Inhalt

Anhang

Zur Verwendung als Arbeitsmaterial für die Fortbildung
sowie für Schüler im Grund- und Leistungsfach Informatik.

Die Programmiersprache Oberon[1)]

1 Einführung

Oberon ist eine universelle Programmiersprache, die aus Modula-2 hervorgegangen ist. Die wesentliche Neuerung ist das Konzept der *Typerweiterung,* das es erlaubt, neue Datentypen auf der Basis von existierenden zu definieren und Relationen zwischen beiden herzustellen.

Dieser Report ist keine Einführung in das Programmieren. Er ist bewusst knapp gehalten und soll als Referenz für Programmierer, Implementierer und Autoren von Handbüchern dienen. Was fehlt, wurde (meist) mit Absicht weggelassen, entweder weil es aus allgemeinen Regeln abgeleitet werden kann, oder weil es unklug erschien, die Spezifikation zu stark einzuschränken.

2 Syntax

Eine Sprache ist eine unendliche Menge von Sätzen, deren Aufbau bestimmten Vorschriften, der Syntax, genügt. In Oberon heißen diese Sätze Übersetzungseinheiten. Jeder Satz besteht aus einer endlichen Folge von Symbolen, die aus einem endlichen Vokabular stammen.
Das Vokabular der Sprache Oberon besteht aus Namen, Zahlen, Zeichenketten, Operatoren, Begrenzern und Kommentaren. Sie werden lexikalische Symbole genannt und bestehen ihrerseits aus Zeichenfolgen (man beachte den Unterschied zwischen Symbolen und Zeichen).
Zur Beschreibung der Syntax wird ein erweiterter Backus-Naur-Formalismus (EBNF) verwendet. Alternative Satzteile werden dabei durch einen Strich | getrennt, eckige Klammern [ ] umschließen optionale Teile und die geschwungenen Klammern { } bedeuten beliebige Wiederholung (einschließlich null mal) des geklammerten Satzteils. Syntaktische Einheiten (Nichtterminalsymbole) werden durch englische Wörter bezeichnet, welche die intuitive Bedeutung ausdrücken. Symbole des Sprachvokabulars (Terminalsymbole) werden in Anführungszeichen geschrieben oder, im Fall von sogenannten reservierten Wörtern, ausschließlich mit Großbuchstaben. Syntaxregeln (Produktionsregeln) sind durch einen senkrechten Strich am linken Rand gekennzeichnet.

3 Vokabular und Darstellung

Die Darstellung von Symbolen durch Zeichen ist mittels des ASCII-Zeichensatzes definiert. Symbole sind Namen, Zahlen, Zeichenketten, Operatoren, Begrenzer und Kommentare. Folgende lexikalische Regeln sind zu beachten: Leerzeichen und Zeilenumbrüche dürfen nicht innerhalb eines Symbols auftreten (ausgenommen innerhalb von Kommentaren und Leerzeichen in Zeichenketten). Sie werden ignoriert, es sei denn, sie sind notwendig, um zwei aufeinanderfolgende Symbole zu trennen. Groß- und Kleinbuchstaben werden unterschieden.

1. *Namen* (identifiers) sind Folgen von Buchstaben und Ziffern, wobei des erste Zeichen ein Buchstabe sein muß.

```
ident = letter {letter | digit}.
```

Beispiele:

```
x  scan  Oberon  GetSymbol  firstLetter
```

2. *Zahlen* (numbers) sind entweder ganze Zahlen (integers) oder Gleitkommazahlen (reals). Ganze Zahlen sind Folgen von Ziffern, optional gefolgt vom Buchstaben »H«. Der Typ einer Zahl ist der kleinste Typ, der die Zahl enthält. Wenn kein »H« folgt, so handelt es sich um eine Zahl in Dezimaldarstellung, folgt ein »H«, dann ist die Darstellung hexadezimal.
Eine Gleitkommazahl enthält immer einen Dezimalpunkt. Optional kann sie auch einen dezimalen Skalierungsfaktor enthalten. Der Buchstabe »E« (oder »D«) bedeutet »mal 10 hoch.« Eine Gleitkommazahl ist vom Typ REAL, es sei denn, sie enthält den Buchstaben »D«; dann ist sie vom Typ LONGREAL.

```
number = integer | real .
integer  = digit {digit} | digit {hexDigit} "H" .
real = digit {digit}  "." {digit} [ScaleFactor] .
ScaleFactor = ("E"| "D") ["+" | "-"] digit {digit} .
hexDigit =  digit |  "A" |  "B" |  "C" |  "D" |  "E" |  "F" .
digit =  "0" |  "1" |  "2" |  "3" |  "4" |  "5" |  "6" |  "7" |  "8" |  "9" .
```

1) Dies ist die Übersetzung einer überarbeiteten Version des Artikels »The programming language Oberon.« *Software - Practice and Experience,* 18, 671-90 (1988) von Niklaus Wirth.

Beispiele:

1987
100H = 256
12.3
4.567E8 = 456700000
0.57712566D-6 = 0.00000057712566

3. Zeichenkonstanten (character constants) werden entweder durch ein einzelnes Zeichen in Anführungszeichen oder durch die hexadezimale Ordnungszahl des Zeichens, gefolgt vom Buchstaben X, bezeichnet.

CharConstant = " " " character " " " | digit {hexDigit} "X".

4. Zeichenketten (strings) sind in Anführungszeichen (") eingeschlossene Zeichenfolgen. Eine Zeichenkette kann selbst keine Anführungszeichen enthalten. Die Anzahl der Zeichen einer Zeichenkette wird als ihre Länge bezeichnet. Zeichenketten können an Zeichen-Arrays zugewiesen und mit diesen verglichen werden.

string = " " " {character} " " ".

Beispiele:

"OBERON" "Don't worry!"

5. Operatoren (operators) und Begrenzer (delimiters) sind die nachfolgend aufgeführten Spezialzeichen, Zeichenpaare und reservierten Wörter. Die reservierten Wörter, auch Schlüsselwörter genannt, bestehen ausschließlich aus Großbuchstaben und können nicht als Namen benutzt werden.

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

6. Kommentare (comments) sind beliebige, in Kommentarklammern (* *) eingeschlossene Zeichenfolgen, die überall im Programm zwischen zwei Symbolen eingefügt werden dürfen. Kommentare haben keinen Einfluss auf die Bedeutung eines Programms.

4 Deklarationen und Sichtbarkeitsregeln

Jeder in einem Programm verwendete Name muss mittels einer Deklaration eingeführt werden, es sei denn, es handelt sich um einen vordeklarierten Namen. Deklarationen dienen auch dazu, bestimmte unveränderliche Eigenschaften eines Objekts festzulegen, zum Beispiel ob es sich um eine Konstante, einen Typ, eine Variable oder eine Prozedur handelt.

