# Änderungsprotokoll

Diese Beschreibung gilt für Pascal-XT auf den Betriebssystemen BS2000 und Sinix. Jeder Pascal-XT Compiler mit einer Versionsbezeichnung 2.1x, unabhängig vom Betriebssystem, akzeptiert genau den in diesem Manual beschriebenen Funktionsumfang.

Die folgende Tabelle gibt nur Kapitel an, die fachliche Änderungen enthalten.

| Kapitel | Stichwort | neu | geän-dert | ent-fallen |
|---|---|---|---|---|
| 3.8 | Pseudokommentare "{%..}" | | | x |
| 6.3.3 | Syntax von Variantteil korrigiert | | x | |
| 9.3.1 | Beispiele mit MOD-Operator korrigiert | | x | |
| 14.3 | Beispiele zur Fehlerpropagierung | x | | |
| 15.6 | Setmax und Setmin mit leerer Menge | | x | |
| 15.11 | Raise (0) - Fehlerpropagierung | x | | |
| 16.1 | Argument L0 bei Option Standard | x | | |

Die bisher im Text verwendete kursive Schreibweise wurde aus Konsistenzgründen geändert.

# Inhalt


| | | Seite |
|---|---|---|
| **1** | **Einleitung** | **1** |
| 1.1 | Hinweise zum Lesen des Manuals | 1 |
| 1.2 | Die Pascal-XT Compiler-Familie | 4 |
| 1.3 | Zur Sprache Pascal | 5 |
| 1.4 | Erfüllung der Pascal-Norm | 7 |
| **2** | **Definitionen** | **9** |
| 2.1 | Metasprache | 9 |
| 2.2 | Implementierungsdefinierte und implementierungsabhängige Eigenschaften | 11 |
| 2.3 | Klassifikation von Fehlern | 12 |
| **3** | **Lexikalische Elemente** | **13** |
| 3.1 | Allgemeines | 13 |
| 3.2 | Spezialsymbole | 14 |
| 3.3 | Bezeichner | 15 |
| 3.4 | Direktiven | 17 |
| 3.5 | Zahlen | 18 |
| 3.6 | Marken | 20 |
| 3.7 | Zeichenketten | 21 |
| 3.8 | Trennung lexikalischer Einheiten und Kommentare | 23 |
| **4** | **Markendeklarationen** | **25** |
| **5** | **Konstanten** | **27** |
| 5.1 | Konstantendefinitionen | 27 |
| 5.2 | Vordefinierte Konstanten | 30 |
| **6** | **Typen** | **33** |
| 6.1 | Typdefinitionen | 35 |
| 6.2 | Einfache Typen | 37 |
| 6.2.1 | Integer-Typen | 37 |
| 6.2.2 | Real-Typen | 38 |
| 6.2.3 | Der Typ Char | 40 |
| 6.2.4 | Der Typ Boolean | 41 |
| 6.2.5 | Aufzählungstypen | 42 |
| 6.2.6 | Teilbereichstypen | 44 |

6.3 Strukturierte Typen 45
6.3.1 ARRAY-Typen 46
6.3.2 Zeichenkettentypen 48
6.3.2.1 Zeichenkettentypen fester Länge 48
6.3.2.2 Zeichenkettentypen variabler Länge (String-Typen) 49
6.3.3 RECORD-Typen 51
6.3.3.1 RECORD-Typen mit Varianten 52
6.3.3.2 RECORD-Typen mit Angaben zur Speicherdarstellung 55
6.3.4 SET-Typen 58
6.3.5 FILE-Typen 60
6.3.5.1 Allgemeine Dateien 60
6.3.5.2 Der FILE-Typ Text 61
6.4 Zeigertypen 63
6.5 Generische Typen 65
6.5.1 Der vordefinierte FILE-Typ Any File 65
6.5.2 Der vordefiniete Zeigertyp Pointer 65
6.5.3 Der vordefinierte Typ Any Type 66
6.6 Identität und Verträglichkeit von Typen 67
6.6.1 Identität von Typen 67
6.6.2 Verträgliche Typen 69
6.6.3 Zuweisungsverträglichkeit von Typen 71
6.7 Attribute von Typen 75

**7 Variablen 77**
7.1 Variablendeklaration 78
7.2 Einteilung von Variablen 80
7.3 Definierte und undefinierte Werte von Variablen 82

**8 Prozeduren und Funktionen 85**
8.1 Prozedurdeklaration 87
8.2 Funktionsdeklaration 89
8.3 Inline-Unterprogramme 93
8.4 Entry-Unterprogramme 94
8.5 Parameter 95
8.5.1 Wertparameter 96
8.5.2 Variablenparameter 98
8.5.3 Parameterprozeduren und -funktionen 101
8.5.4 Konformreihungs-Parameter 104
8.6 Die Direktiven und Prozedur- bzw. Funktionsidentifikationen 108
8.7 Aufrufe von Unterprogrammen 111

**9 Ausdrücke 115**
9.1 Allgemeines 115
9.2 Statische Ausdrücke 118
9.3 Operatoren 119

9.3.1 Arithmetische Operatoren 120
9.3.2 Boolesche Operatoren 127
9.3.3 Mengenoperatoren 129
9.3.4 Vergleichsoperatoren 131
9.4 Mengenbildner 135
9.5 Aggregate 138
9.5.1 ARRAY-Aggregate 139
9.5.2 RECORD-Aggregate 141
9.6 Objekte 143
9.6.1 Allgemeines 143
9.6.2 Indizierte Objekte 145
9.6.3 Selektierte Objekte 148
9.6.4 Dereferenzierte Objekte 152
9.6.5 Puffervariable 153

**10 Anweisungen 157**
10.1 Einfache Anweisungen 158
10.1.1 Leeranweisung 158
10.1.2 Zuweisungen 160
10.1.3 Prozeduraufrufe 164
10.1.4 GOTO-Anweisung 165
10.1.5 EXIT-Anweisung 166
10.1.6 RETURN-Anweisung 167
10.2 Verbundanweisung 168
10.3 Bedingte Anweisungen 169
10.3.1 IF-Anweisung 169
10.3.2 CASE-Anweisung 172
10.4 Wiederholungsanweisungen 174
10.4.1 REPEAT-Anweisung 175
10.4.2 WHILE-Anweisung 177
10.4.3 FOR-Anweisung 179
10.5 WITH-Anweisung 182

**11 Hauptprogramm und Pakete 187**
11.1 Hauptprogramm 187
11.2 Pakete 190
11.2.1 Paket-Spezifikation 193
11.2.2 Paket-Implementierung 195
11.3 Kontextspezifikation 196
11.3.1 WITH-Liste 196
11.3.2 USE-Liste 198
11.4 Private Typen 199
11.5 Programmparameter 200

| | | |
|---|---|---|
| **12** | **Sichtbarkeitsregeln** | **203** |
| 12.1 | Blöcke | 203 |
| 12.2 | Definitionspunkte und Gebiete von Bezeichnern und Marken | 206 |
| 12.3 | Gültigkeitsbereiche und Verwendung von Bezeichnern | 209 |
| **13** | **Programmstruktur, Übersetzungen und Ausführungen** | **213** |
| 13.1 | Programmstruktur | 213 |
| 13.2 | Übersetzungseinheiten und Übersetzungsreihenfolge | 216 |
| 13.3 | Ausführen eines Programms bzw. Unterprogramms | 218 |
| 13.3.1 | Ausführung eines Blocks | 218 |
| 13.3.2 | Ausführung eines Unterprogramms | 219 |
| 13.3.3 | Ausführen eines Programms | 220 |
| **14** | **Ausnahmebehandlung** | **221** |
| 14.1 | Vordefinierte und benutzerdefinierte Ausnahmen | 222 |
| 14.2 | Ausnahmebehandlungs-Teil (EXCEPTION-Teil) | 224 |
| 14.3 | Behandlung von Ausnahmen | 226 |
| 14.4 | Ausnahmebehandlung und Optimierung | 232 |
| **15** | **Vordefinierte Unterprogramme** | **233** |
| 15.1 | Unterprogramme zur Dateiverarbeitung | 234 |
| 15.2 | Unterprogramme zur Haldenverwaltung | 261 |
| 15.3 | Unterprogramme zur Zeichenkettenverarbeitung | 272 |
| 15.4 | Arithmetische Funktionen | 281 |
| 15.5 | Umwandlungsfunktionen | 284 |
| 15.6 | Ordinalfunktionen | 286 |
| 15.7 | Boolesche Funktionen | 289 |
| 15.8 | Transferprozeduren | 290 |
| 15.9 | Attributfunktionen | 294 |
| 15.10 | Funktion zur unüberprüften Typkonvertierung | 299 |
| 15.11 | Unterprogramme zur Ausnahmebehandlung | 300 |
| 15.12 | Prozedur zur expliziten Paketinitialisierung | 302 |
| **16** | **Steueranweisungen an den Compiler** | **303** |
| 16.1 | Globale Optionen | 305 |
| 16.2 | Lokale Optionen | 307 |
| **17** | **Das Paket-Konzept** | **311** |
| 17.1 | Anwendungen für Pakete | 316 |
| 17.1.1 | Sammlung von Deklarationen | 316 |
| 17.1.2 | Sammlung von Unterprogrammen | 318 |
| 17.1.3 | Abstrakter Datentyp | 319 |
| 17.1.4 | Automat | 320 |

| | | |
|---|---|---|
| **18** | **Ausnahmebehandlungskonzept** | **323** |
| **19** | **Ein-/Ausgabe** | **329** |
| 19.1 | Zuordnung einer physikalischen Datei | 332 |
| 19.2 | Eröffnen von Dateien zum Lesen oder Schreiben | 334 |
| 19.3 | Die Lese-/Schreibprozeduren Get und Put | 335 |
| 19.4 | Die Lese-/Schreibprozeduren Read und Write | 338 |
| 19.5 | Textdateien | 342 |
| 19.5.1 | Lesen von einer Textdatei | 344 |
| 19.5.2 | Lesen von der Datensichtstation | 355 |
| 19.5.3 | Schreiben in eine Textdatei | 356 |
| **20** | **Dynamische Daten und Speicherverwaltung** | **359** |
| | Beispiel 1: Einfach verkettete Liste | 362 |
| | Beispiel 2: Binärbaum | 364 |
| **A** | **Anhang** | **369** |
| A.1 | Syntax von Pascal-XT | 369 |
| A.2 | Vordefinierte Bezeichner | 378 |
| A.3 | Bedeutung der Wortsymbole und Spezialsymbole | 381 |
| A.4 | Erweiterungen von Pascal-XT gegenüber der Norm | 384 |
| A.5 | Liste der Laufzeitfehler | 386 |
| A.6 | Implementierungsdefinierte Eigenschaften | 396 |
| A.7 | Implementierungsabhängige Eigenschaften | 399 |
| A.8 | Paket CLOCK | 401 |
| **Literatur** | | **403** |
| **Stichwörter** | | **405** |

# Einleitung

Die vorliegende Sprachbeschreibung des Programmiersystems Pascal-XT enthält die Beschreibung der Elemente der Sprache Pascal-XT, die allen Implementierungen dieses Systems gemeinsam sind. Auf implementierungsdefinierte Besonderheiten wird an den entsprechenden Stellen hingewiesen.

Allgemeine EDV-Kenntnisse sowie die wesentlichen Grundlagen der Booleschen Algebra sind für das Verständnis der Sprachbeschreibung ausreichend.

Als Grundlage der Beschreibung dient die Pascal-DIN-Norm 66256 (siehe Literaturverzeichnis [3]). Über diese Norm hinausgehende Elemente und Eigenschaften sind durch Rasterung hervorgehoben.

## Hinweise zum Lesen des Manuals

**Aufbau des Manuals**

Die Sprachbeschreibung ist als Nachschlagewerk gedacht, in dem die Spracheigenschaften möglichst exakt beschrieben werden. Zum Erlernen von Pascal gibt es eine Reihe von guten Lehrbüchern (siehe z. B. [5]). Im Kapitel 2 werden die notwendigen Definitionen zur Beschreibung der Sprache angegeben. In den Kapiteln 3 bis 16 werden die Eigenschaften der Sprache beschrieben. In den Kapiteln 17 bis 20 werden Hintergründe und Anwendungsmöglichkeiten einiger Sprachkonzepte erklärt, die erfahrungsgemäß Schwierigkeiten bereiten oder durch Pascal-XT neu hinzugekommen sind.

Am Ende eines jeden Abschnitts befindet sich eine Querverweisliste auf weitere, zum Verständnis dieses Abschnitts relevante Abschnitte. Im Anhang befinden sich Zusammenfassungen in Tabellenform.

Literaturhinweise werden im Text in Kurztiteln angegeben. Der vollständige Titel jeder Druckschrift, auf die verwiesen wird, ist im Literaturverzeichnis aufgeführt. Daran anschließend finden Sie Hinweise zur Bestellung von Manualen.

### Syntax und Semantik

Ein Pascal-Programm besteht aus einer endlichen Folge von Symbolen, die durch die Sprachdefinition vorgegeben sind oder nach gewissen Vorschriften vom Programmierer selbst gebildet werden. Die Syntax der Sprache schreibt vor, welche Symbole bzw. Symbolkombinationen in dieser Folge an welcher Stelle kommen dürfen. Da man die Syntax exakt angeben kann, wird sie als Ausgangspunkt der Sprachdefinition benutzt. Zur Beschreibung der Syntax wird die Backus-Naur-Form (siehe 2.1) verwendet.

Die Bedeutung eines Programms (seine Semantik) kann jedoch nicht aus den syntaktischen Regeln abgeleitet werden. Sinnvollerweise wird die Semantik der Sprache jedoch ausgehend von ihrer Syntax erklärt. In diesem Manual findet man zu jeder Sprachkonstruktion zunächst die entsprechenden Syntaxformeln, aus denen dann verbal und mit Hilfe von Programmbeispielen die zugehörige Semantik beschrieben wird.

Oft wird das Zusammensuchen der entsprechenden Syntaxformeln aus anderen Abschnitten des Manuals notwendig sein. Hilfe hierbei leistet die alphabetisch zusammengefaßte Syntax im Anhang A.1.

### Groß-/Kleinschreibung

Zur besseren Lesbarkeit wird für die Wortsymbole und Bezeichner bei der Wahl von Groß-, Klein- oder gemischter Schreibweise folgende Schreibregel eingehalten:

- Wortsymbole (Schlüsselwörter) → durchgehend groß ("PROGRAM")
- vordefinierte Bezeichner → großer Anfangsbuchstabe ("Integer")
- anwenderdefinierte Bezeichner → Kleinschreibung ("meine_zahl")

### Verwendung von Binde- und Unterstrichen

Da Pascal eine international verbreitete Programmiersprache ist, sind die Wortsymbole (3.2) und vordefinierten Bezeichner (Anhang A2, A3) in englisch gehalten. Wenn aufgrund des deutschen Manualtextes deutsch-englische Mischworte als Metabezeichner (2.1) entstehen, so wird der Bindestrich zur Trennung verwendet.

Sonstige Metabezeichner, die sich aus mehreren Worten zusammensetzen, werden durch Unterstriche miteinander verbunden. Bei der Verwendung dieser Metabezeichner im Text werden die Unterstriche durch Leerzeichen ersetzt.

### Präfixe in Kursivschrift

Ein Metabezeichner mit einem Präfix in kursiver Darstellung ist aus Sicht der Syntax äquivalent zu einem Metabezeichner ohne diesen Präfix. Zum Beispiel sind *Typ*-Name und *Variablen*-Name identisch mit Name. Der kursiv geschriebene Präfix soll lediglich semantische Eigenschaften zum Ausdruck bringen. So bedeutet z. B. *Ordinal*-Konstante, daß an dieser Stelle eine Konstante von einem Ordinaltyp verlangt wird.

Die Angaben in Kursivschrift sind ausschließlich in den Syntaxregeln angegeben. Im Text werden die Präfixe nicht kursiv dargestellt.

### Hinweise

Hinweise gehören nicht zur Sprachdefinition. Sie liefern weitere Informationen zum Verständnis oder geben Hinweise zur Programmierung.

### Aufbau des Stichwortverzeichnisses

Die Stichwörter sind, wenn es sich um zusammengesetzte Begriffe handelt, derart im Index-Verzeichnis aufgeführt, daß der Hauptbegriff vorn steht, und dahinter die Vorsilben. In einigen Fällen werden die Teile zusammengesetzter Begriffe gleich signifikant sein (auch wenn im Kontext verschieden), so daß für diese auch mehrere Einträge unter verschiedenen Anfangsbuchstaben existieren können.

*Querverweise*

Backus-Naur-Form: 2.1
Metabezeichner: 2.1
vordefinierte Bezeichner: A.2, A.3
Konzepte: 17 bis 20
vollständige Syntax: A.1

# Die Pascal-XT Compiler-Familie

Die Pascal-XT-Compiler-Familie ist eine Familie von Pascal-Compilern für verschiedene Siemens-Rechner, die alle den in diesem Manual beschriebenen Sprachumfang akzeptieren.

Diese Einheitlichkeit der Pascal-XT-Compiler-Familie ermöglicht ein problemloses Portieren von Pascal-XT-Programmen von einem Rechner zum anderen. Insbesondere kann z. B. auf einem Arbeitsplatzrechner die Software für einen Großrechner entwickelt werden. Umgekehrt kann in Pascal-XT auf einem Großrechner entwickelte Software ohne weiteres auch auf einem Arbeitsplatzrechner übersetzt werden und dort ablaufen.

# Zur Sprache Pascal

### Entstehung und Normung

Die Programmiersprache Pascal wurde von Prof. Niklaus Wirth (ETH Zürich) entwickelt. Die erste Beschreibung der Sprache wurde 1970 veröffentlicht. Im Jahr 1974 erschien der "Pascal User Manual and Report" [4] von K. Jensen und N. Wirth. Dieser Report wurde als die Definition der Sprache angesehen. Leider war diese Beschreibung in einigen Punkten zu ungenau. Diese Schwierigkeiten wurden erst mit der Normung von Pascal beseitigt.

Die Standardisierung von Pascal wurde 1977 in Großbritannien in Angriff genommen. Im November 1983 wurde die erste internationale Pascal-Norm ISO 7185 verabschiedet. Im März 1984 wurde die deutsche Pascal-Norm DIN 66256 [3] in deutscher Sprache, aber inhaltsgleich zur ISO-Norm verabschiedet. Sie ist die Grundlage des vorliegenden Handbuchs. Für die Abdruckgenehmigung sei dem Springer Verlag Berlin/New York/Tokio herzlich gedankt.

### Spracheigenschaften

Pascal ist eine höhere Programmiersprache, die von N. Wirth vor allem für die Lehre in den Bereichen Algorithmenentwurf und Programmiermethodik entwickelt wurde. Pascal spielt daher bei der Veröffentlichung von Algorithmen eine große Rolle und wird auch in der Ausbildung an den Universitäten eingesetzt.

Der wesentliche Ansatzpunkt von Pascal ist die Bereitstellung problemorientierter Datenstrukturen. Viele andere Sprachen kennen nur implizit Datenstrukturen oder solche, wie sie von der zur Verfügung stehenden Hardware angeboten werden, wie z. B. Fixed(31) Binary. Pascal hingegen bietet ein durchgehendes Konzept, maschinenunabhängige Datentypen durch den Benutzer definierbar zu machen. Wichtig ist dabei auch die Möglichkeit, Datentypen Namen zu geben und auf diese in anderen Datentypen Bezug nehmen zu können.
In einem Pascal-Programm werden Daten als Wertemengen beschrieben. Diese werden als Datentyp bezeichnet. Beispiele für elementare Datentypen sind:

- ein Bereich, wie z. B. -5..99
- Aufzählung von Werten, z. B.
  (montag, dienstag, mittwoch, donnerstag, freitag)
- ein vordefinierter Datentyp wie Boolean, Char, Integer, Real

Neben diesen elementaren Datentypen lassen sich komplexe Datenstrukturen definieren. Die wesentlichen Konstruktionen sind:

- Reihungen, Felder: ARRAY
- Verbunde, Datensätze: RECORD
- Mengen: SET
- Dateien: FILE

Daneben sind Zeigertypen (Verweise, Pointer) zur Bearbeitung von Listen- und Baumstrukturen definierbar, die eine ganze Welt verketteter Datenstrukturen eröffnen.

Das Block-Konzept mit Prozeduren und Funktionen erlaubt es, Programmieraufgaben wohlstrukturiert und nach der Methode der schrittweisen Verfeinerung zu lösen. Die Formatfreiheit der Sprache bietet die Möglichkeit, Programme übersichtlich zu gestalten.

Gemäß der Pascal-Norm können den Einsatzgebieten angepaßte Spracherweiterungen vorgenommen werden. Ziel ist es hierbei, die ursprünglichen Eigenschaften der Sprache beizubehalten, und doch durch sparsame, gezielte Ergänzungen ein kommerziell, industriell und wissenschaftlich noch besser verwertbares Programmierwerkzeug anzubieten.

### Entstehung und Ziel von Pascal-XT

Die Sprache Pascal-XT bietet eine Reihe von Erweiterungen gegenüber der Norm. Die neuen Konzepte von Pascal-XT entsprechen den heutigen Anforderungen an moderne Programmiersprachen. Sie basieren auf der Notwendigkeit, immer größere Softwaresysteme immer rationeller und in immer kürzerer Zeit entwickeln zu müssen. Es ist daher notwendig, die Software-Entwicklung in Teamarbeit von mehreren Mitarbeitern durchzuführen und auf bereits in anderen Projekten entwickelte wiederverwendbare und anpaßbare Softwarebausteine zurückzugreifen. Ferner basieren die Anforderungen an moderne Programmiersprachen auf der Notwendigkeit, Wartungskosten zu reduzieren. Die zu entwickelnde Software muß daher leicht verständlich und änderungsfreundlich sein.

Die Spracherweiterungen sind in diesem Handbuch durch Unterlegung gekennzeichnet. Im Anhang A.4 sind alle Erweiterungen stichwortartig zusammengefaßt.

# Erfüllung der Pascal-Norm

Pascal-XT erfüllt die Anforderungen der Stufen 0 und 1 von DIN 66256 (siehe aber auch Kap. 16.1)

- **Erweiterungen zu DIN 66256 Pascal**

  Alle Erweiterungen zu DIN 66256 Pascal sind in diesem Sprachmanual an den entsprechenden Stellen durch Unterlegungen hervorgehoben. Im Anhang A.4 sind die Erweiterungen nochmals zusammengefaßt.
  Bei Angabe der Steueranweisung "Standard" (siehe Kap. 16.1) wird nur die Norm (DIN 66256, Stufe 1 bzw. Stufe 0) akzeptiert. Erweiterungen werden zur Übersetzungszeit als Fehler gemeldet.

- **Fehler**

  Alle Fehler werden in der Sprachbeschreibung in den entsprechenden Abschnitten beschrieben und sind im Anhang A.5 nochmals tabellarisch zusammengefaßt. In den Benutzerhandbüchern [1,2] wird beschrieben, welche Fehler erkannt werden.

- **Implementierungsdefinierte Eigenschaften**

  Alle implementierungsdefinierten Eigenschaften werden in den entsprechenden Abschnitten beschrieben und sind im Anhang A.6 nochmals tabellarisch zusammengefaßt. Im Benutzerhandbuch [1] werden die einzelnen Eigenschaften spezifiziert.

- **Implementierungsabhängige Eigenschaften**

  Implementierungsabhängige Eigenschaften werden in den entsprechenden Abschnitten beschrieben und sind im Anhang A.7 nochmals tabellarisch zusammengefaßt.
  Die Ausnutzung solcher Eigenschaften ist nicht normkonform und daher fehlerhaft. Verstöße dieser Art werden jedoch nicht wie ein Fehler gemeldet.

*Querverweise*

| | |
|---|---|
| Implementierungsdefiniert: | 2.2, A.6 |
| Implementierungsabhängig: | 2.2, A.7 |
| Fehler: | 2.3, A.5 |
| Erweiterungen: | A.4 |
| Steueranweisungen: | 16 |

# Definitionen

## Metasprache

Die in diesem Handbuch verwendete Metasprache (der Sprache zur Beschreibung der Programmiersprache) zur Angabe der Syntax von Pascal beruht auf der Backus-Naur-Form. Tabelle 2-1 zeigt die Bedeutung der verschiedenen Metasymbole.

| Metasymbol | Bedeutung |
|---|---|
| = | ist definiert als |
| . | Ende der Definition |
| [ x ] | kein oder ein Auftreten von x |
| { x } | kein, ein- oder mehrfaches Auftreten von x |
| x \| y bzw. ( x \| y ) | alternativ x oder y |
| "xyz" | die Zeichenfolge xyz tritt in der Quelle auf (terminales Symbol) |
| Metabezeichner | bezeichnet eine syntaktische Einheit (nichtterminales Symbol) |

Tabelle 2-1: Metasprachliche Symbole

Ein Metabezeichner ist eine Folge von Buchstaben und einzelnen Bindestrichen bzw. Unterstrichen (siehe 1.1), die mit einem Buchstaben beginnt und endet.

Eine Folge von terminalen und nichtterminalen Symbolen in einer Syntax-Regel bedeutet die Verkettung des Textes, den die Symbole letztendlich (d. h. nach Ersetzung der Nichtterminalen gemäß ihrer Syntax-Regel) darstellen. In den Syntax-Regeln in Kapitel 3 ist diese Verkettung direkt, d. h. kein weiteres Zeichen darf dazwischen stehen. In den Syntax-Regeln in anderen Teilen des Manuals richtet sich die Verkettung nach den im Abschnitt 3.8 eingeführten Regeln.

Die Worte **"in"**, **"enthalten"** und **"unmittelbar enthalten"** besitzen, wenn sie zur verbalen Beschreibung einer Syntaxkonstruktion verwendet werden, folgende Bedeutung:

- **das x in einem y**

  bezieht sich auf jenes x, das unmittelbar auf der rechten Seite einer Syntax-Regel zur Definition von y auftritt.

- **ein y, das ein x enthält**

  bezieht sich auf jedes y, von welchem ein x unmittelbar oder mittelbar durch Anwendung von Syntax-Regeln abgeleitet werden kann.

- **das y, das ein x unmittelbar enthält**

  bezieht sich auf dasjenige y, das ein x enthält, aber kein anderes y enthält, welches dasselbe x enthält.

  Diese Konventionen zur Syntax dienen zur Darlegung von bestimmten Syntax-Anforderungen und der darauf aufbauenden Beschreibung der Semantik.

*Beispiel*

```
Feldliste      = [(Festteil [";" Variantteil] |
                              Variantteil) [";"] ].
Festteil       = Record-Abschnitt {";" Record-Abschnitt}.
```

Mögliche verbale Beschreibungen, die sich auf diese Syntax beziehen, wären:

Der **Festteil** in einer **Feldliste** L ...

Die **Feldliste**, die den **Record-Abschnitt** A enthält, ...
(A kann auch in einer eingeschachtelten Record-Definition enthalten sein)

Die **Feldliste**, die den **Record-Abschnitt** A unmittelbar enthält, ...

- **"...eines Integer-Typs"**

  Die Aussage "...eines Integer-Typs" ist gleichwertig mit der Aussage "...des Typs Short_Integer oder des Typs Long_Integer oder eines Teilbereichs davon".

- **"...eines Real-Typs"**

  Die Aussage "...eines Real-Typs" ist gleichwertig mit der Aussage "...des Typs Short_Real oder des Typs Long_Real".

## Implementierungsdefinierte und implementierungsabhängige Eigenschaften

Das vorliegende Manual beschreibt den vollständigen Leistungsumfang der Programmiersprache Pascal-XT. Die exakt beschriebenen Merkmale und Eigenschaften sind in den verschiedenen Implementierungen, in denen Pascal-XT existiert, identisch. In einigen Punkten gibt es aber implementierungsbedingte Besonderheiten. An den entsprechenden Stellen erfolgt ein entsprechender Hinweis.

- **Implementierungsdefinierte Eigenschaften:**

  Eine implementierungsdefinierte Eigenschaft kann für ein Pascal-System spezifisch sein, sie ist aber auf jeden Fall definiert. Die Beschreibung ist dem Benutzerhandbuch der jeweiligen Implementierung zu entnehmen.

- **Implementierungsabhängige Eigenschaften:**

  Eine implementierungsabhängige Eigenschaft kann für ein Pascal-System spezifisch sein, sie ist jedoch nicht notwendig definiert.

Unter **Pascal-System** (Pascal-Prozessor) ist dabei ein komplettes System zu verstehen, das einen Pascal-Quelltext als Eingabe akzeptiert, diesen zur Abarbeitung aufbereitet (Interpreter, Compiler mit Binder) und in der Lage ist, die Ausführung des so aufbereiteten Programms (siehe 13.3) vorzunehmen.

Bei der Portierung von Pascal-XT Programmen von einem Pascal-System auf ein anderes dürfen keine implementierungsdefinierten und implementierungsabhängigen Eigenschaften ausgenutzt werden.

*Querverweise*

Implementierungsdefiniert: A.6
Implementierungsabhängig: A.7
Ausführung eines Programms: 13.3

## Klassifikation von Fehlern

Bei der Abarbeitung eines Pascal-Quelltextes kann ein Pascal-System Fehler erkennen. Fehler werden in 2 Klassen eingeteilt.

a) Zur Übersetzungszeit erkannte Fehler

Verstöße gegen die Syntax und statische Semantik, wie sie in diesem Manual beschrieben sind, werden vom Pascal-XT Compiler zur Übersetzungszeit erkannt und gemeldet. Programme, die solche Verstöße enthalten, sind nicht ausführbar, da für sie kein Objektcode erzeugt wird. Diese Verstöße widersprechen der Sprachbeschreibung und werden deshalb nicht explizit als Fehler erwähnt.

Beispiele für derartige Verstöße:

- ";" vor ELSE in einer IF-Anweisung
- Verwendung nicht deklarierter Bezeichner
- Die Laufvariable in einer FOR-Anweisung ist nicht lokal definiert
- Anwendungen von Operatoren auf Operanden mit unverträglichen Typen
- Der statische Ausdruck in einer Konstantendefinition ist nicht berechenbar
  (z. B. CONST c = 2**31; führt zu einem Überlauf)

b) Zur Laufzeit erkannte Fehler (Laufzeitfehler)

Das sind Fehler, die wegen falscher Daten oder falscher Programmlogik während der Ausführung eines Programms auftreten. Die Erkennung dieser Fehler wird durch die Implementierung, evtl. in Abhängigkeit von der Check-Option, festgelegt. Die möglichen Laufzeitfehler werden in den einzelnen Abschnitten dieser Sprachbeschreibung explizit erwähnt und im Anhang A.5 nochmals tabellarisch zusammengefaßt.

Beispiele für derartige Fehler:

- Division durch Null
- Dereferenzierung eines NIL-Zeigers
- Überschreitung von Index- oder Teilbereichsgrenzen
- Lesen von einer Datei f, wenn bereits Eof (f) = True ist.

*Querverweise*

Check-Option: 16
Fehler: A.5

# Lexikalische Elemente

## Allgemeines

Die lexikalischen Elemente zur Bildung von Pascal-Programmen sind:

- Spezialsymbole
- Bezeichner
- Direktiven
- vorzeichenlose Zahlen
- Marken
- Zeichenketten

Die typographische Darstellung eines Buchstabens (Groß-/Kleinschreibung, Schriftart usw.) außerhalb von Zeichenketten hat keinen Einfluß auf die Bedeutung des Programms, in dem er auftritt.
Also ist ein Name in Pascal immer derselbe, ob er groß- oder kleingeschrieben wird.
Die in dieser Sprachbeschreibung verwendete Schreibweise ist in Abschnitt 1.1 beschrieben.

# Spezialsymbole

Die Spezialsymbole umfassen Sonderzeichen und Wortsymbole.

- **Sonderzeichen**

Die Bedeutung der Sonderzeichen, die sich in Abhängigkeit des Kontextes ändern kann, wird in den entsprechenden Kapiteln beschrieben.

```
Spezialsymbol =  "+"  | "-"  | "*"  | "/" | "=" | "<"
               | ">"  | "["  | "]"  | "." | "," | ":"
               | ";"  | "↑"  | "("  | ")" | "**" | "<>"
               | "<=" | ">=" | ":=" | ".." .
```

- **Wortsymbole (Schlüsselworte)**

Die Wortsymbole sind reservierte Namen, die nicht mehr als Bezeichner verwendet werden dürfen. Falls die Option "Standard" (Kapitel 16) eingeschaltet ist, sind die unterlegten Wortsymbole keine reservierten Bezeichner.

```
Wortsymbol =
      "AND"        | "ARRAY"     | "BEGIN"     | "BODY"     |
      "CASE"       | "CONST"     | "DIV"       | "DO"       |
      "DOWNTO"     | "ELSE"      | "END"       | "ENTRY"    |
      "EXCEPTION"  | "EXIT"      | "FILE"      | "FOR"      |
      "FROM"       | "FUNCTION"  | "GOTO"      | "IF"       |
      "IN"         | "INLINE"    | "LABEL"     | "MOD"      |
      "NIL"        | "NOT"       | "OF"        | "OR"       |
      "PACKAGE"    | "PACKED"    | "PROCEDURE" | "PROGRAM"  |
      "RECORD"     | "REPEAT"    | "RETURN"    | "SET"      |
      "THEN"       | "TO"        | "TYPE"      | "UNTIL"    |
      "USE"        | "VAR"       | "WHILE"     | "WITH" .
```

- **Ersatzdarstellungen**

Für einige Spezialsymbole gibt es Ersatzdarstellungen. Man kann wahlweise die eine oder die andere Darstellung, oder aber auch beide gemischt verwenden.

```
↑ = @ = ^    { = (*     } = *)     [ = (.     ] = .)
```

*Querverweis*

Optionen: 16
Bedeutung der Spezialsymbole: A.3

# Bezeichner

Die Syntax von Bezeichner lautet:

```
Bezeichner = Buchstabe { ["__"] ( Buchstabe | Ziffer ) }.

Buchstabe  = "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".

Ziffer     = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Für den Aufbau von Bezeichnern gelten also folgende Regeln:

- Bezeichner setzen sich zusammen aus Buchstaben, Ziffern und dem Unterstreichungszeichen ("_").
- Das 1. Zeichen muß ein Buchstabe sein.
- Alle Zeichen innerhalb eines Bezeichners sind für diesen signifikant.
- Bezeichner dürfen keine Wortsymbole sein.
- Groß- bzw. kleingeschriebene Buchstaben werden nicht unterschieden.
- Bezeichner dürfen beliebig lang sein (jedoch setzt das Zeilenende der Bezeichnerlänge eine Grenze).
- Es dürfen nicht 2 Unterstreichungszeichen aufeinanderfolgen.
- Das letzte Zeichen darf kein Unterstreichungszeichen sein.

Vordefinierte Bezeichner (z. B. Input, Integer) haben schon eine bestimmte Bedeutung, können aber vom Benutzer im nachhinein auch umdefiniert verwendet werden. Die vordefinierte Bedeutung besitzen sie schon vor dem 1. Zeichen des Programms, so daß eine etwaige Umdefinition schon im Vereinbarungsteil des Hauptprogramms vorgenommen werden kann.

*Beispiele für*

| gültige Bezeichner | ungültige Bezeichner | |
|---|---|---|
| Y | 5zahlen | (Ziffer am Anfang) |
| summe | gesamt-laenge | ("-" nicht erlaubt) |
| Summe1 | kein blank | (Blank nicht erlaubt) |
| ABC_4 | _summe | (Unterstrich am Anfang) |
| Feld_Inhalt | A__B | (2 Unterstriche hintereinander) |

*Querverweise*

Vordefinierte Bezeichner: A.2

## Direktiven

Folgende Direktiven werden untertützt:

```
Direktive = "forward" | "c" | "cobol" | "fortran"
          | "external" | "internal".
```

Direktiven treten nur in Prozedur- und Funktionsdeklarationen auf und stehen ersatzweise für den Prozedur- bzw. Funktionsblock. Sie sind keine vordefinierten Bezeichner. **Forward** ist die einzige von Norm-Pascal vorgeschriebene Direktive.

*Querverweise*

Direktiven: 8.1, 8.2, 8.6

# Zahlen

Die Syntax von Zahlen lautet:

```
vorzeichenlose_Integer-Zahl
                = Ziffernfolge | "#" Sedezimal-Ziffernfolge.

vorzeichenlose_Real-Zahl
                = Ganzteil "." Bruchteil ["e" Exponent]
                | Ganzteil "e" Exponent.

Ziffernfolge    = Ziffer {Ziffer}.

Ziffer          = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

Sedezimal-Ziffernfolge
                = Sedezimal-Ziffer {Sedezimal-Ziffer}.

Sedezimal-Ziffer
                = Ziffer|"a"|"b"|"c"|"d"|"e"|"f".

Ganzteil        = Ziffernfolge.

Bruchteil       = Ziffernfolge.

Exponent        = ["+" | "-"] Ziffernfolge
```

- **Zahlen eines Integer-Typs**

Vorzeichenlose Integer-Zahlen sind Ziffernfolgen. Sie stehen für Werte eines Integer-Typs. Da Pascal-XT zwei Integer-Typen kennt, besitzen die Zahlen, in Abhängigkeit ihres Wertes, den folgenden Typ:

– **Short_Integer** für Integer-Zahlen im Bereich von 0 bis 32767
– **Long_Integer** für Integer-Zahlen im Bereich von 32768 bis 2147483647
– In Norm-Pascal sind vorzeichenlose Integer-Zahlen immer vom Typ Integer.

Vorzeichenlose Integer-Zahlen im Programmtext sind lexikalische Einheiten und ihr Wert muß im Bereich 0 bis Long_Maxint (siehe 5.2) liegen. Integer-Zahlen mit Vorzeichen sind innerhalb von Programmen Ausdrücke (siehe 9) und keine lexikalischen Einheiten. Aus diesem Grund kann im Programmtext die Zahl -2147483648 nicht angegeben werden, da 2147483648 größer als Long_Maxint ist. Diese negative Zahl kann aber bei der Eingabe mit Read oder Readstring gelesen werden (siehe Kap. 15).

*Beispiele für vorzeichenlose Integer-Zahlen*

```
1
100
3056
```

### Angabe in Sedezimalform

Ganze Zahlen können auch in Sedezimalform angegeben werden. Sie werden durch das vorangestellte Zeichen "#", die Ziffern 0 bis 9 und die Buchstaben a bis f (in Groß- oder in Kleinschreibung) gebildet. Da für Integer-Zahlen maximal 32 Bit (4 Byte) zur Verfügung stehen, kann eine Zahl in sedezimaler Schreibweise maximal 8 Sedezimalzeichen umfassen. Eine Sedezimalzahl mit 8 Stellen und mit gesetztem vorderen Bit (= Vorzeichenbit) gilt als negative Integer-Zahl, alle anderen als nichtnegative Integer-Zahlen.

*Beispiele für Integer-Zahlen in Sedezimalform*

```
#40            hat denselben Wert wie 64
 FF             hat denselben Wert wie 255
 100A           hat denselben Wert wie 4106
#FFFFFFFF      hat denselben Wert wie -1
#7FFFFFFF      hat denselben Wert wie Long_Maxint
#80000000      hat denselben Wert wie Long_Minint
```

- **Zahlen eines Real-Typs**

Eine vorzeichenlose Real-Zahl steht für einen Wert eines Real-Typs. Bei Verwendung einer vorzeichenlosen Real-Zahl im Programm paßt sich deren Typ dem Kontext an, d. h. ob sie als Wert des Typs Short_Real oder Long_Real verwendet wird (siehe 9). Der Buchstabe "e" bzw. "E" in einer Zahl, gefolgt von einer ganzen Zahl (Exponent) bedeutet "mal 10 hoch" ("e" und "E" werden nicht unterschieden). Die Grenzen der darstellbaren Zahlenbereiche sind durch vordefinierte Konstanten-Bezeichner (siehe 5.2) festgelegt.

Eine Zahl eines Real-Typs kann sein:
– eine ganze Zahl, gefolgt von einem Exponententeil, z. B. 2E10
– eine ganze Zahl, gefolgt von einem Dezimalteil, z. B. 3.14
– eine ganze Zahl, gefolgt von einem Dezimalteil und einem Exponententeil, z. B. 3.14E-10

*Beispiele für vorzeichenlose Real-Zahlen*

```
0.1
3.14159
5e-3           bedeutet  5 • 10^-3
67.32E+18      bedeutet  67.32 • 10^18
2e9            bedeutet  2 • 10^9
```

*Querverweise*

| | |
|---|---|
| Integer-Typen: | 6.2.1 |
| Real-Typen: | 6.2.2 |
| Vordefinierte Konstanten: | 5.2 |

Ausdrücke : 9
Ein-/Ausgabe: 19

# Marken

```
Marke        = Ziffernfolge .

Ziffernfolge = Ziffer {Ziffer}.

Ziffer       = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".
```

Marken sind Ziffernfolgen, die wie ganze Zahlen zu interpretieren sind. Ihr Wert muß im Bereich 0 bis 9999 liegen. Verschiedene Zifferfolgen, die denselben Wert haben, repräsentieren die gleiche Marke (siehe auch Kap. 4).

Marken dienen dazu, Anweisungen zu markieren, zu denen dann mittels GOTO-Anweisungen gesprungen werden kann.

*Beispiele für Marken*

```
0
13
4711
09999
```

*Querverweise*

| | |
|---|---|
| Markendeklaration: | 4 |
| Goto-Anweisung: | 10.1.4 |
| Sichtbarkeitsregeln: | 12 |

# Zeichenketten

```
Zeichenkette    = "'"  {Zeichenkettenelement } "'"
                | "#'" {Sedezimal-Ziffernpaar} "'".

Zeichenkettenelement
                = Apostrophdarstellung | Zeichen.

Apostrophdarstellung
                = "''".

Sedezimal-Ziffernpaar
                = Sedezimal-Ziffer Sedezimal-Ziffer.

Sedezimal-Ziffer
                = Ziffer|"a"|"b"|"c"|"d"|"e"|"f".
```

Zeichenketten sind Folgen von Elementen, eingeschlossen in Apostrophe. Jedes Element repräsentiert einen Wert des vordefinierten Typs Char (siehe 6.2.3). Groß- und Kleinbuchstaben werden unterschieden. Zeichenketten, die mehr als 1 Zeichenkettenelement enthalten, stehen für einen Wert eines Zeichenkettentyps mit so vielen Komponenten, wie die Zeichenkette Elemente enthält. Zeichenketten mit genau einem Element stehen für einen Wert des vordefinierten Typs Char (siehe 6.3.1.3). Zeichenketten dürfen sich, wie andere lexikalische Elemente auch, nicht über das Ende einer Zeile hinaus erstrecken.

### Darstellung von Apostrophen in Zeichenketten

Soll eine Zeichenkette ein oder mehrere Apostrophe enthalten, so muß jedes Apostroph durch zwei aufeinanderfolgende Apostrophe dargestellt werden, die dann jeweils als ein Zeichenelement zählen.

### Leere Zeichenkette

Eine Zeichenkette, die kein Element enthält, heißt "leere Zeichenkette". Sie wird durch zwei aufeinanderfolgende Apostrophe dargestellt. In Norm-Pascal sind leere Zeichenketten nicht erlaubt.

*Beispiel*

```
kodiert:       gedruckt:
'PASCAL'       PASCAL
''''           '
'Don''t'       Don't
''                       {leere Zeichenkette}
```

### Angabe in Sedezimalform

Zeichenketten können auch in Sedezimalform dargestellt werden. Dazu wird der Zeichenkette das Zeichen "#" vorangestellt. Zwischen den Hochkommata muß dann immer eine gerade Anzahl von Sedezimal-Ziffern stehen.

*Beispiele (in EBCDI-Code)*

```
#'C1C6C6C5'          hat die selbe Bedeutung wie 'AFFE'
#'D781A2838193'      hat die selbe Bedeutung wie 'Pascal'
```

So können auch nichtabdruckbare Zeichen in Zeichenketten kodiert werden.

Die Zeichenkette

#'0D0A'

beinhaltet die beiden ASCII-Zeichen für die Funktionen "Wagenrücklauf" und "Zeilenvorschub", mit denen beispielsweise bei einem angeschlossenen Drucker mit ASCII-Dekodierung die entsprechenden Bewegungen von Walze und Druckkopf ausgelöst werden.

*Querverweise*

Typ Char: 6.2.3
Zeichenkettentypen: 6.3.2

# Trennung lexikalischer Einheiten und Kommentare

Lexikalische Einheiten in Pascal-Programmen werden durch

- Kommentare
- Leerzeichen (Leerzeichen außerhalb von Zeichenketten)
- Zeilenenden

voneinander getrennt. Diese nennt man Trenner.

Beliebig viele solcher Trenner können zwischen je zwei aufeinanderfolgenden lexikalischen Elementen bzw. vor dem ersten lexikalischen Element eines Programms auftreten. Mindestens ein Trenner muß jedes Paar lexikalischer Elemente trennen, das Bezeichner, Wortsymbole, vorzeichenlose Zahlen oder Marken enthält. Innerhalb eines lexikalischen Elements darf kein Trenner vorkommen (Leerzeichen in Kommentaren und Zeichenketten zählen nicht als Trenner).

Auch die Sonderzeichen haben in den meisten Fällen eine separierende Funktion, jedoch wird eine bessere Übersichtlichkeit des Programms durch extensive Anwendung von Trennern erreicht.

### Kommentare

Eine beliebige Folge von Zeichen, eingeschlossen in geschweifte Klammern (oder die Ersatzdarstellung), gilt als Kommentar.

Ein Kommentar, der unmittelbar nach der öffnenden Kommentarklammer mit einem Dollarzeichen beginnt, heißt Pseudokommentar und enthält Steueranweisungen an den Compiler (siehe Kapitel 16).

*Regeln*

- In der Zeichenfolge dürfen auch Zeilenenden vorkommen, d. h. ein Kommentar kann sich über mehrere Zeilen erstrecken.
- In der Zeichenfolge darf keine schließende geschweifte Klammer (oder ihre Ersatzdarstellung) vorkommen, insbesondere sind Kommentare innerhalb von Kommentaren nicht zulässig.
- Ersatzdarstellungen für geschweifte Klammern:

```
"{" = "(*"

"}" = "*)"
```

*Beispiel*

```
{Dies ist ein Kommentar}

(*Dies ist auch ein Kommentar*)

{Klammern können auch gemischt verwendet werden*)

{Kommentare können über
  viele Zeilen
  hinweg führen}

'{Dies ist kein Kommentar} sondern eine Zeichenkette'
```

*Querverweise*

Sonderzeichen: 3.2
Wortsymbol: 3.2
Zahlen: 3.5
Marken: 3.6, 4
Steueranweisungen: 16

# Markendeklarationen

Ein Markendeklarationsteil hat folgende Syntax:

```
Markendeklarationsteil = "LABEL" Marke { "," Marke } ";" .
Marke                  = Ziffernfolge .
```

Marken sind Ziffernfolgen. Sie unterscheiden sich voneinander durch ihren ganzzahligen Wert, der im Intervall 0..9999 liegen muß. Mit Marken werden Anweisungen markiert, zu denen dann mit der GOTO-Anweisung (siehe 10.1.4) gesprungen werden kann. Durch die Deklaration von Marken wird ihre Verwendung angekündigt. Eine in einem Block deklarierte Marke muß im Anweisungsteil dieses Blocks (Hauptprogramm oder Unterprogramm) auch genau einmal zur Markierung einer Anweisung verwendet werden. Ferner darf sie in beliebig vielen GOTO-Anweisungen in diesem Block oder in eingeschachtelten Blöcken vorkommen.

Die Sichtbarkeitsregeln für Marken sind in Kapitel 12 beschrieben.

*Beispiel*

```
LABEL 1, 13, 9999;
```

*Hinweise*

- `0000013, 0013, 013 und 13 sind gleiche Marken.`
- Die Verwendung von Marken und GOTO-Anweisungen ist zu vermeiden, da bei zu intensiver Benutzung "Spaghetti-Programme" entstehen können, bei denen der Kontrollfluß unübersichtlich ist. Die Verwendung von IF-, CASE, FOR, WHILE und REPEAT-Anweisungen (siehe 10) sowie von Unterprogrammen (siehe 8) erlaubt es übersichtliche Programme zu erstellen. Außerdem erschweren GOTO-Anweisungen automatische Codeoptimierungen durch den Compiler.

*Querverweise*

| | |
|---|---|
| Marken: | 3.6, 4 |
| Block: | 8.1, 8.2, 11.1, 12.1 |
| GOTO-Anweisung: | 10.1.4 |
| Sichtbarkeitsregeln: | 12 |

# Konstanten

## Konstantendefinitionen

Eine Konstantendefinition führt einen Bezeichner ein, der für einen festen Wert steht. Auf diesen Wert kann dann durch Angabe des eingeführten Konstanten-Bezeichners zugegriffen werden. Ein Konstantendefinitionsteil hat folgende Syntax:

```
Konstantendefinitionsteil
                = "CONST" Konstantendefinition { Konstantendefinition } .

Konstantendefinition
                = Bezeichner "=" Konstante ";" .
Konstante       = statischer_Ausdruck .

Konstanten-Name = [ Paket-Bezeichner "." ] Konstanten-Bezeichner.
```

In Norm-Pascal sind als Konstante nur Zahlen mit oder ohne Vorzeichen, Konstanten-Bezeichner und Zeichenketten zugelassen. Ein Vorzeichen vor einem Konstanten-Bezeichner ist nur dann erlaubt, wenn dieser Bezeichner für eine Integer-Zahl oder eine Real-Zahl steht.

In Pascal-XT kann statt einer Konstanten ein statischer Ausdruck stehen, also ein Ausdruck, der bereits zur Übersetzungszeit berechnet werden kann. Die Auswertung des Ausdrucks darf zu keinem zur Übersetzungszeit erkannten Fehler (siehe 2.3) führen. In Abschnitt 9.2 wird beschrieben, wann ein Ausdruck statisch ist.

Der statische Ausdruck auf der rechten Seite der Definition darf den Bezeichner der linken Seite nicht enthalten.

Ist der statische Ausdruck auf der rechten Seite ein (nicht qualifizierter) Mengenbildner, dann wird die definierte Konstante als nicht gepackt betrachtet, es sei denn der Kontext erfordert dies.

Jede Verwendung des Bezeichners wird als Konstanten-Name bezeichnet und steht für den Wert, mit dem er in der Konstantendefinition verknüpft wurde.
In Pascal-XT kann ein Konstanten-Name auch durch Voranstellung eines Paket-Bezeichners gebildet werden und bezieht sich dann auf eine Konstantendefini-

tion innerhalb der Spezifikation des bezeichneten Pakets.

Die Sichtbarkeitsregeln für Konstanten-Bezeichner sind in Kapitel 12 beschrieben.

*Hinweise*

- Selten verwendete Konstanten können auch im Programmtext direkt als Zahlen, Zeichenketten oder statische Ausdrücke angegeben werden, ohne vorher in einer Konstantendefinition mit einem Bezeichner verknüpft worden zu sein. Ein typisches Beispiel ist die Ausgabe von Zeichenketten in Dialoganwendungen. Der Nachteil dieser Vorgehensweise bei mehrfach verwendeten Konstanten liegt klar auf der Hand: Sobald einmal der Wert einer Konstanten geändert werden soll, müssen sämtliche im Programm verstreuten Literale dazu gefunden werden. Eine Konstantendefinition reduziert diesen Aufwand auf eine Änderung der Definition.
- In Pascal-XT darf NIL auch als Konstante auf der "rechten" Seite einer Konstantendefinition verwendet werden.

*Beispiele*

```
CONST
   drei        = 3;
   e           = 2.718282;
   minimum     = - drei;
   on          = True;
   dot         = '.';
   titel       = 'PASCAL';
                                    { Angabe in Sedezimalform }
   max_byte    =  #FF;
   affe        =  #'C1C6C6C5';
                                    { Verwendung von Nil }
   end_of_list= NIL;
                                    { arithmetische Ausdruecke }
   konst1      = 1/2 * e;
   numbits     = 13;
   maxval      = 2 ** numbits - 1;
                                    { boolescher Ausdruck }
   test        = numbits > 12;
                                    { Mengenausdruecke }
   digits      = ['0' .. '9'];
   hexletters = ['A', 'B', 'C', 'D', 'E', 'F']
   hexdigits   = digits + hexletters;
                                    { Zeichenkettenausdruck }
   version     = '2.0';
   header      = Concat (titel, version, '88/05/01');

                                    { Verwendung einer Konstanten pi }
   pi2         = numeric.pi * 2.0;  { aus einem Paket numeric        }

TYPE                                { Array-Aggregat }
   hextab      = array [0..15] of char;
CONST
   hexchar     = hextab ('0','1','2','3','4','5','6','7','8','9',
                         'A','B','C','D','E','F');
```

*Querverweise*

| | |
|---|---|
| Integer-Zahl: | 3.5 |
| Real-Zahl: | 3.5 |
| Zeichenkette: | 3.7 |
| Sedezimalform: | 3.5, 3.7 |
| Einfacher Typ: | 6.2 |
| Generischer Zeigertyp: | 6.5.2 |
| Statischer Ausdruck: | 9.2 |
| Mengenbildner: | 9.4 |
| Aggregat: | 9.5 |
| Sichtbarkeitsregel: | 12 |

# Vordefinierte Konstanten

In Pascal-XT sind die in den Tabellen 5-1 bis 5-3 aufgeführten Konstanten-Bezeichner vordefiniert.

| Bezeichner | Datentyp | Wert |
|---|---|---|
| False | Boolean | Ord(False) = 0 |
| True | Boolean | Ord(True) = 1 |

Tabelle 5-1: Vordefinierte Konstanten-Bezeichner vom Typ Boolean

| Bezeichner | Datentyp | Wert |
|---|---|---|
| Long_Maxint | Long_Integer | 2147483647 = $2^{31}-1$ |
| Long_Minint | Long_Integer | -2147483648 = $-2^{31}$ |
| Short_Maxint | Short_Integer | 32767 = $2^{15}-1$ |
| Short_Minint | Short_Integer | -32768 = $-2^{15}$ |
| Maxint | Integer | (1) |
| Minint | Integer | (2) |
| Long_Minreal | Long_Real | (3) |
| Long_Maxreal | Long_Real | (4) |
| Short_Minreal | Short_Real | (5) |
| Short_Maxreal | Short_Real | (6) |
| Minreal | Real | (7) |
| Maxreal | Real | (8) |

Tabelle 5-2: Vordefinierte numerische Konstanten-Bezeichner

### Implementierungsdefinierte Eigenschaften

(1) Maxint ist implementierungsdefiniert gleich Short_Maxint oder Long_Maxint

(2) Minint ist implementierungsdefiniert gleich Short_Minint oder Long_Minint

(3) Long_Minreal ist der implementierungsdefinierte kleinste positive Wert des Typs Long_Real

(4) Long_Maxreal ist der implementierungsdefinierte größte positive Wert des Typs Long_Real

(5) Short_Minreal ist der implementierungsdefinierte kleinste positive Wert des Typs Short_Real

(6) Short_Maxreal ist der implementierungsdefinierte größte positive Wert des Typs Short_Real

(7) Minreal ist implementierungsdefiniert gleich Short_Minreal oder Long_Minreal

(8) Maxreal ist implementierungsdefiniert gleich Short_Maxreal oder Long_Maxreal

*Hinweis*

Aussagen zur Genauigkeit von Realzahlen sind in 6.2.2 zu finden.

Im Zusammenhang mit der programmierten Ausnahmebehandlung (siehe 14) sind folgende Fehlernummern als negative Integer-Konstanten reserviert:

| Bezeichner | Wert | Kurzbeschreibung |
|---|---|---|
| Numeric_Error | -02 | arithmetischer Überlauf |
| Range_Error | -03 | Wert außerhalb des Teilbereichs |
| Set_Error | -04 | Wert außerhalb des Mengentyps |
| String_Error | -05 | Maximallänge eines Strings überschritten |
| Index_Error | -06 | Wert außerhalb des Index-Teilbereichs |
| Pointer_Error | -07 | Dereferenzierung von NIL |
| Variant_Error | -08 | Zugriff auf eine nicht eingestellte Variante |
| Case_Error | -09 | keine CASE-Alternative gefunden |
| File_Error | -10 | Datei in falschem Modus |
| Eof_Error | -11 | Versuch, über Dateiende zu lesen |
| Open_Error | -12 | Fehler beim Eröffnen einer Datei |
| Read_Error | -13 | Syntax-Fehler beim Lesen einer Zahl |
| Memory_Error | -14 | Speicherüberlauf |
| Break_Error | -15 | Unterbrechung |
| Elab_Error | -16 | zyklische Initialisierungsabhängigkeit |
| System_Error | -01 | sonstige Systemfehler |

Tabelle 5-3: Vordefinierte Konstanten-Bezeichner für die Ausnahmebehandlung

*Querverweise*

Integer-Zahlen: 3.5
Real-Zahlen: 3.5
Integer-Typen: 6.2.1
Real-Typen: 6.2.2
Boolean: 6.2.4
Ausnahmebehandlung: 14
Ord-Funktion: 15

# Typen

Mit den Datentypen begegnet uns eine grundlegende Eigenschaft der Programmiersprache Pascal. Jeder Wert und jede Variable besitzt einen Typ. Der Typ definiert folgende Eigenschaften:

- Die Menge der zulässigen Werte (Wertebereich), die ein Objekt annehmen kann.
- Die Menge der auf den Objekten anwendbaren Operationen.

Typen werden in drei Klassen eingeteilt:

| Einteilung | zugehörige Typen | Wertebereiche / Beispiele |
|---|---|---|
| Einfache Typen | Integer<br>Short_Integer<br>Long_Integer<br>Char<br>Boolean<br>Aufzählungstypen<br>Teilbereichstypen<br><br>Real<br>Short_Real<br>Long_Real | Minint .. Maxint<br>Short_Minint .. Short_Maxint<br>Long_Minint .. Long_Maxint<br>Zeichen des Zeichensatzes<br>(False, True)<br>z. B. (open, closed, locked)<br>z. B. 0..255 oder '0'..'9' |
| Strukturierte Typen | ARRAY [...] OF ...<br>RECORD ... END<br>SET OF ...<br>FILE OF ...<br>Text<br>String [...] | Reihungen<br>Verbunde<br>Mengen<br>Nicht-Text-Dateien<br>Textdateien<br>Zeichenketten variabler Länge |
| Zeigertypen | ↑ ... | z. B. ↑node |
| Generische Typen | Pointer<br>Any_File<br>Any_Type | generischer Zeigertyp<br>generischer FILE-Typ<br>generischer Typ |

Tabelle 6-1: Einteilung von Typen

Einfache (oder skalare) Typen können nicht weiter unterteilt werden, d. h. sie enthalten keine Komponenten. Bis auf die Aufzählungs- und Teilbereichstypen sind diese Typen über vordefinierte Typ-Bezeichner ansprechbar. Die einfachen Typen, mit Ausnahme der Real-Typen, werden als Ordinaltypen bezeichnet. Ein strukturierter Typ wird durch die Typen seiner Komponenten und die Art der Strukturierung beschrieben. Die Komponenten des Typs können einfache Typen, Zeigertypen oder wiederum strukturierte Typen besitzen. Die Strukturierung wird z. B. durch die Schlüsselwörter ARRAY, RECORD etc. festgelegt. Mit Zeigertypen können dynamische Variable auf der Halde (heap) erzeugt, bearbeitet und vernichtet werden. Die Halde ist ein dynamisch wachsender und schrumpfender Speicherbereich. Zeigertypen werden bei der Verarbeitung von Graphen (z. B. Bäume oder Listen) benötigt.

Das Typen-Konzept hat folgende Auswirkungen:

- Im Vergleich zu anderen Programmiersprachen hat die Definition der Datenstrukturen bei Pascal einen wesentlich höheren Anteil am Problemlösungsansatz.
- Der Mehraufwand während der Definitionsphase zahlt sich während der Entwicklungsphase durch mehr Transparenz und einfachere Schnittstellen-Definitionen bei mehrköpfigen Entwicklungsteams aus.
- Das Typ-Konzept führt zur rascheren Fehlererkennung, da viele Fehler bereits zur Übersetzungszeit erkannt werden, die sich bei anderen Sprachen erst zur Laufzeit auswirken.
- Die Festlegung der Datenzugriffs-Methoden bereits zur Übersetzungszeit führt in der Regel zur Erzeugung von sehr effizientem Programmcode.

# Typdefinitionen

Eine Typdefinition führt einen neuen Bezeichner ein, der für einen Typ steht. Dieser neue Bezeichner wird als Typ-Bezeichner bezeichnet. Der Typ kann ein neuer Typ oder ein bereits definierter Typ-Bezeichner sein. Ein Typdefinitionsteil hat folgende Syntax:

```
Typdefinitionsteil
                = "TYPE" Typdefinition { Typdefinition } .

Typdefinition   = Bezeichner "=" Typangabe ";" .

Typangabe       = Typ-Name | neuer_Typ .

neuer_Typ       = Aufzählungstyp | Teilbereichstyp | Stringtyp |
                  Zeigertyp | strukturierter_Typ .

Aufzählungstyp  = "(" Bezeichnerliste ")" .

Bezeichnerliste = Bezeichner { "," Bezeichner } .

Teilbereichstyp = Ordinal-Konstante ".." Ordinal-Konstante .

Stringtyp       = String-Bezeichner "[" Stringlaenge "]" .

Stringlaenge    = Integer-Konstante .

strukturierter_Typ
                = [ "PACKED" ] ungepackter_strukturierter_Typ .

ungepackter_strukturierter_Typ
                = ARRAY-Typ | RECORD-Typ | SET-Typ | FILE-Typ .

Typ-Name        = [ Paket-Bezeichner "." ] Typ-Bezeichner .
```

Eine Typdefinition führt einen Bezeichner ein, der für den Typ steht, der in der Typangabe beschrieben ist.

Jede Verwendung dieses Bezeichners wird als Typ-Name bezeichnet und steht für den Typ, der durch die Typangabe bestimmt ist.
In Pascal-XT kann ein Typ-Name auch durch Voranstellung eines Paket-Bezeichners gebildet werden. Dann bezieht er sich auf eine Typdefinition in der Spezifikation des betreffenden Pakets.

In der Typangabe kann auf einen bereits definierten Typ durch Angabe seines Typ-Namens Bezug genommen werden. Dabei darf die Typangabe jedoch keine Anwendung des Typ-Bezeichners der linken Seite der Typdefinition enthalten, außer er wird als Domänentyp für einen Zeigertyp verwendet (siehe 6.4). Jeder neue Typ unterscheidet sich von jedem anderen neuen Typ. Ein neuer Typ wird explizit angegeben, d. h. er wird nicht über einen bereits definierten Typ-Bezeichner angesprochen.

Die optionale Angabe einer Konstanten ist nur unmittelbar hinter dem vordefinierten Typ-Bezeichner "String" erlaubt und definiert dessen maximale Länge (siehe 6.3.2.2).
Die Sichtbarkeitsregeln für Typ-Bezeichner sind in Kapitel 12 beschrieben.

*Hinweis*

Die Definition komplexerer Typen wird dadurch erleichtert, daß ein Typ-Name überall dort benutzt werden kann, wo die Angabe eines Typs erwartet wird. Dies verbessert die Lesbarkeit von Programmen, erleichtert ihre Änderung und gestattet es, einen einmal definierten Typ an vielen Stellen im Programm zu verwenden.

*Beispiel für einen Typdefinitionsteil*

```
TYPE
   posint     = 0..Maxint;
   zahl       = Integer;
   bereich    = 1..150;
   farbe      = (rot, gruen, gelb, blau);
   loch_karte = PACKED ARRAY[1..80] OF Char;
   zeile      = String [80];
   person     = ↑personinfo;
   personinfo = RECORD
      name, vorname    : zeile;
      alter            : zahl;
      CASE verheiratet : Boolean OF
         True  : (partner : person);
         False : ();
      END;
   datei      = FILE OF loch_karte;
```

*Querverweise*

| | |
|---|---|
| Einfacher Typ: | 6.2 |
| Strukturierter Typ: | 6.3 |
| String-Typ: | 6.3.2.2 |
| Zeigertyp: | 6.4 |
| Domänentyp: | 6.4 |
| Verträglichkeit: | 6.6 |
| Identität von Typen: | 6.6.1 |
| Sichtbarkeitsregeln: | 12 |

# Einfache Typen

Ein einfacher Typ definiert eine geordnete Wertemenge. Wenn sich diesen Werten eindeutig ganzen Ordnungszahlen zuordnen lassen, spricht man von einem Ordinaltyp. Die Real-Typen gehören demzufolge nicht zu den Ordinaltypen.

In folgenden Fällen sind nur Ordinaltypen zulässig:
- als Wirtstyp eines Teilbereichstyp
- als Indextyp eines ARRAY-Typ
- als Basistyp eines SET-Typ
- als Kennungstyp in einem RECORD-Typ
- als Typ der Laufvariablen einer FOR-Anweisung
- als Typ des Fallindex einer CASE-Anweisung
- als Typ des Parameters der vordefinierten Funktionen Chr, Ord, Pred oder Succ

Die Typen Integer, Long_Integer, Short_Integer, Real, Long_Real, Short_Real, Char und Boolean sind vordefiniert, d. h., ihre Namen sind vordefinierte Typ-Bezeichner.

Integer, Long_Integer, Short_Integer, Char, Boolean, Aufzählungstypen und Teilbereichstypen sind Ordinaltypen.

*Querverweise*


Teilbereichstyp: 6.2.6
ARRAY-Typ: 6.3.1
RECORD-Typ: 6.3.3
SET-Typ: 6.3.4
Kennungstyp: 6.3.3.1
CASE-Anweisung: 10.3.2
FOR-Anweisung: 10.4.3
Ordinalfunktionen: 15.6


### Integer-Typen

Die Integer-Typen sind Ordinaltypen und repräsentieren Teilmengen der ganzen Zahlen. Pascal-XT unterscheidet die beiden Typen Long_Integer und Short_Integer, deren Wertebereiche durch die vordefinierten Konstanten (siehe 5.2) definiert sind:

```
Short_Integer = Short_Minint .. Short_Maxint;
Long_Integer  = Long_Minint  .. Long_Maxint;
Integer       = Minint .. Maxint;
```

Norm-Pascal kennt nur einen Integer-Typ: Integer. Außerdem garantiert Norm-Pascal nur, daß die Werte des Bereichs -Maxint .. Maxint zum Wertebereich des Typs Integer gehören. Der durch den vordefinierten Konstanten-Bezeichner Minint repräsentierte Wert gehört nicht mehr notwendig zum Typ Integer.

#### Implementierungsdefinierte Eigenschaft

> Der vordefinierte Typbezeichner Integer ist implementierungsdefiniert entweder gleich dem Typ Short_Integer oder gleich dem Typ Long_Integer

Auf Werte eines Integer-Typs können arithmetische Operatoren und Vergleichsoperatoren angewandt werden. Sie können zugewiesen, von Textdateien und Strings eingelesen und auf diese ausgegeben werden. Ferner gibt es zahlreiche vordefinierte Unterprogramme, deren Parameter bzw. Funktionsergebnisse Integer-Typen besitzen.

*Querverweise*

Zahlen: 3.5
Vordefinierte Konstanten: 5.2
Ordinaltyp: 6.2
Arithmetische Operatoren: 9.3.1
Vergleichsoperatoren: 9.3.4
Zuweisung: 10.1.2
vordefinierte Unterprogramme: 15
Ein- / Ausgabe: 19

### Real-Typen

Die Werte der Typen Long_Real, Short_Real und Real bilden eine Untermenge der reellen Zahlen.
Es existieren die vordefinierten Konstanten-Bezeichner Minreal, Maxreal, Short_Minreal, Short_Maxreal, Long_Minreal und Long_Maxreal für die kleinste, bzw. größte darstellbare positive reelle Zahl des jeweiligen Real-Typs.

Auf Werte eines Real-Typs können arithmetische Operatoren und Vergleichsoperatoren angewandt werden. Sie können zugewiesen, von Textdateien und Strings eingelesen und auf diese ausgegeben werden. Ferner gibt es zahlreiche vordefinierte Unterprogramme, deren Parameter bzw. Funtionsergebnisse Real-Typen besitzen.

**Implementierungsdefinierte Eigenschaften**

– Die Werte der Typen Short_Real und Long_Real stellen implementierungsdefinierte Teilmengen der reellen Zahlen dar.

– Der vordefinierte Typbezeichner Real ist implementierungsdefiniert gleich dem Typ Short_Real oder Long_Real.

– Die Ergebnisse von arithmetischen Real-Operatoren und -Funktionen sind Näherungswerte der mathematischen Ergebnisse. Die Genauigkeit dieser Näherungen sind implementierungsdefiniert.

*Hinweis*

Werte eines Real-Typs können auf einem Rechner i. a. nicht exakt dargestellt werden. Bei arithmetischen Operationen muß der Anwender durch geeignete Algorithmen die Fortpflanzung von Rundungsfehlern in Grenzen halten. Auf Vergleiche von Real-Zahlen auf Gleichheit sollte wegen der Ungenauigkeit verzichtet werden.

*Querverweise*

Vordefinierte Konstanten: 5.2
arithmetische Operatoren: 9.3.1
Vergleichsoperator: 9.3.4
Zuweisung: 10.1.2
vordefinierte Unterprogramme: 15
Ein- / Ausgabe: 19

### Der Typ Char

Die Werte des Ordinaltyps Char ergeben sich durch Aufzählung der Zeichen des Zeichensatzes, der für die jeweilige Implementierung definiert ist (ASCII, EBCDIC, ISO-7bit etc.). Bei den meisten dieser Werte handelt es sich um darstellbare (abdruckbare) Zeichen. Einige Werte dienen jedoch der Steuerung von peripheren Geräten oder der Realisierung von Datenaustausch-Protokollen mit anderen Geräten.

Den Zeichenwerten sind, bei 0 beginnend und lückenlos aufsteigend, Ordinalzahlen vom Typ Integer zugeordnet. Die Zuordnung ist implementierungsdefiniert (siehe unten). In jeder Implementierung gelten jedoch folgende Beziehungen:

– Die Ziffern ’0’ bis ’9’ sind numerisch aufsteigend geordnet und lückenlos.
– Die Großbuchstaben ’A’ bis ’Z’ sind alphabetisch geordnet, aber nicht notwendig lückenlos.
– Die Kleinbuchstaben ’a’ bis ’z’ sind alphabetisch geordnet, aber nicht notwendig lükkenlos.
– Die Ordnungsrelation zwischen 2 beliebigen Zeichenwerten ist dieselbe wie die zwischen den entsprechenden Ordinalzahlen.

#### Implementierungsdefinierte Eigenschaften

– Der Wertebereich des Typs Char ist implementierungsdefiniert.
– Die Zuordnung von Ordinalzahlen vom Typ Integer zu den Werten des Typs Char ist implementierungsdefiniert.

Auf Werte des Typs Char können Vergleichsoperatoren angewandt werden. Sie können zugewiesen, von Textdateien und Strings eingelesen und auf diese ausgegeben werden. Ferner gibt es zahlreiche vordefinierte Unterprogramme, deren Parameter bzw. Funtionsergebnisse der Typ Char ist.
In Pascal-XT ist der Typ Char auch ein Zeichenkettentyp (siehe 6.3.2).

*Hinweise*

- Die Darstellung abdruckbarer Zeichen kann für verschiedene Terminal- und Druckertypen unterschiedlich sein.
- Es widerspricht der Norm, wenn ein Pascal-Programm Ordnungsrelationen voraussetzt, die zwar von einem bestimmten Zeichensatz erfüllt werden, aber nicht in der oben aufgeführten Normanforderung enthalten sind. Z. B. ist ein Sortier-Algorithmus, der die lückenlos aufsteigend Folge der Buchstaben "A" bis "Z" oder ' ' < 'A' voraussetzt, nicht normgerecht.
- Im EBCDIC-Zeichensatz gibt es Lücken zwischen den Buchstaben. Das führt bei den vordefinierten Funktionen Succ und Pred dazu, daß sie in einigen Fällen nicht das erwartete Zeichen liefern.

*Querverweise*


Ordinaltyp: 6.2
Zeichenketten: 6.3.2
Vergleichsoperatoren: 9.3.4
Zuweisung: 10.1.2
vordefinierte Unterprogramme: 15
Ein- / Ausgabe: 19


## Der Typ Boolean

Die Werte des Ordinaltyps Boolean ergeben sich aus der Aufzählung der Wahrheitswerte, die durch die vordefinierten Konstanten-Bezeichner True und False repräsentiert werden:

Boolean = (False, True)

Die vordefinierten Funktionen Ord, Pred und Succ (siehe 15.6) liefern folgende Werte:

```
Ord (False)  = 0
Ord (True)   = 1
Pred (True)  = False
Succ (False) = True.
```

Auf Werte des Typ Boolean können boolesche Operatoren und Vergleichsoperatoren angewandt werden. Sie können zugewiesen und auf Textdateien und Strings ausgegeben werden. Die Ergebnisse aller Vergleichsoperatoren und einiger vordefinierter Funktionen sind vom Typ Boolean. Ausdrücke vom Typ Boolean dienen in IF-, WHILE und REPEAT-Anweisungen zur Steuerung des Programmflusses.

*Querverweise*

| | |
|---|---|
| Vordefinierte Konstanten: | 5.2 |
| Ordinaltyp: | 6.2 |
| Ausdrücke: | 9 |
| Boolesche Operatoren: | 9.3.2 |
| Vergleichsoperatoren: | 9.3.4 |
| Zuweisung: | 10.1.2 |
| IF-Anweisung: | 10.3.1 |
| REPEAT-Anweisung: | 10.4.1 |
| WHILE-Anweisung: | 10.4.2 |
| Sichtbarkeitsregeln: | 12 |
| vordefinierte Funktionen: | 15 |
| Ein- / Ausgabe: | 19 |

## Aufzählungstypen

Aufzählungstypen sind Ordinaltypen und haben folgende Syntax:

```
Aufzählungstyp   = "(" Bezeichnerliste ")" .

Bezeichnerliste  = Bezeichner { "," Bezeichner } .
```

Ein Aufzählungstyp bestimmt eine geordnete Menge von Werten durch Aufzählen der Bezeichner, die diese Werte repräsentieren sollen. Diese Bezeichner können dann als Konstanten-Bezeichner verwendet werden (siehe 12). Die Liste der Bezeichner wird in runde Klammern eingeschlossen. Die Ordnung dieser Werte wird durch die Reihenfolge bestimmt, in der ihre Bezeichner aufgezählt werden, d. h., wenn x vor y kommt, dann ist x kleiner als y.
Die Ordinalzahl jedes Wertes eines Aufzählungstyps ergibt sich aus der Abbildung aller Werte dieses Typs auf die aufeinanderfolgenden nichtnegativen Werte des Typs Integer, beginnend bei 0. Somit ist die Ordinalzahl des 1. Bezeichners 0, die des zweiten 1 usw.

Alle Bezeichner in einem Deklarationsteil müssen voneinander verschieden sein (siehe 12). Aus diesem Grund dürfen auch nicht verschiedene Aufzählungstypen die gleichen Bezeichner einführen. Die Aufzählungen (weiss, rot, blau) und (gelb, gruen, rot) dürfen daher nicht im gleichen Deklarationsteil vorkommen, da beide den Konstanten-Bezeichner "rot" einführen.

Da die Bezeichner keine Wortsymbole sein dürfen, ist z. B. die Aufzählung (mo, di, mi, do, fr) unzulässig, da "DO" ein Wortsymbol ist (siehe 3.2 und 3.3).

Die Sichtbarkeitsregeln für die in Aufzählungen definierten Bezeichner sind im Kapitel 12 beschrieben.

Auf Werte eines Aufzählungstyps sind die Vergleichsoperatoren und die vordefinierten Funktionen Pred, Succ und Ord anwendbar. Ferner können sie in Zuweisungen verwendet werden.

*Hinweis*

Aufzählungstypen ermöglichen ein sehr elegantes Arbeiten mit Informationen, die in den meisten anderen Programmiersprachen nur durch Bitcodierungen dargestellt werden können.

*Beispiele*

```
TYPE
   werktag   = (montag, dienstag, mittwoch, donnerstag, freitag);
   karte     = (treff, pique, coeur, karo);
   farbe     = (rot, blau, gelb, gruen, weiss, violett, orange);
   form      = (kreis, ellipse, viereck);
```

Es gilt damit beispielsweise:

```
Ord (rot)        = 0
Ord (gruen)      = 3
Ord (orange)     = 6
Succ (dienstag) = mittwoch
Pred (dienstag) = montag
donnerstag       > montag
```

*Querverweise*

| | |
|---|---|
| Bezeichner: | 3.3 |
| Vergleichsoperatoren: | 9.3.4 |
| Zuweisung: | 10.1.2 |
| Sichtbarkeitsregeln: | 12 |
| vordefinierte Funktionen: | 15 |

## Teilbereichstypen

Teilbereichstypen sind Ordinaltypen und haben folgende Syntax:

```
Teilbereichstyp  = Ordinal-Konstante ".." Ordinal-Konstante .
```

Ein Teilbereichstyp ist ein Teilbereich eines Ordinaltyps, der als Wirtstyp bezeichnet wird. Ein Wirtstyp kann vordefiniert sein (z. B. Integer) oder bezieht sich auf einen bereits zuvor definierten Aufzählungstyp. Ein Teilbereichstyp wird definiert durch die Angabe des kleinsten und des größten Wertes im Teilbereich, getrennt durch zwei Punkte. Die erste Konstante gibt den kleinsten Wert an. Dieser muß kleiner oder gleich dem größten Wert sein, der durch die 2. Konstante angegeben wird. Beide Konstanten müssen vom selben Ordinaltyp sein.

In Pascal-XT dürfen die beiden Konstanten auch zu verschiedenen Integer-Typen gehören (Long_Integer, Short_Integer). In diesem Fall ist der Wirtstyp der Typ Long_Integer.

Auf Werte eines Teilbereich-Typs sind dieselben Operationen wie auf Werte des zugehörigen Wirtstyps anwendbar.

*Hinweis*

Teilbereichstypen werden sinnvollerweise für solche Aufgaben eingesetzt, bei denen Werte in einem bestimmten Bereich liegen müssen. Einerseits wird dadurch die Verständlichkeit von Programmen erhöht, weil der Wertebereich von Variablen klar zum Ausdruck kommt. Andererseits wird bei eingeschalteter Check-Option (siehe Kap. 16) zur Laufzeit überprüft, ob eine Variable eines Teilbereichstyps auch nur zulässige Werte annimmt.

*Beispiele*

```
TYPE
                                { Wirtstyp Integer: }
   stunde     =  0 .. 24;
   range2     = -5 .. +4711;
   range3     =  Ord('0') .. Ord('9');
                                { Wirtstyp farbe, siehe 6.2.5: }
   grundfarbe =  rot .. gelb;
   mischfarbe =  gruen .. orange;
                                { Wirtstyp Char: }
   ziffer     = '0' .. '9';
```

*Querverweise*

Konstante: 5.1
Ordinaltypen: 6.2

Identität von Typen: 6.6.1

# Strukturierte Typen

Strukturierte Typen haben folgende Syntax:

```
strukturierter_Typ
              = [ "PACKED" ] ungepackter_strukturierter_Typ .

ungepackter_strukturierter_Typ
              = ARRAY-Typ | RECORD-Typ | SET-Typ | FILE-Typ .
```

Ein strukturierter Typ wird aus anderen Typen aufgebaut. Die Komponenten eines strukturierten Typs können einfache Typen, Zeigertypen oder wiederum strukturierte Typen besitzen. Ein strukturierter Typ wird durch die Typen seiner Komponenten und die Art seines Aufbaus charakterisiert.

**Gepackte strukturierte Typen**

Vor den in Typdefinitionen verwendeten Wortsymbolen ARRAY, RECORD, SET und FILE kann jeweils das Wortsymbol PACKED stehen. PACKED ist ein Hinweis an den Compiler, daß Werte dieses Typs rechnerintern platzsparend dargestellt werden sollen, auch wenn sich dadurch die Effizienz beim Arbeiten mit Variablen eines gepackten Typs oder ihren Komponenten verringert.

Die Festlegung eines strukturierten Typs als gepackt bezieht sich nur auf die Darstellung dieses Typs selbst, nicht aber auf die Komponenten des strukturierten Typs. Eine Ausnahme macht hier die Kurzschreibweise von mehrdimensionalen ARRAY-Typen (siehe 6.3.1).

String-Typen sind gepackte Typen.

Außer den möglichen Auswirkungen auf Effizienz und Speicherplatz sind noch folgende Regeln für gepackte und ungepackte Typen zu beachten:

- Werte eines gepackten SET-Typs können in Ausdrücken nicht mit Werten eines ungepackten SET-Typs verknüpft werden (9.3).
- Bei Variablenparameterübergabe (siehe 8.5.2) darf der Aktualparameter keine Komponente einer Variablen sein, deren Typ gepackt ist (Ausnahmen sind die Parameter vordefinierter Unterprogramme z. B. Read oder New).
- Bei der Konformität von Konformreihungen müssen der Aktualparameter und das Konformreihungsschema beide gepackt oder beide ungepackt sein (8.5.4).
- Bei den vordefinierten Prozeduren Pack und Unpack (15.8) muß jeweils ein ARRAY-Parameter gepackt, der andere ungepackt sein.

*Querverweise*

| | |
|---|---|
| einfacher Typ: | 6.2 |
| Zeigertyp: | 6.4 |
| Variablenparameter: | 8.5.2 |
| Konformreihungsschemata: | 8.5.4 |
| vordefinierte Prozeduren Pack und Unpack: | 15.8 |

## ARRAY-Typen

Ein ARRAY-Typ definiert eine Struktur, die aus einer festen Anzahl von Komponenten besteht, die alle denselben Typ besitzen. Ein ARRAY-Typ hat folgende Syntax:

```
ARRAY-TYP      = "ARRAY" "[" Indextyp { "," Indextyp } "]" "OF" Komponententyp.

Indextyp       = Ordinal-Typangabe .

