

Latin 1 - Zeichentabelle

DEZ	HEX	ZEICHEN	DEZ	HEX	ZEICHEN	DEZ	HEX	ZEICHEN
32	20	sp	64	40	@	96	60	`
33	21	!	65	41	A	97	61	a
34	22	"	66	42	B	98	62	b
35	23	#	67	43	C	99	63	c
36	24	\$	68	44	D	100	64	d
37	25	%	69	45	E	101	65	e
38	26	&	70	46	F	102	66	f
39	27	'	71	47	G	103	67	g
40	28	(72	48	H	104	68	h
41	29)	73	49	I	105	69	i
42	2A	*	74	4A	J	106	6A	j
43	2B	+	75	4B	K	107	6B	k
44	2C	,	76	4C	L	108	6C	l
45	2D	-	77	4D	M	109	6D	m
46	2E	.	78	4E	N	110	6E	n
47	2F	/	79	4F	O	111	6F	o
48	30	0	80	50	P	112	70	p
49	31	1	81	51	Q	113	71	q
50	32	2	82	52	R	114	72	r
51	33	3	83	53	S	115	73	s
52	34	4	84	54	T	116	74	t
53	35	5	85	55	U	117	75	u
54	36	6	86	56	V	118	76	v
55	37	7	87	57	W	119	77	w
56	38	8	88	58	X	120	78	x
57	39	9	89	59	Y	121	79	y
58	3A	:	90	5A	Z	122	7A	z
59	3B	;	91	5B	[123	7B	{
60	3C	<	92	5C	\	124	7C	
61	3D	=	93	5D]	125	7D	}
62	3E	>	94	5E	^	126	7E	~
63	3F	?	95	5F	_	127	7F	

Zeichencodes 0 - 127 der Unicode - Tabelle (ASCII -Tabelle)

Bei den nicht aufgeführten Zeichen 0 - 31 sowie Zeichen 127 handelt es sich um systemabhängige Steuerzeichen. Zeichen 32 ist das Leerzeichen (space).

Latin 1 - Zeichentabelle

DEZ	HEX	ZEICHEN	DEZ	HEX	ZEICHEN	DEZ	HEX	ZEICHEN
160	A0	nbsp	192	C0	À	224	E0	à
161	A1	ı	193	C1	Á	225	E1	á
162	A2	€	194	C2	Â	226	E2	â
163	A3	£	195	C3	Ã	227	E3	ã
164	A4	¤	196	C4	Ä	228	E4	ä
165	A5	¥	197	C5	Å	229	E5	å
166	A6	¦	198	C6	Æ	230	E6	æ
167	A7	§	199	C7	Ç	231	E7	ç
168	A8	¨	200	C8	È	232	E8	è
169	A9	©	201	C9	É	233	E9	é
170	AA	ª	202	CA	Ê	234	EA	ê
171	AB	«	203	CB	Ë	235	EB	ë
172	AC	¬	204	CC	Ì	236	EC	ì
173	AD	sh	205	CD	Í	237	ED	í
174	AE	®	206	CE	Î	238	EE	î
175	AF	¯	207	CF	Ï	239	EF	ï
176	B0	°	208	D0	Ð	240	F0	ð
177	B1	±	209	D1	Ñ	241	F1	ñ
178	B2	²	210	D2	Ò	242	F2	ò
179	B3	³	211	D3	Ó	243	F3	ó
180	B4	´	212	D4	Ô	244	F4	ô
181	B5	µ	213	D5	Õ	245	F5	õ
182	B6	¶	214	D6	Ö	246	F6	ö
183	B7	·	215	D7	×	247	F7	÷
184	B8	,	216	D8	Ø	248	F8	ø
185	B9	ˆ	217	D9	Ù	249	F9	ù
186	BA	°	218	DA	Ú	250	FA	ú
187	BB	»	219	DB	Û	251	FB	û
188	BC	¼	220	DC	Ü	252	FC	ü
189	BD	½	221	DD	Ý	253	FD	ý
190	BE	¾	222	DE	Þ	254	FE	þ
191	BF	¿	223	DF	ß	255	FF	ÿ

Zeichencodes 128 - 255 der Unicode - Tabelle

Bei den nicht aufgeführten Zeichen 128 - 159 handelt es sich um systemabhängige Steuerzeichen. Zeichen 160 ist das feste Leerzeichen (non breaking space), Zeichen 173 ist das "weiche" Trennzeichen (soft hyphen, dies wird nur bei Bedarf in der Randzone eines Zeilenumbruchs angezeigt).