Nach der Deklaration wird der Name verwendet, um sich auf das damit verbundene Objekt zu beziehen. Das ist aber nur in jenen Programmteilen möglich, in denen die Deklaration sichtbar ist. Der Sichtbarkeitsbereich erstreckt sich textuell vom Ort der Deklaration bis zum Ende des Blocks (Prozedur oder Modul), zu dem die Deklaration gehört; das Objekt ist daher lokal zu diesem Block. Ein Name darf innerhalb eines Blocks nur einmal definiert werden. Folgende Zusätze gelten für diese Regel:

1) Wird ein Typ ***T*** als ***POINTER TO T1*** definiert, dann darf ***T1*** auch textuell nach ***T*** deklariert werden, aber innerhalb des gleichen Blocks.

2) Feldnamen einer Record-Deklaration sind nur innerhalb von Feldbezeichnern sichtbar.

Ein im äußersten Block deklarierter Name kann von einer Exportmarkierung (*) gefolgt werden, die anzeigt, dass der Name vom deklarierenden Modul exportiert wird. In diesem Fall kann der Name auch in anderen Modulen verwendet werden, wenn diese das deklarierende Modul importieren.

Dem Namen des importierten Objekts wird, durch einen Punkt getrennt, der Name des exportierenden Moduls als Präfix vorangestellt. Das Präfix und der Name werden zusammen als ***qualifizierter Name*** bezeichnet.

qualident = [ident “.“] ident .
identdef = ident [“*“] .

Folgende Namen sind vordeklariert:

ABS	COPY	INC	LONGREAL	REAL
ASH	DEC	INCL	MAX	SET
BOOLEAN	ENTIER	INTEGER	MIN	SHORT
CAP	EXCEL	LEN	NEW	SHORTINT
CHAR	FALSE	LONG	ODD	SIZE
CHR	HALT	LONGINT	ORD	TRUE

5 Konstantendeklarationen

Eine Konstantendeklaration bindet einen Namen an einen konstanten Wert.

ConstantDeclaration = identdef “=“ ConstExpression .
ConstExpression = expression .

Ein konstanter Ausdruck (constant expression) kann durch eine statische Analyse des Programmtexts ausgewertet werden, ohne das Programm tatsächlich auszuführen. Seine Operanden sind Konstanten.

Beispiele:
N = 100
limit = 2*N - 1
all = {0 .. WordSize-1}

6. Typdeklarationen

Eine Typdeklaration legt die Menge der Werte fest, die Variablen dieses Typs annehmen können, und definiert die darauf anwendbaren Operationen. Eine Typdeklaration bindet einen Namen an einen Typ, der entweder unstrukturiert sein kann (Grundtyp) oder strukturiert. Im letzteren Fall definiert er auch die Struktur von Variablen dieses Typs und, implizit,

die Operationen, die auf die Komponenten angewendet werden können. Es gibt zwei Arten von strukturierten Typen: den Array (Reihung) und den Record (Verbund), mit unterschiedlichem Zugriff auf die Komponenten.

TypeDeclaration = identdef "=" type .
type = qualident | ArrayType | RecordType | PointerType | ProcedureType .

Beispiel:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
Node = RECORD key: INTEGER;
  left, right: Tree
END
CenterNode = RECORD (Node)
  name: ARRAY 32 OF CHAR;
  subnode: Tree
END
Function*  =  PROCEDURE (x: INTEGER): INTEGER
```

6.1 Grundtypen

Die folgenden vordefinierten Namen bezeichnen die in Oberon verfügbaren Grundtypen.

Die Grundtypen und ihre Wertebereiche sind:

1.	BOOLEAN	die Wahrheitswerte TRUE und FALSE.
2.	CHAR	die Zeichen des erweiterten ASCII-Zeichensatzes (0X...0FFX).
3.	SHORTINT	die ganzen Zahlen zwischen MIN(SHORTINT) und MAX(SHORTINT).
4.	INTEGER	die ganzen Zahlen zwischen MIN(INTEGER) und MAX(INTEGER).
5.	LONGINT	die ganzen Zahlen zwischen MIN(LONGINT) und MAX(LONGINT).
6.	REAL	die reellen Zahlen zwischen MIN(REAL) und MAX(REAL).
7.	LONGREAL	die reellen Zahlen zwischen MIN(EONGREAL) und MAX(LONGREAL).
8.	SET	die Mengen der ganzen Zahlen zwischen 0 und MAX(SET).

Die Typen 3 bis 5 heißen ***Integer-Typen****, 6* und 7 sind ***reelle* Typen**, und zusammen heißen sie *numerische* Typen. Sie bilden eine Hierarchie; der größere Typ ***schließt die (Werte der) kleineren Typen ein:***

LONGREAL ⊇ REAL ⊇ LONGINT ⊇ INTEGER ⊇ SHORTINT

6.2 Reihung

Ein Array ist eine Struktur, die aus einer festen Anzahl von Elementen desselben Typs, ***Elementtyp*** genannt, besteht. Die Anzahl der Elemente eines Arrays heißt seine *Länge.* Die Elemente eines Arrays werden durch Indizes bezeichnet, die ganze Zahlen zwischen 0 und der Länge minus 1 sind.

ArrayType = Array length {"," length} OF type
length = ConstExpression .

Eine Deklaration der Form

ARRAY N0, N1, ... , NK OF T

wird als Abkürzung folgender Deklaration betrachtet:

```
ARRAY  N0 OF
   ARRAY N1 OF
      ...
        ARRAY NK OF T
```

Beispiele für Array-Typen sind:

```
ARRAY N OF INTEGER
ARRAY 10, 20 OF REAL
```

6.3 Verbund

Ein Verbund ist eine Struktur bestehend aus einer festen Anzahl von Komponenten mit möglicherweise verschiedenen Typen. Die Deklaration eines Record-Typs spezifiziert für jede Komponente, genannt ***Feld***, den Typ und einen Namen, der das Feld bezeichnet. Der Sichtbarkeitsbereich dieser Feldnamen ist die Record-Deklaration selbst, sie sind aber auch innerhalb von Feldbezeichnern sichtbar, die sich auf Komponenten von Record-Variablen beziehen.