Komponententyp = Typangabe .
```

Der Wert eines ARRAY-Typs ist eine Zuordnung von je einem Wert des Komponententyps zu jedem Wert des Indextyps. Der Indextyp muß ein Ordinaltyp sein, also Char, Boolean, ein Integer-Typ, Aufzählungen oder Teilbereiche. Der Komponententyp darf ein beliebiger Typ sein.

Werte eines ARRAY-Typs können als Ganzes zugewiesen werden. Auf die Komponenten kann durch Indizierung zugegriffen werden. Ferner sind die vordefinierten Prozeduren Pack und Unpack anwendbar. Auf Werte bestimmter ARRAY-Typen, nämlich Zeichenketten fester Länge sind weitere Operationen anwendbar (siehe 6.3.2.1).

*Hinweis*

Komponenten eines gepackten ARRAY können nicht als Variablenparameter an Unterprogramme übergeben werden.

*Beispiele*

In dem letzten Beispiel wird ein RECORD-Typ als Komponententyp verwendet (siehe 6.3.3).

```
TYPE
   farb_array = ARRAY [Boolean] OF (rot, gelb, gruen, blau);
   int_array  = ARRAY [Char] OF Integer;
   char_array = ARRAY [0..255] OF Char;
   real_array = ARRAY [-10..10] OF Real;
   rec_array  = ARRAY [1..10] OF RECORD k, g : Real; END;
```

**Abkürzende Schreibweise bei mehrdimensionalen ARRAY-Typen**

Der Komponenten-Typ eines ARRAY-Typs kann insbesondere wieder ein ARRAY-Typ sein. Für solche geschachtelten ARRAY-Typen ist eine abkürzende Schreibweise zulässig. Die folgenden Beispiele zeigen verschiedene Möglichkeiten, denselben Datentyp zu definieren.

```
ARRAY [Boolean] OF ARRAY [1..10] OF ARRAY [1..9] OF Real
ARRAY [Boolean] OF ARRAY [1..10,          1..9] OF Real
ARRAY [Boolean,          1..10] OF ARRAY [1..9] OF Real
ARRAY [Boolean,          1..10,           1..9] OF Real
```

Durch die abkürzende Schreibweise entstehen mehrdimensionale Reihungen. Das Attribut PACKED bezieht sich bei der abgekürzten Schreibweise auf alle abgekürzten Definitionen. Gleichwertig sind also:

```
PACKED ARRAY [1..10,                  1..8] OF Boolean
PACKED ARRAY [1..10] OF PACKED ARRAY [1..8] OF Boolean
```

Typische Beispiele für diese abgekürzten Definitionen sind Matrizen:

```
TYPE
     matrix = ARRAY [1..100, 1..100] OF Real;
```

*Querverweise*

Ordinaltypen: 6.2
PACKED: 6.3
Zeichenketten: 6.3.2
Indizierung: 9.6.2
Zuweisung: 10.1.2

## Zeichenkettentypen

Unter dem Begriff Zeichenkettentyp werden folgende Typen zusammengefaßt:

- Der vordefinierte Typ Char.
- PACKED ARRAY [1..n] OF Char mit 1 < n ≤ Short_Maxint.
- String und String [n] mit 1 < n ≤ Short_Maxint

In Norm-Pascal werden nur die Typen PACKED ARRAY [1..n] OF Char als Zeichenkettentyp bezeichnet, wobei n eine beliebige Integer-Konstante größer 1 ist.

In Pascal-XT wird ein PACKED ARRAY [1..n] OF Char nur dann als Zeichenkettentyp aufgefaßt, wenn n ≤ Short_Maxint ist. Dies ist zugleich der maximal zulässige Wert für n bei String [n].

Auf Werte eines Zeichenkettentyps sind die Vergleichsoperatoren anwendbar. Sie können ferner (als ganzes) zugewiesen und auf Textdateien und Strings ausgegeben werden. Auf ihren Komponenten kann indiziert zugegriffen werden.
Es gibt einige vordefinierte Unterprogramme, deren Parameter bzw. Funktionsergebnisse Zeichenkettentypen besitzen.

*Querverweise*


| | |
|---|---|
| Vergleichsoperatoren: | 9.3.4 |
| Typ Char: | 6.2.3 |
| Zuweisung: | 10.1.2 |
| Vordefinierte Unterprogramme: | 15 |
| Ein-/Ausgabe: | 19 |


### Zeichenkettentypen fester Länge

Ein Zeichenkettentyp mit einer festen Länge ist ein gepackter ARRAY-Typ PACKED ARRAY [1..n] OF Char mit einer Konstanten n, 1 < n ≤ Short_Maxint.

Jeder Wert eines solchen Zeichenkettentyps fester Länge muß genau n Zeichen enthalten.

*Hinweis*

Ein Zeichenkettentyp fester Länge hat sowohl die Eigenschaften eines ARRAY-Typs als auch die eines Zeichenkettentyps.

*Beispiel*

```
lochkarte = PACKED ARRAY [1..80] OF Char;
```

*Querverweise*

PACKED: 6.3
ARRAY-Typ: 6.3.1

Zeichenkettentypen variabler Länge (String-Typen)

Zeichenkettentypen variabler Länge, kurz auch String-Typen genannt, werden durch den vordefinierten Typ-Bezeichner String gemäß folgender Syntax definiert.

```
String-Typ   = "String" ["[" Konstante "]"].
```

Die Konstante hinter dem vordefinierten Typ-Bezeichner String bestimmt die Maximallänge des String-Typs. Der Wert der Konstanten muß vom Typ Short_Integer sein und im Bereich 1 .. Short Maxint liegen. Ein String-Typ wird als gepackter Typ aufgefaßt.

**Implementierungsdefinierte Eigenschaft**

Wird hinter dem Typ-Bezeichner String keine Konstante angegeben, dann wird eine implementierungsdefinierte Maximallänge angenommen.

Jeder Wert eines String-Typs ist eine Zeichenkette, deren aktuelle Länge (Anzahl der Zeichen) kleiner oder gleich der Maximallänge des String-Typs ist. Somit ist auch die leere Zeichenkette ein zulässiger Wert eines String-Typs.

Sowohl die Zeichen der Zeichenkette als auch deren Anzahl (die Länge der Zeichenkette) sind Komponenten des String-Typs. Auf die Zeichen der Zeichenkette kann durch Indizierung (siehe 9.6.2) und auf die Länge durch die vordefinierte Funktion Length (siehe 15.3) zugegriffen werden.

Auf Werte eines String-Typs sind die Vergleichs-Operatoren anwendbar. Sie können (als ganzes) zugewiesen, auf Textdateien oder Strings ausgegeben und von diesen (als Zeilenrest) eingelesen werden. Numerische Werte können in abdruckbarer Form in eine String-Variable ausgegeben und von ihr gelesen werden (siehe vordefinierte Prozeduren Readstring und Writestring 15.3). Mit weiteren vordefinierten Unterprogrammen können Strings manipuliert werden (siehe 15.3).

*Hinweis*

Komponenten eines Strings können nicht als Variablenparameter an Unterprogramme übergeben werden.

*Beispiele*

```
TYPE
   stdstring = String;         { implementierungsdefinierte Maximallänge }

   string10  = String [10];    { Maximallänge 10 }

   str       = String [20000]; { Maximallänge 20000 }
```

*Querverweise*

| | |
|---|---|
| Vergleichsoperatoren: | 9.3.4 |
| Indizierung: | 9.6.2 |
| Zuweisung: | 10.1.2 |
| vordefinierte Unterprogramme: | 15.3 |
| Ein- / Ausgabe: | 19 |

## RECORD-Typen

Ein RECORD-Typ definiert eine Struktur mit einer festen Anzahl von Komponenten die unterschiedliche Typen besitzen können. RECORD-Typen haben folgenden syntaktischen Aufbau:

```
RECORD-Typ      = "RECORD" Feldliste "END" .

Feldliste       = [ ( Festteil [ ";" Variantteil ] | Variantteil ) [ ";" ] ].

Festteil        = RECORD-Abschnitt { ";" RECORD-Abschnitt } .

RECORD-Abschnitt
                = Felddarstellungsliste ":" Typangabe .

Felddarstellungsliste
                = Felddarstellung { "," Felddarstellung } .

Felddarstellung = Bezeichner [ "(" Abstand [ ":" Bitbereich ] ")" ] .

Abstand         = Integer-Konstante .

Bitbereich      = Integer-Konstante ".." Integer-Konstante .

Variantteil     = "CASE" Variantenselektor "OF" Variante { ";" Variante } .

Variantenselektor
                = [ Kennungsfeld ":" ] Kennungstyp .

Kennungsfeld    = Felddarstellung .

Kennungstyp     = Ordinaltyp-Name .

Variante        = Auswahlliste ":" "(" Feldliste ")" .

Auswahlliste    = Auswahl {"," Auswahl} | "ELSE".

Auswahl         = Fallkonstante [".." Fallkonstante]

Fallkonstante   = Ordinal-Konstante.
```

Jede Komponente eines RECORD-Typs wird Feld genannt. Ein RECORD-Typ definiert für jedes Feld einen Feld-Bezeichner und einen Typ. Alle Feld-Bezeichner in einem RECORD-Typ einschließlich eventueller Varianten müssen verschieden sein.

Ist die Feldliste eines RECORD-Typs leer, dann besitzt der RECORD-Typ keine Felder und beschreibt einen Nullwert.

Werte eines RECORD-Typs können (als ganzes) zugewiesen werden. Auf die einzelnen Komponenten kann durch Feld-Selektion zugegriffen werden (siehe 9.6.3). Variablen eines RECORD-Typs können in WITH-Anweisungen vorkommen (siehe 10.5).

*Beispiel*

Beispiel eines RECORD-Typs mit nur einem Festteil und einem eingeschachtelten RECORD-Typ.

```
TYPE
   person = RECORD
      name,
      vorname    : String [20];
      alter      : 0..100;
      geburtstag : RECORD
         jahr : 0..2100;
         monat: 1..12;
         tag  : 1..31;
         END;
      END;
```

Das Feld geburtstag ist wiederum von einem RECORD-Typ, dessen Felder nicht Felder des Typs person sind.

*Hinweis*

Komponenten eines gepackten RECORD können nicht als Variablenparameter an Unterprogramme übergeben werden.

*Querverweise*

Konstante: 5.1
Feldselektion: 9.6.3
Zuweisung: 10.1.2
WITH-Anweisung: 10.5

## RECORD-Typen mit Varianten

In einem RECORD-Typ mit Variantteil können weitere Felder in Abhängigkeit des Wertes eines anderen Feldes definiert werden. Der Variantteil besteht aus mehreren Varianten, die alternativ zur Verfügung stehen. Die Feldliste in einem Variantteil hat wieder die allgemeine Form, kann also aus einem Festteil, einem Variantteil oder beidem bestehen.

Die Anzahl der Varianten in einem Variantteil wird durch den Kennungstyp im Variantenselektor festgelegt. Der Kennungstyp muß ein Ordinaltyp sein und der Typ jeder Auswahlkonstanten des Variantteils muß verträglich sein mit dem Kennungstyp.
Die letzte Variante eines Variantteils kann anstelle einer Auswahlliste das Wortsymbol "ELSE" enthalten. "ELSE" steht dann abkürzend für alle Werte des Kennungstyps, die nicht in der Auswahlliste dieses Variantteils auftreten.
Treten alle Werte des Kennungstyps bereits in den Auswahlliste auf, so darf ein ELSE-Teil nicht vorkommen.

```
Eine Auswahl der Form
         c1 .. cn
steht abkürzend für eine Auswahlliste
         c1, c2, ..., cn,
die alle Werte ci des Kennungstyps mit
         c1 ≤ ci ≤ cn
enthält.
```

Die Werte der Auswahlkonstanten in einem Variantteil müssen paarweise verschieden sein und die Menge ihrer Werte muß gleich der Menge der Werte des Kennungstyps sein, wenn kein ELSE-Teil angegeben wird.

Der Variantenselektor kann optional ein Kennungsfeld des Kennungstyps enthalten, d. h. es sind folgende Angaben möglich:

CASE Kennungsfeld: Kennungstyp OF...

oder

CASE Kennungstyp OF...

- **Kennungsfeld nicht vorhanden**

Vor einem lesenden oder schreibenden Zugriff auf ein Feld einer Variante wird automatisch genau diese Variante eingestellt. Wird dadurch die Variante gewechselt, so sind alle Felder der neu eingestellten Varianten undefiniert. Es ist deshalb ein Fehler, wenn in einem lesenden Zugriff auf ein Feld in einer Varianten die Variante gewechselt wird (da ein undefinierter Wert gelesen wird, siehe 9).

- **Kennungsfeld vorhanden**

Durch Zuweisung eines Wertes an das Kennungsfeld wird die Variante eingestellt, die mit dem Wert des Kennungsfeldes verknüpft ist. Mit dem Wechseln einer Variante sind alle Felder dieser Variante undefiniert, bis ihnen Werte zugewiesen werden.

Das Kennungsfeld wird, quasi als Bestandteil des Festteils, mit dem RECORD mitgeführt, um die jeweils eingestellte Variante abfragen zu können. Wenn kein Kennungsfeld angegeben ist, kann die eingestellte Variante nicht abgefragt werden.

*Beispiel für einen RECORD-Typ mit Fest- und Variantteil*

```
TYPE
   string20 = String[20];
   person = RECORD
      vorname : string20;
      nachname: string20;
      alter   : 0..99;
      CASE verheiratet : Boolean OF
         True  : (name_des_partners : string20);
         False : ( );
      END;
```

Das Kennungsfeld ist vom Typ Boolean. Die Feldliste hinter der Auswahlkonstanten "False" ist die sog. leere Feldliste. Zu dieser Variante sollen also außer den Feldern des Festteils und des Kennungsfeldes überhaupt keine weiteren Felder existieren.

*Hinweise*

- Ein RECORD-Typ mit Varianten ermöglicht die Definition von Strukturen mit verschiedenen Ausprägungen. Jede Ausprägung kann unterschiedliche Felder enthalten und damit i. a. unterschiedliche Speicherplatzanforderungen haben. Eine Implementierung kann die Varianten im Speicher "übereinander" legen (Regelfall), sie kann sie aber auch hintereinander allokieren.
- Es ist nicht empfehlenswert, in Varianten ein Feld von einem FILE-Typ zu definieren. Nach Umschaltung der Variante ist das Feld undefiniert und als Folge treten Dateifehler auf (siehe auch Hinweise in 6.3.3.2).
- Nach Aktivierung einer Variante (Umschalten der Variante) müssen erst alle Felder dieser Variante mit Werten belegt werden, da durch die Umschaltung die Werte der Felder undefiniert sind. Die häufig geübte Praxis, in einer Variante die Felder mit Werten zu besetzen und in einer anderen Variante unter einem anderen Typ wieder auszulesen, ist fehlerhaft. Es kann auch Implementierungen geben, die Varianten nicht "übereinander" legen, und damit nicht die erhoffte Funktion bewirken. Pascal-XT bietet für solche Typkonversionen die vordefinierte Prozedur Convert an (siehe 15.10).
- Fehlerhafte Zugriffe auf nicht eingestellte Varianten können nur bei Angabe eines Kennungsfeldes und eingeschalteter Check-Option erkannt werden (siehe 9.6.3).

*Querverweise*

Check-Option: 9.6.3, 16.2
Convert: 15.10

RECORD-Typen mit Angaben zur Speicherdarstellung

Pascal-XT erlaubt, bei RECORD-Typen Angaben über die Darstellung von Werten des RECORD-Typs im Speicher zu machen. Diese Angaben haben keinen Einfluß auf die Ausführung eines Pascal-Programms. Sie sind jedoch notwendig, wenn Schnittstellen zu fremden Systemen oder Anschlüsse an andere Sprachen in Pascal-XT-Programmen eingehalten werden müssen.

Die Speicherdarstellung eines Wertes von einem RECORD-Typ wird durch die Darstellung der Werte der Felder bestimmt. Für jedes Feld kann der Abstand relativ zum Anfang des RECORD-Typs und der gewünschte Bitbereich für Werte des Feldes definiert werden.

Es darf keine Überlappung der Speicherbereiche für verschiedene Felder des Festteils einer Feldliste geben. Für Felder in verschiedenen Varianten einer Feldliste kann es implementierungsdefinierte Einschränkungen (siehe unten) geben, insbesondere wenn in den Varianten Felder eines FILE-Typs enthalten sind.

**Implementierungsdefinierte Eigenschaften**

– Die Größe einer Speichereinheit ist implementierungsdefiniert. Sie kann ein Byte oder ein Vielfaches (i. a. eine Zweierpotenz) von einem Byte sein.

– Für die Abstands- und Bitbereichs-Angabe in Feldbezeichnern eines RECORD-Typs können implementierungsdefinierte Einschränkungen gelten (z. B. Ausrichtungsbedingungen für Abstandsangaben, Alignment der Felder, Allokierung von Varianten, siehe auch Hinweise).

- **Abstand**

Für jeden Wert eines RECORD-Typs werden die den einzelnen Feldern entsprechenden Komponenten mit einem festen Abstand zum Anfang des gesamten RECORDs im Speicher abgelegt. Dieser Abstand ist nur vom RECORD-Typ selbst, nicht aber von den einzelnen Variablen oder Komponenten dieses Typs anhängig. Wird für ein RECORD-Feld keine Angabe über den Abstand gemacht, so wird der Abstand vom Compiler bestimmt. Eine Abstandsangabe hat zur Folge, daß alle mit mit dem deklarierten Feld assoziierten Komponenten-Werte stets gemäß dieser Angabe relativ zum Anfang des für den gesamten RECORD-Wert reservierten Speichers abgelegt werden.

Der Wert einer als Abstand angegebenen Konstanten muß vom Typ Integer sein und er darf nicht negativ sein. Die Werte der Abstände, die in einer Feldliste angegeben sind, müssen in aufsteigender Folge entsprechend der textlichen Reihenfolge der Feld-Bezeichnungen angegeben werden.

- **Bitbereich**

Bitbereiche dürfen nur in gepackten RECORD-Typen und nur für Felder eines Ordinaltyps oder eines Zeigertyps angegeben werden. Der Bitbereich gibt dann die relative Lage einer Bit-Folge zu der durch den Abstand spezifizierten Speichereinheit an, in der die Werte des Feldes rechtsbündig abgelegt werden. Die Numerierung der Bits beginnt mit "0" (dies ist das 1. Bit innerhalb der durch den Abstand bestimmten Speichereinheit), wird bis zum letzten Bit dieser Speichereinheit weitergeführt und dann mit dem 1. Bit der nächsten Speichereinheit fortgesetzt (das ist die durch den um 1 erhöhten Wert des Abstands bestimmte Speichereinheit) usw.

Die Konstanten in einem Bitbereich müssen von einem Integer-Typ sein und dürfen nicht negativ sein. Die erste Konstante bestimmt das erste, die zweite das letzte zu der Bit-Folge gehörende Bit. Bitbereiche müssen stets groß genug gewählt werden, um alle möglichen Werte des Typs des Feldes aufnehmen zu können.

Die Bitbereiche, die sich auf ein und denselben Abstandswert beziehen, müssen entsprechend der textlichen Reihenfolge der Feld-Bezeichnungen in aufsteigender, nicht überlappender Reihenfolge angegeben werden.

*Hinweise*

– Der Speicherbedarf für die Darstellung von Werten kann im Benutzerhandbuch nachgelesen werden.

– Die Speicherdarstellung von FILE-Variablen ist implementierungsabhängig und dem Anwender nicht bekannt. RECORD-Variable, die Felder eines FILE-Typs enthalten, sollen daher auch nicht an externe Systeme weitergereicht werden. Insbesondere sind Abstandsangaben in einem solchen RECORD-Typ nicht zu empfehlen.

– Eine Implementierung kann die zu einem Variantteil gehörigen Varianten zur Speicherplatzoptimierung "übereinander" legen, es wird aber nicht vorgeschrieben. Insbesondere wenn Varianten Felder eines FILE-Typs enthalten, kann implementierungsabhängig eine andere Allokierung (z.B. nacheinander) erfolgen. Bei Verwendung von Feldern eines FILE-Typs kann die Verwendung von Abstandsangaben weiteren implementierungsdefinierten Einschränkungen unterworfen sein (siehe Benutzerhandbuch).

*Beispiel 1*

Im folgenden Beispiel wird eine gepackte Darstellung des Datums als RECORD-Typ angegeben und im Bild 6-1 die Darstellung des Datums 27.12.87 im Speicher veranschaulicht. Die Abstands- und Bitbereichsangaben beziehen sich auf eine Implementierung, bei der die Speichereinheit ein Byte ist.

```
TYPE
   date = PACKED RECORD
      tag   (0: 0..4) : 1..31;
      monat (0: 5..8) : 1..12;
      jahr  (1: 1..15): 0..3000;
      END;
```

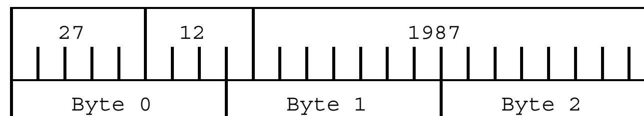



Bild 6-1: Darstellung des RECORD-Typs im Speicher

*Beispiel 2*

Abstands- und Bitbereichsangaben in einem RECORD-Typ mit Variantteil. Die angegebenen Abstands- und Bitbereichs-Angaben können implementierungsdefiniert auch unterschiedlich sein.

```
TYPE
   t1 = PACKED RECORD
      a (0: 0..3) : 0..15;
      b (0: 4..7) : (state1, state2, state3);
      CASE tag (1) : Boolean OF
         True   : (c (2) : Integer);
         False  : (d (2), e (3) : Char)
      END;
```

*Querverweise*

PACKED: 6.3
New, Dispose : 15.2

## SET-Typen

Mengen werden in Pascal durch SET-Typen beschrieben, die folgende Syntax haben:

```
SET-Typ  = "SET" "OF" Basistyp .

Basistyp = Ordinal-Typangabe .
```

Als Elemente einer Menge sind nur Werte eines Ordinaltyps zugelassen, der dann Basistyp der Menge heißt. Struktur und Werte eines SET-Typs werden durch die Potenzmenge der Wertemenge dieses Basistyps bestimmt. Somit ist jeder Wert eines SET-Typs eine Menge, die entweder leer ist, oder aus einer beliebigen Kombination paarweise verschiedener Werte des Basistyps besteht.

Es gibt keine Einschränkung den kleinsten oder den größten Wert des Basistyps betreffend, solange die Anzahl der Werte des Basistyps eine implementierungsdefinierte Grenze nicht überschreitet; insbesondere kann der Basistyp auch ein ganz oder teilweise im Negativen liegender Teilbereich eines Integer-Typs sein.
Im letzteren Fall müssen Mengenbildner qualifiziert angegeben werden (siehe 9.4).

### Implementierungsdefinierte Eigenschaft

Die maximale Anzahl der Werte eines Basistyps einer Menge kann auf einen implementierungsdefinierten Bereich beschränkt sein.

Werte eines SET-Typs können (als ganzes) zugewiesen werden. Die Mengen- und Vergleichsoperatoren sind auf sie anwendbar. Mit dem IN-Operator kann abgefragt werden, ob ein bestimmter Wert Element einer Menge ist. Werte eines Mengentyps werden durch Mengenbildner erzeugt.

*Beispiele für zulässige Mengentypen*

```
SET OF 0..50
SET OF -20..20
SET OF -30000..-29990
SET OF 12321..12345
SET OF Char
SET OF Boolean
SET OF (gelb, rot, blau)
SET OF '0'..'9'
```

Werte des Typs SET OF (gelb, rot, blau) können sein:

```
[]                          {  die leere Menge }

[gelb], [rot], [blau]       {  einelementige Menge }

[gelb, rot], [gelb, blau]   {  zweielementige Mengen }
[rot, blau]

[gelb, rot, blau]           {  dreielementige Menge }
```

*Hinweis*

SET-Typen ermöglichen ein sehr elegantes Arbeiten mit Informationen, die in den meisten anderen Programmiersprachen durch Bitcodierungen dargestellt werden müssen.

*Querverweise*

Ordinaltypen: 6.2
Mengenoperator: 9.3.3
IN-Operator: 9.3.3
Vergleichsoperator: 9.3.4
Mengenbildner: 9.4
Zuweisung: 10.1.2

## FILE-Typen

Ein FILE-Typ (Dateityp) beschreibt (theoretisch beliebig lange) Folgen von Werten desselben Komponententyps, zusammen mit einer aktuellen Position in jeder Folge und einem Bearbeitungsmodus, der anzeigt, ob die Folge gelesen oder geschrieben wird. Die aktuelle Position innerhalb der Datei wird auch als Dateizeiger bezeichnet. Eine leere Datei entspricht einer leeren Folge.

Pascal kennt nur sequentielle Dateien. Diese können nur sequentiell von vorn nach hinten bearbeitet werden. Zu einem Zeitpunkt kann immer nur auf eine Komponente zugegriffen werden. Dazu ist mit jeder Variablen eines FILE-Typs eine Puffervariable assoziiert (siehe 7).

Es gibt zwei vordefinierte *FILE-Typ*-Bezeichner: Any_File (siehe 6.5.1) und Text (siehe 6.3.5.2).

Zulässige Operationen auf Dateien sind ausschließlich durch Zugriffe auf ihre Puffervariable und durch vordefinierte Unterprogramme realisiert.

*Hinweis*

Pascal-XT-Implementierungen können in vordefinierten Paketen Erweiterungen zur Dateibearbeitung, wie Direktzugriff, Mehrfachbearbeitung von Dateien, unterschiedliche Eröffnungsmodi usw. anbieten.

*Querverweise*

| | |
|---|---|
| Puffervariable: | 7 |
| Sichtbarkeitsregeln: | 12 |
| Vordefinierte Unterprogramme: | 15 |
| Ein-/Ausgabe: | 19 |

### Allgemeine Dateien

Die Syntax eines FILE-Typs lautet:

```
FILE-Typ       = "FILE" "OF" Komponententyp .

Komponententyp = Typangabe.
```

In einem FILE-Typ ist als Komponententyp eine Typangabe unzulässig, die selbst wieder für einen FILE-Typ steht oder für einen strukturierten Typ mit irgendeiner (direkten oder indirekten) Komponente eines FILE-Typs.

*Hinweis*

Die Verwendung von Zeigertypen oder von strukturierten Typen, die (direkt oder indirekt) Komponenten eines Zeigertyps besitzen, ist nur für lokale Dateien, die innerhalb derselben Programmausführung geschrieben und wieder gelesen werden, sinnvoll. Nach Beendigung einer Programmausführung sind nämlich alle Verweiswerte ungültig, so daß in einer späteren Programmausführung die Dereferenzierung von aus Dateien gelesenen Verweiswerten zu Fehlern führt.

*Beispiele für FILE-Typen*

```
FILE OF ARRAY [1..50] OF Integer
FILE OF RECORD
   feld : ARRAY [1..50] OF Char;
   zahl : Integer;
   bool : Boolean;
   END;
```

*Querverweise*

Zeigertypen, Verweiswerte: 6.4

### Der FILE-Typ Text

Der vordefinierte Typ-Bezeichner Text repräsentiert einen speziellen FILE-Typ, der eine zusätzliche Eigenschaft hat: Er beschreibt eine Folge von Zeilen, und jede Zeile besteht aus einer Folge von Zeichen vom Typ Char. Jede Zeile wird durch einen speziellen Komponentenwert Zeilenende abgeschlossen, der innerhalb eines Pascal-Programms wie ein Leerzeichen (Blank, ' ') behandelt wird, außer bei den vordefinierten Unterprogrammen Reset, Readln, Eoln, Writeln und Page. Eine Leerzeile besteht nur aus der Komponente Zeilenende.

Beim Lesen einer Textdatei können immer nur vollständige Zeilen auftreten, d. h. Zeilen, die mit einem Zeilenende abgeschlossen sind, selbst dann, wenn beim Schreiben dieser Datei die letzte Zeile nicht mit Writeln abgeschlossen wurde.

Eine Datei vom Typ Text wird als Textdatei bezeichnet. Alle vordefinierten Unterprogramme zur Bearbeitung von Variablen vom Typ FILE OF typ sind auch auf Textdateien anwendbar. Zusätzlich gibt es die vordefinierten Unterprogramme

- Readln
- Writeln
- Eoln
- Page

die nur auf Textdateien anwendbar sind.

Bei der Ein-/Ausgabe auf Textdateien können außer Zeichen auch Zeichenketten, Integer-Zahlen, Real-Zahlen und boolesche Werte ausgegeben werden. Die Ein-/Ausgabe von Zahlen beinhaltet auch eine Konversion von der eingelesenen Zeichenfolge in die Interndarstellung (Binärformat) und umgekehrt. Boolesche Werte werden bei der Ausgabe entsprechend ihrem Wert als Zeichenkette ausgegeben (siehe 15.1, 19).

**Input** und **Output**

Die vordefinierten Variablen-Bezeichner Input und Output stehen für Textdateien (siehe 11.5).

*Querverweise*

| | |
|---|---|
| Input / Output: | 11.5, 15 |
| vordefinierte Unterprogramme: | 15 |
| Unterprogramme zur Dateibearbeitung: | 15.1 |

# Zeigertypen

Zeiger sind Verweise auf dynamische Variable, die während der Ausführung eines Programms erzeugt und vernichtet werden können. In Kapitel 7 sind dynamische Variable eingehender beschrieben.

Die Erzeugung und Vernichtung dynamischer Variablen erfolgt nur über vordefinierte Unterprogramme (15.2). Das Arbeiten mit Zeigern und dynamischen Variablen ist in Kapitel 20 beschrieben.

Zeiger können zugewiesen werden. Dadurch wird aber nicht die dynamische Variable, sondern nur der Verweis auf sie kopiert. Die Vergleichsoperatoren "=" und "<>" sind auf Zeiger anwendbar. Auch hier werden nicht die Werte der dynamischen Variablen sondern nur die Verweise verglichen. Zugriffe auf die dynamische Variable erfolgen durch Dereferenzierung des Zeigers.

Das Schreiben mehrfach verwendbarer Pakete ist mit Hilfe des generischen Zeigertyps Pointer möglich, der in 6.5.2 beschrieben ist.

Die Syntax von Zeigertypen lautet:

```
Zeigertyp   = "↑" Domänentyp .

Domänentyp  = Typ-Name .
```

Der Typ-Bezeichner des Domänentyps kann bereits vor seiner Definition verwendet werden (siehe auch private Zeigertypen). Damit wird die Definition rekursiver Datenstrukturen möglich (siehe Beispiel).

Die Werte eines Zeigertyps, bestehen aus dem NIL-Wert und einer Menge von Verweiswerten, von denen jeder auf eine eigene dynamische Variable des Domänentyps verweist.

Die Menge der Verweiswerte eines Zeigertyps ändert sich während der Ausführung eines Programms durch die Erzeugung und Vernichtung dynamischer Variablen. Jeder Verweiswert eines Zeigertyps kann aber immer nur auf eine dynamische Variable mit dem in der Definition des Zeigertyps angegebenen Domänentyp verweisen. Diese strenge Typbindung von Zeigern dient der Sicherheit von Programmen.

In Norm-Pascal hat das Wortsymbol NIL keinen speziellen Zeigertyp, sondern nimmt einen passenden Zeigertyp an, um den Regeln der Verträglichkeit bzw. Zuweisungsverträglichkeit zu genügen. Da der NIL-Wert kein Verweiswert ist, zeigt er auf keine dynamische Variable.
In Pascal-XT wird NIL als Wert des generischen Zeigertyps Pointer (6.5.2) aufgefaßt.

Sei T ein beliebiger Typ, dann ist ↑T der Zeigertyp zum Domänentyp T. Ist TP eine Variable vom Zeigertyp ↑T, dann sind die Werte von TP Verweise (Zeiger) auf dynamische Variable des Typs T.

*Beispiel*

Der Domänentyp wird vor seiner Definition verwendet.

```
TYPE
   person_ptr = ↑person;
   person     = RECORD
                   name:  string;
                   vater: person_ptr;
                END;
```

*Querverweise*


| | |
|---|---|
| Generischer Zeigertyp: | 6.5.2 |
| Zeigervariable: | 7 |
| Vergleichsoperatoren: | 9.3.4 |
| Dereferenzierung: | 9.6.4 |
| Zuweisung | 10.1.2 |
| Privater Zeigertyp: | 11.4 |
| Sichtbarkeitsregeln: | 12 |
| Vordefinierte Unterprogramme: | 15.2 |
| Konzept | 20 |

# Generische Typen

## Der vordefinierte FILE-Typ Any_File

Um Erweiterungen zur Dateibearbeitung in vordefinierten Paketen anbieten zu können, wird in Pascal-XT zusätzlich der generische FILE-Typ eingeführt, der durch den vordefinierten Typ-Bezeichner Any_File bezeichnet wird.

Der Typ Any_File ist nur als Typ von Variablen-Parametern zulässig, nicht aber als Typ einer Variablen oder einer Komponente einer Variablen. Die vordefinierten Unterprogramme zur Dateibearbeitung können nicht mit einem Aktualparameter dieses Typs aufgerufen werden.

*Querverweis*

Unterprogramme zur Dateibearbeitung: 15.1

## Der vordefiniete Zeigertyp Pointer

Pascal-XT kennt zusätzlich den vordefinierten generischen Zeigertyp, der durch den vordefinierten Typ-Bezeichner Pointer bezeichnet wird. Mit Hilfe dieses Typs können allgemein verwendbare Unterprogramme zur Bearbeitung von dynamischen Variablen geschrieben werden. So ist es z.B. möglich, einen Satz von Unterprogrammen zur Verwaltung von linear verketteten Listen zu schreiben, die universell für Listen verschiedener Element-Typen verwendet werden können (siehe Beispiel).

Die Menge der Verweiswerte des generischen Zeigertyps Pointer ist die Vereinigung der Verweiswerte aller im Programm definierter Zeigertypen. Der generische Zeigertyp ist verträglich mit jedem anderen Zeigertyp. Mit dem generischen Zeigertyp ist die strenge Typbindung aufgehoben, da eine Zeigervariable dieses Typs auf dynamische Variable eines beliebigen Domänentyps verweisen kann. Es ist jedoch ein Fehler, wenn dies indirekt zur Typkonversion mißbraucht wird (siehe 6.6.2).

Objekte des generischen Zeigertyps können nicht dereferenziert werden oder als Parameter beim Aufruf der vordefinierten Prozeduren New oder Dispose verwendet werden. Vor dem Dereferenzieren müssen die Verweiswerte erst in Variable eines im Programm definierten Zeigertyps kopiert werden.

*Beispiel*

```
TYPE
```

```
list = ↑list_element;
list_element = RECORD
   object: Pointer;
   tail  : list
   END;
```

```
PROCEDURE insert (VAR queue : list; item : Pointer);
VAR
   l : list;
BEGIN
   New (l);
   l↑.object := item;
   l↑.tail   := queue;
   queue      := l;
END;
```

## Der vordefinierte Typ Any_Type

Bei der Parameterübergabe an Variablenparameter muß der Aktualparameter denselben Typ wie der Formalparameter besitzen. Bei Sprachgemischen ist diese strikte Typüberprüfung manchmal hinderlich, wenn z.B. Adressen von Variablen verschiedenen Typs an dieselbe externe Prozedur übergeben werden müssen. Ein vordefinierter generischer Typ, der durch den vordefinierten Typ-Bezeichner Any_Type bezeichnet wird, ermöglicht eine Umgehung der Typüberprüfung.

Der Typ Any_Type kann nur als Typ von formalen Variablenparametern in solchen Prozeduren und Funktionen auftreten, in denen eine von Forward verschiedene Direktive angegeben ist.

*Beispiel*

In diesem Beispiel können an die externe Prozedur ext Variable beliebigen Typs übergeben werden, d. h. in ext sind die Adressen der Variablen verfügbar. Als zweiter Parameter wird noch die Größe der jeweiligen Variable mit übergeben.

```
VAR
   buf1 : ARRAY [1..100] OF Integer;
   buf2 : ARRAY [1..50]  OF Real;
   buf3 : RECORD  ... END;

PROCEDURE ext (VAR buffer: Any_Type; size: Integer); external;

BEGIN
   ext (buf1, Sizeof(buf1));
   ext (buf2, Sizeof(buf2));
   ext (buf3, Sizeof(buf3));
END
```

*Querverweise*

Variablenparameter: 8.5.2
Direktiven: 8.6

# Identität und Verträglichkeit von Typen

## Identität von Typen

Identität von Typen wird gefordert:

- Bei der VAR-Parameterübergabe (siehe 8.5.2)
- Bei den Parametern und Funktionsergebnissen von Parameter-Prozeduren und Funktionen (siehe 8.5.3)
- Bei mehreren Aktualparametern, die Formalparametern aus einer einzigen Konformreihungs-Parameter-Spezifikation zugeordnet werden sollen (siehe 8.5.4)
- Bei der Konformität von Konformreihungsschemata (bezüglich der Komponententypen, siehe 8.5.4).

Eine Typangabe kann gemäß der Syntax in 6.1 eines der beiden folgenden Formate besitzen:

a) neuer Typ:
In diesem Fall definiert die Typangabe einen neuen Typ, der mit keinem anderen neuen Typ identisch ist (auch dann nicht, wenn die beiden Typangaben textlich völlig identisch sind).

b) Typ-Name:
In diesem Fall ist der durch die Typangabe festgelegte Typ identisch mit dem in der Typdefinition des Typ-Namen angegebenen Typ. Durch die Verwendung von Typ-Namen als Typangaben in Typdefinitionen ist es möglich, mehrere Typ-Namen zu definieren, die identische Typen bezeichnen.

*Beispiele zu a)*

```
TYPE
   t1 = ARRAY [1..5] OF Char;
   t2 = ARRAY [1..5] OF Char;
```

Hier bezeichnen t1 und t2 trotz völlig gleicher Schreibweise der Typangaben zwei verschiedene, mit einander nicht einmal verträgliche Typen.

```
VAR
   i:    1..5;
   j, k: 1..5;
```

Die Variablen i und j besitzen verschiedene Typen (die allerdings als Teilbereichstypen desselben Wirtstyps Integer miteinander verträglich sind, siehe 6.6.2).

Die Variablen j und k besitzen hingegen den selben Typ.

*Beispiele zu b)*

```
TYPE
   range    = 0 .. Maxint;
   cardinal = range;
   positiv  = cardinal;
   natural  = range;
VAR
   a: range;
   b: cardinal;
   c: positiv;
   d: natural;
```

Hier bezeichnen range, cardinal, positiv und natural alle den selben Typ. Ebenso besitzen die Variablen a, b, c und d alle den selben Typ.

Die folgenden Beispiele illustrieren, daß der Typ-Name auch vordefiniert oder in einem anderen Paket definiert sein kann:

```
TYPE
   generic_ptr = Pointer;
   compl_nr    = complex_definitions.complex;
```

Hier bezeichnen generic_ptr und der vordefinierte Typ-Bezeichner Pointer den selben Typ, nämlich den vordefinierten generischen Zeigertyp.
Ebenso bezeichnet compl_nr den selben Typ wie der in einem (mit einer WITH-Klausel referierten) Paket complex_definitions definierten Typ-Bezeichner complex (siehe Beispiel in Kapitel 17).

```
VAR
   x: Long_Real;
   y: Long_Real;
```

Da die Typangabe Long_Real ein (vordefinierter) Typ-Name (und nicht wie 1..5 im Beispiel oben ein neuer Typ) ist, besitzen die Variablen x und y den selben Typ.

*Querverweise*

| | |
|---|---|
| VAR-Parameter: | 8.5.2 |
| Parameter-Unterprogramme: | 8.5.3 |
| Konformreihungsschemata: | 8.5.4 |
| Typangabe: | 6.1 |
| neuer Typ: | 6.1 |
| Typ-Name | 6.1 |
| Verträglichkeit von Typen: | 6.6.2 |
| Teilbereichstyp, Wirtstyp: | 6.2.6 |
| Pointer, generischer Zeigertyp: | 6.5.2 |
| Pakete: | 11.2 |

## Verträgliche Typen

Die Verträglichkeit von Typen wird gefordert:

- Für die Operanden von Vergleichsoperatoren (siehe 9.3.4).
- Bei der Konformität von Konformreihungsschemata (bzgl der Indextypen) (siehe 8.5.4).
- Für Anfangs- und Endwert in der FOR-Anweisung (10.4.3)
- Bei der Zuweisungsverträglichkeit (6.6.2).
- Für die Operanden von Mengenoperatoren (siehe 9.3.3).
- Für die Auswahlkonstanten in Variantteilen
- Für Fall-Konstanten in CASE-Anweisungen

Die Verträglichkeitsanforderung wird bereits zur Übersetzungszeit überprüft.

Zwei Typen T1 und T2 heißen **verträglich**, wenn irgendeine der folgenden Aussagen wahr ist:

a) T1 und T2 sind derselbe Typ (siehe 6.6.1).

b) T1 ist ein Teilbereich von T2, oder T2 ist ein Teilbereich von T1, oder T1 und T2 sind Teilbereiche desselben Wirtstyps T3.

c) T1 und T2 sind SET-Typen mit verträglichen Basistypen und sie sind entweder beide gepackt oder beide nicht gepackt.

d) T1 und T2 sind Zeichenkettentypen fester Länge, wobei die Anzahl der Komponenten von T1 und T2 verschieden sein kann.
In Norm-Pascal müssen T1 und T2 die gleiche Anzahl von Komponenten enthalten.

e) T1 (T2) ist ein String-Typ und T2 (T1) ist ein Zeichenkettentyp (Typ Char, Zeichenkettentyp fester Länge, String-Typ).

f) T1 (T2) ist ein Zeigertyp und T2 (T1) ist der generische Zeigertyp.

g) T1 (T2) ist der Typ Short_Integer und T2 (T1) ist der Typ Long_Integer.

i) T1 (T2) ist der Typ Short_Real und T2 (T1) ist der Typ Long_Real.

*Querverweise*

| | |
|---|---|
| Integer-Typ: | 6.2.1 |
| Real-Typ: | 6.2.2 |
| Teilbereichstyp, Wirtstyp: | 6.2.6 |
| Zeichenkettentyp: | 6.3.2 |
| Zeichenkettentyp fester Länge: | 6.3.2.1 |
| String-Typ: | 6.3.2.2 |
| Varianteil: | 6.3.3.1 |
| SET-Typ, Basis-Typ: | 6.3.4 |
| Zeigertyp: | 6.4 |
| Konformreihungsschemata: | 8.5.4 |
| Mengenoperator: | 9.3.3 |
| Vergleichsoperator: | 9.3.4 |
| CASE-Anweisung: | 10.3.2 |
| FOR-Anweisung: | 10.4.3 |

## Zuweisungsverträglichkeit von Typen

Wird ein Wert an eine Variable zugewiesen, dann muß der Typ des Werts zuweisungsverträglich zum Typ der Variablen sein. Die Zuweisungsverträglichkeit wird gefordert, wenn eine Zuweisung explizit (Punkte 1 bis 3) oder implizit (Punkte 4 bis 9) erfolgt:

1) Bei der Zuweisung eines Wertes an eine Variable (siehe 10.1.2).
2) Bei der Zuweisung eines Wertes an einen Funktionsbezeichner (siehe 8.5.3).
3) Für Anfangs- und Endwert in der FOR-Anweisung (10.4.3)
4) Bei Wertparameterübergabe (siehe 8.5.1).
5) Bei Aggregaten (siehe 9.5).
6) Bei Read (f, v) (siehe 15).
7) Bei Write (f, a) (siehe 15).
8) Bei der Indizierung (siehe 9.6.2)
9) Bei Mengenbildnern (siehe 9.4)

Ein Wert vom Typ T2 heißt **zuweisungsverträglich** zu einem Typ T1, wenn irgendeine der folgenden Aussagen wahr ist:

a) T1 und T2 sind derselbe Typ und dieser Typ ist weder ein FILE-Typ noch enthält er (direkt oder indirekt) eine Komponente eines FILE-Typs.

b) T1 ist ein Real-Typ, T2 ein Integer-Typ.

c) T1 und T2 sind verträgliche Ordinaltypen und der Wert vom Typ T2 liegt im Wertebereich des Typs T1.
Bei der Ausführung eines Programms tritt ein Range_Error auf, bzw. ein Index_Error oder ein Set_Error, wenn diese Bedingung nicht erfüllt ist.

d) T1 ist der Typ Short_Real und T2 ist der Typ Long_Real und eine Approximation des Wertes vom Typ T2 liegt im Wertebereich von Typ T1. Bei der Ausführung eines Programms tritt ein Numeric_Error auf, wenn diese Bedingung nicht erfüllt ist.

e) T1 und T2 sind verträgliche SET-Typen und alle Elemente des Wertes vom Typ T2 liegen im Wertebereich des Basistyps von T1.
Bei der Ausführung eines Programms tritt ein Set_Error auf, wenn diese Bedingung nicht erfüllt ist.

f) T1 und T2 sind verträgliche Zeichenkettentypen fester Länge und der gleichen Anzahl von Komponenten.

g) T1 ist ein String-Typ mit der Maximallänge n und T2 ist ein Zeichenkettentyp mit höchstens n Komponenten (siehe Tabelle 6-1).
Bei der Ausführung eines Programms tritt ein String_Error auf, wenn T2 mehr als n Komponenten enthält.

h) T1 ist ein Zeichenkettentyp fester Länge mit n Komponenten und T2 ist ein String-Typ. Die aktuelle Länge der Zeichenkette vom Typ T2 muß genau n sein.
Bei der Ausführung eines Programms tritt ein String_Error auf, wenn die aktuelle Länge der Zeichenkette vom Typ T2 ungleich n ist.

i) T1 und T2 sind verträgliche Zeigertypen.
Bei der Ausführung eines Programms tritt ein Fehler (mit undefinierten Auswirkungen) auf, wenn T2 der generische Zeigertyp ist und der Wert von T2 ist ein Verweiswert, der auf eine dynamische Variable zeigt, deren Typ nicht mit dem Domänentyp von T1 übereinstimmt.

| v := e | | |
|---|---|---|
| Typ von v | Typ von e | Zuweisungsverträglich |
| Char | Char<br>PACKED ARRAY [1..k] OF Char<br>String[n] | ja<br>nein<br><br>nein |
| PACKED ARRAY [1..m] OF Char | Char<br><br>PACKED ARRAY [1..k] OF Char<br>String[n] | nein<br><br>wenn m = k<br><br>wenn m = Length(e) |
| String[n] | Char<br>PACKED ARRAY [1..k] OF Char<br>String[k] | ja<br>wenn n >= k<br><br>wenn n >= Length(e) |

Tabelle 6-1: Zuweisungsverträglichkeit bei Zeichenketten

*Hinweise*

– Ein PACKED ARRAY [1..n] OF Char ist in Pascal-XT nur dann ein Zeichenketten-Typ, wenn gilt: 1 < n <= 32767 (siehe 6.3.2).

– Welche der oben genannten Fehler bei der Ausführung eines Programmes erkannt werden, ist implementierungsdefiniert.

- Bei der Zuweisung eines Wertes von einem String-Typ an eine Variable eines Zeichenkettentyps mit n Komponenten muß die aktuelle Länge der Zeichenkette gleich n sein. Hat der String eine von n verschiedene Länge, dann kann mit Hilfe einer benutzerdefinierten Funktion der String auf die gewünschte aktuelle Länge gebracht werden (z. B. durch Abschneiden bzw. Auffüllen mit Leerzeichen). Die Definition der Funktion könnte folgendermaßen aussehen:

```
CONST
   std_str_length = 80;
TYPE
   std_string     = String[std_str_length];

FUNCTION adjust (s: std_string; len: Short_Integer): std_string;
TYPE
   blanks = PACKED ARRAY [1 .. std_str_length] OF Char;
CONST
   filler = blanks (' ': std_str_length);
BEGIN
   IF Length(s) >= len THEN
      adjust := Substring (s, 1, len)
   ELSE
      adjust := Concat
                 (s, substring (filler, 1, len - Length(s)))
END;
```

- Bei der Zuweisung eines Ausdrucks von einem Integer-Typ an eine Real-Variable wird der (Integer-) Wert des Ausdrucks vor der Zuweisung in einen Realwert konvertiert. Die Berechnung des Ausdrucks erfolgt allerdings in Integer-Arithmetik (siehe auch 9.3.1).

*Beispiele*

```
VAR  range      : 1 .. 5;
     string5    : String [5];
     fixstr3    : PACKED ARRAY [1..3] OF Char;
     menge      : SET OF 0 .. 255;
     short      : Short_Real;


 { Beispiele, in denen die Zuweisungsverträglichkeit  }
 { während der Programmausführung verletzt wird       }

BEGIN
     range      := 6;           { führt zu einem Range_Error }

     string5    := '123456';    { führt zu einem String_Error }

     string5    := '12';
     fixstr3    := string5;     { führt zu einem String_Error }

     menge      := [1, 256];    { führt zu einem Set_Error }

     short      := 10E100       { führt zu einem Numeric_Error }
END;
```

*Querverweise*

| | |
|---|---|
| Ordinal-Typ: | 6.2 |
| Integer-Typ: | 6.2.1 |
| Real-Typ: | 6.2.2 |
| Komponente: | 6.3 |
| Zeichenkettentyp: | 6.3.2 |
| String-Typ: | 6.3.2.2 |
| SET-Typ: | 6.3.4 |
| FILE-Typ: | 6.3.5 |
| Zeigertyp: | 6.4 |
| Funktion: | 8.2 |
| Wertparameter: | 8.5.1 |
| Mengenbildner: | 9.4 |
| Aggregat: | 9.5 |
| Indizierung: | 9.6.2 |
| Zuweisung: | 10.1.2 |
| Read, Write: | 15.1 |
| Length: | 15.3 |

# Attribute von Typen

Typen besitzen neben dem sie definierenden Wertebereich und den zulässigen Operatoren bzw. Funktionen eine Reihe von Attributen, die z.B. die Darstellung der Werte der Typen betrifft. Die Werte der Attribute können über die verschiedenen vordefinierten Funktionen (15.9) geliefert werden.

### Ausrichtung

Werte eines Typs müssen im Speicher evtl. implementierungsabhängigen Ausrichtungen unterliegen. Diese Ausrichtung ist ein Vielfaches der implementierungsdefinierten Speichereinheit (6.3.3.2). Die Funktion Alignof liefert die geforderte Ausrichtung.

### Speicherbedarf in Speichereinheiten

Zur Darstellung der Werte eines Typs wird eine implementierungsabhängige Anzahl von Speichereinheiten benötigt. Die Funktion Sizeof liefert die Anzahl der belegten Speichereinheiten.

### Speicherbedarf in Bits

Zur Darstellung der Werte eines Ordinaltyps wird eine implementierungsdefinierte Mindestanzahl von Bits benötigt. Die Funktion Bitsizeof liefert die Minimalanzahl der benötigten Bits.

### Kleinster Wert eines Ordinaltyps

Der kleinste Wert eines Ordinaltyps wird durch die Funktion First geliefert.

### Größter Wert eines Ordinaltyps

Der größte Wert eines Ordinaltyps wird durch die Funktion Last geliefert.

### Maximallänge eines String-Typs

Die Werte eines String-Typs sind Zeichenketten variabler Länge. Die maximal zulässige Länge einer Zeichenkette eines String-Typs wird durch die in der Typdefinition angegebene Maximallänge des String-Typs beschränkt. Die Funktion Maxlength liefert die Maximallänge eines String-Typs.

*Querverweise*

Attributfunktionen: 15.9

# Variablen

Eine Variable ist ein Platzhalter für Werte. Einer Variablen können während der Programmausführung verschiedene Werte zugewiesen werden (siehe 10.1.2). Variablen können auch als Operanden in Ausdrücken vorkommen und repräsentieren dann den ihnen zuletzt zugewiesenen Wert (siehe 9). In Pascal besitzt jede Variable einen Typ. Einer Variablen können nur Werte ihres Typs zugewiesen werden.

Variable können nach der Art des Zugriffs unterschieden werden:

| | |
|---|---|
| Ganzvariable | sind deklarierte Variable,die als Ganzes angesprochen werden. |
| Komponentenvariable | sind Komponenten von Variablen mit einem strukturiertem Typ (ARRAY-Typ, RECORD-Typ, String-Typ). Auch die Komponenten sind wieder Variable und können einzeln angesprochen werden. |
| Dynamische Variable | sind Variable, auf die ein Verweiswert zeigt (siehe auch 7.2). |
| Puffervariable | sind Variable, die mit Variablen eines FILE-Typs verknüpft sind (siehe auch 7.2). |

Alle diese Arten von Variablen werden auch unter dem Begriff verallgemeinerte Variable (Variablen-Objekt) zusammengefaßt. Die Zugriffe auf die Variablen sind in 9.6 ausführlich beschrieben.

*Querverweise*

Ausdruck: 9
Zuweisung: 10.1.2

# Variablendeklaration

Variablendeklarationen haben folgende Syntax:

```
Variablendeklarationsteil
                = "VAR" Variablendeklaration
                       {Variablendeklaration}.

Variablendeklaration
                = Bezeichnerliste ":" Typangabe ";".

Bezeichnerliste = Bezeichner {"," Bezeichner}.

Variablen-Name  = [ Paket-Bezeichner "." ] Bezeichner.
```

Jeder Bezeichner in der Bezeichnerliste einer Variablendeklaration steht für eine eigene Variable, deren Typ durch die Typangabe in der Variablendeklaration bestimmt wird. Eine Variable kann nur Werte des angegebenen Typs annehmen.

Jede Verwendung dieses Bezeichners ist ein Variablen-Name, der für diese Variable steht.
Ein Variablen-Name kann auch durch Voranstellung eines Paket-Bezeichners gebildet werden und bezieht sich dann auf eine Variable, die in der Spezifikation des bezeichneten Pakets deklariert ist.

Werte von Variablen eines einfachen Typs oder eines Zeigertyps sind nicht weiter zerlegbar. Werte von Variablen eines strukturierten Typs setzen sich aus den Werten ihrer Komponenten zusammen.

*Beispiel für einen Variablendeklarationsteil*

```
VAR
   a, b, c : Char;
   x, y    : Integer;
   i       : -10..+10;
   bool    : Boolean;
   feld1   : ARRAY [1..10, 1..100] OF Real;
   p1, p2  : person;   { person wurde in 6.3.3 definiert }
```

*Querverweise*

| | |
|---|---|
| Bezeichner: | 3.3 |
| Typangabe: | 6.1 |
| einfacher Typ: | 6.2 |
| strukturierter Typ: | 6.3 |
| Zeigertyp: | 6.4 |
| Variablendeklarationsteil: | 11.1 |
| Pakete: | 11.2 |
| Sichtbarkeitsregeln: | 12 |

# Einteilung von Variablen

Variable werden deklariert (statische Variable), dynamisch erzeugt (dynamische Variable) oder implizit deklariert (Puffervariable).

### Deklarierte Variable

Deklarierte Variable werden in einer Variablendeklaration deklariert und können über ihren Bezeichner angesprochen werden. Sie existieren während der Ausführung des Blocks (Programm, Prozedur oder Funktion), in dem sie deklariert wurden (siehe hierzu 13.3). Variablen, die unmittelbar in einer Paket-Spezifikation, einer Paket-Implementierung oder einem Hauptprogramm deklariert sind, existieren während der gesamten Ausführung des Programms.

### Dynamische Variable

Dynamische Variable werden nicht in einer Variablendeklaration deklariert und können auch nicht direkt über einen Bezeichner angesprochen werden. Sie werden vielmehr während der Ausführung eines Programms durch die vordefinierte Prozedur New (siehe 15.2) erzeugt. Sie existieren so lange, wie es Verweiswerte auf sie gibt, längstens bis zum Programmende. Verweiswerte können auch durch die vordefinierte Prozedur Dispose (siehe 15.2) vernichtet werden. Der Zugriff auf die dynamische Variable erfolgt durch Dereferenzierung eines Objekts (Variable, Konstante, Funktionsergebnis) eines Zeigertyps.

### Puffervariable

Mit der Deklaration einer Variablen f eines FILE-Typs, kurz FILE-Variable genannt, wird implizit eine Puffer-Variable definiert, die durch f↑ angesprochen wird. Der Typ dieser Variablen ist der Komponententyp des FILE-Typs. Bei Textdateien ist der Komponententyp der vordefinierte Typ Char. Die Puffervariable wird für die Zugriffe auf die Komponenten einer FILE-Variablen benötigt.

*Querverweise*

| | |
|---|---|
| Textdatei: | 6.3.5.2 |
| FILE-Typ: | 6.3.5 |
| Zeigertyp: | 6.4 |
| Domänentyp: | 6.4 |
| Objekt: | 9.6 |
| Variablenzugriff: | 9.6 |
| Block: | 12.1 |
| Programmausführung: | 13.3 |
| Puffervariable: | 6.3.5, 15.1, 19 |

# Definierte und undefinierte Werte von Variablen

Variable müssen bei ihrer Verwendung in einem Ausdruck einen gültigen Wert ihres Typs besitzen. In den folgenden Punkten wird beschrieben, wann Variable definiert sind, d. h. einen definierten Wert haben.

Eine Variable ist unmittelbar nach ihrer Erzeugung vollständig undefiniert. Eine deklarierte Variable wird erzeugt, wenn der Block in dem sie unmittelbar definiert ist, ausgeführt wird und wieder vernichtet, wenn die Ausführung des Block beendet wird (siehe 12). Eine dynamische Variable wird durch die vordefinierte Prozedur New erzeugt.

- **Variable eines einfachen Typs**

Der Wert einer einfachen Variablen ist nur definiert, wenn der Variablen ein Wert zugewiesen wurde, ansonsten ist sie undefiniert. Einer Variablen kann mit der Zuweisung (siehe 10.1.2) ein Wert zugewiesen werden.

- **Variable eines strukturierten Typs**

Der Wert einer strukturierten Variablen ist entweder definiert (dann besitzen alle Komponenten einen gültigen Wert) oder vollständig undefiniert (dann besitzt noch keine einzige Komponente derselben einen gültigen Wert) oder undefiniert (dann besitzen nicht alle Komponenten einen gültigen Wert). Man kann Komponenten einer undefinierten strukturierten Variablen in Ausdrücken verwenden, sofern diese Komponenten einen definierten Wert haben. Die gesamte strukturierte Variable kann erst dann in Ausdrücken verwendet werden, wenn alle Komponenten einen definierten Wert haben.

- **Variable eines Zeigertyps**

Eine Zeiger-Variable ist definiert, wenn sie einen gültigen Verweiswert oder den Wert NIL besitzt. Ein gültiger Verweiswert kann einer Zeigervariablen durch Aufruf der vordefinierten Prozedur New (siehe 15.2) oder durch Zuweisung (10.1.2) zugewiesen werden. Verweiswerte werden durch Aufruf der vordefinierten Prozeduren Dispose und Release (15.2) vernichtet.

Nach Aufruf von New besitzt die angegebene Zeiger-Variable einen gültigen Wert, nämlich einen Verweis auf die erzeugte dynamische Variable. Die erzeugte dynamische Variable ist hingegen vollständig undefiniert.

- **Variable von einem FILE-Typ**

Eine FILE-Variable hat einen definierten Wert, wenn eine der folgenden Aussagen gilt:

1) Sie wurde durch die vordefinierte Prozedur Rewrite definiert.
2) Der Variablen ist eine existierende Datei außerhalb des Programms zugeordnet. Diese Zuordnung kann durch die vordefinierte Prozedur Assignfile (siehe 15.1) oder durch Angabe als Programmparameter (siehe 11.5) hergestellt werden.

Eine Variable eines FILE-Typs ist folglich nicht definiert, wenn sie nicht mit Rewrite definiert wurde oder wenn keine Zuordnung an eine existierende Datei außerhalb des Programms existiert.

*Hinweise*

– Nicht initialisierte Variable sind eine häufige Fehlerquelle, die sich meistens erst zu einem späteren Zeitpunkt während der Programmausführung durch zufällige Folgefehler bemerkbar machen (z. B. Überschreiben von Code und Daten). Die Wirkung der Ausführung solcher fehlerhafter Programme ist undefiniert und kann sich abhängig von anderen Faktoren (Ladeadresse, Bindereihenfolge etc) ändern.

– Die Compiler-Option Initialize zusammen mit der Check-Option hilft in einigen Fällen die Verwendung nicht initialisierter Variable zu erkennen. Beim Erzeugen der Variablen (siehe oben) werden die Speicherbereiche für die Variablen mit einem bestimmten "Bitmuster" vorbesetzt. Stellt dieses Bitmuster einen ungültigen Wert der Variablen dar, dann tritt in vielen Fällen bei der Verwendung ein Laufzeitfehler auf.
  Die Initialize-Option bewirkt aber nicht die Belegung von Variablen mit gültigen Anfangswerten (siehe 16).

*Querverweise*

| | |
|---|---|
| Laufzeitfehler: | 2.3, A.5 |
| Strukturierte Typen: | 6.3 |
| einfacher Typ: | 6.2 |
| Komponentenvariable: | 6.3 |
| Programmparameter: | 11.5 |
| Assignfile: | 15.1 |
| Rewrite: | 15.1 |
| New, Dispose: | 15.2 |
| Puffervariable: | 7.2, 19 |
| Compiler-Optionen: | 16 |
| dynamische Variable: | 7.2, 20 |

# Prozeduren und Funktionen

Prozeduren und Funktionen werden im folgenden auch zusammenfassend als Unterprogramme bezeichnet.

Prozeduren dienen der übersichtlichen Gliederung eines Programms in Teilschritte (Prinzip der "schrittweiten Verkleinerung"). Der Aufruf einer Prozedur, d.h. die Ausführung der in ihr enthaltenen Anweisungsfolge, geschieht durch einen Prozeduraufruf (siehe 10.1.3).

Funktionen werden zur übersichtlichen Gestaltung von Ausdrücken benutzt, indem komplizierte oder umfangreiche Zwischenschritte als Funktionen geschrieben werden. Eine Funktion hat im Unterschied zu einer Prozedur die Eigenschaft, innerhalb eines Ausdrucks einen Wert zu repräsentieren. Dieser Wert wird durch eine Zuweisung an die Ergebnisvariable im Anweisungsteil der Funktion gebildet. Der Bezeichner dieser Ergebnisvariablen ist mit dem der Funktion identisch.
Ein Funktionsaufruf ist immer ein Ausdruck oder der Teil eines Ausdrucks (siehe Kapitel 9).

Unterprogramme können auch eigene Deklarationen besitzen. Die darin definierten Bezeichner sind "lokal", d. h. sie sind nur innerhalb des Unterprogramm-Blocks gültig, in dem sie vereinbart wurden, unabhängig davon, ob außerhalb (global) bereits gleichnamige Bezeichner existieren, die dort evtl. eine ganz andere Bedeutung besitzen. Unterprogramme können auch selbst wieder Unterprogramm-Deklarationen beinhalten, d. h. Unterprogramme können beliebig geschachtelt werden.

Unterprogramme können mehrfach aktiviert werden. Dies kann von außerhalb der eigenen Anweisungsfolge geschehen oder auch aus der eigenen heraus (rekursiv). Rekursionen sind dadurch Grenzen gesetzt, daß für jeden Aufruf neuer Speicherplatz benötigt wird, so daß der zur Verfügung stehende Speicher evtl. nicht ausreicht.

Die folgende (endlose) Rekursion hätte in jedem Fall einen Speicherüberlauf zur Folge (bitte deshalb nicht ausprobieren):

```
FUNCTION again_and_again (i:Integer) : Integer;
BEGIN
   again_and_again := again_and_again (i)
END
```

Unterprogramme können auf folgende Objekte zugreifen:

- eigene Parameter
- lokale Variablen und Konstanten (Definitionspunkt liegt im eigenen Deklarationsteil)
- globale Variablen, Konstanten und Parameter (Definitionspunkt liegt außerhalb des eigenen Deklarationsteils).

Über Parameter können an Unterprogramme Werte übergeben werden oder an den aufrufenden Teil zurückgeliefert werden.

Es ist sehr zu empfehlen, die Definitionspunkte der verwendeten Variablen so "dicht" wie möglich an die "Orte" ihrer Bearbeitung zu legen, d. h. möglichst wenig globale Variablen zu verwenden. Selbstverständlich wird man auf letztere nie ganz verzichten können, jedoch sollte man sich bewußt machen, daß Fehler, Seiteneffekte etc. durch globale Daten eher begünstigt werden. Nicht zuletzt zwingen sie den Compiler häufig, für den Zugriff auf diese Daten recht aufwendigen und entsprechend langsameren Code zu generieren.

Neben den vom Benutzer vereinbarten Unterprogrammen gibt es noch eine Reihe vordefinierter Unterprogramme, die teils durch die Norm vorgeschrieben, teils Erweiterungen von Pascal-XT sind. Sie sind im Kapitel 15 ausführlich beschrieben.

*Querverweise*


| | |
|---|---|
| Fehler: | 2.3, A.5 |
| Seiteneffekte: | 8.2 |
| Ausdruck: | 9 |
| Definitionspunkt: | 12.2 |
| Vordefinierte Unterprogramme: | 15 |

# Prozedurdeklaration

Prozedurdeklarationen haben folgende Syntax:

```
Prozedurdeklaration
                 = Prozedurkopf ";" Direktive ";"
                 | Prozedurkopf ";" Prozedur-Block ";"
                 | Prozeduridentifikation ";" Prozedur-Block ";"
                 | INLINE-Prozedurdeklaration.

Prozedurkopf     = "PROCEDURE" Bezeichner [Formalparameterliste].

Prozeduridentifikation
                 = "PROCEDURE" Prozedur-Bezeichner
                   [Formalparameterliste].

INLINE-Prozedurdeklaration
                 = "INLINE" Prozedurkopf ";" Prozedur-Block ";".

Block            = {  Markendeklarationsteil
                    | Konstantendefinitionsteil
                    | Typdefinitionsteil
                    | Variablendeklarationsteil
                    | Prozedurdeklaration
                    | Funktionsdeklaration
                   }
                   Anweisungsteil.

Anweisungsteil   = Verbundanweisung.

Prozedur-Name    = [ Paket-Bezeichner "." ] Bezeichner.
```

Eine Prozedurdeklaration führt einen Bezeichner ein, der mit dem Prozedur-Block verknüpft ist. Jede Verwendung dieses Bezeichners ist ein Prozedur-Name und steht, außer bei der Prozeduridentifikation, für die Ausführung der Anweisungen des Prozedur-Blocks (siehe 8.7).
Ein Prozedur-Name kann auch durch Voranstellen eines Paket-Bezeichners gebildet werden und bezieht sich dann auf eine Prozedur-Deklaration in der bezeichneten Paketspezifikation.

"Block" steht für einen Vereinbarungsteil, dem ein Anweisungsteil folgt. Ein Anweisungsteil ist eine Verbundanweisung.

"Formalparameterliste" ist eine in Klammern eingeschlossene Liste von formalen Parametern (8.5), die als Platzhalter für die aktuellen Parameter beim Prozeduraufruf dienen.

In Pascal-XT darf als Erweiterung zur Norm in einer Prozeduridentifikation die Formalparameterliste der zugehörigen Prozedurdeklaration wiederholt werden.
Die Parameterlisten müssen dann aber textlich übereinstimmen.

INLINE-Prozeduren sind in Abschnitt 8.3, Direktiven und Prozeduridentifikationen sind in Abschnitt 8.6 beschrieben.

*Beispiel für eine Prozedurdeklaration*

```
PROCEDURE vertausche (VAR x, y: Integer);

VAR
   hilf : Integer;

BEGIN
   hilf := x;
   x := y;
   y := hilf;
END {vertausche};
```

*Querverweise*

| | |
|---|---|
| Markendeklarationsteil: | 4 |
| Konstantendefinitionsteil: | 5 |
| Typdefinitionsteil: | 6 |
| Variablendeklarationsteil: | 7 |
| INLINE-Unterprogramm | 8.3 |
| Parameter: | 8.5 |
| Direktive und Prozeduridentifikation: | 8.6 |
| Verbundanweisung: | 10.2 |
| Block: | 12.1 |

# Funktionsdeklaration

Funktionen repräsentieren einen Wert, das Funktionsergebnis. Sie können daher in Ausdrücken verwendet werden. Der Typ des Ergebnisses muß bei einer Funktionsdeklaration nach dem Funktionskopf als Typ-Name angegeben werden.

```
Funktionsdeklaration
                = Funktionskopf ";" Direktive ";"
                | Funktionskopf ";" Funktions-Block ";"
                | Funktionsidentifikation ";" Funktions-Block ";"
                | INLINE-Funktionsdeklaration.

Funktionskopf   = "FUNCTION" Bezeichner
                  [Formalparameterliste] ":" Ergebnistyp.

Ergebnistyp     = Typ-Name.

Funktionsidentifikation
                = "FUNCTION" Funktions-Bezeichner
                  [[Formalparameterliste] ":" Ergebnistyp].

INLINE-Funktionsdeklaration
                = "INLINE" Funktionskopf ";" Funktions-Block ";".
Funktions-Block = Block .

Funktions-Name  = [Paket-Bezeichner "." ] Bezeichner
```

Eine Funktionsdeklaration führt einen Bezeichner ein, der mit dem Funktions-Block verknüpft ist. Jede Verwendung dieses Bezeichners ist ein Funktions-Name und steht, außer bei der Funktionsidentifikation, für den Wert, der während der Ausführung der Anweisungen des Funktions-Blocks zugewiesen wird (siehe 8.7).
Ein Funktions-Name kann auch durch Voranstellen eines Paket-Bezeichners gebildet werden und bezieht sich dann auf eine Funktionsdeklaration in der bezeichneten Paketspezifikation.

Funktionen müssen eine Zuweisung an den Funktions-Bezeichner enthalten, der in diesem Moment als Platzhalter für den Funktionswert dient. Diese Zuweisung kann im Anweisungsteil der Funktion oder in einem innerhalb der Funktion deklarierten Unterprogramm stehen. Beim Abarbeiten des Funktions-Blocks der Funktion muß mindestens eine solche Zuweisung durchlaufen werden. Auf diese Weise wird das Ergebnis der Funktion bestimmt. Werden mehrerer solcher Zuweisungen durchlaufen, dann wird das Funktionsergebnis durch die dynamisch zuletzt abgearbeitete Zuweisung bestimmt.

In Norm-Pascal sind als Ergebnistyp nur einfache Typen und Zeigertypen zugelassen.

In Pascal-XT darf der Ergebnistyp ein beliebiger Typ sein, der kein FILE-Typ ist und auch keinen FILE-Typ (direkt oder indirekt) als Komponententyp besitzt. Bei einem strukturierten Ergebnistyp muß dem Funktions-Bezeichner als Ganzes ein Wert zugewiesen werden, eine komponentenweise Zuweisung ist nur mit Aggregaten möglich (siehe Beispiel 2).

"Formalparameterliste" ist eine in Klammern eingeschlossene Liste von formalen Parametern (8.5), die als Platzhalter für die aktuellen Parameter beim Funktionsaufruf dienen.

In Pascal-XT dürfen als Erweiterung zur Norm, in einer Funktionsidentifikation die Formalparameterliste und der Ergebnistyp der zugehörigen Funktionsdeklaration wiederholt werden. Sie müssen dann aber textlich übereinstimmen.

INLINE-Funktionen sind in Abschnitt 8.3, Direktiven und Funktionsidentifikation sind in Abschnitt 8.6 beschrieben.

*Hinweis*

Die Aufgabe einer Funktion ist die Berechnung eines Wertes. Es trägt zur Übersichtlichkeit, Verständlichkeit von Programmen bei und erleichtert die Verfassung korrekter Programme, wenn das Funktionsergebnis nur aus den Werten der Parameter berechnet wird. Die Einbeziehung der Werte globaler Variablen ist nicht empfehlenswert. Noch gefährlicher und verwirrender sind Funktionen mit Seiteneffekten, die also nach der Berechnung des Funktionsergebnisses auch globale oder dynamische Variablen verändern.

*Beispiel 1*

Die Funktion max liefert den größeren Wert der beiden Parameter zurück.

```
FUNCTION max (a, b: Integer): Integer;
BEGIN
  IF a <= b THEN
     max := b
  ELSE
     max := a
END { max };
```

*Beispiel 2*

Die beiden Funktionen add und sub liefern Werte eines RECORD-Typs (strukturierten Typs) zurück. In der Funktion add wird zur Berechnung des Ergebnisses die Hilfsvariable temp benutzt, in der Funktion sub wird ein Aggregat (9.5) verwendet, da komponentenweise Zuweisungen der Form add.re := x.re + y.re nicht zulässig ist.

```
TYPE
   complex = RECORD
                re,
                im: real
             END;

FUNCTION add (x, y: complex): complex;
VAR
   temp: complex;
BEGIN
   temp.re := x.re + y.re;
   temp.im := x.im + y.im;
   add     := temp;
END { add };


FUNCTION sub (x, y: complex): complex;
BEGIN
   sub := complex (x.re - y.re, x.im - y.im);
END { sub };
```

*Beispiel 3*

Die Funktion ersetze liefert eine Zeichenkette zurück. Das erste Auftreten der Zeichenkette alt in orginal soll durch neu ersetzt werden. Ist die Zeichenkette nicht enthalten, dann wird die ursprüngliche Zeichenkette zurückgegeben.

```
FUNCTION ersetze (orginal, alt, neu: String): String;
VAR
   i: Short_Integer;
BEGIN
   i := Position (alt, orginal);
   IF i > 0 THEN
      ersetze :=
         Concat
            (Substring (orginal, 1, i-1),
             neu,
             Substring
                (orginal,
                 i + Length (alt),
                 Length (orginal) - Length (alt) - i + 1))
   ELSE
      ersetze := orginal;
END { ersetze };
```

*Querverweise*


| | |
|---|---|
| einfacher Typ: | 6.2 |
| FILE-Typ: | 6.3.5 |
| globale Variable: | 7 |
| dynamische Variable: | 7 |
| INLINE-Unterprogramme: | 8.3 |
| Parameter: | 8.5 |
| Aggregate: | 9.5 |
| Zuweisung: | 10.1.2 |
| Block: | 12.1 |
| Stringfunktionen: | 15.3 |

# Inline-Unterprogramme

Ein Unterprogramm kann durch Angabe des Schlüsselworts INLINE vor PROCEDURE oder FUNCTION als Inline-Unterprogramm deklariert werden. In diesem Fall darf der Unterprogramm-Block nicht durch eine Direktive ersetzt werden (siehe 8.6). An jeder Aufrufstelle eines INLINE-Unterprogramms wird der Aufruf durch eine Kopie des Unterprogrammblocks ersetzt. Dabei werden die formalen Parameter so durch die aktuellen ersetzt, daß die Wirkung der Ausführung die gleiche ist wie bei einem "echten" Unterprogrammaufruf. Diese sog. Inline-Expandierung dient zur Laufzeit-Optimierung von Unterprogrammaufrufen, was zu effizienterem, aber möglicherweise zu längerem Code führt.

Innerhalb eines INLINE-Unterprogramms dürfen keine Nicht-INLINE-Unterprogramme deklariert werden und der Block eines INLINE-Unterprogramms darf keinen rekursiven Aufruf des eigenen Unterprogramms enthalten.

In einem Inline-Unterprogramm dürfen keine Variablen eines FILE-Typs deklariert werden.

Auf die Besonderheiten von INLINE-Unterprogrammen in Paket-Spezifikationen wird in Abschnitt 11.2 eingegangen.

*Beispiel*

```
INLINE FUNCTION max (a, b: Integer): Integer;
BEGIN
  IF a <= b THEN
     max := b
  ELSE
     max := a
END { max };
```

*Querverweise*

| | |
|---|---|
| Direktiven: | 8.6 |
| Unterprogrammaufruf: | 8.7 |
| Block: | 12.1 |
| Paket-Spezifikation: | 11.2 |

# Entry-Unterprogramme

In Paket-Spezifikationen können Prozeduren und Funktionen durch Angabe des Schlüsselworts ENTRY vor PROCEDURE oder FUNCTION als Entry-Unterprogramm deklariert werden. Die so deklarierten Unterprogramme können auch von Programmteilen aufgerufen werden, die nicht in Pascal-XT geschrieben sind. Dabei sind ggf. implementierungsdefinierte Schnittstellen einzuhalten.

In Abschnitt 11.2 sind ENTRY-Unterprogramme genauer beschrieben.

*Querverweise*

Paket-Spezifikation: 11.2

# Parameter

Unterprogramme kommunizieren mit ihrer Umgebung über Parameter und globale (außerhalb des Unterprogramms vereinbarte) Variablen und Konstanten. Parameter müssen in der Formalparameterliste der Unterprogrammdeklaration vereinbart sein. Sie werden beim Aufruf durch aktuelle Parameter ersetzt.

Es gibt fünf Arten von Formalparametern:

- Wertparameter (Value-Parameter)
- Variablenparameter (Var-Parameter)
- Parameterprozeduren
- Parameterfunktionen
- Konformreihungs-Parameter

Die Formalparameterlisten haben folgende Syntax:

```
Formalparameterliste
                = "(" Formalparameterabschnitt
                      {";" Formalparameterabschnitt} ")".

Formalparameterabschnitt
                = Wertparameter-Spezifikation
                | Variablenparameter-Spezifikation
                | Parameterprozedur-Spezifikation
                | Parameterfunktions-Spezifikation
                | Konformreihungs-Parameter-Spezifikation.
```

Eine Formalparameterliste besteht aus einzelnen, jeweils durch ein Semikolon getrennten Formalparameterabschnitten. In jedem Formalparameterabschnitt steht entweder genau eine Parameterprozedur bzw. -funktion (siehe 8.5.3), oder es werden ein oder mehrere Werte- bzw. Variablen-Parameter desselben Typs oder Konformreihungs-Schemata deklariert.

Die Konformreihungen genügen der Erfüllungsstufe 1 der Norm (siehe 2.2) und werden im Abschnitt 8.5.4 behandelt.

*Hinweis*

In Pascal-XT darf die Formalparameterliste in der Identifikation eines Unterprogramms wiederholt werden. Sie muß aber textlich identisch sein mit der in der Deklaration. Bei Funktionsidentifikationen müssen Parameterliste, sofern vorhanden, und Funktionsergebnis gemeinsam wiederholt oder beide weggelassen werden.

### Wertparameter

Wertparameter werden definiert durch:

```
Wertparameter-Spezifikation
                = Bezeichnerliste ":" Typ-Name.
```

Wertparameter dienen zur Übergabe von Werten an Unterprogramme. Sie haben innerhalb eines Unterprogramms die Eigenschaften von lokalen Variablen, denen vor Eintritt in den Unterprogramm-Block die aktuellen Werte zugewiesen werden.

Es ist nicht möglich, über Wertparameter die Werte von Variablen, die als Aktualparameter übergeben wurden, durch das Unterprogramm zu verändern, d. h., man kann mit Wertparametern keinen Wert nach außen geben.

Beim Aufruf des Unterprogramms muß für jeden formalen Wertparameter ein Ausdruck (siehe 9) angegeben werden. Dies kann z. B. eine Variable, eine Konstante, ein Funktionsergebnis sein und kann Operationen (wie "+" oder "<") enthalten. Der Wert des Ausdrucks muß zuweisungsverträglich (siehe 6.6.3) mit dem Typ des zugehörigen Formalparameters sein. Bei der Übergabe von Wert-Parametern können während der Programmausführung die folgenden Laufzeitfehler auftreten.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Numeric_Error | - Bei Wertparameterübergabe liegt der Wert des Aktualparameters vom Typ Long_Real nicht im Wertebereich des Formalparameters vom Typ Short_Real. |
| Range_Error | - Bei Wertparameterübergabe liegt der Wert des Aktualparameters von einem Ordinal-Typ nicht im Wertebereich des Typs des Formalparameters. |
| Set_Error | - Bei Wertparameterübergabe liegt der Wert des Aktualparameters von einem Set-Typ nicht im Wertebereich des Typs des Formalparameters. |
| String_Error | - Bei Wertparameterübergabe ist die aktuelle Länge der Zeichenkette des Aktualparameters größer als die Maximallänge des String-Typs des Formalparameters. |
| | - Bei Wertparameterübergabe ist die aktuelle Länge der Zeichenkette des Aktualparameters ungleich der Länge des Zeichenkettentyps fester Länge des Formalparameters. |
| undefinierte Auswirkungen | - Bei Wertparameterübergabe ist der Typ des Aktualparameters vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamische Variable, deren Typ vom Domänentyp des Typs des Formalparameters verschieden ist. |

Weitere Laufzeitfehler können bei der Auswertung des als Aktualparameter erhaltenen Ausdrucks auftreten (siehe 9).

*Beispiel für eine Prozedur mit Wertparametern und ihren Aufruf*

```
{ i, j, z seien "globale" Variable, f eine Funktion vom Typ Integer oder
eines Teilbereichs davon. }

   PROCEDURE summenausgabe (x, y: Integer; b: Boolean);
   BEGIN
      IF b THEN
         Writeln (x + y);
   END;

   BEGIN
     ...
     summenausgabe (i * j, f(z), True);
     ...
   END.
```

*Querverweise*

| | |
|---|---|
| Laufzeitfehler: | 2.3, 14, A.5 |
| Zuweisungsverträglichkeit: | 6.6.3 |
| Variable: | 9.6 |
| Ausdruck: | 9 |

## Variablenparameter

Variablenparameter werden definiert durch:

```
Variablenparameter-Spezifikation
                = "VAR" Bezeichnerliste ":" Typ-Name.
```

Ein Variablenparameter wird für die Zeit der Ausführung des Unterprogramms mit der (verallgemeinerten) Variablen identifiziert, die beim Aufruf als Aktualparameter angegeben wurde. Eine Operation auf dem Formalparameter wird damit auf der als Aktualparameter angegebenen (verallgemeinerten) Variablen ausgeführt.

Beim Aufruf des Unterprogramms muß für jeden formalen Variablen-Parameter ein Aktual-Parameter übergeben werden, der folgende Bedingungen erfüllt.

a) Der Aktualparameter muß ein Variablen-Objekt (einer verallgemeinerten Variablen) sein.

b) Der Aktualparameter muß denselben Typ (siehe 6.6.1) wie der zugehörige Formalparameter besitzen.
Ist aber der Typ eines Formalparameters der generische Zeigertyp (Pointer) bzw. der generische FILE-Typ (Any_File), dann darf der entsprechende Aktualparameter von einem beliebigen Zeigertyp bzw. FILE-Typ sein. Ist der Typ des Formatparameters der generische Typ Any_Type, dann darf der entsprechende Aktualparameter von einem beliebigen Typ sein.

c) Der Aktualparameter darf kein Kennungsfeld eines Variantteils eines RECORD-Typs sein (siehe 6.3.3).

d) Der Aktualparameter darf keine Komponente einer Variablen eines gepackten Typs (siehe 6.3) sein.

*Hinweise*

- Auch String-Typen gelten als gepackte Typen. Demnach dürfen auch die Komponenten von String-Variablen nicht als aktuelle Variablenparameter übergeben werden.
- In Variablenparameter-Spezifikation muß ein Typ-Name angegeben sein. Der vordefinierte Typ-Bezeichner String gefolgt von einer Längenangabe (z.B. String [4]) ist kein Typ-Name sondern eine Typangabe.
- Wenn der Aktualparameter ein dereferenziertes oder ein indiziertes Objekt ist (siehe 9.6), so wird die Dereferenzierung bzw. Indizierung bereits beim Aufruf des Unterprogramms durchgeführt, also nicht erst während der Ausführung des Anweisungsteils des Unterprogrammblocks. Wenn der Wert des Zeigers bzw. eines Index-Ausdrucks während der Ausführung des Unterprogramms verändert wird, so ändert sich dadurch nichts an der Bindung des Formalparameters an die beim Aufruf des Unterprogramms ermittelte verallgemeinerte Variable.

*Beispiel einer Prozedur mit Variablenparametern und deren Aufruf*

```
VAR
   i, j: Integer;

PROCEDURE tausch (VAR x, y : Integer);
VAR
   hilf : Integer;
BEGIN
   hilf := x;
   x    := y;
   y    := hilf;
END;

BEGIN
   i := 5;
   j := 8;
   tausch (i, j);
   Writeln(i);                { ergibt jetzt 8 }
   Writeln(j);                { ergibt jetzt 5 }
END {tausch}
```

*Beispiele unzulässiger Aktualparameter*

```
TYPE
   string4 = String [4];
VAR
   a : String [4];

PROCEDURE parametertest (VAR par : string4);

   BEGIN ... END;

BEGIN
  a := '123';
  parametertest (a);        { verschiedene Typen }
  parametertest ('abcd');   { keine Variable }
END {parametertest}
```

Beide Aufrufe von Parametertest sind semantisch fehlerhaft. Die Variable a hat nicht denselben Typ (siehe 6.6.1) wie der Var-Parameter par und 'abcd' ist eine Konstante und keine verallgemeinerte Variable.

*Querverweise*

| | |
|---|---|
| Laufzeitfehler: | 2.3, 14, A.5 |
| Gepackte Typen: | 6.3 |
| String-Typen | 6.3.2.2 |
| Kennungsfeld: | 6.3.3 |
| Generischer FILE-Typ: | 6.5.1 |
| Generischer Zeigertyp: | 6.5.2 |
| Typenidentität: | 6.6.1 |
| Objekte: | 7 |
| Variable: | 7.1, 9.6 |
| Ausdruck: | 9 |

## Parameterprozeduren und -funktionen

Unterprogramme als Parameter werden definiert durch:

```
Parameterfunktions-Spezifikation = Funktionskopf.