Component Pascal Datentypen

Übersicht

Die in Component Pascal vordefinierten Datentypen sind folgendermaßen klassifizierbar:

1. Einfache Datentypen
2. Strukturierte Datentypen
3. Zeiger- (Referenz-) Datentypen

1. Einfache Datentypen

Einfache Datentypen werden auch skalare Datentypen genannt, sie gehören zu einer der folgenden Kategorien:

- 1.1. Anordnungen
- 1.2. Fließkommazahlen
- 1.3. Wahrheitswerte

1.1. Anordnungen

Anordnungen werden oft auch ordinale Datentypen genannt. Ordinale Datentypen sind geordnete Mengen von Elementen, die einen Anfang und ein Ende haben. Jedes der Elemente (mit Ausnahme des ersten) hat einen Vorgänger, ebenso hat jedes (mit Ausnahme des letzten) einen Nachfolger.

Vordefinierte Ordinaltypen

- 1.1.1. Zeichen
- 1.1.2. Ganzzahltypen

1.1.1. Zeichen

- 1.1.1.1. Eine Variable vom Typ *SHORTCHAR* belegt ein Byte Speicherplatz und kann als Werte die 256 Elemente des Latin1 Zeichensatzes annehmen.
- 1.1.1.2. Eine Variable vom Typ *CHAR* belegt zwei Byte Speicherplatz und kann als Werte die 65536 Elemente des Unicode Zeichensatzes annehmen.

1.1.2. Ganzzahltypen

1.1.2.1. Eine Variable vom Typ *BYTE* enthält ein Byte lange ganzzahlige Werte (mit Vorzeichen).

Bereich: -2^7 bis einschließlich $2^7 - 1$

1.1.2.2. Eine Variable vom Typ *SHORTINT* enthält zwei Byte lange (ganzzahlige) Werte (mit Vorzeichen).

Bereich: -2^{15} bis einschließlich $2^{15} - 1$

1.1.2.3. Eine Variable vom Typ *INTEGER* enthält vier Byte lange ganzzahlige Werte (mit Vorzeichen).

Bereich: -2^{31} bis einschließlich $2^{31} - 1$

1.1.2.4. Eine Variable vom Typ *LONGINT* enthält acht Byte lange ganzzahlige Werte (mit Vorzeichen).

Bereich: -2^{63} bis einschließlich $2^{63} - 1$

Für Variablen der Ganzzahl-Typen sind folgende Operatoren definiert (s. a. **Relationen**, B 11):

+	Addition
-	Subtraktion
*	Multiplikation
<i>DIV</i>	Ganzzahldivision
<i>MOD</i>	Divisionsrest

1.2. Fließkommazahlen

Reelle Zahlen werden in der Informatik oft auch Fließkommazahlen (floating point numbers) genannt. Da es sich bei diesen Zahlen um endliche und unendlich-periodische oder nichtperiodische Dezimalbrüche handelt, erfordern sie rechnerintern eine spezielle Darstellung, die mit Näherungswerten, d.h. mit endlicher Genauigkeit arbeitet (die Genauigkeit kann nicht allgemein angegeben werden, da sie abhängig vom jeweiligen Prozessor ist).

1.2.1. Eine Variable vom Typ *SHORTREAL* enthält vier Byte lange Dezimalzahlen (mit Vorzeichen).

Bereich: $-3.402823 \cdot 10^{+38}$ bis $3.402823 \cdot 10^{+38}$

1.2.2. Eine Variable vom Typ *REAL* enthält acht Byte lange Dezimalzahlen (mit Vorzeichen).

Bereich: $-1.797693 \cdot 10^{+308}$ bis $1.797693 \cdot 10^{+308}$

Für Variablen der Fließkomma-Typen sind folgende Operatoren definiert (s. a. **Relationen**, B 11):

+	Addition
-	Subtraktion
*	Multiplikation
/	Division

Variablen der Zahltypen sind aufwärts zuweisungskompatibel im Sinne der Inklusionen

BYTE \subset *SHORTINT* \subset *INTEGER* \subset *LONGINT* \subset *SHORTREAL* \subset *REAL*

1.3. Wahrheitswerte

Eine Variable vom Typ *BOOLEAN* belegt ein Byte Speicherplatz und kann die Werte *FALSE* oder *TRUE* annehmen.

Für Variablen des Typs *BOOLEAN* sind folgende Operatoren definiert (s. a. **Relationen**, B 11):

&	UND-Verknüpfung zweier Wahrheitswerte
OR	ODER-Verknüpfung zweier Wahrheitswerte
~	Negation eines Wahrheitswertes

2. Strukturierte Datentypen

Strukturierte Datentypen können je nach Typ aus Entitäten gleichartiger oder auch verschiedener Datentypen zusammengesetzt sein. Die Entitäten können dabei selber einfache Datentypen, strukturierte oder Zeigertypen (s. Abschnitt 3) sein.