```
RecordType =   RECORD ["("BaseType")"] FieldListSequence  END .
BaseType   =   qualident .
FieldListSequence = FieldList  {";"FieldList} .
FieldList = [IdentList ":" type] .
IdentList = identdef {"," identdef} .
```

Wenn ein Record-Typ exportiert wird, so müssen jene Feldnamen, die auch außerhalb des deklarierenden Moduls sichtbar sein sollen, als exportiert markiert werden. Sie heißen ***öffentliche Felder***, nicht markierte Felder heißen ***privat.***
Record-Typen sind erweiterbar, das heißt, ein Record-Typ kann als Erweiterung eines anderen Record-Typs definiert werden.

> ***Definition:*** Ein Typ ***T0*** *erweitert* einen Typ ***T***, wenn er gleich T ist oder wenn er die direkte Erweiterung einer Erweiterung von ***T*** ist. Umgekehrt ist ein Typ ***T*** ein ***Basistyp*** von ***T0***, wenn er gleich ***T0*** ist, oder wenn er der direkte Basistyp eines Basistyps von ***T0*** ist.

Beispiele für Record-Typen sind:

```
RECORD  day, month, year: INTEGER END

RECORD
     name, firstname: ARRAY  32  OF CHAR;
     age: INTEGER;
   salary: REAL
END
```

6.4 Zeigertypen

Variablen eines Zeigertyps P enthalten als Werte Zeiger (pointer) auf Variablen eines Typs T. Man sagt, der Zeigertyp P sei an T *gebunden* und T sei der *Zeigerbasistyp* von P. T muss ein Record- oder Array-Typ sein. Zeigertypen übernehmen die Erweiterungsbeziehung ihrer Basistypen.

Wenn ein Typ T0 eine Erweiterung von T ist und P0 ein Zeiger, der an T0 gebunden ist, dann wird auch PO als Erweiterung von *P* bezeichnet.

```
PointerType   =   POINTER T0 type .
```

Wenn *p* eine Variable vom Typ *P* = POINTER TO T ist, dann hat ein Aufruf der vordefinierten Prozedur NEW(p) folgenden Effekt: Eine Variable vom Typ T wird im Freispeicher angelegt, und ein Zeiger auf sie wird *p* zugewiesen. Dieser Zeiger *p* ist vom Typ P; die referenzierte Variable p↑ ist vom Typ T. Versagt die Allokation, so bekommt *p* den Wert NIL. Jeder Zeigervariablen kann der Wert NIL zugewiesen werden, der auf keine Variable zeigt.

6.5 Prozedurtypen

Variablen eines Prozedurtyps T haben eine Prozedur (oder NIL) als Wert. Damit eine Prozedur P einer Prozedurvariablen vom Typ *T* zugewiesen werden kann, müssen die (Typen der) Formalparamter von P dieselben sein, wie die Formalparameter, die für T angegeben sind.
Das gleiche gilt für den Ergebnistyp im Fall von Funktionsprozeduren. P darf weder eine vordefinierte Prozedur noch lokal zu einer anderen Prozedur deklariert sein.

ProcedureType = PROCEDURE [FormalParameters] .

7. Variablendeklarationen

Variablendeklarationen führen Variablen ein und binden einen Namen und einen Typ daran.

VariableDeclaration = IdentList ":" type .

Variablen, deren Namen in derselben Liste erscheinen, sind vom selben Typ.

Beispiele für Variablendeklarationen sind:

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF
        RECORD ch: CHAR;
        count: INTEGER
          END
t: Tree
```

Variablen eines Zeigertyps T0 und Var- Parameter eines Record-Typs *T0* können Werte annehmen, deren Typ eine Erweiterung des deklarierten Typs *T0* ist.

8. Ausdrücke

Ausdrücke (expressions) beschreiben Rechenvorschriften, nach denen Konstante und die aktuellen Werte von Variablen kombiniert werden, um durch Anwendung von Operatoren und Funktionsprozeduren neue Werte zu liefern. Ausdrücke bestehen aus Operanden und Operatoren. Runde Klammern können benutzt werden, um spezielle Assoziationen von Operatoren und Operanden auszudrücken.

8.1 Operanden

Mit Ausnahme von Mengen und Literalen, also Zahlen und Zeichenketten, werden Operanden durch Bezeichner dargestellt. Ein Bezeichner (designator) besteht aus einem Namen, der sich auf die bezeichnete Konstante, Variable oder Prozedur bezieht.
Dieser Name kann durch einen Modulnamen qualifiziert sein und von Selektoren gefolgt werden, falls das bezeichnete Objekt strukturiert ist. Wenn *A* ein Array und E einen ganzzahligen Ausdruck bezeichnen, dann bezeichnet *A[E]* jenes Element von *A,* dessen Index der aktuelle Wert von *E* ist.
Ein Bezeichner der Form $A[E_1, E_2, \ldots, E_N]$ steht für $A[E_1] [E_2] \ldots [E_N]$.

Wenn *p* einen Zeiger bezeichnet, so bezeichnet *p*↑ die durch *p* referenzierte Variable. Wenn *r* einen Record bezeichnet, dann bezeichnet *r.f* das Feld *f* von *r.*
Wenn *p* einen Zeiger bezeichnet, dann bezeichnet *p.f* das Feld *f* des Records *p*↑ (d.h. der Punkt impliziert die Dereferenzierung und *p. f* steht für *p*↑.*f*) und p[E] bezeichnet das Element von p↑ mit Index E.

Die ***Typzusicherung*** (type guard) ***v (T0)*** stellt sicher, dass die Variable ***v*** vom Typ T0 ist; falls nicht, wird die Programmausführung abgebrochen. Eine Typzusicherung ist anwendbar, falls

1) ***T0*** eine Erweiterung des deklarierten Typs T von ***v*** ist und
2) ***v*** ein formaler Var- Parameter mit Record-Typ oder ein Zeiger auf ein Record ist.

designator = qualident { "." ident | "[" ExpList "]" | "(" qualident ")" | "↑" } .
ExpList = expression { "," expression}.

Ist das bezeichnete Objekt eine Variable, so bezieht sich der Bezeichner auf den aktuellen Wert der Variablen. Ist das Objekt eine Prozedur, dann bezieht sich ein Bezeichner ohne Parameterliste auf die Prozedur selbst.
Folgt eine (möglicherweise leere) Parameterliste, so bewirkt der Bezeichner einen Aufruf der Prozedur und steht für den Wert, der aus der Prozedurausführung resultiert. Die (Typen der) Aktualparameter müssen mit den in der Prozedurdeklaration angegebenen Formalparametern übereinstimmen.