Parameterprozedur-Spezifikation  = Prozedurkopf.
```

Parameterprozeduren bzw. -funktionen werden in Form eines kompletten Prozedur- bzw. Funktionskopfes als Formalparameter in der Formalparameterliste einer anderen Prozedur oder Funktion angegeben.

Der entsprechende Aktualparameter beim Aufruf muß ein Prozedur-Name oder ein Funktions-Name sein und folgende Bedingungen erfüllen:

a) Er darf kein vordefiniertes Unterprogramm bezeichnen.

b) Er darf kein Inline-Unterprogramm bezeichnen.

c) Er darf kein Unterprogramm mit einer der Direktiven c, cobol, fortran, internal oder external bezeichnen.

d) Falls der Formalparameter selbst wieder eine Formalparameterliste enthält, so muß auch der Aktualparameter eine besitzten und beide müssen übereinstimmen (siehe unten).

e) Falls der Formalparameter eine Parameterfunktion ist, so muß auch der Aktualparameter eine Funktion bezeichnen, die denselben Ergebnistyp wie der Formalparameter besitzt.
Ist aber der Ergebnistyp des Formalparameters der generische Zeigertyp, so darf der Ergebnistyp des Aktualparameters ein beliebiger Zeigertyp sein.

### Übereinstimmung von Formalparameterlisten

Die Formalparameterliste eines Formalparameters und eines Aktualparameters stimmen überein, wenn Sie die gleiche Anzahl von Formalparameterabschnitten haben und für je zwei Abschnitte entsprechender Position eine der folgenden Aussagen gilt:

a) Sie sind beide Wertparameter-Spezifikationen oder beide Variablenparameter-Spezifikationen mit der gleichen Anzahl von Parametern und demselben Typ.
Ist aber der Typ in der Formalparameterliste des Formalparameters der generische Zeigertyp (Pointer), dann kann der Typ in der Formalparameterliste des Aktualparameters ein beliebiger Zeigertyp sein.

b) Sie sind beide Parameterprozedur-Spezifikationen mit übereinstimmenden Formalparameterlisten.

c) Sie sind beide Parameterfunktions-Spezifikationen mit übereinstimmenden Formalparameterlisten und demselben Ergebnistyp.
Ist der Ergebnistyp in der Formalparameterliste des Formalparameters der generische Zeigertyp (Pointer), dann kann der Ergebnistyp in der Formalparameterliste des Aktualparameters ein beliebiger Zeigertyp sein.

d) Sie sind beide Wert- oder beide Variablen-Konformreihungsspezifikationen (8.5.4) mit derselben Anzahl von Parametern und äquivalenten Konformreihungs-Schemata.
Äquivalent bedeutet, daß jede der folgenden Aussagen gilt:

1) Es gibt gleich viele Indextypspezifikationen in beiden Schemata.

2) Der Ordinaltyp-Name muß in beiden Indextypspezifikationen denselben Typ repräsentieren.

3) Die Komponenten beider Schemata müssen denselben Typ haben oder wiederum äquivalente Schemata sein.

4) Beide Schemata müssen gepackt oder beide müssen ungepackt sein.

*Beispiel eines Programms mit Parameterprozeduren*

```
PROGRAM rechner (Input, Output);

VAR
   a, b, c: Integer;

PROCEDURE addition
   (    summand_1, summand_2 : Integer;
    VAR summe            : Integer);
BEGIN
   summe := summand_1 + summand_2;
END {addition};

PROCEDURE division
   (    dividend, divisor : Integer;
    VAR quotient : Integer);
BEGIN
   quotient := dividend DIV divisor;
END {division};
...
{ Hier könnten weitere Operationen vereinbart sein }
...
```

```
PROCEDURE berechnen_und_ausgeben
   (linker_operand, rechter_operand: Integer;
    PROCEDURE operation (x, y: Integer; VAR k: Integer));
VAR
   ergebnis : Integer;
BEGIN
   operation (linker_operand, rechter_operand, ergebnis);
   writeln (ergebnis);
END {berechnen_und_ausgeben};

BEGIN {Hauptprogramm}
   berechnen_und_ausgeben (4711, 15, addition);
   berechnen_und_ausgeben (2048, 32, division);
END {rechner}.
```

*Hinweis*

Beim Aufruf von "berechnen_und_ausgeben" wird der Name des zu übergebenden Unterprogramms (addition bzw. division) als Aktualparameter übergeben.
Die Parameter-Namen der Parameterprozedur Operation (x, y, z) werden nur in deren Parameterliste gebraucht.

*Querverweise*

generischer Zeigertyp: 6.5.2
Typidentität: 6.6.1
Prozedurkopf: 8.1
Funktionskopf: 8.2
INLINE-Unterprogramme: 8.3
Direktiven: 8.6
Vordefinierte Unterprogramme: 15

## Konformreihungs-Parameter

Konformreihungs-Parameter stellen die erweiterte Normanforderung (Erfüllungsstufe 1) dar. Sie bieten die Möglichkeit, innerhalb von Unterprogrammen Array-Parameter mit unterschiedlichen Größen zu bearbeiten.

Die Syntax für Konformreihungsparameter lautet:

```
Konformreihungs-Parameter-Spezifikation
                = Wert-Konformreihungs-Spezifikation
                | Variablen-Konformreihungs-Spezifikation.

Variablen-Konformreihungs-Spezifikation
                = "VAR" Bezeichnerliste ":" Konformreihungs-Schema.

Wert-Konformreihungs-Spezifikation
                = Bezeichnerliste ":" Konformreihungs-Schema.

Konformreihungs-Schema
                = gepacktes_Konformreihungs-Schema
                | ungepacktes_Konformreihungs-Schema.

gepacktes_Konformreihungs-Schema
                = "PACKED" "ARRAY" "[" Indextyp-Spezifikation "]"
                  "OF" Typ-Name.

ungepacktes_Konformreihungs-Schema
                = "ARRAY" "[" Indextyp-Spezifikation {";"
                              Indextyp-Spezifikation} "]" "OF"
                     ( Typ-Name | Konformreihungs-Schema ).

Indextyp-Spezifikation
                = Bezeichner ".." Bezeichner ":" Ordinaltyp-Name.
```

Konformreihungs-Parameter sind formale Parameter von ARRAY-Typen, bei denen innerhalb des Konformreihungs-Schemas ein umfassender Indextyp und ein fester Komponententyp angegeben sind. Eine Indextyp-Spezifikation definiert Bezeichner, die innerhalb des Unterprogramms als Grenz-Bezeichner verwendet werden können (siehe Beispiel unten).

Beim Aufruf eines Unterprogramms mit Konformreihungs-Parametern kann der jeweilige Indextyp des Aktualparameters ein beliebiger Teilbereichstyp des umfassenden Indextyps der entsprechenden Konformreihungs-Parameter-Spezifikation sein. Der Wert der Grenz-Bezeichner ergibt sich aus dem kleinsten und größten Wert des Indextyps des Aktualparameters. Diese Festlegung wird als "Konformität" bezeichnet (im Gegensatz zur Zuweisungsverträglichkeit, die bei der Variablenparameterübergabe gefordert wird).

Bezüglich der Unterscheidung zwischen Wert- und Variablenparametern gilt auch das unter 8.5.1 und 8.5.2 gesagte.

### Abkürzung bei mehrdimensionalen Konformreihungs-Schemata

Wenn ein Konformreihungs-Schema ein weiteres unmittelbar enthält, dann kann eine Kurzform zur Darstellung gewählt werden. Diese Kurzform ersetzt die Zeichenfolge "] OF ARRAY [" der Langform durch ein Semikolon. Die Kurzform und die Langform sind äquivalent.

*Beispiel*

```
  ARRAY [u..v : t1] OF ARRAY [j..k : t2] OF t3
```

und

```
  ARRAY [u..v : t1;           j..k : t2] OF t3
```

sind äquivalente Konformreihungs-Schemata.

### Aktualparameter

Beim Aufruf des Unterprogramms muß für jeden formalen Konformreihungs-Parameter als aktueller Parameter ein ARRAY-Objekt übergeben werden, das folgenden Bedingungen genügt:

a) Wenn der Formalparameter ein Variablen-Konformreihungs-Schema ist, muß der aktuelle Parameter ein Variablen-Objekt (verallgemeinerte Variable) sein.

b) Die Aktualparameter, die Formalparametern aus einer einzigen Konformreihungs-Parameter-Spezifikation zugeordnet werden sollen, müssen alle denselben Typ haben.

c) Der Typ der Aktualparameter muß konform zu dem Konformreihungs-Schema sein (siehe unten).

d) Bei Wertparameterübergabe darf ein Aktualparameter nicht wieder ein Konformreihungs-Parameter sein.

### Konformität von Konformreihungen

T1 sei ein ARRAY-Typ mit einem einzigen Indextyp und T2 der Ordinaltyp in der Indextyp-Spezifikation eines Konformreihungs-Schemas. T1 ist zu dem Konformreihungs-Schema konform, wenn alle folgenden Bedingungen erfüllt sind:

a) Der Indextyp von T1 ist verträglich zu T2.

b) Der kleinste und der größte Wert des Indextyps von T1 liegen innerhalb des Wertebereichs des Typs T2.

c) Der Komponententyp von T1 repräsentiert denselben Typ wie der Typ-Bezeichner in dem Konformreihungs-Schema oder ist konform zu dem Konformreihungs-Schema im Konformreihungs-Schema.

d) Entweder sind T1 und das Konformreihungs-Schema ungepackt, oder T1 und das Konformreihungs-Schema sind gepackt.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Index_Error | - Bei einem Konformreihungsparameter ist der Indextyp des Aktualparameters nicht ein Teilbereich des umfassenden Indextyps des Konformreihungsschemas. |

*Beispiel*

```
PROGRAM add_vektor (Input, Output);

TYPE
   bereich = -1000..1000;

VAR
   feld_1, feld_11, feld_111: ARRAY [    1.. 100] OF Integer;
   feld_2, feld_22, feld_222: ARRAY [  -10..  10] OF Integer;
   feld_3, feld_33, feld_333: ARRAY [-1000..1000] OF Integer;

PROCEDURE add_felder (VAR v1,v2,v3 : ARRAY [ug..og : bereich] OF Integer);

VAR
   i : bereich;

BEGIN
   FOR i := ug TO og DO
      v3[i] := v1[i] + v2[i];
END {add_felder};

BEGIN {Hauptprogramm}
   ... {Einlesen der Felder}
   add_felder (feld_1, feld_11, feld_111);   {ug ist 1, og ist 100 }
   add_felder (feld_2, feld_22, feld_222);   {ug ist -10, og ist 10}
   add_felder (feld_3, feld_33, feld_333);   {ug ist -1000, og ist 1000}
   ... {Ausgabe der Felder}
END {add_vektor}.
```

Die beiden Bezeichner in der Indextyp-Spezifikation (im Beispiel "ug" und "og") heißen Grenz-Bezeichner. Die Objekte, für die sie stehen, sind weder Konstanten noch Variablen, sondern stellen eine eigene Klasse von Objekten dar, die semantisch fest an die Anwendung innerhalb eines Konformreihungs-Schemas gebunden sind. Es dürfen ihnen weder Werte zugewiesen werden, noch dürfen sie (als Konstante) in einer Typdefinition verwendet werden. Es darf lediglich lesend auf sie zugegriffen werden.

*Querverweise*

Gepackte Typen: 6.3
ARRAY-Typ: 6.3.1
Typidentität: 6.6.1

# Die Direktiven und Prozedur- bzw. Funktionsidentifikationen

Eine Direktive steht stellvertretend für den Block des Unterprogramms, der im selben Deklarationsteil weiter unten im Text folgen kann oder auch in einer fremden Sprache geschrieben ist und nicht zum Pascal-XT-Programmtext gehört.

Norm-Pascal schreibt als einzige Direktive forward vor.
Pascal-XT bietet weitere Direktiven zum Anschluß fremdsprachiger Unterprogramme an, die getrennt übersetzt und dann zum Pascal-XT-Programm hinzugebunden werden müssen.

Ist in einer Prozedur- bzw. Funktionsdeklaration eine von Forward verschiedene Direktive angegeben, dann darf es keine zugehörige Prozedur- bzw. Funktionsidentifikation geben.

| Direktive | Bedeutung |
|---|---|
| forward | Der zugehörige Unterprogramm-Block ist weiter hinten im Programm angegeben (siehe unten). |
| c | Das Unterprogramm ist in C geschrieben. |
| cobol | Das Unterprogramm ist in COBOL geschrieben. |
| fortran | Die Unterprogramm ist in FORTRAN geschrieben. |
| external | Das Unterprogramm ist nicht in Pascal-XT geschrieben und wird über eine systemspezifische Unterprogramm-Schnittstelle aufgerufen. |
| internal | Das Unterprogramm ist nicht in Pascal-XT geschrieben. Der Aufruf erfolgt, im Gegensatz zur Direktive External, nach dem internen Aufrufmechanismus von Pascal-XT-Unterprogrammen. Der Compiler erwartet, daß es sich bezüglich der Schnittstellen wie ein in Pascal geschriebenes Unterprogramm verhält. |

Tabelle 8-1: Direktiven in Pascal-XT

- **Direktive Forward**

Die Direktive Forward wird benötigt, um nur den Prozedurkopf bzw. Funktionskopf zu deklarieren. Damit kann der Prozedur-Bezeichner bzw. Funktions-Bezeichner verwendet werden, bevor der zugehörige Prozedur-Block bzw. Funktions-Block deklariert wird. Diese Direktive wird benötigt, um gegenseitig rekursive Unterprogrammaufrufe zu ermöglichen (siehe Beispiel).

Der Bezeichner einer mit der Direktive Forward versehenen Prozedurdeklaration bzw. Funktionsdeklaration muß genau einmal in einer Prozeduridentifikation bzw. Funktionsidentifikation vorkommen, die zum selben Deklarationsteil (12.1) gehört.

*Beispiel*

```
PROCEDURE p2 (k : Integer); Forward;

PROCEDURE p1 (j : Integer);
BEGIN
   Writeln (j);
   IF j>0 THEN p2 (j-1);
   . . .
END {p1};

PROCEDURE p2;
BEGIN
   Writeln (k);
   IF k>0 THEN p1 (k-1);
   . . .
END {p2};
```

Da sich die Prozeduren p1 und p2 gegenseitig aufrufen, ist es gemäß 12.1 notwendig, daß p1 vor p2 und p2 vor p1 deklariert wird. Durch die Direktive Forward ist es möglich, den Prozedurkopf von p2 schon vor der vollständigen Vereinbarung von p2 bekanntzugeben. Dadurch ist der von außen sichtbare Teil von p2 schon bekannt und kann nunmehr verwendet werden.
Der Block von p2 folgt dann erst nach der Deklaration von p1, wobei die nochmalige Angabe der Formalparameterliste (der Prozedur p2) entfallen kann und in Norm-Pascal entfallen muß. Die in Pascal-XT gestattete Wiederholung der Formalparameterliste ist jedoch aus Gründen der besseren Lesbarkeit zu empfehlen, weil i. a. der Block weit hinter der FORWARD-Deklaration folgt, und man sich so umständliches Blättern im Compiler-Protokoll bzw. am Bildschirm ersparen kann. Sofern normgemäß programmiert werden soll, kann man die Formalparameterliste Kommentarklammern setzen.

*Hinweis*

Zwischen der FORWARD-Deklaration und der zugehörigen Identifikation können wiederum beliebige andere Deklarationen stehen (LABEL-, CONST-, TYPE, VAR-, PROCEDURE- und FUNCTION-Deklarationen). Dies entspricht der im Kapitel 12 angegebenen Norm-Erweiterung, das die einzelnen Deklarationen in beliebiger Reihenfolge auftreten dürfen.

- **Sonstige Direktiven**

Wird in einer Prozedur- bzw. Funktionsdeklaration eine der Direktiven C, Cobol, Fortran, External oder Internal angegeben, dann darf es keine zugehörige Prozedur- bzw. Funktionsidentifikation geben.

Für solche externen Unterprogramme kann es eine Reihe implementierungsdefinierter Einschränkungen geben.

### Implementierungsdefinierte Eigenschaften

- Eine Pascal-XT Implementierung muß, außer forward, nicht alle Direktiven unterstützen.
- Bei Unterprogrammen mit einer der Direktiven c, cobol, fortran, external und internal kann es bzgl. der Parameterart, den Parametertypen und der Parameteranzahl implementierungsdefinierte Einschränkungen geben.

*Querverweise*


| | |
|---|---|
| Direktiven: | 3.4 |
| Prozeduridentifikation: | 8.1 |
| Funktionsidentifikation: | 8.2 |
| Formalparamterliste | 8.5 |
| Deklarationsteil: | 11.1 |
| Block: | 12.1 |

# Aufrufe von Unterprogrammen

Unterprogrammaufrufe haben folgende Syntax:

```
Funktionsaufruf = Funktions-Name [Aktualparameterliste].

Prozeduraufruf  = Prozedur-Name [Aktualparameterliste].

Aktualparameterliste
                = "(" Aktualparameter {"," Aktualparameter} ")".

Aktualparameter = Variablen-Objekt | Ausdruck | Typ-Name
                | Prozedur-Name    | Funktions-Name
                | Paket-Bezeichner | Ausdruck ":" Formatangabe.

Formatangabe    = Integer-Ausdruck [":" Integer-Ausdruck].
```

Ein Prozeduraufruf veranlaßt die Ausführung des Blockes, der zu der aufgerufenen Prozedur gehört.

Ein Funktionsaufruf liefert einen Wert (das Funktionsergebnis), der durch die Ausführung des Funktions-Blocks ermittelt und durch den Funktions-Name repräsentiert wird.

Wenn das Unterprogramm Formalparameter hat, dann muß der Unterprogrammaufruf eine Aktualparameterliste mit den Aktualparametern enthalten, die für die entsprechenden Formalparameter eingesetzt werden:

a) Für jeden Wertparameter muß ein Ausdruck angegeben werden. Eine Formatangabe ist nur bei den vordefinierten Prozeduren Write, Writeln und Writestring zulässig.

b) Für jeden Variablenparameter muß ein Variablen-Objekt (eine verallgemeinerte Variable, siehe 9) angegeben werden.

c) Für jede Parameterprozedur bzw. -funktion muß ein Prozedur-Name bzw. Funktions-Name angegeben werden.

d) Ein Typ-Name darf nur bei den vordefinierten Attributfunktionen angegeben werden (siehe 15.9).

e) Ein Paket-Bezeichner darf nur bei der vordefinierten Prozedur Elaborate angegeben werden (siehe 15.12).

Ein Unterprogrammaufruf führt zu folgenden Aktivitäten:

- Auswertung der Aktualparameter und Zuordnung an die Formalparameter
- Bereitstellung eines Speicherbereichs für die lokalen Variablen und Parameter der Prozedur bzw. Funktion
- Ausführung der Anweisungen des Prozedur- bzw. Funktionsblocks
- Übergabe der Ergebnisse im Falle einer Funktion
- Freigabe des zugewiesenen Speicherbereichs.

In 13.3 sind die Aktivitäten bei der Ausführung eines Blocks nochmals genau beschrieben.

**Mögliche Laufzeitfehler:**

In 8.5.1, 8.5.4 und 13.3 sind weitere Fehler beschrieben, die bei der Parameterübergabe und dem Aufruf einer Prozedur bzw. Funktion auftreten können.

| | |
|---|---|
| undefinierte Auswirkungen | - Das Ergebnis einer Funktion ist nach Ausführung des Funktions-Blocks undefiniert, wenn dem Funktions-Bezeichner kein Wert zugewiesen wurde. |

**Implementierungsabhängige Eigenschaft**

Die Reihenfolge des Zugriffs, der Auswertung und der Zuordnung von Aktualparametern bei einem Prozeduraufruf bzw. Funktionsaufruf ist implementierungsabhängig.

*Beispiele für Prozeduraufrufe*

```
vertausche (aktx, akty);
   { Aufruf der in 8.1 vereinbarten Prozedur vertausche }
   { mit geeigneten Aktualparametern aktx, akty   }

Proz1;

trans (a, x, y);

bisect (fct (x + y), 2.0, 100, y);
```

*Beispiele für Funktionsaufrufe*

```
s := add (aktx, akty);
     { Aufruf der in 8.2 vereinbarten Funktion add }
     { mit 2 Aktualparametern aktx und akty }

wert := func (x);

bool := f (x, y, z);

s := g (x) + h (y);

Writeln ('Fakultaet ist ', fac(i));
```

*Querverweise*


| | |
|---|---|
| Objekt: | 9.6 |
| Variablen: | 7.1, 9.6 |
| Ausdruck: | 9 |
| Block: | 12.1 |
| Write, Writeln: | 15.1 |
| Writestring: | 15.3 |
| Attributfunktionen: | 15.9 |
| Elaborate: | 15.12 |

# Ausdrücke

## Allgemeines

Ein Ausdruck ist eine Rechenvorschrift, die nach ihrer Auswertung einen Wert liefert, es sei denn, während der Auswertung tritt eine Ausnahme auf oder der Ausdruck enthält einen Funktionsaufruf und der zugehörige Funktions-Block wird durch eine GOTO-Anweisung verlassen. Der Wert des Ausdrucks hängt von den Werten der Operanden (Variablen, Konstanten, Funktionsaufrufen) und von den Operatoren ab.

Ein Ausdruck hat folgende Syntax:

```
Ausdruck        = einfacher_Ausdruck [ Vergleichsoperator einfacher_Ausdruck ] .

Vergleichsoperator
                = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN" .

einfacher_Ausdruck
                = [ Vorzeichen ] Term { Summationsoperator Term } .

Vorzeichen      = "+" | "-" .

Summationsoperator
                = "+" | "-" | "OR" | "OR" "ELSE" .

Term            = Faktor { Multiplikationsoperator  Faktor } .

Multiplikationsoperator
                = "*" | "/" | "DIV" | "MOD" | "AND" | "AND" "THEN" .

Faktor          = Primitiv [ "**" Primitiv ] | "NOT" Faktor .

Primitiv        = vorzeichenlose_Konstante
                | Grenz-Bezeichner
                | "(" Ausdruck ")"
                | Mengenbildner
                | qualifizierter_Mengenbildner
                | Objekt .

vorzeichenlose_Konstante
                = Konstanten-Name
                | vorzeichenlose_Integer-Zahl
                | vorzeichenlose_Real-Zahl
                | Zeichenkette
                | "NIL" .
```

Als Erweiterung der Norm ist in Pascal-XT der Exponential-Operator ** eingeführt. Er hat die gleiche Priorität wie der NOT-Operator. Als Folge daraus wurde in Pascal-XT das Primitiv eingeführt. In Erweiterung der Norm kennt Pascal-XT auch die Kurzschlußoperatoren AND THEN und OR ELSE. Ein Primitiv kann auch ein "qualifizierter Mengenbildner" (9.4) oder ein "Objekt" (siehe 9.6) sein, welche ebenfalls eine Norm-Erweiterung darstellen.
Grenzbezeichner spielen im Zusammenhang mit Konformreihungs-Parametern (siehe 8.5.4) eine Rolle.

Ein Primitiv, ein Faktor, ein Term oder ein einfacher Ausdruck wird im folgenden als Operand bezeichnet.

Ausdrücke werden entsprechend den algebraischen Regeln berechnet. Da Ausdrücke aus einer Folge von Operanden und Operatoren bestehen, ist die Ausführung der einzelnen Operationen von Bedeutung und wie folgt festgelegt:

- Die Auswertung eines Ausdrucks besteht aus einer Reihe von Operationen mit Operanden und Operatoren, wobei Operatoren mit der höchsten Präzedenz zuerst ausgeführt werden (siehe 9.3).
- Eine Folge von Operationen mit gleicher Präzedenz werden von links nach rechts abgearbeitet (links-assoziativ).
- Ausdrücke in Klammern werden zuerst ausgewertet.
- Diese Regeln gelten wiederum für Ausdrücke in Klammern.

Die Auswertungsreihenfolge der Operanden dyadischer Operatoren ist nicht definiert (siehe implementierungsabhängige Eigenschaften in 9.3).

Der Wert eines Ausdrucks ist nicht definiert, wenn der Ausdruck eine verallgemeinerte Variable (als Objekt in einem Primitiv) enthält, der noch kein Wert zugewiesen wurde (undefinierte Variable, siehe 7.3).

Ist der Typ des Ojektes eines Primitivs ein Teilbereichstyp, so ist der Typ des Primitivs der Wirtstyp dieses Teilbereichstyps. Ist der Typ des Objekts eines Primitivs ein SET-Typ SET OF t, so ist der Typ des Primitivs SET OF w, dabei ist w der Wirtstyp des Typs t bzw. ein implementierungsdefinierter Teilbereich von Integer (siehe 6.3.4), wenn t ein Teilbereichstyp von Integer ist. In allen anderen Fällen ist der Typ des Primitivs identisch mit dem Typ des Objekts.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| undefinierte Auswirkungen | - Eine verallgemeinerte Variable, die als Objekt in einem Ausdruck verwendet wird, hat zum Zeitpunkt der Auswertung des (Teil-) Ausdrucks einen undefinierten Wert (siehe auch 7.3). |

*Beispiele für Primitive sind:*

```
-  Mengenbildner                     [1,5,x..y,z+23]
-  qualifizierter Mengenbildner      set_of_t_type ([1, 2])
-  Funktionsaufruf                   Sin (x+y)
```

*Beispiele für Faktoren sind:*

```
-  negierter Faktor                  NOT (a = b)
-  Potenz                            x ** 3
```

*Beispiele für Terme sind:*

```
x * y
i / (j - i)
(x <= y) AND (y < z)
(p <> NIL) AND THEN (p↑.i > 0)
```

*Beispiele für einfache Ausdrücke sind:*

```
x + y
-x
i * j + 1
b OR (x = y)
(Length (s) = 0) OR ELSE (s[1] = '.')
```

*Beispiele für Ausdrücke sind:*

```
x = 1.5
p <= q
(a + b) > (a * b)
next_char IN letters
```

*Querverweise*

vorzeichenlose Integer-Zahl: 3.5
vorzeichenlose Real-Zahl: 3.5
Zeichenkette: 6.3.2
Konstante: 5
Variablen: 7
undefinierter Wert: 7.3
Grenz-Bezeichner: 8.5.4
Konformreihungsparameter: 8.5.4
Operatoren: 9.3
Mengenbildner: 9.4
Objekt: 9.6.1
GOTO-Anweisung: 10.1.4
Block: 12.1
Ausnahme: 14

# Statische Ausdrücke

Ein statischer Ausdruck ist ein Ausdruck, dessen Wert bereits zur Übersetzungszeit ausgewertet werden kann. Demzufolge darf ein statischer Ausdruck keine verallgemeinerte Variable oder einen Funktionsaufruf enthalten. Folgende vordefinierten Unterprogramme dürfen in einem statischen Ausdruck auftreten, wenn ihre Aktualparameter wiederum statische Ausdrücke sind:

```
Abs            Long           Round          Sqr            Trunc
Short_Round    Long_Round     Short_Trunc    Long_Trunc
Chr            Ord            Odd            Pred           Succ
Card           Setmax         Setmin
Concat         Length         Substring
Alignof        Bitsizeof      First          Last           Maxlength  Sizeof
Convert
```

Für die Funktion Convert gelten noch weitere Einschränkungen, die in Kapitel 15.10 beschrieben sind.

Statische Ausdrücke können in Pascal-XT (als Erweiterung der Norm) überall dort eingesetzt werden, wo normgemäß nur Konstanten zulässig sind.

In den folgenden Kapiteln wird jeweils beschrieben, wann ein Ausdruck statisch ist.

*Beispiele statischer Ausdrücke*

```
2 * 3.14159
Maxint - 1
Length ('String-Konstante')
2 ** 8 -1
Concat (#'14', 'hervorgehoben', #'15')
Card (['a'..'z']  { liefert nicht immer 26 }
heute.jahr        {siehe statisches RECORD-Aggregat in 9.5.2 }
null_vector[1]    {siehe statisches ARRAY-Aggregat in 9.5.1 }
```

*Querverweise*

Konstanten: 5
Unterprogramme: 15

# Operatoren

Durch die Syntax für Ausdruck, einfacher Ausdruck, Term, Faktor und Primitiv werden Präzedenzregeln für die Operatoren festgelegt.

- Der Operator NOT und der Exponentionaloperator ** besitzen Vorrang vor allen anderen Operatoren.
- Es folgen die Multiplikationsoperatoren.
- Danach folgen die Summationsoperatoren und die Vorzeichen.
- Mit niedrigstem Vorrang versehen sind die Vergleichsoperatoren.

Folgen von zwei oder mehr Operatoren gleicher Präzedenz werden von links nach rechts ausgewertet (links-assoziativ). Diese Präzedenzregeln können durch die Verwendung von Klammern durchbrochen werden.

### Implementierungsabhängige Eigenschaften

Mit Ausnahme der beiden Kurzschlußoperatoren OR ELSE und AND THEN ist die Reihenfolge der Berechnung der Operanden eines dyadischen Operators implementierungsabhängig. Die Operanden können in der Reihenfolge ihrer Aufschreibung, in umgekehrter Reihenfolge, gleichzeitig oder möglicherweise teilweise gar nicht ausgewertet werden.

*Beispiele:*

```
a + b * c           ist gleichbedeutend mit a + (b * c) im Gegensatz zu
(a + b) * c

NOT b1 OR b2        bedeutet (NOT b1) OR b2 im Gegensatz zu
NOT (b1 OR b2)

a = b AND c = d     würde (a = (b AND c)) = d bedeuten, daher ist zu klammern:
(a = b) AND (c = d)
```

## Arithmetische Operatoren

Die Tabellen 9-1 und 9-2 geben eine Übersicht über die zulässigen Operandentypen für monadische und dyadische Operationen. In den Tabellen 9-3 bis 9-6 werden die Ergebnistypen für die dyadischen Operatoren im Detail beschrieben.

| Operator | Operation | Typ der Operanden | Typ des Ergebnisses |
|---|---|---|---|
| + | Identität | Integer<br>Real<br>Short_Integer<br>Long_Integer<br>Short_Real<br>Long_Real | Integer<br>Real<br>Integer<br>Long_Integer<br>Short_Real<br>Long_Real |
| - | Vorzeichenumkehr | Integer<br>Real<br>Short_Integer<br>Long_Integer<br>Short_Real<br>Long_Real | Integer<br>Real<br>Integer<br>Long_Integer<br>Short_Real<br>Long_Real |

Tabelle 9-1: Monadische arithmetische Operatoren (Vorzeichen)

| Operator | Operation | Typ der Operanden | Typ des Ergebnisses |
|---|---|---|---|
| +<br>-<br>* | Addition<br>Subtraktion<br>Multiplikation | Integer- oder Real-Typ | siehe Tabelle 9-3 |
| / | Division | Integer- oder Real-Typ | siehe Tabelle 9-4 |
| ** | Exponentiation | links Integer- oder Real-Typ, rechts Integer-Typ | siehe Tabelle 9-5 |
| DIV | ganzzahlige Division | Integer-Typ | siehe Tabelle 9-6 |
| MOD | Modulo | Integer-Typ | siehe Tabelle 9-6 |

Tabelle 9-2: Dyadische arithmetische Operatoren

*Hinweis*

Die Symbole +, -, * und / werden auch als Mengenoperatoren verwendet (siehe

9.3.3).

- **Integer-Operationen**

Bei Operationen auf Werten von einem Integer-Typ gelten folgende Eigenschaften:

Die monadischen arithmetischen Operationen "+" und "-", sowie die dyadischen Operationen "+", "-", "*", "**", "DIV", "MOD" werden gemäß den mathematischen Vorschriften ausgeführt, wenn die Operanden Integer-Werte besitzen und das Ergebnis im Wertebereich des Ergebnistyps (laut Tabelle 9-1 und 9-2) liegt.

Es tritt der Fehler Numeric_Error (siehe unten) auf, wenn der Ergebniswert einer arithmetischen Integer-Operation außerhalb des Wertebereichs liegt, d.h., wenn ein Überlauf auftritt.

*Hinweise*

- Auch bei der monadischen Operation "-" kann in Pascal-XT ein Fehler auftreten. Der Wert des Ausdrucks "-Long_Minint" liegt z. B. nicht im Wertebereich Long_Integer.
- Wenn Zwischenergebnisse in Ausdrücken außerhalb des jeweiligen Wertebereichs liegen, mit ihnen aber trotzdem mathematisch korrekt weitergerechnet werden kann, so ist es implementierungsabhängig, ob korrekt weitergerechnet wird oder ein Numeric_Error auftritt.
- Eine dyadische Operation mit Operanden vom Typ Short_Integer liefert als Ergebnis einen Wert vom Typ Integer, der implementierungsdefiniert Short_Integer oder Long_Integer sein kann. Bei der Portierung von Programmen muß dies bedacht werden, da sich die Programme sonst unterschiedlich verhalten.
- Zur Vermeidung von Überläufen bei Operanden vom Typ Short_Integer muß mindestens einer der Operanden mit der vordefinierten Funktion Long (siehe 15.5) in einen Wert des Typs Long_Integer umgewandelt werden, z. B.

  ```
  x := y * Long (z)
  ```

- Ein Ausdruck, der nur aus Operanden eines Integer-Typs besteht und dessen Wert an eine Real-Variable zugewiesen werden soll, wird in Integer-Arithmetik berechnet und erst bei der Zuweisung in einen Realwert konvertiert.

- **Real-Operationen**

Real-Operationen sind monadische und dyadische arithmetische Operationen mit mindestens einem Operanden eines Real-Typs, sowie der Operator "/" mit Operanden vom Integer-Typ.

Die Werte der Real-Typen bilden Teilmengen der reellen Zahlen. Die Ergebnisse der arithmetischen Real-Operatoren sind daher i. a. nur Näherungswerte der entsprechenden mathematischen Ergebnisse (siehe 6.2.2). Real-Operationen können maschinenabhängig intern auch mit einer höheren Genauigkeit berechnet werden, wobei dann Genauigkeitsverluste beim Abspeichern der Werte auftreten können. Vergleichs-Operationen können z.B. unterschiedliche Ergebnisse liefern, abhängig davon ob Variable miteinander oder Variable mit dem Ergebnis eines Ausdrucks verglichen werden. Zusätzlich beeinflußt das Rundungsverhalten der Maschine das Ergebnis. Insbesondere sollten Vergleiche auf Gleichheit vermieden werden.

Bei den mathematischen Funktionen können maschinenabhängig ebenfalls Ungenauigkeiten auftreten (z.B. liefert Round (355.5) auf Rechnern mit IEEE-Arithmetik den falschen Wert 355).

Es tritt ein Numeric_Error auf, wenn der Nährungswert des Ergebnisses einer Real-Operation außerhalb des Wertebereichs der Ergebnistyps (laut Tabellen 9-1 und 9-2) liegt.

### Operationen mit Real-Konstanten

Der Typ einer Real-Konstanten ist ein sog. universeller Real-Typ, dessen Wertebereich identisch mit dem von Long_Real ist. Ist in einer dyadischen Operation der Typ eines Operanden der universelle Real-Typ, dann wird die Operation mit der Genauigkeit durchgeführt, die durch den Typ des anderen Operanden bestimmt wird. Sind beide Operanden vom universellen Real-Typ, dann wird mit der Genauigkeit des Typs Long_Real gerechnet.

### Mischung von Integer- und Real-Operanden

Ist bei einem dyadischen Operator einer der Operanden von einem Integer-Typ und der andere von einem Real-Typ, dann erfolgt zuerst implizit eine Typumwandlung. Der Wert des Integer-Operanden wird in eine Real-Zahl umgewandelt, die dann den universellen Real-Typ besitzt.

- **Die arithmetischen Operatoren "+", "-" und "*"**

<table>
<tr><th colspan="2">Typ der Operanden</th><th>Ergebnistyp</th></tr>
<tr><th>x</th><th>y</th><th>x (+|-|*) y</th></tr>
<tr><td>Integer</td><td>Integer<br>Real</td><td>Integer<br>Real</td></tr>
<tr><td>Real</td><td>Integer<br>Real</td><td>Real<br>Real</td></tr>
<tr><td>Short_Integer</td><td>Short_Integer<br>Long_Integer<br>Short_Real<br>Long_Real</td><td>Integer<br>Long_Integer<br>Short_Real<br>Long_Real</td></tr>
<tr><td>Long_Integer</td><td>Short_Integer<br>Long_Integer<br>Short_Real<br>Long_Real</td><td>Long_Integer<br>Long_Integer<br>Short_Real<br>Long_Real</td></tr>
<tr><td>Short_Real</td><td>Short_Real<br>Long_Real<br>Integer-Typ</td><td>Short_Real<br>Long_Real<br>Short_Real</td></tr>
<tr><td>Long_Real</td><td>Real-Typ<br>Integer-Typ</td><td>Long_Real<br>Long_Real</td></tr>
<tr><td>universeller<br>Real-Typ</td><td>universeller<br>Real-Typ<br>Long_Real<br>Short_Real<br>Integer-Typ</td><td>universeller<br>Real-Typ<br>Long_Real<br>Short_Real<br>universeller<br>Real-Typ</td></tr>
</table>

Tabelle 9-3: Ergebnistyp bei den Operationen "+", "-", "*"

- **Divisionsoperator "/"**

Bei der Division ist das Ergebnis immer von einem Real-Typ.

| Typ der Operanden | | Ergebnistyp |
|---|---|---|
| x | y | x / y |
| Integer | Integer<br>Real | Real<br>Real |
| Real | Integer<br>Real | Real<br>Real |
| Short_Real | Short_Real<br>Long_Real<br>Integer-Typ | Short_Real<br>Long_Real<br>Short_Real |
| Long_Real | Real-Typ<br>Integer-Typ | Long_Real<br>Long_Real |
| Integer-Typ | Short_Real<br>Long_Real<br>Integer-Typ | Short_Real<br>Long_Real<br>Long_Real |
| universeller Real-Typ | universeller Real-Typ<br>Long_Real<br>Short_Real<br>Integer-Typ | universeller Real-Typ<br>Long_Real<br>Short_Real<br>universeller Real-Typ |

Tabelle 9-4: Ergebnistyp bei der Operation "/"

- **Die Exponentialoperation "**"**

| Typ der Operanden | | Ergebnistyp |
|---|---|---|
| x | n | x ** n |
| Integer<br>Real | Integer<br>Integer | Integer<br>Real |
| Short_Integer<br>Long_Integer | Integer-Typ<br>Integer-Typ | Integer<br>Long_Integer |
| Short_Real<br>Long_Real | Integer-Typ<br>Integer-Typ | Short_Real<br>Long_Real |
| universeller Real-Typ | Integer-Typ | universeller Real-Typ |

Tabelle 9-5: Ergebnistyp bei der Exponentation

- **Die Operatoren "DIV" und "MOD"**

| Typ der Operanden | | Ergebnistyp |
|---|---|---|
| x | y | x (DIV\|MOD) y |
| Integer | Integer | Integer |
| Short_Integer | Short_Integer<br>Long_Integer | Integer<br>Long_Integer |
| Long_Integer | Integer-Typ | Long_Integer |

Tabelle 9-6: Ergebnistyp bei den Integer-Operationen "DIV" und "MOD"

Für den Operator DIV gilt:

```
Abs(x) - Abs(y) < Abs (x DIV y) * y <= Abs(x)
```

Das Ergebnis ist null, wenn Abs(x) < Abs(y). Das Ergebnis ist positiv, wenn x und y dasselbe Vorzeichen haben, und negativ, wenn beide verschiedene Vorzeichen haben.

*Beispiele*

```
  5 DIV 3   ergibt  1
 -5 DIV 3   ergibt -1

  5 MOD 3   ergibt  2
  4 MOD 3   ergibt  1
  3 MOD 3   ergibt  0
  2 MOD 3   ergibt  2
  1 MOD 3   ergibt  1
  0 MOD 3   ergibt  0
(-1) MOD 3   ergibt  2
(-2) MOD 3   ergibt  1   (usw.)
(-3) MOD 3   ergibt  0   (der Rest wird bis zum nächstkleineren
                          Vielfachen des Divisors bestimmt)
```

Bei der Ausführung einer arithmetischen Operation tritt in verschiedenen Fällen ein Numeric_Error auf:

### Mögliche Laufzeitfehler:

| | |
|---|---|
| `Numeric_Error` | `- In einem Ausdruck der Form x/y ist y = 0.`<br><br>`- In einem Ausdruck der Form i DIV j ist j = 0.`<br><br>`- In einem Ausdruck der Form i MOD j ist j <= 0.`<br><br>`- Das Ergebnis einer arithmetischen Operation liegt nicht im Wertebereich des Ergebnistyps.`<br><br>`- In einem Ausdruck der Form x**n ist`<br>`-  x von einem Integer-Typ und n < 0, oder`<br>`-  x = 0 bzw. 0.0 und n <= 0.` |

*Querverweise*

Fehler: 2.3, A.5
Konstanten: 5
Integer-Typen: 6.2.1
Real-Typen: 6.2.2

## Boolesche Operatoren

Operanden und Ergebnisse Boolescher Operatoren sind vom Typ Boolean.

| Operator | Operation | Typ der Operanden | Typ des Ergebnisses |
|---|---|---|---|
| OR | logische Disjunktion | Boolean | Boolean |
| AND | logische Konjunktion | Boolean | Boolean |
| NOT | logische Negation | Boolean | Boolean |
| OR ELSE | logische Disjunktion | Boolean | Boolean |
| AND THEN | logische Konjunktion | Boolean | Boolean |

Tabelle 9-7: Boolesche Operatoren

| a | b | a OR b |
|---|---|---|
| False | False | False |
| False | True | True |
| True | False | True |
| True | True | True |

Disjunktion

| a | b | a AND b |
|---|---|---|
| False | False | False |
| False | True | False |
| True | False | False |
| True | True | True |

Konjunktion

| a | NOT a |
|---|---|
| False | True |
| True | False |

Negation

Tabelle 9-8: Wahrheitswerte bei Disjunktion, Konjunktion und Negation

Die Reihenfolge der Auswertung der Operanden bei OR und AND ist nach 9.3 implementierungsabhängig.
Bei den beiden Kurzschluß-Operatoren AND THEN und OR ELSE wird hingegen immer der linke Operand zuerst und der rechte Operand in Abhängigkeit vom Ergebnis des linken Operanden ausgewertet (Kurzschluß).

```
a OR ELSE b      Falls der Wert von a True ist, so wird b nicht ausgewertet
                 und das Ergebnis ist True.

a AND THEN b     Falls der Wert von a False ist, so wird b nicht ausgewertet
                 und das Ergebnis ist False.
```

Ein Anwendungsbereich für die Kurzschlußoperatoren ist z. B. die Bearbeitung von Listen, wenn zusätzlich zum Abbruchkriterium Listenende noch weitere Kriterien vorgegeben sind. Beim Erreichen des Listenendes dürfen die weiteren Operanden keinesfalls mehr ausgewertet werden, wenn sie Zeigerderefenzierungen

beinhalten (siehe Beispiel 1).

*Beispiel 1*

Das Durchlaufen der Liste list soll beendet werden, wenn das Listenende erreicht ist oder ein Element gefunden wird, dessen Komponente zahl den Wert 1 hat.

```
WHILE (list <> NIL) AND THEN (list↑.zahl <> 1) DO
   list := list↑.next;
```

*Beispiel 2*

Es soll eine Aktion erfolgen, wenn das Ende eines Arrays erreicht oder ein negatives Element gefunden wird:

```
IF (index > max_index) OR ELSE (a[index] < 0) THEN ...
```

*Querverweise*

Typ Boolean: 6.2.4
Auswertung: 9.3

## Mengenoperatoren

Die Vorschriften für die Typen der Operanden und der Ergebnisse der Mengenoperationen finden sich in der folgenden Tabelle.

| Operator | Operation | Typ der Operanden | Typ des Ergebnisses |
|---|---|---|---|
| + | Mengen-vereinigung | SET-Typ | Typ der Operanden |
| - | Mengen-differenz | SET-Typ | Typ der Operanden |
| * | Mengen-durchschnitt | SET-Typ | Typ der Operanden |
| / | symmetrische Mengen-Differenz | SET-Typ | Typ der Operanden |

Tabelle 9-9: Mengenoperationen

Die SET-Typen der beiden Operanden eines Mengenoperators müssen miteinander verträglich sein, d. h., sie müssen verträgliche Basistypen besitzen und entweder beide gepackt oder beide ungepackt sein. Wenn einer der Operanden ein nicht qualifizierter Mengenbildner ist, so gilt sein Typ abhängig vom Typ des anderen Operanden als gepackt bzw. ungepackt (siehe 9.4).

Pascal-XT kennt zusätzlich die symmetrische Mengendifferenz "/" (entspricht dem exklusiven ODER), die wie folgt definiert ist:

```
A / B = (A - B) + (B - A)
```

*Hinweise*

- In Pascal-XT können Mengenoperationen nicht ausgeführt werden, wenn die implementierungsdefinierte maximale Anzahl von Werten des Basistyps der Menge überschritten wird. So können z. B. die Vereinigung und die symmetrische Differenz zweier Mengen eine zu große Menge ergeben.
- Die Symbole "+", "-", "*" und "/" werden auch als arithmetische Operatoren verwendet.

*Beispiele*

```
      digits     = ['0' .. '9'];
      hexletters = ['a','b','c','d','e','f'];
      hexdigits  = digits + hexletters;

   VAR
      menge1 : SET OF 0 .. 200;
      menge2 : SET OF 100 .. 300;
      menge3 : SET OF 0 .. 1000;

   BEGIN
      menge1 := [0, 10..20, 100, 200];
      menge2 := [99, 101, 300];
      menge3 := menge1 + menge2;        { = [0, 10..20, 99..101, 200, 300] }
      menge3 := menge1 * menge2;        { = [], die leere Menge }
   END
```

*Querverweise*

| | |
|---|---|
| Verträglichkeit: | 6.6.2 |
| SET-Typ: | 6.3.4 |
| Basistyp: | 6.3.4 |
| gepackt, ungepackt: | 6.1 |
| Mengenbildner: | 9.4 |

## Vergleichsoperatoren

Die Vorschriften für die Typen der Operanden und der Ergebnisse der Vergleichsoperationen finden sich in der folgenden Tabelle.

| Operator | Typ der Operanden | Typ des Ergebnisses |
|---|---|---|
| = <> | Ordinal-Typ, Real-Typ, Zeigertyp, Zeichenkettentyp oder SET-Typ | Boolean |
| < > | Ordinal-Typ, Real-Typ, Zeichenkettentyp oder SET-Typ | Boolean |
| <= >= | Ordinal-Typ, Real-Typ, Zeichenkettentyp oder SET-Typ | Boolean |
| IN | linker Operand: ein Ordinaltyp<br>rechter Operand: ein SET-Typ | Boolean |

Tabelle 9-10: Vergleichsoperationen

Bedeutung der Vergleichsoperatoren:

```
u =  v   u gleich v
u <> v   u ungleich v
u <  v   für Ordinaltypen und Real-Typen: u kleiner als v
u <  v   bei SET-Typen: u echte Teilmenge von v (d. h. nicht gleich)
u >  v   für Ordinaltypen und Real-Typen: u größer als v
u >  v   bei SET-Typen: v echte Teilmenge von u (nicht gleich)
u <= v   für Ordinaltypen und Real-Typen: u kleiner oder gleich v
u >= v   für Ordinaltypen und Real-Typen: u größer oder gleich v
u <= v   bei SET-Typen: u Teilmenge von v
u >= v   bei SET-Typen: v Teilmenge von u
a IN v   a ist Element der Menge v
```

- **IN - Operator**

Der Operator IN liefert den Wert True, wenn der Wert des linken Operanden ein Element des Wertes des rechten Operanden ist; ansonsten ergibt sich der Wert False.

Der Ordinaltyp des linken Operanden muß verträglich sein mit dem Basistyp des SET-Typs des rechten Operanden.

- **Vergleiche von Werten einfacher Typen**

Die Typen der Operanden müssen entweder verträglich sein oder einer der Operanden ist von einem Real-Typ und der andere Operand ist von einem Integer-Typ.

Da Real-Zahlen i. a. nur Näherungswerte der reellen Zahlen darstellen, können Vergleiche zweier Real-Zahlen bzw. einer Real-Zahl mit einer Integer-Zahl nicht das erwartete Ergebnis liefern. Insbesondere sollte die Abfrage auf Gleichheit ("=") oder Ungleichheit ("<>") bei Real-Operanden vermieden werden.

- **Vergleiche von Zeigern**

Die Zeigertypen müssen verträglich sein, d. h. die beiden Operanden besitzen denselben Typ (siehe 6.6.1) oder einer der beiden Operanden besitzt den generischen Zeigertyp (siehe 6.5.2). Auf Zeigertypen sind nur die Vergleiche "=" und "<>" anwendbar.

Es können also nur Zeiger miteinander verglichen werden, die auf dynamische Variable desselben Typs verweisen. Durch die Vergleiche mit "=" und "<>" kann festgestellt werden, ob zwei Zeiger auf dieselbe dynamische Variable verweisen bzw. ob eine Zeigervariable den Wert NIL hat.

- **Vergleiche von Mengen**

Die SET-Typen der beiden Operanden müssen verträglich sein. D. h., sie müssen verträgliche Basistypen besitzen und entweder beide gepackt oder beide ungepackt sein. Wenn einer der beiden Operanden ein nicht qualifizierter Mengenbildner ist, so gilt sein Typ als ungepackt, es sei denn der SET-Typ des anderen Operanden ist gepackt.

Pascal-XT läßt als Erweiterung zur Norm auch die Vergleiche "<" und ">" (echte Teilmenge bzw. echte Obermenge) zu.

- **Vergleiche von Zeichenketten**

Beim Vergleich von Zeichenketten müssen deren Typen verträglich sein. In Tabelle 9-11 sind die möglichen Kombinationen von Operanden angegeben, wobei PACKED ARRAY abkürzend für die Zeichenkettentypen fester Länge (siehe 6.3.2.1) und String [n] bzw. String [m] für die String-Typen (siehe 6.3.2.2) steht.

| | PACKED ARRAY [1..n] OF Char | String [m] | Char |
|---|---|---|---|
| PACKED ARRAY [1..k] OF Char | wenn k = n auch k <> n | ja | nein |
| String [n] | ja | ja | ja |
| Char | nein | ja | ja |

Tabelle 9-11: Verträglichkeit von Operandentypen bei Zeichenkettenvergleichen

Der Vergleich zweier Zeichenketten wird nach dem unten beschriebenen lexikographischen Vergleich durchgeführt. Dieser lexikographische Vergleich definiert eine Totalordnung auf der Menge aller Zeichenketten. Der Vergleich erfolgt zeichenweise von links nach rechts, wobei beim Vergleich einander entsprechenden Zeichen die Ordnung der implementierungsdefinierten Zeichen des Typs Char maßgebend ist.
Bei ungleicher aktueller Länge von s1 und s2 wird in der Länge des kürzeren Strings verglichen. Ergibt dieser Vergleich Gleichheit, gilt der String mit der kleineren aktuellen Länge als kleiner.

s1 und s2 seien 2 Zeichenketten. Die (aktuelle) Länge von s1 sei n1, die von s2 sei n2; n sei der kleinere Wert von n1 und n2. Dann gilt:

```
s1 = s2 genau dann, wenn n1 = n2 und für alle i in [1..n] gilt:
                    s1[i] = s2[i].

s1 < s2 genau dann, wenn es ein p in [1..n] gibt
                         und für alle i in [1..p-1] gilt:
                         s1[i] = s2[i] und s1[p] < s2[p],
               oder wenn für alle i in [1..n] gilt:
                         s1[i] = s2[i] und n1 < n2 ist.

s1 >  s2 ist gleichbedeutend mit s2 < s1
s1 <= s2 ist gleichbedeutend mit (s1 < s2) OR (s1 = s2)
s1 >= s2 ist gleichbedeutend mit (s1 > s2) OR (s1 = s2)
s1 <> s2 ist gleichbedeutend mit NOT (s2 = s1)
```

*Beispiel*

Tabelle 9-12 zeigt die Auswertung des Ausdrucks
"a Vergleichsoperator b", wobei a und b Zeichenketten sind:

| a | b | < | = | <= | > | >= | <> |
|---|---|---|---|---|---|---|---|
| 'ABC' | 'ABCD' | True | False | True | False | False | True |
| 'ABCD' | 'ABC' | False | False | False | True | True | True |
| 'ABCD' | '' | False | False | False | True | True | True |
| 'ABC' | 'BBB' | True | False | True | False | False | True |
| 'A' | 'A' | False | True | True | False | True | False |
| 'B' | 'AAAA' | False | False | False | True | True | True |
| 'B' | 'A' | False | False | False | True | True | True |
| 'A' | 'A ' | True | False | True | False | False | True |

Tabelle 9-12: Vergleich von Zeichenketten

*Querverweise*

gepackt, ungepackt: 6.1
Ordinal-Typen: 6.2
Real-Typen: 6.2.2
Zeichenkettentypen: 6.3.2
Mengentypen: 6.3.4
Zeigertypen: 6.4
Typidentität: 6.6.1
Typenverträglichkeit: 6.6.2

# Mengenbildner

Die Syntax von Mengenbildner lautet:

```
Mengenbildner   = "[" [Elementbestimmung
                       {"," Elementbestimmung} ] "]".

Elementbestimmung
                = Ordinal-Ausdruck [".." Ordinal-Ausdruck].

qualifizierter_Mengenbildner
                = Typ-Name "(" Mengenbildner ")".
```

Ein Mengenbildner steht für einen Wert eines SET-Typs. Der Mengenbildner "[]" ohne Elementbestimmungen wird als leere Menge bezeichnet und ist ein Wert eines jeden SET-Typs.

Ein Mengenbildner repräsentiert einen Wert, der null, ein oder mehrere Elemente enthält. Jedes Element wird durch mindestens eine der Elementbestimmungen des Mengenbildners angegeben. Eine Elementbestimmung, die aus einem einzigen Ausdruck besteht, steht für den Wert, den der Ausdruck hat. Eine Elementbestimmung der Form `a .. b` beschreibt ein Interval, d. h. alle Werte x mit $a \le x \le b$. Ist $a > b$, dann beschreibt die Elementbestimmung a..b kein Element.

Ein Mengenbildner, der eine oder mehrere Elementbestimmungen enthält, steht für einen Wert des Typs SET OF w, wobei w der gemeinsame Typ (Wirtstyp) aller Ausdrücke ist, die in den Elementbestimmungen vorkommen. Dieser Wirtstyp muß ein Ordinal-Typ sein.
Die Größe des Wirtstyps kann implementierungsdefiniert beschränkt sein. In diesem Fall wird ein maximal zulässiger Teilbereich des Wirtstyps, dessen kleinster Ordinalwert 0 und dessen größter Ordinalwert implementierungsdefiniert ist, als Basistyp des SET-Typs angenommen.

Der Typ eines (nicht qualifizierten) Mengenbildners gilt als ungepackt, außer der Kontext erfordert einen gepackten SET-Typ. Das ermöglicht es, Mengenbildner an gepackte und ungepackte SET-Variablen zuzuweisen und sie mittels Mengen- und Vergleichsoperatoren zu verknüpfen.
Wenn ein Mengenbildner aber als Konstante in einer Konstantendefinition auftritt, dann gibt es dort keinen Kontext, der einen gepackten SET-Typ erfordert. Daher besitzt die dadurch definierte Konstante einen ungepackten SET-Typ (siehe auch 5.1).

- **Qualifizierter Mengenbildner**

Pascal-XT kennt zusätzlich den qualifizierten Mengenbildner, dessen Typ explizit durch Angabe eines Typ-Namens bestimmt wird und nicht mehr implizit

aus dem gemeinsamen Typ aller Elementbestimmungen. Die Werte für die die Elementbestimmungen eines qualifizierten Mengenbildners stehen, müssen zuweisungsverträglich zum Basistyp des angegebenen SET-Typs sein.

### Implementierungsdefinierte Eigenschaft

Der größte Ordinalwert des Basistyps eines SET-Typs eines nicht qualifizierten Mengenbildners ist implementierungsdefiniert.

### Implementierungsabhängige Eigenschaft

Die Reihenfolge der Auswertung der Elementbestimmungen und der Ausdrücke in den Elementbestimmungen in einem Mengenbildner ist implementierungsabhängig.

### Mögliche Laufzeitfehler

| | |
|---|---|
| `Set_Error` | `- In einem Mengenbildner liegt der Wert einer Elementbestimmung nicht im Wertebereich des Basis-Typs des Mengenbildners.` |

*Hinweise*

- Die maximale Anzahl der Werte des Basistyps einer Menge kann in Pascal-XT beschränkt sein (siehe 6.3.4).
- Qualifizierte Mengenbildner werden insbesondere dann benötigt, wenn für einen SET OF t der Basistyp t ein Teilbereich des Typs Integer ist, der nicht vollständig im Teilbereich 0 .. n enthalten ist, wobei n der implementierungsdefinierte größte Ordinalwert des Basistyps eines SET-Typs ist.
- Die Reihenfolge, in der die Elementbestimmungen in einem Mengenbildner angegeben werden, kann beliebig sein; es ist keine Ordnung vorgeschrieben.

*Beispiele*

```
[ ]                              { Die leere Menge }

[40, 49, 11..33])                { Reihenfolge der Werte ist beliebig }

[True, False]                    { Alle Werte des Booleschen Datentyps }

[rot, gelb]                      { Nur die Werte rot und gelb }

[1..5, x..y, a + b, 17, 31]      { Variablen und Ausdrücke in
                                   Elementbestimmungen }

[f(x)..f(y)]                     { Funktionsaufrufe in Elementbestimmungen }

[-10]                            { qualifizierter Mengenbildner notwendig,
                                   da -10 nicht im Intervall 0 .. n liegt }

st1 ([-10, 10])                  { qualifizierter Mengenbildner, z. B.
                                   st1 = SET OF -10..10 }

st2 ([8000..9000])               { qualifizierter Mengenbildner, z. B.
                                   st2 = SET OF 8000..9000 }
```

*Querverweise*

| | |
|---|---|
| Ordinaltyp: | 6.2.2 |
| Wirtstyp: | 6.2.6 |
| Teilbereich: | 6.2.6 |
| Basistyp: | 6.3.4 |
| SET-Typ: | 6.3.4 |
| Zuweisungsverträglich: | 6.6.3 |
| Ausdruck: | 9 |

# Aggregate

Aggregate dienen der Bildung von Werten von RECORD-Typen und ARRAY-Typen aus den Werten ihrer Komponenten (siehe 9.5.1 und 9.5.2).

```
Aggregat = ARRAY-Aggregat | RECORD-Aggregat .
```

Ein Aggregat kann als statischer Ausdruck auf der rechten Seite einer Konstantendefinition stehen. Damit können also auch RECORD- und ARRAY-Konstanten definiert werden.

Ein Aggregat, dessen Aggregat-Elemente alle statische Ausdrücke sind, ist ein statisches Aggregat.

**Implementierungsabhängige Eigenschaft**

Die Reihenfolge der Auswertung der Ausdrücke und der Zuordnung zu den Komponenten des Typs eines Aggregats ist implementierungsabhängig.

Der Wert jedes Aggregatelements muß zuweisungsverträglich zum Typ der entsprechenden RECORD- bzw. ARRAY-Komponenten sein.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Numeric_Error | - In einem Aggregat liegt der Wert eines Aggregat-Elements vom Typ Long_Real nicht im Wertebereich des Typs Short_Real der zugehörigen Komponente des Aggregats. |
| Range_Error | - In einem Aggregat liegt der Wert eines Aggregat-Elements eines Ordinal-Typs nicht im Wertebereich des Ordinal-Typs der zugehörigen Komponente des Aggregats. |
| Set_Error | - In einem Aggregat liegt der Wert eines Aggregat-Elements von einem SET-Typ nicht im Wertebereich des SET-Typs der zugehörigen Komponente des Aggregats. |
| String_Error | - In einem Aggregat ist die aktuelle Länge einer Zeichenkette eines Aggregat-Elements größer als die Maximallänge des String-Typs der zugehörigen Komponente des Aggregats.<br>- In einem Aggregat ist die aktuelle Länge einer Zeichenkette eines Aggregat-Elements (von einem String-Typ) ungleich der Länge der zugehörigen Komponente des Aggregats (von einem Zeichenketten-Typ fester Länge). |
| undefinierte Auswirkungen | - In einem Aggregat ist der Typ eines Zeigerwertes vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamische Variable, deren Typ vom Domänentyp des Typs der entsprechenden Aggregatkomponente verschieden ist. |

## ARRAY-Aggregate

ARRAY-Aggregate haben folgende Syntax:

```
ARRAY-Aggregat  = ARRAY-Typ-Name "(" ARRAY-Aggregat-Element
                             { "," ARRAY-Aggregat-Element } ")".

ARRAY-Aggregat-Element
                = Ausdruck [":" Wiederholfaktor].

Wiederholfaktor = Integer-Konstante.
```

Die Werte der ARRAY-Aggregat-Elemente müssen zum Komponententyp des ARRAY-Typs zuweisungsverträglich sein. Den Werten des Indextyps werden in aufsteigender Reihenfolge die Werte der ARRAY-Aggregat-Elemente in textlicher Reihenfolge zugeordnet. Der Wert des ARRAY-Aggregates besteht dann aus den zugeordneten Komponenten-Werten.

Bei einem ARRAY-Aggregat-Element muß ein eventuell vorhandener Wiederholfaktor ein statischer Ausdruck eines Integer-Typs sein, dessen Wert > 0 ist.
Ein ARRAY-Aggregat-Element mit Angabe eines Wiederholfaktors mit dem Wert n steht für eine n-malige Wiederholung desselben Komponenten-Wertes. In dem ARRAY-Aggregat müssen (unter Berücksichtigung eventueller Wiederholfaktoren) genau so viele ARRAY-Aggregat-Elemente vorkommen, wie der zu dem ARRAY-Typ des Aggregats gehörige Indextyp Werte hat.

Ist das ARRAY-Aggregat-Element von einem ARRAY- oder RECORD-Typ, dann muß es wieder als Aggregat angegeben werden (siehe matrix im Beispiel).

*Beispiel*

```
TYPE
   vector = ARRAY [1..5] OF Integer;
   matrix = ARRAY [1..5] OF vector;
CONST
   null_vektor = vector ( 0: 5 );
   null_matrix = matrix (null_vector : 5);
VAR
   m : matrix;
BEGIN
   m := matrix (null_vektor : 5);        { = null_matrix }
   m := matrix (vector (1, 0:4),         { 1 0 0 0 0 }
                vector (0, 1, 0:3),      { 0 1 0 0 0 }
                vector (0:2, 1, 0:2),    { 0 0 1 0 0 }
                vector (0:3, 1, 0) : 2); { 0 0 0 1 0 }
                                         { 0 0 0 1 0 }
END
```

*Querverweise*

| | |
|---|---|
| ARRAY-Typ: | 6.3.1 |
| Indextyp: | 6.3.1 |
| Komponententyp: | 6.3.1 |
| Konstante: | 5 |
| Zuweisungsverträglichkeit: | 6.6.3 |
| Ausdruck: | 9 |
| statischer Ausdruck: | 9.2 |

## RECORD-Aggregate

RECORD-Aggregate haben folgende Syntax:

```
RECORD-Aggregat = RECORD-Typ-Name "(" Ausdruck {"," Ausdruck} ")".
```

Der Typ eines RECORD-Aggregats muß ein nicht leerer RECORD-Typ sein. Die ersten RECORD-Aggregat-Elemente werden zunächst in textlicher Reihenfolge den Feldern des Festteils der Feldliste des RECORD's zugeordnet.

Besitzt diese Feldliste einen Variantteil, so muß dieser ein Kennungsfeld besitzen und das nächste Aggregat-Element muß ein statischer Ausdruck sein. Der Wert dieses statischen Ausdrucks wird dann dem Kennungsfeld zugeordnet und bestimmt die Variante des Wertes des RECORD-Aggregates.

Die restlichen RECORD-Aggregat-Elemente werden dann in gleicher Weise den Feldern der zu dieser Variante gehörenden Feldliste zugeordnet. Die Anzahl der RECORD-Aggregat-Elemente muß mit der Gesamtzahl der Felder der in der oben beschriebenen Weise ausgewählten Varianten und deren Feldlisten exakt übereinstimmen.

Der Wert jedes RECORD-Aggregat-Elements muß zuweisungsverträglich mit dem Typ des zugeordneten Feldes bzw. Kennungsfeldes sein. Der Wert des RECORD-Aggregats besteht dann aus den so zugeordneten Komponenten-Werten.

Ist ein ARRAY-Aggregat-Element von einem ARRAY- oder RECORD-Typ, dann muß es wieder als Aggregat angegeben werden.

*Beispiel*

```
TYPE
   rectyp = RECORD
               x : Integer;
               CASE b : Boolean OF
                  True  : (u : Integer);
                  False : (v,w : Char);
            END;

CONST
   const1  = rectyp (3, True, 5);
   const2  = rectyp (3, False, 'A', 'B');

TYPE
   datum = RECORD
             tag:   1..31;
             monat: 1..12;
             jahr:  1..2000;
           END;
```

```
CONST
   heute  = datum (1, 1, 1986);

FUNCTION eingabe_datum: datum;
VAR
   t, m, j: Integer;
BEGIN
   Writeln ('Bitte Tag, Monat und Jahr eingeben!');
   Readln; Read (t, m, j);
   eingabe_datum := datum (t, m, j);
END;
```

*Querverweise*


RECORD-Typ: 6.3.3<br>
Festteil, Feldliste: 6.3.3<br>
Kennungsfeld, Variantteil: 6.3.3.1<br>
Zuweisungsverträglichkeit: 6.6.3<br>
Ausdruck: 9

# Objekte

## Allgemeines

In Pascal-XT gibt es neben Komponenten von Variablen auch Komponenten von strukturierten Konstanten und von strukturierten Werten, die durch Aggregate und Funktionsergebnisse bestimmt werden. Aus diesem Grund wurde als Erweiterung von Norm-Pascal der Begriff Objekt mit folgender Syntax eingeführt:

```
Objekt  = Konstanten-Name          | Variablen-Name
        | Aggregat                 | Funktionsaufruf
        | indiziertes_Objekt       | selektiertes_Objekt
        | dereferenziertes_Objekt  | Puffervariable.
```

Objekte sind Konstanten, Variablen, Aggregate, Ergebnisse von Funktionsaufrufen, oder Puffervariablen. Sind die Typen dieser Objekte strukturiert, dann kann auch auf Komponenten dieser Objekte zugegriffen werden. So ist ein indiziertes Objekt eine Komponente eines Objekts von einem ARRAY-Typ oder String-Typ, auf die indiziert zugegriffen wird. Ein selektiertes Objekt ist eine Komponente eines Objekts von einem RECORD-Typ, auf das durch Selektion (Feldauswahl) zugegriffen wird. Ein dereferenziertes Objekt ist der Zugriff auf eine dynamisch erzeugte Variable (siehe 9.6.4), auf die durch Zeigerdereferenzierung zugegriffen wird. Eine Puffervariable wird implizit zu jeder FILE-Variable deklariert.

An folgenden Stellen müssen Objekte verallgemeinerte Variablen (Variablen-Objekte) sein:

- als Aktualparameter, wenn der entsprechende Formalparameter ein Variablenparameter ist
- als Aktualparameter an bestimmten Positionen in den Aufrufen der vordefinierten Prozeduren New, Mark, Pack, Unpack, Read Readln, Readstring, Writestring, Delete und Insert
- auf der linken Seite einer Zuweisung
- als RECORD-Variable in einer WITH-Anweisung

Es gelten folgende Regeln:

- Objekte, die durch einen Konstanten-Namen bezeichnet werden, sind keine verallgemeinerten Variablen.
- Objekte, die durch einen Variablen-Namen bezeichnet werden, sind verallgemeinerte Variablen.
- Aggregate sind keine verallgemeinerten Variablen.
- Funktionsaufrufe sind keine verallgemeinerten Variablen.
- Unter welchen Bedingungen indizierte und selektierte Objekte verallgemeinerte Variablen sind, ist in den folgenden Kapiteln beschrieben.

In Norm-Pascal gelten folgende Einschränkungen:

- Es gibt nur Konstanten von einem einfachen Typ
- Als Ergebnistyp von Funktionen ist nur ein einfacher Typ zulässig
- Funktionen mit einem Zeigertyp als Ergebnistyp können nicht dereferenziert werden
- Es können nur Variable einen strukturierten Typ besitzen
- Es können nur Variablen-Objekte (verallgemeinerte Variable) indiziert, selektiert oder dereferenziert werden
- Bei dynamischen Objekten muß das Zeiger-Objekt stets eine Zeiger-Variable sein.

*Querverweise*


| | |
|---|---|
| verallgemeinerte Variable | 7 |
| Variablen-Name | 7.1 |
| dynamische Variable | 7.2 |
| Variablenparameter | 8.5.2 |
| Funktionsaufruf | 8.7 |
| statischer Ausdruck | 9.2 |
| Aggregat | 9.5 |
| Zuweisung | 10.1.2 |
| WITH-Anweisung | 10.5 |
| vordefinierte Funktion | 15 |

## Indizierte Objekte

Eine Komponente eines ARRAY-Objekts wird durch Indizierung ausgewählt. Ein indizierter Zugriff wird nach folgender Syntax gebildet:

```
indiziertes_Objekt
                = ARRAY-Objekt
                  "[" Indexausdruck {"," Indexausdruck} "]"
                | String-Objekt "[" Indexausdruck "]".

Indexausdruck   = Ordinal-Ausdruck.
```

Ein indiziertes Objekt ist eine Komponente eines Objekts von einem ARRAY-Typ oder String-Typ. Der Typ des indizierten Objekts ist der Komponententyp des ARRAY-Objekts bzw. der Typ Char bei einem String-Objekt. Ist das Objekt eine verallgemeinerte Variable, dann ist das indizierte Objekt wieder eine verallgemeinerte Variable.

In Norm-Pascal können nur verallgemeinerte Variable indiziert werden.

Ist der Typ des Objekts ein ARRAY-Typ, so muß der Wert des Indexausdrucks zuweisungsverträglich zu dem entsprechenden Indextyp des ARRAY-Typs sein, der Wert muß also im Wertebereich des Indextyps liegen.

Ist der Typ des Objekts ein String-Typ, dann muß der Indexausdruck einen Integer-Typ besitzen. Der Wert des Indexausdrucks darf nicht kleiner als 1 und nicht größer als die aktuelle Länge des String-Objekts sein.

Ein indiziertes Objekt ist genau dann ein statisches Objekt, wenn das ARRAY-Objekt bzw. das String-Objekt ein statischer Ausdruck ist und wenn alle Indexausdrücke statische Objekte sind.

### Abkürzende Schreibweise bei mehrdimensionalen ARRAY-Objekten

Die Folge "[]" in der Langform wird in der Kurzform durch ein Komma ersetzt. Die Kurzform und die Langform sind äquivalent.

*Beispiel*

Gegeben seien folgende Deklarationen:

```
TYPE
   t1  = Integer;
   t2  = ARRAY ['A'..'Z'] OF t1;
   t3  = ARRAY ['0'..'9'] OF t2;
VAR
   v1  : t1;
   v3  : t3;
```

Dann sind die beiden folgenden Zuweisungen äquivalent:

```
v1 := v3['0']['A'] ;
v1 := v3['0', 'A'] ;
```

### Implementierungsabhängige Eigenschaft

Die Reihenfolge der Auswertung von Index-Ausdrücken in einem indizierten Objekt ist implementierungsabhängig.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Index_Error | - Bei einem indizierten *ARRAY*-Objekt liegt der Wert des Indexausdrucks nicht im Wertebereich des Indextyps des Typs des *ARRAY*-Objekts.<br><br>- Bei einem indizierten *String*-Objekt ist der Wert des Indexausdrucks kleiner als 1 oder größer als die aktuelle Länge des *String*-Objekts. |
| undefinierte Auswirkungen | - Bei einem indizierten *ARRAY*- oder *String*-Objekt wurde das *ARRAY*- bzw. *String*-Objekt verkürzt durch den Aufruf von New(p,e) oder New (p, c1, .., cn, e) erzeugt und der Wert des Indexausdrucks im indizierten Objekt ist größer als e.<br><br>- In einem indizierten *String*-Objekt ist der Wert des *String*-Objekts nicht definiert (unabhängig davon, ob das indizierte Objekt in einem Ausdruck oder z. B. als verallgemeinerte Variable auf der linken Seite einer Zuweisung auftritt).<br><br>- Die Länge einer String-Variablen wird verändert, obwohl noch eine Referenz auf eine Komponente der String-Variablen existiert. |

*Hinweise*

- String-Variablen sollten nur in Ausnahmefällen indiziert benutzt werden. Indizierte Zugriffe außerhalb der aktuellen Länge (aber durchaus innerhalb der vereinbarten Maximallänge) sind fehlerhaft und lösen bei eingeschalteter Check-Option den Fehler Index_Error aus. Aus diesem Grund sollten soweit wie möglich die vordefinierten Unterprogramme zur Zeichenkettenverarbeitung (siehe 15.3) verwendet werden.
- Es ist besonders zu beachten, daß der Wert einer String- Variablen (und damit ihre aktuelle Länge) definiert sein muß, also eine Zuweisung an die (ganze) String-Variable erfolgt sein muß, ehe auf ihre einzelnen Komponenten (lesend oder schreibend) indiziert zugegriffen werden kann.
  Durch Zuweisung von Zeichen zu einzelnen Komponenten einer String-Variablen ändert sich die Länge dieser String-Variablen nicht.

*Beispiele*

```
TYPE
   byte     = 0 .. 255;
   hex_pair = PACKED ARRAY [1..2] OF Char;


FUNCTION hex (i: byte) : hex_pair;
CONST
   hex_base = 16;
TYPE
   tab      = PACKED ARRAY [0 .. hex_base - 1] OF Char;
CONST
   hex_tab  = tab ('0','1','2','3','4','5','6','7',
                   '8','9','A','B','C','D','E','F');

BEGIN
   hex := hex_pair (hex_tab [i DIV hex_base],  ———————————————————— (1)
                    hex_tab [i MOD hex_base]);
END;


VAR
   i   : byte;
   pair: hex_pair;
   ch,
   ch2 : Char;
   ptr : ↑hex_pair;


BEGIN
   Read (i);
   pair     := hex (i);
   ch       := pair[1];  ——————————————————————————————————————————— (2)
   ch2      := hex (i) [1];  ——————————————————————————————————————— (3)
   New (ptr);
   ptr↑ [1] := ch;  ————————————————————————————————————————————————— (4)
END
```

(1) Die Konstante hex_tab ist von einem ARRAY-Typ und kann daher indiziert werden.

(2) Die Variable pair ist von einem ARRAY-Typ und kann daher indiziert werden.

(3) Das Ergebnis des Funktionsaufrufs hex (i) ist von einem ARRAY-Typ und kann daher indiziert werden.

(4) Die dynamische Variable ptr↑ ist von einem ARRAY-Typ und kann daher indiziert werden.

Da auch das Aggregat von einem ARRAY-Typ ist und daher indiziert werden kann, hätte die Deklaration der Konstanten hex_tab auch entfallen und überall durch das Aggregat ersetzt werden können, also z.B.

    hex_tab [i DIV hex_base]

durch das (recht komplizierte) Objekt

    tab ('0','1','2','3','4','5','6','7',
           '8','9','A','B','C','D','E','F') [i DIV hex_base]

*Querverweise*

| | |
|---|---|
| Komponenten-Typ: | 6.3 |
| ARRAY-Typ: | 6.3.1 |
| Indextyp: | 6.3.1 |
| String-Typ | 6.3.2.2 |
| Zuweisungsverträglichkeit: | 6.6.3 |
| Ausdruck: | 9 |
| statischer Ausdruck: | 9.2 |
| dynamische Variable: | 9.6.4 |
| Zuweisung: | 10.1.2 |
| New: | 15.2 |

## Selektierte Objekte

Eine Komponente eines RECORD-Objekts wird durch Selektion (Feldauswahl) ausgewählt. Ein selektierter Zugriff wird nach folgender Syntax gebildet:

```
selektiertes_Objekt
                = RECORD-Objekt "." Feld-Bezeichner
                | Feldauswahlbezeichner.