Vordefinierte strukturierte Datentypen

Vordefinierte strukturierte Datentypen sind Mengen, Reihungen und Verbunde:

- 2.1. *SET*
- 2.2. *ARRAY*
- 2.3. *RECORD*

2.1. SET

Eine Variable vom Typ *SET* enthält 32 Elemente vom Typ *INTEGER*. Mengen nehmen nur wenig Speicher in Anspruch, da für jedes Element einer Menge nur ein Bit benötigt wird, das anzeigt, ob ein Objekt des Basistyps in der Menge enthalten ist.

Die generelle Deklaration einer Mengenvariablen lautet:

```
VAR set: SET;
```

Mengen werden mit geschweiften Klammern notiert. Die einzelnen Elemente bzw. zusammenhängende Bereiche werden durch Kommata voneinander getrennt.

Beispiel einer Wertzuweisung:

```
set := {1, 4, 6..11};
```

Für Mengen sind folgende Operationen erklärt (s. a. **Relationen**, B 11):

- + Vereinigungsmenge
- Differenzmenge
- * Schnittmenge
- / Symmetrische Differenz

2.2. ARRAY

Der Typ **ARRAY** definiert zusammenhängende Aneinanderreihungen von Basisvariablen desselben Typs, der ein einfacher, (nicht rekursiver) strukturierter oder Zeiger-Typ (s. Abschnitt 3.) sein kann.

Die generelle Deklaration lautet:

```
TYPE
  Reihung = ARRAY <Indexbereich>{, <Indexbereich>} OF <Basistyp>;
```

Indexbereich gibt die Anzahl der **ARRAY**-Basisvariablen an. **ARRAY**-Indizes laufen stets von Null bis Indexbereich - 1. Die Erstellung n-dimensionaler Felder ist möglich, indem n Indexbereiche (Unter-**ARRAY**s), durch Kommata getrennt, angegeben werden.

Beispiele:

```
CONST
  Breite = 8; Höhe = Breite;
TYPE
  Feld = ARRAY 3 OF INTEGER;
  Schachbrett = ARRAY Breite, Höhe OF BOOLEAN;
```

Spezielle **ARRAY**-Typen sind Zeichenketten.

Mit

```
TYPE
  Zeichenkette = ARRAY 5 OF CHAR;
VAR
  String1: Zeichenkette;
  String2: ARRAY 5 OF CHAR;
  String3: ARRAY 17 OF CHAR;
```

werden drei Zeichenketten unterschiedlicher Typen deklariert. Diesen seien die folgenden Werte zugewiesen

```
String1 := "Hals"; String2 := "Tuch"; String3 := "Mörder";
```

Zeichenketten sind **NUL** terminierte Reihungen, eine Zeichenkettenvariable des Typs **ARRAY <n> OF CHAR** kann Zeichenketten der maximalen Länge n - 1 aufnehmen, das letzte, terminierende Zeichen ist 0X.

Die Zuweisung

```
String1 := String2;
```

ist wegen der Inkompatibilität der beiden Zeichenkettentypen trotz identischer Länge der Zeichenketten nicht möglich.

Dagegen ist die Zuweisung

```
String1 := String2$;
```

erlaubt, mit dem reservierten Zeichen \$ wird die in **String2** enthaltene Zeichenkette in die Variable **String1** kopiert (die aufnehmende Variable muß hinreichend groß sein).

Für Zeichenkettenvariablen ist folgender Operator definiert (s. a. **Relationen**, B 11):

- + Zeichenkettenverbindung (Concatenation)

Die Zuweisung

```
String3 := String1 + String2 + String3 + "In";
```

kopiert die Zeichenkette **HalsTuchMörderIn** in die Variable **String3**.

2.3 RECORD

Der Typ **RECORD** deklariert einen Verbund von Entitäten gleicher oder verschiedener Typen. Diese können einfache, (nicht rekursive) strukturierte oder Zeiger-Typen (s. Abschnitt 3.) sein.

Die generelle Deklaration eines Verbundes lautet:

```

TYPE
  Verbund = <Attribut> RECORD [( <Basistyp> )]
    <Feldliste>
END;

```

Die Feldliste kann Null oder mehr Felddeklarationen enthalten. Jede Felddeklaration besteht wie eine Variablendeklaration aus einem Bezeichner, einem Doppelpunkt und einem Datentyp. Die einzelnen Felder werden durch Semikola voneinander getrennt. Im Gegensatz zu **ARRAY**-Typen können **RECORD**-Typen verschiedenartige Datentypen enthalten, die (mit Ausnahme von Zeigertypen) nicht rekursiv sein dürfen. **RECORD**-Typen sind erweiterbar, bei einem erweiterten **RECORD**-Typ wird der Basistyp dem Schlüsselwort **RECORD** in runden Klammern angefügt. Damit sind alle (exportierten) Felder des Basistyps Bestandteile des Erweiterungstyps.

