auch Bezeichner von benannten Konstanten sowie vom Kompilierer berechenbare Ausdrücke enthalten.

Erläuterungen:

Beispiel (Zuordnung der Elemente der Initialisierungsliste):

```
...  
DCL Anzahl_Geraete FIXED INIT(12),  
      (UGR, OGR) FIXED INIT(2,15) ;  
...  
FOR i FROM 1 TO Anzahl_Geraete  
...
```

Hier haben die Variablen Anzahl_Geraete, UGR und OGR die Werte 12, 2 und 15, d.h. die Elemente einer angegebenen Konstanten-Liste werden in der Reihenfolge der Niederschrift der angegebenen Bezeichner-Liste zugeordnet.

Beispiel (Initialisierung von Strukturen und Feldern):

```
...  
DCL Adresse STRUCT [   Postleitzahl FIXED,  
                        Ort CHAR(20),  
                        Strasse CHAR(20),  
                        HausNr FIXED ]  
      INIT ( 21337 , 'Lueneburg' , 'Erbstorfer Landstr.' , 14 ) ;  
DCL Farben      ( 3 ) CHAR(7) INIT ( 'schwarz' , 'weiss ' , 'rot ' ) ;  
DCL alles_Sechsen ( 6 ) FIXED INIT ( 6 ) ;  
...
```

Beispiel (Initialisierung von Zeigern):

```
DCL TaskZeiger REF TASK INIT (StartTask);
```

6.14 Zuweisungsschutz (INV)

Syntax:

Zuweisungsschutz ::=
INV

Semantik:

Skalare Variablen, Felder und Strukturen können mit einem Zuweisungsschutz mittels des Attributs INV (von "invariant"), deklariert werden, um - abgesehen von der Initialisierung - Zuweisungen an diese Objekte zu verbieten. Das Attribut INV wird unmittelbar vor dem Typ-Attribut angegeben.

Ein einmal für ein Objekt in seiner Deklaration vereinbarter Zuweisungsschutz kann nicht aufgehoben werden; er muß deshalb in Spezifikationen dieses Objekts berücksichtigt werden, aber auch bei Übertragungen als aktueller Prozedurparameter oder bei der Benutzung als Wert von Referenz-Variablen. Andererseits darf auf eine dieser Arten ein bei der Deklaration noch nicht vereinbarter Zuweisungsschutz neu entstehen.

Erläuterungen:

Benannte Konstanten (INV-Konstanten)

Mit dem Attribut INV deklarierte Objekte vom Typ FIXED, BIT, CHAR(1), CLOCK und DURATION behandelt der Kompilierer als (benannte) Konstanten. Die Namen dieser Objekte dürfen nach ihrer Deklaration mit Initialisierung bei der Deklaration und Initialisierung weiterer Objekte benutzt werden. Dabei dürfen sie sowohl als Dimensionsgrenzenangabe bei Felddeklarationen, als auch in Initialisierungslisten eingesetzt werden.

Da der Kompilierer für die benannten Konstanten keine Laufzeit-Objekte erzeugt, dürfen sie nicht einer Referenz-Variablen zugewiesen werden. Ebenso können sie nicht als aktueller Parameter einer Prozedur benutzt werden, wenn der formale Parameter mit dem IDENTICAL-Attribut oder als Referenzvariable spezifiziert wurde.

Beispiel (Deklaration einer skalaren Variablen mit Zuweisungsschutz):

```
...  
DCL  Pi INV FLOAT INIT (3.141) ;
```

```
Pi := 3 ; /* Ergibt Fehlermeldung des Kompilierers */
```

Beispiel (Inkonsistenzen bei der Verwendung von INV):

```
...
P:  PROC ( A (,) INV FIXED IDENT , x  FLOAT IDENT ) ;
      /* Prozedurkörper */
      END ; ! P

DCL  Tab (10,20)  FIXED,
      Pi  INV  FLOAT  INITIAL (3.141),
      R1  REF  FLOAT,
      R2  REF  INV  FIXED ;

...

SPC  pi  FLOAT  IDENT  (Pi) ;      /* Falsch */

...

CALL P (Tab, Pi) ;                /* Falsch */
R1 := Pi ;                        /* Falsch */
R2 := Tab (5, 7) ;
```

Die Spezifikation von pi ist falsch, weil Zuweisungen an pi erlaubt werden und somit der Zuweisungsschutz von Pi unterlaufen wird.