Feldauswahlbezeichner
                = Feld-Bezeichner.
```

Ein selektiertes Objekt ist eine Komponente eines Objekts von einem RECORD-Typ (ein RECORD-Objekt). Die Komponente wird entweder durch Angabe eines RECORD-Objekts und des Feld-Bezeichners oder durch einen Feldauswahlbezeichner angegeben. Die Angabe eines Feldauswahlbezeichners ist nur innerhalb einer WITH-Anweisung (siehe 10.5) möglich.

Ist das RECORD-Objekt eine verallgemeinerte Variable, dann ist auch das selektierte Objekt wieder eine verallgemeinerte Variable. Der Typ des selektierten Objekts ist der Typ der durch den Feld-Bezeichner bzw. Feldauswahlbezeichner bezeichneten Komponente des RECORD-Objekts.

Ist das RECORD-Objekt ein statischer Ausdruck, dann ist auch das selektierte Objekt ein statischer Ausdruck.

Auf Felder im Variantteil kann nur zugegriffen werden, wenn die entsprechende Variante eingestellt (aktiv) ist. Die Werte aller Komponenten aller Varianten sind vollständig undefiniert, solange noch keine Variante eingestellt wurde (siehe auch 7.3). Wird eine neue Variante eingestellt, so sind alle Werte von Komponenten der alten Variante verloren und können auch durch erneutes Einstellen der alten Variante nicht wiedergewonnen werden.

- **Kennungsfeld vorhanden**

  Übertragen eines gültigen Wertes auf das Kennungsfeld (tagfield) stellt die mit dem Wert verknüpfte Variante ein. Ab diesem Zeitpunkt kann man auf die zu dieser Variante gehörenden Komponenten zugreifen. Die Werte aller Komponenten sind zu diesem Zeitpunkt aber noch vollständig undefiniert falls die Variante unmittelbar davor nicht schon eingestellt war.

  Wenn das Kennungsfeld keinen definierten Wert hat, ist keine Variante eingeschaltet und alle Komponenten-Werte aller Varianten sind vollständig undefiniert.

- **Kein Kennungsfeld vorhanden**

  Der Zugriff (lesend oder schreibend) auf eine Komponente einer Variante stellt diese ein. Die Komponenten dieser Variante sind in diesem Zeitpunkt noch vollständig undefiniert, falls die Variante unmittelbar davor nicht eingestellt war. Damit führt das Umstellen einer Varianten durch einen lesenden Zugriff auf eine Komponente zu dem Fehler, daß ein undefinierter Wert gelesen wird.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Variant_Error | - Es wird auf eine nicht aktive Variante eines *RECORD*-Objekts zugegriffen, obwohl die Variante ein Kennungsfeld besitzt. |
| undefinierte Auswirkungen | - Die Variante einer *RECORD*-Variablen ist nicht für die Gesamtdauer jeglicher Referenz auf jede ihrer Komponenten aktiv.<br>- In einer mit New(p, c1, .., cn) oder New(p, c1, .., cn, e) erzeugten dynamischen Variable wird eine andere Variante eingestellt, als die durch Selektorkonstanten c1 bis cn bestimmte. |

*Hinweise*

- Die vielfach geübte Praxis, den Komponenten einer Variante Werte zuzuweisen, danach eine andere Variante einzustellen und die Werte der Komponenten (unter anderen Typen) wieder zu lesen, ist fehlerhaft. Derartige Typkonversionen sollten (wenn überhaupt nötig) mit Hilfe der vordefinierten Funktion Convert (siehe 15.10) erfolgen.
- Der Zugriff auf eine nicht aktive Variante kann mit der Check-Option nur dann erkannt werden, wenn für diesen Variantteil ein Kennungsfeld existiert.

*Beispiel*

```
VAR
   person : RECORD
              name,
              vorname    : PACKED ARRAY [1..20] OF Char;
              alter      : 0..100;
              geburtstag: RECORD
                            jahr : 0..2000;
                            monat: 1..12;
                            tag  : 1..31;
                          END;
            END;
```

Die folgenden selektierten Objekte sind Komponentenvariablen von person:

`person.name, person.vorname, person.alter und person.geburtstag.`

jahr, monat und tag sind wiederum Komponenten von person.geburtstag, nicht aber von person. Diese werden wie folgt bezeichnet:

```
person.geburtstag.jahr
person.geburtstag.monat
person.geburtstag.tag
```

*Beispiel für ein Objekt, das keine verallgemeinerte Variable ist*

```
TYPE
   complex = RECORD
               real_teil,
               imag_teil : Real
             END;

FUNCTION add_complex (x,y : complex) : complex;
BEGIN
  add_complex := complex (x.real_teil + y.real_teil,
                          x.imag_teil + y.imag_teil);
END;

VAR
   x,y : complex;
   r   : Real;

BEGIN
    ...
    r := add_complex (x,y).real_teil; ——————————— (1)
    ...
END
```

(1) Der Funktionsaufruf add_complex (x, y) liefert als Ergebnis einen Wert] eines RECORD-Typs. Das selektierte Objekt add_complex (x, y).real_teil ist keine verallgemeinerte Variable, da es Komponente eines Funktionsergebnisses und nicht Komponente einer Variablen ist.

*Querverweise*

| | |
|---|---|
| RECORD-Typ: | 6.3.3 |
| Definierte Variable: | 7.3 |
| WITH-Anweisung: | 10.5 |
| Sichtbarkeit: | 12 |
| Convert: | 15.10 |
| Check-Option: | 16 |

## Dereferenzierte Objekte

```
dereferenziertes_Objekt = Zeiger-Objekt "↑" .
```

Falls der Wert des Zeiger-Objekts in einem dereferenzierten Objekt definiert und ungleich NIL ist, also ein Verweis auf eine dynamische Variable ist, so steht das dereferenzierte Objekt für diese dynamische Variable.

Das Zeiger-Objekt kann in Pascal-XT eine Konstante, eine Variable, eine Aggregatkomponente oder ein Funktionsaufruf sein. Das dereferenzierte Objekt ist aber stets eine verallgemeinerte Variable.

In Norm-Pascal sind als Zeiger-Objekte hingegen nur verallgemeinerte Variablen zugelassen.

Der Typ des Zeiger-Objekts muß ein Zeigertyp sein, der weder der generische Zeigertyp (siehe 6.5.2) noch ein privater Zeigertyp eines fremden Paketes (siehe 11.2) ist. Der Typ des dereferenzierten Objekts ist der Domänentyp des Typs des Zeiger-Objekts.

Die Dereferenzierung eines Zeiger-Objekts ist in keinem Fall ein statischer Ausdruck.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Pointer_Error | - In einem dereferenzierten Objekt ist der Wert des *Zeiger*-Objekts gleich NIL. |
| undefinierte Auswirkungen | - In einem dereferenzierten Objekt ist der Wert des *Zeiger*-Objekts undefiniert.<br>- Eine dynamische Variable, die durch New(p,c1,...,cn), New(p,e) oder New(p,c1,...,cn,e) erzeugt wurde, kommt als Ganzes (d. h. nicht nur einzelne ihrer Komponenten) als Operand in einem Ausdruck vor. |

*Hinweise*

– Es sei p eine Zeiger-Variable. Eine Referenz auf die dynamische Variable p↑ wird bei einer Variablenparameterübergabe, in einer WITH-Anweisung (WITH p↑ DO ...) oder bei einer Zuweisung (p↑ := ...) hergestellt. In Unterprogrammen mit Parameter p↑ oder innerhalb von WITH-Anweisungen darf kein Dispose (p) ausgeführt werden. In Zuweisungen darf auf der rechten Seite keine Funktion stehen, die als Seiteneffekt ein Dispose (p) ausführt.

– Eine dynamische Variable, die verkürzt auf der Halde angelegt wurde, darf nicht als Ganzes in einem Ausdruck auftreten, z. B. auf der rechten Seite einer Zuweisung. Dabei würde nämlich auf die Variable in der gesamten, ihren Typ entsprechenden Größe zugegeriffen, werden, also auch auf die Bereiche, die der Variablen wegen der verkürzten Anlage gar nicht zugeordnet wurden.
  Diese Einschränkung gilt nicht für dynamische Variable von String-Typen, da diese die aktuelle Länge immer mitführen.

*Beispiele*

Zwei große Beispiele sind in 18.3 angegeben.

```
p↑.vorname               { einfache Dereferenzierung }

p↑.ehegatte↑.vorname     { Dereferenzierung eines selektierten Objekts }

f(x,y,z)↑.vorname        { Dereferenzierung eines Funktionsaufrufes }
```

*Querverweise*

| | |
|---|---|
| Zeigertypen: | 6.4 |
| NIL: | 6.4 |
| generischer Zeigertyp: | 6.5.2 |
| dynamische Variable: | 7.2 |
| Funktionen: | 8.2 |
| privater Zeigertyp: | 11.4 |
| New, Dispose, Release: | 15.2 |

## Puffervariable

```
Puffervariable  = FILE-Objekt "↑".
```

Ein FILE-Objekt (FILE-Variable) ist eine verallgemeinerte Variable eines FILE-Typs. Zu jeder FILE-Variablen wird implizit eine Puffervariable f↑ vereinbart. Eine Puffervariable, die zu einer Textdatei gehört, besitzt den Typ Char, ansonsten ist der Typ der Puffervariablen der Komponententyp des FILE-Typs.

Der Typ des FILE-Objekts darf nicht der generische FILE-Typ Any_File sein.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| undefinierte Auswirkungen | - Der Dateizeiger einer Dateivariablen f wird (z. B. durch Lesen oder Schreiben) geändert, obwohl noch eine Referenz auf die Puffervariable f↑ existiert. |

*Hinweis*

Variable vom generischen FILE-Typ Any_File können nicht deklariert werden und besitzen daher auch keine Puffervariable.

Beispiele für Zugriffe auf Puffervariablen im Zusammenhang mit Read und Write sind in Kapitel 15.1 und 19 angegeben.

Solange eine Referenz auf die Puffervariable einer Dateivariablen existiert, darf der Wert der Dateivariablen (insbesondere des Dateizeigers) nicht geändert werden, es sei f ist eine FILE-Variable. Eine Referenz auf die Puffervariable f↑ wird bei einer Variablenparameterübergabe (p (.., f↑, ..)), in einer WITH-Anweisung (WITH f↑ DO ...) oder bei einer Zuweisung (f↑ := ...) hergestellt. Der Wert der FILE-Variablen f wird z. B. geändert, wenn mit Put (f) oder Get (f) (und damit auch Read und Write) innerhalb des gerufenen Unterprogramms oder der WITH-Anweisung der Dateizeiger weitergeschaltet wird. Ähnlich könnte auf der rechten Seite einer Zuweisung eine Funktion als Seiteneffekt den Dateizeiger unerlaubt weiterschalten.

*Beispiele*

```
VAR
   i        :  Integer;
   s        :  String;
   daten    :  FILE OF Integer;
   ausgaben:  ARRAY [1..3] OF Text;
   club     :  FILE OF person;          { siehe 6.3.3 }

BEGIN
   ...
   daten↑ := 1;
   i      := daten ↑;

   ausgaben [i] ↑ := 'x';

   club↑.nachname := pascal;
   club↑ := person ('blaise', 'pascal', 50, False);
END
```

*Querverweise*

FILE-Typ: 6.3.5
Textdatei: 6.3.5.2
generischer FILE-Typ: 6.5.1
Puffervariable: 7.3, 15, 16.1
Variablen-Parameter: 8.5.2
WITH-Anweisung: 10.5
Read, Write: 15.1

# Anweisungen

Durch Anweisungen wird der Algorithmus des Programms dargestellt. Vor Anweisungen (auch leeren) kann eine Marke stehen. Anweisungen werden in Pascal durch Semikolons getrennt. Nach einer Anweisung muß nur dann ein Semikolon stehen, wenn ihr eine weitere Anweisung folgt. Die Verbundanweisungen, die bedingten Anweisungen, die Wiederholungsanweisungen und die WITH-Anweisung enthalten jeweils eine oder mehrere innere Anweisungen.

```
Anweisung  = [Marke ":"]
             (   einfache_Anweisung     | bedingte_Anweisung
               | Wiederholungsanweisung | Verbundanweisung
               | WITH-Anweisung ).
```

Eine Anweisung, vor der eine Marke angegeben wurde, kann mit der GOTO-Anweisung angesprungen werden.

| Anweisungsklasse | Anweisungen | Kapitel | Anmerkungen / Beispiele |
|---|---|---|---|
| einfache Anweisungen | Leeranweisung | 10.1.1 | |
| | Zuweisung | 10.1.2 | a := b + c |
| | Prozedudraufruf | 10.1.3 | plus (erg, b, c) |
| | GOTO-Anweisung | 10.1.4 | GOTO 4711 |
| | EXIT-Anweisung | 10.1.5 | Verlassen der umgebenden Wiederholungsanweisung |
| | RETURN-Anweisung | 10.1.6 | Verlassen des umgebenden Blocks (z.B. Unterprogramms) |
| bedingte Anweisungen | IF-Anweisung | 10.3.1 | IF bed THEN ... ELSE ; |
| | CASE-Anweisung | 10.3.2 | Mehrfachauswahl |
| Wiederholungs-anweisungen | REPEAT-Anweisung | 10.4.1 | REPEAT ... UNTIL bed |
| | WHILE-Anweisung | 10.4.2 | WHILE bed DO ... |
| | FOR-Anweisung | 10.4.3 | FOR i:=a1 TO b1 DO ... |
| Verbundanweisung | | 10.2 | BEGIN ... END |
| WITH-Anweisung | | 10.5 | WITH rec_var DO ... |

Tabelle 10-1: Anweisungen

*Querverweise*

Marken: 3.6, 4

# Einfache Anweisungen

Eine einfache Anweisung ist eine Anweisung, die keine weitere Anweisung enthält.

```
einfache_Anweisung  = Leeranweisung   | Zuweisung
                    | Prozeduraufruf | GOTO-Anweisung
                    | EXIT-Anweisung | RETURN-Anweisung.
```

## Leeranweisung

Durch eine Leeranweisung werden keine Aktionen ausgeführt. Sie ist z. B. dann notwendig, wenn eine Sprungmarke vor END gesetzt werden soll.

```
Leeranweisung   =.
```

*Hinweis*

Es ist zwar nicht notwendig, aber durchaus sinnvoll, auch Anweisungen vor END mit einem Semikolon abzuschließen. Die Tatsache, daß dadurch zwischen dem Semikolon und dem END eine Leeranweisung eingefügt wird, hat keinerlei Auswirkungen auf die Code-Generierung.
Es ergeben sich dadurch aber folgende Vorteile:

- alle Anweisungen werden gleichermaßen durch ein Semikolon beendet,
- beim Einfügen einer Anweisung vor dem END muß dann die vorherige Anweisung nicht mehr durch ein Semikolon ergänzt werden.

Aber Vorsicht:
Vor dem ELSE einer IF-Anweisung darf nie ein Semikolon stehen!

*Beispiele*

I := 1;;;;

Im obigen Beispiel steht zwischen je zwei aufeinander folgenden Semikolons eine (natürlich unsichtbare) Leeranweisung (also stehen in obiger Zeile insgesamt drei Leeranweisungen).

```
BEGIN
   I := 5;
   J := 2;
END;
```

In diesem Beispiel steht vor dem Schlüsselwort END eine Leeranweisung (wegen des

Semikolons nach "2", siehe Hinweis).

```
   BEGIN
      ...
      GOTO 13;
      ...
13:END;
```

In obigem Beispiel ist "13:" eine markierte Leeranweisung. Sprungmarken können nämlich nur Anweisungen, nicht aber END markieren.

```
   CASE i OF
      1: Writeln ('Sonderfall 1');
      2: Writeln ('Sonderfall 2');
      ELSE:
   END;
```

Die CASE-Anweisung in diesem Beispiel enthält einen leeren ELSE-Zweig (also einen ELSE-Zweig, der eine Leeranweisung enthält). Damit wird erreicht, daß die Werte von i, die weder 1 noch 2 sind, ignoriert werden (also nicht zu einem Case_Error führen).

```
   FOR i := 1 TO 10000 DO;
```

```
   FOR i := 1 TO 10000 DO BEGIN END;
```

Die beiden obigen FOR-Anweisungen machen jeweils 10000 mal "nichts", da zwischen DO und dem Semikolon bzw. zwischen BEGIN und END je eine Leeranweisung steht. (Ob diese "Leer-Schleifen" wirklich 10000 mal durchlaufen werden, bleibt jeder Pascal-XT-Implementierung (Code-Optimierung) überlassen. Bei Verwendung als "aktive Warteschleifen" ist daher Vorsicht geboten.)

## Zuweisungen

Zuweisungen haben die Form:

```
Zuweisung   = Variablen-Objekt ":=" Ausdruck
            | Funktions-Bezeichner ":=" Ausdruck.
```

Eine Zuweisung überträgt den Wert, der bei der Auswertung des Ausdrucks auf der rechten Seite der Zuweisung ermittelt wird, auf die verallgemeinerte Variable (Variablen-Objekt) bzw. den Funktions-Bezeichner auf der linken Seite der Zuweisung. Der Wert des Ausdrucks muß zuweisungsverträglich (siehe 6.6.3) zum Typ der verallgemeinerten Variablen bzw. des Funktions-Bezeichners sein. Die Zuweisung an den Funktions-Bezeichner muß im zugehörigen Funktions-Block enthalten sein (siehe auch 8.2).

Eine Variable oder ein Funktionsergebnis ist undefiniert, wenn ihr/ihm noch kein Wert zugewiesen wurde (siehe auch 7.3).

### Implementierungsabhängige Eigenschaft

Die Reihenfolge, in der die linke und rechte Seite einer Zuweisung ausgewertet werden, ist implementierungsabhängig. Die bei der Auswertung des Variablen-Objekt (linke Seite) ermittelte Referenz bleibt während der ganzen Abarbeitung der Anweisung erhalten. Dies ist insbesondere im Zusammenhang mit möglichen Laufzeitfehlern von Bedeutung (siehe 9.6.2, 9.6.3, 9.6.4, 9.6.5).

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Range_Error | - In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) von einem Ordinal-Typ nicht im Wertebereich des *Ordinal*-Typs der verallgemeinerten) Variablen bzw. des *Funktions*-Bezeichners (linke Seite). |
| Numeric_Error | - In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) vom Typ Long_Real nicht im Wertebereich der verallgemeinerten Variablen bzw. des *Funktions*-Bezeichners (linke Seite) vom Typ Short_Real. |
| Set_Error | - In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) von einem SET-Typ nicht im Wertebereich des SET-Typs der verallgemeinerten Variablen bzw. des *Funktions*-Bezeichners (linke Seite). |
| String_Error | - In einer Zuweisung ist die aktuelle Länge des Zeichenkettenwertes des Ausdrucks (rechte Seite) größer als die Maximallänge des String-Typs der verallgemeinerten Variablen bzw. des *Funktions*-Bezeichners (linke Seite). |
| | - In einer Zuweisung ist die aktuelle Länge des Zeichenketten-Ausdrucks von einem String-Typ (rechte Seite) ungleich der Länge des Zeichenkettentyps fester Länge der verallgemeinerten Variablen bzw. des *Funktions*-Bezeichners (linke Seite). |
| undefinierte Auswirkungen | - In einer Zuweisung ist der Typ des Ausdrucks (rechte Seite) vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamische Variable, deren Typ vom Domänentyp des Typs der verallgemeinerten Variablen bzw. des *Funktions*-Bezeichners (linke Seite) verschieden ist. |

*Beispiele für Zuweisungen an Variable*

```
i      := 0;
a [50] := 100;
feld [x + y] := pi;
b := (1 < i) AND (i < 100);
p↑.alter := 3;
```

*Beispiel für Zuweisungen an Funktions-Bezeichner*

```
FUNCTION max (x, y : Integer);
BEGIN
   IF x > y THEN max := x ELSE max := y;
END;
```

*Hinweise*

- Bei Funktionen mit einem strukturierten Ergebnistyp muß dem Funktions-Bezeichner als Ganzes ein Wert zugewiesen werden. Komponentenweise Zuweisung ist nur mit Aggregaten möglich (siehe 9.5).
- Die Zuweisung eines Wertes an einen Funktions-Bezeichner sollte unmittelbar im Anweisungsteil der Funktion stehen. Die ebenfalls mögliche Zuweisung innerhalb einer eingeschachtelten Funktion oder Prozedur ist weniger übersichtlich.
- Die implementierungsabhängige Reihenfolge des Zugriffs auf die verallgemeinerte Variable und die Auswertung des Ausdrucks kann zu unterschiedlichen Ergebnissen führen.
- Bei einer Zuweisung kann ein Numeric_Error nur dann auftreten, wenn die Typen Short_Real und Long_Real unterschiedliche Wertebereiche besitzen.

*Beispiel für implementierungsabhängige Reihenfolge der Abarbeitung:*

```
VAR
   i : Integer;
   a : ARRAY [1 .. 9] OF Char;

FUNCTION funktion_mit_Seiteneffekt:Char;
BEGIN
   i := 8; { <—— Dies ist der Seiteneffekt }
   funktion_mit_Seiteneffekt := 'X' ;
END;

BEGIN
   i    := 2;
   a[i] := funktion_mit_Seiteneffekt;
END.
```

Im Beispiel ist die Wirkung von der Reihenfolge des Zugriffs auf a[i] und dem Aufruf von funktion_mit_Seiteneffekt abhängig:

Falls zuerst die Referenz auf a[i] hergestellt und dann erst die Funktion aufgerufen wird, so wird das Zeichen 'X' der Komponente a[2] zugewiesen.

Falls hingegen die Funktion zuerst aufgerufen wird, so wird das Zeichen 'X' der Komponente a[8] zugewiesen.

Gemäß 1.4 gelten Programme, deren Wirkung von implementierungsabhängigen Eigenschaften abhängen, als fehlerhaft.

*Querverweise*

| | |
|---|---|
| implementierungsabhängig: | 2.2 |
| Ordinaltyp: | 6.2 |
| Real-Typ: | 6.2.2 |
| Zeichenkettentyp: | 6.3.2 |
| String-Typ: | 6.3.2.2 |
| SET-Typ: | 6.3.4 |
| Zeigertyp: | 6.4 |
| generischer Zeigertyp: | 6.5.2 |
| Zuweisungsverträglichkeit: | 6.6.3 |
| verallgemeinerte Variable: | 7 |
| Funktionsergebnis: | 8.2 |
| Seiteneffekte: | 8.2 |
| Ausdruck: | 9 |
| Aggregate: | 9.5 |
| Objekt: | 9.6 |
| Anweisungsteil: | 12.1 |

## Prozeduraufrufe

Prozeduraufrufe haben die Form:

```
Prozeduraufruf  = Prozedur-Name [Aktualparameterliste].

Aktualparameterliste
                = "(" Aktualparameter {"," Aktualparameter} ")".
```

Ein Prozeduraufruf veranlaßt die Ausführung des Blockes der aufgerufenen Prozedur (siehe 8.7).

Wenn die Prozedur Formalparameter hat, dann muß der Prozeduraufruf eine Aktualparameterliste mit den Aktualparametern enthalten.
Durch die Angabe eines Paket-Bezeichners im Prozedur-Namen wird eine Prozedur des entsprechenden Pakets aufgerufen.

Bei einem Prozeduraufruf können die in 8.5.1, 8.5.4 und 8.7 beschriebenen Laufzeitfehler bei der Parameterübergabe auftreten. Ferner können während der Abarbeitung der Anweisungen des Prozedur-Blocks Laufzeitfehler auftreten und (falls sie in der Prozedur nicht behandelt werden, siehe 14) and die Aufrufstelle propagiert werden.

*Beispiel*

```
proz1;

trans (a, x, y);

numerics.bisect (fct (x + y), 2.0, 100, y);

Writeln ('Das Ergebnis ist ', a * b :10:5);
```

*Querverweise*

| | |
|---|---|
| Aktual- und Formalparameter: | 8.5 |
| Unterprogrammaufruf: | 8.7 |
| Block: | 12.1 |
| Ausnahmebehandlung: | 14 |

## GOTO-Anweisung

Die GOTO-Anweisung hat die Form:

```
GOTO-Anweisung  = "GOTO" Marke.
```

Eine GOTO-Anweisung veranlaßt die Fortsetzung der Programmausführung an dem Punkt, der durch die Marke in der GOTO-Anweisung bezeichnet ist.

Die Verwendung einer Marke in einer GOTO-Anweisung ist nur zulässig, wenn die Marke in einem Markendeklarationsteil vereinbart wurde und wenn die Marke vor einer Anweisung auf gleichem oder höherem Anweisungsniveau steht. Es ist also verboten, von außen in das Innere eines Unterprogramms oder einer strukturierten Anweisung hineinzuspringen. Ebensowenig darf innerhalb einer strukturierten Anweisung mit Alternativen (IF, CASE) über die Grenze(n) von Alternativ-Zweigen gesprungen werden.

*Beispiel*

```
GOTO 4711;
GOTO 1;
```

*Hinweis*

GOTO-Anweisungen sollten nur restriktiv eingesetzt werden. Die meisten Algorithmen lassen sich auch ohne Verwendung von GOTO-Anweisungen elegant programmieren, da die bedingten, die Wiederholungs-, die EXIT- und die RETURN-Anweisung zur strukturierten Steuerung des Programmflusses zur Verfügung stehen.

*Querverweise*

Marken: 3.6
Markendeklarationsteil: 4
Marken vor Anweisungen: 10
Block: 12.1

## EXIT-Anweisung

```
EXIT-Anweisung  = "EXIT".
```

Sie darf nur innerhalb einer Wiederholungsanweisung (also einer WHILE-, REPEAT- oder FOR-Anweisung) vorkommen. Bei geschachtelten Wiederholungsanweisungen wird nur die innerste Wiederholungsanweisung beendet, die die EXIT-Anweisung unmittelbar enthält.

*Beispiele*

```
FOR i:= 1 TO n DO BEGIN
   IF a[i] = 0 THEN EXIT;
   s := s + a[i];
   END;
Writeln (s);
```

In diesem Beispiel werden die Werte der Komponenten des ARRAYs a bis zur ersten Komponente mit dem Wert 0 aufsummiert.

Im folgenden Beispiel werden in jeder Eingabezeile die Zeichen einer Zeile bis zum ersten Semikolon verarbeitet, die restlichen Zeichen der Zeile werden ignoriert.

```
Reset;
WHILE NOT Eof DO BEGIN
   WHILE NOT Eoln DO BEGIN
      Read (c);
      IF c = ';' THEN EXIT;
      verarbeitung (c);
      END;
   Readln;
   END;
```

*Querverweise*

Wiederholungsanweisung: 10.4

## RETURN-Anweisung

```
RETURN-Anweisung = "RETURN" .
```

Eine RETURN-Anweisung bewirkt die sofortige Beendigung der Ausführung des Blocks, in dem sie unmittelbar enthalten ist. Eine RETURN-Anweisung im Anweisungsteil des Hauptprogramm-Blocks führt zur Programmbeendigung.

*Hinweis*

Vor dem Verlassen eines Funktions-Blocks muß dem Funktions-Bezeichner ein Wert zugewiesen werden, ansonsten ist das Funktionsergebnis undefiniert.

*Beispiel*

```
FUNCTION finde (x : Integer) : Integer;
VAR i : Integer;
BEGIN
   FOR i := 1 TO 100 DO
      IF a[i] = x THEN BEGIN
         finde := i;
         RETURN; {gefunden}
         END;
   finde := 0; {nicht gefunden}
END;
```

In diesen Beispiel wird (im Gegensatz zu EXIT) bei RETURN nicht nur die FOR-Anweisung beendet sondern auch die Ausführung des Funktionsblocks. Damit wird die Anweisung finde := 0; nicht mehr ausgeführt und das Funktionsergebnis ist finde := i, der gefundene Index.

*Querverweise*

Funktion: 8.2
Block: 12.1

# Verbundanweisung

```
Verbundanweisung
                = "BEGIN" Anweisungsfolge [EXCEPTION-Teil] "END".

EXCEPTION-Teil  = "EXCEPTION" Anweisungsfolge.

Anweisungsfolge = Anweisung {";" Anweisung}.
```

Mit einer Verbundanweisung werden mehrere Anweisungen syntaktisch zu einer Anweisung zusammengefaßt. Das ist überall dort notwendig, wo nur eine Anweisung stehen darf, aber eine längere Anweisungsfolge erforderlich ist (z. B. in bedingten Anweisungen). Die Anweisungen der Anweisungsfolge werden im Normalfall in der Reihenfolge der Aufschreibung ausgeführt.

Diese Abarbeitung der Anweisungsfolge kann durch die Ausführung einer GOTO-Anweisung (siehe 10.1.4), einer EXIT-Anweisung (siehe 10.1.5), einer RETURN-Anweisung (siehe 10.1.6), dem Aufruf der vordefinierten Prozedur Raise (siehe 15.11), oder dem Auftreten einer Ausnahmesituation abgebrochen werden.

Innerhalb einer Verbundanweisung kann ein Ausnahme-Behandlungs-Teil (EXCEPTION-Teil) definiert werden. In diesem Teil können Ausnahmen behandelt werden, die während der Abarbeitung der Anweisungsfolge der Verbundanweisung auftreten können (siehe 14).

Die Anweisungsfolge im EXCEPTION-Teil, sofern dieser vorhanden ist, wird nur beim Auftreten einer Ausnahmesituation ausgeführt.

*Beispiele*

```
BEGIN
   z := x;
   x := y;
   y := z;
END;

BEGIN
   erg := a * b;
EXCEPTION
   Writeln ('Überlauf');
   erg := 0;
END;
```

*Querverweise*

| | |
|---|---|
| GOTO-Anweisung: | 10.1.4 |
| EXIT-Anweisung: | 10.1.5 |
| RETURN-Anweisung: | 10.1.6 |
| Ausnahmebehandlung: | 14 |
| Raise: | 15.11 |

# Bedingte Anweisungen

Bedingte Anweisungen haben die Form:

```
bedingte_Anweisung  = IF-Anweisung | CASE-Anweisung.
```

Die bedingten Anweisungen umfassen die IF-Anweisungen und die CASE-Anweisungen. Abhängig vom Wert eines Ausdrucks werden bestimmte Anweisungen im Inneren der bedingten Anweisung ausgeführt oder übergangen.

## IF-Anweisung

IF-Anweisungen können mit oder ohne ELSE-Teil geschrieben werden:

```
IF-Anweisung   = "IF" boolescher-Ausdruck "THEN" Anweisung
                 [ELSE-Teil].

ELSE-Teil      = "ELSE" Anweisung.
```

Bei der Abarbeitung der IF-Anweisung wird als erstes der boolesche Ausdruck ausgewertet. Hat er den Wert True, so wird die Anweisung hinter THEN, andernfalls die des ELSE-Teils (falls vorhanden), ausgeführt.

Sollen anstatt jeweils einer Anweisung ganze Anweisungsfolgen abgearbeitet werden, so sind sie mit einer Verbundanweisung zusammenzufassen.

IF-Anweisungen können beliebig ineinander geschachtelt werden. Folgt auf die Anweisung hinter THEN ein ELSE, so wird es immer der innersten, noch nicht abgeschlossenen IF-Anweisung zugeordnet, die noch keinen ELSE-Teil besitzt (siehe Beispiel).

Vor ELSE darf auf keinen Fall ein Semikolon stehen. Ein Semikolon hinter der Anweisung nach dem Schlüsselwort THEN beendet die IF-Anweisung.

*Beispiele*

```
IF a > maximum THEN
   maximum := a;

IF a > maximum THEN
   maximum := a
ELSE IF a < minimum THEN
   minimum := a;

IF j = 0 THEN
   IF i = 0 THEN
      Writeln ('undefiniert')
   ELSE
      Writeln ('unendlich')
ELSE
   Writeln (i / j);

IF a = b THEN
   IF c = d THEN
      x := x + 1
   ELSE
      y := y + 1
```

Im obigen Beispiel gehört der ELSE-Teil zur inneren IF-Anweisung (IF c = d ...), d.h. die Anweisung y := y + 1 wird ausgeführt, wenn a = b und c <> d gilt.

Wenn der ELSE-Teil hingegen zur äußeren IF-Anweisung (IF a = b ...) gehören soll, d.h. wenn die Anweisung y := y + 1 bei a <> b ausgeführt werden soll, so muß eine der beiden folgenden Schreibweisen verwendet werden.

```
IF a = b THEN BEGIN
   IF c = d THEN
      x := x + 1
   END
ELSE
   y := y + 1
```

Hier wird die innere IF-Anweisung durch das Ende der umgebenden Verbundanweisung (BEGIN ... END) abgeschlossen, sodaß nur mehr die äußere IF-Anweisung "offen" ist und der ELSE-Teil daher ihr zugeordnet wird.

```
IF a = b THEN
   IF c = d THEN
      x := x + 1
   ELSE
      {in diesem Fall geschieht nichts,}
      {hier steht eine Leeranweisung}
ELSE
   y := y + 1
```

Hier wird die innere IF-Anweisung durch einen eigenen ELSE-Zweig (der eine Leeranweisung enthält) abgeschlossen. Der zweite ELSE-Zweig (mit der Anweisung y := y + 1) wird daher der äußeren IF-Anweisung zugeordnet.

Für die Wirkung einer IF-Anweisung ist es (anders als bei einer CASE-Anweisung!) gleichwertig, ob sie einen ELSE-Teil mit einer Leeranweisung oder gar keinen ELSE-Teil enthält. Daß ein "leerer" ELSE-Teil aber für den Kontext wichtig sein kann, zeigt das obige Beispiel.

*Querverweise*


Boolean: 6.2.4
Ausdruck: 9
Leeranweisung: 10.1.1
Verbundanweisung: 10.1.2

## CASE-Anweisung

CASE-Anweisungen haben die Form:

```
CASE-Anweisung  = "CASE" Fallindex "OF" Fall-Liste [";"] "END".

Fallindex       = Ordinal-Ausdruck.

Fall-Liste      = Fall-Listenelement {";" Fall-Listenelement}.

Fall-Listenelement
                = Auswahlliste ":" Anweisung.

Auswahlliste    = Auswahl {"," Auswahl} | "ELSE".

Auswahl         = Fallkonstante [".." Fallkonstante] .

Fallkonstante   = Ordinal-Konstante.
```

Mit der CASE-Anweisung wird durch einen Ausdruck eine Anweisung aus einer Liste alternativer Anweisungen ausgeführt.

Der Ausdruck des Fallindex muß von einem Ordinaltyp sein. Die Fallkonstanten der Auswahllisten müssen Werte dieses Ordinaltyps sein.
Eine Auswahl der Form a..b steht für eine Aufzählung aller Konstanten k mit $a \leq k \leq b$. Die einzelnen Werte bzw. Wertebereiche müssen paarweise verschieden sein.

Die Auswahlliste des letzten Fall-Listenelements kann das Wortsymbol ELSE sein. Dieses Fall-Listenelement nennt man ELSE-Teil. ELSE steht dann für alle die Werte des Fallindex-Typs, die nicht in den zuvor aufgeführten Auswahllisten aufgetreten sind. Falls alle Werte des Fallindex-Typs in den Auswahllisten auftreten, so ist der ELSE-Teil wirkungslos.

Bei Eintritt in die CASE-Anweisung wird zunächst der Wert des Fallindex berechnet. Ist er in einer Auswahlliste enthalten, dann wird die zugehörige Anweisung ausgeführt. Falls der Wert in keiner Auswahlliste enthalten ist, dann wird die Anweisung des ELSE-Teils ausgeführt, sofern dieser vorhanden ist.

*Hinweis*

In der CASE-Anweisung steht im Gegensatz zur IF-Anweisung vor dem ELSE ein Semikolon.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Case_Error | - In einer CASE-Anweisung entspricht keine Fallkonstante dem Wert des Fallindex, und es ist auch keine Else-Teil angegeben. |

*Beispiele*

```
CASE zeichen OF
   '0'..'9': z := Ord (zeichen) - Ord ('0');
   '+'     : z := x + y;
   '-'     : z := x - y;
   '*'     : z := x * y;
   END;
```

Es tritt ein Laufzeitfehler (Case_Error) auf, wenn "zeichen" einen anderen Wert als '+', '-', '*' oder eine Ziffer annimmt.

```
CASE zeichen OF
   '0'..'9': z := Ord (zeichen) - Ord ('0');
   '+'     : z := x + y;
   '-'     : z := x - y;
   '*'     : z := x * y;
   ELSE    : Writeln ('falsche Eingabe');
END;
```

Für alle Werte von "zeichen", die nicht '+', '-', '*' oder eine Ziffer sind, wird die Anweisung im ELSE-Teil ausgeführt.

```
CASE zeichen OF
   '0'..'9': ...;
   ...:      ...;
   '*':      ...;
   ELSE:     ; {in diesem Fall geschieht nichts}
   END;
```

Alle Werte von "zeichen", die nicht '+', '-', '*' oder eine Ziffer sind, werden ignoriert (da der ELSE-Teil eine Leeranweisung enthält).

Der Vergleich des ersten Beispiels mit den letzten zeigt, daß es für die Wirkung einer CASE-Anweisung (anders als bei einer IF-Anweisung) keinesfalls gleichwertig ist, ob sie einen ELSE-Teil mit einer Leeranweisung oder gar keinen ELSE-Teil enthält. "Falsche" Werte des Fallindex lösen in einem Fall keine Aktion aus, führen im anderen Fall hingegen zu einem Laufzeitfehler (wird bei Check=On als Case_Error erkannt).

*Querverweise*

Konstante: 5
Ausdruck: 9
Ordinalausdruck: 9
Leeranweisung: 10.1.1

# Wiederholungsanweisungen

Wiederholungsanweisungen sind:

```
Wiederholungsanweisung  = REPEAT-Anweisung
                        | WHILE-Anweisung
                        | FOR-Anweisung.
```

Wiederholungsanweisungen veranlassen die wiederholte Ausführung der in der Wiederholungsanweisung enthaltenen ("inneren") Anweisung. Die REPEAT-Anweisungen und die WHILE-Anweisungen sind bedingungsgesteuerte Wiederholungsanweisungen. FOR-Anweisungen werden mittels einer Zählvariablen gesteuert, deren Anfangs- und Endwert man festlegt.
Wiederholungsanweisungen können mit der EXIT-Anweisung abgebrochen werden.

## REPEAT-Anweisung

REPEAT-Anweisungen haben die Form:

```
REPEAT-Anweisung  = "REPEAT" Anweisungsfolge
                    "UNTIL" boolescher-Ausdruck.
```

Die Anweisungsfolge in der REPEAT-Anweisung wird so lange wiederholt, bis die Auswertung des booleschen Ausdrucks hinter UNTIL den Wert True ergibt. Die Anweisungsfolge wird mindestens einmal abgearbeitet, da die Bedingung erst nach Abarbeitung der Anweisungsfolge ausgewertet wird.

Diese Abarbeitung der Anweisungsfolge kann durch die Ausführung einer GOTO-Anweisung (siehe 10.1.4), einer EXIT-Anweisung (siehe 10.1.5), einer RETURN-Anweisung (siehe 10.1.6) den Aufruf der vordefinierten Prozedur Raise (siehe 15.11) oder das Auftreten einer Ausnahmesituation abgebrochen werden.

*Beispiele*

Das folgende Beispiel realisiert den Euklid'schen Algorithmus zur Berechnung des größten gemeinsamen Teilers (GGT) zweier ganzer Zahlen i und j mittels des Modulo-Operators (siehe 6.3.1). Der "Divisor" von MOD darf niemals kleiner gleich Null sein, was zu Beginn dieser Anweisungsfolge sichergestellt sein muß. Nach dem letzten Durchlauf steht in i der GGT der Ausgangswerte i und j.

```
{Voraussetzung: j > 0}
REPEAT
   k := i MOD j;
   i := j;
   j := k;
UNTIL j = 0;
```

Die REPEAT-Anweisung läßt sich gut zur Verwirklichung eines benutzergesteuerten Bearbeitungsendes in einem Zyklus verwenden, etwa so:

```
REPEAT
   { Anfang der Bearbeitung }
   ...
   { Ende der Bearbeitung }
   Writeln('Weiter? J/N');
   Readln;
   Read(zeichen);
UNTIL zeichen IN ['N','n'];
```

Wenn von vornherein noch nicht klar ist, ob überhaupt eine Bearbeitung stattfinden soll, so ist die WHILE-Anweisung besser geeignet.

*Querverweise*


| | |
|---|---|
| Boolean: | 6.2.4 |
| Ausdruck: | 9 |
| GOTO-Anweisung: | 10.1.4 |
| EXIT-Anweisung: | 10.1.5 |
| RETURN-Anweisung: | 10.1.6 |
| Anweisungsfolge: | 10.2 |
| Ausnahmebehandlung: | 14 |
| Raise: | 15.11 |

## WHILE-Anweisung

WHILE-Anweisungen haben die Form:

```
WHILE-Anweisung = "WHILE" boolescher-Ausdruck "DO" Anweisung.
```

Falls der boolesche Ausdruck in einer WHILE-Anweisung den Wert True ergibt, wird die in ihr enthaltene ("innere") Anweisung abgearbeitet und solange wiederholt, bis der boolesche Ausdruck den Wert False ergibt. Falls eine Folge von Anweisungen wiederholt werden soll, so muß diese in einer Verbundanweisung zusammengefaßt werden.

Diese Abarbeitung der Anweisungsfolge kann durch die Ausführung einer GOTO-Anweisung (siehe 10.1.4), einer EXIT-Anweisung (siehe 10.1.5), einer RETURN-Anweisung (siehe 10.1.6) den Aufruf der vordefinierten Prozedur Raise (siehe 15.11) oder das Auftreten einer Ausnahmesituation abgebrochen werden.

*Beispiel*

Das folgende Beispiel realisiert den Euklidschen Algorithmus, wie er schon bei der REPEAT-Anweisung (siehe 10.4.1) vorgestellt wurde.

```
WHILE j > 0 DO BEGIN
   k := i mod j;
   i := j;
   j := k;
   END;
```

Im folgenden Beispiel werden von jeder Eingabezeile der Datei f zwei Zahlen gelesen und verarbeitet, bis das Ende der Eingabedatei erreicht ist. Aufgrund der Verwendung der WHILE-Anweisung werden (anders als bei einer REPEAT-Anweisung) auch leere Eingabedateien richtig behandelt.

```
Reset(f);           { hole 1. Zeile bzw. erkenne das Ende }
WHILE NOT Eof(f) DO BEGIN
   Read(f,i);       { versorge i und k aus der laufenden Zeile }
   Read(f,k);
   verarbeite (i, k);  { Verarbeitung }
   Readln(f);       { hole neue Zeile bzw. erkenne das Ende  }
   END;
```

*Querverweise*

| | |
|---|---|
| Boolean: | 6.2.4 |
| Ausdruck: | 9 |
| GOTO-Anweisung: | 10.1.4 |
| EXIT-Anweisung: | 10.1.5 |
| RETURN-Anweisung: | 10.1.6 |
| Ausnahmebehandlung: | 14 |
| Raise: | 15.11 |

## FOR-Anweisung

FOR-Anweisungen haben die Form:

```
FOR-Anweisung   = "FOR" Laufvariable ":=" Anfangswert
                  ( "TO" | "DOWNTO" ) Endwert "DO" Anweisung.

Laufvariable    = Variablen-Bezeichner.

Anfangswert     = Ordinal-Ausdruck.

Endwert         = Ordinal-Ausdruck.
```

Die FOR-Anweisung veranlaßt die wiederholte Ausführung der in ihr enthaltenen ("inneren") Anweisung. Dabei wird der Laufvariablen bei jedem Durchlauf ein neuer Wert (aus der Folge sukzessiver Werte zwischen Anfangs- und Endwert einschließlich, bei TO aufsteigend, bei DOWNTO absteigend) zugewiesen.

Diese Abarbeitung der Anweisungsfolge kann durch die Ausführung einer GOTO-Anweisung (siehe 10.1.4), einer EXIT-Anweisung (siehe 10.1.5), einer RETURN-Anweisung (siehe 10.1.6) den Aufruf der vordefinierten Prozedur Raise (siehe 15.11) oder das Auftreten einer Ausnahmesituation abgebrochen werden.

Die Laufvariable muß eine Ganzvariable sein, deren Bezeichner im Variablendeklarationsteil in dem Block definiert ist, der die FOR-Anweisung unmittelbar enthält. Das bedeutet, daß globale Variablen nicht als Laufvariablen zugelassen sind. Die Laufvariable muß von einem Ordinaltyp sein, und der Typ des Anfangs- und des Endwertes müssen mit diesem verträglich sein.

Der Anfangs- und der Endwert der FOR-Anweisung werden nur vor dem ersten Durchlauf ausgewertet. Das hat zur Folge, daß die Schleife nicht durch Änderung des Endwertes abgebrochen werden kann.

Wenn der Anfangswert größer (bei DOWNTO kleiner) als der Endwert ist, wird die innere Anweisung überhaupt nicht ausgeführt; sind beide gleich, dann genau einmal.

Falls die innere Anweisung mindestens einmal ausgeführt wird, so müssen Anfangs- und Endwert zuweisungsverträglich mit dem Typ der Laufvariablen sein.

Nach der Ausführung einer FOR-Anweisung ist der Wert der Laufvariablen undefiniert - es sei denn, die FOR-Anweisung wird durch eine GOTO- oder EXIT-Anweisung verlassen. Dann besitzt sie den gerade aktuellen Wert. "Undefiniert" bedeutet hier insbesondere, daß man sich nicht darauf verlassen darf, daß die Laufvariable hinter der FOR-Anweisung den Endwert besitzt.

Weder die FOR-Anweisung noch eine Prozedur- oder Funktionsdeklaration des Blockes, der die FOR-Anweisung unmittelbar enthält, darf eine Anweisung enthalten, die die Laufvariable gefährden (d. h. ihren Wert potentiell verändern) kann.

Eine Anweisung A gefährdet eine Variable V eines Ordinaltyps, wenn eine der folgenden Aussagen gilt:

a) A ist eine Zuweisung und V ist ihre linke Seite

b) A ist ein Prozeduraufruf oder enthält einen Funktionsaufruf, in dem V als Aktualparameter auftritt und der zugehörige Formalparameter ist ein Variablenparameter.

c) A ist ein Aufruf der Standardprozedur Read, Readln oder Readstring und V ist ein Parameter in A.

d) A ist eine FOR-Anweisung und V repräsentiert die Laufvariable in A.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Range_Error | - Bei der Ausführung der Anweisung in einer FOR-Anweisung liegt der Anfangs- oder Endwert der FOR-Anweisung nicht im Wertebereich des Typs der Laufvariablen. |

*Beispiel*

```
FOR i := 1 TO 100 DO
   FOR j := 1 TO i-1 DO BEGIN
      h := a [i, j];
      a [i, j]: = a [j, i];
      a [j, i] := h;
      END;
```

*Hinweise*

– Der Wert einer Laufvariablen darf nicht vom Anwender verändert (gefährdet) werden. Eine Gefährdung kann auch in einem eingeschachtelten Unterprogramm erfolgen, das in der FOR-Schleife aufgerufen wird. Der Aufruf kann von Daten abhängen, die erst zur Laufzeit bekannt sind. Aus diesem Grund wurden die Einschränkungen auf alle eingeschachtelten Unterprogramme ausgeweitet.

– FOR-Anweisungen sind besonders gut geeignet, um Arrays zu bearbeiten, da bei diesen die Anzahl der Elemente in einer Zeile, Spalte usw. von vornherein bekannt ist. Am Beispiel erkennt man, daß Anfangs- und Endwert der Laufvariablen nicht konstant sein müssen, jedoch zu Beginn der Ausführung der FOR-Anweisung einen definierten Wert besitzen müssen.

*Querverweise*


| | |
|---|---|
| Ordinal-Typ: | 6.2 |
| Verträglich: | 6.6.2 |
| Zuweisungsverträglich: | 6.6.3 |
| Unterprogrammaufruf: | 8.7 |
| Ausdruck: | 9 |
| GOTO-Anweisung: | 10.1.4 |
| EXIT-Anweisung: | 10.1.5 |
| RETURN-Anweisung: | 10.1.6 |
| Ausnahmebehandlung: | 14 |
| Read, Readln: | 15.1, 19 |
| Readstring: | 15.3 |

# WITH-Anweisung

WITH-Anweisungen haben die Form:

```
WITH-Anweisung  = "WITH" RECORD-Variablen-Liste "DO" Anweisung.

RECORD-Variablen-Liste
                = RECORD-Variablen-Objekt {"," RECORD-Variablen-Objekt}.
```

Die WITH-Anweisung eröffnet den Gültigkeitsbereich einer oder mehrerer RECORD-Variablen, sodaß in der "inneren" Anweisung auf die Feld-Bezeichner dieser Variablen ohne Angabe der RECORD-Variablen direkt zugegriffen werden kann (siehe 9.6.2).

Die sichtbar gewordenen Bezeichner verdecken alle weiter außen definierten gleichnamigen Bezeichner. Insbesondere bei verschachtelten WITH-Anweisungen ist auf die Sichtbarkeitsregeln zu achten.

Der Zugriff auf die RECORD-Variablen erfolgt zu Beginn der Ausführung der WITH-Anweisung und die Referenzen auf die Variablen bleiben für die gesamte Dauer der Abarbeitung der WITH-Anweisung erhalten.

Dies ist in zweifacher Hinsicht von Bedeutung:

1. Für das Auftreten bestimmter Laufzeitfehler (siehe 9.5.2, 9.5.3, 9.5.4 und die Beispiele 2, 3 und 4).
2. Wenn eine RECORD-Variable in einer WITH-Anweisung ein indiziertes Objekt oder eine Zeigerdereferenzierung enthält, so wirken sich Änderungen der Indexausdrücke bzw. der Zeiger-Objekte während der Abarbeitung der inneren Anweisung in der WITH-Anweisung auf die Referenz nicht mehr aus (siehe Beispiele 5 und 6).

Die Anweisung

```
WITH v1, v2, ..., vn DO Anweisung
```

ist gleichbedeutend mit

```
WITH v1 DO
   WITH v2 DO
      ...
         WITH vn DO Anweisung
```

*Hinweise*

- WITH-Anweisungen werden gewöhnlich verwendet, um Schreibaufwand zu sparen. Dies geht allerdings auf Kosten der Lesbarkeit.
- Besitzen mehrere RECORD-Variablen gleichnamige Komponenten, dann stellen geschachtelte WITH-Anweisungen eine potentielle Gefahr dar, da Feldauswahlbezeichner sich gegenseitig überdecken. Damit ist nicht immer sofort ersichtlich, welche Komponente welcher RECORD-Variablen angesprochen wird.

*Beispiel 1*

Mit einer WITH-Anweisung kann die Anweisung

```
IF datum.monat = 12 THEN BEGIN
   datum.monat := 1;
   datum.jahr  := datum.jahr + 1;
   END
ELSE
   datum.monat := datum.monat + 1;
```

wie folgt verkürzt werden:

```
WITH datum DO
   IF monat = 12 THEN BEGIN
      monat := 1;
      jahr  := jahr + 1;
      END
   ELSE
      monat := monat + 1;
```

*Beispiel 2*

Dieses Beispiel zeigt einen Fehler bei der Verwendung einer WITH-Anweisung. Der Wert (nämlich der Dateizeiger) der FILE-Variablen f wird durch Get(f) verändert, obwohl (aufgrund der WITH-Anweisung) noch eine Referenz auf die Puffervariable f↑ existiert.

```
VAR
   f : FILE OF record_typ;
BEGIN
   ...
   WITH f↑ DO BEGIN
      ...
      Get (f); {das ist ein Fehler mit undefinierten Auswirkungen}
      ...
      END;
   ...
   END
```

*Beispiel 3*

Das folgende Beispiel zeigt einen anderen Fehler bei der Verwendung einer WITH-Anweisung. Mit Dispose (p) wird ein Verweiswert entfernt, obwohl (aufgrund der WITH-Anweisung) noch eine Referenz auf die dynamische Variable p↑ existiert (siehe 9.5.3).

```
VAR
   p : ↑record_typ;
BEGIN
   New (p);
   WITH p↑ DO BEGIN
      ...
      Dispose (p); {das ist ein Fehler mit undefinierten
                    Auswirkungen}
      ...
      END;
   ...
END.
```

*Beispiel 4*

Das folgende Beispiel zeigt einen dritten Fehler bei der Verwendung einer WITH-Anweisung. Infolge der Anweisung r.b := False ist die "True-Variante" der RECORD-Variablen r nicht für die Gesamtdauer der (aufgrund der WITH-Anweisung bestehenden) Referenz auf ihre Komponente r.y aktiv (siehe 9.5.2).

```
VAR
   r :  RECORD
           CASE b : Boolean OF
              False: (x : Integer);
              True:  (y : record_typ);
        END;

BEGIN
   r.b := True;
   WITH r.y DO BEGIN
      ...
      r.b := False; {das ist ein Fehler mit undefinierten
                     Auswirkungen}
      ...
      END;
   ...
END.
```

*Beispiel 5*

Das folgende Beispiel zeigt den Fall, daß sich eine Änderung eines Indexausdrucks nicht auf die Referenz eine indizierte, Objektes auswirkt.

```
VAR
   i : Integer;
   a : ARRAY [1..10] OF RECORD x : ... END;

BEGIN
   i := 2;
   WITH a[i] DO BEGIN {Referenz auf a[2] (=a[i]) wird aufgebaut}
      ...
      i := 8; {Referenz auf a[2] bleibt bestehen}
      ...
      x := ...; {bezieht sich daher noch immer auf a[2].x
                 nicht aber auf a[8].x (=a[i].x)}
      ...
      END;
   ...
END.
```

*Beispiel 6*

Das folgende Beispiel zeigt den Fall, daß sich eine Änderung eines Zeigerobjekts nicht auf die Referenz auf die dynamische Variable auswirkt.

```
TYPE
   record_ptr = ↑record_typ;
   record_typ = RECORD
                   x,
                   y    : Integer;
                   next : record_ptr;
                END;

VAR
   p,
   liste : record_ptr;

BEGIN
   ...
   p := liste;
   WITH p↑ DO BEGIN {Referenz auf liste↑ (=p↑) wird aufgebaut}
      WHILE (p <> NIL) AND THEN (x > 0) DO BEGIN
         bearbeite (x, y);
         p := next; {Referenz auf liste↑ bleibt bestehen}
         END {WHILE};
      END {WITH};
   ...
END.
```

Dieser Programmteil hat nicht den erwünschten Effekt (Zugriffe auf die Komponenten der dynamischen RECORD-Variablen liste↑, liste↑.next↑, liste↑.next↑.next↑, ...), denn während der gesamten Ausführung der WITH-Anweisung (und daher während sämtlicher Durchläufe der WHILE-Anweisung) bleibt die (am Beginn der WITH-Anweisung aufgebaute) Referenz auf liste↑ bestehen, obwohl die Zeigervariable p durch p := next verändert wird. Alle Zugriffe auf x, y und next beziehen sich daher immer wieder nur auf liste↑.x, liste↑,y und liste↑.next.

Richtigerweise muß der Anweisungsteil daher folgendermaßen lauten:

```
BEGIN
   ...
   p := liste;
   WHILE (p <> NIL) AND THEN (p↑.x > 0) DO BEGIN
      WITH p↑ DO BEGIN
         bearbeite (x, y);
         p := next;
         END {WITH};
      END {WHILE};
   ...
END.
```

*Querverweise*

| | |
|---|---|
| Feldauswahlbezeichner: | 6.3.3 |
| RECORD-Typ: | 6.3.3 |
| Objekt: | 9.6 |
| Sichtbarkeitsregeln: | 12 |

# Hauptprogramm und Pakete

Ein Pascal-XT-Programm besteht aus einer oder mehreren Übersetzungseinheiten, von denen genau eine ein Hauptprogramm sein muß.

In Norm-Pascal besteht ein Programm nur aus einem einzigen Hauptprogramm.

## Hauptprogramm

```
Hauptprogramm   = {Kontext-Spezifikation}
                  "PROGRAM" Bezeichner
                  ["(" Programmparameterliste ")"] ";"
                  Hauptprogramm-Block ".".

Programmparameterliste
                = Bezeichnerliste.
```

Ein Hauptprogramm ist eine Übersetzungseinheit. Der Bezeichner hinter dem Schlüsselwort PROGRAM ist der Programmname.
In Pascal-XT muß der Programmname verschieden sein von den Bezeichnern aller Pakete, die zu einem Programm gehören (siehe 13.1).
In Norm-Pascal besitzt dieser Bezeichner keine Bedeutung innerhalb des Programms.

In der Kontext-Spezifikation werden die Bezeichner der Pakete aufgeführt, auf die im Hauptprogramm zugegriffen werden soll. Die Kontext-Spezifikation ist in 11.3 beschrieben.

Die Bezeichner in der Programmparameterliste müssen voneinander verschieden sein. Die Parameter sind in 11.5 ausführlich beschrieben.

*Querverweise*

Kontext-Spezifikation: 11.3
Programmparameter: 11.5
Block: 12.1
Sichtbarkeitsregeln: 12.2
Programmstruktur: 13.1
Übersetzungseinheiten: 13.2

Programmausführung: 13.3

Beispielprogramm Labyrinth

Das folgende Beispiel zeigt ein vollständiges Pascal-Programm zur Lösung des Labyrinth-Problems, das in ähnlicher Form in vielen Bereichen auftaucht wie z.B. in der Vermittlungstechnik oder der Verkehrsleitplanung.

Aufgabe:

Es sollen in einem Labyrinth von einem gegebenen Ausgangspunkt aus alle Wege zum Ausgang gefunden werden.

Das Labyrinth selbst wird durch ein zweidimensionales Feld

```
VAR lab : ARRAY[0..n,0..n] OF Char;
```

beschrieben. Die Wände werden durch ’#’, die Gänge durch Leerzeichen dargestellt. Der zurückgelegte Weg soll durch Punkte markiert werden.

In der Prozedur "suchen" wird die hier verwendete Suchstrategie realisiert, der ein rekursives Verfahren zugrunde liegt. Die Prozedur verdeutlicht die Möglichkeiten der Programmiersprache Pascal, die u. a. erlauben, mit wenigen Sprachelementen leistungsfähige Algorithmen zu formulieren.

Folgende Tätigkeiten werden von der Prozedur "suchen" ausgeführt:

- Suche nach dem nächsten freien Platz,
- Punkte in Sackgassen rückgängig machen,
- Vermeiden, im Kreis zu Laufen,
- Rückverfolgen eines gefundenen Auswegs bis zur ersten Alternative und Suche nach einem neuen Weg,
- Beenden der Suche nach dem letzten möglichen Weg.

```
PROGRAM labyrinth (Output, lfile);

CONST  n      = 16;
TYPE   labtyp = ARRAY [0..n, 0..n] OF Char;
VAR    lab    : labtyp;
       lfile  : Text;

PROCEDURE read_lab (VAR l : labtyp);
VAR i, k: Integer;
BEGIN
   Reset (lfile);
   FOR i := 0 TO n DO BEGIN
      FOR k := 0 TO n DO
         Read (lfile, l[i,k]);
      Readln;
      END;
END { read_lab };

PROCEDURE write_lab (l : labtyp);
VAR i, k: Integer;
BEGIN
   FOR i := 0 TO n DO BEGIN
      FOR k := 0 TO n DO
         Write (l[i,k]);
      Writeln;
      END;
END { write_lab };

PROCEDURE suchen (i, k: Integer);
BEGIN
  IF lab[i,k] = ' ' THEN BEGIN   { freier Platz ? }
    lab[i,k] := '.';             { Wegpunkt markieren }
    IF (i MOD n = 0) OR
       (k MOD n = 0)             { Ausgang ? }
    THEN write_lab (lab)         { Weg ausgeben }
    ELSE BEGIN                   { rekursiver Aufruf }
       suchen (i+1,k);           { entgegen Uhrzeigersinn }
       suchen (i,k+1);
       suchen (i-1,k);
       suchen (i,k-1);
       END;
    lab[i,k] := ' ';             { Punkt loeschen. Die Rueck- }
                                 { verfolgung des Wegs ergibt }
                                 { sich automatisch aus der   }
                                 { Rekursionsaufloesung       }
    END;                         { IF freier Platz }
END; { suchen }

BEGIN { labyrinth }
   read_lab (lab);             { Einlesen des Labyrinths }
   suchen (n DIV 2, n DIV 2)   { Beginn der Suche im }
                               { Mittelpunkt         }
END.
```

# Pakete

Pakete erlauben die Zusammenfassung logisch zusammengehöriger Deklarationen. In der einfachsten Form können Pakete gemeinsam verwendete Konstanten-, Typ- und Variablendeklarationen enthalten. Im allgemeineren Fall spezifizieren Pakete zusammengehörige Datenstrukturen und Unterprogramme, die auf diesen Datenstrukturen arbeiten. Eine ausführliche Beschreibung über die Verwendung von Paketen befindet sich im Kapitel 17.

Ein Paket wird in eine Paket-Spezifikation und eine Paket-Implementierung aufgeteilt, die getrennte Übersetzungseinheiten sind. Die Paket-Spezifikation enthält den nach außen sichtbaren Teil des Pakets, die Paket-Implementierung die Realisierung, die vor Zugriffen von außen geschützt ist.

Die Syntax für Paket-Spezifikation und Paket-Implementierung lautet:

```
Paket-Spezifikation
                = {Kontext-Spezifikation}
                  "PACKAGE" Bezeichner
                  ["(" Programmparameterliste ")"] ";"
                  {  Konstantendefinitionsteil
                   | Typdefinitionsteil
                   | Variablendeklarationsteil
                   | Prozedurkopf  ";" [Direktive ";"]
                   | Funktionskopf ";" [Direktive ";"]
                   | "ENTRY" Prozedurkopf  ";"
                   | "ENTRY" Funktionskopf ";"
                   | INLINE-Prozedurdeklaration
                   | INLINE-Funktionsdeklaration
                  }
                  "END" ".".

Paket-Implementierung
                = {Kontext-Spezifikation}
                   "PACKAGE" "BODY" Paket-Bezeichner
                  ["(" Programmparameterliste ")"] ";"
                  {  Konstantendefinitionsteil
                   | Typdefinitionsteil
                   | Variablendeklarationsteil
                   | Prozedurdeklaration
                   | Funktionsdeklaration
                  } Anweisungsteil ".".
```

Der Bezeichner hinter dem Schlüsselwort PACKAGE ist der Name des Pakets. Der gleiche Name muß auch als Paket-Bezeichner hinter dem Schlüsselwort BODY der zugehörigen Paket-Implementierung auftreten (siehe auch 13.1).

Zu jeder Paket-Spezifikation muß eine Paket-Implementierung existieren. Diese kann auch leer sein, d. h. nur aus

```
"PACKAGE" "BODY" Bezeichner; "BEGIN" "END" "."
```

bestehen (siehe 11.2.2).

Die Deklarations- und Definitionsteile in der Paket-Spezifikation, gefolgt von den Deklarations- und Definitionsteilen und dem Anweisungsteil der Paket-Implementierung bilden zusammen den Paket-Block. Ein Paket-Block ist analog aufgebaut wie ein Prozedur-, Funktions- oder Hauptprogramm-Block. Er darf aber unmittelbar keine Markendeklarationen enthalten.

Wegen der Zusammenfassung der Paket-Spezifikation und Paket-Implementierung zu einem Paket-Block dürfen die Bezeichner aus der Paket-Spezifikation nicht nochmals in der Implementierung deklariert bzw. definiert werden.

In der Kontext-Spezifikation werden die Bezeichner derjenigen fremden Pakete aufgeführt, auf die zugegriffen werden soll. Die in der Paket-Spezifikation aufgeführten Bezeichner sind auch in der Paket-Implementierung bekannt, die die in der Paket-Implementierung aufgeführten jedoch nicht in der Paket-Spezifikation. Die Kontext-Spezifikation ist in 11.3 beschrieben.

Paket-Spezifikation und Paket-Implementierung können getrennte Programmparameterlisten besitzen. Die Bezeichner der Programmparameterliste der Paket-Spezifikation müssen in der Paket-Spezifikation deklariert werden und sind im gesamten Paket sichtbar, die der Programmparameterliste der Paket-Implementierung müssen in der Paket-Implementierung deklariert werden und sind nur dort sichtbar. Die Programm-Parameter sind in 11.5 ausführlich beschrieben.

Zu jedem in der Paket-Spezifikation angegebenen Prozedur-Kopf bzw. Funktions-Kopf, für den keine (von Forward verschiedene) Direktive angegeben wurde, muß in der Paket-Implementierung die Prozeduridentifikation bzw. Funktionsidentifikation angegeben werden. Nur für INLINE-Unterprogramme wird die Deklaration bereits in der Paket-Spezifikation angegeben (siehe auch 11.2.1).

*Hinweis*

Ein Markendeklarationsteil unmittelbar im Paket-Block wird nicht zugelassen, da sonst aus einem Unterprogramm in den Anweisungsteil der Paket-Implementierung, der nur ein einziges Mal ausgeführt wird (siehe 11.2.2), gesprungen werden könnte.

*Beispiel*

```
PACKAGE stack;
PROCEDURE push (x: Integer);
PROCEDURE pop  (VAR x: Integer);
FUNCTION  is_empty: Boolean;
END.
```

```
PACKAGE BODY stack;

VAR
   representation: ARRAY [1..100] OF Integer;
   top_of_stack  : 0..100;

PROCEDURE push (x: Integer);
BEGIN
   top of_stack := top_of_stack + 1;
   representation [top_of_stack] := x;
END {push};

PROCEDURE pop (VAR x: Integer);
BEGIN
   x := representation [top_of_stack];
   top_of_stack := top_of_stack - 1;
END {pop};

FUNCTION is_empty: Boolean
BEGIN
   is_empty := top_of_stack = 0;
END {is_empty};

BEGIN
   top_of_stack := 0;
END {stack}.
```

*Querverweise*


| | |
|---|---|
| Konstantendefinition: | 5 |
| Typdefinition: | 6 |
| Variablendeklaration: | 7 |
| Prozedurdeklaration: | 8.1 |
| Prozedurkopf: | 8.1 |
| Funktionsdeklaration: | 8.2 |
| Funktionskopf: | 8.2 |
| INLINE-Unterprogramme: | 8.3 |
| ENTRY-Unterprogramme: | 8.4 |
| Kontext-Spezifikation: | 11.3 |
| Programmparameter: | 11.5 |
| Sichtbarkeitsregeln: | 12 |
| Programmstruktur: | 13.1 |
| Übersetzungseinheiten: | 13.2 |
| Programmausführung: | 13.3 |
| Paket-Konzept: | 17 |

## Paket-Spezifikation

Alle in der Paket-Spezifikation deklarierten und definierten Bezeichner können auch außerhalb des Pakets verwendet werden. Deklarierte Variable können auch außerhalb des Pakets verändert werden.

Bei Unterprogrammen muß beachtet werden, ob eine Direktive oder die Schlüsselwörter ENTRY bzw. INLINE angegeben wurden:

- **Unterprogramme ohne Direktiven**

Für Prozeduren und Funktionen ohne Angabe einer Direktive wird nur der Kopf angegeben, die zugehörigen Identifikationen werden erst in der Paket-Implementierung angegeben.

- **Unterprogramme mit Direktiven**

Für Unterprogramme mit einer von Forward verschiedenen Direktive darf es in der Paket-Implementierung keine Identifikation dazu geben. Der Unterprogrammblock muß in einer anderen Sprache, also außerhalb des Pakets vorliegen.

- **ENTRY - Unterprogramme**

Geht einem Prozedur- bzw. einem Funktionskopf in einer Paket-Spezifikation das Wortsymbol ENTRY voraus, so kann die so deklarierte Prozedur bzw. Funktion auch von Programm-Komponenten, die nicht in Pascal-XT geschrieben sind, aufgerufen werden, wobei implementierungsdefinierte Schnittstellen (siehe Benutzerhandbuch) eingehalten werden müssen. Vor dem erstmaligen Aufruf einer mit ENTRY gekennzeichneten Prozedur oder Funktion aus einer fremdsprachigen Programm-Komponente werden das Paket, das die Prozedur- bzw. Funktions-Deklaration enthält, sowie alle von diesem direkt oder indirekt in WITH-Listen importierten und nicht bereits initialisierten Pakete initialisiert (siehe 13.3).

**Implementierungsdefinierte Eigenschaft**

Für ENTRY-Unterprogramme kann es implementierungsdefinierte Einschränkungen geben.

- **INLINE - Unterprogramme**

Für ein (außerhalb des Pakets verwendbares) INLINE-Unterprogramm muß in der Paket-Spezifikation auch der zugehörige Block angegeben werden, da der Unterprogrammaufruf, der außerhalb des Pakets liegen kann, durch den Unterprogramm-

Block ersetzt (expandiert) wird (siehe auch 8.3). Durch die Angabe des Unterprogramm-Blocks müssen möglicherweise weitere Deklarationen und Definitionen, die sonst nur in der Paket-Implementierung benötigt würden, bereits in die Paket-Spezifikation übernommen und damit nach außen sichtbar gemacht werden.

*Hinweis*

Eine Paket-Spezifikation soll nur die Deklarationen und Definitionen enthalten (und dadurch nach außen sichtbar machen), die eine andere Übersetzungseinheit unbedingt benötigt.

*Beispiel*

Das Beispiel zeigt die Realisierung eines Datentyps mit den möglichen Zugriffsoperationen. Zur Vereinfachung enthält der Typ knoten nur ein einziges Feld art, auf das nur über die angegebenen Unterprogramme zugegriffen wird. Durch die Realisierung als INLINE-Unterprogramme werden an der Aufrufstelle die Zugriffe durch die in den Unterprogramm-Blöcken angegebenen Anweisungen ersetzt.

```
PACKAGE data;

TYPE
   bereich      = 0..255;
   knotenzeiger = ↑knoten;
   knoten       = RECORD
                     art : bereich;
                  END;

FUNCTION neuer_knoten: knotenzeiger;

INLINE FUNCTION art_von (zeiger: knotenzeiger): bereich;
BEGIN
   art_von := zeiger↑.art
END;

INLINE PROCEDURE setze_art (zeiger: knotenzeiger; i: bereich);
BEGIN
   zeiger↑.art := i;
END;

END.
```

*Querverweise*

Prozedurkopf: 8.1
Funktionskopf: 8.2
INLINE-Unterprogramme: 8.3
ENTRY-Unterprogramme: 8.4

## Paket-Implementierung

Zu jeder Paket-Spezifikation muß eine Paket-Implementierung existieren. Enthält eine Paket-Spezifikation keine Prozedur- und Funktionsköpfe, zu denen noch Prozedur- bzw. Funktionsidentifikationen angegeben werden müssen, so kann die zugehörige Paket-Implementierung "leer" sein.

Im Gegensatz zu den in der Paket-Spezifikation deklarierten Bezeichnern sind die in der Paket-Implementierung deklarierten Bezeichner nur innerhalb der Paket-Implementierung sichtbar und können außerhalb des Pakets nicht verwendet werden.

Der Anweisungsteil der Paket-Implementierung wird nur ein einziges Mal bei der Ausführung eines Programms ausgeführt. Der Anweisungsteil kann Anweisungen zur Initialisierung von Variablen enthalten (siehe Beispiel am Ende des Abschnitts 11.2).

*Hinweise*

- Zu einer gemeinsamen Paket-Spezifikation kann es in verschiedenen Programmen verschiedene Paket-Implementierungen geben, die unterschiedliche Anforderungen an Speicherplatz- und/oder Laufzeit-Bedingungen genügen. In Paket stack (siehe 11.2) wird der Stack (Keller) durch die ARRAY-Variable representation realisiert und hat damit eine definierte Größe. Eine alternative Implementierung könnte den Stack als eine Liste realisieren, deren Größe (theoretisch) nicht beschränkt ist. An der Schnittstelle (Paket-Spezifikation) ändert sich dabei nichts.
- Der Wert einer in der Paket-Implementierung deklarierten Variable kann nur in der Implementierung geändert werden, der Wert einer in der Paket-Spezifikation deklarierten Variablen kann auch außerhalb des Pakets geändert werden.
- Zur Verbesserung der Lesbarkeit können für die in der Paket-Spezifikation angegebenen Unterprogramme die Formalparameterliste und bei Funktionen noch der Ergebnistyp wiederholt werden.

*Querverweise*

| | |
|---|---|
| Anweisungen: | 10 |
| Programmparameterliste: | 11.5 |
| Programmausführung: | 13.3.3 |

# Kontextspezifikation

In der Kontext-Spezifikation wird die Verbindung zu anderen Paketen hergestellt.

```
Kontext-Spezifikation
                = WITH-Liste | USE-Liste.

WITH-Liste      = "WITH" Paket-Bezeichner {","
                         Paket-Bezeichner} ";".

USE-Liste       = "FROM" Paket-Bezeichner "USE"
                     importierter_Bezeichner {","
                     importierter_Bezeichner} ";".

importierter_Bezeichner
                = Konstanten-Bezeichner | Typ-Bezeichner
                | Variablen-Bezeichner  | Prozedur-Bezeichner
                | Funktions-Bezeichner.
```

### WITH-Liste

Durch die WITH-Liste wird eine Beziehung zwischen den Übersetzungs-Einheiten eines Programms festgelegt: Tritt der Paket-Bezeichner P in der WITH-Liste der Übersetzungs-Einheit U auf, so "nimmt U auf die Paket-Spezifikation von P Bezug". Jede Paket-Implementierung nimmt automatisch auf die zugehörige Paket-Spezifikation Bezug.

Die Beziehung "Übersetzungs-Einheit U nimmt Bezug auf Übersetzungs-Einheit V" muß eine partielle Ordnung auf der Menge aller zu einem Programm gehörigen Übersetzungs-Einheiten sein. Diese Forderung schließt aus, daß 2 Spezifikationen auf sich gegenseitig Bezug nehmen. Es ist jedoch nicht verboten, daß sich die Implementierung eines Pakets P auf die Spezifikation eines Pakets Q bezieht und sich gleichzeitig die Spezifikation oder die Implementierung von Q auf die Spezifikation von P bezieht.

*Beispiel*

```
PACKAGE P;
 .
 .
 .
END.
```

```
PACKAGE Q;
 .
 .
 .
END.
```

```
WITH Q;
PACKAGE BODY P;
 .
 .
 .
BEGIN
END.
```

```
WITH P;
PACKAGE BODY Q;
 .
 .
 .
BEGIN
END.
```

Tritt der gleiche Paket-Bezeichner mehrmals in derselben oder in verschiedenen WITH-Listen einer Übersetzungseinheit auf, so hat nur das erste Auftreten eine Bedeutung, die folgenden Nennungen werden ignoriert. Dies gilt insbesondere auch dann, wenn das erste Auftreten in einer WITH-Liste einer Paket-Spezifikation und ein weiteres Auftreten in einer WITH-Liste der zugehörigen Paket-Implementierung erfolgt.

Durch die Angabe eines Paket-Bezeichners P in einer WITH-Liste einer Übersetzungseinheit U können in U alle unmittelbar in der Spezifikation von P deklarierten Bezeichner durch Angabe von

```
Paket-Bezeichner "." Bezeichner
```

verwendet werden. Die Bezeichner aus dem Paket P werden durch Angabe des Paket-Bezeichners (voll) qualifiziert. In einer Paket-Implementierung werden Bezeichner aus der zugehörigen Paket-Spezifikation ohne Voranstellung des Paket-Bezeichners verwendet.

In einer WITH-Liste kann der Name eines Hauptprogramms (Programmname) nicht angegeben werden.

*Beispiel*

In der Paket-Implementierung b wird der Aufzählungstyp farbe mit seinen Konstanten-Bezeichnern aus dem Paket a verwendet.

```
PACKAGE a;
TYPE
   farbe = (rot, blau, gelb, gruen, weiss, violett, orange);
END.


WITH a;
PACKAGE BODY b;
VAR
    f: set of a.farbe;
...
BEGIN
   f := [ a.rot, a.blau, a.gelb ];
END { b }.
```

## USE-Liste

Importierte Bezeichner aus einem fremden Paket, die in einer USE-Liste aufgeführt werden, können ohne Voranstellen des Paket-Bezeichners verwendet werden.

Der in einer USE-Liste aufgeführte Paket-Bezeichner muß auch in einer vorausgehenden WITH-Liste aufgeführt sein und die importierten Bezeichner müssen in der Paket-Spezifikation des genannten Pakets deklariert sein.

Bezeichnet der in einer USE-Liste aufgeführte Bezeichner einen Aufzählungstyp, so werden implizit auch alle Konstanten-Bezeichner, die die Werte des Aufzählungstyps bilden, importiert.

Tritt derselbe Bezeichner mehrmals in derselben oder in verschiedenen USE-Listen einer Übersetzungseinheit auf, so hat nur das erste Auftreten eine Bedeutung, die folgenden Nennungen werden ignoriert. Dies gilt insbesondere auch dann, wenn das erste Auftreten in einer USE-Liste einer Paket-Spezifikation und ein weiteres Auftreten in einer USE-Liste der zugehörigen Paket-Implementierung erfolgt.

*Hinweis*

Zur Verbesserung der Lesbarkeit eines Programms sollten USE-Listen nur restriktiv eingesetzt werden, damit aus der Verwendung eines Bezeichners der Ort seiner Deklaration bzw. Definition klar ersichtlich ist.

*Beispiel*

Das Beispiel aus 11.3.1 sieht bei Verwendung der USE-Liste folgendermaßen aus:

```
PACKAGE a;
TYPE
   farbe = (rot, blau, gelb, gruen, weiss, violett, orange);
END.


WITH a;
FROM a USE farbe;    { damit werden auch die Bezeichner }
                     { rot bis orange importiert        }
PACKAGE BODY b;
VAR
    f: set of farbe;
...
BEGIN
   f := [ rot, blau, gelb ];
END { b }.
```

# Private Typen

Ein Zeigertyp heißt privater Zeigertyp, wenn der Zeigertyp in einer Paket-Spezifikation und der Domänentyp erst in der zugehörigen Paket-Implementierung definiert werden. Außerhalb des Pakets ist nur der Typ-Bezeichner, nicht jedoch die zugrundeliegende Struktur des Domänentyps bekannt.

Außerhalb dieses Paketes und in Inline-Unterprogrammen, die in dieser Paket-Spezifikation deklariert sind, ist eine Zeigerdereferenzierung mit einem Zeiger des privaten Zeigertyps nicht erlaubt. Zugriffe auf dynamische Variable dieses Typs können nur über Zugriffs-Prozeduren und -Funktionen, die in der Paket-Spezifikation deklariert sind, erfolgen. Die vordefinierten Prozeduren New und Dispose können für private Zeigertypen ebenfalls nur innerhalb dieses Pakets aufgerufen werden.

*Hinweis*

Mit dem Konzept der privaten Zeigertypen unterstützt Pascal-XT das Konzept des "information hiding", d. h. dem Verstecken der Implementierungsdetails. Das erhöht die Sicherheit eines Programms, da von außen Datenstrukturen nicht unbeabsichtigt verändert werden können.

*Beispiel*

```
PACKAGE_queue manager;

TYPE queue = ↑element;
FUNCTION tail (q: queue): queue;

END {queue_manager}.



PACKAGE BODY queue_manager;

TYPE element = RECORD
                  next: queue
               END;
FUNCTION tail (q: queue): queue;
BEGIN
   tail := q↑.next
END {Tail};
BEGIN

END {queue_manager}.
```

*Querverweise*

Zeigertypen: 6.4

Domänentyp: 6.4
Dereferenzierung: 9.6.4
New: 15.2
Dispose: 15.2

# Programmparameter

Die Bezeichner in einer Programmparameterliste heißen Programmparameter. Jeder Programmparameter, außer den vordefinierten Bezeichnern Input und Output, muß in dem Block der Übersetzungseinheit, in deren Programmparameterliste er vorkommt, unmittelbar als FILE-Variable deklariert sein (siehe auch implementierungsdefinierte und implementierungsabhängige Eigenschaften).

Die Programmparameter aller Übersetzungseinheiten, die zu einem Programm gehören (siehe auch 13.1), müssen paarweise verschieden sein.
Da Input und Output als Programmparameter eine besondere Bedeutung haben (siehe unten), dürfen sie in verschiedenen Übersetzungseinheiten als Programmparameter aufgeführt werden. Sie bezeichnen dann immer dieselbe Datei.

In Pascal-XT werden die Bezeichner Input und Output beim Auftreten als Programmparameter automatisch aus einem nicht direkt ansprechbaren Paket Input_Package bzw. Output_Package importiert, das diese Bezeichner als Variable vom Typ Text deklariert. Bei der Initialisierung des Pakets Input_Package wird Reset (Input) und bei der Initialisierung des Pakets Output_Package wird Rewrite (Output) implizit ausgeführt.

```
PACKAGE Input_Package (Input);
VAR Input: Text;
END { Input_Package }.

PACKAGE Output_Package (Output);
VAR Output: Text;
END { Output_Package }.
```

In Norm-Pascal sind Input und Output Variablen-Bezeichner vom vordefinierten Typ Text. Erst durch das Auftreten in der Programmparameterliste werden sie sichtbar (im Gegensatz zu den sonstigen vordefinierten Bezeichnern) und vor dem ersten Zugriff automatisch durch Reset (Input) bzw. Rewrite (Output) eröffnet.

Die etwas andere Interpretation der vordefinierten Bezeichner Input und Output in Pascal-XT und in Norm-Pascal hat jedoch keinen Einfluß auf ein Programm.

### Implementierungsdefinierte Eigenschaft

- Für Programmparameter ist die Zuordnung an Objekte außerhalb des Programms implementierungsdefiniert.
- Die Wirkung der vordefinierten Prozeduren Reset bzw. Rewrite auf eine der vordefinierten Textdateien Input oder Output ist nach der Pascal-Norm implementierungsdefiniert.
  In Pascal-XT ist diese Eigenschaft für alle Implementierungen definiert: Es tritt ein Open_Error auf.

### Implementierungsabhängige Eigenschaft

Für Programmparameter, deren Variable keinen FILE-Typ besitzen, ist die Zuordnung an Objekte außerhalb des Programms implementierungsabhängig.
In Pascal-XT kann eine Variable nicht in der Programmparameterliste angegeben werden, wenn sie nicht einen FILE-Typ besitzt.

*Hinweis*

Die paarweise Verschiedenheit der Programmparameter in allen Übersetzungseinheiten, die zu einem Programm gehören, kann von der Pascal-XT Implementierung nicht überprüft werden.

*Beispiel*

```
PACKAGE reporter (meldungsausgabe);
VAR
   meldungsausgabe: Text;
...
END.


PACKAGE BODY reporter (meldungstexte, Output);
VAR
   meldungstexte: Text;
BEGIN
...
END.