Beispiele für Bezeichner sind:

i	(INTEGER)
a[i]	(REAL)
w[3].ch	(CHAR)
t.key	(INTEGER)
t.left.right	(Tree)
t(CenterNode).subnode	(Tree)

8.2 Operatoren

Die Syntax von Ausdrücken unterscheidet zwischen vier Klassen von Operatoren mit unterschiedlicher Bindungsstärke. Der Operator ~ hat die höchste Bindungsstärke, gefolgt von Multiplikationsoperatoren, Additionsoperatoren und Relationen. Operatoren mit gleicher Bindungsstärke binden von links nach rechts (linksassoziativ). Zum Beispiel steht:
x - y - z für *(x - y) - z.*

```
expression  =  SimpleExpression  [relation SimpleExpression] .
relation =  "=" |  "#"  |  "<" | "<=" |  ">"  | ">="|  IN  |  IS .
SimpleExpression = ["+" | "-"] term {AddOperator term} .
AddOperator = "+"  |  "-"  | OR .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&" .
factor = number | CharConstant | string | NIL | set |
        designator [ActualParameters]  | "(" expression ") |
          "~" factor .
set = "{ " [element {"," element}]  "}".
element = expression [".." expression] .
ActualParameters = "(" [ExpList] ")" .
```

Die verfügbaren Operatoren sind in den Abschnitten 8.2.1- 8.2.4 aufgelistet.

Manchmal werden verschiedene Operationen durch das gleiche Symbol bezeichnet. In diesen Fällen wird die tatsächliche Operation durch den Typ der Operanden bestimmt.

Symbol	*Ergebnis*
OR	logische Disjunktion
&	logische Konjunktion
~	Negation

Logische Operatoren

Diese Operatoren sind auf Operanden vom Typ BOOLEAN anwendbar und liefern ein Ergebnis vom Typ BOOLEAN.

p OR q steht für »wenn p, dann TRUE, sonst q«
p & q steht für »wenn p, dann q, sonst FALSE«
~ p steht für »nicht p«

Symbol	*Ergebnis*
+	Summe
-	Differenz
*	Produkt
/	Quotient
DIV	ganzzahliger Quotient
MOD	ganzzahliger Rest

Arithmetische Operatoren

Die Operatoren +, -, * und / sind auf alle numerischen Typen anwendbar. Der Typ des Ergebnisses ist derjenige Operandentyp, der den Typ des anderen Operanden einschließt, außer bei der Division (/), deren Ergebnis der kleinste reelle Typ ist, der beide Operandentypen einschließt.
Als einstellige Operatoren bezeichnet Minus (-) die Vorzeichenumkehr und Plus (+) die Identität.
Die Operatoren DIV und MOD sind nur auf ganzzahlige Operanden anwendbar.
Sie sind durch folgende Beziehung zwischen beliebigem *x* und positivem *y* definiert:

x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y

Symbol	*Ergebnis*
+	Vereinigung
-	Differenz
*	Durchschnitt
/	symmetrische Differenz

Mengen- Operatoren

Mengen sind Werte vom Typ SET. Mengenoperatoren sind auf Operanden dieses Typs anwendbar.
Das einstellige Minus (-) bedeutet das Komplement von *x;* das heißt, -*x* bezeichnet die Menge der ganzen Zahlen zwischen 0 und MAX(SET), die nicht Element von *x* sind.

x - y = x * (-y)
x / y = (x-y) + (y-x)

Symbol	*Relation*
=	gleich
#	ungleich
<	kleiner
<=	kleiner oder gleich
>	größer
>=	größer oder gleich
IN	Mengenzugehörigkeit
IS	Typtest

Relationen

Relationen liefern ein Ergebnis vom Typ BOOLEAN. Die Ordnungsrelationen <, <=, > und >= sind auf numerische Typen, CHAR und Zeichen-Arrays (Zeichenketten) anwendbar.
Die Relationen = und # sind auch auf die Typen BOOLEAN und SET sowie auf Zeiger und Prozedurvariablen anwendbar, *x* IN *s* steht für „*x* ist Element von *s*“, *x* muß dabei ganzzahlig sein und s vom Typ SET. *v* IS T steht für „*v* ist vom Typ T“ und wird Typtest genannt. Er ist anwendbar, wenn

1) T eine Erweiterung des deklarierten Typs T0 von *v* ist und
2) *v* ein Var-Parameter mit Record-Typ oder ein Zeiger auf ein Record ist.

Angenommen, ***T*** erweitert ***T0*** und *v* ist eine Variable mit deklariertem Typ ***T0***, dann gibt der Typtest „*v* IS T“ an, ob der Typ des aktuellen Werts von *v* (nicht nur ***T0*** sondern) ***T*** ist.

Der Wert von NIL IS ***T*** ist undefiniert. Beispiele für Ausdrücke sind:

1987	(INTEGER)
i DIV 3	(INTEGER)
~p OR q	(BOOLEAN)
(i + j) * (i - j)	(INTEGER)
s - {8,9,13}	(SET)
i + x	(REAL)
a[i + j] *a[i - j]	(REAL)
(0<=i) & (i<100)	(BOOLEAN)
t.key = 0	(BOOLEAN)
k IN {i..j-1}	(BOOLEAN)
i IS CenterNode	(BOOLEAN)

9. Anweisungen

Anweisungen (statements) bezeichnen Aktionen. Es gibt elementare und strukturierte Anweisungen. Elementare Anweisungen enthalten keine Teile, die selbst wieder Anweisungen sind. Dazu gehören die Wertzuweisung, der Prozeduraufruf, die Return- und die Exit-Anweisung. Strukturierte Anweisungen bestehen aus Teilen, die selbst wieder Anweisungen sind. Sie werden verwendet, um Hintereinanderausführung, bedingte Ausführung, Auswahl und Wiederholung auszudrücken.

Eine Anweisung kann auch leer sein; in diesem Fall bezeichnet sie keine Aktion. Die leere Anweisung dient dazu, um die Interpunktionsregeln in Anweisungsfolgen zu vereinfachen.

statement = [assignment | ProcedureCall | IfStatement | CaseStatement | WhileStatement | RepeatStatement | LoopStatement | WithStatement | EXIT | RETURN [expression]] .

9.1 Wertzuweisungen

Wertzuweisungen (assignments) ersetzen den aktuellen Wert einer Variablen durch einen neuen Wert, der als Ausdruck spezifiziert ist. Der Zuweisungsoperator »:=« wird ausgesprochen als ***wird zu***.

assignment = designator ":=" expression .

Der Typ des Ausdrucks muss im Typ der Variablen eingeschlossen sein oder den Typ der Variablen erweitern. Dabei gelten folgende Ausnahmen:

1) Die Konstante NIL kann jeder Zeiger- oder Prozedurvariablen zugewiesen werden.
2) Zeichenketten können jeder Variablen vom Typ ARRAY *N* OF CHAR zugewiesen werden, vorausgesetzt die Länge der Zeichenkette ist kleiner als *N.*

Wird eine Zeichenkette s der Länge *n* einem Array *a* zugewiesen, dann ist das Ergebnis *a[i]* = s_i für *i* = 0, ..., *n*-1, und a[*n*]= 0X.

