Die Programmiersprachen Oberon und Oberon-07

S. Mertens, Einbeck, 1. November 2025

Einleitung

Oberon ist ein von Niklaus Wirth und Jürg Gutknecht ursprünglich für die Ceres-1 Workstation Computer der ETH Zürich entwickeltes Betriebssystem. Es wurde inklusive der gleichnamigen Programmiersprache 1988 zum ersten Mal veröffentlicht und in den folgenden Jahren auf handelsübliche Computersysteme portiert. Oberon folgt dem Prinzip der Reduktion auf das Wesentliche. Die Syntax der Sprache, die die Nachfolge von Modula-2 und Pascal antritt, ist formal definiert, bewusst einfach gehalten und wird auch in zahlreichen Varianten auf anderen Betriebsystemen genutzt. Strikte Typisierung und Strukturierung sind kennzeichnend für Oberon. Durch die Möglichkeit zur Erweiterung von bestehenden Datentypen, unterstützt die Sprache objektorientierte Programmierung in ihren Grundzügen. Sie realisiert Kapselung und Abstraktion durch Module. Oberon Programme können durch die klare Notation in ihrer Gänze verstanden und erklärt werden. Die formale Beschreibung der Sprache erleichtert die Implementierung von Compilern.

Dieser Text soll als Referenz und Überblick über Oberon dienen und eignet sich nicht als Lerntext für Programmieranfänger. Der Schwerpunkt liegt auf Revised Oberon [With 2016] mit ein, das auch als Oberon-07 bezeichnet wird. Wo Unterschiede zum klassischen Oberon [Wirth 1990] bestehen, wird explizit darauf hingewiesen. Oberon-2[1] ist nicht Gegenstand dieser Betrachtung.

[1] Oberon-2 ergänzt die Definition von klassischem Oberon mit typgebundene Prozeduren, die eine Entsprechung zu *Methoden* in der objektorientierten Programmiersprache Smalltalk sind. (In C++ werden Methoden als *virtual member functions* bezeichnet.)

Inhaltsverzeichnis

1 Lexikalische Eigenschaften

Der Quelltext jedes Programmmoduls ist eine Sequenz aus Kommentaren, Bezeichnern, Schlüsselwörtern, Operatoren und Begrenzern. Diese Symbole setzen sich ihrerseits aus einer Folge von Zeichen zusammen. Kommentare und Begrenzer mit dem Charakter eines Leerzeichens (z. B. Tabulatoren und Umbrüche) nehmen dabei eine Sonderrolle ein, weil sie in den formalen Grammatikregeln selbst nicht vorkommen.

1.1 Zeichen und Bezeichner

Bezeichner in Oberon werden nicht nach ihrem Verwendungszweck unterschieden. Ein Bezeichner, der für eine Konstante verwendet wird, unterscheidet sich auch lexikalisch nicht von dem Bezeichner eines Moduls, einer Prozedur, eines Typs oder einer Variablen[1].

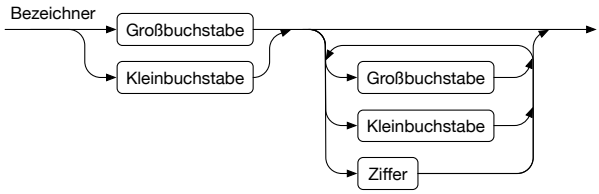

Abbildung 1: Syntaxdiagramm für Bezeichner

Das oben abgebildete Syntaxdiagramm[2] zeigt, wie in Oberon Bezeichner gebildet werden. Im weiteren Verlauf werden die wichtigsten Syntaxgruppen als solche Diagramme dargestellt. Dabei wird den formaleren Syntaxbeschreibungen, bei denen (aufgrund der stärker strukturierten Gliederung) Zusammenhänge gelegentlich nicht so schnell erstichtlich sind, teilweise deutlich vorgegriffen. Die Diagramme bilden weiterhin nur Oberon-07 ab. Auf Diagramme für klassisches Oberon wurde verzichtet. Grundlegende Syntaxelemente wie ganze Zahlen haben ebenfalls keine Diagrammform. Es gibt in der Diagrammdarstellung auch kleinere Abweichungen zur unten folgenden textuellen Darstellung, die rein formaler Natur sind[3].

Bezeichner, die sich in Groß-/Kleinschreibung unterscheiden, werden als **verschieden** angesehen. Alle Bezeichner, die im gleichen Sichtbarkeitsbereich sind, teilen sich einen gemeinsamen Namensraum. Es ist z. B. nicht möglich Prozeduren mit einem Bezeichner zu definieren, der bereits für eine Variable genutzt wird. Ein Bezeichner kann grundsätzlich nicht vor seiner Deklaration verwendet werden. Die Reihenfolge im Quelltext ist ausschlaggebend.