WITH reporter;
FROM reporter USE meldungsausgabe;
PROGRAM beispiel (Input, Output, dateneingabe);
VAR
   dateneingabe: Text;
BEGIN
...
END.
```

Die Programmparameter des Programms sind, bis auf Input und Output, alle verschieden. Die Datei meldungstexte ist nur in der Paket-Implementierung von reporter bekannt, und es kann auf sie nicht von außerhalb des Pakets zugegriffen werden. Dagegen kann auf meldungsausgabe der Paket-Spezifikation auch von außerhalb zugegriffen werden (siehe Hauptprogramm).

*Querverweise*


| | |
|---|---|
| Textdatei: | 6.3.5.2 |
| Sichtbarkeitsregeln: | 12 |
| Reset, Rewrite: | 15.1 |
| Input, Output: | 6.3.5.2, 11.5, 19, A.2 |

# Sichtbarkeitsregeln

Jeder Bezeichner und jede Marke muß einen Definitionspunkt haben. Zu jedem Definitionspunkt gibt es ein Gebiet (siehe 12.2), das Teil des Programmtextes ist, und einen Gültigkeitsbereich (siehe 12.3) der das gesamte Gebiet oder nur einen Teil davon umfaßt. Der Definitionspunkt eines Bezeichners muß vor jeder Verwendung dieses Bezeichners liegen, außer Verwendungen als Domänentyp in einem Zeigertyp und Verwendung von Programmparametern.

## Blöcke

```
Block =    {    Markendeklarationsteil
              | Konstantendefinitionsteil
              | Typdefinitionsteil
              | Variablendeklarationsteil
              | Prozedurdeklaration
              | Funktionsdeklaration
           }
           Anweisungsteil .
```

Ein Block besteht aus Deklarationen und Definitionen, kurz Deklarationsteil genannt, und einem Anweisungsteil. Durch eine Deklaration werden Objekte mit bestimmten Eigenschaften (Werte, Algorithmen) eingeführt. Durch eine Definition werden Objekte durch Gleichsetzung eines Namens mit einem anderen Objekt definiert. Die Definition ist durch ein Gleichheitszeichen gekennzeichnet. Der Anweisungsteil bestimmt die algorithmischen Aktionen, die während der Ausführung des Blocks abgearbeitet werden.

Alle Bezeichner und Marken, die in einem Block verwendet werden, müssen vorher deklariert bzw. definiert werden, sofern sie nicht bereits vordefiniert sind.

Ein Block, der einen Markendeklarationsteil unmittelbar enthält, in dem eine Marke definiert wird, muß genau eine Anweisung enthalten, die mit dieser Marke markiert ist. Es kann beliebig viele Sprunganweisungen (GOTO-Anweisung) mit Angabe dieser Marke geben.

In Pascal-XT können Deklarationen und Definitionen in einer beliebigen Reihenfolge angegeben werden und auch mehrmals auftreten.

In Norm-Pascal ist die Syntax für einen Block

```
Block =    [ Markendeklarationsteil    ]
           [ Konstantendefinitionsteil ]
           [ Typdefinitionsteil        ]
           [ Variablendeklarationsteil ]
           {   Prozedurdeklaration
             | Funktionsdeklaration
           }
           Anweisungsteil .
```

Die Definition von Block ist rekursiv, da Prozedur- und Funktionsdeklarationen wiederum einen Block besitzen, sofern sie nicht als externe Unterprogramme deklariert sind (siehe Direktiven). Ein Unterprogramm besitzt daher wiederum einen Deklarationsteil und einen Anweisungsteil. Ein Programm kann folglich viele geschachtelte Blöcke enthalten.

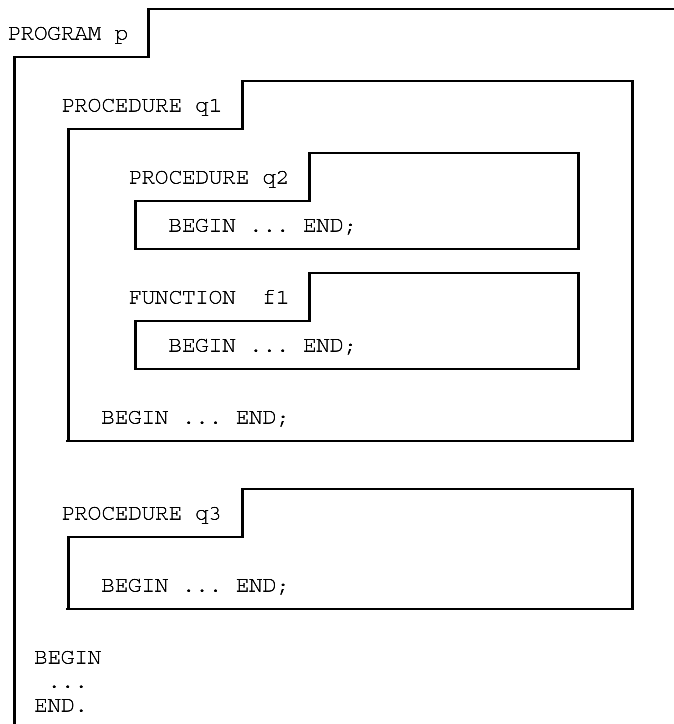



Bild 12-1: Blockstruktur in einem Programm

*Querverweise*


| | |
|---|---|
| Markendeklarationsteil: | 4 |
| Konstantendefinitionsteil: | 5 |
| Typdefinitionsteil: | 6 |
| Variablendeklarationsteil: | 7 |
| Prozedurdeklaration: | 8.1 |
| Funktionsdeklaration: | 8.2 |
| Prozedur-Block: | 8.1 |
| Funktions-Block: | 8.2 |
| Direktiven: | 8.6 |
| Anweisungen: | 10 |
| GOTO-Anweisung: | 10.1.4 |
| Hauptprogramm-Block: | 11.1 |
| Paket-Block: | 11.2 |
| Block: | 12.1 |
| Gebiet: | 12.2 |
| Definitionspunkt: | 12.2 |
| Gültigkeitsbereich: | 12.3 |
| Ausführen eines Blocks: | 13.3.1 |
| Vordefinierte Bezeichner: | A.2 |

# Definitionspunkte und Gebiete von Bezeichnern und Marken

Jeder Bezeichner und jede Marke innerhalb eines Programmtextes muß einen eindeutigen Definitionspunkt für ein zugehöriges Gebiet haben, das Teil des Programmtextes ist (siehe 13.1).

- **Marken**

Der Definitionspunkt einer Marke ist ihr Auftreten in einem Markendeklarationsteil (siehe Kapitel 4) und das zugehörige Gebiet ist der gesamte Block, der den Definitionspunkt unmittelbar enthält.

- **Konstanten-Bezeichner, Typ-Bezeichner, Variablen-Bezeichner, Prozedur-Bezeichner und Funktions-Bezeichner**

Der Definitionspunkt des Bezeichners ist sein Auftreten in seiner Definition bzw. seiner Deklaration (siehe Kapitel 5 bis 8). Eine Prozedur- bzw. Funktionsidentifikation enthält keinen Definitionspunkt, sondern eine Verwendung des Prozedur-Bezeichners bzw. Funktions-Bezeichners (siehe 8.1, 8.2).

Das zugehörige Gebiet ist der gesamte Block, der den Definitionspunkt unmittelbar enthält.
Ist der Definitionspunkt unmittelbar in einer Paket-Spezifikation enthalten, so ist das Gebiet der gesamte Paket-Block. Ist der Definitionspunkt jedoch unmittelbar in einer Paket-Implementierung enthalten, so ist das zugehörige Gebiet lediglich die Paket-Implementierung. Demnach können Bezeichner, die in einer Paket-Spezifikation definiert bzw. deklariert sind, auch direkt in der Paket-Implementierung verwendet werden, nicht jedoch umgekehrt.

- **Paket-Bezeichner**

Der Definitionspunkt eines Paket-Bezeichners ist sein Auftreten in einer Paket-Spezifikation (unmittelbar hinter PACKAGE). Das Auftreten des Paket-Bezeichners in einer Paket-Implementierung (unmittelbar hinter PACKAGE BODY) ist kein Definitionspunkt sondern eine Verwendung des Paket-Bezeichners.

Das Gebiet für den Definitionspunkt eines Paket-Bezeichners ist der gesamte Programmtext, ausgenommen die Übersetzungseinheiten, die den Paket-Bezeichner nicht in einer WITH-Liste ihrer Kontext-Spezifikation aufgeführt haben. Wird der Paket-Bezeichner in der WITH-Liste der Kontextspezifikation einer Kontextspezifikation einer Paket-Spezifikation aufgeführt, so umfaßt das Gebiet des Paket-Bezeichners auch die zugehörige Paket-Implementierung, unabhängig davon, ob der Paket-Bezeichner auch in einer WITH-Liste der Kontext-Spezifikation dieser Paket-Implementierung genannt ist.

- **Importierte Bezeichner**

Der Definitionspunkt eines importierten Bezeichners ist sein Auftreten in einer USE-Liste. Das Gebiet für den Definitionspunkt eines in eine Paket-Spezifikation importierten Bezeichners (der also in einer USE-Liste der Kontextspezifikation der Paket-Spezifikation aufgeführt ist) ist der gesamte zugehörige Paket-Block. Das Gebiet für den Definitionspunkt eines in eine Paket-Implementierung importierten Bezeichners ist die Paket-Implementierung. Der Definitionspunkt eines in ein Hauptprogramm importierten Bezeichners ist der Programm-Block. Importierte Bezeichner können also innerhalb ihres Gebietes (genauer Gültigkeitsbereich, siehe 12.3) direkt ohne Voranstellung des Paket-Bezeichners verwendet werden.

- **Feld-Bezeichner**

Der Definitionspunkt eines Feld-Bezeichners ist sein Auftreten in einer Felddarstellung. Das zugehörige Gebiet ist der RECORD-Typ, der den Definitionspunkt unmittelbar enthält (siehe 6.3.3).

- **Konstanten-Bezeichner eines Aufzählungs-Typs**

Das Gebiet für den Definitionspunkt eines Konstanten-Bezeichners, der als Wert eines Aufzählungstyps definiert ist, ist der gesamte Block, der den Aufzählungstyp unmittelbar enthält (siehe 6.2.5). Dies bedeutet, daß in folgendem Beispiel das Gebiet für den Definitionspunkt von rot der gesamte umgebende Block und nicht nur der RECORD-Typ ist:

```
RECORD
   farbe : (rot, gelb, gruen);
   ...
END
```

Falls der Aufzählungstyp in einer Paket-Spezifikation unmittelbar enthalten ist, so ist das Gebiet für die Konstanten-Bezeichner im Aufzählungstyp der gesamte Paket-Block; falls der Aufzählungstyp unmittelbar in einer Paket-Implementierung enthalten ist, so ist das Gebiet die Paket-Implementierung.

- **Parameter-Bezeichner**

Der Definitionspunkt eines Parameter-Bezeichners ist sein Auftreten in einer Formalparameterliste. Das zugehörige Gebiet ist die Formalparameterliste, die diesen einer Prozedur- bzw. Funktionsdeklaration, zu der es einen Prozedur-Block bzw. Funktions-Block gibt, so ist der Definitionspunkt eines Parameter-Bezeichners zugleich auch Definitionspunkt für einen Variablen-Bezeichner, Prozedur-Bezeichner bzw. Funktions-Bezeichner (in Abhängigkeit von der Parameterart) für das Gebiet, das der Prozedur-Block bzw. Funktions-Block ist.

- **Grenz-Bezeichner**

Der Definitionspunkt eines Grenz-Bezeichners ist sein Auftreten in einem Konformreihungsschema. Das zugehörige Gebiet besteht aus der Formalparameterliste, in der der Definitionspunkt unmittelbar enthalten ist, sowie dem zugehörigen Prozedur-Block bzw. Funktions-Block.

- **Feldauswahl-Bezeichner in einer WITH-Anweisung**

Die RECORD-Variable in einer WITH-Anweisung ist Definitionspunkt aller im zugehörigen RECORD-Typ deklarierten Feld-Bezeichner als Feldauswahl-Bezeichner. Das zugehörige Gebiet (in dem diese Feldauswahlbezeichner dann direkt verwendet werden können) ist die Anweisung in der WITH-Anweisung (10.5).

- **Vordefinierte Bezeichner**

Bezeichner für vordefinierte Konstanten, Typen, Prozeduren und Funktionen besitzen (vor dem Programmtext) einen imaginären Definitionspunkt, dessen Gebiet der gesamte Programmtext ist.

Dies gilt nicht für die vordefinierten Bezeichner Input und Output, die für Variable vom Typ Text stehen.
In Pascal-XT haben Input und Output ihren Definitionspunkt in den Paket-Spezifikationen der vordefinierten Pakete Input_Package bzw. Output_Package (siehe auch 11.5).

In Norm-Pascal haben Input und Output genau dann einen imaginären Definitionspunkt vor dem Hauptprogramm-Block, wenn sie in der Programmparameterliste auftreten.

*Querverweise*

| | |
|---|---|
| Markendeklaration: | 4 |
| Konstantendefinition: | 5 |
| Typdefinition: | 6 |
| Variablendeklaration: | 7.1 |
| Prozedurdeklaration: | 8.1 |
| Funktionsdeklaration: | 8.2 |
| Formalparameter: | 8.5 |
| Grenz-Bezeichner: | 8.5.4 |
| WITH-Anweisung: | 10.5 |
| Programmtext: | 13.1 |
| Vordefinierte Bezeichner: | A.2 |

# Gültigkeitsbereiche und Verwendung von Bezeichnern

Die Verwendung eines Bezeichners (einer Marke) ist nur innerhalb des Gültigkeitsbereichs seines (ihres) Definitionspunktes möglich. Der Gültigkeitsbereich eines Definitionspunktes ist sein Gebiet, abzüglich der darin enthaltenen Gebiete anderer Definitionspunkte gleichnamiger Bezeichner (Marken). Wenn also, wie in Bild 12-2, ein Bezeichner x im Hauptprogramm einen Definitionspunkt D1 hat und ein gleichnamiger Bezeichner einen Definitionspunkt D2 in der Prozedur proc hat, so ist der Gültigkeitsbereich von D1 unterbrochen durch das Gebiet von D2, so daß innerhalb der Prozedur proc nur D2, nicht aber D1 gültig ist.

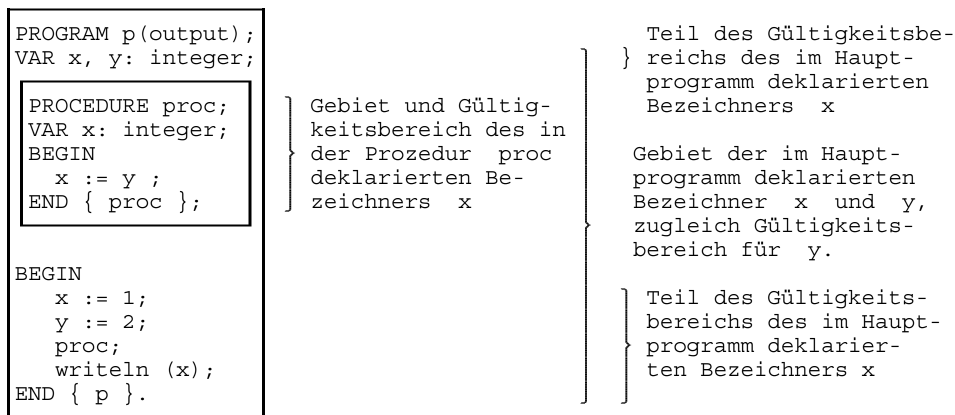



Bild 12-2: Beispiel von Gebieten und Gültigkeitsbereichen

In dem Gebiet eines Definitionspunktes eines Bezeichners (einer Marke) dürfen keine anderen gleichnamigen Bezeichner (Marken) einen Definitionspunkt für das gleiche Gebiet haben. Verschiedene, aber gleichnamige Bezeichner (Marken) können demnach nicht im selben Block deklariert bzw. definiert werden. Der folgende Typdefinitionsteil ist demnach unzulässig, da es im selben Block zwei verschiedene Definitionspunkte für die gleichnamigen Bezeichner unbekannt gibt:

```
TYPE
   farbe = (rot, gelb, gruen, unbekannt);
   form  = (rund, eckig, unbekannt);
              { fehlerhaft, da unbekannt doppelt definiert }
```

Bezeichner (Marken) können nur innerhalb ihres Gültigkeitsbereichs verwendet werden. Die Verwendung stellt dann eine Beziehung zu dem einzigen Definitionspunkt eines gleichnamigen Bezeichners (Marke) in diesem Gültigkeitsbereich her. Aufgrund dessen bezieht sich die Verwendung des Bezeichners x im Anweisungsteil der Prozedur proc in Bild 12-2 auf den Definitionspunkt D2 von x; die im Hauptprogramm deklarierte Variable x bleibt durch die Anweisung x := y unverändert. Das Programm gibt also 1 aus. Die Verwendung von y innerhalb der Prozedur bezieht sich dagegen auf die Variable y, die im Hauptprogramm deklariert ist.

Der Definitionspunkt eines Bezeichners (einer Marke) muß allen zugehörigen Verwendungen im Programmtext vorausgehen, mit zwei Ausnahmen:

- Neue Zeigertypen dürfen als Domänentyp einen Typ-Bezeichner vor seinem Definitionspunkt verwenden, sofern die Verwendung und der Definitionspunkt im gleichen Deklarationsteil unmittelbar enthalten sind (siehe 6.4).
- Programmparameter sind Verwendungen von Bezeichnern, die erst später im Hauptprogramm- bzw. Paket-Block deklariert werden.

Aufgrund dieser Regeln ist folgendes Programm fehlerhaft:

```
PROGRAM fehlerhaft;

   CONST max = 13;          { Definitionspunkt D1 für max     }

   PROCEDURE p;             { Beginn des Gebietes für D2      }

      TYPE t = 1..max;      { Verwendung vor Definitionspunkt }

      VAR max: t;           { Definitionspunkt D2 für max     }

   BEGIN
   END;                     { Ende des Gebietes für D2        }

BEGIN
END.
```

Es gibt im Hauptprogramm einen Definitionspunkt D1 für einen Bezeichner max mit dem gesamten Hauptprogramm-Block als Gebiet. In der Variablendeklaration innerhalb der Prozedur p gibt es einen Definitionspunkt D2 für einen zweiten Bezeichner max, dessen Gebiet der gesamte Prozedur-Block ist, also auch der Bereich des Prozedur-Blocks, der vor der Deklaration von max liegt. Aus dem Gültigkeitsbereich von D1 ist demnach dieser gesamte Prozedur-Block ausgenommen. Die Verwendung des Bezeichners max in der Typdefinition innerhalb der Prozedur p kann sich deshalb nur auf den Definitionspunkt D2 beziehen, da diese Verwendung nicht im Gültigkeitsbereich von D1 liegt. Dann geht jedoch diese Verwendung von max dem zugehörigen Definitionspunkt D2 voraus, was unzulässig ist.

Da der Paket-Bezeichner in einer Paket-Implementierung die Verwendung eines Bezeichners ist, muß der zugehörige Definitionspunkt dieses Paket-Bezeichners und damit die zugehörige Paket-Spezifikation im Programmtext (der gedachten Aneinanderreihung aller Übersetzungseinheiten, siehe Kapitel 13) der Paket-Implementierung vorausgehen. Analog muß im Programmtext die Paket-Spezifikation eines in einer WITH-Liste einer Kontext-Spezifikation genannten Pakets der mit dieser Kontext-Spezifikation beginnenden Übersetzungseinheit vorausgehen.

Wenn in einer WITH-Liste einer Kontext-Spezifikation einer Übersetzungseinheit ein Paket-Bezeichner aufgeführt ist, so können in dieser Übersetzungseinheit alle Bezeichner, die unmittelbar in der zugehörigen Paket-Spezifikation definiert bzw. deklariert sind, durch Voranstellung des Paket-Bezeichners verwendet werden:

```
Konstanten-Name = [Paket-Bezeichner"."] Konstanten-Bezeichner.

Typ-Name        = [Paket-Bezeichner"."] Typ-Bezeichner.

Variablen-Name  = [Paket-Bezeichner"."] Variablen-Bezeichner.

Prozedur-Name   = [Paket-Bezeichner"."] Prozedur-Bezeichner.

Funktions-Name  = [Paket-Bezeichner"."] Funktions-Bezeichner.
```

Formal ist in diesen Konstrukten der Bereich des Bezeichners hinter dem Punkt "." aus allen in Abschnitt 12.2 definierten Gebieten ausgenommen. Stattdessen ist der Paket-Bezeichner zugleich ein neuer Definitionspunkt aller Bezeichner, die bereits einen Definitionspunkt für das Gebiet der zugehörigen Paket-Spezifikationen haben. Das Gebiet dieser neuen Definitionspunkte ist dann genau der Bereich des Bezeichners hinter dem ".", der aus allen anderen Gebieten ausgenommen ist.

Analoges gilt für selektierte Objekte (9.6.3).

# Programmstruktur, Übersetzungen und Ausführungen

In diesem Kapitel werden der Aufbau eines ausführbaren Programms, Eigenschaften der getrennten Übersetzung und die Ausführung eines Programms beschrieben.

## Programmstruktur

In Pascal-XT besteht ein Programm aus genau einem Hauptprogramm und einer beliebigen Anzahl von Paketen. Ein Programm kann als die Aneinanderreihung der Pakete und des Hauptprogramms aufgefaßt werden. Diese gedachte Aneinanderreihung aller zu einem Programm gehörigen Übersetzungseinheiten heißt Programmtext.

In Norm-Pascal besteht ein Programm nur aus dem Hauptprogramm.

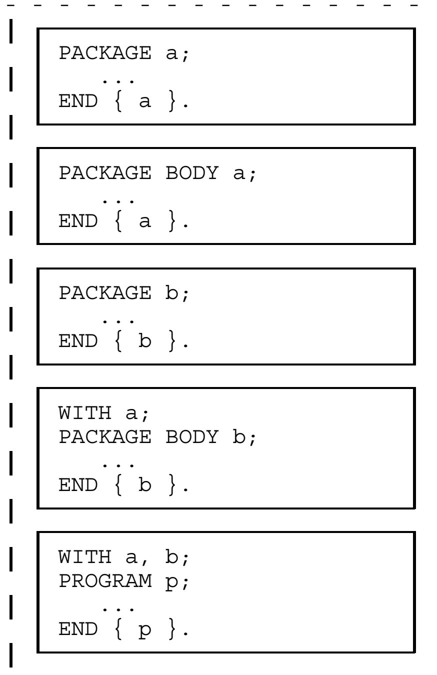



Bild 13-1: Beispiel eines Programmtextes

*Hinweis*

Zu einem Paket kann es mehrere unterschiedliche Paket-Implementierungen geben. Ein Programm kann aber immer nur eine Implementierung eines Pakets enthalten.

*Beispiel*

In diesem Beispiel wird das Paket stack aus Abschnitt 11.2 von einem Hauptprogramm verwendet.

```
WITH stack;
PROGRAM calc (Input);
VAR
   i, j: Integer;
BEGIN
   FOR i := 1 to 6 DO BEGIN
      Read (Input, j);
      stack.push (j);
      END;
   WHILE NOT stack.is_empty DO BEGIN
      stack.pop (j);
      { verarbeite j }
      END;
END { calc }.
```

# Übersetzungseinheiten und Übersetzungsreihenfolge

Paket-Spezifikationen, Paket-Implementierungen und Hauptprogramme sind Übersetzungseinheiten.

```
Übersetzungseinheit =  Paket-Spezifikation
                     | Paket-Implementierung
                     | Hauptprogramm .
```

Jede Übersetzungseinheit ist in einer eigenen Quell-Datei abgelegt und wird getrennt von allen anderen Übersetzungseinheiten übersetzt.

Die Regeln für die Übersetzungsreihenfolge der Übersetzungseinheiten kann direkt aus den Sichtbarkeitsregeln und den in den Kontext-Spezifikationen angegebenen Beziehungen abgeleitet werden:

- Eine Übersetzungseinheit muß nach allen Paket-Spezifikationen übersetzt werden, die in ihrer WITH-Liste aufgeführt sind.
- Eine Paket-Implementierung muß nach der zugehörigen Paket-Spezifikation übersetzt werden.

Eine Übersetzungseinheit ist abhängig von einer Paket-Spezifikation, wenn diese in der WITH-Liste der Übersetzungseinheit aufgeführt ist. Eine Änderung in einer Paket-Spezifikation (z.B. Änderung von Konstanten, Bezeichnern, Operatoren, WITH-Listen, usw.) macht alle abhängigen Übersetzungseinheiten ungültig. Nach der Übersetzung einer modifizierten Paket-Spezifikation müssen daher alle abhängigen Übersetzungseinheiten erneut übersetzt werden. Eine Änderung in einer Paket-Implementierung hat keine Auswirkungen auf andere Übersetzungseinheiten.

*Hinweis*

Die Art und Weise, wie Übersetzungseinheiten abgespeichert werden und wie der Compiler auf die in den WITH-Klauseln aufgeführten Paket-Spezifikationen zugreift, ist im Benutzerhandbuch [1] beschrieben.

*Beispiel*

```
PACKAGE a;
 ...
END.


WITH a;
PACKAGE b;
 ...
END.


WITH b;
PACKAGE c;
 ...
END.

PACKAGE d;
 ...
END.


WITH c, d;
PROGRAM beispiel;
 ...
BEGIN
END.
```

Eine Änderung der Paket-Spezifikation b macht eine Neuübersetzung der abhängigen Paket-Spezifikation c notwendig, und als Folge davon die Neuübersetzung des Hauptprogramms beispiel. Durch die Neuübersetzungen der Paket-Spezifikationen b und c müssen natürlich auch die zugehörigen müssen natürlich auch die zugehörigen Paket-Implementierungen Paket-Implementierungen neu übersetzt werden. Die Paket-Spezifikationen a und c sind von diesen Änderungen nicht betroffen.

*Querverweise*

Hauptprogramm: 11.1
Paket-Spezifikation: 11.2.1
Paket-Implementierung: 11.2.2
WITH-Liste: 11.3.1
Sichtbarkeitsregeln: 12.2

# Ausführen eines Programms bzw. Unterprogramms

## Ausführung eines Blocks

Zur Ausführung eines Blocks wird ein eigener Satz (Inkarnation) der in dem Block vereinbarten Variablen angelegt. Die in dieser Inkarnation angelegten Variablen sind verschieden von den Variablen jeder anderen Inkarnation desselben oder eines anderen Blocks. Insbesondere hat jede rekursive Inkarnation einer Prozedur bzw. Funktion ihren eigenen Satz von Variablen. Die Lebensdauer der in einer Inkarnation angelegten Variablen endet mit der Beendigung der Ausführung des Blocks (außer bei Paketblöcken, siehe 13.3.3). Die Ausführung eines Blocks besteht in der Ausführung der Anweisungen im Anweisungsteil und wird beendet durch

- die Beendigung der Ausführung der letzten Anweisung im Anweisungsteil des Blocks
- die Ausführung einer RETURN-Anweisung - die Ausführung einer GOTO-Anweisung, deren Sprungmarke außerhalb des Blocks vereinbart und definiert ist
- das Auftreten einer Ausnahme (z.B. durch Raise), zu der kein Ausnahmebehandler im Block existiert.

Im Gegensatz dazu leben dynamische Variable von ihrer Erzeugung durch einen Aufruf von New bis zum Ende der Ausführung des Hauptprogramms oder bis Verweiswerte auf sie durch einen Aufruf von Dispose bzw. von Release vernichtet werden.

*Querverweise*

| | |
|---|---|
| dynamische Variable: | 7.2, 9.6.4, 18.3 |
| GOTO-Anweisung: | 10.1.4 |
| RETURN-Anweisung: | 10.1.6 |
| Block: | 12.1 |
| Raise: | 14, 15.11 |
| New, Dispose, Release: | 15.2, 18.3 |

## Ausführung eines Unterprogramms

Die Ausführung eines Prozedur-Blocks bzw. Funktions-Blocks wird durch einen Prozedur- bzw. einen Funktionsaufruf veranlaßt.

*Querverweis*

Prozeduraufruf: 8.7
Funktionsaufruf: 8.7, 9.6.1

## Ausführen eines Programms

Analog zur Ausführung eines Unterprogramm-Blocks haben Variable, die unmittelbar in einem Hauptprogramm-Block oder einem Paket-Block vereinbart sind, als Lebensdauer die gesamte Dauer der Ausführung des Programms.

Die Ausführung des Programms beginnt mit der Initialisierung aller zu dem Programm gehörigen Pakete gefolgt von der Ausführung des Hauptprogramm-Blocks.

Die Initialisierung eines Pakets ist die Ausführung des Paket-Blocks. Jedes Paket wird genau einmal initialisiert. Soll während der Ausführung des Anweisungsteils eines Paket-Blocks auf Variablen, die in einer fremden Paket-Spezifikation deklariert sind, zugegriffen werden, oder sollen Prozeduren und Funktionen, die in einer fremden Paket-Spezifikation deklariert sind, aufgerufen werden, so muß vorher das fremde Paket initialisiert worden sein. Die Initialisierung der Pakete erfolgt in einer implementierungsabhängigen Reihenfolge. Um die rechtzeitige Initialisierung eines Pakets zu erzwingen, kann die vordefinierte Prozedur Elaborate (siehe 15.12) aufgerufen werden.

### Mögliche Laufzeitfehler

| | |
|---|---|
| Memory_Error | - Die Ausführung des Programms kann wegen fehlenden Speichers nicht fortgesetzt werden (z. B. bei Aufruf eines Unterprogramms oder bei New). |
| Elab_Error | - Die Initialisierung der Pakete eines Programms kann nicht fortgesetzt werden, da durch die Verwendung der vordefinierten Prozedur Elaborate Zyklen bei der Initialisierung entstehen. |
| undefinierte Auswirkungen | - Die Bezeichner der Programmparameter des Hauptprogramms und aller dazugehörigen Pakete sind, mit Ausnahme von Input und Output, nicht paarweise verschieden.<br>- Die Namen aller zu einem Programm gehörigen Pakete und der Name des Hauptprogramms sind nicht paarweise verschieden. |

*Querverweise*

| | |
|---|---|
| Hauptprogramm-Block: | 11.1 |
| Paket-Block: | 11.2 |
| Paket-Spezifikation: | 11.2 |
| Block: | 12.1 |
| Elaborate: | 15.12 |

# Ausnahmebehandlung

In diesem Kapitel werden die Spracherweiterungen in Pascal-XT beschrieben, die der Behandlung von Fehlern und Ausnahmesituationen, die während der Ausführung eines Programms auftreten können, dienen.

Eine Ausnahme (exception) ist ein Ereignis, das den normalen Programmablauf unterbricht. Sie kann implizit während der Ausführung einer Anweisung oder explizit durch Aufruf der vordefinierten Prozedur Raise auftreten (ausgelöst werden). Die Reaktion auf eine Ausnahme wird als Ausnahmebehandlung bezeichnet. Auszuführende Aktionen können im Ausnahmebehandlungs-Teil (EXCEPTION-Teil) einer Verbundanweisung angegeben sein. Tritt eine Ausnahme auf, dann wird die Ausführung des Programms im Ausnahmebehandlungs-Teil (falls vorhanden) der Verbundanweisung fortgesetzt.

Mit der vordefinierten Prozedur Raise (siehe 15.11) kann eine Ausnahmesituation durch den Anwender erzeugt oder eine bereits aufgetretene Ausnahme propagiert werden. Die Nummer der Ausnahme wird als Parameter übergeben.

Nach Auftreten einer Ausnahmesituation kann mit der vordefinierten Funktion Error_Number die zuletzt aufgetretene Ausnahme abgefragt werden.

*Querverweise*

Verbundanweisung: 10.2
Raise: 15.11
Error_Number: 15.11

# Vordefinierte und benutzerdefinierte Ausnahmen

Eine Ausnahme wird durch eine Integer-Zahl repräsentiert. Die negativen Zahlen einschließlich der Null sind für die Pascal-XT Implementierungen reserviert, die positiven Zahlen können von den Anwendern frei verwendet werden.

Die reservierten Ausnahmen mit den Werten -1 bis -16 sind als Konstanten mit vordefinierten Bezeichnern verknüpft (siehe auch 5.2). Diese Ausnahmen treten bei folgenden Situationen auf (für Details siehe Anhang A.5):

| | | |
|---|---|---|
| Numeric_Error | = -2 | Tritt bei Überlauf während der Berechnung eines arithmetischen Ausrucks oder einer vordefinierten arithmetischen Funktion auf, bzw. bei der Zuweisung eines Realwertes an eine Short_Real-Variable. |
| Range_Error | = -3 | Tritt auf, wenn ein Ordinalwert nicht im erlaubten Wertebereich liegt. |
| Set_Error | = -4 | Tritt auf, wenn nicht alle Elemente der Menge im Wertebereich des betreffenden SET-Typs liegen. |
| String_Error | = -5 | Tritt bei der Bearbeitung von Zeichenketten auf, wenn Bedingungen bzgl. der Längen verletzt werden. |
| Index_Error | = -6 | Tritt bei der Indizierung von Objekten auf, wenn der Wert des Indexausdrucks nicht im erlaubten Wertebereich liegt. |
| Pointer_Error | = -7 | Tritt bei der Dereferenzierung eines Zeiger-Objekts mit dem Zeigerwert NIL auf. |
| Variant_Error | = -8 | Tritt beim Zugriff auf eine Komponente in einer nicht eingestellten Variante eines RECORD-Typs auf. Der Fehler tritt nur auf, wenn der Variantteil, der diese Variante enthält, ein Kennungsfeld besitzt. |
| Case_Error | = -9 | Tritt in einer CASE-Anweisung auf, wenn es zu dem Wert des Fallindex keine Fallkonstante gibt und kein ELSE-Teil angegeben ist. |
| File_Error | = -10 | Tritt beim Zugriff auf eine Datei auf, wenn diese sich nicht im richtigen Modus befindet oder bei implementierungsabhängigen Einschränkungen (z. B. interner Puffer ist zu klein). |
| Eof_Error | = -11 | Tritt auf, wenn versucht wird über das Dateiende hinaus zu lesen. |
| Open_Error | = -12 | Tritt auf, wenn eine Datei nicht eröffnet werden kann z.B. Datei nicht vorhanden, Zugriffsschutz etc.). |

| | | |
|---|---|---|
| Read_Error | = -13 | Tritt beim Lesen (Read, Readstring) einer Integer-Zahl bzw. einer Real-Zahl auf, wenn die Zahl syntaktisch fehlerhaft oder ihr Wert zu groß ist. |

| | | |
|---|---|---|
| Memory_Error | = -14 | Tritt bei der Ausführung eines Programms auf, wenn nicht genügend Hauptspeicher (z. B. bei Aufrufen von Unterprogrammen bzw. der vordefinierten Prozedur New) vorhanden ist. |
| Break_Error | = -15 | Tritt auf, wenn die Ausführung eines Programms durch den Anwender unterbrochen wird. |
| Elab_Error | = -16 | Tritt auf, wenn die Initialisierung der Pakete eines Programms nicht fortgesetzt werden kann, da durch die Verwendung der vordefinierten Prozedur Elaborate Zyklen bei der Initialisierung entstehen. |
| System_Error | = -1 | Tritt als Sammelmeldung für sonstige Fehler auf. |

Die Erkennung dieser vordefinierten Ausnahmen wird durch die Pascal-XT Implementierung, evtl. in Abhängigkeit von der Check-Option, festgelegt.

*Hinweise*

– Bei ausgeschalteter Check-Option (siehe Kapitel 16) werden die Fehler Numeric_Error, Range_Error, Set_Error, String_Error, Index_Error, Pointer_Error, Variant_Error und Case_Error i.a. nicht erkannt und haben undefinierte Auswirkungen zur Folge (siehe 2.3). Eine Pascal-XT Implementierung kann implementierungsabhängig aber auch Fehler bei ausgeschalteter Check-Option erkennen.

– Ein Pointer_Error tritt oftmals auch als Folgefehler auf, wenn z.B. durch Programmfehler Speicher überschrieben wurde oder Zeiger-Objekte, deren Wert undefiniert ist, dereferenziert werden.

– In einem Programm sollten alle benutzerdefinierten Ausnahmenummern paarweise verschieden sein. Nur dadurch kann in einem Ausnahmebehandlungs-Teil die Herkunft der Ausnahme eindeutig bestimmt werden.

*Querverweise*

| | |
|---|---|
| Fehler: | 2.3 |
| Vordefinierte Ausnahmen: | 5.2, A.5 |
| Zuweisungsverträglichkeit: | 6.6.3 |
| Programmausführung: | 13.3 |
| Formatangabe: | 15.1 |
| Check-Option: | 16.2 |

# Ausnahmebehandlungs-Teil (EXCEPTION-Teil)

Die Reaktion auf eine oder mehrere Ausnahmen kann in einem Ausnahmebehandlungs-Teil (EXCEPTION-Teil) festgelegt werden. Der Ausnahmebehandlungs-Teil ist eine Erweiterung der Verbundanweisung. Er besteht aus einer Anweisungsfolge, die nach dem Schlüsselwort EXCEPTION steht. Die Syntax für die erweiterte Verbundanweisung lautet somit:

```
Verbundanweisung = "BEGIN"
                    Anweisungsfolge
                    [ EXCEPTION-Teil ]
                   "END".

EXCEPTION-Teil   = "EXCEPTION" Anweisungsfolge.
```

Ein EXCEPTION-Teil kann also im Anweisungsteil (Verbundanweisung) eines Unterprogramms, eines Hauptprogramms oder einer Paket-Implementierung und in jeder anderen Verbundanweisung angegeben werden.

Tritt während der Abarbeitung der Anweisungsfolge einer Verbundanweisung mit EXCEPTION-Teil keine Ausnahme auf, dann werden die Anweisungen im Ausnahmebehandlungs-Teil nicht ausgeführt.

Tritt während der Abarbeitung der Anweisungsfolge einer Verbundanweisung mit EXCEPTION-Teil eine Ausnahme auf, dann werden alle folgenden Anweisungen bis zum Schlüsselwort EXCEPTION übersprungen, und die Abarbeitung wird mit der ersten Anweisung innerhalb des Ausnahmebehandlungs-Teils fortgesetzt. Nach normaler Abarbeitung der Anweisungsfolge im Ausnahmebehandlungs-Teil gilt die Ausnahmesituation als behandelt, und die Abarbeitung der Verbundanweisung wird normal beendet. Tritt während der Abarbeitung der Anweisungen im Ausnahmebehandlungs-Teil erneut eine Ausnahmesituation ein oder wird eine weitere Ausnahme mittels Raise erzeugt, dann wird die Ausnahme propagiert (siehe 14.3).

*Beispiel 14-1*

```
VAR
   source : Text;
   s      : String;

BEGIN
   Reset (source);                                   {1}  {2}{3}
   WHILE NOT Eof (source) DO BEGIN                   {1}     {3}
      Read    (source, s);                           {1}     {3}
      Writeln (Output, s);                           {1}
      Readln  (source);                              {1}
      END;                                           {1}
EXCEPTION
   IF Error_Number = Open_Error THEN                      {2}{3}
      Writeln (Output, 'Fehler beim Oeffnen der Datei!') {2}
   ELSE IF Error_Number = String_Error THEN                  {3}
      Writeln (Output, 'Gelesene Zeile ist zu lang')         {3}
   ELSE
      Raise (0);
END;
```

In diesem Beispiel sind in Kommentarklammern die Zeilen markiert, die in verschiedenen Situationen abgearbeitet werden:

{1} Anweisungen, die im Normalfall ausgeführt werden.

{2} Anweisungen, die im Fall eines Fehlers bei Reset (Open_Error) ausgeführt werden. Die Ausnahme gilt als behandelt.

{3} Anweisungen, die beim Einlesen einer zu langen Zeile (String_Error) ausgeführt werden. Die Ausnahme gilt als behandelt und der Lesevorgang wird beendet.

Alle anderen Ausnahmen werden durch Raise (0) propagiert, da sie nicht behandelt werden können.

*Hinweis*

Kann im Ausnahmebehandlungs-Teil eine Ausnahme nicht behandelt werden, so sollte dieselbe Ausnahme mit Raise (0) propagiert werden, da sonst eine nicht behandelte Ausnahmesituation fälschlich als behandelt betrachtet wird.

*Querverweise*

Prozedur-Block: 8.1
Funktions-Block: 8.2
Verbundanweisung: 10.2
Hauptprogramm-Block: 11.1
Paket-Implementierung: 11.2
Paket-Block: 11.2
Block: 12.1

# Behandlung von Ausnahmen

Die Behandlung einer Ausnahme, die bei der Abarbeitung einer Anweisungsfolge in einer Verbundanweisung auftritt, hängt davon ab, ob die Verbundanweisung einen Ausnahmebehandlungs-Teil enthält. Existiert ein Ausnahmebehandlungs-Teil, dann wird die Anweisungsfolge innerhalb dieses Ausnahmebehandlungs-Teils abgearbeitet.

Enthält die Verbundanweisung keinen Ausnahmebehandlungs-Teil bzw. wenn im Ausnahmebehandlungs-Teil wieder eine Ausnahme auftritt, dann werden folgende Fälle unterschieden:

1) Eine Ausnahmesituation in einer Verbundanweisung, die nicht den gesamten Anweisungsteil eines Blocks darstellt, wird nochmals unmittelbar hinter dem Schlüsselwort END dieser Verbundanweisung erzeugt (siehe Beispiel 14.2).
   Bei geschachtelten Verbundanweisungen kann der zuständige Ausnahmebehandlungs-Teil anhand der statischen Aufschreibung des Programms bestimmt werden.

2) Eine Ausnahmesituation in der Verbundanweisung (Anweisungsteil) eines Prozedur-Blocks bzw. Funktions-Blocks wird nochmals an der Aufrufstelle des Unterprogramms erzeugt (siehe Beispiel 14-3). Der zuständige Ausnahmebehandlungs-Teil kann also nicht aus der statischen Aufschreibung bestimmt werden, sondern muß durch Rückverfolgung der Unterprogrammaufrufe dynamisch ermittelt werden.

3) Eine Ausnahmesituation in der Verbundanweisung (Anweisungsteil) eines Hauptprogramm-Blocks oder Paket-Blocks führt zum Programmabbruch. Der Fehler wird durch das jeweilige Betriebs- bzw. Laufzeitsystem behandelt.

Eine Ausnahme wird durch den Aufruf von Raise (Error_Number) oder Raise (0) propagiert. Raise (Error_Number) erzeugt dieselbe Ausnahme nochmals und verändert den Ursprungsort der Ausnahme. Raise (0) dagegen propagiert dieselbe Ausnahme ohne den Ursprungsort der Ausnahme zu verändern (siehe 15.11 und die Beispiele 14-4 und 14-5).

*Beispiel 14-2*

```
VAR
   i : Long_Integer;

BEGIN
   FOR i := 1 TO 5 DO BEGIN
      Reset (datei);
      EXIT;
   EXCEPTION
      IF (i < 5) AND (Error_Number = Open_Error) THEN
         warte
      ELSE
         Raise (Error_Number);
   END;                              {1}

   { Bearbeitung der Datei }
   ...
END                                  {1}
```

Wird im EXCEPTION-Teil Raise aufgerufen, dann wird nach Regel 1 die Ausnahme an den mit {1} markierten Stellen nochmals erzeugt.

*Beispiel 14-3*

```
PROCEDURE r;
BEGIN
   ...
END {r};

PROCEDURE q;
BEGIN
   r;
EXCEPTION
   ...               { Ausnahmebehandlung A2 }
END {q};

PROCEDURE p;
BEGIN
   q;
EXCEPTION
   ...               { Ausnahmebehandlung A1 }
END {p};
```

Folgende Situationen können auftreten:

1) Tritt eine Ausnahme im Anweisungsteil der Prozedur p auf, dann ist der Ausnahmebehandlungs-Teil A1 zuständig.

2) Tritt eine Ausnahme im Anweisungsteil der Prozedur q auf, die von p gerufen wird, dann ist der Ausnahmebehandlungs-Teil A2 zuständig. Nach normaler Beendigung von A2 wird die Kontrolle an die Aufrufstelle von q (im Anweisungsteil von p) zurückgegeben. Tritt in A2 wieder eine Ausnahme auf, dann wird q abgebrochen und die Ausnahme nochmals an der Aufrufstelle von q (in p) erzeugt. Dort ist dann der Ausnahmebehandlungs-Teil A1 zuständig.

3) Tritt ein Fehler in der Prozedur r auf, die von q gerufen wird, dann wird r abgebrochen und die Ausnahme nochmals an der Aufrufstelle in q erzeugt. Die Ausnahme wird in A2 behandelt.

*Hinweis*

Beim Auftreten einer Ausnahme ist darauf zu achten, daß im Ausnahmebehandlungs-Teil ggf. Aktionen zur Sicherung der Konsistenz der Daten durchzuführen sind. Insbesondere muß in der Verbundanweisung einer Funktion vor dem Verlassen des Ausnahmebehandlungs-Teils sichergestellt sein, daß dem Funktions-Bezeichner ein Wert zugewiesen wurde.

*Beispiel 14-4*

Fehlerpropagierung durch Raise (Error_Number). Damit geht der ursprüngliche Fehlerort verloren.

```
PROGRAM quadr_equation (Input, Output);

PROCEDURE get_value (name: String; VAR value: Real);
BEGIN
   Writeln ('Bitte den Wert fuer ', name, ' eingeben:');
   Read (value); ——————————————————————————————————————————— (1)
END;

PROCEDURE solve;
VAR
  p, q, d: Real;
BEGIN
   Writeln;
   Writeln ('Quadratische Gleichung  x**2 + p*x + q = 0');
   get_value ('p', p);
   get_value ('q', q);
   d := Sqrt (Sqr (p) / 4 - q);
   IF d = 0
      THEN Writeln ('x = ', -p / 2)
      ELSE Writeln ('x = ', -p / 2 + d, ' oder ', -p / 2 - d);
EXCEPTION
   IF Error_Number = Numeric_Error
      THEN Writeln ('Die Gleichung hat keine Loesung')
      ELSE Raise (Error_Number); ————————————————————————————— (2)
END;

BEGIN (*PROGRAM quadr_equation*)
   WHILE True
      DO solve;
EXCEPTION
   IF Error_Number = Eof_Error
      THEN Writeln ('Auf Wiedersehen')
      ELSE Raise (Error_Number); ————————————————————————————— (3)
END.
```

(1) Tritt beim Lesen der Zahl ein Fehler auf, dann wird ein Read_Error ausgelöst. Der Ursprungsort des Fehlers liegt also in der Prozedur get_value (1).

(2) Der Read_Error wird propagiert, d.h. neu ausgelöst. Damit ist der ursprüngliche Fehlerort (1) verloren und (2) ist der neue Fehlerort.

(3) Verhalten wie bei (2). Der neue Fehlerort ist nun das Hauptprogramm (3).

Diese Art der Fehlerpropagierung erschwert die Fehlerdiagnose, da der Fehlerort verlorengegangen ist. Die Reaktion des Programms ist im Benutzerhandbuch ausführlich beschrieben.

*Beispiel 14-5*

Mit Raise (0) wird dieselbe Ausnahme propagiert, wobei der ursprüngliche Fehlerort für die Fehlerdiagnose erhalten bleibt. Raise (0) löst keine Ausnahme mit der Nummer 0 aus.

```
PROGRAM quadr_equation (Input, Output);

PROCEDURE get_value (name: String; VAR value: Real);
BEGIN
   Writeln ('Bitte den Wert fuer ', name, ' eingeben:');
   Read (value); ---------------------------------------------  (1)
END;

PROCEDURE solve;
VAR
  p, q, d: Real;
BEGIN
   Writeln;
   Writeln ('Quadratische Gleichung  x**2 + p*x + q = 0');
   get_value ('p', p);
   get_value ('q', q);
   d := Sqrt (Sqr (p) / 4 - q);
   IF d = 0
      THEN Writeln ('x = ', -p / 2)
      ELSE Writeln ('x = ', -p / 2 + d, ' oder ', -p / 2 - d);
EXCEPTION
   IF Error_Number = Numeric_Error
      THEN Writeln ('Die Gleichung hat keine Loesung')
      ELSE Raise (0); ---------------------------------------  (2)
END;

BEGIN (*PROGRAM quadr_equation*)
   WHILE True
      DO solve;
EXCEPTION
   IF Error_Number = Eof_Error
      THEN Writeln ('Auf Wiedersehen')
      ELSE BEGIN
         Writeln ('Programm-Abbruch');
         Raise (0); ------------------------------------------  (3)
         END;
END.
```

(1) Es trete beim Lesen der Zahl wieder ein Read_Error auf.

(2) Die Ausnahme wird durch Raise (0) propagiert. Dabei bleibt der ursprüngliche Fehlerort (1) in Prozedur get_value erhalten.

(3) Verhalten wie bei (2). Der Ursprungsfehlerort (1) bleibt erhalten.

Diese Art der Fehlerpropagierung erleichtert die Fehlerdiagnose erheblich gegenüber Beispiel 14-4. Die Reaktion des Programms ist im Benutzerhandbuch ausführlich beschrieben.

*Querverweise*


| | |
|---|---|
| Prozedur-Block: | 8.1 |
| Funktions-Block: | 8.2 |
| Verbundanweisung: | 10.2 |
| Hauptprogramm-Block: | 11.1 |
| Paket-Implementierung: | 11.2 |
| Paket-Block: | 11.2 |
| Block: | 12.1 |
| Error_Number | 15.11 |
| Raise | 15.11 |

# Ausnahmebehandlung und Optimierung

In diesem Abschnitt werden die Bedingungen beschrieben, unter denen eine Implementierung Anweisungen und Ausdrücke in einer anderen Reihenfolge abarbeiten darf, als durch die Sprache vorgegeben ist.

Soweit in der Sprache Pascal-XT Regeln für die Reihenfolge (kanonische Reihenfolge) gewisser Aktionen angegeben sind, darf eine Implementierung nur dann eine alternative Reihenfolge wählen, wenn die Wirkung des Programms dadurch nicht geändert wird. Insbesondere darf während der Abarbeitung der alternativen Reihenfolge keine Ausnahmesituation auftreten, wenn diese nicht auch während der Abarbeitung in der kanonischen Reihenfolge auftreten würde.

Tritt aber während der Abarbeitung in der kanonischen Reihenfolge eine Ausnahmesituation auf, so darf ein Programm sich nicht darauf verlassen, daß die Ausdrücke in einer Verbundanweisung in der kanonischen Reihenfolge ausgewertet werden. Da bei der Optimierung Ausdrücke in einer anderen Reihenfolge als der kanonischen abgearbeitet werden können, kann während dieser vorzeitigen Auswerung eine andere Ausnahme auftreten, als dies ohne Optimierung der Fall wäre. Es ist jedoch garantiert, dkaß durch die Optimierung keine zusätzlichen Ausnahmesituationen auftreten.

*Beispiel*

```
VAR
   i,n : Integer;

BEGIN
   n := 0;
   FOR i := 1 TO 5 DO
      n := n + i ** f[a];  { f und a seien global }
EXCEPTION
   IF Numeric_Error THEN
      Writeln (Output, 'Fehler bei der Berechnung')
   ELSE
      Writeln (Output, 'unbekannter Fehler')
END.
```

f und a seien globale Variable. Die Auswertung von f[a] ist unabhängig von von der FOR-Schleife und kann daher vor der Schleife und auch noch vor der Zuweisung "n := 0" durchgeführt werden, auch wenn sie zu einer Ausnahmesituation führt. Innerhalb des Ausnahmebehandlungs-Teils kann der Wert von n demzufolge undefiniert sein. Die Auswertung von f[a] kann aber nicht vor dem BEGIN erfolgen, da sonst eine auftretende Ausnahme durch einen anderen Ausnahmebehandlungs-Teil behandelt würde.

# Vordefinierte Unterprogramme

In diesem Kapitel werden alle vordefinierten Unterprogramme (Prozeduren und Funktion), eingeteilt in thematisch zusammengehörende Gruppen, beschrieben. Innerhalb jeder Gruppe sind die Unterprogramme alphabetisch sortiert. Im Anhang A.4 befindet sich eine alphabetisch sortierte Liste aller vordefinierten Unterprogramme.

Die Bezeichner der vordefinierten Unterprogramme sind keine Wortsymbole (Schlüsselworte), sie können daher auch umdefiniert werden.

Die vordefinierten Unterprogramme können ähnlich wie benutzerdefinierte Unterprogramme aufgerufen werden:

**Standardprozeduren/funktionen dürfen jedoch nicht als Parameterprozeduren/funktionen übergeben werden**.

Außerdem gehorchen diese Unterprogramme nicht notwendigerweise allen Regeln (siehe Kap.8), wie sie für benutzerdefinierte Unterprogramme spezifiziert sind.

# Unterprogramme zur Dateiverarbeitung

### Assignfile (f, ext)

Mit dieser Prozedur wird einer FILE-Variablen f eine Datei zugeordnet, die außerhalb des Programms existiert.

f ist eine Variable eines beliebigen FILE-Typs.
ext ist ein Zeichenkettenausdruck und beinhaltet die implementierungsdefinierte Beschreibung der physikalischen Datei, die der FILE-Variablen zugeordnet wird (siehe Benutzerhandbuch [1]).

Assignfile bewirkt gegebenenfalls implizit das Schließen einer Datei, die der FILE-Variablen f bisher zugeordnet ist.

#### Implementierungsdefinierte Eigenschaft

Die Beschreibung der physikalischen Datei in der vordefinierten Prozedur Assignfile und die Wirkung dieser Prozedur sind implementierungsdefiniert.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| `File_Error` | `- Bei Assignfile (f, ext) ist die Beschreibung der physikalischen Datei im Operanden "ext" fehlerhaft.` |

*Hinweis*

In Norm-Pascal können nur den in der Programmparameterliste angegebenen Dateien physikalische Dateien zugeordnet werden. Mittels Assignfile ist das mit beliebigen FILE-Variablen möglich, also auch für lokale und mit New dynamisch erzeugte FILE-Variablen.

*Querverweise*

| | |
|---|---|
| FILE-Typ: | 6.3.5 |
| Variable: | 7.2 |
| Ein- / Ausgabe: | 16 |

**Eof (f)**
**Eof**

Die Funktion Eof (f) liefert den booleschen Wert True, wenn das Ende der Datei f (End of File) erreicht ist, ansonsten den booleschen Wert False.

Wird der Parameter f nicht angegeben, dann wird die Funktion auf die vordefinierte Textdatei Input angewandt.

Wurde die Datei zum Schreiben eröffnet, so liefert die Funktion Eof immer den Wert True.

Nach Reset ist Eof True, falls die Datei leer ist. Nach Get ist Eof True, falls vor Get bereits die letzte Komponente der Datei in der Puffervariable war. In allen anderen Fällen ist Eof False. Wenn Eof True ist, so verursachen Aufrufe von Get, Read, Readln und Eoln den Laufzeitfehler Eof_Error.

Wenn Eof True ist, dann ist die Puffervariable f↑ undefiniert.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| `File_Error` | `- Vor dem Aufruf von Eof (f) ist die Datei f undefiniert (siehe 7.3).` |

*Querverweise*

Puffervariable: 8.2, 16.1
Get: 15.1
Definierte Variable: 7.3

**Eoln (f)**
**Eoln**

Eoln ist eine Funktion zum Erkennen der Zeilenende-Komponente in Textdateien. Eoln ist nur auf Textdateien, die zum Lesen eröffnet wurden, anwendbar. Fehlt der Parameter f, dann wird die Funktion auf die vordefinierte Textdatei Input angewandt.

Die Funktion Eoln ergibt den Wert True, wenn zuletzt die Zeilenende-Komponente gelesen wurde. Die Puffervariable enthält dann ein Blank (' '). Andernfalls liefert Eoln den Wert False.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Vor dem Aufruf von Eoln (f) ist die Datei f undefiniert. |
| Eof_Error | - Beim Aufruf von Eoln (f) ist das Dateiende bereits erreicht, d.h. Eof (f) ist True. |

*Querverweise*

Textdateien: 6.3.5.2, 16
Zeilenende-Komponente: 16.1

## Get (f)

Mit Get (f) wird die nächste Komponente aus der Datei f gelesen und in die Puffervariable f↑ übernommen. f kann eine Variable eines beliebigen FILE-Typs sein, die zum Lesen eröffnet wurde. Eof (f) muß vor dem Aufruf von Get (f) False sein.

Wurde eine Komponente gelesen, dann hat Eof (f) den Wert False. Konnte keine weitere Komponente gelesen werden (Dateiende), so ist die Puffervariable f↑ undefiniert und Eof (f) hat den Wert True.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Vor dem Aufruf von Get (f) wurde die Datei f nicht zum Lesen eröffnet.<br>- Vor dem Aufruf von Get (f) ist die Datei f undefiniert. |
| Eof_Error | - Vor dem Aufruf von Get (f) war Eof (f) True. |

*Beispiel*

Die Komponenten der Datei f werden sequentiell eingelesen und verarbeitet. Durch Reset (f) wird bereits die erste Komponente in die Puffervariable gelesen.

```
VAR
   f : FILE OF komponenten_typ;
BEGIN
   Reset (f);
   WHILE NOT Eof (f) DO BEGIN
      verarbeiten (f↑);
      Get (f);
      END;
END;
```

## Page (f)
## Page

Die Prozedur Page dient zur Seitenformatierung von Textdateien. Der Text, der nach dem Aufruf von Page (f) ausgegeben wird, kommt auf eine neue Seite, falls das Ausgabegerät die Möglichkeit einer Seitensteuerung besitzt. Die Art, wie der Seitenvorschub erzeugt wird, ist implementierungsdefiniert (siehe Benutzerhandbücher [1,2]). Ist vor Aufruf von Page eine Ausgabezeile noch nicht mit Writeln (f) abgeschlossen, dann führt Page implizit dieses Writeln (f) aus.

Page ist nur anwendbar auf eine Textdatei, die zum Schreiben eröffnet wurde. Die Prozedur wird auf die vordefinierte Textdatei Output angewendet, wenn keine FILE-Variable angegeben wurde.

### Implementierungsdefinierte Eigenschaft

Die Wirkung der vordefinierten Prozedur Page auf Textdateien ist implementierungsdefiniert.

### Implementierungsabhängige Eigenschaft

Die Wirkung beim Lesen einer Textdatei, während deren Generierung die vordefinierte Prozedur Page angewendet wurde, ist implementierungsabhängig. Für eine Implementierung kann aber die Wirkung definiert sein.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Die Datei f wurde vor dem Aufruf von Page nicht zum Schreiben eröffnet.<br>- Vor dem Aufruf von Page (f) ist die Datei f undefiniert. |

### Put (f)

Die Prozedur Put (f) ist auf eine zum Schreiben eröffnete FILE-Variable anwendbar. Der aktuelle Wert der Puffervariablen f↑ wird am Ende der bearbeiteten Datei angehängt. Nach dem Aufruf von Put ist die Puffervariable f↑ undefiniert und Eof bleibt True.

Die Prozedur Put bietet außerdem die Möglichkeit, Komponenten verkürzt zu schreiben. Diese Aufrufform ist weiter unten erklärt.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Die Datei f wurde vor dem Aufruf von Put nicht zum Schreiben eröffnet<br>- Vor dem Aufruf von Put (f) ist die Datei f undefiniert. |
| undefinierte Auswirkungen | - Der Wert der Puffervariablen f↑ ist nicht definiert.<br>- Bei der verkürzten Ausgabe eines ARRAYs mit Put (f, e) oder Put (f, c1, ..., cn, e) liegt der Wert des Indexausdrucks e nicht im Wertebereich des Indextyps des ARRAYs.<br>- Bei der verkürzten Ausgabe einer Zeichenkette variabler Länge mit Put (f, e) oder Put (f, c1, ..., cn, e) ist der Wert des Indexausdrucks e kleiner als 1 oder größer als die aktuelle Länge der Zeichenkette. |

*Beispiel*

Die Datei f wird sequentiell mit zuvor berechneten Werten geschrieben.

```
VAR
   f : FILE OF komponenten_typ;
BEGIN
   Rewrite (f);
   WHILE NOT fertig DO BEGIN
      f↑ := neuer_wert;
      Put (f);
      END;
END;
```

### Put (f, e)

Der Basistyp des FILE-Typs f muß ein ARRAY-Typ, ein String-Typ oder ein RECORD-Typ, dessen letztes Feld seiner Feldliste einen ARRAY-Typ oder einen String-Typ besitzt, sein.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Tyup muß der Ausdruck e einen Integer-Typ besitzen und sein Wert darf nicht kleiner als 1 oder größer als die aktuelle Länge des Strings sein.

Eine Implementierung kann bei einem solchen Aufruf von Put die Komponente verkürzt in die Datei schreiben. Bei einem ARRAY [m..n] werden nur die Komponenten m bis e in die Datei geschrieben. Bei einer Zeichenkette muß ihre aktuelle Länge ≤ e sein.

*Beispiel*

```
VAR
   h : FILE OF ARRAY [1..100] OF Real;
   a : Integer;
   s : FILE OF String;

BEGIN
   ...

   Put (h, 45); { Schreibt mindestens die ersten 45 Real-Werte }

   a := 23;
   Put (h, a);  { Schreibt mindestens die ersten 23 Real-Werte }

   Put (s, a);  { Schreibt s↑, falls length (s↑) ≤ 23 }
END;
```

### Put (f, c1, ..., cn)

Der Basistyp des FILE-Typs f muß ein RECORD-Typ sein. Er muß ineinandergeschachtelte Varianten enthalten, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung von Variantenteile definierten Reihenfolge aufgezählt sein. Für den Wert von f ↑ müssen diese Varianten eingestellt sein. Varianten, die nicht aufgeführt sind, müssen in einer tieferen Verschachtelungstiefe stehen als cn. Eine Implementierung kann bei einem solchen Aufruf von Put einen verkürzten Satz in die Datei schreiben, der nur Platz für die RECORD-Fehler dieser Varianten beansprucht.

*Beispiel*

Gegeben ist folgender RECORD-Typ, der aus ineinander geschachtelten Varianten besteht, auf die in den folgenden Beispielen verwiesen wird.

```
TYPE
  char_range = '0'..'1';
  t = RECORD
         t1: Integer;
         CASE t2: Boolean OF
            False:
               (t21: Real;
                CASE t22: char_range OF
                   '0': (t221: ARRAY [0..10] OF Integer);
                   '1': (t222: Short_Integer));
            True: (t23: Char);
      END;
VAR
  g: FILE OF t;
```

Die folgenden Put sind möglich. In den Kommentaren ist angegeben, wie die Länge berechnet wird. Das Schreiben findet nur dann verkürzt statt, wenn es in der jeweiligen Implementierung vorgesehen ist.

```
Put (g, False);      { Schreibt t1, t2, t21, t22, (t221 bzw. t222) }
Put (g, False, '0'); { Schreibt t1, t2, t21, t22, t221 }
Put (g, False, '1'); { Schreibt t1, t2, t21, t22, t222 }
Put (g, True);       { Schreibt t1, t2, t23 }
```

### Put (f, c1, ..., cn, e)

Der Basistyp des FILE-Typs f muß ein RECORD-Typ mit ineinandergeschachtelten Varianten sein, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachteltung der Varianteile definierten Reihenfolge aufgezählt sein. Die Feldliste der zur letzten Selektorkonstanten cn gehörigen Variante darf keine weiteren eingeschachtelten Varianten enthalten und das letzte Feld dieser Feldliste muß einen ARRAY- oder String-Typ besitzen.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen und sein Wert darf nicht kleiner als 1 oder größer als die aktuelle Länge des Strings sein.

Eine Implementierung kann bei einem solchen Aufruf von Put einen verkürzten Satz in die Datei schreiben, der nur die RECORD-Felder der durch c1 ... cn bestimmten Varianten enthält. Ist das letzte RECORD-Feld ein ARRAY [m..n], dann werden nur die Komponenten m bis e in die Datei geschrieben. Ist das letzte RECORD-Feld ein String, so muß seine aktuelle Länge $\leq$ e sein.

*Beispiel*

```
Put (g, False, '0', 2); { Schreibt t1, t2, t21, t22, }
                        {  t221[0], t221[1], t221[2] }
```

**Read (f, v1, ..., vn)**

Die Prozedur Read dient zum Einlesen von Werten aus einer Datei f in die Variablen v1 bis vn.

Die Datei-Variable f kann von einem beliebigen FILE-Typ sein, außer dem vordefinierten FILE-Typ Any_File. Die Datei f muß zum Lesen eröffnet sein und Eof (f) muß False sein.

Für die Anzahl der Parameter vi gilt $n \geq 1$.

Read (f, v1, ..., vn)

ist definiert durch:

BEGIN Read (f, v1); ...; Read (f, vn); END

Die Parameter vi müssen zwar verallgemeinerte Variablen sein, sind aber keine Variablenparameter. Sie können daher auch Komponenten gepackter Strukturen sein.

- **Read - von einer Nicht-Textdatei**

**Read (f, v)**

Ist die FILE-Variable f nicht von dem vordefinierten FILE-Typ Text, dann ist

Read (f, v)

definiert durch:

```
BEGIN    v := f↑; Get (f); END
```

v muß eine verallgemeinerte Variable sein. Der Wert der Puffervariablen f↑ muß zuweisungsverträglich zum Typ von v sein.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Vor dem Aufruf von Read (f, ...) wurde die Datei f nicht zum Lesen eröffnet.<br><br>- Vor dem Aufruf von Read (f, ...) ist die Datei f undefiniert. |
| Eof_Error | - Beim Aufruf von Read (f, ...) ist das Dateiende bereits erreicht, d.h., Eof (f) ist True. |
| Numeric_Error | - Beim Lesen aus einer Nicht-Text-Datei mit Read(f,v) liegt der Wert der Puffervariablen f↑ vom Typ Long_Real nicht im Wertebereich des Typs Short Real der Variablen v. |
| Range_Error | - Beim Lesen aus einer Nicht-Text-Datei mit Read (f,v) liegt der Wert der Puffervariablen f↑ von einem Ordinal-Typ nicht im Wertebereich des Typs Ordinal-Typs der Variablen v. |
| Set_Error | - Beim Lesen aus einer Nicht-Text-Datei mit Read (f, v) liegt der Wert der Puffervariablen f↑ von einem Set-Typ nicht im Wertebereich des Set-Typs der Variablen v. |
| String_Error | - Beim Lesen aus einer Nicht-Text-Datei mit Read (f, v) ist die aktuelle Länge der Zeichenkette von einem String-Typ in der Puffervariablen f↑ größer als die Maximallänge des String-Typs der Variablen v.<br><br>- Beim Lesen aus einer Nicht-Text-Datei mit Read (f, v) ist die aktuelle Länge der Zeichenkette von einem String-Typ in der Puffervariablen f↑ ungleich der Länge des Zeichenkettentyps fester Länge der Variablen v. |

*Beispiel*

Die Komponenten der Datei f werden sequentiell gelesen und verarbeitet

```
VAR
   f : FILE OF komponenten_typ;
   v : komponenten_typ;
BEGIN
   Reset (f);
   WHILE NOT Eof (f) DO BEGIN
      Read (f, v);
      Verarbeite (v);
      END;
END;
```

*Querverweise*


allgemeine Dateien: 6.3.5
Zuweisungsverträglichkeit: 6.6.2
Variablenparameter: 8.5.2
Verallgemeinerte Variablen: 7

- **Read - von einer Textdatei**

  **Read (f, v1, ..., vn)**
  **Read (v1, ..., vn)**

  Besitzt die FILE-Variable f den vordefinierten Typ Text, dann kann jede Variable vi, (i = 1...n) einen der folgenden Typen besitzen:
  - den Typ Char oder einen Teilbereich von Char,
  - einen Integer-Typ oder einen Teilbereich eines Integer-Typs,
  - einen Real-Typ,
  - einen String-Typ,
  - einen Typ PACKED ARRAY [1..n] OF Char.

  Ist der Typ einer Variablen vi ein Integer-Typ oder ein Real-Typ, dann wird der durch die Zeichenkette repräsentierte Wert auf die Variable vi übertragen.

  Wird die FILE-Variable f nicht angegeben, so wird von der vordefinierten Textdatei Input gelesen.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| File_Error | - Vor dem Aufruf von Read (f, ...) wurde die Datei f nicht zum Lesen eröffnet.<br><br>- Vor dem Aufruf von Read (f, ...) ist die Datei f undefiniert. |
| Eof_Error | - Beim Aufruf von Read (f, ...) oder beim Überlesen führender Leerzeichen und Zeilenende-Komponenten wird Eof (f) True. |
| Numeric_Error | - Der Wert der eingelesenen Realzahl ist intern noch darstellbar, liegt aber außerhalb des Wertebereichs des Leseparameters v. |
| Read_Error | - Beim Lesen einer Integerzahl oder Realzahl aus einer Textdatei (mit Read (f, v)) gilt:<br>- die Zahl ist syntaktisch fehlerhaft oder<br>- der Wert der eingelesenen Zahl ist so groß, daß er intern nicht mehr darstellbar ist. Bei Integerzahlen liegt der Wert außerhalb des Bereichs Long_Minint .. Long_Maxint und bei Realzahlen liegt der Wert außerhalb des Bereichs -Long_Maxreal .. Long_Maxreal. |
| Range_Error | - Der Wert der eingelesenen Integerzahl ist intern noch darstellbar, liegt aber außerhalb des Wertebereichs des Leseparameters v. |
| String_Error | - Bei Read (f,v) ist die Maximallänge der String-Variablen v kleiner als die Länge der eingelesenen Zeichenkette. |

– **Lesen eines Zeichens**

**Read (f, v), v ist eine Variable des Typs Char**

Ist v eine Variable vom Typ Char oder eines Teilbereichs von Char, dann ist Read (f, v) definiert durch

```
BEGIN  v := f↑; Get (f)  END
```

Die Variable v enthält das Leerzeichen (Blank) ' ', wenn vor dem Aufruf
Read (f, v) Eoln (f) True ist.

– **Lesen eines Integer-Wertes**

**Read (f, v), v ist eine Variable eines Integer-Typs**

Ist v eine Variable eines Integer-Typs oder eines Teilbereichs davon, so wird eine Folge von Zeichen gelesen. Führende Leerzeichen und Zeilenenden werden überlesen. Unmittelbar vor der Zahl kann ein Vorzeichen stehen. Der Lesevorgang wird beendet, wenn ein gelesenes Zeichen nicht mehr Teil einer Integer-Zahl sein kann. Dieses Zeichen steht dann in f↑.

Die eingelesene Zeichenkette muß einer Integer-Zahl mit Vorzeichen gemäß folgender Syntax entsprechen:

```
Integer-Zahl  =  [Vorzeichen] vorzeichenlose_Integer-Zahl.
Vorzeichen    =  "+" | "-".
```

Die eingelesene Zeichenkette wird zu einem Integer-Wert konvertiert und auf die die Variable v übertragen. Der Integer-Wert muß zuweisungsverträglich zu dem Typ von v sein.

– **Lesen eines Real-Wertes**

**Read (f, v), v ist eine Variable eines Real-Typs**

Ist v eine Variable eines Real-Typs, dann wird eine Folge von Zeichen gelesen. Führende Leerzeichen und Zeilenenden werden überlesen. Unmittelbar vor der Zahl kann ein Vorzeichen stehen. Der Lesevorgang wird beendet, wenn ein gelesenes Zeichen nicht mehr Teil einer vorzeichenlosen Integer-Zahl oder Real-Zahl sein kann. Dieses Zeichen steht dann in f↑.

Die eingelesene Zeichenkette muß einer Zahl mit Vorzeichen gemäß folgender Syntax entsprechen:

```
Real-Zahl     =  [Vorzeichen] ( vorzeichenlose_Real-Zahl
                              | vorzeichenlose_Integer-Zahl ).
Vorzeichen    =  "+" | "-".
```

Die eingelesene Zeichenkette wird zu einem Real-Wert konvertiert und auf die Variable v übertragen. Der Real-Wert muß zuweisungsverträglich zu dem Typ von v sein.

– **Lesen in eine Variable eines String-Typs**

**Read (f, v), v ist eine Variable eines String-Typs**

Es werden alle Zeichen bis zum Ende der aktuellen Zeile gelesen und auf v übertragen. Nach Read (f, v) ist Eoln (f) True und f↑ enthält ein Leerzeichen ('␣').

Read (f, v) ist äquivalent zu:

```
BEGIN
   v := '';
   WHILE NOT Eoln (f) DO BEGIN
      v := Concat (v, f↑);
      GET (f);
      END;
END;
```

– **Lesen in eine Variable eines Zeichenkettentyps fester Länge**

**Read (f, v), v ist eine Variable eines Zeichenkettentyps der Länge n**

Das Einlesen von Zeichen wird beendet, wenn eine der beiden Bedingungen erfüllt ist:

1) Die Anzahl der eingelesenen Zeichen ist gleich der Länge n des Zeichenkettentyps von v.
2) Es wurde das Ende der Zeile erreicht, d. h. Eoln (f) ist True. Ist die Anzahl der gelesenen Zeichen kleiner als n, dann werden die restlichen Komponenten von v mit Leerzeichen (' ') aufgefüllt.

Read (f, v) mit v vom Typ PACKED ARRAY [1..n] OF Char ist äquivalent zu:

```
   FOR i := 1 TO n DO
      IF Eoln (f) THEN
         v[i] := ' '
      ELSE
         Read (f, v[i]);
```

*Beispiel*

Das Beispiel zeigt die möglichen Auswirkungen von Read mit unterschiedlichen Parametern. Die Datei f enthalte folgende Zeilen (Leerzeichen werden zur besseren Sichtbarkeit durch '_' dargestellt):

```
1. Zeile : #_Hallo
2. Zeile : A123____3.1415PI
3. Zeile : ____
4. Zeile : 0001_DM_Schluss

VAR
   f    : Text;
   i, j : Integer;
   r    : Real;
   c    : Char;
   s    : String;
   a    : PACKED ARRAY [1..3] OF Char;
BEGIN
   Reset (f);

   Read (f, c);       { c enthält '#' }

   Read (f, s);       { s enthält '_Hallo' }

   Read (f, s);       { s enthält '', da Eoln (f) = True }

   Readln;            { positionieren auf die nächste Zeile }

   Read (f, c);       { c enthält 'A' }

   Read (f, i);       { i enthält 123 }

   Read (f, r);       { r enthält 3.1415 }

   Read (f, a);       { a enthält 'PI_' }

   Read (f, j);       { Zeile 3 wird überlesen, j enthält 1 }

   Read (f, a);       { a enthält '_DM' }

   Read (f, s);       { s enthält '_Schluss' }
END
```

Die Anweisungsfolge ist äquivalent zu:

```
BEGIN
   Reset (f);
   Readln (f, c, s, s);
   Read (f, c, i, r, a, j, a, s);
END
```

**Readln (f, v1, ..., vn)**
**Readln (v1, ..., vn)**
**Readln (f)**
**Readln**

Die FILE-Variable f muß vom Typ Text sein. Wird f weggelassen, so wird Readln auf die vordefinierte Datei Input angewandt. Die zulässigen Parameter v1 bis vn sind bei Read für Textdateien beschrieben.

Durch den Aufruf von Readln wird auf den Anfang der nächsten Zeile in der Datei positioniert.

Readln (f, v1, ..., vn) ist äquivalent zu

```
BEGIN
   Read (f, v1); ...; Read (f, vn);
   Readln (f);
END
```

und Readln (f) ist äquivalent zu

```
BEGIN
   WHILE NOT Eoln (f) DO Get (f);
   Get (f);
END
```

Die restlichen Zeichen der aktuellen Eingabezeile, falls vorhanden, werden überlesen. Das erste Zeichen der neuen Zeile wird in die Puffervariable f↑ übertragen. Wurde das Dateiende erreicht, so ist nach Readln die Puffervariable undefiniert und Eof ist True.

**Mögliche Laufzeitfehler wie bei Read (s.o.).**

*Hinweis*

Die Besonderheiten von Readln im Zusammenhang mit der Eingabe von Dialogstationen sind in Kapitel 19 beschrieben.

## Reset (f)

Reset (f) eröffnet eine Datei f zum Lesen und liest die erste Komponente in die Puffervariable f↑. Ist die Datei leer, dann liefert Eof den Wert True und die Puffervariable f↑ ist undefiniert.

### Implementierungsdefinierte Eigenschaft

Die Wirkung der vordefinierten Prozedur Reset auf eine der vordefinierten Textdateien Input oder Output ist nach Norm-Pascal implementierungsdefiniert.
In Pascal-XT ist diese Eigenschaft für alle Implementierungen gleich definiert: es tritt ein Open_Error auf.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Open_Error | - Bei Aufruf von Reset wurde die vordefinierte Textdatei Input bzw. Output angegeben.<br>- Beim Aufruf von Reset (f) ist die Datei f undefiniert (siehe 7.3).<br>- Bei Reset (f) kann die f zugeordnete, außerhalb des Programms liegende Datei nicht zum Lesen eröffnet werden. |

## Rewrite (f)

Rewrite eröffnet die Datei f zum Schreiben. Nach Rewrite (f) ist die Datei leer und kann neu beschrieben werden. Es ist Eof (f) True. Der Inhalt von f↑ ist undefiniert. Ein etwaiger alter Dateiinhalt ist verloren.

### Implementierungsdefinierte Eigenschaft

Die Wirkung der vordefinierten Prozedur Rewrite auf eine der vordefinierten Textdateien Input oder Output ist nach Norm-Pascal implementierungsdefiniert.
In Pascal-XT ist diese Eigenschaft für alle Implementierungen gleich definiert: es tritt ein Open Error auf.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Open_Error | - Bei Aufruf von Rewrite wurde die vordefinierte Textdatei Input bzw. Output angegeben.<br><br>- Bei Rewrite (f) kann die f zugeordnete, außerhalb des Programms liegende Datei nicht zum Schreiben eröffnet werden. |

**Write (f, a1, ..., an)**

Die Prozedur Write dient zum Schreiben der Werte a1 bis an in die Datei f.

Die Datei-Variable f kann von einem beliebigen FILE-Typ sein, außer dem vordefinierten FILE-Type Any_File. Die Datei f muß zum Schreiben eröffnet worden sein.

Der Aufruf

```
Write (f, a1, ..., an)
```

```
ist definiert durch:
```

```
BEGIN Write (f, a1); ...; Write (f, an); END
```

- **Write - auf eine Nicht-Textdatei**

  **Write (f, a1, ..., an)**

  Ist die FILE-Variable f nicht von dem vordefinierten FILE-Typ Text, dann ist

  Write (f, a)

  definiert durch:

```
BEGIN    f↑ := a; Put (f) END
```

  Der Ausdruck a muß zuweisungsverträglich zu der Puffervariablen f↑ sein.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Die Datei f wurde vor dem Aufruf von Write nicht zum Schreiben eröffnet.<br><br>- Vor dem Aufruf von Write (f) ist die Datei undefiniert. |
| Numeric_Error | - Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) liegt der Wert des Ausdrucks a vom Typ Long_Real nicht im Wertebereich der Puffervariablen f↑ vom Typ Short_Real. |
| Range_Error | - Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) liegt der Wert des Ausdrucks a von einem Ordinal-Typ nicht im Wertebereich des Ordinal-Typs der Puffervariablen f↑. |
| Set_Error | - Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) liegt der Wert des Ausdrucks a von einem Set-Typ nicht im Wertebereich des Set-Typs der Puffervariablen f↑. |
| String_Error | - Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) ist die aktuelle Länge der Zeichenkette größer als die Maximallänge des String-Typs der Puffervariablen f↑.<br><br>- Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) ist die aktuelle Länge der Zeichenkette a von einem String-Typ ungleich der Länge der Puffervariablen f↑ von einem Zeichenkettentyp fester Länge. |

*Beispiel*

Die berechneten Daten werden sequentiell in die Datei f geschrieben.

```
VAR
   f : FILE OF komponenten_typ;
   v : komponenten_typ;
BEGIN
   Rewrite (f);
   WHILE NOT fertig DO BEGIN
      v := neuer_wert;
      Write (f, v);
      END;
END;
```

- **Write - auf eine Textdatei**

**Write (f, a1, ..., an)**
**Write (a1, ..., an)**

Ist die FILE-Variable f vom vordefinierten Typ Text, dann ist

```
Write (f, a1, ..., an)
```

`definiert durch`

```
BEGIN Write (f, a1); ..; Write (f, an) END
```

und jeder Parameter ai kann eines der drei Formate haben:

```
a
a : n
a : n : m.
```

Wird die FILE-Variable f nicht angegeben, so wird die vordefinierte Textdatei Output angenommen.

Der Wert des Ausdrucks a, der in die Datei f geschrieben wird, kann einen der folgenden Typen besitzen:

- den Typ Char oder einen Teilbereich von Char,
- den Typ Boolean,
- einen Integer-Typ oder einen Teilbereich eines Integer-Typs,
- einen Real-Typ oder
- einen Zeichenkettentyp fester oder variabler Länge.

Bei Integer- und Real-Typen wird der Wert der Zahl vor dem Schreiben zu einer Zeichenkette konvertiert.

Die Angaben n und m dienen der Formatierung der ausgegebenen Werte. Dabei steht n für die Gesamtausgabelänge und m für die Anzahl der Ziffern nach dem Dezimalpunkt. n und m sind Ausdrücke von einem Integer-Typ, deren Werte größer oder gleich 1 sein müssen.
Ist der Typ von a ein String-Typ, dann darf der Wert von n auch 0 sein.

Das Format a ist äquivalent zum Format a:n, wobei je nach Typ von a ein vorgegebener Wert für n verwendet wird. Für Parameter vom einem Integer-Typ, einem Real-Typ und vom Typ Boolean ist dieser Wert von n implementierungsdefiniert.

Das Format a : n : m ist nur anwendbar, wenn der Ausdruck a von einem Real-Typ ist. Damit wird die Festpunktdarstellung gewählt und m gibt die Anzahl der Stellen nach dem Dezimalpunkt an (siehe weiter unten).

### Mögliche Laufzeitfehler:

| | |
|---|---|
| File_Error | - Die Datei f wurde vor dem Aufruf von Write nicht zum Schreiben eröffnet.<br><br>- Vor dem Aufruf von Write (f) ist die Datei undefiniert. |
| Range_Error | - Bei Write (f,a:n) ist die Gesamtausgabelänge n < 1 bzw. n < 0 wenn a von einem String-Typ ist.<br><br>- Bei Write(f,a:n:m) ist die Gesamtausgabelänge n < 1 oder die Anzahl der Ziffern nach dem Dezimalpunkt m < 1. |

– **Ausgeben eines Zeichens**

**Write (f, a) oder**
**Write (f, a : n), a ist ein Ausdruck des Typs Char**

Ist a ein Ausdruck vom Typ Char oder eines Teilbereichs von Char, dann ist

Write (f, a) definiert durch

```
BEGIN    f↑ := a; Put (f) END
```

Die Wirkung von Write (f, a) ist gleich der von Write (k, a:1). Durch den Aufruf Write (f, a:n) werden vor a zusätzlich (n-1) Leerzeichen ausgegeben.

*Beispiel*

Das Beispiel soll eine Pyramide aus Sternen zeilenweise ausgeben, wobei die Stellung des ersten Sterns pro Zeile mittels Längenparameter bestimmt wird.

```
PROGRAM pyramide d (Output);
CONST
   grenze = 5;
   stern  = '*';
VAR
   i : 1..grenze;
   k : 0..8;
BEGIN
   FOR i := 1 TO grenze DO BEGIN
      Write (stern : (grenze - i + 1));
         { schreibt einen Stern mit führenden Blanks}
      FOR k := 1 TO 2 * (i - 1) DO
         Write (stern);
      Writeln;
      END
END.
```

Ausgabe des Programms:

```
1. Zeile:          *
2. Zeile:         ***
3. Zeile:        *****
4. Zeile:       *******
5. Zeile:      *********
```

– **Ausgeben von Zeichenketten**

**Write (f, a) oder Write (f, a : n), a ist im Ausdruck eines Zeichenkettentyps**

Ist a von einem Zeichenkettentyp fester Länge mit k Komponenten, dann ist k der Standardwert für die Gesamtausgabelänge.

Ist der Ausdruck a von einem String-Typ, dann ist die aktuelle Länge von a (Length (a)) der Standardwert für die Gesamtausgabelänge.

Im Fall einer Längenangabe erfolgt die Ausgabe rechtsbündig mit führenden Leerzeichen, falls die aktuelle Länge der auszugebenden Zeichenkette kleiner ist als die Gesamtausgabelänge n. Falls die Zeichenkette hingegen länger ist, so werden nur ihre ersten n Elemente ausgegeben.

Die Längenangabe kann bei String-Werten auch Null sein. Es erfolgt dann keine Ausgabe.

– **Ausgeben von Integer-Werten**

**Write (f, a) oder**
**Write (f, a : n), a ist ein Ausdruck eines Integer-Typs**

Der Wert des Ausdrucks a wird in Dezimaldarstellung in die Datei f geschrieben:

- Eine Folge von Leerzeichen (Anpassen der Ausgabelänge),
- ein Minuszeichen ('-'), falls der Ausdruck negativ ist,
- und der Wert des Ausdrucks (Dezimalzahl).

Die Anzahl der auszugebenden Zeichen wird durch den Ausdruck n bestimmt, falls die Form Write (f, a:n) gewählt wurde. Ansonsten wird eine implementierungsdefinierte Anzahl von Zeichen ausgegeben (siehe Benutzerhandbuch).

Wird mit einer Anweisung Write (f, a:n) ein Ausgabefeld definiert, dessen Länge nicht ausreichend ist, den Wert des Ausdrucks in dezimaler Notation darzustellen, so wird n durch die benötigte Länge ersetzt. Ist n größer, als zur Ausgabe der Zahl notwendig ist, so wird bis zur angegebenen Länge mit führenden Leerzeichen aufgefüllt.

**Implementierungsdefinierte Eigenschaft**

Die Standard-Ausgabelänge für Werte eines Integer-Typs ist implementierungsdefiniert.

*Beispiel*

Leerzeichen werden zur besseren Sichtbarkeit durch '_' dargestellt.