RECORD-Typen können die folgenden Attribute und Eigenschaften besitzen:

Attribut	erweiterbar	allozierbar	zuweisbar
– (final)	Nein	Ja	Ja
EXTENSIBLE	Ja	Ja	Nein
ABSTRACT	Ja	Nein	Nein
LIMITED	Im definierenden Modul	Im definierenden Modul	Nein

3. Referenz-Datentypen

Referenz-Datentypen werden auch Zeiger-Typen genannt. Variablen eines Referenz-Datentyps enthalten ("zeigen auf") die Anfangsadresse eines Speicherbereichs, der den Inhalt der anonymen, vom Zeiger referenzierten Variablen vom Typ **ARRAY** oder **RECORD** aufnehmen kann.

Vordefinierte Referenz-Datentypen

Vordefinierte Referenz-Datentypen sind

- 3.1. **Zeiger**
- 3.1.1. **POINTER TO RECORD**
- 3.1.2. **POINTER TO ARRAY**
- 3.2. **PROCEDURE**

3.1. Zeiger

Eine Zeiger- (**POINTER**-) Variable enthält als Wert eine Speicheradresse, d.h. sie referenziert einen mit dieser Adresse beginnenden Speicherbereich, der den eigentlichen Inhalt der Zeigerbasisvariablen aufnimmt. Die Länge des Speicherbereichs wird bestimmt durch den Zeigerbasistyp, das ist der Typ der anonymen Basisvariablen. Zeigerbasistypen können (benannte oder anonyme) **RECORD**- und **ARRAY**-Typen sein.

3.1.1. POINTER TO RECORD

Beispiel:

```

TYPE
  VerbundZeiger = POINTER TO RECORD ... END;
VAR
  ZeigerVariable1, ZeigerVariable2: VerbundZeiger;

```

Der für die Basisvariable vom Typ **RECORD** benötigte Speicherplatz muß durch die Standardprozedur **NEW** (s. Anhang C) bereitgestellt (alloziert) werden.

Beispiel:

```

NEW(ZeigerVariable1);

```

Die Basisvariable ist stets anonym und wird durch einen dem Bezeichner der Zeigervariablen nachgestellten Zeigeroperator ↑ (im Latin1-Zeichensatz symbolisiert durch ^ [Zeichen 94]) dereferenziert (Beispiel: **ZeigerVariable1**[↑]).

Ein deklarerter Zeiger hat stets einen - eventuell undefinierten - Inhalt. Um zu verhindern, daß ein unkontrollierter Zugriff auf eine nicht definierte Adresse erfolgt, weist man dem Zeiger den Wert **NIL** zu, dadurch zeigt der Zeiger auf "Nichts".

Die Zuweisung des Wertes (der Speicheradresse) von *ZeigerVariable1* an *ZeigerVariable2* geschieht folgendermaßen:

```
ZeigerVariable2 := ZeigerVariable1;
```

dadurch referenziert *ZeigerVariable2* denselben Speicherbereich wie *ZeigerVariable1*.

Dagegen wird durch die Zuweisung

```
ZeigerVariable2^ := ZeigerVariable1^;
```

der Wert der anonymen Basisvariablen *ZeigerVariable1^* in die anonyme Basisvariable *ZeigerVariable2^* kopiert, die Zeiger *ZeigerVariable1* und *ZeigerVariable2* referenzieren unterschiedliche Speicherbereiche mit dem gleichen Inhalt.

3.1.2. POINTER TO ARRAY

Ein *ARRAY*-Zeigertyp wird deklariert als

```
DynArray = POINTER TO ARRAY OF {ARRAY OF ...} <Basistyp>;
```

die Dimension des Typs (s. Abschnitt 2.2.) wird durch die Anzahl *n* der Schlüsselwörter *ARRAY OF* festgelegt. Für Variablen dieses Typs muß der benötigte Speicher mit der Standardprozedur *NEW* alloziert werden.

Die Syntax von *NEW* ist für *ARRAY*-Zeigertypen erweitert:

```
NEW(DynArrayVariable, L0, ... , Ln - 1);
```

wobei die *n* Parameter *L0*, ... , *Ln - 1* die Längen der entsprechenden Indexbereiche des *n*-dimensionalen Feldes sind.