Beispiele für Wertzuweisungen sind:

```
i :=0
p:=i = j
x := i + 1
k := log2(i+j)
F := log2
s := {2, 3, 5, 7, 11,13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].ch := "A"
```

9.2 Prozeduraufrufe

Ein Prozeduraufruf (procedure call) aktiviert eine Prozedur. Der Aufruf kann eine Liste von Aktualparametern enthalten, die für die korrespondierenden, in der Prozedurdeklaration angegebenen Formalparameter eingesetzt werden. Die Korrespondenz ergibt sich aus der Position der Parameter in der Aktual- und Formalparameterliste. Zwei Arten von Parametern werden unterschieden: *Variablen-* und *Wertparameter.*
Im Fall eines Variablenparameters (auch Var-Parameter genannt) muss der korrespondierende Aktualparameter eine Variable bezeichnen. Bezeichnet er eine Komponente einer strukturierten Variablen, so wird der Selektor bei der Parameterübergabe ausgewertet, also schon vor der Ausführung der Prozedur. Ist der Parameter ein Wertparameter, muß der korrespondierende Aktualparameter ein Ausdruck sein. Dieser Ausdruck wird vor dem Prozeduraufruf ausgewertet und das Ergebnis dem Formalparameter zugewiesen, der innerhalb der gerufenen Prozedur eine lokale Variable darstellt.

ProcedureCall = designator [ActualParameters] .

Beispiele für Prozeduraufrufe sind:

```
ReadInt(i)
WriteInt(j*2+1,6)
INC(w[k].count)
```

9.3 Anweisungsfolgen

Eine Anweisungsfolge (statement sequence) bedeutet die Hintereinanderausführung der enthaltenen Anweisungen, die durch Strichpunkte voneinander getrennt sind.

```
StatementSequence  =  statement  {";" statement} .
```

9.4 If-Anweisungen

```
IfStatement = IF expression THEN StatementSequence
{ELSIF expression THEN StatementSequence}
[ELSE StatementSequence]
END .
```

If-Anweisungen spezifizieren die bedingte Ausführung von Anweisungen. Der boolesche Ausdruck vor einer bedingten Anweisung wird *Wache* (guard) genannt. Die Wachen werden in der Reihenfolge ihres Auftretens ausgewertet, bis eine davon den Wert TRUE ergibt; die damit verbundene Anweisungsfolge wird ausgeführt. Ist keine der Wachen erfüllt, so wird die Anweisungsfolge hinter dem Symbol ELSE ausgeführt, sofern diese vorhanden ist.

Ein Beispiel ist:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifer
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF ch = 22X THEN ReadString
END
```

9.5 Case-Anweisungen

Case-Anweisungen spezifizieren die Auswahl und Ausführung einer Anweisungsfolge gemäß dem Wert eines Ausdrucks. Zuerst wird der Ausdruck berechnet, dann wird jene Anweisungsfolge ausgeführt, die mit dem berechneten Wert markiert ist. Der Ausdruck und alle Markierungen müssen vom gleichen Typ sein, der entweder ein ganzzahliger Typ oder CHAR sein muß. Die Markierungen (labels) sind Konstanten, und kein Wert darf mehr als einmal vorkommen. Wenn der berechnete Wert keiner Markierung entspricht, so wird die Anweisungsfolge nach dem Symbol ELSE ausgeführt, sofern dieses vorhanden ist, andernfalls liegt ein Fehler vor.

```
CaseStatement =CASE expression OF case { "|" case}
                [ELSE StatementSequence] END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels} .
CaseLabel =  ConstExpression [".." ConstExpression] .
```

Ein Beispiel ist:

```
CASE ch OF
     "A" . . "Z" : ReadIdentifier
  |  "0" . . "9" : ReadNumber
  |  22X : ReadString
ELSE SpecialCharacter
END
```

9.6 While-Anweisungen

While-Anweisungen spezifizieren die Wiederholung einer Anweisungsfolge. Wenn ein boolescher Ausdruck (guard) TRUE ergibt, wird die Anweisungsfolge ausgeführt. Das Auswerten des Ausdrucks und das Ausführen der Anweisungsfolge werden so lange wiederholt, wie der Ausdruck TRUE ergibt.

WhileStatement = WHILE expression DO StatementSequence END .

Beispiele sind:

```
WHILE j > 0 DO
      j := j DIV 2; i:= i+1
END

WHILE (t# NIL) & (t.key # i) DO
      t:= t.left
END
```

9.7 Repeat-Anweisungen

Repeat-Anweisungen spezifizieren die wiederholte Ausführung einer Anweisungsfolge bis eine Bedingung erfüllt ist. Die Anweisungsfolge wird mindestens einmal ausgeführt.

RepeatStatement = REPEAT StatementSequence UNTIL expression .

9.8 Loop-Anweisungen

Eine Loop-Anweisung spezifiziert die wiederholte Ausführung einer Anweisungsfolge. Sie wird durch Ausführung einer Exit-Anweisung innerhalb der Anweisungsfolge beendet.

LoopStatement = LOOP StatementSequence END .

Ein Beispiel ist:

```
LOOP
      IF t1  = NIL THEN  EXIT  END;
      IF  k < t1.key THEN t2 := t1.left; p := TRUE
      ELSIF k > t1.key THEN t2 := t1.right; p  :=  FALSE
      ELSE  EXIT
      END;
      t1   := t2
END
```

Obwohl While- und Repeat-Anweisungen durch Loop-Anweisungen mit einem einzelnen EXIT ausgedrückt werden können, wird die Verwendung von WHILE und REPEAT für die sehr häufigen Situationen empfohlen, in denen die Terminierung von einer einzigen Bedingung am Anfang oder Ende der wiederholten Anweisungsfolge abhängt. Die Loop-Anweisung ist dann hilfreich, wenn mehrere Abbruchbedingungen und -punkte vorliegen.

9.9 Return- und Exit-Anweisungen

Eine Return- Anweisung besteht aus dem Symbol RETURN, möglicherweise gefolgt von einem Ausdruck. Sie zeigt die Beendigung einer Prozedur an und gibt, im Fall von Funktionsprozeduren, den Wert des Ausdrucks als Ergebnis des Funktionsaufrufs zurück. Der Typ des Ausdrucks muß mit dem im Prozedurkopf angegebenen Ergebnistyp kompatibel sein. Funktionsprozeduren verlangen die Anwesenheit einer Return-Anweisung, die den Ergebniswert angibt.
Mehrere Return-Anweisungen in einer Prozedur sind erlaubt, aber nur eine wird ausgeführt. Bei gewöhnlichen Prozeduren impliziert das Prozedurende eine Return- Anweisung. Ein explizites RETURN stellt daher einen zusätzlichen (meist in Ausnahmesituationen verwendeten) Abschlusspunkt dar.
Eine Exit-Anweisung besteht aus dem Symbol EXIT. Sie spezifiziert die Beendigung der umschließenden Loop-Anweisung und die Fortsetzung der Programmausführung mit der auf die Loop-Anweisung folgenden Anweisung.

Exit-Anweisungen sind also immer an eine umschließende Loop-Anweisung gebunden, auch wenn das nicht durch die Syntax ausgedrückt ist.

9.10 With-Anweisungen

Eine Zeigervariable oder ein Var-Parameter mit Record-Struktur vom Typ ***T0*** darf im Kopf einer With-Anweisung zusammen mit einem Typ ***T*** angegeben werden, der eine Erweiterung von ***T0*** ist. Die Variable wird innerhalb der With-Anweisung so behandelt, als wäre sie vom Typ T deklariert. Die With-Anweisung spielt eine ähnliche Rolle wie die Typzusicherung, nur daß die Zusicherung über eine ganze Anweisungsfolge gilt. Sie kann daher als ***regionale Typzusicherung*** aufgefaßt werden.