```
Write-Anweisung          Ausgabe                  tatsächliche
                                                  Ausgabelänge

Write ( 1325:8)          ____1325                     8
Write (-1235:8)          ___-1235                     8
Write ( 9876:2)          9876                         4
Write (-9876:2)          -9876                        5
```

- **Ausgeben von Real-Werten in Gleitpunktdarstellung**

  **Write (f, a) oder**
  **Write (f, a:n), a ist ein Ausdruck eines Real-Typs**

  Der Wert des Ausdrucks a wird in dezimaler Gleitpunktdarstellung mit Exponentenangabe in die Datei f geschrieben und zwar gerundet auf die gewünschte Anzahl signifikanter Stellen:

  - Eine Folge von Leerzeichen (Anpassen der Ausgabelänge),
  - ein Minuszeichen ('-'), falls der Ausdruck negativ ist,
  - und der Wert der Mantisse (Vor- und Nachkommastellen),
  - der Exponent

  Wird die Gesamtausgabelänge n nicht angegeben, so wird eine implementierungsdefinierte Ausgabelänge angenommen (siehe Benutzerhandbuch). Ist n für eine Ausgabe mit maximaler Genauigkeit zu klein, so werden Dezimalstellen weggerundet. Ist n auch für eine Ausgabe mit nur einer Nachkommastelle zu klein, so wird n durch die dafür benötigte Länge ersetzt. Ist n größer als nötig, werden an die Dezimalstellen Nullen angehängt. In jedem Falle erscheint vor der führenden Ziffer entweder ein Minuszeichen oder ein Leerzeichen.

**Implementierungsdefinierte Eigenschaft**

- Die Standard-Ausgabelänge für Werte eines Real-Typs ist implementierungsdefiniert.
- Die Darstellung des Exponentenzeichens (als ’E’ oder ’e’) und die Anzahl der Dezimalziffern für den Exponenten bei Ausgabe von Real-Werten in Gleitpunktdarstellung sind implementierungsdefiniert.

*Beispiel*

Leerzeichen werden zur besseren Sichtbarkeit durch ’_’ dargestellt.

```
Write-Anweisung                     impl. def.          tatsächliche
                                    Ausgabe             Ausgabelänge

Write (8.12231E17:12)               _8.12231E+17           12
Write (-10.052173:14)               -1.0052173E+01         14
Write (5.0123E-6:11)                _5.0123E-6             11
Write (0.5:13)                      _5.000000E-01          13
Write (12.3:9)                      _1.23E+01               9
Write (2.3128143561231E11:20)       _2.3128143561231E+11   20
Write (3.14:1)                      _3.1E+00                8
```

- **Ausgeben von Real-Werten in Festpunktdarstellung**

**Write (f, a : n : m), a ist ein Ausdruck eines Real-Typs**

Der Wert des Ausdrucks a wird in dezimaler Festpunktdarstellung mit m Nachkommastellen folgendermaßen in die Datei f geschrieben:

- Eine Folge von Leerzeichen (Anpassen der Ausgabelänge),
- ein Minuszeichen (’-’), falls der Ausdruck negativ ist, andernfalls ein Leerzeichen (’ ’),
- und der Wert des Ausdrucks (Vor- und Nachkommastellen)

n ist die Gesamtausgabelänge. Wird n zu klein angegeben, so wird der kleinste mögliche Wert genommen. Die minimale Gesamtausgabelänge ergibt sich aus der Anzahl der Vor- und Nachkommastellen, einer Stelle für den Dezimalpunkt sowie einer Stelle für das Vorzeichen (’ ’ oder ’-’).

m bestimmt die Anzahl der auszugebenden Nachkommastellen. Ist m größer, als zur Ausgabe mit maximaler Genauigkeit notwendig ist, so werden an die Dezimalstellen Nullen angehängt. Ist m kleiner, so wird die kleinste auszugebende Stelle gerundet.

*Beispiel*

Leerzeichen werden zur besseren Sichtbarkeit durch ’_’ dargestellt.

```
Write-Anweisung                        Ausgabe             tatsächliche
                                                           Ausgabelänge

Write ( 12219.378:10:2)                __12219.38             10
Write ( 12219.378:10:3)                _12219.378             10
Write (-12219.378:10:3)                -12219.378             10
Write (-12219.378:6:1)                 -12219.4                8
```

- **Ausgeben von Booleschen-Werten**

  **Write (f, a) oder**
  **Write (f, a : n), a ist ein Ausdruck des Typs Boolean**

  Die Ausgabe entspricht folgenden Aufbau

  - Eine Folge von Leerzeichen (Auffüllen auf Ausgabelänge),
  - die Zeichenfolgen ’TRUE’ oder ’FALSE’

  Write (f, a : n) wird wie folgt behandelt:
  Die Ausgabe erfolgen so, als ob ein String der angegebenen Länge ausgegeben würde.

  - n > Ausgabelänge : führende Leerzeichen, Text rechtsbündig
  - n < Ausgabelänge : die ersten n Zeichen werden ausgegeben

**Implementierungsdefinierte Eigenschaft**

- Die Schreibweise (Groß-/Kleinschreibung) der einzelnen Buchstaben der Booleschen Werte True und False bei der Ausgabe ist für jeden Buchstaben implementierungsdefiniert.
- Die Standard-Ausgabelänge für Werte des Typs Boolean ist implementierungsdefiniert.

*Beispiel*

Leerzeichen werden zur besseren Sichtbarkeit durch ’_’ dargestellt.

```
Aufruf                 impl. def.
                       Ausgabe

Write (1=2:10)         _____FALSE
Write (1=1:3)          TRU
```

**Writeln (f, a1, ..., an)**
**Writeln (a1, ..., an)**
**Writeln (f)**
**Writeln**

Die FILE-Variable f muß vom Typ Text und zum Schreiben eröffnet sein. Wirdf weggelassen, so wird Writeln auf die vordefinierte Textdatei Output angewendet. Die zulässigen Parameter a1 bis an sind bei Write beschrieben.

Durch den Aufruf von Writeln wird eine Zeilenende-Komponente in die Datei f geschrieben und auf den Anfang einer neuen Zeile positioniert.

Writeln (f, p1, ..., pn) ist äquivalent zu

BEGIN Write (f, p1, ..., pn); Writeln (f) END

und Writeln (f) ist äquivalent zu

```
BEGIN
   f↑ := "Zeilenende-Komponente";
   Put(f);
END
```

**Mögliche Laufzeitfehler wie bei Write auf Textdateien.**

*Hinweis*

Bei gepufferter Ausgabe auf einer Datensichtstation wird erst durch Writeln eine Zeile abgeschlossen und erscheint auf dem Bildschirm.

# Unterprogramme zur Haldenverwaltung

### Dispose (p)

Der Parameter p muß ein Ausdruck eines vom generischen Zeigertyp verschiedenen Zeigertyps sein. Sein Wert muß auf eine (mit New erzeugte) dynamische Variable p↑ verweisen. Durch Dispose (p) wird der für die dynamische Variable p↑ benötigte Speicherplatz freigegeben.

Das Pascal-XT-System kann den nun nicht mehr benötigten Speicherplatz von p↑ bei späteren Anforderungen durch New wieder verwenden.

Die dynamische Variable p↑ ist danach nicht mehr zugreifbar, d.h. alle Verweiswerte, die auf sie zeigen, werden durch Dispose (p) ungültig.

Wurde die Form New (q, ...) mit mehreren Parametern gewählt (verkürztes Anlegen einer dynamischen Variable), so muß die gleiche Form auch bei Dispose gewählt werden.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Pointer_Error | - Beim Aufruf von Dispose (p) hat p den Wert NIL. |
| undefinierte Auswirkungen | - Mit Dispose (p) wird Speicherplatz einer dynamischen Variablen freigegeben, obwohl noch eine Referenz auf sie existiert.<br>- Beim Aufruf von Dispose (p) ist der Wert von p undefiniert.<br>- Vor dem Aufruf von Dispose (p) ist p↑ durch New (q, c1, ..., cn) oder New (q, c1, ..., cn, e) oder New (q, e) erzeugt worden.<br>- Vor dem Aufruf von Dispose (p, k1, ..., km) ist die dynamische Variable p↑ durch New (q, c1, ..., cn) erzeugt worden, wobei m ungleich n ist.<br>- Bei Dispose (p, c1, ..., cn) oder Dispose (p, c1, ..., cn, e) sind in der dynamischen Variablen p↑ andere Varianten eingestellt, als durch die Auswahlkonstanten c1 bis cn angegeben.<br>- Vor dem Aufruf von Dispose (p, e) ist p↑ durch New (q, a) erzeugt worden, wobei a ungleich e ist. Analog gilt dies auch für Dispose (p, c1, ..., cn, e) und New (q, k1, ..., kn, a)<br>- Bei Dispose (p, e) oder Dispose (p, c1, ..., cn, e) liegt der Wert von e nicht im Wertebereich des Indextyps des entsprechenden ARRAY-Typs bzw. ist kleiner 1 oder größer als die Maximallänge des entsprechenden String-Typs. |

## Dispose, bei verkürzten Speicherbereichen

### Dispose (p, e)

Die Wirkung ist analog zu Dispose (p).

Der Domänentyp von p muß ein ARRAY-Typ, ein String-Typ oder ein RECORD-Typ, dessen letztes Feld der Feldliste einen ARRAY-Typ oder einen String-Typ besitzt, sein.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen. Der Wert von e darf nicht kleiner als 1 und nicht größer als die Maximallänge des String-Typs sein.

Der Zeigerausdruck p muß auf eine dynamische Variable p↑ zeigen, die durch New (q, e) entstanden ist. Dabei muß jeweils der gleiche Wert e angegeben werden.

*Beispiel*

```
TYPE
   t : ARRAY [1..100] OF Real;
VAR
   p : ↑t;
BEGIN
   New (p, 45);
   ... { Verarbeitung }
   Dispose (p, 45);
END
```

### Dispose (p, c1, ..., cn)

Die Wirkung ist analog zu Dispose (p).

Der Domänentyp von p muß ein RECORD-Typ sein. Er hat ineinandergeschachtelte Varianten, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein. Varianten, die nicht aufgeführt sind, müssen in einer tieferen Verschachtelungsebene stehen als cn.

Die Zeigerausdruck p muß auf eine dynamische Variable p↑ zeigen, die durch New (q, c1, ..., cn) entstanden ist. Dabei müssen jeweils die gleichen Selektorkonstanten angegeben worden sein.

*Beispiel*

Gegeben ist folgende Datenstruktur t, die aus ineinander geschachtelten Varianten besteht, auf die in den folgenden Beispielen verwiesen wird.

```
TYPE
  char_range = '0'..'1';
  t = RECORD
         t1: Integer;
         CASE t2: Boolean OF
            False:
               (t21: Real;
                CASE t22: char_range OF
                   '0': (t221: ARRAY [0..10] OF Integer);
                   '1': (t222: Short_Integer));
            True: (t23: Char);
      END;
VAR
   p, q : ↑t;
BEGIN
   New (p, False);
   New (q, False, '0');
   ...  {Verarbeitung}
   Dispose (q, False, '0');
   Dispose (p, False);
END;
```

### Dispose (p, c1, ..., cn, e)

Die Wirkung ist analog zu Dispose (q).

Der Domänentyp des FILE-Typs von f muß ein RECORD-Typ mit Varianten sein, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein.

Die Feldliste der zur letzten Selektorkonstanten cn gehörenden Variante darf keine weiteren eingeschachtelten Varianten besitzen. Das letzte Feld der Feldliste der innersten Variante muß einen ARRAY- oder String-Typ besitzen.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen. Der Wert von e darf nicht kleiner als 1 und nicht größer als die Maximallänge des String-Typs sein.

Die Zeigerausdruck p muß auf eine dynamische Variable p↑ zeigen, die durch New (q, c1, ..., cn, e) entstanden ist.

*Beispiel*

Dieses Beispiel bezieht sich auf Definitionen des vorhergehenden Beispiels.

```
New (p, False, '0', 2);
```

```
...
Dispose (p, False, '0', 2);
```

### Mark (p)

Mit Mark wird der aktuelle Haldenpegel in der Variablen p vom generischen Zeigertyp Pointer gespeichert. Die Halde ist jetzt durch den Zeiger p an ihrer aktuellen Position markiert. Weitere mit New erzeugte Objekte werden jenseits dieser Haldenmarkierung angelegt. Mit Release (p) kann der Haldenpegel wieder auf diesen Wert zurückgesetzt werden. Dadurch wird der Speicherplatz aller seit dem zugehörigen Aufruf von Mark (p) mit New erzeugten dynamischen Variablen freigegeben.

*Beispiel*

```
VAR
   Anfangspegel : Pointer;
BEGIN
   Mark (Anfangspegel);
   WHILE bedingung DO BEGIN
      ...
      New (...);
      ...
      END;
   Release (Anfangspegel);
END;
```

**New (p)**

Der Parameter p muß eine verallgemeinerte Variable eines vom generischen (damit auch privaten) Zeigertyps verschiedener Zeigertyp sein.

Mit New (p) wird auf der Halde Platz für eine neue dynamische Variable vom Domänentyp von p angefordert. Nach New (p) zeigt p auf diese dynamische Variable p↑, deren Wert noch undefiniert ist.

Obwohl der Parameter p eine verallgemeinerte Variable sein muß, ist er kein aktueller Variablenparameter. Insbesondere darf p auch eine Komponente eines gepackten RECORDs oder ARRAYs sein.

Die Prozedur New bietet außerdem die Möglichkeit die Speicheranforderung zu minimieren. Bei RECORD-Typen mit Varianten kann die benötigte Variantenkombination angegeben werden, so daß nur Platz für diese bereitgestellt wird. Bei ARRAY- oder String-Typen oder RECORD-Typen, die als letztes Komponente einen ARRAY- oder String-Typ enthalten, kann der maximale Indexwert angegeben werden, für den Speicherplatz bereitgestellt werden soll.

**Mögliche Laufzeitfehler:**

In dieser Tabelle sind alle möglichen Fehler, die im Zusammenhang mit New (p) oder mit New mit verkürzter Speicheranforderung auftreten können, zusammengefaßt.

| | |
|---|---|
| Memory_Error | - Der für die zu erzeugende dynamische Variable benötigte Speicherplatz ist nicht verfügbar. |
| undefinierte Auswirkungen | - Einer dynamischen String-Variablen, die mit New (p, e) erzeugt wurde oder an die letzte Komponente einer dynamischen String-Variablen, die mit New (p, c1, ..., cn, e) erzeugt wurde, wird eine Zeichenkette zugewiesen, die länger ist als e.<br>- In einer mit New (p, c1, ..., cn) erzeugten dynamischen Variable wird eine andere Variante eingestellt, als durch die Selektorkonstanten c1 bis cn angegeben wurde.<br>- Eine dynamische Variable, die durch New (p, c1, ..., cn), New (p, e) oder New (p, c1, ..., cn, e) erzeugt wurde, kommt als Ganzes in einem Ausdruck oder als linke Seite in einer Wertzuweisung vor, oder wird als Parameter übergeben.<br>- Bei New (p, e) oder New (p, c1, ..., cn, e) liegt der Wert von e nicht im Wertebereich des Indextyps des entsprechenden ARRAY-Typs bzw. ist kleiner 1 oder größer als die Maximallänge des entsprechenden String-Typs. |

**New (p, e)**

Die Wirkung ist analog zu New (p). Der Speicherbereich kann verkürzt angefordert werden.

Der Domänentyp von p muß ein ARRAY-Typ, ein String-Typ oder ein RECORD-Typ, dessen letztes Feld der Feldliste einen ARRAY-Typ oder einen String-Typ besitzt, sein.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen und sein Wert darf nicht kleiner als 1 oder größer als die Maximallänge des String-Typs sein.

Es wird nur soviel Speicherplatz angelegt, wie bei einem ARRAY [m..n] für die Komponenten m bis e und bei einer Zeichenkette für die Komponenten 1 bis e benötigt wird.

### New (p, c1, ..., cn)

Die Wirkung ist analog zu New (p). Der Speicherbereich kann verkürzt angefordert werden.

Der Domänentyp von p muß ein RECORD-Typ sein. Er muß ineinandergeschachtelte Varianten enthalten, zu denen die Auswahlkonstanten c1, ..., cn gehören. Diese Auswahlkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein. Varianten, die nicht aufgeführt sind, müssen in einer tieferen Verschachtelungsebene stehen als cn.

Für die so definierte Variantenkombination wird eine dynamische Variable mit genau dieser Platzanforderung erzeugt.

Der Wert von p↑ ist undefiniert, insbesondere sind entsprechende Kennungsfelder noch nicht initialisiert. Es ist damit noch keine Variante eingeschaltet.

Die Variantenkombination einer solcherart erzeugten Variablen darf nicht durch Zuweisen abweichender Werte auf die Selektorfelder geändert werden.

*Beispiel*

```
TYPE
   r = RECORD
          ri : Integer;
          CASE rb : Boolean OF
             True  : (rta : ARRAY[1..10] OF Char);
             False : (rfr : Real;
                     CASE rfc : Char OF
                        'A': (rfai : Integer);
                        'B': (rfbr : Real);
                        ELSE: ();
                     );
       END;

VAR
   p : ↑r;

BEGIN
   New (p, True);
   { erzeugt ein Objekt mit den Feldern ri, rb, rta }
   New (p, False, 'A');
   { erzeugt ein Objekt mit den Feldern ri, rb, rfr,
     rfc, rfai }
END;
```

**New (p, c1, ..., cn, e)**

F=TMP
fordert werden.

Der Domänentyp von p muß ein RECORD-Typ mit Varianten sein, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein.

Die Feldliste der zur letzten Selektorenkonstanten cn gehörigen Variante darf keine weiteren eingeschachtelten Varianten enthalten.

Das letzte Feld der Feldliste der innersten Variante muß einen ARRAY- oder String-Typ besitzen.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen und sein Wert drf nicht kleiner als 1 oder größer als die Maximallänge des String-Typs sein.

*Beispiel*

```
TYPE
   r = RECORD
          ri : Integer;
          CASE rb : Boolean OF
             TRUE  : (rta : ARRAY[9..17,1..10] OF Char);
             False : (rfr : Real;
                        CASE rc : Char of
                           'A': (rfa : Integer);
                           'B': (rfb : String [26]);
                           ELSE: ();
                        );
        END;

VAR
   p : ↑r;

BEGIN

  New (p, True, 11);
    { erzeugt ein Objekt mit den Feldern ri, rb,
      rta [9..11, 1..10] }

  New (p, False, 'B', 20);
    { erzeugt ein Objekt mit den Feldern ri, rb,
      rfr, rc, rfb [1..20]  }
END.
```

*Hinweis für New mit verkürzter Speicheranforderung*

Dynamische Variable, die durch New (p, e), New (p, c1, ..., cn) bzw. New (p, c1, ..., cn, e) entstanden sind, dürfen nicht als Ganzes in Ausdrücken oder als linke Seite in einer Wertzuweisung vorkommen und nicht als Parameter übergeben werden. Diese Einschränkung muß vorgenommen werden, weil in den genannten Fällen die Variable als Ganzes beschrieben bzw. gelesen wird. Bei verkürzt angelegten Variablen hat dies zur Folge, daß nachfolgende Speicherplätze überschrieben werden bzw. daß auf nicht zugeteilte Speicherplätze versucht wird, zuzugreifen.

### Release (p)

Der Parameter p muß ein Ausdruck vom generischen Zeigertyp Pointer sein.

Der Wert von p muß durch einen Aufruf von Mark entstanden sein.

Die Halde wird auf den in p angegebenen Pegel zurückgesetzt. Der Speicherplatz aller seit dem zugehörigen Aufruf von Mark (p) mit New erzeugten dynamischen Variablen wird freigegeben. Ebenso werden alle seit diesem Aufruf von Mark (p) mit Mark (q) erzeugten Verweiswerte vernichtet und können in keinen weiteren Aufrufen von Release (q) verwendet werden.

Mit Release können, im Gegensatz zu Dispose, mehrere Objekte auf der Halde freigegeben werden.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| Pointer_Error | - Beim Aufruf von Release (p) wurde der Verweiswert von p nicht durch einen Aufruf von Mark erzeugt. |
| undefinierte Auswirkung | - Mit Release (p) wird der Speicherplatz einer dynamischen Variablen freigegeben, obwohl noch eine Referenz auf sie besteht.<br>- Der beim Aufruf von Release (p) übergebene Verweiswert p wurde durch einen anderen Aufruf von Release (q) vernichtet. |

*Beispiel*

Die beiden Anweisungsfolgen

```
Mark (p);
New  (p1);
New  (p2);
New  (p3);
bearbeite (p1, p2, p3);
Release (p);
```

und

```
New  (p1);
New  (p2);
New  (p3);
bearbeite (p1, p2, p3);
Dispose (p1);
Dispose (p2);
Dispose (p3);
```

sind äquivalent.

# Unterprogramme zur Zeichenkettenverarbeitung

### Concat (s1, s2, ..., sn)

Mit der Funktion Concat kann eine beliebige Anzahl von Zeichenketten zu einer einzigen Zeichenkette zusammengefaßt werden. Die Zeichenketten si werden in der angegebenen Reihenfolge von links nach rechts fortschreibend abgelegt.

s1, ..., sn müssen Zeichenketten-Ausdrücke sein.

```
Concat (s1, s2, ..., sn)
```

`ist für n >= 2 äquivalent zu`

```
Concat (s1, Concat (s2, ..., sn)).
```

Concat (s1) ergibt den Wert s1 selbst und besitzt einen String-Typ, dessen Maximallänge die Länge der Zeichenkette s1 ist, unabhängig vom Typ von si.

Concat (s1, s2) liefert eine Zeichenkette der Länge Length (s1) + Length (s2).

Sind s1, ..., sn statische Ausdrücke, so ist auch ein Aufruf Concat (s1, ..., sn) ein statischer Ausdruck.

*Hinweis*

Die Summe Maxlength (s1) +...+ Maxlength (sn) muß kleiner oder gleich der maximal zulässigen Stringlänge ($2^{15}$-1) sein.

*Beispiel*

```
CONST
   s1 = 'das ist';
VAR
   s2 : String;
BEGIN
   s2 := 'stringbeispiel';
   s2 := Concat (s1, ' ein ', s2);
   { s2 = 'das ist ein stringbeispiel' }
END.
```

### Delete (s, i, n)

Mit der Prozedur Delete werden in der verallgemeinerten String-Variablen s ab Position i n Zeichen gelöscht und die restlichen Zeichen nach links verschoben. s muß von einem String-Typ sein. i und n müssen Ausdrücke eines Integertyps sein, für die gilt $i \geq 1$, $n \geq 0$, $i+n-1 \leq$ length (s)

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| String_Error | - Bei Delete(s,i,n) ist i<1 oder n<0 oder (i+n-1) > Length(s). |

*Beispiel*

```
VAR
   a : String;
BEGIN
   a := 'Dies ist eine String-Eliminierung';
   Delete(a,15,7);
   { jetzt hat a den wert 'Dies ist eine Eliminierung' }
END.
```

### Insert (s1, s2, i)

Mit der Prozedur Insert wird die Zeichenkette s1 in der verallgemeinerten String-Variablen s2 ab Position i eingefügt.

s1 muß ein Zeichenketten-Ausdruck sein. s2 muß eine Variable von einem String-Typ sein. i muß ein Ausdruck sein, dessen Wert zuweisungsverträglich zu Short_Integer ist.

Ist i größer als die aktuelle Länge von s2, dann werden s2 und s1 konkateniert.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| String_Error | - Bei Insert (s1, s2, i) ist i<1 oder Length (s2) + Length (s1) > Maxlength (s2) |

```
Insert (s1, s2, i) ist für i ≤ length(s2) äquivalent zu

   s2 := Concat (Substring (s2, 1, i-1),
                 s1,
                 Substring (s2, i, Length (s2)-i+1) )

und für i > length (s2) äquivalent zu

   s2 := Concat (s2, s1)
```

*Beispiel*

```
PROGRAM beispiel;
VAR
   b : String [10];
   c : String [20];

BEGIN
   b := ' ist ok';
   Insert ('Der String', b, 1);  { erzeugt String_Error }

   c := b;
   Insert ('Der String', c, 1);  { korrekter Aufruf }
   { in c steht jetzt 'Der String ist ok' }

   Insert ('ay', c, 100);        { korrekter Aufruf }
    { in c steht jetzt 'Der String ist okay' }
 END.
```

## Length (s)

Der Funktionsaufruf Length (s) liefert die Länge des Zeichenketten-Ausdrucks s als einen Wert vom Typ Short_Integer.

Ist s von einem String-Typ, so liefert Length(s) die aktuelle Länge des Wertes von s.

Ist s von einem Zeichenkettentyp fester Länge, so liefert Length (s) die Anzahl der Werte im Indextyp von s.

Ist s vom Typ Char, so liefert Length (s) den Wert 1.

Ist s ein statischer Ausdruck oder der Typ von s ein ARRAY-Typ (Zeichenkettentyp fester Länge) oder der Typ Char, so ist der Aufruf von Length (s) ebenfalls ein statischer Ausdruck.

*Beispiel*

```
PROGRAM beispiel;
CONST
   buchstaben ='abc';

VAR
   ispacked : PACKED ARRAY [1..5] OF Char;
   string10 : String [10];
   laenge   : Integer;
BEGIN
   string10 := ';';
   laenge := length (string10);    { laenge = 1 }

   laenge := length (buchstaben);  { laenge = 3 }

   laenge := length (ispacked);    { laenge = 5 }

   laenge := length ('');          { laenge = 0 }
END.
```

### Position (s1, s2)

Mit der Funktion Position kann man feststellen, ob und wo die Zeichenkette s1 in der Zeichenkette s2 enthalten ist. s1 und s2 müssen Zeichenketten-Ausdrücke sein.

Der Aufruf Position (s1, s2) liefert den Wert 0, wenn die Zeichenkette s1 nicht als Teilzeichenkette in der Zeichenkette s2 vorkommt bzw. falls s1 den Typ Char besitzt und das Zeichen s1 nicht in der Zeichenkette s2 Zeichenkette s2 vorkommt. vorkommt. Andernfalls liefert Position (s1, s2) den kleinsten positiven Integer-Wert i, so daß

```
s1 = Substring (s2, i, Length (s1))
```

gilt.

Position liefert den Wert 1, wenn s1 der Leerstring ist.

s1 und s2 müssen Zeichenkettenausdrücke sein. Der Ergebnistyp von Position ist Short_Integer.

*Beispiel*

```
PROGRAM beispiel;
VAR
   a   : String;

   pos : Integer;

BEGIN
   a   := 'affe';

   pos := position ('affe', a);         { pos = 1 }

   pos := position ('f', a);            { pos = 2 }

   pos := position ('', a);             { pos = 1 }

   pos := position (a, 'menschenaffe'); { pos = 9 }

   pos := position (a, 'x');            { pos = 0 }
END.
```

### Readstring (s, v1, ..., vn)

Die Prozedur Readstring liest Werte aus dem Zeichenketten-Ausdruck s in die verallgemeinerten Variablen v1, ..., vn. Die Prozedur wirkt analog wie die Prozedur Read für Textdateien, nur daß anstelle der FILE-Variablen ein Zeichenketten-Ausdruck stehen muß.

s muß ein Zeichenkettenausdruck sein, und die verallgemeinerten Variablen v1, ..., vn müssen einen der folgenden Typen besitzen:

- den Typ Char oder einen Teilbereich davon,
- einen Integer-Typ oder einen Teilbereich davon,
- einen Real-Typ,
- einen String-Typ,
- einen Zeichenkettentyp fester Länge.

Der Aufruf

```
Readstring (s, v1, ..., vn)
```

ist dann äquivalent zu der Verbundanweisung

```
BEGIN
   Rewrite (f); Writeln (f, s);
   Reset (f);   Read (f, v1, ..., vn)
EXCEPTION
   IF Error_Number = Eof_Error THEN
      Raise (String_Error)
   ELSE
      Raise (0);
END
```

wobei f eine sonst nirgends im Programm verwendete FILE-Variable vom Text-Typ ist. Dabei unterliegen v1, ..., vn den gleichen Konventionen wie die Parameter der Prozedur Read (siehe 15.1). Die äquivalente Verbundanweisung ist nicht die Realisierung, sondern dient der Erklärung.

*Hinweis*

Ist vi eine Variable eines String-Typs, dann werden von vi alle restlichen Zeichen des Zeichenkettenausdrucks s konsumiert (sofern die Maximallänge von s ausreicht, ansonsten tritt ein String_Error auf). Folgen auf vi weitere Parameter, dann können diesen keine weiteren Werte zugewiesen werden und es wird ein String_Error gemeldet.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Numeric_Error | - Der Wert der eingelesenen Realzahl ist intern noch darstellbar, liegt aber außerhalb des Wertebereichs des Leseparameters v. |
| Read_Error | - Beim Lesen einer Integerzahl oder Realzahl aus einem Zeichenkettenausdruck gilt:<br>- die Zahl ist syntaktisch fehlerhaft oder<br>- der Wert der eingelesenen Zahl ist so groß, daß er intern nicht mehr darstellbar ist. Bei Integerzahlen liegt der Wert außerhalb des Bereichs Long_Minint .. Long_Maxint und bei Realzahlen liegt der Wert außerhalb des Bereichs -Long_Maxreal .. Long_Maxreal. |
| Range_Error | - Der Wert der eingelesenen Integerzahl ist intern noch darstellbar, liegt aber außerhalb des Wertebereichs des Leseparameters v. |
| String_Error | - Bei Readstring (s, v) ist die Maximallänge der verallgemeinerten *String*-Variablen v kleiner als die Länge der eingelesenen Zeichenkette.<br><br>- Bei Readstring (s, v1, ..., vn) enthält der Zeichenkettenausdruck s nicht soviele Zeichen, wie durch die Leseparameter v1, ..., vn angefordert werden. |

*Querverweise*

Read: 15.1

## Substring (s, i, n)

Der Funktionsaufruf Substring (s, i, n) liefert eine Zeichenkette der Länge n, deren k-tes Zeichenkettenelement das (i+k-1)-te Zeichenkettenelement von s ist, also die Teilzeichenkette von s der Länge n, die bei i beginnt.
s muß ein Zeichenketten-Ausdruck sein. i und n müssen Ausdrücke eines Integertyps sein, für die gilt $i \geq 1$, $n \geq 0$, $i+n-1 \geq$ Length (s). Der Ergebnistyp von Substring ist ein String-Typ (variabel langer Zeichenketten-Typ).

Sind s, i, n statische Ausdrücke, so ist auch ein Aufruf von Substring (s, i, n) ein statischer Ausdruck.

### Mögliche Laufzeitfehler:

| | |
|---|---|
| String_Error | - Bei Substring(s,i,n) ist i<1 oder n<0 oder (i+n-1) > Length(s). |

*Beispiel*

```
VAR
   a : String;
BEGIN
   a := 'Verarbeitungspaket';
   a := Substring (a, 4, 6);
   { a hat jetzt den Wert 'arbeit' }
END.
```

## Writestring (s, a1, ..., an)

Die Prozedur Writestring schreibt die Werte der Ausdrücke a1, ..., an in die String-Variable s. Damit wirkt Writestring analog wie die Prozedur Write für Textdateien, nur daß anstelle der FILE-Variablen eine verallgemeinerte String-Variable s angegeben wird. s muß eine Variable von einem String-Typ sein und jeder Ausdruck ai (i=1, ..., n) muß einen der folgenden Typen be-
besitzen:

- den Typ Char oder einen Teilbereich davon,
- einen Integer-Typ oder einen Teilbereich davon,
- einen Real-Typ,
- einen Zeichenkettentyp,
- den Typ Boolean.

Für jeden Ausdruck ai kann eine Formatangabe wie bei den Schreibparametern für Write (siehe 15.1) angegeben werden.

Der Aufruf

```
Writestring (s, a1, ..., an)
```

ist dann äquivalent zu der Verbundanweisung

```
BEGIN
   Rewrite (f); Writeln (f, a1, ..., an);
   Reset (f);   Read (f, s);
END
```

wobei f eine sonst nirgends im Programm vorkommende FILE-Variable vom Text-Typ ist. Dabei unterliegen a1, ..., an den gleichen Konventionen wie die Parameter der Prozedur Write (siehe 15.1). Die äquivalente Verbundanweisung ist nicht die Realisierung, sondern dient der Erklärung.

### Mögliche Laufzeitfehler:

| Fehler | Beschreibung |
|---|---|
| String_Error | - Bei Writestring (s, a1, ..., an) ist die Maximallänge der String-Variablen s kleiner als die aus den Schreibparametern a1, ..., an gebildete Zeichenkette. |
| Range_Error | Bei Writestring (s, a:n) ist die Gesamtausgabelänge n < 1 bzw. n < 0, falls a von einem String-Typ ist. |
| | - Bei Writestring (s, a:n:m) ist die Gesamtausgabelänge n < 1 oder die Anzahl der Ziffern nach dem Dezimalpunkt m < 1. |
| undefinierte Auswirkungen | - Bei Writestring (s, p1,..., pn) enthält einer der Schreibparameter p1,...,pn eine Referenz auf die Stringvariable s. |

*Querverweise*

Write: 15.1

# Arithmetische Funktionen

Die arithmetischen Funktionen dienen numerischen Berechnungen. Der Typ des Parameters kann ein Real-Typ oder ein Integer-Typ sein. Der Ergebnistyp richtet sich nach dem Typ des Parameters.

Für die Funktionen Abs und Sqr gilt:

| Typ des Parameters | Typ des Ergebnisses |
|---|---|
| Integer | Integer |
| Real | Real |
| Short_Integer | Integer |
| Long_Integer | Long_Integer |
| Short_Real | Real |
| Long_Real | Long_Real |

Tabelle 15-1: Ergebnistypen bei Abs und Sqr

Für die anderen arithmetischen Funktionen gilt:

| Typ des Parameters | Typ des Ergebnisses |
|---|---|
| Integer | Real |
| Real | Real |
| Short_Real | Short_Real |
| Long_Real | Long_Real |
| universeller Real-Typ | universeller Real-Typ |
| Integer-Typ | universeller Real-Typ |

Tabelle 15-2: Ergebnistypen der anderen arithmetischen Funktionen

Der universelle Real-Typ paßt sich dem Kontext an, siehe 9.3.1.

Ein Aufruf der Funktionen Abs und Sqr mit einem statischen Ausdruck als Aktualparameter ist wieder ein statischer Ausdruck. Aufrufe der anderen arithmetischen Standardfunktionen sind keine statischen Ausdrücke.

Die Berechnung der arithmetischen Funktionen wird mindestens mit der Genauigkeit des Ergebnistyps vorgenommen, kann aber auch mit einer höheren Genauigkeit erfolgen (siehe auch Kapitel 9.3.1).
Beim universellen Real-Typ als Ergebnistyp wird mindestens mit der Genauigkeit von Long_Real gerechnet.

### Abs(x)

berechnet den Absolutbetrag von x.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| `Numeric_Error` | - Bei Abs(x) liegt das Funktionsergebnis nicht im Wertebereich des Ergebnistyps (Integer bzw. Long_Integer bzw. Short_Real bzw. Long_Real). |

### Arctan(x)

berechnet den Hauptwert des Arcustangens von x im Bogenmaß.

### Cos(x)

berechnet die Cosinusfunktion von x. Dabei muß x im Bogenmaß angegeben werden.

### Exp(x)

berechnet `e**x`, wobei e die Basis des natürlichen Logarithmus ist.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| `Numeric_Error` | - Das Ergebnis von Exp(x) liegt nicht im Wertebereich des Ergebnistyps. |

### Ln(x)

berechnet den natürlichen Logarithmus von x.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Numeric_Error | - Bei Ln(x) ist x <= 0. |

### Sin(x)

berechnet die Sinusfunktion von x. Dabei muß x im Bogenmaß angegeben werden.

### Sqr(x)

berechnet das Quadrat von x (also x*x).

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Numeric_Error | - Bei Sqr(x) liegt das Funktionsergebnis nicht im Wertebereich des Ergebnistyps (Integer bzw. Long_Integer bzw. Short Real_bzw. Long_Real). |

### Sqrt(x)

berechnet die Wurzel von x.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Numeric_Error | - Bei Sqrt(x) ist x < 0. |

# Umwandlungsfunktionen

Der Aufruf einer Umwandlungsfunktionen mit einem statischen Ausdruck als Aktualparameter ist wieder ein statischer Ausdruck.

### Long (x)

wandelt den Wert des Ausdrucks x vom Typ Short_Integer in den gleichen Wert vom Typ Long_Integer um.

### Round(x), Short_Round(x), Long_Round(x)

Für den Ausdruck x, der von einem Real-Typ sein muß, liefern diese Funktionen ein Ergebnis des entsprechenden Integer-Typs:

- Der Ergebnistyp bei Round ist Integer,
- bei Short_Round ist es Short_Integer und
- bei Long_Round ist es Long_Integer.

Der Wert von Round(x) ist wie folgt definiert:

- `Round(x) = Trunc(x + 0.5) für x ≥ 0,`
- `Round(x) = Trunc(x - 0.5) für x < 0.`

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Numeric_Error | - Das Ergebnis von Round(x) oder Short_Round(x) oder Long_Round(x) liegt nicht im Wertebereich des Ergebnistyps (Integer bzw. Short_Integer bzw. Long_Integer). |

*Beispiel*

```
Round(3.5)   ergibt  4
Round(-3.5)  ergibt -4
```

### Trunc(x), Short_Trunc(x), Long_Trunc(x)

berechnet den ganzzahligen Anteil von x.

Für den Ausdruck x, der von einem Real-Typ sein muß, liefern diese Funktionen ein Ergebnis des entsprechenden Integer-Typs:

- Der Ergebnistyp bei Trunc ist Integer,
- bei Short_Trunc ist es Short_Integer und
- bei Long_Trunc ist es Long_Integer.

Der Wert von Trunc (x) wird durch folgende Forderungen definiert.

- Trunc(x) ist ganzzahlig,
- $0 \leq$ x-Trunc(x) $< 1$ für $x \geq 0$,
- $-1 <$ x-Trunc(x) $\leq 0$ für $x \leq 0$.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| Numeric_Error | - Das Ergebnis von Trunc(x) oder Short_Trunc(x) oder Long_Trunc(x) liegt nicht im Wertebereich des Ergebnistyps (Integer bzw. Short_Integer bzw. Long_Integer). |

*Beispiel*

```
Trunc(3.5)   ergibt  3
Trunc(-3.5)  ergibt -3
```

# Ordinalfunktionen

Der Aufruf einer Ordinalfunktionen mit einem statischen Ausdruck als Aktualparameter ist wieder ein statischer Ausdruck.

### Card (s)

liefert für den Ausdruck s, der von einem SET-Typ sein muß, die Anzahl der Elemente, die in der durch den Wert von s bestimmten Menge enthalten sind. Das Ergebnis ist vom Typ Integer.

### Chr (x)

Für den Ausdruck x von einem Integer-Typ liefert diese Funktion denjenigen Wert des Char-Typs, der durch den Wert von x codiert ist.
Für jeden Wert c des Char-Typs gilt: Chr (Ord (c)) = c

| | |
|---|---|
| `Range_Error` | `- Der Zeichenwert Chr(x) liegt nicht im Wertebereich des Typs Char.` |

*Hinweis*

Der Zeichensatz eines Pascal-XT-Prozessors ist implementierungsdefiniert (siehe auch 6.2.3). Falls ein Programm für unterschiedliche Pascal-XT-Implementierungen geschrieben wird, darf über die Zuordnung Integer-Wert zu Zeichen-Wert und umgekehrt keine weitere Annahme gemacht werden.

*Querverweis*

Char: 6.2.3

### Ord (x)

Für den Ausdruck x, dessen Typ ein Ordinaltyp sein muß, liefert diese Funktion als Ergebnis diejenige Zahl vom Typ Integer, die als Ordinalzahl des Wertes des Ausdrucks x definiert ist.

*Querverweise*

Char: 6.2.3
Aufzählungstyp: 6.2.5

### Pred(x)

Für den Ausdruck x, dessen Typ ein Ordinaltyp sein muß, liefert diese Funktion den Wert desselben Typs, dessen Ordinalzahl gegenüber der des Ausdrucks x um 1 erniedrigt ist (predecessor).

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Range_Error | - Das Ergebnis von Pred (x) liegt nicht im Wertebereich des Typs von x. |

### Setmax (s)

Setmax (s) liefert für einen Ausdruck s, der von einem SET-Typ sein muß, den Wert des größten Elements, das in der durch den Wert von s bestimmten Menge enthalten ist. Der Ergebnistyp ist der Basistyp des SET-Typs von s. Die leere Menge (d.h. "[]") darf nicht als Parameter angegeben werden.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Set_Error | - Bei Setmax (s) ist der Wert des Ausdrucks s gleich der leeren Menge (siehe 9.4). |

### Setmin (s)

Setmin (s) liefert für einen Ausdruck s, der von einem SET-Typ sein muß, den Wert des kleinsten Elements, das in der durch den Wert von s bestimmten Menge enthalten ist. Der Ergebnistyp ist der Basistyp des SET-Typs von s. Die leere Menge (d.h. "[]") darf nicht als Parameter angegeben werden.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Set_Error | - Bei Setmin (s) ist der Wert des Ausdrucks s gleich der leeren Menge (siehe 9.4). |

*Querverweise*

Mengen: 6.3.4
Mengenbildner: 9.4

### Succ (x)

Für den Ausdruck x, dessen Typ ein Ordinaltyp sein muß, liefert diese Funktion den Wert desselben Typs, dessen Ordinalzahl gegenüber der des Ausdrucks x um 1 erhöht ist (successor).

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Range_Error | - Das Ergebnis von Succ(x) liegt nicht im Wertebereich des Typs von x. |

# Boolesche Funktionen

Die booleschen Funktionen Eof und Eoln sind bereits in Abschnitt 15.1 (Dateibearbeitung) beschrieben worden.

**Odd (x)**

Die Funktion liefert für den Ausdruck x, der vom Integer-Typ sein muß, den booleschen Wert True, wenn x ungerade ist, und den booleschen Wert False, wenn x gerade ist.
Der Aufruf der booleschen Funktion Old mit einem statischen Ausdruck als Aktualparameter ist wieder ein statischer Ausdruck.

# Transferprozeduren

Die Transferprozeduren Pack und Unpack ermöglichen die Übertragung von Komponenten zwischen gepackten und ungepackten ARRAYs, die denselben Komponententyp haben.
In Pascal-XT können zusätzlich Komponenten zwischen ungepackten ARRAYs mit dem Komponententyp Char und Strings übertragen werden.
Insbesondere ist es möglich, nur einen Teilbereich zu übertragen.

Beide Prozeduren besitzen die Parameter z, a und i, für die gilt:

z ist vom Typ PACKED ARRAY [s2] OF t oder vom Typ String[n]
a ist vom Typ ARRAY [s1] OF t
i ist ein Ausdruck, dessen Wert zuweisungsverträglich mit s1 ist

Ist z vom Typ PACKED ARRAY, dann müssen z und a denselben Komponententyp t besitzen. t darf kein FILE-Typ sein und keine Komponente eines FILE-Typs enthalten.
Ist z vom Typ String[n], dann muß der Komponententyp t von a der Typ Char sein.

### Pack (a, i, z)

Die Prozedur Pack übertragt Komponenten eines Ausdrucks von einem ungepackten ARRAY-Typ in eine Variable eines gepackten ARRAY-Typs oder String-Typs.
In Norm-Pascal muß a eine verallgemeinerte Variable sein.

#### Packen in ein ARRAY

Pack überträgt aus dem Ausdruck a eines ungepackten ARRAY-Typs, beginnend ab Index i, soviele Komponenten in die Variable z, wie in diese hineinpassen. a muß ab Index i noch mindestens soviele Komponenten enthalten, wie übertragen werden müssen, d. h. es muß gelten

```
Last(s1) - i + 1 >= Last(s2) - First(s2) + 1
```

#### Packen in einen String

Pack überträgt aus dem ungepackten ARRAY-Ausdruck a alle Zeichen von a[i] bis a[Last(s1)] in die String-Variable z. Die Anzahl der übertragenen Zeichen muß kleiner oder gleich der Maximallänge von z sein, d.h. es muß gelten:

Last(s1) - i + 1 <= Maxlength(z)

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Index_Error | - Bei Pack (a,i,z) liegt der Wert des Ausdrucks i nicht im Wertebereich des Indextyps des ungepackten Array-Parameters a.<br><br>- Bei Pack (a,i,z) wird beim Übertragen der Komponenten aus dem ungepackten Array a ab Index i in das gepackte gepackte Array z der Indexbereich von a überschritten. |
| String_Error | - Bei Pack (a,i,z) ist die Maximallänge der String-Variable z zu klein, um alle Zeichen aus dem ungepackten Array a ab Index i aufzunehmen. |
| undefinierte Auswirkungen | - Bei Pack (a,i,z) wird auf eine Komponente des ungepack-ten Arrays a zugegriffen, die undefiniert ist. |

*Beispiele*

Die Komponenten a[11] bis a[20] werden in die Variable p übertragen.

```
TYPE
   t = ARRAY [-40..40] OF Boolean;
   tp = PACKED ARRAY [1..10] OF Boolean;
VAR
   a : t;
   p : tp;
BEGIN
   Pack (a, 11, p);
END
```

Die Komponenten a[17] bis a[35] werden in die String-Variable s übertragen.

```
TYPE
   t = ARRAY [-10..35] OF Boolean;
VAR
   a : t;
   s : String;
BEGIN
   Pack (a, 17, s);
END
```

*Querverweise*

PACKED: 6.3
ARRAY-Typen: 6.3.1
String-Typen: 6.3.2.2
First, Last: 15.9
Maxlength: 15.9

## Unpack (z, a, i)

Die Prozedur Unpack überträgt Daten von einem gepackten ARRAY oder einem String in einen angebbaren Bereich eines ungepacktes ARRAY.
In Norm-Pascal muß z eine verallgemeinerte Variable sein.

### Entpacken eines gepackten ARRAY

Unpack überträgt aus dem gepackten ARRAY-Ausdruck z alle Komponenten in die ungepackte ARRAY-Variable a, beginnend ab a[i]. Die Anzahl der Komponenten des ARRAY a von i bis Last (s1) muß größer oder gleich der Anzahl der Komponenten des gepackten ARRAY z sein, d. h. es muß gelten:

Last(s1) - i + 1 >= Last(s2) - First(s2) + 1

### Entpacken eines Strings

Unpack überträgt alle Zeichen des variablen Zeichenkettenausdrucks z in die ungepackte ARRAY-Variable a, beginnend ab a[i]. Die Anzahl der übertragenen Zeichen ist gleich der aktuellen Länge (Length (z)) des Ausdrucks z und es muß gelten:

Last(s1) - i + 1 >= Length(z)

### Mögliche Laufzeitfehler:

| | |
|---|---|
| Index_Error | - Bei Unpack (z,a,i) liegt der Wert des Ausdrucks i nicht im Wertebereich des Indextyps des ungepackten Array-Parameters a.<br><br>- Bei Unpack (z,a,i) ist der Bereich im ungepackten Array a ab Index i zu klein, um alle Komponenten des gepackten Array z aufzunehmen.<br><br>- Bei Unpack (z,a,i) enthält der Zeichenkettenausdruck z mehr Zeichen, als in das ungepackte Array a ab Index i übertragen werden können. |
| undefinierte Auswirkungen | - Bei Unpack (z,a,i) ist irgendeine Komponente des gepackten Arrays z undefiniert. |

*Beispiele*

Die Komponenten a[4] bis a[11] werden durch die Komponenten z[3] bis z[10] ersetzt.

```
TYPE
   tz = PACKED ARRAY [3 .. 10] OF Integer;
   ta = ARRAY [0 .. 39] OF Integer;
VAR
   z : tz;
   a : ta;
BEGIN
   z := tz (1, 2, 3, 4, 5, 6, 7, 8);
   unpack (z, a, 4);
END
```

Die Komponenten c[13] bis c[17] werden durch die 5 Zeichen ('H','a','l','l,'o') der String-Variablen s ersetzt.

```
TYPE
   tc = ARRAY [10..60] OF Char;
VAR
   c : tc;
   s : String;
BEGIN
   s := 'Hallo';
   Unpack (s, c, 13);
END
```

*Querverweise*

| | |
|---|---|
| PACKED: | 6.3 |
| ARRAY-Typen: | 6.3.1 |
| String-Typen: | 6.3.2.2 |
| Length: | 15.3 |
| First, Last: | 15.9 |

# Attributfunktionen

Attribut-Funktionen liefern primär Angaben zu Typen. Daher ist i. a. das Argument einer Attribut-Funktion ein Typ-Bezeichner. Es ist jedoch auch eine Variable als Argument erlaubt. In diesem Fall liefern die Attribut-Funktionen die Werte der Attribute des Typs der Variablen.

Die Angaben beziehen sich auf die

- Ausrichtung (Alignof),
- Anzahl der Bits zur Darstellung der Werte eines Ordinaltyps (Bitsizeof),
- den kleinsten Wert eines Ordinaltyps (First),
- den größten Wert eines Ordinaltyps (Last),
- die Maximallänge eines String-Typs (Maxlength),
- den Offset von Feldern eines Records (Offsetof) und
- den Speicherplatzbedarf (Sizeof).

Die Attributfunktionen sind stets statische Ausdrücke, mit Ausnahme von Sizeof (t, e) und Sizeof (t, c1, ..., cn, e), wenn der Ausdruck e nicht statisch ist.

Der Ergebnistyp bei den Funktionen First und Last ist derselbe Typ wie der des Aktualparameters. Bei den anderen Funktionen ist es Short_Integer, falls der Wert des Ergebnisses innerhalb des Wertebereichs von Short_Integer liegt. Andernfalls ist der Ergebnistyp Long_Integer.

Bei den nicht statischen Aufrufen Sizeof (t, e) oder Sizeof (t, c1, ..., cn, e), ist der Ergebnistyp derselbe wie der eines Aufrufs von Sizeof ohne den letzten Parameter e.

**Implementierungsdefinierte Eigenschaft**.

Die Größe einer Speichereinheit ist implementierungsdefiniert. Sie kann 1 Byte oder ein Vielfaches (i.a. eine Zweierpotenz) von einem Byte sein.

*Hinweis*

Existieren in einem Programm Schnittstellen zu Programmteilen, die in anderen Sprachen formuliert sind, so kann mit den Attributfunktionen Auskunft über die Speicherdarstellung der zu übergebenden Daten erhalten werden.

*Querverweise*

Speicherdarstellung: 6.3.3.2
Attribute: 6.7

### Alignof (t)

Alignof liefert die Ausrichtung im Speicher, die für Variablen des Typs t gefordert wird. Die Ausrichtung ist eine ganze positive Zahl, normalerweise eine Zweierpotenz und richtet sich nach den Zugriffsmöglichkeiten des Prozessors auf den Speicher. Die Adresse jeder Variablen des Typs t ist ein ganzzahliges Vielfaches von Alignof (t).

### Bitsizeof (t)

Die Funktion Bitsizeof (f) liefert die minimale Anzahl der Bits, die notwendig sind, um die Werte von t darzustellen. Der Typ t muß ein Ordinaltyp sein.

*Beispiel*

```
TYPE
   t1 = (c1, c2, c3);
   t2 = (k1, k2, k3, k4);
   r = PACKED RECORD
       a (0 : 0..bibsizeof (t1)) : t1;
       b (0 : bitsizeof (t1)..7) : t2;
       END;
```

### First (t)

Liefert den kleinsten Wert des Ordinaltyps t.

### Last (t)

Liefert den größten Wert des Ordinaltyps t.

*Beispiel*

```
VAR
   i : ordinal_typ;
BEGIN
   ...
   FOR i := First (ordinal_typ) TO Last (ordinal_typ) DO ... ;
   ...
END
```

Die gleiche Wirkung hätte folgende Anweisung. In diesem Fall werden die Werte von First und Last von dem Typ von i abgeleitet.

```
BEGIN
   ...
   FOR i := First (i) TO Last (i) DO ... ;
   ...
END
```

### Maxlength (t)

Liefert die maximale Länge des String-Typs t.

*Beispiel*

```
BEGIN
   ...
   IF Length (s1) + Length (s2) > Maxlength (s) THEN
      fehlerbehandlung
   ELSE
      s := Concat (s1, s2);
   ...
END
```

### Offsetof (t, f)

Die Funktion liefert den Abstand eines RECORD-Feldes relativ zum RECORD-Anfang in Anzahl von Speichereinheiten. t bezeichnet einen beliebigen RECORD-Typ oder eine Variable eines beliebigen RECORD-Typs. f ist ein Feld-Bezeichner, der unmittelbar in t enthalten sein muß, d.h., f darf nicht in einem eingeschachtelten RECORD enthalten sein.
Die Größe einer Speichereinheit ist implementierungsdefiniert (siehe A.6).

### Sizeof (t)

Sizeof (t) liefert die Anzahl der Speichereinheiten, die zur Darstellung der Werte des Typs t benötigt werden. Die Größe einer Speichereinheit ist implementierungsdefiniert (siehe A.6).

### Sizeof (t, e)

Sizeof (t, e) liefert die Anzahl der Speichereinheiten, die für eine dynamische Variable p↑ vom Typ t bei einem Aufruf New (p, e) belegt werden.

Der Typ t muß ein ARRAY-Typ, ein String-Typ oder ein RECORD-Typ, dessen letztes Feld der Feldliste einen ARRAY-Typ oder einen String-Typ besitzt, sein.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen und sein Wert darf nicht kleiner 1 und nicht größer als die Maximallänge des String-Typs sein.

### Sizeof (t, c1, ..., cn)

Sizeof (t, c1, ..., cn) liefert die Anzahl der Speichereinheiten, die für eine dynamische Variable p↑ vom Typ t bei einem Aufruf New (p, c1, ..., cn) belegt werden.

Der Typ t muß ein RECORD-Typ sein. Er muß ineinandergeschachtelte Varianten enthalten, zu denen die Selektorkonstanten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein. Varianten, die nicht aufgeführt sind, müssen in einer tieferen Verschachtelungsebene stehen als cn.

### Sizeof (t, c1, ..., cn, e)

Sizeof (t, c1, ..., cn, e) liefert die Anzahl der Speichereinheiten, die für eine dynamische Variable p↑ vom Typ t bei einem Aufruf New (p, c1, ..., cn, e) belegt werden.

Der Typ t muß ein RECORD-Typ mit Varianten sein, zu denen die Selektorkonstaten c1, ..., cn gehören. Diese Selektorkonstanten müssen in der durch die Verschachtelung der Variantenteile definierten Reihenfolge aufgezählt sein.

Die Feldliste der zur letzten Selektorkonstanten cn gehörigen Variante darf keine weiteren eingeschachtelten Varianten enthalten. Das letzte Feld der

Feldliste der innersten Variante muß einen ARRAY- oder String-Typ besitzen.

Bei einem ARRAY-Typ muß der Ausdruck e zuweisungsverträglich zum Index-Typ des ARRAY-Typs sein. Bei einen String-Typ muß der Ausdruck e einen Integer-Typ besitzen und sein Wert darf nicht kleiner als 1 und nicht größer als die Maximallänge des String-Typs sein.

*Beispiel zur Verwendung von Attributfunktionen*

```
...
TYPE
   farbe = (rot,gelb,gruen,blau,braun,schwarz);
   daten = PACKED RECORD
      faerbung (0: 0..Bitsizeof(farbe)-1): farbe;
      ...
      END;
   bytes = PACKED ARRAY [1..Sizeof(daten)] OF Char;
...

VAR
   f : farbe;
   d : daten;
   b : bytes;
...

BEGIN
   ...

   FOR f := First (farbe) TO Last (farbe) DO ...

   ...
END
```

# Funktion zur unüberprüften Typkonvertierung

In Pascal-XT gibt es eine Funktion zur unüberprüften Typkonvertierung zwischen Werten gleicher Speicherdarstellung.

### Convert (x, t)

Für den Ausdruck x und den Typ-Bezeichner t liefert diese Funktion jenen Wert des Typs t, der die gleiche Darstellung wie der Wert x im Speicher hat. Der Typ t darf kein FILE-Typ sein und er darf keine Komponente eines FILE-Typs enthalten. Ist sowohl der Typ t als auch der Typ von x ein Ordinaltyp, so muß gelten

```
Ord (First (t)) <= Ord (x) <= Ord (Last (t)),
```

Für alle übrigen Typkombinationen muß gelten

```
Sizeof (x) = Sizeof (t).
```

Ist der Ausdruck x ein statischer Ausdruck eines skalaren oder Zeigertyps und ist der Typ t ebenfalls ein skalarer Typ oder ein Zeigertyp, so ist auch der Aufruf der Funktion Convert ein statischer Ausdruck.

#### Mögliche Laufzeitfehler:

| | |
|---|---|
| undefinierte Auswirkung | - Bei Convert (x, t) repräsentiert die Speicherdarstellung von x keinen zulässigen Wert von t. |

*Hinweis*

Es ist leicht zu ersehen, daß diese Funktion eine Hintertür zum Verlassen des Pascal-gemäßen Programmierstils ist. Sie hat jedoch eine gewisse Berechtigung, wenn es um die Lösung von Problemen in maschinennaher Umgebung geht (z.B. direkte Programmierung der Datenpufferung für Massenspeicher). Bei der Lösung von Problemen auf Anwenderebene sei vor der Umgehung des Typ-Konzepts dringend gewarnt!

# Unterprogramme zur Ausnahmebehandlung

### Error_Number

Diese parameterlose Funktion liefert die Fehlernummer (siehe 14.2) der zuletzt aufgetretenen Ausnahmesituation (Fehlersituation). Ihr Wert ist 0, falls noch keine Ausnahmesituation aufgetreten ist.

### Raise (n)

Mit Raise kann eine programmierte Ausnahmesituation (Fehlersituation) mit der Fehlernummer n erzeugt oder innerhalb eines Ausnahmebehandlungsteils propagiert werden. Der Parameter n muß ein Ausdruck eines Integer-Typs sein und hat folgende Bedeutung:

n < 0: Die negativen Fehlernummern (und die 0) sind für das Pascal-XT System reserviert. Für die Werte -1 bis -16 gibt es vordefinierte Bezeichner (siehe 14.1, 5.2).

n = 0: Eine zuvor aufgetretene Ausnahme wird propagiert (weitergeleitet), ohne daß die Fehlernummer verändert wird und ohne daß die Informationen über den ursprünglichen Fehlerort verloren gehen (im Gegensatz zu Raise (Error_Number)). Es tritt ein System_Error auf, wenn vor dem Aufruf von Raise (0) keine Ausnahme aufgetreten ist. n = 0 stellt somit keine Fehlernummer dar.

n > 0: Die positiven Fehlernummern können beliebig verwendet werden.

Ist n <> 0, dann ist die Wirkung analog zu der bei Auftreten eines "echten" Fehlers, d.h. an der Stelle des Raise-Aufrufs wird die Ausnahme mit der Nummer n erzeugt. Kann ein Fehler in einem Ausnahmebehandlungsteil nicht behandelt werden, dann kann er mit Raise auf zwei Arten propagiert weitergeleitet werden:

1) Raise (Error_Number)
   Der zuletzt aufgetretene Fehler wird nochmals ausgelöst. Die Zeile mit dem Raise-Aufruf stellt den neuen Fehlerort dar und die Informationen über den ursprünglichen Fehlerort sind damit verloren gegangen.

2) Raise (0)
   Der zuletzt aufgetretene Fehler wird propagiert, ohne daß die Informationen über den ursprünglichen Fehlerort verloren gehen.

Zur Propagierung von Fehlern sollte immer Raise (0) verwendet werden, damit Informationen über den ursprünglichen Fehlerort für weitere (systemabhän-

gige) Diagnosezwecke erhalten bleiben (siehe auch Beispiele 14-5 in Kapitel 14 und Benutzerhandbuch).

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| System_Error | - Raise (0) wurde aufgerufen, obwohl davor in diesem Programm noch kein Fehler aufgetreten und daher keine Fehler-Propagierung möglich ist. |

*Hinweise*

- In einem Programm sollten die vom Anwender benutzten Fehlernummern eindeutig sein, damit im Fehlerfall der Ursprungsort des Fehlers eindeutig identifiziert werden kann.
- Die Pascal-XT Implementierungen auf den verschiedenen Systemen bieten Unterstützung an bei der Ausgabe der dynamischen Aufrufkette im Fehlerfall.
- Die Pascal-XT Implementierungen auf den verschiedenen Systemen bieten in einem vordefinierten Paket Errors eine Prozedur ReRaise zum Propagieren von Fehlern an. Zur besseren Verständlichkeit und Portierbarkeit kann kann diese Prozedur anstelle von Raise (0) verwendet werden.

*Querverweise*

Fehlernummer, vordefinierte Ausnahmen: 14.1, 5.2
Behandlung von Ausnahmen: 14.3

*Beispiel für Error_Number und Raise*

```
PROGRAM beispiel (Output);
VAR
   i, j : Long_Integer;

PROCEDURE check_ueberlauf (n,m:Long_Integer);
BEGIN
   IF (Long_Maxint-m) < n THEN
      Raise (1);
END;

BEGIN
   ....
   check_ueberlauf (i,j):
   ....
EXCEPTION
   IF Error_Number = 1 THEN
      Writeln ('Überlauf aufgetreten')
   ELSE
      Raise (0);
END.
```

# Prozedur zur expliziten Paketinitialisierung

### Elaborate (p)

Der Parameter p muß ein Paket-Bezeichner sein. Ein Aufruf der Prozedur Elaborate (p) stellt sicher, daß die Initialisierung des Paketes p spätestens nach dem Aufruf von Elaborate (p) erfolgt ist. Wenn vor dem Aufruf von Elaborate die Initialisierung bereits erfolgt war, so hat der Aufruf keine Wirkung.

Elaborate darf nur innerhalb der Anweisungsfolge eines Paket-Blocks aufgerufen werden. Innerhalb von
- Wiederholungsanweisungen (WHILE, REPEAT, FOR)
- bedingten Anweisungen (IF, CASE)

des Paket-Blocks darf Elaborate nicht auftreten.

**Mögliche Laufzeitfehler:**

| | |
|---|---|
| Elab_Error | - Die Initialisierung der Pakete eines Programms kann nicht fortgesetzt werden, da durch die Verwendung der vordefinierten Prozedur Elaborate Zyklen bei der Initialisierung entstehen. |

*Beispiel*

Das Paket a verwendet bei der Initialisierung aus dem Paket b die globale Variable status, die vor ihrer Verwendung im Paket b initialisiert werden muß. Durch den Aufruf von Elaborate (b) vor der Verwendung von status im Paket a wird garantiert, daß Paket b initialisiert ist.

```
PACKAGE a;
   ...

END.

WITH b;
PACKAGE BODY a;
   ...
BEGIN
   Elaborate (b);
   IF b.status THEN ...;
   ...
END.
```

```
PACKAGE b;
VAR
   status: Integer;
END.


PACKAGE BODY b;
   ...
BEGIN
   status := 0;
   ...

END.
```

*Querverweise*

Programmausführung: 13.3.3

# Steueranweisungen an den Compiler

Ein Kommentar, der unmittelbar nach der öffnenden Kommentarklammer mit einem Dollarzeichen beginnt, heißt Pseudokommentar und enthält Steueranweisungen an den Compiler.

```
Pseudokommentar  = "{$" Steueranweisung {"," Steueranweisung} "}".
Steueranweisung  = Option [ "=" ("On"         |
                                 "Off"        |
                                 "Restricted" |
                                 Zeichenkette) ].
Option           = Bezeichner.
```

Pseudokommentare haben, wie Kommentare, keinen Einfluß auf die logische Ausführung eines korrekten Pascal-Programms, u.U. jedoch auf dessen physikalisches Verhalten (Ablaufgeschwindigkeit, Speicherplatzbedarf, zusätzliche Fehlermeldungen zur Laufzeit, auf den Umfang des Übersetzungsprotokolls oder auf die Verwendbarkeit der Testhilfe).

Optionen werden in 2 Klassen eingeteilt:

- Globale Optionen
  gelten für die gesamte Übersetzungseinheit. Sie können nur durch Steueranweisungen, die vor dem eigentlichen Beginn der Übersetzungseinheit (d.h. vor den Wortsymbolen WITH, PROGRAM bzw. PACKAGE) stehen, eingestellt werden.
- Lokale Optionen
  können an beliebigen Stellen des Programms durch Steueranweisungen eingestellt werden.

In ihrer Wirkung lassen sich die Steueranweisungen weiter einteilen:

- Steuerung des Übersetzungsprotokolls (z.B. vollständiges Quellprotokoll, nur Fehlerliste, Querverweisliste, Objektcode-Protokoll).
- Beeinflussung des Laufzeitverhaltens (z.B. Abschalten der Fehlererkennung zur Laufzeit, Optimierung des Objektcodes).
- Generierung von Testtabellen für die symbolische Testhilfe.

Viele der Steueranweisungen haben die Wirkung von "Schaltern" zum Ein- bzw. Ausschalten von Optionen. In diesen Fällen kann der für die Option angegebene Wert nur

On oder Off

sein. On dient zum Einschalten, Off zum Ausschalten der betreffenden Option.

Zum Einschalten der Option genügt es auch, nur die Option, ohne "= On", anzugeben.

Der Wert Restricted hat nur eine Bedeutung für die Steueranweisung zur Generierung von Testtabellen. Eine Zeichenkette kann nur bei der Option Title angegeben werden.

### Implementierungsdefinierte Eigenschaften

Die Voreinstellungen der Compileroptionen sind implementierungsdefiniert.

Ob und wie Optionen bei der Bedienung (beim Aufruf des Compilers) eingestellt werden können, ist implementierungsdefiniert.

*Beispiele für Steueranweisungen*

```
{$ List=On, Check=Off}
```

```
{$ Page}
```

*Hinweise*

- Während der Entwicklung von Pascal-Programmen sollten unbedingt die Steueranweisungen Check = On und Initialize = On angegeben werden. Damit können frühzeitig Fehler entdeckt werden, die u.U. sonst unentdeckt bleiben. Werden die beiden Optionen nicht angegeben, so ist die Auswirkung eines auftretenden Fehlers, der mit den Optionen erkannt wird, undefiniert. Für die Laufzeitüberprüfungen wird allerdings zusätzlich Code erzeugt und damit auch die Laufzeit der Programme erhöht. Daher sollte man bei der Übersetzung von Benchmark-Programmen für Laufzeitmessungen darauf achten, daß Check = Off, Initialize = Off und Optimize = On eingestellt ist.

# Globale Optionen

Die globalen Optionen gelten für die gesamte Übersetzungseinheit und müssen vor dem Schlüsselwort WITH, PROGRAM bzw. PACKAGE angegeben werden.

**Generate**

schaltet die Erzeugung von Objekt-Code ein oder aus. Bei der Übersetzung von Paketspezifikationen wird generell kein Code erzeugt.

[= On] Für diese Übersetzungseinheit wird Objektcode erzeugt.

= Off Für diese Übersetzungseinheit wird kein Objektcode erzeugt. Die lokalen Optionen Assembler, Check, Initialize und Optimize, sowie die globalen Optionen Debug und Map haben dann in dieser Übersetzungseinheit keine Wirkung.

**Debug**

steuert die Erzeugung von Testtabellen. Diese Testtabellen werden beim Testen mit einer symbolischen Testhilfe benötigt (siehe Benutzerhandbuch [1]). Die Testtabellen werden nur für Programme und Paket-Implementierungen erzeugt, wenn die Option Generate = On ist. Bei Paket-Spezifikationen wird die Option Debug erst bei der Übersetzung der Paket-Implementierung wirksam.

Durch die Angabe von Debug = On oder Debug = Restricted in einer Paket-Spezifikation, können alle in ihr definierten Konstanten-, Variablen- und Typ-Bezeichner in Testhilfe-Anweisungen verwendet werden. Wurde die Option in einer Paket-Implementierung angegeben, dann können sowohl die Deklarationen und Zeilennummern dieser Paket-Implementierung als auch die Deklarationen der zugehörigen Paket-Spezifikation in Testhilfe-Anweisungen verwendet werden.

Ist Debug = On oder Debug = Restricted angegeben, dann ist die Option Optimize wirkungslos.

[= On] Diese Übersetzungseinheit kann mit der symbolischen Testhilfe getestet werden.

= Restricted
Diese Übersetzungseinheit kann mit der symbolischen Testhilfe eingeschränkt getestet werden. An Testpunkten innerhalb dieser Übersetzungseinheit sind Zuweisungen als Testhilfe-Anweisungen nicht möglich.

= Off Diese Übersetzungseinheit kann nicht mit der symbolischen Testhilfe getestet werden.

**Map**

schaltet die Ausgabe der Adreßtabellen in das Übersetzungsprotokoll ein oder aus. Die Option Map ist nur wirksam bei der Übersetzung von Programmen und Paket-Implementierungen, wenn die Option Generate = On ist. Bei Paket-Implementierungen enthalten die ausgegebenen Tabellen auch die Objekte der zugehörigen Spezifikation.

[= On] Für diese Übersetzungseinheit werden Adreßtabellen erzeugt.

= Off Es werden keine Adreßtabellen erzeugt.

**Standard**

legt fest, ob der Sprachumfang von Norm-Pascal oder Pascal-XT akzeptiert wird.

[= On] Es wird der Sprachumfang von Norm-Pascal Level 1 akzeptiert. Abweichungen von Norm-Pascal werden als Fehler gemeldet. Die in Pascal-XT zusätzlich eingeführten Wortsymbole können als normale Bezeichner verwendet werden.

= L0 Wie Argument On, nur daß Norm-Pascal Level 0 (ohne Konformreihungsschema) akzeptiert wird. Dieses Argument kann nur beim Aufruf der Compiler angegeben werden.

= Off Es wird der gesamte Sprachumfang von Pascal-XT akzeptiert.

**Xref**

steuert die Ausgabe einer Querverweisliste.

[= On] Es wird eine Querverweisliste aller in dieser Übersetzungseinheit verwendeten Bezeichner ausgegeben. Dabei werden auch solche Bezeichner berücksichtigt, die ihren Definitionspunkt in einer fremden Übersetzungseinheit haben oder vordefiniert sind.

= Off Es wird keine Querverweisliste ausgegeben.

# Lokale Optionen

Lokale Optionen können überall im Programm angegeben werden. Der Gültigkeitsbereich ist für jede Option angegeben.

**Assembler**

steuert die Ausgabe des Objektcode-Protokolls in Assembler-Format. Die Option gilt für den gesamten Anweisungsteil eines Blocks, wenn sie vor oder innerhalb des Anweisungsteils eingeschaltet wird. Das Ausschalten der Option in einem Anweisungsteil wird erst am Ende dieses Anweisungsteils wirksam.

[= On] Die Ausgabe des Objektcode-Protokolls wird eingeschaltet.

= Off Die Ausgabe des Objektcode-Protokolls wird ausgeschaltet.

**Check**

steuert die Generierung von zusätzlichen Befehlen (Check-Code) zur Erkennung von Laufzeitfehlern.

Sie gilt für alle Anweisungen, in denen die Option eingeschaltet ist.

*Hinweis:*

Während der Programm-Entwicklung sollte die Check-Option zur frühzeitigen Fehlererkennung für die ganze Übersetzungseinheit eingeschaltet werden.

[= On] Die Erzeugung von Check-Code wird eingeschaltet.

= Off Die Erzeugung von Check-Code wird ausgeschaltet.
Die Fehler Numeric_Error, Range_Error, Set_Error, String_Error, Index_Error, Pointer_Error, Variant_Error und Case_Error werden nicht unbedingt erkannt und haben i. a. undefinierte Auswirkungen zur Folge.

**Optimize**

steuert die Optimierung des Objektcodes. Es werden Optimierungen zur Verringerung der Laufzeit durchgeführt. Die Option gilt für den gesamten Anweisungsteil eines Blocks, wenn sie vor oder innerhalb des Anweisungsteils eingeschaltet wird. Das Ausschalten der Option in einem Anweisungsteil wird erst am Ende dieses Anweisungsteils wirksam.

[= On] Die Optimierung wird eingeschaltet.

= Off Die Optimierung wird ausgeschaltet.

### Initialize

steuert die Initialisierung von Speicherbereichen für Variable mit einem implementierungsabhängigen Wert. Die Initialisierung der Speicherbereiche erfolgt

des Programms
- für lokale Variable eines Unterprogramms beim Aufruf des Unterprogramms
- für dynamisch erzeugte Variable beim Aufruf von New.

Der Initialisierungs-Wert wird so gewählt, daß z. B. eine nicht definierte Zeigervariable als undefiniert erkannt wird (bei Check = On). Er ist jedoch nicht als Ersatz für eine programmierte Erst-Zuweisung eines Wertes an eine Variable, sondern ausschließlich zur Auffindung von Fehlern gedacht! Mit dieser Option können i.a. nicht alle undefinierte Variablen erkannt werden.

Die Option Initialize ist nur bei eingeschalteter Option Check von Bedeutung. Während der Programm-Entwicklung sollten diese beiden Optionen unbedingt für die ganze Übersetzungseinheit eingeschaltet werden.

Die Option gilt für den gesamten Anweisungsteil eines Blocks, wenn sie vor oder innerhalb des Anweisungsteils eingeschaltet wird. Das Ausschalten der Option in einem Anweisungsteil wird erst am Ende dieses Anweisungsteils wirksam.

[= On] Es wird zusätzlicher Code zur Initialisierung erzeugt.

= Off Die Speicherbereiche werden nicht initialisiert.

### List

steuert die Ausgabe des Übersetzungsprotokolls.

[= On] Mit Beginn der Zeile, die diese Steueranweisung enthält, wird die Ausgabe des Übersetzungsprotokolls eingeschaltet.

= Off Mit Ende der Zeile, die diese Steueranweisung enthält, wird die Ausgabe des Übersetzungsprotokolls ausgeschaltet. Es werden dann nur noch fehlerhafte Quellzeilen zusammen mit den Fehlermeldungen protokolliert.

### Page

bewirkt einen Seitenvorschub unmittelbar nach der Zeile, in der diese Steueranweisung beginnt.

**Title = Zeichenkette**

steuert die Ausgabe eines Untertitels, der ab der nächsten Seite im Quellprotokoll ausgegeben wird. Durch die Angabe eines Leerstrings ('') wird die Ausgabe eines Untertitels beendet. Der Untertitel wird als Zeichenkette in Hochkommata angegeben.

*Beispiel*

```
{$ Title = 'Schnittstellenbeschreibung' }

{$ Title = '' }
```

# Das Paket-Konzept

Abstraktion und des Information Hiding. Der Unterschied von Pascal-XT zu vielen anderen Programmiersprachen ist, daß diese Prinzipien durch Pakete unterstützt und die Anwender der Sprache ermutigt werden, sie zu verwenden.

Für das Pascal-XT Paket-Konzept wurde der Begriff Programm weiter gefaßt als in Norm-Pascal vorgesehen. Ein Programm besteht aus einem Hauptprogramm und beliebig vielen Paketen. Wenn keine Pakete vorhanden sind, entspricht das dem alleinigen Hauptprogramm von Norm-Pascal.

Ein Paket besteht aus der Paket-Spezifikation und der Paket-Implementierung. In der Spezifikation wird beschrieben, WAS das Paket leistet und in der Implementierung, WIE es diese Leistung realisiert. Spezifikationen können an einen Anwender weitergegeben werden, ohne daß er die Implementierung kennt, da Paket-Spezifikation und Paket-Implementierung physikalisch getrennt verwaltet werden.

Als Beispiel sei die Bedienung eines Autos genannt. Die Schnittstelle bilden Lenkrad, Bremse und Gas. Das sind die sichtbaren Größen des Wagens. Der Fahrer muß nicht wissen, wie diese Bedienelemente funktionieren. Das sind Implementierungsdetails, die beliebig kompliziert sein können. Pascal-XT gibt in ähnlicher Weise die Möglichkeit, Einzelheiten in der Paket-Implementierung zu verstecken.

**Paket-Spezifikation**

In der Spezifikation sind die sichtbaren Größen eines Paketes definiert, die von anderen Paketen aus angesprochen werden können. Dies können Konstante, Typen, Variable und die Köpfe von Unterprogrammen sein. Nur auf diese Größen kann von außerhalb des Pakets zugegriffen werden.

*Beispiel*

```
PACKAGE complex_definitions;

TYPE complex = RECORD
                 realpart : Real;
                 imagpart : Real;
               END;

{ Funktionen zur Manipulation Complexer Zahlen }
FUNCTION make_complex (r, i : Real) : Complex;
   {r wird Real-Teil, i wird Imaginaer-Teil}

FUNCTION imag_part_of (c : complex) : Real;
   {Liefert den Imaginaer-Teil der Complexenzahl }

FUNCTION real_part_of (c : complex) : Real;
   {Liefert den Real-Teil der Complexenzahl }

{ Arithmetische Funktionen }
FUNCTION add      (left, right : complex) : complex;

FUNCTION subtract (left, right : complex) : complex;

FUNCTION multiply (left, right : complex) : complex;

FUNCTION divide   (left, right : complex) : complex;

END.
```

### Paket-Implementierung

Eine Paket-Implementierung besteht aus einem Vereinbarungs- und Anweisungs-Teil, wobei jeder leer sein kann. Wurden in der Spezifikation Unterprogramme deklariert, dann müssen in der Paket-Implementierung die Prozedur- bzw. Funktions-Identifikation angegeben werden. Dies gilt nicht für INLINE-Unterprogramme, für die in der Paket-Spezifikation bereits der Unterprogramm-Block angegeben werden muß, und für Unterprogramme mit einer Direktive. Desweiteren können Konstanten, Typen, Variablen und Unterprogramme vereinbart werden, die nur innerhalb der Paket-Implementierung benötigt werden und außerhalb des Pakets nicht sichtbar sind. Da Spezifikation und Implementierung logisch eine Einheit bilden, dürfen demzufolge Bezeichner, die in der Spezifikation deklariert wurden, in der Implementierung nicht nochmals deklariert werden.

Der Anweisungsteil dient der Paketinitialisierung und wird nur ein einziges Mal vor Ausführung des Programmes ausgeführt (siehe 13.3). Die im Deklarationsteil eines Paketes definierten Variablen können im Anweisungsteil mit Anfangswerten belegt werden. Wenn die erste Anweisung des Hauptprogramms ausgeführt wird, sind bereits alle Pakete initialisiert.

Zu einer Spezifikation muß es mindestens eine Implementierung geben. In Abhängigkeit von Randbedingungen kann es auch mehrere Implementierungen geben, die dieselbe Leistung des Pakets nur auf unterschiedliche Weise erbringen. So können z. B. verschiedene Versionen einer Implementierung Meldungstexte in verschiedenen Sprachen enthalten. Je nach Bedarf wird die gewünschte Realisierung ausgewählt, ohne daß das restliche Programm davon betroffen ist.

Zu der oben angegebenen Spezifikation sieht die Implementierung folgendermaßen aus:

```
PACKAGE BODY complex_definitions;

FUNCTION make_complex (r, i : Real) : complex;
   {r wird Real-Teil, i wird Imaginaer-Teil}
BEGIN
   make_complex := complex (r, i);
END;

FUNCTION imag_part_of (c : complex) : Real;
   {Liefert den Imaginaer-Teil der Complexenzahl }
BEGIN
   imag_part_of := c.imagpart;
END;

FUNCTION real_part_of (c : complex) : Real;
   {Liefert den Real-Teil der Complexenzahl }
BEGIN
   real_part_of := c.realpart;
END;

{ Arithmetische Funktionen }

FUNCTION add      (left, right : complex) : complex;
BEGIN ... END;

FUNCTION subtract (left, right : complex) : complex;
BEGIN ... END;

FUNCTION multiply (left, right : complex) : complex;
BEGIN ... END;

FUNCTION divide   (left, right : complex) : complex;
BEGIN ... END;

BEGIN
END { complex }.
```

In diesem Beispiel ist keine Initialisierung nötig. Im Falle eines Zufallszahlengenerators braucht man einen Startwert. Die Initialisierung geschieht folgendermaßen:

```
PACKAGE random_number_generator;

FUNCTION random : real;

END.



PACKAGE BODY random_number_generator;
VAR
   seed : integer;

FUNCTION random : Real;
BEGIN
   ...
END { random };

BEGIN { Initialisierung }
   seed := 123456;
END.
```

Die Variable Seed ist nur innerhalb des Pakets random_number_generator bekannt. Im Anweisungsteil wird der Variablen ein Startwert zugewiesen. Zum Zeitpunkt der Initialisierung des Pakets random_number_generator wird diese Zuweisung ausgeführt, bevor das Programm (das dieses Paket verwendet) gestartet wird.

### Übersetzungseinheiten

Hauptprogramme, Paket-Spezifikationen und Paket-Implementierungen werden als übersetzungseinheiten bezeichnet. Sie werden getrennt verwaltet und auch einzeln übersetzt.

### Beziehungen zwischen Übersetzungseinheiten

Sollen in einer Übersetzungseinheit auf Größen in einer Spezifikation eines anderen Paketes zugegriffen werden, dann muß der Name des Pakets in der WITH-Liste der Übersetzungseinheit angegeben werden. Damit werden alle Größen dieses Pakets sichtbar und können durch Angabe des Paket-Namens und des gewünschten Bezeichners verwendet werden. Das folgende Beispiel zeigt die Verwendung des Pakets complex_definitions im Programm some_program.

```
WITH complex_definitions;
PROGRAM some_program
VAR
   x, y, z : complex_definitions.complex;
BEGIN
   ...
   z := complex_definitions.add (x,y);
   ...
END.
```

Um nicht jedesmal den Paketnamen angeben zu müssen, können in der USE-Liste die Bezeichner des Pakets angegeben werden, die importiert werden sollen. Importierte Bezeichner können ohne Voranstellung des Paketnamens verwendet werden. Das Paket, aus dem die Bezeichner importiert werden, muß zuvor in einer WITH-Liste angegeben werden.