3.2. PROCEDURE

3.2.1. Prozedurtypen

Eine Variable eines Prozedurtyps ist ein Zeiger auf eine Speicheradresse, die den eigentlichen Unterprogrammcode enthält. Sie erfordert kein *NEW*. Die Typdeklaration legt Anzahl, Typ und Reihenfolge der Parameter (sowie bei Funktionsprozeduren den Rückgabety) fest, entspricht also einem Prozedur- (bzw. Funktions-) Kopf, jedoch ohne Prozedur-Namensbezeichner (zu Prozedurparametern s. Abschnitt 3.2.3.).

Die generelle Deklaration eines [Funktions-] Prozedurtyps lautet:

```
TYPE
  ProcTyp = PROCEDURE [( <Parameterliste> )][: <Rückgabe-Typ>];
```

Eine Variable dieses Typs wird deklariert durch:

```
VAR
  ProcVar: ProcTyp;
```

Die Anweisung

```
ProcVar := Proc;
```

weist der Prozedurvariablen *ProcVar* die Prozedur mit dem Namen *Proc* zu, die Parameterliste von *Proc* muß mit *ProcTyp* kompatibel sein.

3.2.2. Typgebundene Prozeduren

Bei typgebundenen Prozeduren handelt es sich um global deklarierte Prozeduren, die an einen (ebenfalls global deklarierten) *RECORD*-Typ gebunden sind. Die Bindung erfolgt über einen Empfänger-Parameter (receiver parameter), der in runden Klammern dem Prozedurnamen vorangestellt wird.

Der Empfänger-Parameter ist dabei entweder ein VAR- oder IN-Parameter (s. Abschnitt 3.2.3.), durch den die Prozedur an einen *RECORD*-Typ, oder ein VAL-Parameter, durch den sie an einen Zeiger-Typ gebunden wird, dessen Basistyp ebenfalls ein *RECORD*-Typ sein muß.

Beispiel:

```
TYPE
  record: RECORD ... END;
PROCEDURE (VAR actor: record) do [( <Parameterliste> )]
  [: <Rückgabety>][, <Attributliste>];
```

oder

```
TYPE
  pointer: POINTER TO RECORD ... END;
PROCEDURE (actor: pointer) do [( <Parameterliste> )]
  [: <Rückgabety>][, <Attributliste>];
```

In rein objektorientierten Sprachen gibt es nur typgebundene Prozeduren, sie werden in diesen Sprachen Methoden genannt, die zugehörigen *RECORD*-Typen heißen Klassen.

Die Attributliste einer Methode kann leer sein, ein oder zwei Attribute enthalten; Methoden-Attribute sind:

NEW dies Attribut ist bei neu eingeführten Methoden verbindlich

Zusätzlich zum **NEW**-Attribut, aber auch unabhängig davon kann eine Methode eines der folgenden Attribute haben:

- Methode kann aufgerufen, aber nicht überschrieben werden (finale Methode)
EXTENSIBLE Methode kann aufgerufen und überschrieben werden
ABSTRACT Methode kann nicht aufgerufen, muß überschrieben werden
EMPTY Methode kann aufgerufen und überschrieben werden

Methoden mit dem Attribut **ABSTRACT** müssen, Methoden mit den Attributen **EXTENSIBLE** oder **EMPTY** können in Erweiterungstypen überschrieben (redefiniert) werden. Abstrakte und leere Methoden enthalten keine Deklarationen oder Anweisungen sondern bestehen nur aus einem Prozedurkopf, der Signatur. Nicht überschriebene abstrakte Methoden führen zu einem Kompilationsfehler. Der Aufruf einer leeren, nicht überschriebenen Methode hat keinen Effekt. Leere Methoden können keine OUT-Parameter enthalten und keine Funktionsprozeduren sein.

Methoden können wie andere Bezeichner exportiert werden. Mit der "*" Marke exportierte Methoden können im importierenden Modul aufgerufen und überschrieben werden. Dagegen sind Methoden, die mit der "-" Marke exportiert werden, nur innerhalb des exportierenden Moduls aufrufbar, im importierenden Modul können sie lediglich überschrieben werden (implement only export).

3.2.3. Prozedurparameter

Prozedurparameter werden auch formale Parameter genannt. Bei der Aktivierung einer Prozedur werden sie durch die aktuellen Parameter des Prozeduraufrufs ersetzt. Aktuelle und formale Parameter müssen typkompatibel sein. Formale Parameter werden unterschieden in Wert- (value-) und Referenz- oder Variablen- (variable-) Parameter. Ein Wertparameter ist eine zur Prozedur lokale Kopie des übergebenen aktuellen Parameters, dagegen ist ein Referenzparameter ein Verweis auf den aktuellen Parameter. Veränderungen der Werte von Referenzparametern sind stets in den aktuellen Parametern sichtbar.