```
WithStatement = WITH qualident  ":"  qualident DO
                    StatementSequence END.
```

Ein Beispiel ist:

```
WITH t: CenterNode DO name := t.name; L := t.subnode END
```

10. Prozedurdeklarationen

Prozedurdeklarationen bestehen aus einem ***Prozedurkopf*** (procedure heading) und einem ***Prozedurrumpf*** (procedure body). Der Kopf gibt den Prozedurnamen, die *Formalparameter* und den Ergebnistyp (falls vorhanden) an. Der Rumpf enthält Deklarationen und Anweisungen. Am Ende der Prozedurdeklaration wird der Name der Prozedur wiederholt.
Es gibt zwei Arten von Prozeduren, nämlich *gewöhnliche Prozeduren* (proper procedures) und *Funktionsprozeduren* (function procedures). Die letzteren werden durch einen Funktionsbezeichner als Bestandteil eines Ausdrucks aktiviert und liefern ein Ergebnis, das ein Operand des Ausdrucks ist. Gewöhnliche Prozeduren werden durch einen Prozeduraufruf aktiviert. Funktionsprozeduren enthalten im Prozedurkopf hinter den Formalparametern zusätzlich den Ergebnistyp. Ihr Rumpf muß eine Return- Anweisung enthalten, die das Ergebnis des Funktionsaufrufs bestimmt.
Alle innerhalb eines Prozedurrumpfs deklarierten Konstanten, Variablen, Typen und Prozeduren sind *lokal* zu dieser Prozedur. Die Werte der lokalen Variablen sind zum Zeitpunkt des Prozeduraufrufs undefiniert. Da auch Prozeduren als lokale Objekte deklariert werden dürfen, können Prozeduren geschachtelt sein.

Zusätzlich zu den Formalparametern und lokal deklarierten Objekten sind auch die Objekte aus der Umgebung der Prozedur innerhalb der Prozedur sichtbar (außer jenen Objekten, die durch lokal deklarierte Objekte mit gleichem Namen verdeckt werden). Das Aufrufen einer Prozedur innerhalb ihrer eigenen Deklaration bedeutet einen rekursiven Prozeduraufruf.

```
ProcedureDeclaration    =
        ProcedureHeading  ";"  ProcedureBody ident.
ProcedureHeading    =
        PROCEDURE ["*"] identdef [FormalParameters].
ProcedureBody   =
                DeclarationSequence [BEGIN StatementSequence] END.
ForwardDeclaration    =
        PROCEDURE " ↑ identdef [FormalParameters].  OK
DeclarationSequence    =
        {CONST  {ConstantDeclaration  ";"} |
        TYPE  {TypeDeclaration  ";"} |
VAR {VarilableDeclaration  ";"} }
      {ProcedureDeclaration  ";"  |  ForwardDeclaration  ";"}.
```

Eine *Vorwärtsdeklaration* (forward declaration) erlaubt Referenzen auf Prozeduren, die erst später im Text deklariert werden. Die tatsächliche Deklaration - die auch den Rumpf enthält - muss dieselben Parameter und denselben Ergebnistyp (falls vorhanden) aufweisen wie die Vorwärtsdeklaration und muss im selben Block liegen.
Ein Stern hinter dem Symbol PROCEDURE weist den Compiler darauf hin, daß diese Prozedur einer Prozedurvariablen zuweisbar sein soll (je nach Implementierung ist der Hinweis optional oder erforderlich).

10.1 Formalparameter

Formalparameter sind Namen, welche die aktuellen Parameter eines Prozeduraufrufs repräsentieren. Die Korrespondenz zwischen aktuellen und formalen Parametern wird beim Prozeduraufruf hergestellt. Es gibt zwei Arten von Parametern: *Wert-* und *Variablenparameter.* Die Art wird in der Formalparameterliste festgelegt. Wertparameter stehen für lokale Variablen, die mit dem Ergebnis der Auswertung des korrespondierenden Aktualparameters initialisiert werden. Variablenparameter korrespondieren mit aktuellen Parametern, die Variablen sind, und stehen für diese Variablen. Variablenparameter werden durch das Symbol VAR gekennzeichnet, Wertparameter durch dessen Abwesenheit.
Eine Funktionsprozedur ohne Parameter muss im Prozedurkopf eine leere Parameterliste aufweisen und wird durch einen Funktionsbezeichner mit leerer Aktualparameterliste aufgerufen. Formalparameter sind lokal zur Prozedur; das heißt ihr Sichtbarkeitsbereich ist der Programmtext bis zum Ende der Prozedurdeklaration.

```
FormalParameters = "("[FPSection {";" FPSection}] ")"
                        [":" qualident].
FPSection   =  [VAR] ident  { "," ident}  ":"  FormalType.
FormalType   =  {ARRAY OF}  qualident  |  ProcedureType.
```

Die Parameterliste spezifiziert den Typ jedes Formalparameters. Für Variablenparameter muss dieser Typ identisch sein zum Typ des korre-spondierenden Aktualparameters, außer im Fall von Records, bei dem er ein Basistyp des Aktualparametertyps sein muß. Für Wertparameter gelten die Regeln der Wertzuweisung. Wird der Typ eines Formalparameters als

```
ARRAY OF T
```

angegeben, so nennt man den Parameter einen *offenen Array-Parameter* und der korrespondierende aktuelle Parameter kann ein beliebiges Array mit Elementtyp *T* sein.
Spezifiziert ein Formalparameter einen Prozedurtyp, dann muss der korrespondierende Aktualparameter entweder eine global deklarierte Prozedur sein oder eine Variable (oder Parameter) desselben Prozedurtyps. Eine vordeklarierte Prozedur kann nicht zugewiesen werden. Der Ergebnistyp einer Prozedur darf weder ein Record noch ein Array sein.

Beispiele für Prozedurdeklarationen sind:

```
PROCEDURE ReadInt(VAR x: INTEGER);
    VAR i: INTEGER; ch: CHAR;
BEGIN  i := 0; Read(ch);
    WHILE ("0" <= ch) & (ch <= „9") DO
        i := 10*i + (ORD(ch)-ORD("O")); Read(ch)
END ;
x := i
END ReadInt

PROCEDURE WriteInt(x: INTEGER); (* 0 <= x < 1.0E5 *)
    VAR i: INTEGER;
        buf: ARRAY 5 OF INTEGER;
BEGIN  i := 0;
    INC(i);
    REPEAT buf[i] := x MOD 10;  x := x DIV 10; UNTIL x = 0;
    REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt

PROCEDURE log2(x: INTEGER): INTEGER;
    VAR y: INTEGER;  (*x>0*)
BEGIN  y := 0;
    WHILE x > 1 DO x := x DIV 2; INC(y) END ;
    RETURN y
END log2
```

10.2 Vordeklarierte Prozeduren

Die folgenden drei Tabellen listen alle vordeklarierten Prozeduren auf. Einige sind *generische Prozeduren;* das heißt, sie sind auf mehrere Operandentypen anwendbar, **υ** steht für eine Variable, *x* und *n* für Ausdrücke und T für einen Typ. *Integer* steht für einen der ganzzahligen und *Real* für einen der reellen Typen.

Funktionsprozeduren

Name	*Argumenttyp*	*Ergebnistyp*	*Bedeutung*
ABS(x)	numerisch	*Typ* von *x*	Absolutwert
ODD(x)	integer	BOOLEAN	*x* MOD 2 = 1
CAP(x)	CHAR	CHAR	entsprechender Großbuchstabe
ASH(x, n)	*x, n:* Integer	LONGINT	$x * 2^n$, arithmetischer Shift
LEN(**υ**, *n]*	**υ**: Array *n:* Integer	LONGINT	Länge von **υ** in Dimension *n*
*LEN(***υ***)*	Array	LONGINT	LEN(**υ**, 0)
MAX(T)	*T* = Grundtyp T = SET	*T* INTEGER	größter Wert von *T* größtes Element einer Menge
MIN(T)	*T* = Grundtyp *T* = SET	*T* INTEGER	kleinster Wert von *T* 0
SIZE(T)	*T* = beliebig	Integer	Anzahl Bytes erforderlich für *T*

Typkonversionsprozeduren

Name	*Argumenttyp*	*Ergebnistyp*	*Bedeutung*
ORD(x)	CHAR	INTEGER	Ordnungszahl von *x*
CHR(x)	Integer	CHAR	Zeichen mit Ordnungszahl *x*
SHORT(x)	LONGINT INTEGER LONGREAL	INTEGER SHORTINT REAL	Identität (Überlauf möglich)
LONG(x)	SHORTINT [NTEGER REAL	INTEGER LONGINT LONGREAL	Identität
ENTIER(x)	Real	LONGINT	größte ganze Zahl <= *x*

Beachten Sie: ENTIER(i/j) = i DIV j

Name	*Argumenttypen*	*Bedeutung*
INC(*v*)	**Integer**	***v* = *v* + 1**
INC(*v, x*)	**Integer**	***v* = *v* + *x***
DEC(*v*)	**Integer**	***v* = *v* - 1**
DEC(v, x)	**Integer**	***v* = *v* - *x***
INCL(*v*, x)	***v*: SET; *x*: Integer**	***v* = *v* + {*x*}**
EXCL(*v*, x)	***v*: SET; *x*: Integer type**	***v* = *v* - {*x*}**
COPY(x, *v*)	***x*: Zeichen-Array, String** ***v*: Zeichen-Array**	***v* := *x***
NEW(*v*)	**Zeigertyp**	**erzeuge *v*↑**
HALT(x)	**Integerkonstante**	**beende Programmlauf**

In HALT(x) bleibt die Interpretation des Parameters x der Implementierung überlassen.

11. Module

Ein Modul ist eine Sammlung von Konstanten-, Typen-, Variablen-und Prozedurdeklarationen und eine Anweisungsfolge zum Zweck der Initialisierung der globalen Variablen. Ein Modul stellt typischerweise einen Text dar, der als Einheit übersetzt werden kann.

```
module   = MODULE ident ";"  [ImportList] DeclarationSequence
           [BEGIN StatementSequence] END ident "." .
ImportList  =  IMPORT import {"," import} ";" .
import  =  ident [":=" ident].
```

Die Importliste gibt alle Module an, von denen das importierende Modul Klient ist. Wird ein Name *x* von einem anderen Modul M exportiert und wird M in der Importliste aufgeführt, dann wird auf *x* in der Form *M.x* zugegriffen. Wenn in der Importliste die Form »M := M1« benutzt wird, so bezieht sich *M.x* auf das Objekt *x* aus dem Modul M1. Namen, die in Klientenmodulen sichtbar sein sollen, also außerhalb des deklarierenden Moduls, müssen in ihrer Deklaration mit einer Exportmarkierung versehen werden. Die Anweisungsfolge nach dem Symbol BEGIN wird ausgeführt, wenn das Modul zum System hinzugefügt (geladen) wird. Individuelle (parameterlose) Prozeduren können danach vom System aus aktiviert werden und dienen als Kommandos.

Beispiel:

```
MODULE Out;
(*exportierte Prozeduren: Write, WriteInt, WriteLn*)
IMPORT Texts, Oberon;
VAR W: Texts.Writer;

PROCEDURE Write*(ch: CHAR);
BEGIN Texts.Write(W, ch)
END Write;
```

```
PROCEDURE WriteInt*(x, n: LONGINT);
        VAR i: INTEGER; a: ARRAY 16 OF CHAR;
BEGIN i := 0;
        IF x < 0 THEN Texts.Write(W, "-"); x := -x  END ;
        REPEAT
        a[i] := CHR(x MOD 10 + ORD("O"));  x := x DIV 10; INC(i)
        UNTIL x = 0;
        REPEAT Texts.Write(W, "  " ); DEC(n) UNTIL n <= i;
        REPEAT DEC(i); Texts.Write(W. a[i]) UNTIL i = 0
END WriteInt;
PROCEDURE WriteLn*;
BEGIN Texts.Writeln(W); Texts.Append(Oberon.Log, W.buf)
END WriteLn;