```
WITH complex_definitions;
FROM complex_definitions USE complex, add;
PROGRAM some_program;
VAR
   x, y, z : complex;
BEGIN
   ...
   z := Add (x,y);
   ...
END.
```

Die Verwendung der USE-Klausel ist praktisch, weil sie kürzere Bezeichner zuläßt, jedoch kann darunter die Übersichtlichkeit eines Programmes leiden. Die USE-Klausel verbietet nicht das Voranstellen des Paketnamens, um einen Bezeichner zu verwenden.

### Übersetzungen

Während der Kompilierung einer Übersetzungseinheit prüft der Compiler, ob die Größen aus den in den WITH-Listen angegebenen Paketen richtig verwendet werden, z. B. ob die Parameter für Unterprogramme richtig sind.

Bei der Übersetzung mehrerer Pakete muß eine bestimmte Kompilierungsreihenfolge eingehalten werden, die sich aus der in den WITH-Listen angegebenen Paket-Beziehungen ableiten läßt. Alle Paket-Spezifikationen der in den WITH-Listen einer Übersetzungseinheit angegebenen Pakete müssen vor der Übersetzungseinheit kompiliert werden.

Nach Änderungen einer Paket-Spezifikation müssen die zugehörige Paket-Implementierung und alle Übersetzungseinheiten, die diese Spezifikation benutzen, nochmals neu übersetzt werden.

# Anwendungen für Pakete

In einem Paket sollten nur logisch zusammengehörige Größen zusammengefaßt werden. Dabei lassen sich vier verschiedene Anwendungsbereiche unterscheiden:

- Sammlung von Deklarationen
  Exportiert Konstanten, Typen und Variablen
  Exportiert keine Unterprogramme
- Sammlung von Unterprogrammen
  Exportiert keine Konstanten, Typen und Variablen
  Exportiert Unterprogramme
- Abstrakte Datentypen
  Exportiert Konstanten und Typen
  Exportiert Unterprogramme
  Es wird keine Zustandsinformation im Paket gespeichert
- Automaten
  Exportiert Konstanten und Typen
  Exportiert Unterprogramme
  Zustandsinformationen werden im Paket gespeichert

Im folgenden werden diese vier Anwendungsbereiche näher erklärt.

### Sammlung von Deklarationen

Eine der einfachsten Anwendungen für Pakete ist das Zusammenfassen von Konstanten, Typen und Variablen. Vereinbarungen, die von verschiedenen Übersetzungseinheiten gebraucht werden, können in einem Paket gesammelt werden. Eventuelle Änderungen betreffen dann nur eine Übersetzungseinheit.

Beispielsweise können alle maschinenabhängigen Systemkonstanten eines Programms in einem Paket zusammengefaßt werden. Soll das Programm dann auf einen anderen Rechner portiert werden, so ist sichergestellt, daß diese Konstanten einfach für das ganze Programm verändert werden können.

```
PACKAGE system_constants;

CONST
   core_size       =   64536;
   printer_width   =      80;
   winchester      =   False;
   terminal_width  =      80;
   terminal_heigth =      25;

END.



PACKAGE BODY system_constants;
BEGIN
END.
```

Andererseits ist es oft nützlich, logisch verwandte Typen zusammenzufassen. Das folgende Paket zeigt, wie die Definitionen im Zusammenhang mit Datumsangaben aussehen könnten.

```
PACKAGE datums_definitionen;

TYPE
   tages_namen     = (sonntag,    montag,  dienstag, mittwoch,
                      donnerstag, freitag, samstag);
   tages_datum     = 1..31;
   monats_namen    = (januar,     februar, maerz,    april,
                      mai,        juni,    juli,     august,
                      september,  oktober, november, dezember);
   monats_datum    = 1..12;
   tages_tabelle   = ARRAY [monats_datum] of tages_datum;
   jahres_datum    = 0 .. Maxint;
   datum           = RECORD
                        tag   : tages_datum;
                        monat : monats_datum;
                        jahr  : jahres_datum;
                        END;
CONST
   max_tages_datum = tages_tabelle
                    (31, 28, 31, 30, 31, 30,
                     31, 31, 30, 31, 30, 31);

END.



PACKAGE BODY datums_definitionen;
BEGIN
END.
```

Pakete sollten klein gehalten werden, denn je größer Pakete werden, um so geringer ist die Übersichtlichkeit. Bei schwer lesbaren großen Spezifikationen sollte man sich daher überlegen, ob die Datenstrukturen nicht hierarchisch weiter gegliedert werden sollten oder ob der logische Zusammengang nicht zu weit gewählt wurde.

Desweiteren sollten Pakete nicht als Common-Bereiche wie zu FORTRANs-Zeiten verwendet werden. Werden Variablen in Spezifikationen angeboten, so erfolgt jeder Zugriff auf sie unkontrolliert. Der sichere Weg sind Zugriffsfunktionen auf Zustandsvariablen in der Paket-Implementierung (siehe auch 17.2.3). Dies ermöglicht einerseits die Überwachung des Zugriffs. Anderseits kann die Darstellung der Daten geändert werden, ohne daß andere Übersetzungseinheiten davon betroffen sind. Dies ist wahrscheinlich der wesentlich wichtigere Gesichtspunkt, der vielmehr die Mächtigkeit der Pakete gegenüber Sprachen früherer Generationen bewußt werden läßt.

## Sammlung von Unterprogrammen

Analog zum Zusammenfassen von Konstanten, Typen und Variablen ist es auch möglich, verschiedene Unterprogramme durch ein Paket zu einer Einheit zu verbinden. Zum Beispiel können die hyperbolischen Funktionen als Unterprogramme in einem Paket zusammengefaßt werden:

```
PACKAGE hyperbolic_functions;

FUNCTION sinh (x : Real) : Real;
FUNCTION cosh (x : Real) : Real;
FUNCTION tanh (x : Real) : Real;

END.
```

Die in dieser Spezifikation angegebenen Unterprogramm-Köpfe müssen in der Paket-Implementierung durch Unterprogramm-Blöcke vervollständigt werden. Das folgende Beispiel zeigt eine Möglichkeit der Realisierung.

```
PACKAGE BODY hyperbolic_functions;

FUNCTION sinh (x : Real) : Real;
BEGIN
   sinh := (Exp (x) + Exp (-x)) / 2.0
END;

FUNCTION cosh (x : Real) : Real;
BEGIN ... END;

FUNCTION tanh (x : Real) : Real;
BEGIN ... END;

BEGIN
END.
```

### Abstrakter Datentyp

Unter einem abstrakten Datentyp versteht man die Vereinigung von Typ und Operationen, die auf Werten dieses Typs erlaubt sind. Mit Hilfe von Paketen können abstrakte Datentypen definiert werden. Aus Gründen der Übersichtlichkeit ist es empfehlenswert, für jeden abstrakten Datentyp ein eigenes Paket anzulegen. Um eine logische Abstraktion zu erzwingen wird der Typ am besten als sogenannter ’privater Typ’ vereinbart. Dies ist allerdings nur bei Zeigertypen möglich.

Unter einem privaten Zeigertyp versteht man einen Typ, dessen Typ-Name nach außen bekannt ist, aber dessen Domänentyp außerhalb des Pakets nicht sichtbar ist. Damit sind Zugriffe auf Werte des Typs nur über die definierten Zugriffsfunktionen möglich. Ein privater Zeigertyp wird in der Spezifikation als Zeiger mit einem Domänen-Typ definiert, der erst in der Paket-Implementierung definiert wird.

In Kapitel 11.4 ist am Beispiel einer Warteschlange die Realisierung eines abstrakten Datentyps aufgezeigt.

## Automat

Ein Automat wird durch eine Menge von Zuständen und Zustandsübergängen beschrieben. Zu jedem Zeitpunkt befindet sich der Automat in genau einem Zustand. Bei der Realisierung eines Automaten durch ein Paket wird der jeweilige Zustand des Automaten durch die augenblicklichen Werte der in der Paket-Implementierung vereinbarten Variablen repräsentiert. Die Zustandsinformation ist für einen Benutzer des Automaten nicht zugänglich. Der Automat kann durch Ausführen von Operationen (Prozeduren) seinen Zustand ändern. Mit Hilfe von Funktionen kann der Anwender des Automaten Informationen über den augenblicklichen Zustand abfragen.

Im allgemeinen exportiert ein Automat keine Konstanten, Typen oder Variablen. Das Aussehen gleicht also einer Unterprogrammsammlung. Der wesentliche Unterschied liegt in den Zustandsinformationen, die im Paket gespeichert sind. Demgegenüber sind im Paket hyperbolic_functions die Unterprogramme unabhängig voneinander, da das Ergebnis einer Funktion nicht von anderen Funktionsaufrufen abhängt. Bei einem Automaten hängt das Ergebnis eines Unterprogrammaufrufs von allen vorausgehenden Aufrufen ab.

```
PACKAGE integer_stack;

CONST
   full  = 900; { Exception  bei Stackueberlauf }
   empty = 901; { Exception, bei Zugriff auf leeren Stack }

PROCEDURE push (i : Integer);
   { Ein Integer-Wert wird auf den Stack geschrieben }

PROCEDURE pop  (VAR i : Integer);
   { Ein Integer-Wert wird vom Stack genommen }

FUNCTION  is_empty : Boolean;
   { Sind noch Integer-Werte auf dem Stack }

PROCEDURE clear;
   { Bringt den Stack in den Grundzustand }

END.
```

Das Paket kann beispielsweise zur Realisierung eines Taschenrechners benutzt werden. Die eingelesenen Werte werden auf den Stack geschrieben. Für diesen Zweck wird die Prozedur push angeboten. Mit Hilfe von pop können Werte wieder vom Stack genommen werden. Die beiden Konstanten empty und full definieren die möglichen Ausnahmesituationen des Automaten. Mit der Prozedur clear wird der Automat in den Grundzustand versetzt.

```
PACKAGE BODY integer_stack;

CONST
   stack_size  = 100;

TYPE
   stack_array = ARRAY [1..stacksize] OF Integer;

VAR
   stack      : stack_array;
   stack_index: 0..stack_size;

PROCEDURE push (i : Integer);
   { Ein Wert wird auf den Stack geschrieben }
BEGIN
   IF stack_index = stack_size THEN
      Raise (full);
   stack_index         := stack_index + 1;
   stack [stack_index] := i;
END { push };

PROCEDURE pop  (VAR i : Integer);
   { Ein Wert wird vom Stack genommen }
BEGIN
   IF is_empty THEN
      Raise (empty);
   i           := stack [stack_index];
   stack_index := stack_index - 1;
END { pop };

FUNCTION  is_empty : Boolean;
   { Sind noch Werte auf dem Stack ? }
BEGIN
   is_empty := stack_index = 0;
END { is_empty };

PROCEDURE clear;
   { Bringt den Stack in den Grundzustand }
BEGIN
   stack_index := 0
END { clear };

BEGIN
   clear;
END { integer_stack }.
```

Die Größen stack und stack_index bilden die interne Zustandsinformation. Da sie global im Paket deklariert sind, existieren sie während der gesamten Ausführungszeit des Programms. Auf diese Daten kann nur durch die sichtbaren Prozeduren push, pop und clear und die Funktion is_empty zugegriffen werden.

Der Stack wurde durch einen Array realisiert. Dadurch ist die Anzahl der Werte, die auf den Stack geschrieben werden können, begrenzt. Soll für eine erweiterte Implementierung die Darstellung des Stacks geändert werden, z.B. in eine verkettete Liste, so hat dies keinen Einfluß auf Übersetzungseinheiten, die das Paket stack verwenden.

# Ausnahmebehandlungskonzept

Ausnahmen sind Sonderfälle eines Algorithmus, die beim normalen Ablauf des Algorithmus nicht auftreten. Ihre Berücksichtigung trägt nicht zum Verständnis des Algorithmus bei, sondern im Gegenteil, sie würde das Verständnis des Algorithmus nur erschweren. Beispielsweise können beim Rechnen mit ganzen Zahlen die Zahlenwerte zu groß werden, so daß sie in der Maschine nicht mehr darstellbar sind (z. B. nicht mehr in dem entsprechenden Integer-Typ liegen). Wenn dieser Sonderfall (arithmetischer Überlauf) in einem numerischen Algorithmus berücksichtigt werden sollte, so müßte vor jeder Formel eine umständliche Abfrage mit den Werten der in der Formel verwendeten Operanden stattfinden, um zu überprüfen, ob Zwischen- oder Endergebnis bei eventueller Auswertung der Formel zu groß werden würden. Diese Abfragen würden sicherlich den numerischen Algorithmus völlig unverständlich machen. Deshalb möchte man den eigentlichen Algorithmus erst einmal ohne Berücksichtigung von derartigen Sonderfällen (Ausnahmen) hinschreiben und in einem zweiten Teil (dem Ausnahmebehandlungsteil) Maßnahmen zur Behandlung der Ausnahmen vorsehen; im Falle des arithmetischen Überlaufs würde man als Maßnahme zum Beispiel eine Meldung ausgeben, daß die Zahlenwerte zu groß werden und deshalb die Rechnung nicht mit der Zahlendarstellung der vorliegenden Maschine durchgeführt werden kann. Als Folge würde man das Programm nach Ausgabe dieser Meldung abbrechen, da eine sinnvolle Weiterarbeit nicht möglich ist.

Im Gegensatz dazu sind Sonderfälle eines Algorithmus, die auch im normalen Ablauf des Algorithmus auftreten können oder deren Berücksichtigung zum Verständnis des Algorithmus beiträgt, keine Ausnahmen in dem oben geschilderten Sinn. Es kann daher durchaus möglich sein, daß ein und dasselbe Ereignis in einem Algorithmus als Ausnahme, in einem anderen Algorithmus aber als Normalfall betrachtet werden muß. Als Beispiel sei das Erreichen des Dateiendes beim Lesen von Eingabedaten genannt. Soll eine Datei komponentenweise eingelesen und bearbeitet werden, so ist das Erreichen des Dateiendes ein erwartetes Ereignis, das zur normalen Beendigung des Algorithmus führen soll.

*Beispiel*

```
PROGRAM process (data);
TYPE
   t = ... ;
VAR
   data: FILE OF t;
   elem: t;
BEGIN
   WHILE NOT Eof (data) DO BEGIN
      Read (data, elem);
      { verarbeiten }
      END;
END.
```

Sollen jedoch von einer Textdatei drei Zahlen eingelesen werden, so könnte man das frühzeitige Erreichen des Dateiendes als eine Ausnahme in obigem Sinne auffassen.

*Beispiel*

```
PROGRAM example (datei, Output);

VAR datei: Text;
    zahl1,
    zahl2,
    zahl3: Integer;

BEGIN
   Reset (datei);
   Read  (datei, zahl1, zahl2, zahl3);
   ...
EXCEPTION
   IF Error_Number = Eof_Error THEN
      Writeln ('Fehler in Datei')
   ELSE
      Writeln ('Sonstiger Fehler')
END.
```

In Pascal-XT sind Ausnahmen durch ganze Zahlen vom Typ Integer codiert. Negative Zahlen sind vordefinierten Ausnahmen vorbehalten, positive Zahlen können für benutzerdefinierte Ausnahmen verwendet werden. Für die vordefinierten Ausnahmen sind Konstanten vereinbart (siehe 14.1).

### Programmierte Ausnahmebehandlung

Bei Eintreten einer Ausnahme (eines Fehlers) ist es nicht sinnvoll, die Ausführung des Programms einfach fortzusetzen und so zu tun, als wäre nichts geschehen. Stattdessen gibt es je nach Situation folgende sinnvolle Maßnahmen:

- Abbruch des gesamten Programms mit einer Fehlermeldung und gegebenenfalls notwendigen Endebehandlungen, um Datenbestände auf Dateien in einem konsistenten Zustand zu hinterlassen.
- Abbruch des Unterprogramms, in dem die Ausnahme aufgetreten ist. Auch hier ist eventuell eine vorherige Abschlußbehandlung notwendig, um globale Datenstrukturen in einem konsistenten Zustand zu hinterlassen, so daß eine Fortsetzung des Programms hinter dem Aufruf des abgebrochenen Unterprogramms möglich und sinnvoll ist.
- Ausführung eines alternativen Algorithmus, der mit mehr Aufwand dasselbe leistet, wie der durch Auftreten der Ausnahme unterbrochene Algorithmus. Bei Speicherüberlauf (Memory_Error) könnten zum Beispiel weitere Daten auf eine Datei geschrieben statt im Hauptspeicher abgelegt werden.
- Wiederholung der Operation, die zu der Ausnahmesituation führte. Eine Wiederholung ist natürlich nur dann sinnvoll, wenn damit gerechnet werden kann, daß beim zweiten Versuch die Ausnahmesituation nicht mehr eintritt. Wenn zum Beispiel beim Versuch, eine Datei zu eröffnen (mit Reset oder Rewrite) ein Eröffnungsfehler (Open_Error) eintritt, weil die Datei gerade von einem anderen Programm eröffnet ist, so ist ein zweiter Versuch nach einer Pause sinnvoll, da dann die Datei inzwischen wieder frei (d.h. geschlossen) sein kann. In diesem Fall kann man also davon ausgehen, daß sich die Fehlerursache von allein nach einiger Zeit beseitigt. Im allgemeinen ist jedoch eine Wiederholung der fehlerverursachenden Operation nur dann sinnvoll, wenn vorher die Fehlerursache beseitigt worden ist.

Zur Programmierung all dieser möglichen Reaktionen steht in Pascal-XT das Konzept der Ausnahmebehandlung zur Verfügung. Ein Ausnahmebehandler kann für ein Unterprogramm oder für eine Verbundanweisung angegeben werden (siehe Kapitel 14). Tritt eine Ausnahme ein, so wird die Ausführung des Programms beim dynamisch letzten Ausnahmebehandler fortgesetzt. Dazu wird innerhalb des gerade aktiven Unter- bzw. Hauptprogramms die innerste Verbundanweisung mit Ausnahmebehandler gesucht, die die Anweisung enthält, die die Ausnahme erzeugt hat. Gibt es keine solche Verbundanweisung in dem gerade aktiven Unterprogramm, so wird die Suche auf gleiche Weise im dynamischen Vorgänger, der dieses Unterprogramm aufgerufen hat, fortgesetzt. Es wird dort nach der innersten Verbundanweisung mit Ausnahmebehandler gesucht, die den Unterprogrammaufruf enthält. Die Suche wird so lange fortgesetzt, bis entweder ein Ausnahmebehandler gefunden wird oder, falls keiner gefunden wird, das Programm mit einer Fehlermeldung des Laufzeitsystems abgebrochen wird. Betrachten wir nun die möglichen Reaktionen auf eine Ausnahme im einzelnen:

- **Abbruch eines Programms oder Unterprogramms**

Der Ausnahmebehandler ist im Block des Unter- bzw. Haupt-Programms angegeben. Er hat die Aufgabe, Abschlußbehandlungen durchzuführen. Anschließend wird das gesamte Programm (Unterprogramm) abgebrochen.

*Beispiel*

```
PROGRAM bsp (old_data, new_data, transaction, Output);
VAR
   old_data,
   new_data,
   transaction : Text;

BEGIN
   Reset   (old_data);
   Reset   (transaction);
   Rewrite (new_data);
   ...
   { ausfuehren der Transaktionen }
   ...
EXCEPTION
   { new_data koennen unvollstaendig sein, d.h. loeschen }
   Rewrite (new_data);
   Writeln (Output, 'Programm abgebrochen');
END
```

- **Ausführung eines alternativen Algorithmus**

Die folgende Prozedur soll einen Eröffnungszug in einer Bibliothek suchen. Kann diese Bibliothek nicht geöffnet werden, dann wird mit der normalen heuristischen Suche ein geeigneter Eröffnungszug gesucht.

*Beispiel*

```
PROCEDURE eroeffnungszug;
BEGIN
   Reset (eroeffnungsbibliothek);
   suche_zug;     { Ausnahme 13, wenn nicht vorhanden }
EXCEPTION
   IF (Error_Number = Open_Error) OR
      (Error_Number = 13)         THEN
      eigener_zug
   ELSE
      Raise (0);
END
```

Dieses Beispiel zeigt auch, wie man alle Ausnahmen, die man nicht erwartet oder nicht behandeln kann, propagiert. Dies geschieht durch Aufruf von Raise für alle nicht erwarteten Fehlernummern (siehe 15.11).

- **Wiederholung der fehlerverursachenden Operation**

Die im Fehlerfall (Ausnahmefall) zu wiederholende Operation wird in eine Verbundanweisung mit Ausnahmebehandler gepackt. Diese Verbundanweisung wiederum ist der Rumpf der Wiederholschleife. Das folgende Beispiel zeigt, wie das Eröffnen einer Datei dreimal versucht werden kann, ehe mit Fehlermeldung das Programm abgebrochen wird. Nach dem erfolgreichen Eröffnen der Datei wird die Schleife mit der EXIT-Anweisung (10.1.5) verlassen.

*Beispiel*

```
FOR i := 1 TO 3 DO
   BEGIN
      Reset (datei);
      EXIT;
   EXCEPTION
      IF (i = 3) OR (Error_Number <> Open_Error) THEN
         Raise (13);
   END;
```

Dieses Beispiel zeigt, wie man benutzerdefinierte Ausnahmen erzeugen kann: nämlich durch Aufruf von Raise (siehe 15.11) mit einer Fehlernummer > 0. Im Beispiel wird bei Auftreten eines anderen Fehlers als Open_Error oder bei der dritten Wiederholung die Ausnahme mit der benutzerdefinierten Fehlernummer 13 erzeugt. Da Raise (13) innerhalb des Ausnahmebehandlers steht, wird die Ausnahme gleich an einen übergeordneten Ausnahmebehandler propagiert (siehe 14.3). Mit Raise (13) wird auch die FOR-Schleife verlassen.

In diesem Zusammenhang ist jedoch zu erwähnen, daß Raise nicht als eine besondere Form von GOTO anzugeben ist. Raise sollte nur dem Erzeugen wirklicher Ausnahmesituationen dienen! In diesem Sinne ist Raise (13) in diesem Beispiel gerechtfertigt.

*Querverweise*

| | |
|---|---|
| Verbundanweisung: | 10.2, 14.2 |
| Ausnahme: | 14 |
| Ausnahmebehandlungsteil: | 14.2 |
| Ausnahmebehandlung: | 14.3 |
| Error Number, Raise: | 15.11 |

# Ein-/Ausgabe

Erfahrungsgemäß haben weniger erfahrene Pascal-Anwender Probleme mit der Ein-/Ausgabe. Aus diesem Grund werden die Konzepte nochmals kurz dargestellt und die Problempunkte etwas ausführlicher behandelt. Die genaue Beschreibung der benötigten Prozeduren und Funktionen befindet sich im Abschnitt 15.1.

Die Pascal-Norm befaßt sich nur mit der Dateibearbeitung auf der Programmierebene. Die externe Darstellung einer Datei (sowohl Ein-/Ausgabe-Geräte als auch Datenträger-Medien werden in diesem Zusammenhang als Dateien aufgefaßt) kann angesichts der kaum überschaubaren Vielzahl von Betriebssystemen, Datenaufzeichnungsverfahren etc. nicht Gegenstand der Normung einer Programmiersprache sein. Zur Unterscheidung zwischen externen Dateien und ihren internen (programmierbaren) Darstellungen haben sich die Begriffe "logische Datei" und "physikalische Datei" eingebürgert.

**Implementierungsdefinierte Eigenschaft**

In Pascal werden die logischen Auswirkungen der Dateioperationen beschrieben. Die physikalischen Aktivitäten und der Zeitpunkt ihrer Ausführung sind implementierungs-definiert.

In allen folgenden Darstellungen dieses Kapitels wird eine logische Datei in der folgenden Form dargestellt:

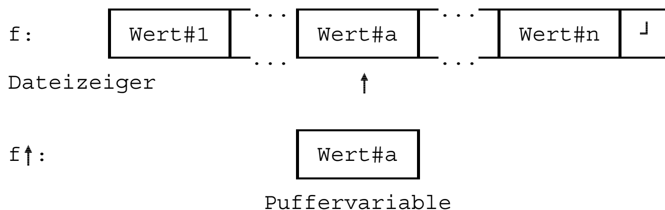



Bild 19-1: Darstellung einer logischen Datei

Sei f eine Variable eines Typs FILE OF t, dann sind Werte von f Folgen von Werten des Komponententyps t, auf die lesend oder schreibend zugegriffen werden kann. Die aktuelle Position, auch als Dateizeiger bezeichnet (dargestellt als ↑), beschreibt die aktuelle Lese- bzw. Schreibposition. Zu jeder FILE-Variablen gibt es implizit eine Puffervariable f↑ vom Typ t, die einen Wert aus der Folge aufnehmen kann. Der Zugriff auf Werte der Datei ist nur über die Puffervariable möglich.

Das Ende der Folge wird in allen Bildern durch das Zeichen ⊥ dargestellt. Dieses Zeichen dient lediglich der Veranschaulichung und sagt nichts über irgendwelche Implementierungen aus.

Der Transport der Daten zwischen Puffervariable und physikalischer Datei wird durch die vordefinierten Prozeduren Get, Read, Put, Write, Reset und Rewrite beschrieben.

Norm-Pascal kennt nur sequentielle Dateien. Pascal-XT Implementierungen können darüber hinaus vordefinierte Pakete anbieten, um z. B. Direktzugriffsdateien bzw. indexsequentielle Dateien zu unterstützen.

Das Ende einer Datei kann mit der vordefinierten Funktion Eof festgestellt werden. Sie liefert den booleschen Wert True, wenn das Ende der Datei f (End of File) erreicht ist, ansonsten den booleschen Wert False. Die Anwendung der Funktion ist nur bei zum Lesen eröffneten Dateien sinnvoll, da man sich beim Schreiben ohnehin immer am Dateiende befindet. Ist das Ende der Datei erreicht, dann verursacht jedes weitere Lesen von Werten einen Laufzeitfehler.

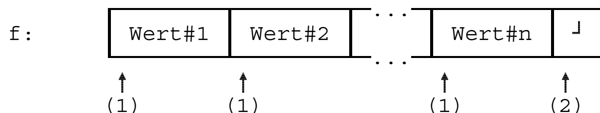



Bild 19-2: Anwendung der vordefinierten Funktion Eof

Befindet man sich beim Lesen an den mit (1) gekennzeichneten Stellen, dann ist Eof(f) False. An der mit (2) markierten Stelle ist Eof(f) True.

Die Dateien in Pascal werden in zwei Hauptklassen unterteilt:

- **Lokale Dateien**

  Eine Datei wird als lokale Datei bezeichnet, wenn der Bezeichner der FILE-Variablen nicht als Programmparameter in einem Hauptprogramm oder Paket angegeben wird. Die Zuordnung zu einer physikalischen Datei während des Ablaufs eines Programms geschieht implizit durch einen implementierungsabhängigen Mechanismus. In Pascal-XT kann mittels Assignfile einer lokalen Datei auch eine physikalische Datei zugeordnet werden.

Die Lebensdauer einer lokalen Datei ist an die des Blocks gebunden, in dem sie als Variable deklariert wurde:

- In einem Hauptprogramm oder Paket deklarierte lokale Dateien existieren während der gesamten Ausführung des Programms.
- In einem Unterprogramm deklarierte lokale Dateien werden beim Aufruf des Unterprogramms erzeugt und beim Verlassen wieder vernichtet.
- Dynamische Variable oder Komponenten dynamischer Variablen können von einem FILE-Typ sein. Ihre Lebensdauer entspricht dann dieser (mit New erzeugten) Variablen.

Die einer lokalen Datei zugeordnete physikalische Datei ist nach dem Verlassen des Blocks ebenfalls nicht mehr vorhanden.

- **Externe Dateien**

  Externe Dateien sind immer unmittelbar im Hauptprogramm-Block oder in einem Paket-Block vereinbart. Sie werden durch Angabe ihrer Variablen-Bezeichner in der Programmparameterliste als externe, also unabhängig vom Programm existierende Dateien angemeldet. Diesen Variablen-Bezeichnern müssen zur Ausführungszeit des Programms physikalische Dateien zugeordnet werden. Externe Dateien können keine Komponenten von Variablen und keine dynamischen Variablen sein.

Zur Bearbeitung einer Datei ist folgendes notwendig:

1) Deklaration einer FILE-Variablen vom gewünschten FILE-Typ
   - als lokale oder globale Variable, oder
   - als dynamische Variable oder Komponente davon
2) Die Bindung an eine physikalische Datei bei externen Dateien
3) Die vordefinierten Textdateien Input und Output müssen in der Programmparameterliste angegeben werden, wenn sie im Programm verwendet werden. Ihnen ist standardmäßig die Datensichtstation zugeordnet.

*Querverweise*


File-Typp: 6.3.5

# Zuordnung einer physikalischen Datei

Die Zuordnung einer physikalischen Datei zu einer FILE-Variablen kann auf zwei Arten erfolgen: durch Angabe des Variablen-Bezeichners in der Programmparameterliste oder mittels der vordefinierten Prozedur Assignfile.

- **Angabe in der Programmparameterliste**

Die Zuordnung einer physikalischen Datei zu einer als Programmparameter angegeben FILE-Variablen erfolgt durch einen implementierungsdefinierten Mechanismus.

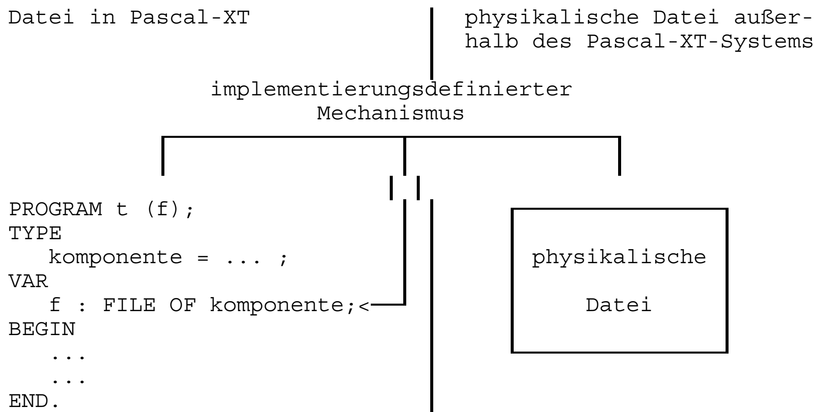



Bild 19-3a: Zuordnung durch implementierungsdefinierten Mechanismus

- **Zuordnung über Assignfile**

Mit der vordefinierten Prozedur Assignfile kann sowohl einer lokalen als auch einer externen Datei eine physikalische Datei zugeordnet werden. Die Art der Beschreibung der physikalischen Datei im zweiten Parameter von Assignfile ist implementierungsdefiniert.

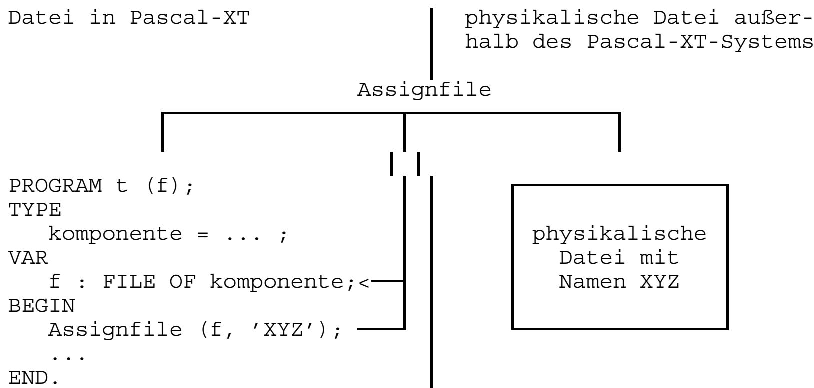



Bild 19-3b: Zuordnung einer externen Datei durch Assignfile

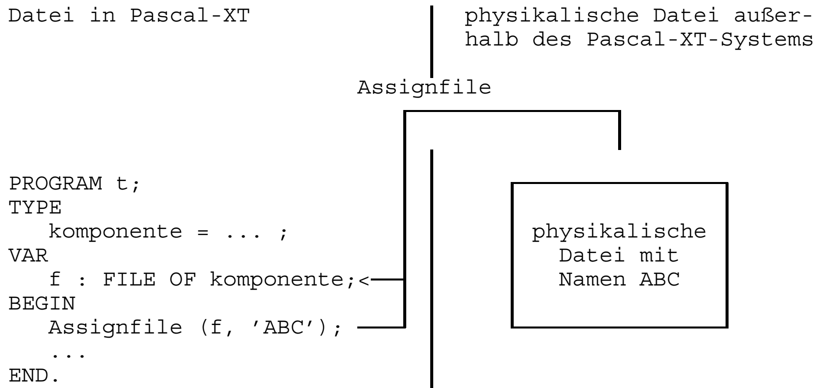



Bild 19-3c: Zuordnung einer lokalen Datei durch Assignfile

# Eröffnen von Dateien zum Lesen oder Schreiben

Dateien müssen vor ihrer Bearbeitung zum Lesen oder Schreiben mit Reset bzw. Rewrite eröffnet werden.

### Reset (f)

Reset (f) eröffnet die Datei f zum Lesen und überträgt bereits die erste Komponente der physikalischen Datei in die Puffervariable. Der Dateizeiger steht auf der ersten Komponente der Datei. Ist die Datei leer, dann isf Eof (f) True und die Puffervariable f↑ ist undefiniert.

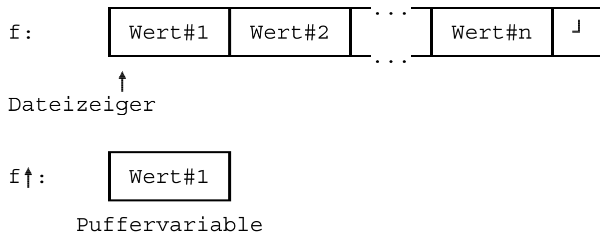



Bild 19-4: Eröffnen einer nicht leeren Datei zum Lesen

### Rewrite (f)

Rewrite eröffnet die Datei f zum Schreiben. Nach Rewrite (f) ist die Datei leer, ein etwaiger alter Dateiinhalt ist verloren. Die Puffervariable f↑ besitzt keinen definierten Wert und Eof (f) hat den Wert True.

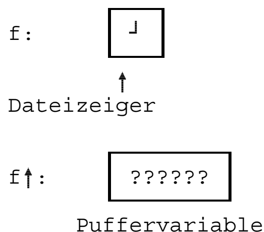



Bild 19-5: Eröffnen einer Datei zum Schreiben

# Die Lese-/Schreibprozeduren Get und Put

Die vordefinierten Prozeduren Get und Put sind die Grundfunktionen, um Werte aus der physikalischen Datei in die Puffervariable bzw. von dort in die Datei zu übertragen.

**Get (f)**

Get (f) rückt den Dateizeiger um eine Position weiter. Existiert ein weiteres Element in der Datei, dann wird dieses in die Puffervariable f↑ übernommen und Eof (f) liefert den Wert False. Existiert kein weiteres Element, dann ist die Puffervariable f↑ undefiniert und Eof (f) ist True.

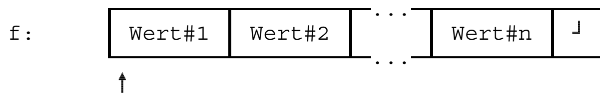


Bild 19-6a: Zustand nach Reset (f)

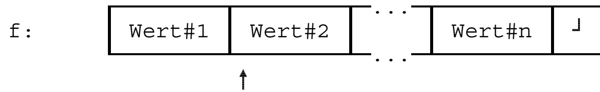


Bild 19-6b: Zustand nach dem ersten Get (f)

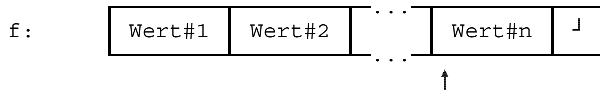


Bild 19-6c: Zustand nach dem (n-1)-ten Get (f)

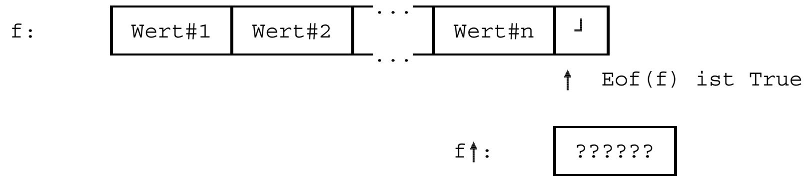



Bild 19-6d: Zustand nach dem n-ten Get (f) (Dateiende)

**Put (f)**

Mit Put (f) wird der aktuelle Wert der Puffervariablen f↑ am Ende der mit Rewrite eröffneten Datei angehängt. Vor dem Aufruf von Put muß also der Puffervariablen ein Wert zugewiesen werden. Nach dem Aufruf von Put ist der Inhalt der Puffervariablen undefiniert.

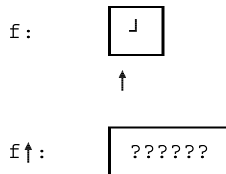



Bild 19-7a: Zustand nach Rewrite (f)

```
f↑ := Wert#1;
Put (f);
```

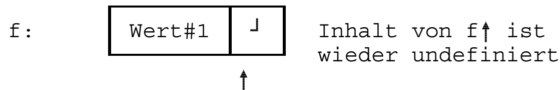



Bild 19-7b: Zustand nach Put (f)

```
f↑ := Wert#2;
Put (f);
```

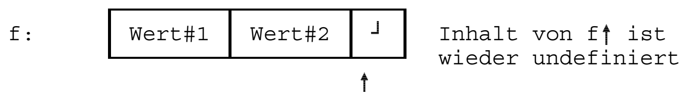



Bild 19-7c: Zustand nach dem zweiten Put (f)

# Die Lese-/Schreibprozeduren Read und Write

Da man vorwiegend Daten aus Variablen des Programms in Dateien und umgekehrt übertragen will, ist der fortwährende Umweg über die Puffervariable lästig. Deshalb sind zur Abkürzung von Lese- und Schreiboperationen die vordefinierten Prozeduren Read und Write definiert.

**Read (f, v1, .., vn)**

In der Read-Parameterliste kann eine beliebige Anzahl von Leseparametern vi, mindestens aber einer, angegeben werden. Eine solche Read-Anweisung wird dann als Folge von einzelnen Read-Anweisungen mit jeweils nur einem Leseparameter aufgefaßt. Read (f, v) steht abkürzend für folgende Anweisungsfolge:

```
BEGIN v := f↑; Get (f); END
```

Mit Read wird also die aktuelle Komponente aus der Puffervariablen in die Variable übernommen und die nächste Komponente aus der Datei in die Puffervariable übertragen. Die folgenden Befehlssequenzen zeigen die Unterschiede bei der Programmierung von Schleifen, abhängig davon ob Get oder Read verwendet wird.

```
Reset (f);                              | Reset (f);
WHILE NOT Eof (f) DO BEGIN              | WHILE NOT Eof (f) DO BEGIN
   { verarbeite f↑ }                    |    Read (f, v);
   Get (f);                             |    { verarbeite v }
   END;                                 |    END;
```

Die folgenden Bilder zeigen verschiedene Zustände beim Lesen von Komponenten aus einer Datei.

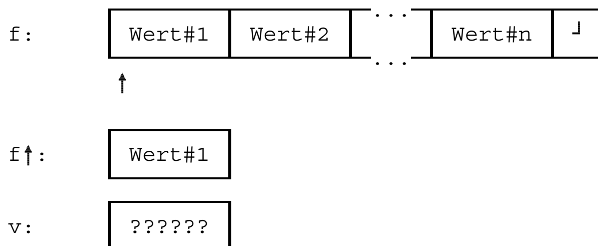



Bild 19-8a: Zustand nach Reset (f)

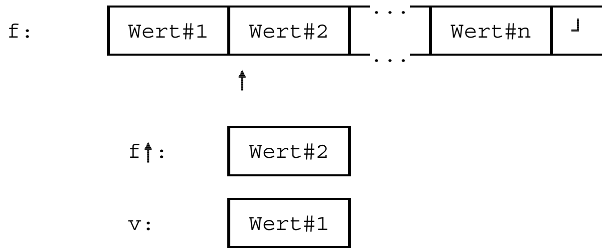



Bild 19-8b: Zustand nach dem ersten Read (f,v)

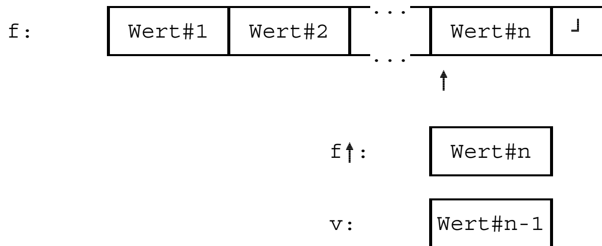



Bild 19-8c: Zustand nach dem (n-1)-ten Read (f,v)

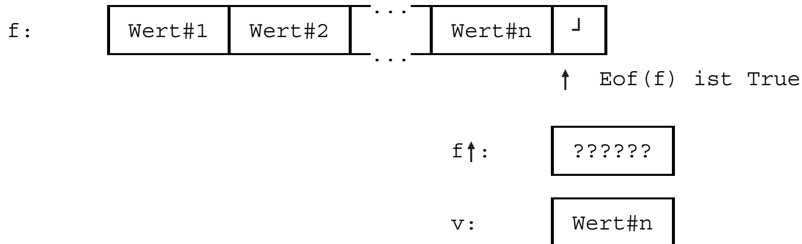



Bild 19-8d: Zustand nach dem n-ten Read (f,v) (Dateiende)

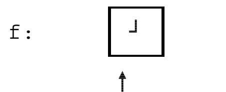
### Write (f, a1, ..., an)

In der Write-Parameterliste kann eine beliebige Anzahl von Parametern ai, mindestens aber einer, angegeben werden. Eine solche Write-Anweisung wird dann als Folge von einzelnen Write-Anweisungen mit jeweils nur einem Parameter aufgefaßt. Jedes Write (f, a) steht abkürzend für folgende Anweisungsfolge:

```
BEGIN f↑ := a; Put (f); END
```

Ist die FILE-Variable f nicht vom vordefinierten Typ Text, dann müssen die Ausdrücke ai zuweisungsverträglich zum Komponententyp des FILE-Typs sein. Ist f eine Textdatei, dann werden die ai als Schreibparameter bezeichnet und können die in Kapitel 19.5 beschriebenen Typen besitzen.

Die folgenden Bilder zeigen verschiedene Zustände beim Schreiben von Komponenten in eine Datei.



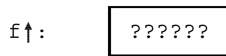
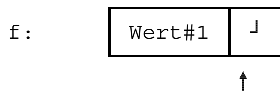




Bild 19-9a: Zustand nach Rewrite (f)



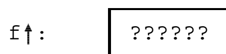
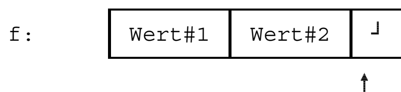




Bild 19-9b: Zustand nach Write(f, Wert1)



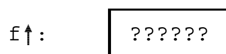




Bild 19-9c: Zustand nach Write(f, Wert2)

Im folgenden sind äquivalente Beispiele zum Kopieren einer Datei mit Realzahlen dargestellt. Bei der Lösung mit Get und Put wird eine Komponente von der Puffervariablen der Eingabedatei sofort in die Puffervariable der Ausgabedatei kopiert. Bei der Lösung mit Read erfolgt der Umweg über eine Hilfsvariable.

```
PROGRAM kopiere(f,g);

VAR
   f, g: FILE OF Real;


BEGIN
   Reset(f);
   Rewrite(g);
   WHILE NOT Eof(f) DO BEGIN
      g↑ := f↑;
      Put(g); Get(f);
      END
END.
```

Kopieren einer Datei mit Get und Put

```
PROGRAM kopiere(f,g);

VAR
   f, g: FILE OF Real;
   r   : Real;

BEGIN
   Reset(f);
   Rewrite(g);
   WHILE NOT Eof(f) DO BEGIN
      Read(f,r);
      Write(g,r);
      END
END.
```

Kopieren einer Datei mit Read und Write

# Textdateien

Dateien vom vordefinierten Typ Text, kurz Textdateien genannt, haben eine zusätzliche Eigenschaft: Sie beschreiben eine Folge von Zeilen, wobei jede Zeile aus einer Folge von Zeichen vom Typ Char besteht. Jede Zeile wird durch ein spezielles Zeichen, der Zeilenende-Komponente abgeschlossen. Diese Komponente wird innerhalb eines Pascal-Programms wie ein Leerzeichen behandelt. Eine Textdatei kann immer nur vollständige Zeilen enthalten, wie im Bild 19-10 dargestellt.

Textdateien dürfen nicht mit Dateien vom Typ FILE OF Char verwechselt werden, die ebenfalls aus einer Folge von Zeichen bestehen, aber keine Zeilenstruktur besitzen. Auf sie können die zusätzlich für Textdateien definierten Unterprogramme nicht angewandt werden. Die Puffervariable für beide Dateien ist vom Typ Char.

Die Bilder 19-10 und 19-11 zeigen nochmals den Unterschied zwischen den beiden Dateiarten.

In allen folgenden Beispielen wird die Zeilenende-Komponente durch das Zeichen "•" und das Ende der Datei durch '┘' dargestellt. Leerzeichen werden zur besseren Sichtbarkeit durch '_' dargestellt.

```
VAR f1: Text;
    f2: FILE OF Char;
```

f1: | Zeile1• | Zeile2• | ┘ |

Bild 19-10: Struktur einer Textdatei

f2: | Z | e | i | l | e | 1 | Z | e | i | l | e | 2 | ┘ |

Bild 19-11: Struktur einer Datei vom Typ FILE OF Char

Für Textdateien gibt es zusätzliche vordefinierte Unterprogramme:

- `Readln`
- `Writeln`
- `Eoln`
- `Page`

Readln und Writeln dienen zur Behandlung der Zeilenende-Komponente. Mit Writeln wird sie erzeugt und mit Readln kann sie überlesen werden. Mit Eoln kann abgefragt werden, ob das Ende einer Zeile vorliegt. Eoln liefert den Wert True, wenn der Dateizeiger auf die Zeilenende-Komponente zeigt (die Puffervariable enthält dann ein Leerzeichen), ansonsten den Wert False. Bild 19-12 zeigt den Wert von Eoln an verschiedenen Positionen des Dateizeigers.

Bild 19-12: Anwenden der vordefinierten Funktion Eoln

Befindet man sich beim Lesen an den mit (1) gekennzeichneten Stellen, dann ist Eoln(f1) False. An den mit (2) markierten Stellen ist Eoln(f1) True. An der Stelle (3) darf Eoln(f1) nicht aufgerufen werden, weil bereits Eof(f1) True ist.

Die Prozedur Page dient zur Seitenformatierung von Textdateien, die auf einem Drucker ausgegeben werden. Text, der nach einem Aufruf von Page in die Datei geschrieben wird, wird beim Drucken auf einer neuen Seite ausgegeben.

Input und Output sind vordefinierte File-Variablen vom Typ Text mit besonderen Eigenschaften. Sie müssen in einem Programm nicht deklariert werden. Sobald sie in der Programmparameterliste angegeben werden, sind sie im Programm sichtbar und werden beim Start des Programms automatisch zum Lesen bzw. Schreiben eröffnet. Den beiden Dateien ist standardmäßig die Datensichtstation zugeordnet.

Im Unterschied zu Nicht-Textdateien können die Parameter der vordefinierten Prozeduren Read, Readln, Write und Writeln folgende Typen besitzen:

- den Typ Char
- Integer- und Real-Typen
- Zeichenkettentypen fester und variabler Länge
- bei Write bzw. Writeln zusätzlich der Typ Boolean.

Bei der Ausgabe von Zahlen mit Write bzw. Writeln erfolgt automatisch eine Konvertierung von der internen Darstellung in abdruckbare Zeichen. Beim Lesen von Zahlen mit Read bzw. Readln erfolgt die umgekehrte Konvertierung.

Bei der Ausgabe boolescher Werte werden die Zeichenketten ’TRUE’ bzw. ’FALSE’ (die Schreibweise ist implementierungsdefiniert) ausgegeben.

In den Unterprogrammaufrufen für Textdateien kann die FILE-Variable weggelassen werden. In diesem Fall bezieht sich

- Readln, Read, Eoln und Eof auf die vordefinierte Textdatei Input.
- Writeln, Write und Page auf die vordefinierte Textdatei Output.

## Lesen von einer Textdatei

**Read (f, v1, ..., vn)**
**Readln (f, v1, ..., vn)**

Die Prozedur Read liest aus der aktuellen Zeile der Textdatei f Werte ein und überträgt sie auf die Leseparameter v1 bis vn. Readln wirkt wie die Prozedur Read mit der zusätzlichen Eigenschaft, daß **nach** dem Einlesen der Leseparameter v1 bis vn die restlichen Zeichen der aktuellen Zeile, sofern vorhanden, überlesen werden, und das erste Zeichen der nächsten Zeile in die Puffervariable übernommen wird.

Jeder Leseparameter vi muß eine verallgemeinerte Variable mit einem der Typen Char, einem Integer-Typ, einem Real-Typ, oder einem Zeichenkettentyp fester oder variabler Länge sein.

Die folgenden Bilder zeigen die Situationen beim Einlesen von Werten aus einer Textdatei (nicht Input). Leerzeichen werden zur besseren Sichtbarkeit durch '_' dargestellt.

*Hinweis*

Die Besonderheiten von Readln im Zusammenhang mit der Eingabe von Dialogstationen sind in 19.5.2 beschrieben.

```
VAR f:  Text;
    r:  Real;
    c1,
    c2: Char;
```

```
f:   |123•|_3.1415•|A•|B•|┘|
       ↑

f↑:  |1|
```

Bild 19-13a: Zustand nach Reset (f)

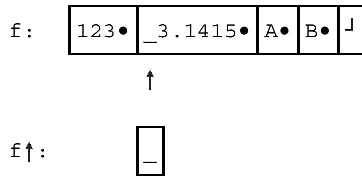



Bild 19-13b: Zustand nach Readln (f)

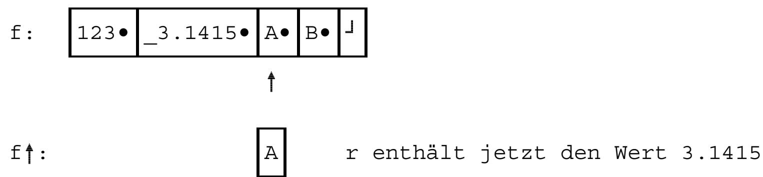



Bild 19-13c: Zustand nach Readln (f, r)

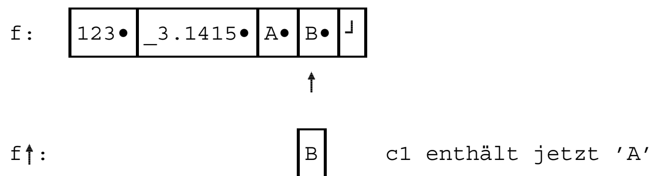



Bild 19-13d: Zustand nach Readln (f, c1)

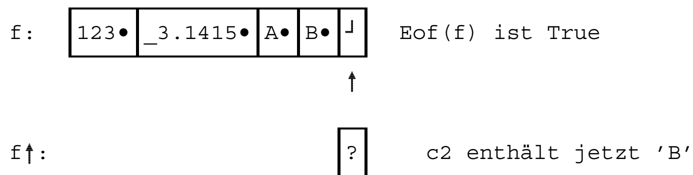



Bild 19-13e: Zustand nach Readln (f, c2)

### Lesen von Zeichen

Ist v eine Variable vom Typ Char oder eines Teilbereichs von Char, dann ist Read (f, v) definiert durch

```
BEGIN v := f↑; Get (f) END
```

Die Variable v enthält das Leerzeichen (Blank) ' ', wenn vor dem Aufruf Read (f, v) Eoln (f) True ist.

```
VAR f: Text;
    c: Char;
```

```
f:   | Zeile1• | Zeile2• | ┘ |
       ↑

f↑:  | Z |
```

Bild 19-14a: Zustand nach Reset (f)

```
f:   | Zeile1• | Zeile2• | ┘ |
        ↑

f↑:  | e |     c enthält jetzt das Zeichen 'Z'
```

Bild 19-14b: Zustand nach Read (f, c)

```
f:   | Zeile1• | Zeile2• | ┘ |
             ↑

f↑:        | _ |     c enthält jetzt das Zeichen '1'
```

Bild 19-14c: Nach weiteren fünf Aufrufen von Read (f, c)

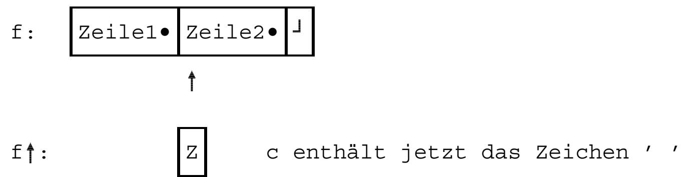



Bild 19-14d: Zustand nach dem nächsten Read (f, c)

### Lesen von Integerzahlen

Ist v eine Variable eines Integer-Typs oder eines Teilbereichs davon, so wird eine Folge von Zeichen gelesen. Führende Leerzeichen und Zeilenenden werden überlesen. Unmittelbar vor der Zahl kann ein Vorzeichen stehen. Der Lesevorgang wird beendet, wenn ein gelesenes Zeichen nicht mehr Teil einer Integer-Zahl sein kann. Dieses Zeichen steht dann in f↑.

Die eingelesene Ziffernfolge muß einer Integer-Zahl in Dezimalschreibweise gemäß 3.5 entsprechen. Der Wert, der dieser Ziffernfolge entspricht, wird an die Variable v zugewiesen.

```
VAR f: Text;
    i: Integer;
```

```
f:   | 123• | __-5___• | • | 88• | ┘ |
        ↑

f↑:  | 1 |
```

Bild 19-15a: Zustand nach Reset (f)

```
f:   | 123• | __-5___• | • | 88• | ┘ |
            ↑

f↑:      | _ |      i enthält jetzt den Wert 123
```

Bild 19-15b: Zustand nach Read (f, i)

```
f:   | 123• | __-5___• | • | 88• | ┘ |
                    ↑

f↑:              | _ |      i enthält jetzt den Wert -5
```

Bild 19-15c: Zustand nach dem nächsten Read (f, i)

```
f:   | 123• | __-5___• | • | 88• | ┘ |
                                 ↑
```

```
f↑:                           | _ |     i enthält jetzt den Wert 88
```

Bild 19-15d: Zustand nach dem nächsten Read (f, i)

### Lesen von Realzahlen

Ist v eine Variable eines Real-Typs, dann wird eine Folge von Zeichen gelesen. Führende Leerzeichen und Zeilenenden werden überlesen. Unmittelbar vor der Zahl kann ein Vorzeichen stehen. Der Lesevorgang wird beendet, wenn ein gelesenes Zeichen nicht mehr Teil einer Real-Zahl sein kann (gemäß 3.5). Dieses Zeichen steht dann in f↑. Der Wert der gelesenen Zahl wird dann auf v übertragen.

```
VAR f: Text;
    r: Real;
```

```
f:   3.1415• __-5E10__• ┘
     ↑

f↑:  3
```

Bild 19-16a: Zustand nach Reset (f)

```
f:   3.1415• __-5E10__• ┘
            ↑

f↑:         _     r enthält jetzt den Wert 3.1415
```

Bild 19-16b: Zustand nach Read (f, r)

```
f:   3.1415• __-5E10__• ┘
                       ↑

f↑:                    _     r enthält jetzt den Wert -5·10^10
```

Bild 19-16c: Zustand nach dem nächsten Read (f, r)

### Leseparameter von einem String-Typ

Ist v eine Variable eines String-Typs, dann werden alle Zeichen bis zum Ende der aktuellen Zeile gelesen und auf v übertragen. Nach dem Read ist Eoln (f) True und f↑ enthält die Zeilenende-Komponente.

```
VAR f: Text;
    s: String;
```

```
f:   | Zeichenkette• | Zeile2• | ┘ |
       ↑

f↑:  | Z |
```

Bild 19-17a: Zustand nach Reset (f)

```
f:   | Zeichenkette• | Zeile2• | ┘ |
                    ↑

f↑:                | _ |      s enthält jetzt 'Zeichenkette'
```

Bild 19-17b: Zustand nach Read (f, s)

Nun muß zwingend ein Readln (f) erfolgen, da sonst bei einem nachfolgenden Read (f, s) nichts eingelesen wird und der Dateizeiger nach wie vor auf der Zeilenende-Komponente stehen bleibt.

```
f:   | Zeichenkette• | Zeile2• | ┘ |
                    ↑

f↑:                | _ |      s enthält die leere Zeichenkette
```

Bild 19-17c: Zustand nach Read (f, s)

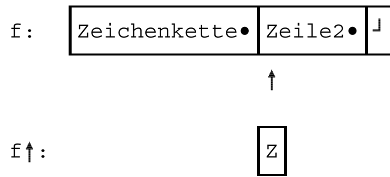



Bild 19-17d: Zustand nach Readln (f)

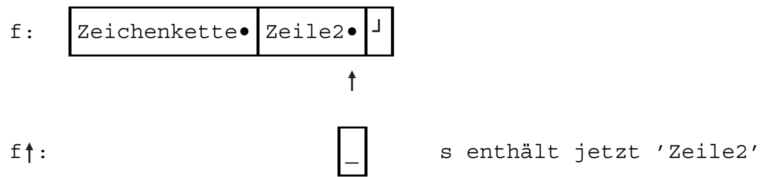



Bild 19-17e: Zustand nach Read (f, s)

### Leseparameter von einem Zeichenkettentyp fester Länge

Ist v eine Variable eines Zeichenkettentyps fester Länge (PACKED ARRAY [1..n] OF Char), dann wird das Einlesen beendet, wenn entweder die Anzahl der gelesenen Zeichen gleich n ist, oder das Ende der Zeile erreicht wird. Im letzteren Fall werden die restlichen Zeichen von v mit Leerzeichen (Blanks) aufgefüllt.

```
VAR f: Text;
    a: PACKED ARRAY [1..6] OF Char;
```

```
f:   |Pascal-XT•|Zeile2•|┘|
      ↑

f↑:  |P|
```

Bild 19-18a: Zustand nach Reset (f)

```
f:   |Pascal-XT•|Zeile2•|┘|
            ↑

f↑:         |-|      a enthält jetzt 'Pascal'
```

Bild 19-18b: Zustand nach Read (f, a)

```
f:   |Pascal-XT•|Zeile2•|┘|
               ↑

f↑:            |_|      a enthält jetzt '-XT   '
```

Bild 19-18c: Zustand nach Read (f, a)

Nun muß zwingend ein Readln (f) erfolgen, da sonst bei einem nachfolgenden Read (f, a) nichts eingelesen wird und der Dateizeiger nach wie vor auf der Zeilenende-Komponente stehen bleibt.

```
f:   | Pascal-XT• | Zeile2• | ┘ |
                 ↑

f↑:              | _ |      a enthält jetzt '      '
```

Bild 19-18d: Zustand nach Read (f, a)

```
f:   | Pascal-XT• | Zeile2• | ┘ |
                   ↑

f↑:                | Z |
```

Bild 19-18e: Zustand nach Readln (f)

```
f:   | Pascal-XT• | Zeile2• | ┘ |
                          ↑

f↑:                     | _ |      a enthält jetzt 'Zeile2'
```

Bild 19-18f: Zustand nach Read (f, a)

## Lesen von der Datensichtstation

Bei Programmanfang wird die Datei Input automatisch zum Lesen eröffnet. Dies bedeutet, daß implizit ein Reset (Input) durchgeführt wird. Reset bedingt aber nicht nur das Eröffnen einer Datei zum Lesen, sondern auch das Laden der ersten Komponente der Datei in die Puffervariable. Da der Datei Input standardmäßig die Datensichtstation zugeordnet ist, würde dies zu einer Eingabeaufforderung führen. Zur Lösung dieses Problems wird in Pascal-XT eine virtuelle Zeile 0 eingeführt, die nur eine Zeilenende-Komponente enthält. Nach dem impliziten Reset(Input) ist Eoln(Input) True und Input↑ enthält ein Leerzeichen.

Zum Einlesen eines Zeichens oder einer Zeichenkette von einer neuen Zeile ist folgende Aufrufsequenz empfehlenswert:

```
Readln (Input);
Read   (Input, v);
```

Beim Einlesen einer Integerzahl oder einer Realzahl werden alle führenden Leerzeichen und Zeilenenden überlesen. Daher kann das vorausgehende Readln entfallen.

Ein weiteres Problem beim Lesen von der Datensichtstation ist das Verhalten von Readln. Readln überliest ja die restlichen Zeichen der aktuellen Zeile und positioniert auf das erste Zeichen der nächsten Zeile. Auf der Datensichtstation gibt es aber noch keine nächste Zeile, sie muß erst vom Benutzer eingegeben werden. Dies bedeutet, daß jedes Readln und auch jedes Lesen der Zeilenende-Komponente mittels Read zu einer Eingabeaufforderung an den Benutzer führt. Die Anweisung der Form

```
Readln (Input, v1,..., vn)
```

hat die Wirkung, daß erst die Werte für die Variablen v1 bis vn eingelesen werden und anschließend eine neue Zeile angefordert wird.

## Schreiben in eine Textdatei

**Write (f, p1,..., pn)**
**Writeln (f, p1,..., pn)**

Die Prozedur Write schreibt Werte (Schreibparameter) p1,...,pn in die aktuelle Zeile der Textdatei f. Writeln wirkt wie die Prozedur Write mit der zusätzlichen Eigenschaft, daß die aktuelle Zeile mit einer Zeilenende-Komponente abgeschlossen wird und auf den Anfang einer neuen Zeile positioniert wird.

In Pascal-XT werden die Ausgaben mit Write zuerst in einen internen Zeilenpuffer ausgegeben. Erst durch Writeln erfolgt die Ausgabe in die Datei, bzw. wird bei Output die Ausgabe auf der Datensichtstation sichtbar.

Die einzelnen Schreibparameter können die Form

a oder a:a1 oder a:a1:a2

haben.

amuß ein Ausdruck von einem der Typen Char, Integer, Real, Boolean, oder ein Zeichenkettentyp (fester oder variabler Länge) sein. Bei der Ausgabe werden Werte eines Integer-Typs oder Real-Typs in die Dezimaldarstellung umgewandelt. Werte vom Typ Boolean werden als Zeichenkette ('True' bzw. 'False', die Schreibweise ist implementierungsdefiniert) ausgegeben.

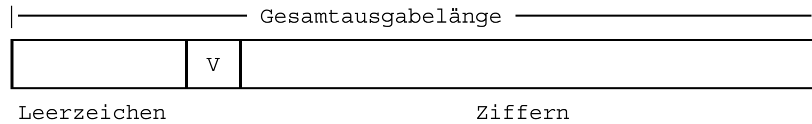
Der Ausdruck a1 bestimmt die Gesamtausgabelänge, in welcher der Wert ausgegeben wird. Ist a1 größer, als für die Darstellung des Wertes benötigt, dann wird der Wert rechtsbündig in dem Feld abgelegt. Ist a1 kleiner als die benötigte Länge, dann gilt:

- Bei Werten von einem Integer- oder Real-Typ wird die benötigte Anzahl von Zeichen ausgegeben.
- Zeichenketten von einem Zeichenkettentyp werden in der Länge a1 ausgegeben, der Rest wird abgeschnitten.
- Werte vom Typ Boolean werden wie Werte eines Zeichenkettentyps ausgegeben.

Bei fehlender Angabe von a1 wird für Werte von einem Integer-Typ, einem Real-Typ und dem Typ Boolean eine implementierungsdefinierte Anzahl von Zeichen ausgegeben. Werte vom Typ Char werden in der Länge 1 und Zeichenketten gemäß ihrer Länge ausgegeben.

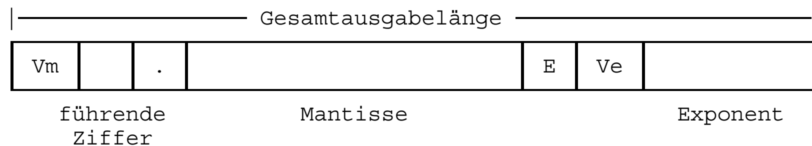
Der Ausdruck a2 kann nur bei Werten eines Real-Typs angegeben werden. Durch diese Angabe werden Realzahlen in der Festpunkt-Darstellung ausgegeben und a2 legt die Anzahl der Ziffern hinter dem Dezimalpunkt fest. Ist a2 größer, als zur Ausgabe mit maximaler Genauigkeit notwendig ist, so werden an die Dezimalstellen Nullen angehängt. Ist a2 kleiner als nötig, so werden Dezimalstellen gerundet.

In den folgenden Bildern werden die Ausgabeformate für Werte von Integer- und Real-Typen dargestellt. Auf die Darstellung der Formate für die anderen Typen wird wegen ihrer Einfachheit verzichtet.



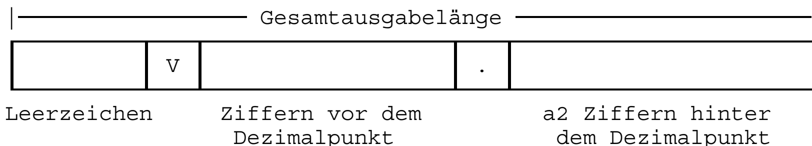

V: Vorzeichen (Leerzeichen oder '-')

Bild 19-19: Ausgabeformat für Integerwerte




Vm: Vorzeichen der Werts a (Leerzeichen oder '-')
Ve: Vorzeichen des Exponenten ('+' oder '-')
E: Exponentenzeichen

Bild 19-20: Gleitpunkt-Darstellung für Realwerte




V: Vorzeichen (Leerzeichen oder '-')

Bild 19-21: Festpunkt-Darstellung für Realwerte

# Dynamische Daten und Speicherverwaltung

Pascal unterscheidet zwei Arten von Variablen, je nachdem, wann und wo für sie Speicher bereitgestellt wird:

- statisch deklarierte Variable und
- dynamisch erzeugte Variable.

Statisch deklarierte Variable werden in einer Variablendeklaration, wie z.B. "VAR a: person;" vereinbart. Der Übersetzer reserviert den Speicherplatz für statische Variable in dem Block, in dem sie vereinbart wurden. Sie werden beim Eintritt in den Block erzeugt und beim Verlassen des Blocks wieder vernichtet.

Dynamische Variable werden erst zur Laufzeit erzeugt. Für sie wird Speicherplatz in einem eigenen Speicherbereich, der Halde (Heap), reserviert. Um sie bearbeiten zu können, vereinbart man zunächst ihren Typ, um dann mit Hilfe dieses Typs (des Domänentyps) Zeiger-Variable zu deklarieren. Durch diese Form der Deklaration sind die Zeiger-Variablen streng an ihren Domänentyp gebunden und können nur auf dynamische Variable desselben Typs zeigen.

Die Werte, die Zeiger-Variable annehmen können, werden Verweiswerte genannt. Die möglichen Werte eines Zeigertyps umfassen den Wert NIL (ein definierter, leerer Verweiswert) und die Menge von Verweiswerten auf dynamische Variable des Domänentyps. Verweiswerte entstehen ausschließlich durch die Anwendung der vordefinierten Prozedur New auf Zeiger-Variablen. Sei p eine Zeiger-Variable mit dem Domänentyp t, dann wird durch New (p) eine dynamische Variable des Typs t erzeugt und p bekommt den Verweiswert auf diese Variable zugewiesen.

Die Generierung und Verwaltung der dynamischen Speicherbereiche erfolgt generell über vordefinierte Prozeduren. Von Norm-Pascal sind die Prozeduren New zur Erzeugung und Dispose zur Freigabe von Speicherplatz für dynamische Variable vorgesehen. In Pascal-XT existieren von diesen vordefinierten Prozeduren zusätzliche Varianten, um Zeiger-Variable mit bestimmten Domänentypen (siehe 15.2) verkürzt auf der Halde anzulegen. Damit kann Speicherplatz eingespart werden. Auf die dynamisch verkürzt angelegten Variable darf aber nur noch komponentenweise zugegriffen werden. Als zusätzliche Erweiterung werden noch die Prozeduren Mark und Release zur Verfügung gestellt. Sie stellen eine Vereinfachung der dynamischen Speicherverwaltung dar, da ein Aufruf von Release alle mit New erzeugten Variablen wieder freigibt, die seit dem zugehörigen Aufruf von Mark erzeugt wurden.

Eine Zeiger-Variable p hat erst nach einem Aufruf von New (p) einen definierten Wert. Vorheriges Ansprechen führt zu einem Laufzeitfehler mit undefinierten Auswirkungen. Die dynamisch erzeugte Variable p↑ ist nach der Erzeugung noch vollständig undefiniert.

Der Zugriff auf die dynamische Variable p↑ erfolgt durch Dereferenzierung der Zeiger-Variablen p. In Pascal-XT wurde als Erweiterung von Norm-Pascal der Begriff des dereferenzierten Objekts eingeführt, der ermöglicht, daß neben Zeiger-Variablen auch Funktions-Aufrufe, deren Ergebnistyp ein Zeigertyp ist, dereferenziert werden können (siehe 9.6.4).

Eine dynamische Variable wird nicht nur durch den Aufruf einer der Prozeduren Dispose oder Release unzugänglich gemacht, sondern auch durch Zuweisung eines anderen Verweiswertes an die Zeiger-Variable p.

Sollen mehrere Variable desselben Domänentyps gleichzeitig bearbeitet werden, dann können

- ebenso viele Zeiger-Variablen deklariert werden, wie dynamische Variable benötigt werden, oder
- es werden die dynamischen Variablen miteinander verkettet.

Letztere Möglichkeit ist nicht nur flexibler und eleganter, sondern sie bildet die Hauptanwendung von Zeigern schlechthin: den Aufbau von dynamischen Datenstrukturen (z.B. verkettete Listen). Die Verkettung erreicht man dadurch, daß im Domänentyp (der dann ein RECORD sein muß) eine Komponente von diesem Zeigertyp deklariert wird, z. B. folgendermaßen:

```
TYPE
   list    = ↑element;
   element = RECORD
                next: list;
                ...
             END;
```

Zusammenfassend seien noch einmal die Wesensmerkmale dynamischer Datenstrukturen und deren praktische Bedeutungen genannt:

- Speicherplatz wird nur solange belegt, wie er benötigt wird. Vor allem kann dies der Programmierer selbst bestimmen.
- Beliebige Datenstrukturen können aufgebaut werden.

Die Anwendungsmöglichkeiten werden anhand einer einfach verketteten Liste (Beispiel 1) und eines Binärbaumes (Beispiel 2) aufgezeigt.

*Querverweise*

| | |
|---|---|
| Zeigertypen: | 6.4 |
| Variablen: | 7 |
| Objekte: | 9.6 |
| Dereferenzierung: | 9.6.4 |

# Beispiel 1: Einfach verkettete Liste

Die Aufgabe sei die Bearbeitung einer Liste von Namen und Telefonnummern, bei der die Namen alphabetisch (Zeichenkettenvergleich) geordnet sein sollen. Eine für die Bearbeitung solcher Listen typische Datenstruktur wird folgendermaßen definiert:

```
TYPE
   string17  = String[17];
   pliste    = ↑listentyp;
   listentyp = RECORD
                  name,
                  telefon_nr : string17;
                  nachfolger : pliste;
               END;
VAR
   liste : pliste;
```

Die Komponente nachfolger soll auf das Listenelement mit dem alphabetisch nachfolgenden Namen verweisen. Die so definierte Liste nennt man einfach verkettet. Ebenso kann man noch ein RECORD-Feld zur Rückwärts-Verkettung hinzufügen, etwa: "vorgaenger : pliste", wobei vorgaenger dann auf das Listenelement verweist, dessen Wert der Komponente name alphabetisch vor dem aktuellen Element kommt. Eine solche Liste wäre dann doppelt verkettet.

Die Definition des Zeigertyps pliste ist hier vor der Definition von listentyp nötig, da dieser Zeigertyp für das Feld nachfolger benötigt wird. Diese Situation ist typisch für rekursive Datenstrukturen. Während in Pascal sonst Bezeichner vor ihrer Anwendung definiert sein müssen, ist im Falle von Zeigertypen eine Ausnahme von dieser Regel zulässig - eben um rekursive Datenstrukturen zu ermöglichen. In einem Deklarationsteil kann ein Domänentyp für die Definition eines Zeigertyps verwendet werden, bevor er selbst (im selben Deklarationsteil) definiert worden ist.

Die Variable liste muß mit "liste := NIL" vor der Bearbeitung durch die folgenden Prozeduren initialisiert werden. Typische Operationen auf einer Liste sind das Einfügen am Anfang einer Liste, bzw. an einer bestimmten Stelle der Liste. Zum Einfügen vor dem ersten Listenelement dient folgende Prozedur:

```
PROCEDURE einfuegen (n, t: string17; VAR p: pliste);
VAR
   q : pliste;
BEGIN
   New (q);
   q↑.name       := n;
   q↑.telefon_nr := t;
   q↑.nachfolger := p;
   p             := q;
END { einfuegen };
```

Mit Hilfe dieser Prozedur kann man eine weitere Prozedur formulieren, die in eine aufsteigend sortierte Liste einen neuen Eintrag an der richtigen Stelle einordnet:

```
PROCEDURE eintragen (n, t: string17; VAR p: pliste);
BEGIN
   IF p = NIL THEN
      einfuegen (n, t, p)
   ELSE IF n > p↑.name THEN
      eintragen (n, t, p↑.nachfolger)
   ELSE IF n < p↑.name THEN
      einfuegen (n, t, p)
   ELSE
      p↑.telefon_nr := t; { neue Telefonnummer }
END { eintragen }
```

Im Falle der Prozedur eintragen ist die Verwendung der Rekursion nicht zwingend. Mit Hilfe einer Wiederholungsanweisung wäre das Problem sogar effizienter zu lösen. In beiden Fällen wird jedoch durch Aufruf der Prozedur eintragen eine lineare Suche in der Liste der vorhandenen Einträge veranlaßt. Die Zahl der Suchschritte für einen beliebigen Eintrag wächst hier proportional zur Anzahl der schon vorhandenen Listenelemente.

# Beispiel 2: Binärbaum

Effizientere Algorithmen sind bei der Verwendung baumförmiger Strukturen möglich. Statt eines Zeigers nachfolger, wie im Falle der Liste in Beispiel 1, werden 2 Zeiger lbaum und rbaum verwendet. Für jeden Knoten des Baumes ergibt sich dann folgende Situation: Der Zeiger lbaum verweist auf einen linken Teilbaum und rbaum verweist auf den rechten Teilbaum. Alle Teilbäume haben dieselbe Datenstruktur wie der ganze Baum. Beim Erstellen des Baumes kann dafür gesorgt werden, daß für jeden Baumknoten folgende Regel gilt: Alle Namen im linken Teilbaum sind kleiner und alle Namen im rechten Teilbaum sind größer als der Name im aktuellen Knoten. Der Vorteil dieser Datenstruktur besteht in den im allgemeinen kurzen Wegen bis zu einem Knoten mit einem bestimmten Namen. Das Einfügen neuer bzw. das Suchen vorhandener Knoten verkürzt sich so erheblich aufgrund der Verzweigungen.

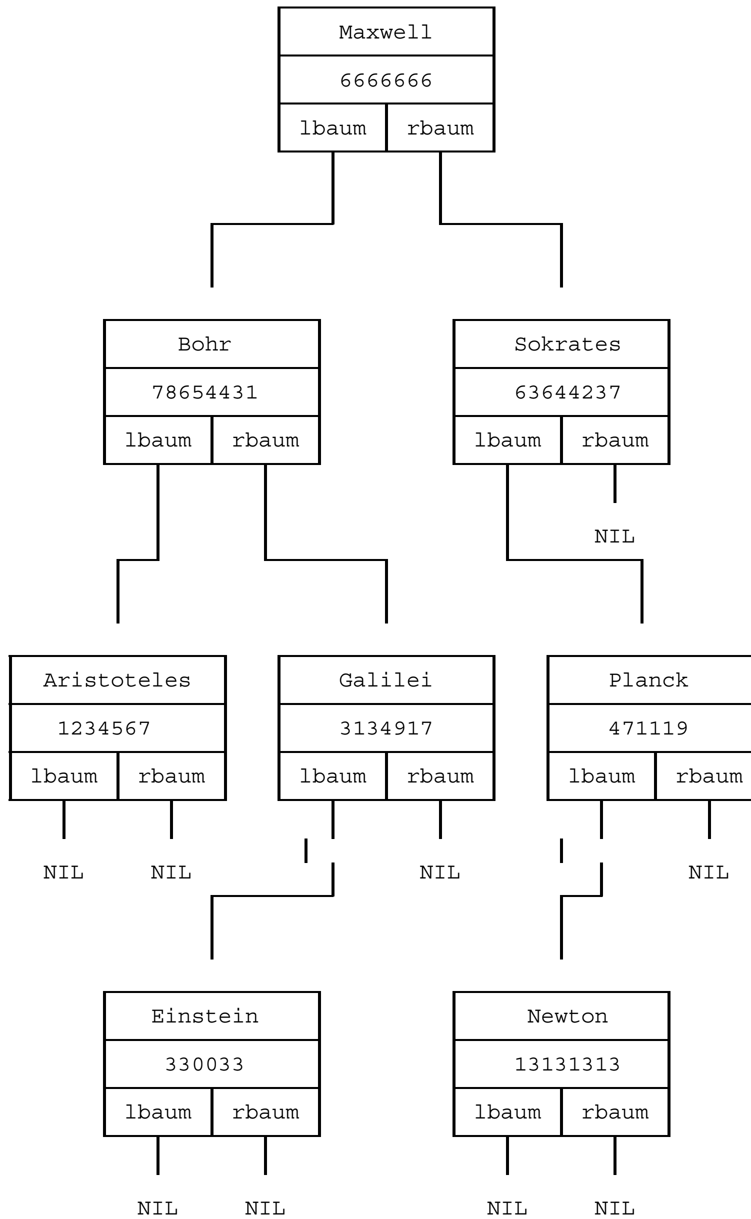



Bild 20-1: Aufbau eines Binärbaumes

Eine für die Bearbeitung solcher Bäume typische Datenstruktur wird folgendermaßen definiert:

```
TYPE
   string17= String[17];
   pbaum   = ↑baumtyp;
   baumtyp = RECORD
                name,
                telefon : string17;
                lbaum,
                rbaum   : pbaum;
             END;

VAR
   baum : pbaum;
```

Die Variable baum sollte mit baum := NIL vor der Bearbeitung durch die folgenden Prozeduren initialisiert worden sein. Typische Operationen auf einer Baumstruktur sind das Anhängen eines neuen Knotens an einen mit NIL endenden Zweig:

```
PROCEDURE anhaengen (n, t: string17; VAR p: pbaum);
BEGIN
   new (p);
   p↑.name    := n;
   p↑.telefon := t;
   p↑.lbaum   := NIL;
   p↑.rbaum   := NIL;
END { anhaengen };
```

Mit Hilfe dieser Prozedur kann man eine weitere Prozedur formulieren, die in einem binär geordneten Baum einen neuen Eintrag an der richtigen Stelle vornimmt:

```
PROCEDURE eintragen (n, t: string17; VAR p: pbaum);
BEGIN
   IF p = NIL THEN
      anhaengen (n, t, p)
   ELSE IF n > p↑.name THEN
      eintragen (n, t, p↑.rbaum)
   ELSE IF n < p↑.name THEN
      eintragen (n, t, p↑.lbaum)
   ELSE
      p↑.telefon := t; { neue Telefonnummer }
END { eintragen }
```

In diesem Fall wäre eine Prozedur mit Wiederholungsanweisungen wesentlich unübersichtlicher. Da die angegebene Datenstruktur rekursiv ist (sie besteht aus einem Baum mit gleichartigen Unterbäumen), ist eine Verwendung rekursiver Algorithmen hier natürlicher.

Ein weiteres Beispiel für die Verwendung von Rekursion ist die folgende Prozedur, die einen Baum der angegebenen Datenstruktur nach Namen sortiert ausgibt:

```
PROCEDURE druckebaum (p: pbaum);
BEGIN
   IF p <> NIL THEN BEGIN
      druckebaum (p↑.lbaum);
      Writeln (p↑.name, ' hat Telefonnummer ', p↑.telefon);
      druckebaum (p↑.rbaum);
   END
END { druckebaum };
```

Diese Prozedur definiert einen bestimmten Weg durch den Baum. An bestimmten Punkten dieses Weges wird eine Ausgabe gemacht. In diesem Fall immer dann, wenn der linke Teilbaum bereits vollständig durchlaufen ist. Dadurch ergibt sich eine alphabetisch geordnete Ausgabe der Namen mit zugehörigen Telefonnummern. Werden die 3 mittleren Zeilen in anderer Reihenfolge angegeben, ergibt sich eine andere Reihenfolge der auf dem Weg anzutreffenden Knoten. So würde beispielsweise ein Vertauschen der beiden Zeilen druckebaum (p↑.lbaum) und druckebaum (p↑.rbaum) eine Ausgabe in umgekehrter alphabetischer Reihenfolge bewirken.

# Anhang

## Syntax von Pascal-XT

*Hinweis*

Die Wurzel der Syntax ist "Übersetzungseinheit".

```
Abstand         = Integer-Konstante.

Aggregat        = ARRAY-Aggregat | RECORD-Aggregat.

Aktualparameter = Variablen-Objekt | Ausdruck | Prozedur-Name
                | Funktions-Name   | Typ-Name
                | Paket-Bezeichner | Ausdruck ":" Formatangabe.

Aktualparameterliste
                = "(" Aktualparameter {"," Aktualparameter} ")".

Anfangswert     = Ordinal-Ausdruck.

Anweisung       = [Marke ":"]
                  (   einfache_Anweisung     | bedingte_Anweisung
                    | Wiederholungsanweisung | Verbundanweisung
                    | WITH-Anweisung ).

Anweisungsfolge = Anweisung {";" Anweisung}.

Anweisungsteil  = Verbundanweisung.

Apostrophdarstellung
                = "''".

ARRAY-Aggregat  = ARRAY-Typ-Name "(" ARRAY-Aggregat-Element
                              {"," ARRAY-Aggregat-Element} ")".