Der Aufruf von P ist falsch, weil der zu Pi gehörige formale Parameter x ohne Zuweisungsschutz INV vereinbart ist und somit im Körper von P über eine Zuweisung an x eine (verbotene) Wertänderung von Pi erfolgen könnte.

Dagegen ist im selben Aufruf die Parameterübertragung von Tab an A korrekt, da lediglich ein bisher noch nicht vereinbarter Zuweisungsschutz für Tab entstanden ist.

Die Zuweisung R1 := Pi ist unzulässig, da andernfalls der Zuweisungsschutz für Pi durch (erlaubte) Zuweisungen an CONT R1 aufgehoben werden könnte. Die Zuweisung R2 := Tab(5,7) ist dagegen zulässig; eine Wertänderung des Tabellenelementes über CONT R2 ist nicht möglich.

Beispiel (Benannte Konstanten):

Ein Überwachungsprogramm soll für verschiedene Projekte mit unterschiedlichen Anzahlen von unterlagerten Steuerungen angepaßt werden. Die Anzahl der Nachrichtenpuffer und zugehörigen Synchronisierungsvariablen lassen sich durch Ändern der Konstante AnzahlSteuerungen leicht anpassen:

```
...
DCL  AnzahlSteuerungen  INV  FIXED  INIT (11) ;

DCL  NachrichtenPuffer  ( AnzahlSteuerungen )  CHAR(100);
DCL  Pufferzugriff      ( AnzahlSteuerungen )  BOLT;

...
```

Für ein weiteres Projekt mit einer anderen Anzahl von Steuerungen muß nur die benannte Konstante AnzahlSteuerungen den aktuellen Gegebenheiten angepaßt werden. Die davon abhängigen Speicherbereiche werden durch den Kompilierer angepaßt.

6.15 Konstante Ausdrücke

Syntax:

```
konstanter-Ausdruck ::=
    { + | - } Gleitpunktzahl
  | { + | - } Dauerkonstante
  | konstanter-FIXED-Ausdruck

konstanter-FIXED-Ausdruck ::=
    Term [ { + | - } Term ] ...

Term ::=
    Faktor [ { * | // | REM } Faktor ] ...

Faktor ::=
    [ + | - ]
    {
        ganze-Zahl
      | ( konstanter-FIXED-Ausdruck )
      | TOFIXED { Zeichenkettenkonstante$der-Länge-1 | Bitkettenkonstante }
      | Bezeichner$benannte-FIXED-Konstante
    }
    [ FIT konstanter-FIXED-Ausdruck ]
```

Semantik:

Konstante Ausdrücke vom Typ FIXED können z.B. an folgenden Stellen verwendet werden:

- bei der Angabe von Dimensionsgrenzen bei Felddeklarationen,
- bei Genauigkeitsangaben (bei FIXED- und FLOAT-Objekten),
- bei Längenangaben (bei CHAR- und BIT-Objekten),
- in Initialisierungslisten (auch bei PRESET),
- bei Prioritätsangaben bei Task-Deklarationen/-Anweisungen
- in den Konstantenlisten der CASE-Anweisung und
- auf der rechten Seite einer Zuweisung.

Erläuterungen:

An allen Stellen, an denen der Kompilierer Konstanten erwartet, dürfen auch konstante Ausdrücke stehen. Diese Ausdrücke dürfen als Operanden nur Konstanten (auch benannte Konstanten) enthalten. Sie werden durch den Kompilierer ausgewertet und das Ergebnis an den entsprechenden Stellen eingesetzt.

Oft sind mehrere Objekte eines Programmes (Variablen, Konstanten) voneinander abhängig. Läßt sich diese Abhängigkeit durch eine einfache Formel beschreiben, kann die Konfigurierung eines Programmes evtl. durch Anpassen einiger weniger Konstanten erledigt werden.

Beispiel (Anwendung konstanter Ausdrücke):

Es soll eine Warnung ausgegeben werden, wenn ein Puffersystem bis auf die letzten 10% ausgefüllt ist:

```
...  
DCL MaxPuffer      INV   FIXED INIT ( 1000 ) ;  
DCL WarnGrenze    INV   FIXED INIT ( MaxPuffer - MaxPuffer // 10 ) ;  
DCL Puffer        ( MaxPuffer ) FIXED ;  
DCL aktuellerPufferIndex FIXED INIT ( 1 ) ;  
...  
aktuellerPufferIndex := aktuellerPufferIndex + 1 ;  
IF aktuellerPufferIndex > WarnGrenze  
THEN /* Warnung ausgeben */  
FIN ;  
...
```