BEGIN Texts.OpenWriter(W)
END Out.
```

Anhang

Der ASCII-Zeichensatz und typische Extremwerte

Dez	Hex	Zeichen	Dez	Hex	Zeichen	Dez	Hex	Zeichen	Dez	Hex	Zeichen
0	OX	NUL	32	20X	SP	64	40X	@	96	60X	`
1	IX	SOH	33	21X	!	65	41X	A	97	61X	a
2	2X	STX	34	22X	„	66	42X	B	98	62X	b
3	3X	ETX	35	23X	#	67	43X	C	99	63X	c
4	4X	EOT	36	24X	$	68	44X	D	100	64X	d
5	5X	ENQ	37	25X	%	69	45X	E	101	65X	e
6	6X	ACK	38	26X	&	70	46X	F	102	66X	f
7	7X	BEL	39	27X	,	71	47X	G	103	67X	g
8	8X	BS	40	28X	(	72	48X	H	104	68X	h
9	9X	HT	41	29X	)	73	49X	I	105	69X	i
10	0AX	LF	42	2AX	*	74	4AX	J	106	6AX	j
11	0BX	VT	43	2BX	+	75	4BX	K	107	6BX	k
12	0CX	FF	44	2CX	,	76	4CX	L	108	6CX	l
13	0DX	CR	45	2DX	-	77	4DX	M	109	6DX	m
14	0EX	SO	46	2EX	.	78	4EX	N	110	6EX	n
15	0FX	SI	47	2FX	/	79	4FX	O	111	6FX	o
16	10X	DLE	48	30X	0	80	50X	P	112	70X	p
17	11X	DC1	49	31X	1	81	51X	Q	113	71X	q
18	12X	DC2	50	32X	2	82	52X	R	114	72X	r
19	13X	DC3	51	33X	3	83	53X	S	115	73X	s
20	14X	DC4	52	34X	4	84	54X	T	116	74X	t
21	15X	NAK	53	35X	5	85	55X	U	117	75X	u
22	16X	SYN	54	36X	6	86	56X	V	118	76X	v
23	17X	ETB	55	37X	7	87	57X	W	119	77X	w
24	18X	CAN	56	38X	8	88	58X	X	120	78X	x
25	19X	EM	57	39X	9	89	59X	Y	121	79X	y
26	1AX	SUB	58	3AX	:	90	5AX	Z	122	7AX	z
27	1BX	ESC	59	3BX	;	91	5BX	[	123	7BX	{
28	1CX	FS	60	3CX	<	92	5CX	\	124	7CX	|
29	1DX	GS	61	3DX	=	93	5DX	]	125	7DX	}
30	1EX	RS	62	3EX	>	94	5EX	^ 1)	126	7EX	~
31	1FX	US	63	3FX	?	95	5FX	_	127	7FX	DEL

Oberon-Fonts drucken einen Pfeil nach oben » ↑ « anstelle des Dachs »^«.

Typ T	SIZE(*T*)	MIN(*T*)	MAX(*T*)
SHORTINT	1	-128	127
INTEGER	2	-32768	32767
LONGINT	4	-2147483648	2147483647
REAL[1)]	4	-3.40282346E38	3.40282346E38
LONGREAL	8	-a[2)]	a
CHAR	1	0	255
SET	4	0	31

1) REAL- und LONGREAL-Werte beziehen sich auf den ANSI/IEEE 754-1985 Standard.
2) a = 1.7976931348623157D308

Grammatik von Oberon

```
module = MODULE ident ";"   [ImportList] DeclarationSequence
        [BEGIN StatementSequence] END ident "." .
ImportList = IMPORT import {"," import} ";" .
import = ident [":=" ident].

DeclarationSequence = {CONST {ConstantDeclaration ";"} |
        TYPE {TypeDeclaration ";"}  VAR {VariableDeclaration ";"}}
        {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
ConstantDeclaration = identdef "=" ConstExpression.
identdef = ident ["*"].
ConstExpression = expression.
TypeDeclaration = identdef  "=" type.
type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
qualident =  [ident "."] ident.
ArrayType = ARRAY length {"," length} OF type.
length = ConstExpression.
RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList =  [IdentList ":" type].
IdentList = identdef {"," identdef}.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].
VariableDeclaration = IdentList ":" type.
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE ["*"] identdef [FormalParameters].
ProcedureBody = DeclarationSequence [BEGIN
        Statement Sequence] END.
FormalParameters =  "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection =  [VAR] ident {"," ident} ":" FormalType.
FormalType = {ARRAY OF} qualident.↑
ForwardDeclaration = PROCEDURE "  " identdef [Formalparameters].

statementSequence = statement {";" statement}.
statement = [assignment | ProcedureCall |
                IfStatement | CaseStatement | WhileStatement | RepeatStatement
                LoopStatement | WithStatement | EXIT | RETURN [expression] ].
assignment = designator ":=" expression.
designator = qualident {"." ident | "[" ExpList "]" |
        "(" qualident ")" | "↑" }.
ExpList = expression {"," expression}.
expression = SimpleExpression [relation SimpleExpression].
relation =  "="  | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression =  ["+" |"-"] term {AddOperator term}.
 AddOperator =  " + "  | "-" | OR .
term = factor (MulOperator factor}.
MulOperator =  "*" | "/" | DIV | MOD | "&" .
factor = number | CharConstant | string | NIL | set |
        designator [ActualParameters] | "(" expression ")" | "~" factor.
set =   "{" [element {"," element}] "}".
element = expression [".." expression].
ProcedureCall = designator [ActualParameters].
ActualParameters =  "(" [ExpList] ")" .
IfStatement = IF expression THEN StatementSequence
        {ELSIF expression THEN StatementSequence}
        [ELSE StatementSequence]
        END.
CaseStatement = CASE expression OF case {"|" case}
        [ELSE StatementSequence] END.
case =  [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
```

```
WhileStatement = WHILE expression DO Statement Sequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
LoopStatement = LOOP StatementSequence END.
WithStatement = WITH qualident ":" qualident DO
        StatementSequence END.

ident = letter {letter | digit}.
number = integer | real.

integer = digit {digit} | digit {hexDigit} "H" .
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | "D") [" + " | "-"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit =  "0" | "I" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant =  " " " character " " " | digit {hexDigit} "X".
string =   " " " {character} " " ".
```

Erweiterungen in Oberon-2

```
identdef = ident ["*" | "-"].
ArrayType = ARRAY [length {"," length}] OF type.
ProcedureHeading = PROCEDURE [Receiver] ["*"] identdef
        [FormalParameters].
Receiver =   "(" [VAR] ident ":" ident ")".
FormalType = type.                                    ↑
ForwardDeclaration = PROCEDURE [Receiver] "  " identdef
        [FormalParameters].
statement = ForStatement.
ForStatement = FOR ident ":=" expression TO expression
        [BY ConstExpression] DO StatementSequence END.
WithStatement = WITH guard DO StatementSequence
        {"I" guard DO StatementSequence} [ELSE StatementSequence] END.
guard = qualident ":" qualident.
CharConstant =   " " " character " " " | " ' " character " ' " |
        digit {hexDigit} „X".
string =  " " " {character} " " " | " ' " {character} " ' ".
```