Formale Wertparameter haben im Prozedurkopf keine Attribute, formale Referenzparameter sind an den vorangestellten Schlüsselwörtern **IN**, **OUT** oder **VAR** erkennbar. IN-Parameter können nur Parameter mit **ARRAY**- oder **RECORD**-Typen sein, ihre Werte sind innerhalb der Prozedur nicht änderbar (read only parameter). Die Werte von OUT-Parametern sind bei Eintritt in die Prozedur unbestimmt (write only parameter). VAR-Parameter können als veränderbare IN-OUT-Parameter gesehen werden, ihre Werte sind beim Prozedureintritt und Prozeduraustritt bestimmt (read-write parameter).

Relationen

Für Variablen und Konstanten der vordefinierten Datentypen sind folgende Vergleichsoperatoren definiert:

Zeichen	Bedeutung	Gültigkeit (Typ)
=	Gleich	Zahltypen, Zeichen, Zeichenketten, BOOLEAN, SET, Zeiger, Prozeduren
#	Ungleich	Zahltypen, Zeichen, Zeichenketten, BOOLEAN, SET, Zeiger, Prozeduren
<	Kleiner Als	Zahltypen, Zeichen, Zeichenketten
<=	Kleiner Als Oder Gleich	Zahltypen, Zeichen, Zeichenketten
>	Größer Als	Zahltypen, Zeichen, Zeichenketten,
>=	Größer Als Oder Gleich	Zahltypen, Zeichen, Zeichenketten,
IN	Enthalten In	SET
IS	Typtest	(POINTER TO) RECORD

Erläuterungen

x IN S bedeutet: x ist Element der Menge S

x muß eine ganze Zahl im Bereich 0 .. MAX(SET) - 1 sein, S muß den vordefinierten Typ SET haben.

v IS T bedeutet: v ist vom (dynamischen) Typ T

v muß IN- oder VAR-Parameter eines RECORD-Typs oder Zeiger auf einen RECORD sein, T muß der Typ von v oder ein Erweiterungstyp davon sein.

Schlüsselbezeichner und Standardprozeduren

Sonderzeichen und Schlüsselwörter (Reservierte Wörter)

+	:=	ABSTRACT	IF	RETURN
-	^	ARRAY	IMPORT	THEN
*	=	BEGIN	IN	TO
/	#	BY	IS	TYPE
~	<	CASE	LIMITED	UNTIL
&	>	CLOSE	LOOP	VAR
.	<=	CONST	MOD	WHILE
,	>=	DIV	MODULE	WITH
;	..	DO	NIL	
	:	ELSE	OF	
\$		ELSIF	OR	
'	"	EMPTY	OUT	
()	END	POINTER	
[]	EXIT	PROCEDURE	
{	}	EXTENSIBLE	RECORD	
(*	*)	FOR	REPEAT	

Reservierte Wörter sind Bezeichner, die nur in ihrem in der Sprachdefinition eindeutig festgelegten Sinn benutzt werden dürfen.

Standard - Bezeichner

ANYPTR	LONGINT
ANYREC	REAL
BOOLEAN	SET
BYTE	SHORTCHAR
CHAR	SHORTINT
FALSE	SHORTREAL
INF	TRUE
INTEGER	

Standard - Prozeduren

ABS	INCL
ASH	LEN
ASSERT	LONG
BITS	MAX
CAP	MIN
CHR	NEW
DEC	ODD
ENTIER	ORD
EXCL	SHORT
HALT	SIZE
INC	

Standard-Bezeichner und -Prozeduren haben eine in der Sprachdefinition vorgegebene Bedeutung. Es ist jedoch möglich, diese Bezeichner innerhalb eines Programms mit einer anderen, in diesem Programm explizit festgelegten Bedeutung zu verwenden.

Standard - Prozeduren

PROCEDURE **ABS** (Zahl: <Zahl-Typ>): <Typ wie Parameter>;

Gibt den Absolutbetrag der Zahl zurück.

PROCEDURE **ASH** (Bezeichner, n: <Ganzzahl-Typ>): LONGINT;

Multipliziert Bezeichner mit 2^n (ASH: Arithmetic Shift).

PROCEDURE **ASSERT** (Ausdruck: BOOLEAN[; Zahl: INTEGER]);

Beendet die Programmausführung, falls Ausdruck = FALSE; die optionale Zahl wird an das aufrufende System zurückgegeben.

PROCEDURE **BITS** (Zahl: INTEGER): SET;

Wandelt eine Zahl um in die Menge $\{k \mid \text{ODD}(\text{Zahl DIV } 2^k)\}$.