ARRAY-Aggregat-Element
                = Ausdruck [":" Wiederholfaktor].

ARRAY-Typ       = "ARRAY" "[" Indextyp {"," Indextyp} "]" "OF"
                     Komponententyp.

Aufzählungstyp  = "(" Bezeichnerliste ")".

Auswahl         = Fallkonstante [".." Fallkonstante].
```

```
Auswahlliste    = Auswahl {"," Auswahl} | "ELSE".

Ausdruck        = einfacher_Ausdruck
                  [Vergleichsoperator einfacher_Ausdruck].

Basistyp        = Ordinal-Typangabe.

bedingte_Anweisung
                = IF-Anweisung | CASE-Anweisung.

Bezeichner      = Buchstabe {["_"] ( Buchstabe | Ziffer )}.

Bezeichnerliste = Bezeichner {"," Bezeichner}.

Bitbereich      = Integer-Konstante ".." Integer-Konstante.

Block           = {  Markendeklarationsteil
                   | Konstantendefinitionsteil
                   | Typdefinitionsteil
                   | Variablendeklarationsteil
                   | Prozedurdeklaration
                   | Funktionsdeklaration
                  }
                  Anweisungsteil.

Bruchteil       = Ziffernfolge.

Buchstabe       = "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".

CASE-Anweisung  = "CASE" Fallindex "OF" Fall-Liste [";"] "END".

Direktive       = "c"       | "cobol" | "external" | "fortran"
                | "forward" | "internal".

Domänentyp      = Typ-Name.

dereferenziertes_Objekt
                = Zeiger-Objekt "↑".

einfache_Anweisung
                = Leeranweisung  | Zuweisung
                | Prozeduraufruf | GOTO-Anweisung
                | EXIT-Anweisung | RETURN-Anweisung.

einfacher_Ausdruck
                = [Vorzeichen] Term {Summationsoperator Term}.

Elementbestimmung
                = Ordinal-Ausdruck [".." Ordinal-Ausdruck].

ELSE-Teil       = "ELSE" Anweisung.

Endwert         = Ordinal-Ausdruck.

Ergebnistyp     = Typ-Name.
```

```
EXCEPTION-Teil  = "EXCEPTION" Anweisungsfolge.

EXIT-Anweisung  = "EXIT".

Exponent        = ["+" | "-"] Ziffernfolge.

Faktor          = Primitiv ["**" Primitiv] | "NOT" Faktor.

Fall-Liste      = Fall-Listenelement {";" Fall-Listenelement}.

Fall-Listenelement
                = Auswahlliste ":" Anweisung.

Fallindex       = Ordinal-Ausdruck.

Fallkonstante   = Ordinal-Konstante.

Feldauswahlbezeichner
                = Feld-Bezeichner.

Felddarstellung = Bezeichner ["(" Abstand [":" Bitbereich] ")"].

Felddarstellungsliste
                = Felddarstellung {"," Felddarstellung}.

Feldliste       = [ ( Festteil    [";" Variantteil]
                     | Variantteil )  [";"] ].

Festteil        = RECORD-Abschnitt {";" RECORD-Abschnitt}.

FILE-Typ        = "FILE" "OF" Komponententyp.

FOR-Anweisung   = "FOR" Laufvariable ":=" Anfangswert
                  ( "TO" | "DOWNTO" ) Endwert "DO" Anweisung.

Formalparameterabschnitt
                = Wertparameter-Spezifikation
                | Variablenparameter-Spezifikation
                | Parameterprozedur-Spezifikation
                | Parameterfunktions-Spezifikation
                | Konformreihungs-Parameter-Spezifikation.

Formalparameterliste
                = "(" Formalparameterabschnitt
                   {";" Formalparameterabschnitt} ")".

Formatangabe    = Integer-Ausdruck [":" Integer-Ausdruck].

Funktionsaufruf = Funktions-Name [Aktualparameterliste].

Funktionsdeklaration
                = Funktionskopf ";" Direktive ";"
                | Funktionskkpf ";" Funktions-Block ";"
                | Funktionsidentifikation ";" Funktions-Block ";"
                | INLINE-Funktionsdeklaration.

Funktionsidentifikation
                = "FUNCTION" Funktions-Bezeichner
```

```
[[Formalparameterliste] ":" Ergebnistyp].
```

```
Funktionskopf    = "FUNCTION" Bezeichner
                   [Formalparameterliste] ":" Ergebnistyp.

Ganzteil         = Ziffernfolge.

gepacktes_Konformreihungs-Schema
                 = "PACKED" "ARRAY" "[" Indextyp-Spezifikation "]"
                   "OF" Typ-Name.

GOTO-Anweisung   = "GOTO" Marke.

Hauptprogramm    = {Kontext-Spezifikation}
                   "PROGRAM" Bezeichner
                   ["(" Programmparameterliste ")"] ";"
                   Hauptprogramm-Block ".".

IF-Anweisung     = "IF" boolescher-Ausdruck "THEN" Anweisung
                   [ELSE-Teil].

importierter_Bezeichner
                 = Konstanten-Bezeichner | Typ-Bezeichner
                 | Variablen-Bezeichner  | Prozedur-Bezeichner
                 | Funktions-Bezeichner.

Indexausdruck    = Ordinal-Ausdruck.

Indextyp         = Ordinal-Typangabe.

Indextyp-Spezifikation
                 = Bezeichner ".." Bezeichner ":" Ordinaltyp-Name.

indiziertes_Objekt
                 = ARRAY-Objekt
                   "[" Indexausdruck {"," Indexausdruck} "]"
                 | String-Objekt "[" Indexausdruck "]".

INLINE-Funktionsdeklaration
                 = "INLINE" Funktionskopf ";" Funktions-Block ";".

INLINE-Prozedurdeklaration
                 = "INLINE" Prozedurkopf ";" Prozedur-Block ";".

Kennungsfeld     = Felddarstellung.

Kennungstyp      = Ordinaltyp-Name.

Komponententyp   = Typangabe.

Konformreihungs-Parameter-Spezifikation
                 = Wert-Konformreihungs-Spezifikation
                 | Variablen-Konformreihungs-Spezifikation.

Konformreihungs-Schema
                 = gepacktes_Konformreihungs-Schema
                 | ungepacktes_Konformreihungs-Schema.

Konstante        = statischer_Ausdruck.
```

```
Konstantendefinition
                = Bezeichner "=" Konstante ";".

Konstantendefinitionsteil
                = "CONST" Konstantendefinition
                         {Konstantendefinition}.

Kontext-Spezifikation
                = WITH-Liste | USE-Liste.

Laufvariable    = Variablen-Bezeichner.

Leeranweisung   = .

Marke           = Ziffernfolge.

Markendeklarationsteil
                = "LABEL" Marke {"," Marke} ";".

Mengenbildner   = "[" [Elementbestimmung
                  {"," Elementbestimmung}] "]".

Multiplikationsoperator
                = "*" | "/" | "DIV" | "MOD" | "AND" | "AND" "THEN".

Name            = [Paket-Bezeichner "."] Bezeichner.

neuer_Typ       = Aufzählungstyp  | Teilbereichstyp | Stringtyp
                | Zeigertyp       | strukturierter_Typ.

Objekt          = Konstanten-Name          | Variablen-Name
                | Aggregat                 | Funktionsaufruf
                | indiziertes_Objekt       | selektiertes_Objekt
                | dereferenziertes_Objekt  | Puffervariable.

Option          = Bezeichner.

Paket-Implementierung
                = {Kontext-Spezifikation}
                  "PACKAGE" "BODY" Paket-Bezeichner
                  ["(" Programmparameterliste ")"] ";"
                  {  Konstantendefinitionsteil
                   | Typdefinitionsteil
                   | Variablendeklarationsteil
                   | Prozedurdeklaration
                   | Funktionsdeklaration
                  }
                  Anweisungsteil ".".
```

```
Paket-Spezifikation
                = {Kontext-Spezifikation}
                  "PACKAGE" Bezeichner
                  ["(" Programmparameterliste ")"] ";"
                  {  Konstantendefinitionsteil
                   | Typdefinitionsteil
                   | Variablendeklarationsteil
                   | Prozedurkopf  ";" [Direktive ";"]
                   | Funktionskopf ";" [Direktive ";"]
                   | "ENTRY" Prozedurkopf  ";"
                   | "ENTRY" Funktionskopf ";"
                   | INLINE-Prozedurdeklaration
                   | INLINE-Funktionsdeklaration
                  }
                  "END" ".".

Parameterfunktions-Spezifikation
                = Funktionskopf.

Parameterprozedur-Spezifikation
                = Prozedurkopf.

Primitiv        = vorzeichenlose_Konstante     | Grenz-Bezeichner
                | "(" Ausdruck ")"             | Mengenbildner
                | qualifizierter_Mengenbildner | Objekt.

Programmparameterliste
                = Bezeichnerliste.

Prozeduraufruf  = Prozedur-Name [Aktualparameterliste].

Prozedurdeklaration
                = Prozedurkopf ";" Direktive ";"
                | Prozedurkopf ";" Prozedur-Block ";"
                | Prozeduridentifikation ";" Prozedur-Block ";"
                | INLINE-Prozedurdeklaration.

Prozeduridentifikation
                = "PROCEDURE" Prozedur-Bezeichner
                  [Formalparameterliste].

Prozedurkopf    = "PROCEDURE" Bezeichner [Formalparameterliste].

Pseudokommentar = "{$" Steueranweisung
                    {"," Steueranweisung} "}".

Puffervariable  = FILE-Objekt "↑".

qualifizierter_Mengenbildner
                = SET-Typ-Name "(" Mengenbildner ")".

RECORD-Abschnitt
                = Felddarstellungsliste ":" Typangabe.

RECORD-Aggregat = RECORD-Typ-Name "(" Ausdruck {"," Ausdruck} ")".
```

```
RECORD-Typ      = "RECORD" Feldliste "END".

RECORD-Variablen-Liste
                = RECORD-Variablen-Objekt
                  {"," RECORD-Variablen-Objekt}.

REPEAT-Anweisung
                = "REPEAT" Anweisungsfolge
                  "UNTIL" boolescher-Ausdruck.

RETURN-Anweisung
                = "RETURN".

selektiertes_Objekt
                = RECORD-Objekt "." Feld-Bezeichner
                | Feldauswahlbezeichner.

SET-Typ         = "SET" "OF" Basistyp.

Sedezimal-Ziffer
                = Ziffer|"a"|"b"|"c"|"d"|"e"|"f".

Sedezimal-Ziffernfolge
                = Sedezimal-Ziffer {Sedezimal-Ziffer}.

Sedezimal-Ziffernpaar
                = Sedezimal-Ziffer Sedezimal-Ziffer.

Spezialsymbol   = "+"|"-"|"*"|"/"|"="|"<"|">"|"["|"]"|"."|","|":"|
                  ";"|"?"|"("|")"|"**"|"<>"|"<="|">="|":="|"..".

Steueranweisung = Option [ "=" "On" | "Off" | "Restricted" |
                                Zeichenkette ].

Stringlaenge    = Integer-Konstante .

Stringtyp       = String-Bezeichner "[" Stringlaenge "]".

strukturierter_Typ
                = ["PACKED"] ungepackter_strukturierter_Typ.

Summationsoperator
                = "+" | "-" | "OR" | "OR" "ELSE".

Teilbereichstyp = Ordinal-Konstante ".." Ordinal-Konstante.

Term            = Faktor {Multiplikationsoperator  Faktor}.

Typangabe       = Typ-Name | neuer_Typ.

Typdefinition   = Bezeichner "=" Typangabe ";".
```

```
Typdefinitionsteil
                = "TYPE" Typdefinition {Typdefinition}.

Übersetzungseinheit
                = Paket-Spezifikation
                | Paket-Implementierung
                | Hauptprogramm.

ungepackter_strukturierter_Typ
                = ARRAY-Typ | RECORD-Typ | SET-Typ | FILE-Typ.

ungepacktes_Konformreihungs-Schema
                = "ARRAY" "[" Indextyp-Spezifikation
                          {";" Indextyp-Spezifikation} "]" "OF"
                     ( Typ-Name | Konformreihungs-Schema ).

USE-Liste       = "FROM" Paket-Bezeichner "USE"
                     importierter_Bezeichner
                     {"," importierter_Bezeichner} ";".

Variablendeklaration
                = Bezeichnerliste ":" Typangabe ";".

Variablendeklarationsteil
                = "VAR" Variablendeklaration
                       {Variablendeklaration}.

Variablen-Konformreihungs-Spezifikation
                = "VAR" Bezeichnerliste ":" Konformreihungs-Schema.

Variablenparameter-Spezifikation
                = "VAR" Bezeichnerliste ":" Typ-Name.

Variante        = Auswahlliste ":" "(" Feldliste ")".

Variantenselektor
                = [Kennungsfeld ":"] Kennungstyp.

Variantteil     = "CASE" Variantenselektor "OF"
                     Variante {";" Variante} .

Verbundanweisung
                = "BEGIN" Anweisungsfolge [EXCEPTION-Teil] "END".

Vergleichsoperator
                = "=" | "<>" | "<" | ">" | "<=" | ">=" | "IN".

Vorzeichen      = "+" | "-".

vorzeichenlose_Integer-Zahl
                = Ziffernfolge | "#" Sedezimal-Ziffernfolge.

vorzeichenlose_Konstante
                = Konstanten-Name
                | vorzeichenlose_Integer-Zahl
                | vorzeichenlose_Real-Zahl
                | Zeichenkette
                | "NIL".
```

```
vorzeichenlose_Real-Zahl
                = Ganzteil "." Bruchteil ["e" Exponent]
                | Ganzteil "e" Exponent.

Wert-Konformreihungs-Spezifikation
                = Bezeichnerliste ":" Konformreihungs-Schema.

Wertparameter-Spezifikation
                = Bezeichnerliste ":" Typ-Name.

WHILE-Anweisung = "WHILE" boolescher-Ausdruck "DO" Anweisung.

Wiederholfaktor = Integer-Konstante.

Wiederholungsanweisung
                = REPEAT-Anweisung
                | WHILE-Anweisung
                | FOR-Anweisung.

WITH-Anweisung  = "WITH" RECORD-Variablen-Liste "DO" Anweisung.

WITH-Liste      = "WITH" Paket-Bezeichner
                    {"," Paket-Bezeichner} ";".

Wortsymbol      = "AND"   | "ARRAY"  | "BEGIN"   | "BODY"   | "CASE" |
                  "CONST"  | "DIV"   | "DO"   | "DOWNTO"   | "ELSE"   |
                  "END"   | "ENTRY"| "EXCEPTION" | "EXIT" | "FILE" |
                  "FOR" | "FROM"   | "FUNCTION"  | "GOTO"  | "IF"   |
                  "IN"  | "INLINE"  | "LABEL"    | "MOD"   | "NIL"   |
                  "NOT"  | "OF"   | "OR"   | "PACKAGE"   | "PACKED"   |
                  "PROCEDURE"  | "PROGRAM"  | "RECORD" | "REPEAT" |
                  "RETURN"  | "SET"  | "THEN"  | "TO"   | "TYPE"    |
                  "UNTIL"  | "USE"  | "VAR"  | "WHILE" | "WITH".

Zeichenkette    = "'" {Zeichenkettenelement } "'"
                | "#'" {Sedezimal-Ziffernpaar} "'".

Zeichenkettenelement
                = Apostrophdarstellung | Zeichen.

Zeigertyp       = "↑" Domänentyp.

Ziffer          = "0"|"1"|"2"|"3"|"4"|"5"|"6"|"7"|"8"|"9".

Ziffernfolge    = Ziffer {Ziffer}.

Zuweisung       = Variablen-Objekt     ":=" Ausdruck
                | Funktions-Bezeichner ":=" Ausdruck.
```

# Vordefinierte Bezeichner

| | Bezeichner | Abschnitt, der die Definition enthält |
|---|---|---|
| Konstanten: | | |
| | Break_Error | 5.2 |
| | Case_Error | 5.2 |
| | Elab_Error | 5.2 |
| | Eof_Error | 5.2 |
| | False | 5.2 |
| | File_Error | 5.2 |
| | Index_Error | 5.2 |
| | Long_Maxint | 5.2 |
| | Long_Maxreal | 5.2 |
| | Long_Minint | 5.2 |
| | Long_Minreal | 5.2 |
| | Maxint | 5.2 |
| | Maxreal | 5.2 |
| | Memory_Error | 5.2 |
| | Minint | 5.2 |
| | Minreal | 5.2 |
| | Numeric_Error | 5.2 |
| | Open_Error | 5.2 |
| | Pointer_Error | 5.2 |
| | Range_Error | 5.2 |
| | Read_Error | 5.2 |
| | Short_Maxint | 5.2 |
| | Short_Maxreal | 5.2 |
| | Short_Minint | 5.2 |
| | Short_Minreal | 5.2 |
| | Set_Error | 5.2 |
| | String_Error | 5.2 |
| | System_Error | 5.2 |
| | True | 5.2 |
| | Variant_Error | 5.2 |
| Typen: | | |
| | Any_File | 6.5.1 |
| | Any_Type | 6.5.3 |
| | Boolean | 6.2.4 |
| | Char | 6.2.3 |
| | Integer | 6.2.1 |
| | Long_Integer | 6.2.1 |
| | Long_Real | 6.2.2 |
| | Pointer | 6.5.2 |
| | Real | 6.2.2 |
| | Short_Integer | 6.2.1 |
| | Short_Real | 6.2.2 |
| | String | 6.3.2.2 |
| | Text | 6.3.5.2 |
| Variablen: | | |
| | Input | 11.5 |

| | |
|---|---|
| `Output` | 11.5 |

Unterprogramme:

| Bezeichner | Abschnitt, der die Definition enthält |
|---|---|
| Abs | 15.4 |
| Alignof | 15.9 |
| Arctan | 15.4 |
| Assignfile | 15.1 |
| Bitsizeof | 15.9 |
| Card | 15.6 |
| Chr | 15.6 |
| Concat | 15.3 |
| Convert | 15.10 |
| Cos | 15.4 |
| Delete | 15.3 |
| Dispose | 15.2 |
| Elaborate | 15.12 |
| Eof | 15.1 |
| Eoln | 15.1 |
| Error_Number | 15.11 |
| Exp | 15.4 |
| First | 15.9 |
| Get | 15.1 |
| Insert | 15.3 |
| Last | 15.9 |
| Length | 15.3 |
| Ln | 15.4 |
| Long | 15.5 |
| Long_Round | 15.5 |
| Long_Trunc | 15.5 |
| Mark | 15.2 |
| Maxlength | 15.9 |
| New | 15.2 |
| Odd | 15.7 |
| Offsetof | 15.9 |
| Ord | 15.6 |
| Pack | 15.8 |
| Page | 15.1 |
| Position | 15.3 |
| Pred | 15.6 |
| Put | 15.1 |
| Raise | 15.11 |
| Read | 15.1 |
| Readln | 15.1 |
| Readstring | 15.3 |
| Release | 15.2 |
| Reset | 15.1 |
| Rewrite | 15.1 |
| Round | 15.5 |
| Setmax | 15.9 |
| Setmin | 15.9 |
| Short_Round | 15.5 |
| Short_Trunc | 15.5 |
| Sin | 15.4 |
| Sizeof | 15.9 |
| Sqr | 15.4 |
| Sqrt | 15.4 |

| | |
|---|---|
| Substring | 15.3 |
| Succ | 15.6 |
| Trunc | 15.5 |
| Unpack | 15.8 |
| Write | 15.1 |
| Writeln | 15.1 |
| Writestring | 15.3 |

# Bedeutung der Wortsymbole und Spezialsymbole

| Wortsymbol | Abschnitt, der die Definition enthält | |
|---|---|---|
| AND | Boolesche Operatoren | 9.3.2 |
| ARRAY | ARRAY-Typen | 6.3.1 |
| BEGIN | Anweisungsteil | 12.1 |
| | Verbundanweisung | 10.2 |
| CASE | Varianten | 6.3.3.1 |
| | CASE-Anweisung | 10.3.2 |
| CONST | Konstantendefinitionsteil | 5.1 |
| DIV | Arithmetische Operatoren | 9.3.1 |
| DO | WHILE-Anweisung | 10.4.2 |
| | FOR-Anweisung | 10.4.3 |
| | WITH-Anweisung | 10.5 |
| DOWNTO | FOR-Anweisung | 10.4.3 |
| ELSE | IF-Anweisung | 10.3.1 |
| | CASE-Anweisung | 10.3.2 |
| END | Anweisungsteil | 12.1 |
| | Verbundanweisung | 10.2 |
| | CASE-Anweisung | 10.3.2 |
| | RECORD-Typen | 6.3.3 |
| FILE | FILE-Typen | 6.3.5 |
| FOR | FOR-Anweisung | 10.4.3 |
| FUNCTION | Funktionsdeklarationen | 8.2 |
| GOTO | GOTO-Anweisung | 10.1.4 |
| IF | IF-Anweisung | 10.3.1 |
| IN | Mengenoperatoren | 9.3.3 |
| LABEL | Markendeklarationsteil | 4 |
| MOD | Arithmetische Operatoren | 9.3.1 |
| NIL | Konstantendefinitionsteil | 5.1 |
| NOT | Boolesche Operatoren | 9.3.2 |
| OF | Varianten | 6.3.3.1 |
| | CASE-Anweisung | 10.3.2 |
| OR | Boolesche Operatoren | 9.3.2 |
| PACKED | Gepackte Typen | 6.3 |
| PROCEDURE | Prozedurdeklarationen | 8.1 |
| PROGRAM | Programmkopf | 11.1 |
| RECORD | RECORD-Typen | 6.3.3 |
| REPEAT | REPEAT-Anweisung | 10.4.1 |
| SET | SET-Typen | 6.3.4 |
| THEN | IF-Anweisung | 10.3.1 |
| TO | FOR-Anweisung | 10.4.3 |

| | | |
|---|---|---|
| TYPE | Typdefinitionsteil | 6.1 |
| UNTIL | REPEAT-Anweisung | 10.4.1 |
| VAR | Variablendeklarationsteil | 7.1 |
| | Variablenparameter | 8.5.2 |
| | Konformreihungs-Parameter | 8.5.4 |
| WHILE | WHILE-Anweisung | 10.4.2 |
| WITH | WITH-Anweisung | 10.5 |

Wortsymbole Abschnitt, der die Definition enthält

1. in Pascal-XT eingeführte Wortsymbole

| | | |
|---|---|---|
| BODY | Pakete und Programme | 11.2 |
| ENTRY | Pakete und Programme | 11.2 |
| EXCEPTION | Ausnahmebehandlung | 14 |
| EXIT | EXIT-Anweisung | 10.1.5 |
| FROM | Beziehungen zwischen Paketen | 11.2 |
| INLINE | Prozedurdeklarationen | 8.3 |
| | Funktionsdeklarationen | 8.3 |
| PACKAGE | Pakete und Programme | 11.2 |
| RETURN | RETURN-Anweisung | 10.1.6 |
| USE | Beziehungen zwischen Paketen | 11.3 |

2. Standard-Wortsymbole mit zusätzlicher semantischer Bedeutung in PASCAL-XT*

| | | |
|---|---|---|
| AND THEN | Boolesche Operatoren | 9.3.2 |
| ELSE | Varianten | 6.3.3.1 |
| | CASE-Anweisung | 10.3.2 |
| OR ELSE | Boolesche Operatoren | 9.3.2 |
| WITH | Beziehungen zwischen Paketen | 11.3 |

Spezialsymbole

| Symbol | Bedeutungen |
| --- | --- |
| + | Addition, Mengenvereinigung |
| – | Subtraktion, Mengendifferenz |
| * | Multiplikation, Schnittmenge |
| ** | potenziert mit |
| / | Division |
| : | Doppelpunkt, definiert Variable, Trenner |
| = | gleich, |
| | definiert Typ, definiert Konstante |
| < | kleiner als |
| | ist echte Teilmenge |
| > | größer als |
| | ist echte Obermenge |
| <> | ungleich |
| <= | kleiner oder gleich, ist Teilmenge |
| >= | größer oder gleich, ist Obermenge |
| [] | eckige Klammer auf und zu |
| () | runde Klammer auf und zu |
| {} | geschweifte Klammer auf und zu (Kommentar) |
| := | linker Variablen wird rechter Wert zugewiesen |
| • | Dezimalpunkt, Trenner |
| , | Komma, Trenner |
| ; | Semikolon, Trenner |
| .. | 2 Punkte, Bereichsangabe |
| ↑ | Aufwärtspfeil, Dereferenzierung |

# Erweiterungen von Pascal-XT gegenüber der Norm

Alle Erweiterungen von Pascal-XT gegenüber der Pascal-Norm DIN 66 256 können bei der Übersetzung eines Programms durch die Angabe des Pseudokommentars

{$ STANDARD = ON }

im Pascal-Quellprogramm (siehe Kapitel 16) gemeldet werden.

- Wortsymbole und Spezialsymbole
  BODY ENTRY EXCEPTION EXIT FROM
  INLINE PACKAGE RETURN USE
  "**"
- Bezeichner
  In Bezeichnern dürfen Unterstriche enthalten sein
- Direktiven
  C Cobol Fortran External Internal
- Zahlen
  Ganzzahlen können in sedezimaler Form angegeben werden
- Zeichenketten
  - Zeichenketten können in sedezimaler Form angegeben werden Leere Zeichenkette
- Kommentare
  Pseudokommentare zur Steuerung des Compilers
- Deklarationen
  Beliebige Reihenfolge der Deklarationen
- Konstanten
  - NIL darf auf der rechten Seite einer Konstantendeklaration stehen.
  - Statische Ausdrücke auf der rechten Seite einer Deklaration.
  - Zusätzliche vordefinierte Konstanten
- Typen
  - Vordefinierte Typbezeichner:
    Short_Integer Long_Integer Short_Real Long_Real
    Pointer Any_File Any_Type String
  - Record-Typ
    - Angaben zur Speicherdarstellung von Recordfeldern
    - ELSE-Teil im Varianteil eines Recordtyps
    - Angaben von Bereichen in der Selektorkonstanten-Liste

- Prozedur- und Funktions- Deklarationen
  - Wiederholung der Parameterlisten bei den Identifikationen
  - Inline - Unterprogramme
  - Entry - Prozeduren in Paketen
  - Ergebnistyp von Funktionen kann ein beliebiger Typ sein

- Ausdrücke
  - Kurzschlußoperatoren OR ELSE und AND THEN
  - Exponentialoperator "**"
  - Symmetrische Mengendifferenz "/"
  - Echte Teilmengenrelationen "<" und ">"
  - Mengenbildner mit Angabe des Mengentyps
  - Statische Ausdrücke dürfen anstelle von Konstanten stehen
  - Array-Aggregate und Record-Aggregate
  - Indizierung strukturierter Werte
  - Selektierung von Komponenten strukturierter Werte
  - Selektieren, Indizieren und Dereferenzieren von Funktionsergebnissen
  - Vergleich von Zeichenketten verschiedener Länge

- Anweisungen
  - Verbundanweisung mit Ausnahmebehandlungsteil
  - EXIT - Anweisung zum Verlassen einer Schleife
  - RETURN - Anweisung zum Verlassen eines Blocks
  - CASE - Anweisung
    - ELSE als Selektorkonstante
    - Angaben von Bereichen in der Selektorkonstanten-Liste

- Paket-Konzept
  - Paket-Spezifikation und Paket-Implementierungen
  - Kontextspezifikationen WITH und USE
  - Konzept des privaren Zeigertyps (bei Paketen)

- Erweiterungen bei vordefinierten Unterprogrammen

  New Dispose Put Read Write<br>
  Pack Unpack

- Neue vordefinierte Unterprogramme

  Mark Release Assignfile<br>
  Delete Insert Readstring Writestring Length<br>
  Position Concat Substring<br>
  Raise Error_Number<br>
  Elaborate<br>
  Long Short_Round Long_Round Short_Trunc Long_Trunc<br>
  Setmin Setmax Card Sizeof Bitsizeof<br>
  Alignof Offsetof First Last Maxlength<br>
  Convert

# Liste der Laufzeitfehler

In den folgenden Tabellen sind alle im Manual beschriebenen Fehler nach Fehlerklassen sortiert. Damit ist leicht ersichtlich, welche Fehler in einer Fehlerklasse auftreten können.

Zu jedem Fehlertext sind in runden Klammern die Verweise auf die diese Sprachbeschreibung und in eckigen Klammern die Verweise auf die Pascal-Norm, sofern diese Fehler dort erwähnt sind, angegeben.

**NUMERIC_ERROR**

---

- In einem Ausdruck der Form x/y ist y = 0 (9.3.1), [D.44].
- In einem Ausdruck der Form i DIV j ist j = 0 (9.3.1), [D.45].
- In einem Ausdruck der Form i MOD j ist j <= 0 (9.3.1), [D.46].
- Das Ergebnis einer arithmetischen Operation liegt nicht im Wertebereich des Ergebnistyps (9.3.1), [D.47]:
  - für Operanden vom Typ Short_Integer oder eines Teilbereichs davon ist dies der Typ Integer
  - für Operanden vom Typ Long_Integer oder eines Teilbereichs davon ist dies der Typ Long_Integer
  - für Operanden vom Typ Short_Real ist dies der Typ Short_Real
  - für Operanden vom Typ Long_Real ist dies der Typ Long_Real
- Bei Abs(x) liegt das Funktionsergebnis nicht im Wertebereich des Ergebnistyps (Integer bzw. Long_Integer bzw. Short_Real bzw. Long_Real) (15.4).
- Bei Sqr(x) liegt das Funktionsergebnis nicht im Wertebereich des Ergebnistyps (Integer bzw. Long_Integer bzw. Short_Real bzw. Long_Real) (15.4), [D.32].
- Das Ergebnis von Exp(x) liegt nicht im Wertebereich des Ergebnistyps (Short_Real bzw. Long_Real) (15.4).
- Bei Ln(x) ist x <= 0 (15.4), [D.33].
- Bei Sqrt(x) ist x < 0 (15.4), [D.34].
- Das Ergebnis von Trunc(x) oder Short_Trunc(x) oder Long_Trunc(x) liegt nicht im Wertebereich des Ergebnistyps (Integer bzw. Short_Integer bzw. Long_Integer) (15.5), [D.35].
- Das Ergebnis von Round(x) oder Short_Round(x) oder Long_Round(x) liegt nicht im Wertebereich des Ergebnistyps (Integer bzw. Short_Integer bzw. Long_Integer) (15.5), [D.36].

- In einem Ausdruck der Form x**n ist
  - x von einem Integer-Typ und n < 0, oder
  - x = 0 bzw. x = 0.0 und n <= 0 (9.3.1).
- In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) vom Typ Long_Real nicht im Wertebereich der verallgemeinerten Variablen bzw. des Funktionsbezeichners (linke Seite) vom Typ Short_Real (10.1.2).
- Bei Wertparameterübergabe liegt der Wert des Aktualparameters vom Typ Long_Real nicht im Wertebereich des Formalparameters vom Typ Short_Real (8.5.1).
- In einem Aggregat liegt der Wert eines Aggregat-Elements vom Typ Long_Real nicht im Wertebereich der zugehörigen Aggregatkomponente vom Typ Short_Real (9.5).
- Beim Lesen aus einer Nicht-Text-Datei mit Read(f,v) liegt der Wert der Puffervariablen f↑ vom Typ Long_Real nicht im Wertebereich der Variablen v vom Typ Short_Real (15.1).
- Beim Schreiben in eine Nicht-Text-Datei mit Write(f,a) liegt der Wert des Ausdrucks a vom Typ Long_Real nicht im Wertebereich der Puffervariablen variablen f↑ vom Typ Short Rea_ (15.1).

---

**RANGE_ERROR**

---

- In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) von einem Ordinal-Typ nicht im Wertebereich des Typs der verallgemeinerten Variablen bzw. des Funktionsbezeichners (linke Seite) (10.1.2), [D.49].
- Bei Wertparameterübergabe liegt der Wert des Aktualparameters von einem Ordinal-Typ nicht im Wertebereich des Typs des Formalparameters (8.5.1), [D.7].
- In einem Aggregat liegt der Wert eines Aggregat-Elements von einem Ordinal-Typ nicht im Wertebereich des Typs der zugehörigen Aggregatkomponente (9.5).
- Beim Lesen aus einer Nicht-Text-Datei mit Read(f,v) liegt der Wert der Puffervariablen f↑ von einem Ordinal-Typ nicht im Wertebereich des Typs der Variablen v (15.1), [D.17].
- Beim Schreiben in eine Nicht-Text-Datei mit Write(f,a) liegt der Wert des Ausdrucks a von einem Ordinal-Typ nicht im Wertebereich des Typs der Puffervariablen f↑ (15.1), [D.18].
- Der Zeichenwert Chr(x) liegt nicht im Wertebereich des Typs Char (15.6), [D.37].

- Das Ergebnis von Succ(x) liegt nicht im Wertebereich des Typs von x (15.6), [D.38].

- Das Ergebnis von Pred(x) liegt nicht im Wertebereich des Typs von x (15.6), [D.39].
- Bei der Ausführung der Anweisung in einer FOR-Anweisung liegt der Anfangs- oder End-Wert der FOR-Anweisung nicht im Wertebereich des Typs der Laufvariablen (10.4.3),[D.52, D.53].
- Beim Lesen einer Integerzahl aus einer Textdatei (mit Read(f,x)) oder aus einem Zeichenkettenausdruck (mit Readstring(s,x)) liegt der Wert der Zahl nicht im Wertebereich der Variablen x und es ist kein Read_Error (s.u.) aufgetreten (15.1, 15.3), [D.55].
- Bei Write(f,a:l1:l2) bzw. Writestring(s,a:l1:l2) ist die Gesamtausgabelänge l1 < 1 oder die Anzahl der Ziffern nach dem Dezimalpunkt l2 < 1. Bei Write(f,a:l1) bzw. Writestring(s,a:l1) ist die Gesamtausgabelänge l1 < 1 bzw. l1 < 0 wenn a von einem String-Typ ist (15.1), [D.58].

---

**SET_ERROR**

---

- In einer Zuweisung liegt der Wert des Ausdrucks (rechte Seite) von einem Set-Typ nicht im Wertebereich des Typs der verallgemeinerten Variablen bzw. des Funktionsbezeichners (linke Seite) (10.1.2), [D.50].
- Bei Wertparameterübergabe liegt der Wert des Aktualparameters von einem Set-Typ nicht im Wertebereich des Typs des Formalparameters (8.5.1), [D.8].
- In einem Aggregat liegt der Wert eines Aggregat-Elements von einem Set-Typ nicht im Wertebereich des SET-Typs der zugehörigen Aggregatkomponente (9.5).
- Beim Lesen aus einer Nicht-Text-Datei mit Read(f,v) liegt der Wert der Puffervariablen f↑ von einem Set-Typ nicht im Wertebereich des Typs der Variablen v (15.1), [D.17].
- Beim Schreiben in eine Nicht-Text-Datei mit Write(f,a) liegt der Wert des Ausdrucks a von einem Set-Typ nicht im Wertebereich des Typs der Puffervariablen f↑ (15.1), [D.18].
- In einem Mengenbildner liegt der Wert einer Elementbestimmung nicht im Wertebereich des Basis-Typs des Mengenbildners (9.4).
- Bei Setmin(s) oder Setmax(s) ist der Wert des Ausdrucks s gleich der leeren Menge (9.4, 14.3.4, 15.6).

---

**STRING_ERROR**

- In einer Zuweisung ist die aktuelle Länge des Zeichenkettenwerts des Ausdrucks (rechte Seite) größer als die Maximallänge des String-Typs der Variablen bzw. des Funktionsbezeichners (linke Seite) (10.1.2).
- In einer Zuweisung ist die aktuelle Länge des Zeichenketten-Ausdrucks von einem String-Typ (rechte Seite) ungleich der Länge des Zeichenkettentyps fester Länge der verallgemeinerten Variablen bzw. des Funktions-Bezeichners (linke Seite) (10.1.2).
- Bei Wertparameterübergabe ist die aktuelle Länge der Zeichenkette des Aktualparameters größer als die Maximallänge des String-Typs des Formalparameters (8.5.1).
- Bei Wertparameterübergabe ist die aktuelle Länge der Zeichenkette des Aktualparameters ungleich der Länge des Zeichenkettentyps fester Länge des Formalparameters (8.5.1).
- In einem Aggregat ist die aktuelle Länge einer Zeichenkette eines Aggregat-Elements als die Maximallänge des String-Typs der zugehörigen Aggregatkomponente (9.5).
- In einem Aggregat ist die aktuelle Länge einer Zeichenkette eines Aggregat-Elements (von einem String-Typ) ungleich der Länge der zugehörigen Komponente des Aggregats (von einem Zeichenketten-Typ fester Länge) (9.5).
- Beim Lesen aus einer Nicht-Text-Datei mit Read(f,v) ist die aktuelle Länge der Zeichenkette von einem String-Typ in der Puffervariablen f↑ größer als die Maximallänge des Typs der String-Variablen v (15.1).
- Beim Lesen aus einer Nicht-Text-Datei mit Read (f,v) ist die aktuelle Länge der Zeichenkette von einem String-Typ in der Puffervariablen f↑ ungleich der Länge des Zeichenkettentyps fester Länge der Variablen v (15.1).
- Beim Schreiben in eine Nicht-Text-Datei mit Write(f,a) ist die aktuelle Länge der Zeichenkette größer als die Maximallänge des String-Typs der Puffervariablen f↑ (15.1).
- Beim Schreiben in eine Nicht-Text-Datei mit Write (f, a) ist die aktuelle Länge der Zeichenkette a von einem String-Typ ungleich der Länge der Puffervariablen f↑ von einem Zeichenkettentyp fester Länge (15.1).
- Bei Read(f,v) bzw. Readstring(s,v) ist die Maximallänge der String-Variablen v kleiner als die Länge der eingelesenen Zeichenkette (15.1, 15.3).
- Bei Readstring(a,v1,...,vn) enthält der Zeichenkettenausdruck a nicht soviele Zeichen, wie durch die Leseparameter v1,...,vn angefordert werden

(15.3).

- Bei Delete(s,i,l) ist i<1 oder l<0 oder (i+l-1) > Length(s) (15.3).

- Bei Insert(s1,s2,i) ist i<1 oder Length(s2) + Length(s1) > Maxlength(s2) (15.3).
- Bei Substring(s,i,l) ist i<1 oder l<0 oder (i+l-1) > Length(s) (15.3).
- Bei Pack(a,i,z) ist die Maximallänge der String-Variable z zu kleine, um alle Zeichen aus dem ungepackten Array a ab Index i aufzunehmen (15.8).

**INDEX_ERROR**

- Bei der Indizierung einer Array-Variablen, einer Array-Konstanten, eines Array-Aggregats oder eines Funktionsergebnisses von einem Array-Typ liegt der Wert des Index-Ausdrucks nicht im Wertebereich des Indextyps des Array-Typs (9.6.2), [D.1].
- Bei der Indizierung einer Variablen, einer Konstanten oder eines Funktionsergebnisses von einem String-Typ ist der Wert des Index-Ausdrucks kleiner als 1 oder größer als die aktuelle Länge der Zeichenkette (9.6.2).
- Bei einem Konformreihungsparameter ist der Indextyp des Aktualparameters nicht ein Teilbereich des Indextyps des Konformreihungsschemas (8.5.4), [D.59].
- Bei Pack(a,i,z) liegt der Wert des Ausdrucks i nicht im Wertebereich des Indextyps des ungepackten Array-Parameters a (15.8), [D.26].
- Bei Pack(a,i,z) wird beim Übertragen der Komponenten aus dem ungepackten
- Bei Unpack(z,a,i) liegt der Ordinalwert des Ausdrucks i nicht im Wertebereich des Indextyps des ungepackten Array-Parameters a (15.8), [D.29].
- Bei Unpack(z,a,i) ist der angegebene Bereich im ungepackten Array a ab Index i zu klein, um alle Komponenten des gepackten Array z aufzunehmen (15.8),[D.31].
- Bei Unpack(z,a,i) enthält der Zeichenkettenausdruck z mehr Zeichen, als in das ungepackte Array a ab Index i übertragen werden können (15.8).

**POINTER_ERROR**

- Bei einer Zeigerdereferenzierung ist der Wert der Variablen, der Konstanten oder des Funktionsergebnisses von einem Zeigertyp gleich Nil (9.6.4), [D.3].
- Beim Aufruf von Dispose(p) hat p den Wert Nil (15.2), [D.23].
- Beim Aufruf von Release(p) wurde der Verweiswert von p nicht durch einen Aufruf von Mark erzeugt (15.2).

### VARIANT_ERROR

– Es wird auf eine nicht aktive Variante einer Variablen, einer Konstanten, eines Aggregats oder eines Funktionsergebnisses von einem Record-Typ zugegriffen (9.6.3), [D.2].

### CASE_ERROR

– In einem Case-Statement entspricht keine Fallkonstante dem Wert des Fall-Index, und es ist auch keine Else-Alternative angegeben (10.3.2), [D.51].

### FILE_ERROR

– Vor dem Aufruf von Put(f), Write(f,...), Writeln(f,...) oder Page(f) wurde die Datei f nicht zum Schreiben eröffnet (15.1), [D.9].

– Vor dem Aufruf von Put(f), Write(f,...), Writeln(f,...) oder Page(f) ist die Datei f undefiniert (15.1), [D.10].

– Vor dem Aufruf von Put(f), Write(f,...), Writeln(f,...) oder Page(f) ist die aktuelle Dateiposition nicht die Dateiende-Position, d.h. Eof(f) ist False. Dieser Fehler kann nur im Zusammenhang mit [D.9] entstehen [D.11].

– Vor dem Aufruf von Get(f) oder Read(f,...) wurde die Datei f nicht zum Lesen eröffnet (15.1), [D.14].

– Vor dem Aufruf von Get(f) oder Read(f,...) ist die Datei f undefiniert (15.1), [D.15].

– Vor dem Aufruf von Read(f,...) ist die Puffervariable f↑ der Datei f undefiniert. Dieser Fehler kann nur im Zusammenhang mit [D.15] auftreten (15.1), [D.57].

– Vor dem Aufruf von Eof(f) ist die Datei f undefiniert (15.1), [D.40].

– Vor dem Aufruf von Eoln(f) ist die Datei f undefiniert (15.1), [D.41].

– Bei Assignfile(f,ext) ist die Beschreibung der externen Datei im Operanden "ext" fehlerhaft (15.1).

### EOF_ERROR

- Beim Aufruf von Get(f) oder Read(f,...) ist das Dateiende bereits erreicht, d.h. Eof(f) ist True (15.1), [D.16].
- Beim Aufruf von Eoln(f) ist das Dateiende bereits erreicht, d.h. Eof(f) ist True (15.1), [D.42]

### OPEN_ERROR

- Bei Aufruf von Reset oder Rewrite wurde die vordefinierte Textdatei Input oder Output angegeben (15.1).
- Beim Aufruf von Reset(f) ist die Datei f undefiniert (15.1) [D.13].
- Bei Reset(f) kann die an f gebundene, außerhalb des Programms liegende Datei nicht zum Lesen eröffnet werden (15.1).
- Bei Rewrite(f) kann die an f gebundene, außerhalb des Programms liegende Datei nicht zum Schreiben eröffnet werden (15.1).

### READ_ERROR

- Beim Lesen einer Integerzahl oder Realzahl aus einer Textdatei (mit Read (f,v)) oder aus einem Zeichenkettenausdruck (mit Readstring(s,v)) gilt:
  - die eingelesene Zeichenfolge ist syntaktisch fehlerhaft [D.54]
  - die eingelesene Zeichenfolge ergibt eine Zahl, die intern nicht mehr darstellbar ist. Bei Integerzahlen liegt der Wert außerhalb des Bereichs Long_Minit .. Long_Maxint und bei Realzahlen liegt der Wert außerhalb des Bereichs -Long_Maxreal .. Long_Maxreal.

### MEMORY_ERROR

- Die Ausführung des Programms kann wegen fehlenden Speichers nicht fortgesetzt werden (z.B. bei Aufruf eines Unterprogramms oder bei New) (13.3.3, 15.2).

### ELAB_ERROR

- Die Initialisierung der Pakete eines Programms kann nicht fortgesetzt werden, da durch die Verwendung der vordefinierten Prozedur Elaborate Zyklen bei der Initialisierung entstehen (13.3.3, 15.12).

### Sonstige Fehler (ohne zugeordneter Fehlernummer)

Die in der folgenden Tabelle angegebenen Fehler haben keine zugeordnete Fehlernummer. Das Auftreten eines solchen Fehlers wird nicht erkannt und führt i.a. zu Folgefehlern mit undefinierten Auswirkungen.

- In einer Zuweisung ist der Typ des Ausdrucks (rechte Seite) vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamische Variable, deren Typ vom Domänentyp des Typs der verallgemeinerten Variablen bzw. des Funktions-Bezeichners (linke Seite) verschieden ist (10.1.2).
- Bei Wertparameterübergabe ist der Typ des Aktualparameters vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamsiche Variable, deren Typ vom Domänentyp des Typs des Formalparameters verschieden ist (8.5.1).
- In einem Aggregat ist der Typ eines Zeigerwertes vom generischen Zeigertyp und der Zeigerwert des Ausdrucks verweist auf eine dynamische Variable, deren Typ vom Domänentyp des Typs der entsprechenden Aggregatkomponente verschieden ist (9.5).
- Die Länge einer String-Variablen wird verändert, ovwohl noch eine Referenz auf eine Komponente der String-Variablen existiert (9.6.2).
- Bei Writestring(s,p1,...,pn) enthält einer der Schreibparameter p1,...,pn eine Referenz auf die String-Variable s (15.3).
- Bei Convert(x,t) repräsentiert die Speicherdarstellung von x keinen zulässigen Wert des Typs t (15.10).
- Die Variante einer Record-Variablen ist nicht für die Gesamtdauer jeglicher Referenz auf jede ihrer Komponenten aktiv (9.6.3), [D.2].
- Der Dateizeiger einer Dateivariablen f wird (z. B. durch Lesen oder Schreiben) geändert, obwohl noch eine Referenz auf die Puffervariable f↑ existiert (9.6.5), [D.6].
- Bei einer Zeigerdereferenzierung ist der Wert der Variablen, der Konstanten oder des Funktionsergebnisses von einem Zeigertyp undefiniert (9.6.4), [D.4].

- Mit Dispose(q) wird ein Verweiswert auf eine dynamische Variable entfernt, obwohl noch eine Referenz auf die dynamische Variable existiert (15.2), [D.5].
- Beim Aufruf von Dispose(p) ist der Wert von p undefiniert (15.2), [D.24].
- Vor dem Aufruf von Dispose(p) ist p↑ durch New(q,c1,...,cn) oder New(q,c1,...,cn,e) oder New(q,e) erzeugt worden (15.2), [D.20].
- Vor dem Aufruf von Dispose(p,k1,...,km) ist die dynamische Variable p↑ durch New(p,c1,...,cn) erzeugt worden, wobei m ungleich n ist (15.2), [D.21].
- Bei Dispose(p,c1,...,cn) oder Dispose(p,c1,...cn,e) sind in der dynamischen Variablen p↑ andere Varianten eingestellt, als durch die Auswahlkonstanten c1 bis cn angegeben (15.2), [D.22].
- Vor dem Aufruf von Dispose(p,e) ist p↑ durch New(q,a) erzeugt worden, wobei a ungleich e ist. Analog gilt dies auch für Dispose(p,c1,...,cn,e) und New(q,k1,...kn,a) (15.2).
- Bei Dispose (p, e) oder Dispose (p, c1,..., cn, e) liegt der Wert von e nicht im Wertebereich des Indextyps des entsprechenden ARRAY-Typs bzw. ist kleiner als 1 oder größer als die Maximallänge des entsprechenden String-Typs (15.2).
- Bei einem indizierten ARRAY- oder String-Objekt wurde das ARRAY- bzw. String-Objekt verkürzt durch den Aufruf von New(p,e) oder New (p, c1, .., cn, e) erzeugt und der Wert des Indexausdrucks im indizierten Objekt ist größer als e (9.6.2).
- In einem indizierten String-Objekt ist der Wert des String-Objekts nicht defieniert (unabhängig davon, ob das indizierte Objekt in einem Ausdruck oder z.B. als verallgemeinerte Variable auf der linken Seite einer Zuweisung auftritt).
- In einer mit New(p,c1,...,cn) oder New (p,c1,...,cn,e) erzeugten dynamischen Variable wird eine andere Variante eingestellt, als die durch die Selektorkonstanten c1 bis cn bestimmte (9.6.3, 15.2),[D.19].
- Einer dynamischen String-Variablen, die mit New(p, e) erzeugt wurde oder an die lezte Komponente einer dynamischen String-Variablen, die mit New (p, c1,..., cn, e) erzeugt wurde, wird eine Zeichenkette zugewiesen, die länger ist als e (15.2).
- Eine dynamische Variable, die durch New(p,c1,...,cn), New(p,e) oder New(p,c1,...,cn,e) erzeugt wurde, kommt als Ganzes in einem Ausdruck oder als linke Seite in einer Wertzuweisung vor, oder wird als Parameter übergeben (9.6.4, 15.2), [D.25].

- Bei New (p, e) oder New (p, c1,..., cn, e) liegt der Wert von e nicht im Wertebereich des Indextyps des entsprechenden ARRAY-Typs bzw. ist kleiner als 1 oder größer als die Maximallänge des entsprechenden String-Typs (15.2).
- Der beim Aufruf von Release (p) übergebene Verweiswert p wurde durch einen anderen Aufruf von Release (q) vernichtet (15.2).
- Vor dem Aufruf von Put(f), Put(f,c1,...,cn), Put(f,e) oder Put(f,c1,...,cn,e) ist die Puffer-Variable f↑ undefiniert (15.1), [D.12].
- Bei der verkürzten Ausgabe eines Arrays mit Put(f,e) oder Put (f,c1,...,cn,e) liegt der Wert des Indexausdrucks e nicht im Wertebereich des Indextyps des Arrays (15.1).
- Bei der verkürzten Ausgabe einer Zeichenkette variabler Länge mit (Put (f,e) oder Put(f,c1,...,cn,e) ist der Wert des Indexausdrucks e kleiner als 1 oder größer als die aktuelle Länge der Zeichenkette (15.1).
- Eine verallgemeinerte Variable, die als Objekt in einem Ausdruck verwendet wird, hat zum Zeitpunkt der Auswertung des Ausdrucks einen undefinierten Wert (9.1), [D.43].
- Das Ergebnis einer Funktion ist nach Ausführung des Funktionsblocks undefiniert, da dem Funktionsbezeichner kein Wert zugewiesen wurde (8.7), [D.48].
- Bei Unpack(z,a,i) ist irgendeine Komponente des gepackten Arrays z undefiniert (15.8), [D.30].
- Bei Pack(a,i,z) wird auf eine Komponente des ungepackten Arrays a zugegriffen, die undefiniert ist (15.8), [D.27].
- Die Bezeichner der Programmparameter des Hauptprogramms und aller dazugehörigen Pakete sind, mit Ausnahme von Input und Output, nicht paarweise verschieden (13.3.3).
- Die Namen aller zu einem Programm gehörigen Pakete und der Name des Hauptprogramms sind nicht paarweise verschieden (13.3.3).

# Implementierungsdefinierte Eigenschaften

Die implementierungsdefinierten Eigenschaften können für jede Pascal-XT Implementierung unterschiedlich sein, sie sind aber für jede Implementierung definiert. Ein Programm kann sich auf implementierungsdefinierte Werte oder Eigenschaften stützen, was aber zu verschiedenen Ergebnissen eines Programms auf verschiedenen Implementierungen führen kann. Ein einfaches Beispiel dafür ist die Ausgabe des Wertes Maxint, der abhängig von der Implementierung Short_Maxint oder Long_Maxint sein kann.

Im Benutzerhandbuch [1] sind alle hier aufgeführten implementierungsdefinierten Eigenschaften spezifiziert. Die Querverweise in den runden Klammern beziehen sich auf die Kapitel in dieser Sprachbeschreibung, die Angaben in den eckigen Klammern auf die Kapitel in der Pascal-Norm, sofern diese Eigenschaften dort erwähnt sind.

Die in Punkt 15 angegebene Eigenschaft ist für alle Pascal-XT Implementierungen gleich definiert.

1) Die Werte folgender vordefinierter Real-Konstanten sind implementierungsdefiniert (5.2):

   Short_Minreal
   Long_Minreal
   Short_Maxreal
   Long_Maxreal

2) Folgende vordefinierte Konstanten haben implementierungsdefinierte Werte (5.2),[6.7.2.2]:

```
Maxint    ist gleich Short_Maxint oder Long_Maxint
Minint    ist gleich Short_Minint oder Long_Minint
Maxreal   ist gleich Short_Maxreal oder Long_Maxreal
Minreal   ist gleich Short_Minreal oder Long_Minreal.
```

3) Für die vordefinierten Typbezeichner Integer und Real gilt implementierungsdefiniert (6.2.1):

```
Integer ist gleich Short_Integer oder Long_Integer
Real    ist gleich Short_Real oder Long_Real.
```

4) Die Werte der Typen Short_Real und Long_Real stellen implementierungsdefinierte Teilmengen der reellen Zahlen dar (6.2.2), [6.4.2.2].

5) Die Ergebnisse von arithmetischen Real-Operatoren und Real-Funktionen sind Näherungswerte der mathematischen Ergebnisse. Die Genauigkeit dieser Näherungen ist implementierungsdefiniert (6.2.2), [6.7.2.2].

6) Die Werte des Typs Char ergeben sich durch Aufzählung der implementierungsdefinierten Zeichen und die Zuordnung von Ordinalzahlen vom Typ Integer zu den Zeichenwerten ist implementierungsdefiniert (6.2.3), [6.4.2.2].

7) Die Maximallänge des String-Typs ohne Angabe eines Typ-Parameters ist implementierungsdefiniert (6.3.2.2).

8) Die Größe einer Speichereinheit ist implementierungsdefiniert. Sie kann 1 Byte oder ein Vielfaches (i.a. eine Zweierpotenz) von einem Byte sein (6.3.3.2).

9) Für die Abstands- und Bitbereichs- Angabe in Feldbezeichnern eines RECORD-Typs können implementierungsdefinierte Einschränkungen gelten (6.3.3.2).

10) Die maximale Anzahl der Werte des Basistyps einer Menge kann implementierungsdefiniert beschränkt sein (6.3.4).

11) Der größte Ordinalwert des Basistyps eines nicht qualifizierten Mengenbildners ist implementierungsdefiniert (9.4).

12) Es ist implementierungsdefiniert, welche der Direktiven C, Cobol, Fortran, External und Internal unterstützt werden (8.6).

13) Bei Unterprogrammen mit einer der Direktiven C, Cobol, Fortran, External und Internal kann es bez. der Parameterart, den Parametertypen und der Parameteranzahl implementierungsdefinierte Einschränkungen geben (8.6).

14) In Pascal werden die logischen Auswirkungen der Dateioperationen beschrieben. Die physikalischen Aktivitäten und der Zeitpunkt ihrer Ausführung sind implementierungsdefiniert (19), [6.6.5.2].

15) Die Beschreibung der externen Datei in der vordefinierten Prozedur Assignfile und die Wirkung dieser Prozedur sind implementierungsdefiniert (15.1).

16) Die Wirkung der vordefinierten Prozeduren Reset bzw. Rewrite auf eine der vordefinierten Textdateien Input oder Output ist nach der Pascal-Norm implementierungsdefiniert.
In Pascal-XT ist diese Eigenschaft für alle Implementierungen definiert:
es tritt ein Open_Error auf (15.1),[6.10].

17) Die Standard-Ausgabelängen für Werte eines Integer-Typs, eines Real-Typs und des Typs Boolean sind implementierungsdefiniert (15.1), [6.9.3.1].

18) Die Darstellung des Exponentenzeichens (als ’E’ oder ’e’) und die Anzahl der Dezimalziffern für den Exponenten bei Ausgabe von Real-Werten in Gleitpunktdarstellung sind implementierungsdefiniert (15.1), [6.9.3.4.1].

19) Die Schreibweise (Groß-/Kleinschreibung) Boolescher Werte bei der Ausgabe ist

für jeden Buchstaben implementierungsdefiniert (15.1), [6.9.3.5].

20) Die Wirkung der vordefinierten Prozedur Page auf Textdateien ist implementierungsdefiniert (15.1), [6.9.5].

21) Für Entry-Unterprogramme kann es implementierungsdefinierte Einschränkungen geben (11.2.1).

22) Für Programmparameter, deren Variable einen FILE-Typ besitzen, ist die Zuordnung an Objekte außerhalb des Programms implementierungsdefiniert (11.5), [6.10].

23) Die Voreinstellungen der Compileroptionen sind implementierungsdefiniert (16).

# Implementierungsabhängige Eigenschaften

Die implementierungsabhängigen Eigenschaften können sich in den verschiedenen Pascal-Implementierungen unterscheiden und sind mit Ausnahme der in Punkt 8 erwähnten Eigenschaft, die für alle Pascal-XT Implementierungen gilt, **nicht notwendigerweise definiert**.

Ein Programm, das sich auf bestimmte implementierungsabhängige Eigenschaften stützt, erfüllt nicht die Norm und ist in Pascal-XT fehlerhaft, sofern die Eigenschaft nicht implementierungsdefiniert ist (siehe Punkt 8).

Die Querverweise in den runden Klammern beziehen sich auf die Kapitel in dieser Sprachbeschreibung, die Angaben in den eckigen Klammern auf die Kapitel in der Pascal-Norm.

1) Die Reihenfolge der Auswertung von Index-Ausdrücken in einer indizierten Variable, in einer indizierten Konstanten, in einem indizierten Aggregat oder bei einem indizierten Funktionsaufruf ist implementierungsabhängig (9.6.2), [6.5.3.2].

2) Die Reihenfolge der Auswertung der Elementbestimmungen und der Ausdrücke in den Elementbestimmungen in einem Mengenbildner ist implementierungsabhängig (9.4), [6.7.1].

3) Mit Ausnahme der Kurzschlußoperatoren OR ELSE und AND THEN ist die Reihenfolge der Berechnung der Operanden eines dyadischen Operators implementierungsabhängig. Die Operanden können in der Reihenfolge ihrer Aufschreibung, in umgekehrter Reihenfolge, gleichzeitig oder möglicherweise überhaupt nicht ausgewertet werden (9.3), [6.7.2.1].

4) Die Reihenfolge der Auswertung der Ausdrücke und der Zuordnung zu den Komponenten eines Aggregats ist implementierungsabhängig (9.5).

5) Die Reihenfolge des Zugriffs, der Auswertung und der Zuordnung von Aktualparametern bei einem Prozeduraufruf bzw. Funktionsaufruf ist implementierungsabhängig (8.7), [6.7.3, 6.8.2.3].

6) Die Reihenfolge des Zugriffs auf die Variable (linke Seite) bzw. der Auswertung des Ausdrucks (rechte Seite) in einer Zuweisung ist implementierungsabhängig (10.1.2), [6.8.2.2].

7) Die Wirkung beim Lesen einer Textdatei, während deren Generierung die vordefinierte Prozedur Page angewendet wurde, ist implementierungsabhängig. Für eine Implementierung kann aber die Wirkung definiert sein (15.1), [6.9.5].

8) Für Programmparameter, deren Variable keinen FILE-Typ besitzen, ist die Zuordnung an Objekte außerhalb des Programms implementierungsabhängig (11.5), [6.10].
In Pascal-XT kann eine Variable nicht in der Programmparameterliste angegeben werden, wenn sie nicht einen FILE-Typ besitzt.

# Paket CLOCK

Das Paket CLOCK liefert Angaben über Zeit, Datum und verbrauchte Prozessorzeit.

```
package CLOCK;

              (*************************************)
              (*  The body of this package is part   *)
              (*  of the Pascal-XT runtime system    *)
              (*************************************)

type

  date_string = packed array [1..12] of char;   (* ISO Date : 'yy-mm-ddjjj ' *)
  time_string = packed array [1.. 8] of char;   (* ISO Time : 'hh:mm:ss'     *)
  cpu_seconds = real;                           (* cpu time in seconds       *)

function date : date_string;
   (* returns the actual date *)

function time : time_string;
   (* returns the actual time *)

function cpu_time : cpu_seconds;
   (* returns the task's used cpu time *)

end (* package CLOCK *).
```

**DATE**

liefert das aktuelle Datum im ISO-Format (yy-mm-ddjjj), wobei jjj die fortlaufende Numerierung der Tage des Jahres angibt.

**TIME**

liefert die augenblickliche Uhrzeit im ISO-Format (hh:mm:ss).

**CPU_TIME**

liefert die seit Prozeßstart verbrauchte CPU-Zeit in Sekunden. Die CPU-Zeit wird in einer implementierungsdefinierten Schrittweite weitergezählt.

# Literatur

Mit * markierte Titel sind nicht von der Siemens Nixdorf Informationssysteme AG oder der Siemens AG herausgegeben.

[1] **Pascal-XT (BS2000)**
Benutzerhandbuch

*Zielgruppe*
Pascal-XT-Anwender unter BS2000.

*Inhalt*
Bedienung des Programmiersystems und des Compilers;
Beschreibung der BS2000-spezifischen Eigenschaften des Compilers;
Binden und Ausführen von Programmen;
Sprachanschlüsse;
Die Testhilfe PATH;
Laufzeitfehlermeldungen;
Beschreibung vordefinierter Pakete;
Vergleich mit Pascal Version 3.

[2] **Pascal-XT (Sinix)**
Benutzerhandbuch

*Zielgruppe*
Pascal-XT-Anwender unter Sinix.

*Inhalt*
Bedienung des Compilers;
Beschreibung der Sinix-spezifischen Eigenschaften des Compilers;
Pascal-Dateien und deren Bindung an Sinix-Dateien;
Sprachanschlüsse;
Die Testhilfe PATH;
Beschreibung vordefinierter Pakete.

[3]* Däßler/Sommer
Pascal - Einführung in die Sprache,
DIN-Norm 66256
Springer-Verlag, Berlin, Heidelberg
New York, Tokio 1983/85
ISBN 3-540-12835-2

[4]* K. Jensen, N. Wirth
Pascal user manual and report
Springer-Verlag, Berlin, Heidelberg
New York, Tokio 1974

[5]* Jacques Tiberghien
Das Pascal Handbuch
Sybex-Verlag, 1982
ISBN 3-887 45-005-1

*Anmerkung*

Dieses Buch ist als Nachschlagewerk gut geeignet. Es bietet dem Programmierer einen Überblick über die am meisten verbreiteten Pascal-Versionen, und somit Vergleichsmöglichkeiten zu dem in diesem Manual beschriebenen Pascal-XT.

### Bestellen von Handbüchern

Die aufgeführten Handbücher finden Sie mit ihren Bestellnummern im *Druckschriftenverzeichnis Datentechnik*. Dort ist auch der Bestellvorgang erklärt. Neu erschienene Titel finden Sie in den *Druckschriften-Neuerscheinungen Datentechnik*.

Beide Veröffentlichungen erhalten Sie regelmäßig, wenn Sie in den entsprechenden Verteiler aufgenommen sind. Wenden Sie sich bitte hierfür an eine Geschäftsstelle unseres Hauses.

# Stichwörter

**A**
Abs 282
Abstand 51, **55**
Addition 120
Aggregat 138, 143
  statisches 138
  Verwendung 138
aktive Variante 149
Aktualparameter 111, 164
Aktualparameterliste 111, 164
Alignof 75, 295
Anfangswert 179
Anweisung 157
  bedingte 169
  einfache 158
  Einteilung 157
  Verbund- 168
  Wiederholungs- 166
Anweisungsfolge 168
Anweisungsteil 87, 89, 190, **203**
Any_File 65
Any_Type 66
Arctan 282
arithmetische Funktion 281
arithmetische Operatoren 120
ARRAY-Aggregat 139
ARRAY-Aggregat-Element 139
ARRAY-Typ 46, 267
Assembler 307
Assignfile 234
Attribut-Funktion 294
Aufzählungskonstante
  Definitionspunkt 207
  Gebiet 207
Aufzählungstyp **42**

Ausdruck 115
  Auswertung 115
  Auswertungsreihenfolge 116, 127
  boolescher 169, 175, 177
  einfacher 115
  statischer 27, **118**, 145, 149, 275, 281
Ausnahme 115, 221
  behandelte 224
  benutzerdefinierte 222
  Erkennung 223
  propagieren 224, 226, 229f, 300
  Repräsentation 222
  reservierte 222
Ausnahmebehandlung 221
Ausnahmebehandlungsteil 221, 224
Ausnahmesituation 221, 300
Ausrichtung 75
Auswahl 172
Auswahlkonstanten 52
Auswahlliste **51f**, 172

**B**
Backus-Naur-Form 9
Basistyp 58
Bearbeitungsmodus 60
bedingte Anweisungen 169
benutzerdefinierte Ausnahmen 222
Bezeichner 15
  Definitionspunkt **203**, 206
  Feld- 51, 148
  Funktions- **89**, 160, 196
  Gültigkeitsbereich 209
  Grenz- **107**
  importierter **196**, 198
  Konstanten- 27, 41, 42, 196, 198
  Paket- 27, 190, 196, 198
  Programmname187
  Programmparameter 200
  Prozedur- 87, 196
  Schreibweise 2
  Syntax 1 5
  Typ- **35**, 196
  Variablen- 62, 179, 196
  Verwendung **203**, 209

voll qualifizierter 197
vordefinierte 378
Bezeichnerliste 78
Bitbereich 51, 55
Bitsizeof 75, 295
Block 87, 89, 167, 187, **203**
ausführen 218
Funktions- 89
Hauptprogramm- 187
Prozedur- 87
Boolean **41**
boolesche Operatoren 127
boolescher Ausdruck 169, 175, 177
Break_Error 223

**C**
C 108
Card 286
CASE-Anweisung 169, 172
Case_Error 222
Char **40**
Char-Typ 245, 277, 280, 286
Check 307
Chr 286
Cobol 108
Compileroptionen 303
Concat 272
Convert 299
Cos 282

**D**
Datei 60
aktuelle Position 60
allgemeine 60
Bearbeitungsmodus 60
externe 331
leere 60
lokale 330
Text- 61
Dateizeiger **60**, 154, 183, 330
Debug 305
definierte Variable 82
Definitionen, Reihenfolge 203
Definitionspunkt 203, 206
Aufzählungskonstante 207

Feld-Bezeichner 207
Feldauswahl-Bezeichner 208
Funktions-Bezeichner 206
Grenz-Bezeichner 208
importierter Bezeichner 207
Konstanten-Bezeichner 206
Marken 206
Paket-Bezeichner 206
Parameter-Bezeichner 207
Prozedur-Bezeichner 206
Typ-Bezeichner 206
Variablen-Bezeichner 206
vordefinierte Bezeichner 208
Definitionsteil
Konstanten- 27, 190
Typ- 35, 190
Deklaration
Funktions- **89**
Prozedur- 87
Reihenfolge 203
Deklarationsteil
Marken- 25
Variablen- 78, 190
deklarierte Variable 80
Delete 273
dereferenziertes Objekt 143, **152**
Direktieren 108
Direktive **17**, 108, 193
Disjunktion 127
Dispose 261, 359
DIV 125
Division 120
ganzzahlige 120
Domänentyp **63**, 199, 359
dynamische Variable 63, 77, 80, 152, 359, 360

**E**
einfache Anweisungen 158
einfacher Ausdruck 115
einfacher Typ 37
Elab_Error 223
Elaborate 302
Elementebestimmung 135
ELSE-Teil 169

Endwert 179
ENTRY 193
ENTRY-Funktion 94, 190, **193**
ENTRY-Prozedur 94, 190, **193**
Eof 235, 330
Eof_Error 222
Eoln 61, 235, 342
Ergebnistyp 89
Error_Number 221
Ersatzdarstellung 14
EXCEPTION-Teil 168, 224
EXIT-Anweisung 158, 166
Exp 282
Exponential-Operator 116
Exponentiation 120
Exponentionaloperator 119
External 108
externe Datei 331

**F**
Faktor 115
Fall-Liste 172
Fall-Listenelement 172
Fallindex 172
Fallkonstante 172
False 30, 41
Fehler **12**, **221**
  Laufzeit- 12
  Übersetzungs- 12
Feld
  Abstand 55
  Bitbereich 55
  Zugriff 51
Feld-Bezeichner **51**, 148, 182
  Definitionspunkt 207
  Gebiet 207
Feldauswahl 148
Feldauswahl-Bezeichner **148**, 183
  Definitionspunkt 208
  Gebiet 208
Felddarstellung 51
Felddarstellungsliste 51
Feldliste 51
  leere 51

Festteil 51
FILE-Objekt 153
FILE-Typ 60, 71, 234, 330
File-Typ, generischer 65
File_Error 222
First 75, 296
FOR-Anweisung 179
Formalparameter 164
Formalparameterabschnitt 95
Formalparameterliste 87, 89, 95
  Wiederholung 87, 90, 95, 195
Formatangabe 111
Fortran 108
Forward 108
Funktion 85
  arithmetische 281
  ausführen 219
  Entry- 94
  Inline- 93
  statische 118
Funktions-Bezeichner **89**, 160, 196
  Definitionspunkt 206
  Gebiet 206
Funktions-Block 89
Funktions-Name **89**, 101, 111, 211
Funktionsaufruf **111**, 143
Funktionsdeklaration **89**, 203
Funktionsidentifikation 89
Funktionskopf **89**, 101
Funktionswert 89

**G**

Gültigkeitsbereich 203, 209
  WITH-Anweisung 182
Ganzvariable 77
ganzzahlige Division 120
Gebiet 203, **206**
  Aufzählungskonstante 207
  Feld-Bezeichner 207
  Feldauswahl-Bezeichner 208
  Funktions-Bezeichner 206
  Grenz-Bezeichner 208
  importierter Bezeichner 207
  Konstanten-Bezeichner 206

Marken 206
Paket-Bezeichner 206
Parameter-Bezeichner 207
Prozedur-Bezeichner 206
Typ-Bezeichner 206
Variablen-Bezeichner 206
vordefinierte Bezeichner 208
Generate 305
generischer FILE-Typ 65
generischer Zeigertyp 65, 265
gepackter strukturierter Typ 45
Get 236, 335
GOTO-Anweisung158, 165
Grenz-Bezeichner 104, **107**, 115
Definitionspunkt 208
Gebiet 208
Großschreibung 13

**H**
Hauptprogramm 187
Hauptprogramm-Block 187, 331
Hexadezimalzahl 19

**I**
Identität von Typen 67
IF-Anweisung 169
Implementierungsabhängig **11**
Implementierungsdefiniert **11**
importierter Bezeichner **196**, 198
Definitionspunkt 207
Gebiet 207
IN 131
Index_Error 222
Indextyp 46
umfassender 104
indiziertes Objekt 143
indiziertes Objekt **145**
Indizierung 145
Kurzform 145
Langform 145
Initialisieren eines Paketes 302
Initialize 308
INLINE 193
Inline-Funktion 93, 190, **193**
Inline-Prozedur 93, 190, **193**

Input 200, 343
Input **62**
Insert 274
Integer **37**
Integer-Operationen 121
Integer-Typ **37**, 71, 245, 277, 280, 285, 287, 300
Internal 108

**K**
Kennungsfeld 51
Kennungstyp 51f
Kleinschreibung 13
Kommentar 23
Komponententyp 46, 60
Komponentenvariable 77
Konformität 106
Konformreihungs-Parameter 95, **104**
Konformreihungs-Schema 104
  mehrdimensionales 105
Konjunktion 127
Konstante, vorzeichenlose 115
Konstanten 118
Konstanten-Bezeichner **27**, 41, 42, 196, 198
  Definitionspunkt 206
  Gebiet 206
Konstanten-Name **27**, 115, 143, 211
Konstantendefinition 27
Konstantendefinitionsteil 27, 190, 203
Kontext-Spezifikation 187, 190f, 196
Kursivschrift 3
Kurzschluß-Operatoren 127
  Anwendungsbereich 127
  Auswertung 127

**L**
Last 75, 296
Laufvariable 179
  Gefährdung 180
Laufzeitfehler 12
Leeranweisung 158
leere Datei 60
leere Menge 135
leere Zeichenkette 49
Length 272, 275
Lexikalische Einheiten 23

lexikographischer Vergleich 133
List 308
Listen, strukturierte 362
Ln 283
lokale Datei 330
Long 121, 284
Long_Integer **37**, 69
Long_Maxint 30, 37
Long_Maxreal 30
Long_Minint 30, 37
Long_Minreal 30
Long_Real **38**, 69, 71

**M**

Map 306
Mark 265, 359
Marke **20**, 25, 165
  Definitionspunkt 206
  Gültigkeitsbereich 209
  Gebiet 206
  Verwendung 209
Markendeklarationsteil 25, 203
Maximallänge 49
Maxint 30, 37
Maxlength 75, 296
Maxreal 30
Memory_Error 223
Menge
  Größe 135
  leere 135
Mengenbildner 58, 135
  nicht qualifizierter 135
  qualifizierter 135
Mengendifferenz 129
  symmetrische 129
Mengendurchschnitt 129
Mengenoperator 129
Mengenvereinigung 129
Metabezeichner 2f, **9**
Metasymbole 9
Minint 30, 37
Minreal 30
MOD 125
Modulo 120

Multiplikation 120
Multiplikationsoperator 115, 119

**N**
Name 3
  Funktions- **89**, 101, 211
  Konstanten- **27**, 211
  Prozedur- **87**, 101, 211
  Typ- **35**, 211
  Variablen- **78**
Negation 127
neuer Typ 35, 67
New 266, 359
nichtterminales Symbol 9
NIL-Wert 63, 359
NOT - Operator 119
Numeric_Error 222

**O**
Objekt 143
  dereferenziertes 143, **152**
  FILE- 153
  indiziertes 143, **145**
  selektiertes 143, **148**
  Variablen- 77, **143f**, 160
  Zeiger- **152**
Offsetof 296
Open_Error 222
Operand 116
Operationen
  Integer- 121
  mit Real-Konstanten 122
  Real- 122
Operatoren 119
  arithmetische 120
  boolesche 127
  dyadische arithmetische 120
  Mengen- 129
  monadische arithmetische 120
  Präzedenzregeln 119
  Vergleichs- 131
Optimize 307
Optionen 303, 305
Ord 287
Ordinaltyp 46, 58, 71, 287f

Ordinaltyp **37**
Ordinalzahl 37, 287
Output **62**, 200, 343

**P**


Page 61, 237, 308
Paket 190
  Definitionsteil 191
  Deklarationsteil 191
  initialisieren 302
  Initialisierung 220
  nicht sichtbarer Teil 195
  sichtbarer Teil 193
Paket-Bezeichner 27, 78, 87, 89, 111, **190**, 196, 198
  Definitionspunkt 206
  Gebiet 206
Paket-Block 191, 331
Paket-Implementierung 190, 195
  leere 190, 195
Paket-Spezifikation 190, 193
Parameter 95
  aktuelle 95
  formale 95
  Übergabe 164
Parameter-Bezeichner
  Definitionspunkt 207
  Gebiet 207
Parameterfunktion 95, **101**
Parameterprozedur 95, **101**
Pointer 65, 265
Pointer_Error 222
Position 276
Präzedenzregeln 119
Pred 287
Primitiv 115, 116
privater Zeigertyp 199
Programm 187
  ausführen 220
  in Norm-Pascal 213
  in Pascal-XT 213
  initialisieren 220
Programmierung, unüberprüfte 299
Programmname187
Programmparameter 191, 200
  verschiedene 200
Programmparameterliste 187, 190
Programmstruktur 213
Programmtext 213

propagieren 300
Prozedur 85
  ausführen 219
  Entry- 94
  Inline- 93
Prozedur-Bezeichner **87**, 196
  Definitionspunkt 206
  Gebiet 206
Prozedur-Block 87
Prozedur-Name **87**, 101, 111, 164, 211
Prozeduraufruf 111, 158, 164
Prozedurdeklaration 87, 203
Prozeduridentifikation 87
Prozedurkopf 87, 101
Pseudokommentar 23, 303
Puf 337
Puffervariable 60, 77, **80**, 143, 153
Put 238

**Q**
qualifizierter Mengenbildner 135

**R**
Raise 221, 300
Range_Error 222
Read 242, 338, 351, 353
Read_Error 222
Readln 61, 249
Readstring 277
Real **38**
Real-Konstanten 122
Real-Operationen 122
Real-Typ **38**, 71, 245, 277, 280, 285
  universeller 122, 281
RECORD-Aggregat 141
RECORD-Aggregat-Element 141
RECORD-Typ 51
Record-Variablen-Liste 182
Record-Variablen-Objekt 182
reelle Zahl 38
Rekursion 85
Release 271, 359
REPEAT-Anweisung 175
reservierte Ausnahmen 222
Reset 250, 334

RETURN-Anweisung 158, 167
Rewrite 251, 334
Round 284

**S**
Sedezimalzahl 19
Seiteneffekte 90, 152, 154, 162
selektiertes Objekt 143, **148**
Selektion 148
Semantik 2
SET-Typ **58**, 69, 71, 129
Set_Error 222
Setmax 288
Setmin 288
Short_Integer **37**, 69, 275, 284
Short_Maxint 30, 37
Short_Maxreal 30
Short_Minint 30, 37
Short_Minreal 30
Short_Real **38**, 69, 71
Sin 283
Sizeof 75, 297
Sonderzeichen 14
Speicherbedarf 75
Speichereinheit 75
Spezialsymbole 14, 381
Spezifikation
  Variablenparameter- 98
  Wertparameter- 96
Sqr 283
Sqrt 283
Standard 306
Standardfunktion 233
Standardprozedur 233
statische Funktion 118
statischer Ausdruck 27, **118**, 145, 149, 275, 281
Steueranweisung 23, 303
String **49**
String-Typ **49**, 71, 245, 277
  Maximallänge 49
String_Error 222
strukturierter Typ 45
Substring 279
Subtraktion 120

Succc 288
Summationsoperator 115, 119
Symbol
  nichtterminales 9
  Spezial- 14
  terminales 9
  Wort- 14
symmetrische Mengendifferenz 129
Syntax 2
System_Error 223

**T**
Teilbereich 69
Teilbereichstyp **44**, 116
terminales Symbol 9
Text 342
Textdatei 61
Texttyp 280
Trenner 23
True 30, 41
Trunc 285
Typ
  ARRAY- 267
  Attribut 75
  Aufzählungs- 42
  Domänen- **63**
  einfacher 37
  FILE- 71
  FILE-Typ 234, 330
  gepackter 45
  gepackter strukturierter 45
  Identität 67
  neuer 35, 67
  Ordinal- **37**
  strukturierter 45
  Teilbereichs- 44
  ungepackter 45
  ungepackter strukturierter 45
  verträglicher 69
  Wirts- 44
  Zeichenketten- **48**
  Zeiger- **63**
Typ-Bezeichner **35**, 196
  Definitionspunkt 206

Gebiet 206
vordefinierte 37
Typ-Name **35**, 51, 63, 67, 89, 96, 98f, 104, 111, 135, 139, 141, 211
Typangabe **35**, 46, 51, 58, 60, 67, 78
Typdefinition 35
Typdefinitionsteil 35, 190, 203
Typverträglichkeit 69

**U**
umfassender Indextyp 104
Umwandlungsfunktion 285
unüberprüfte Programmierung 299
undefinierte Variable 82
ungepackter strukturierter Typ 45
universeller Real-Typ 122, 281
Unterprogrammaufruf 111
Unterprogramme 85
vordefinierte 233
USE-Liste 196, **198**

**Ü**
Übersetzungseinheiten 216
abhängige 216
Abspeicherung 216
Beziehungen 196
Neuübersetzung 217
Übersetzungsfehler 12
Übersetzungsreihenfolge 216

**V**
Variable
definierte 82
deklarierte 80
dynamische 63, 77, 80, 152, 360
gültiger Wert 82
Ganz- 77
Komponenten- 77
Lebensdauer 218
nicht initialisierte 83
Puffer- 77, **80**
undefinierte 82
verallgemeinerte **77**, 98, 105, 116, 118, 143ff, 149, 152f, 160
Variablen-Bezeichner 62, 179, 196
Definitionspunkt 206
Gebiet 206

Variablen-Name **78**, 143
Variablen-Objekt 77, 144, 160
Variablen-Objekt **143**
Variablendeklaration 78
Variablendeklarationsteil 78, 190, 203
Variablenparameter 95, 98
Variablenparameter-Spezifikation 98
Variant_Error 222
Variante 51f
  aktive 149
  eingestellte 53
  Zugriff 149
Variantenselektor 51f
Variantteil 51f
verallgemeinerte Variable **77**, 98, 105, 116, 118, 143ff, 149, 152f, 160
Verbundanweisung 87, 89, **168f**, 224
Vergleich
  lexikographischer 133
  von einfachen Werten 132
  von Mengen 132
  von Zeichenketten 132
  von Zeigerwerten 132
Vergleichsoperator 115, 119, 131
verträgliche Typen 69
Verweise 63
Verweiswert 63, 359
voll qualifizierter Bezeichner 197
vordefinierte Bezeichner 378
  Definitionspunkt 208
  Gebiet 208
vordefinierte Konstanten-Bezeichner 30
vordefinierte Typ-Bezeichner 37
vordefinierte Unterprogramme 233
vorzeichenlose Konstante 115

**W**
Wertparameter 95f
Wertparameter-Spezifikation 96
WHILE-Anweisung 177
Wiederholfaktor 139
Wiederholungsanweisung 166, 174
Wirtstyp **44**, 67, 69, 116, 135
WITH-Anweisung 182
  Gültigkeitsbereich 182

WITH-Liste 196
Wortsymbole 2, 14, 381

Write 252, 340
Writeln 61
Writestring 280

**X**
Xref 306

**Z**
Zahl, reelle 38
Zeichenkette 21
  Apostrophdarstellung 21
  leere 49
  leere **21**
  Sedezimalform 22
Zeichenkettentyp **48**, 69, 71, 280
  fester Länge **48**, 245, 277
  variabler Länge **49**
Zeichenwert 40
Zeiger 63
Zeiger-Konstante 360
Zeiger-Objekt **152**
Zeiger-Variable 359f
Zeigerdereferenzierung 199
Zeigertyp **63**, 69, 72, 199
  generischer 65, 265
  privater 199
Zuweisung 158, 160
Zuweisungsverträglichkeit 71