PROCEDURE **CAP** (Zeichen: CHAR oder SHORTCHAR): <Typ wie Parameter>;

Gibt den Kleinbuchstaben Zeichen als Großbuchstaben zurück.

PROCEDURE **CHR** (Nummer: <Ganzzahltyp>): CHAR;

Gibt das Zeichen der Position Nummer in der Unicode-Tabelle zurück.

PROCEDURE **DEC** (VAR Variable: <Ganzzahl-Typ>[; Dekrement: INTEGER]);

Erniedrigt den Wert von Variable um Dekrement Stufen; wenn der optionale Parameter Dekrement nicht angegeben wird, ist er auf Eins gesetzt.

PROCEDURE **ENTIER** (VAR Zahl: <Fließkomma-Typ>): LONGINT;

Gibt die zu Zahl nächstkleinere ganze Zahl zurück.

PROCEDURE **EXCL** (VAR Menge: SET; Zahl: <Ganzzahl-Typ>);

Entfernt aus der Menge das Element Zahl.

```
PROCEDURE HALT (Rückgabewert: INTEGER);
```

Beendet die Programmausführung und übergibt an das aufrufende System den Rückgabewert.

```
PROCEDURE INC (VAR Variable: <Ganzzahl-Typ>[; Inkrement: INTEGER]);
```

Erhöht den Wert von Variable um Inkrement Stufen; wenn der optionale Parameter Inkrement nicht angegeben wird, ist er auf Eins gesetzt.

```
PROCEDURE INCL (VAR Menge: SET; Zahl: <Ganzzahl-Typ>);
```

Fügt der Menge das Element Zahl hinzu.

```
PROCEDURE LEN (Feld[$]: ARRAY OF <beliebiger Typ>
  [; Dimension: <Ganzzahl-Typ>]): INTEGER;
```

Gibt die Länge von Feld zurück. Feld kann entweder eine Feldvariable oder ein Feldtyp sein. Der optionale Parameter Dimension gibt bei mehrdimensionalen Feldern die Dimension des gewünschten Teilfeldes an. Wird er nicht angegeben, ist er auf Null gesetzt (das erste (äußerste) Teilfeld hat stets die Dimension Null). Der optionale Parameter \$ ist nur bei Zeichenkettenvariablen zugelassen; mit ihm wird die aktuelle Länge der Zeichenkette, nicht die Länge des Zeichenkettentyps zurückgegeben.

```
PROCEDURE LONG (Bezeichner: <Zahl-Typ> bzw. <Zeichen-Typ>
  : <kompatibler Typ>);
```

Gibt den Bezeichner im Format des nächstgrösseren kompatiblen Typs zurück.

```
PROCEDURE MAX (<einfacher Typ> bzw. SET): <derselbe Typ> bzw. INTEGER;
```

Gibt das grösste Element des Typs zurück.

```
PROCEDURE MAX (a, b: <Zahltyp>): <kompatibler Typ>;
```

Gibt die grössere der beiden Zahlen zurück.

```
PROCEDURE MIN (<einfacher Typ> bzw. SET): <derselbe Typ> bzw. INTEGER;
```

Gibt das kleinste Element des Typs zurück.

```
PROCEDURE MIN (a, b: <Zahltyp>): <kompatibler Typ>;
```

Gibt die kleinere der beiden Zahlen zurück.

```
PROCEDURE NEW (VAR Zeigervariable: <Zeiger-Typ>{; Länge: INTEGER});
```

Reserviert Speicherplatz für eine Instanz des Zeigervariable zugeordneten Basistyps; als Basistypen sind nur Felder (ARRAY) oder Verbunde (RECORD) zugelassen. Die optionalen Parameter Länge sind nur für Felder zugelassen, sie setzen die Länge(n) der Teil-Felder fest (die Dimension des ersten (äußersten) Teil-Feldes ist Null).

```
PROCEDURE ODD (Zahl: <Ganzzahl-Typ>): BOOLEAN;
```

Gibt TRUE zurück bei einer ungeraden Zahl, sonst FALSE.

```
PROCEDURE ORD (Variable: CHAR oder SHORTCHAR oder SET
  : INTEGER oder SHORTINT oder INTEGER);
```

Gibt für Argumente der Typen CHAR oder SHORTCHAR den Ordinalwert der Variablen zurück, für Argumente dieser Typen ist ORD die Umkehroperation zu CHR; für Argumente des Typs SET wird eine Zahl zurückgegeben; für Argumente dieses Typs ist ORD die Umkehroperation zu BITS.


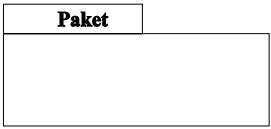
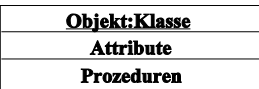
```
PROCEDURE SHORT (Bezeichner: <Zahl-Typ> bzw. <Zeichen-Typ>
  : <kompatibler Typ>);
```

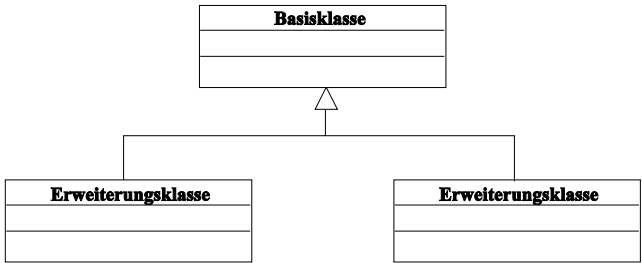
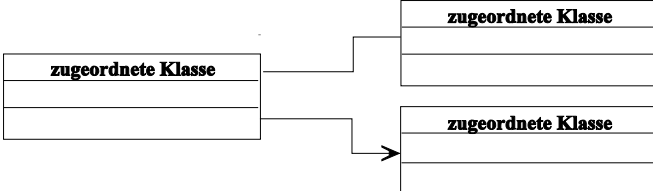
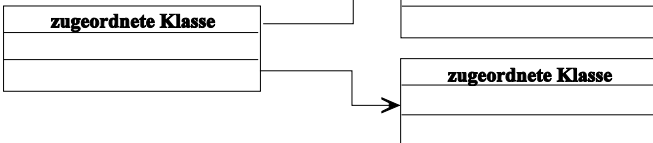
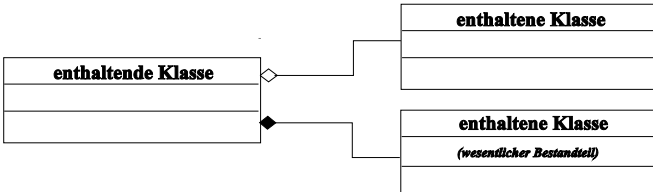
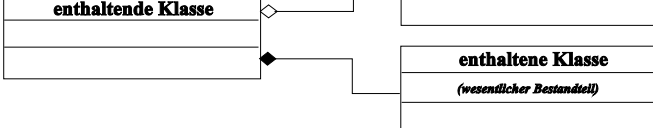
Gibt den Bezeichner im Format des nächstkleineren kompatiblen Typs zurück.

```
PROCEDURE SIZE (<beliebiger Typ>): <Ganzzahl-Typ>;
```

Gibt den Speicherbedarf eines Typs in Byte zurück.

UML - Diagramme

Komponenten, Pakete, Klassen und Objekte	
	Komponente (Subsystem) Ausführbare, abgeschlossene Programmeinheit
	Paket (Modul) Namensraum für Elemente beliebigen Typs (Klassen etc)
<div style="display: flex; justify-content: space-around;"> <div data-bbox="152 758 432 849"> <p>Paket::Klasse</p> <p>Attribute</p> <p>Methoden</p> </div> <div data-bbox="472 758 752 849"> <p><i>Paket::Klasse</i></p> <p><i>Attribute</i></p> <p><i>Methoden</i></p> </div> </div>	Klasse (Paketname fakultativ) Abstrakte Klassen und Methoden werden durch Kursivschrift gekennzeichnet
	Objekt (Instanz, Exemplar) (Klassenname fakultativ) [Name unterstrichen]

Beziehungen zwischen Klassen			
	Vererbung Jede Erweiterungsklasse (Unterklasse) besitzt alle Eigenschaften der Basisklasse (Oberklasse)		
	Assoziation Beziehung zwischen zwei Klassen ("Benutzt"-Beziehung)		
	Gerichtete Assoziation Zugriff nur in Pfeilrichtung		
	Aggregation Einbettung ("Enthaltensein"-Beziehung)		
	Komposition starke Form der Aggregation <i>(wesentlicher Bestandteil)</i>		
Sichtbarkeitskennzeichen			
+: public element	#: protected element	-: private element	~: package element
<p>Anmerkung: In Component Pascal sind die Elemente aller Klassen eines Moduls (package) im Inneren des Moduls über Klassengrenzen hinaus öffentlich (package elements, friends). Außerhalb der Modulgrenzen sind - auch für Unterklassen - ausschließlich exportierte Elemente (public elements) zugänglich.</p> <p>Bei exportierten (public) Attributen gibt es in Component Pascal neben dem gewöhnlichen (read-write) auch den schreibgeschützten (read-only) Export, der lesende Zugriffsmethoden erübrigt.</p>			