Laut Sprachstandard werden Bezeichner so gebildet:

Bezeichner = (Großbuchstabe | Kleinbuchstabe), {Großbuchstabe | Kleinbuchstabe | Ziffer};

Großbuchstabe = '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';

---

[1] Gleichwohl haben Projekte, welche die Sprache Oberon verwenden, gelegentlich Konventionen, die die Wahl der Bezeichner über den Sprachstandard hinaus beschränkt. Diese Regeln gelten aber nicht universell.

[2] Im Englischen ist dieser Typ auch als railway diagram bekannt.

[3] Es werden beispielsweise formale Parameter im Syntaxdiagramm an dem verwendenden Syntaxelement nicht als optional notiert. Stattdessen ist die Parameterangabe so gezeichnet, dass sie selbst vollständig leer sein kann. Für die definierte Sprache sind beide Fälle gleichwertig.

Kleinbuchstabe = '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';

Die obige Darstellung wird im Dokument für alle Grammatikregeln verwendet. Die als EBNF bezeichnete Notation entspricht dem Standard ISO/IEC 14977:1996(E). Wegen der Verwendung von Kursivschrift, soll hier erwähnt sein, dass dar kursive Rückstrich (\) dem kursiven senkrechten Strich (|) ähnlich sieht. Allerdings kommt ersteres Zeichen in der Syntax von Oberon nicht vor, so dass keine Verwechslung möglich ist.

Eine vollständige Aufstellung der Syntax ist am Ende des Dokuments zu finden. (Siehe Seite 30.) Wird dieses Dokument am Bildschirm gelesen, dann sind darüber hinaus die Symbole an ihren Verwendungsstellen zu den Definitionsstellen im Haupttext verknüpft. Es ist also möglich, zu den Beschreibenden Texten einer Syntaxregel zu springen, wenn deren Symbol in anderen Regeln verwendet wird. Steht ein Symbol in unmittelbarer Nähe zu seiner Definition, wurde auf die Verknüpfung verzichtet. In der alphabetischen Aufstellung aller Syntaxregeln am Ende des Dokuments gibt es ebenfalls keine Verknüpfungen.

1.1.1 Beispiele

Oberon07

PI

pi

o023E3

langerCamelCaseBezeichner

1.2 Schlüsselwörter, Operatoren und Begrenzer

Oberon enthält die folgenden Terminalsymbole, die in der Grammatik als Schlüsselwörter, Operatoren oder Begrenzer fungieren.

(	*)*	*&*	*ARRAY*	*END*	*MOD*	*RETURN*
{	*}*	*^*[1]	*BEGIN*	*EXIT*[2]	*MODULE*	*THEN*
[	*]*	*~*	*BY*[3]	*FALSE*	*NIL*	*TO*
.	*,*	*|*	*CASE*	*FOR*[4]	*OF*	*TRUE*
+	*-*	***	*CONST*	*IF*	*OR*	*UNTIL*
/	*;*	*:*	*DIV*	*IMPORT*	*POINTER*	*VAR*
<	*=*	*>*	*DO*	*IN*	*PROCEDURE*	*WHILE*
<=	*#*	*>=*	*ELSE*	*IS*	*RECORD*	*WITH*[5]
:=	*..*		*ELSIF*	*LOOP*[6]	*REPEAT*	

---

1 Das Zirkumflex (^) hat historisch auch die Darstellungsform als Pfeil (↑). Oberon System verwendet den Pfeil.

2 Nur im klassischen Oberon

3 Nur in Oberon-07

4 Nur in Oberon-07

5 Nur im klassischen Oberon

6 Nur im klassischen Oberon

Schlüsselwörter können nicht als Bezeichner genutzt werden. Folgen Bezeichner oder Schlüsselwörter im Quelltext direkt aufeinander, muss ein Begrenzer oder Kommentar zwischen ihnen platziert werden. Ein Begrenzer mit dem Charakter eines Leerzeichens kann beliebig oft vor oder nach einem beliebigen Symbol stehen, ohne die Bedeutung des Programms zu modifizieren.

1.3 Kommentare

Kommentare dürfen beliebige Zeichen enthalten und werden durch die Zeichensequenzen *(** und ***)* eingeschlossen. Im Bericht zu Oberon-07 ist explizit angegeben, dass Kommentare geschachtelt werden können. Ist also innerhalb eines Kommentars ein weiteres *(** enthalten, so muss je ein zusätzliches ***)* im Kommentar enthalten sein, bevor der Kommentar mit einem darauf folgenden ***)* geschlossen werden kann.

Ein Kommentar kann zwischen je zwei beliebigen Symbolen und am Anfang und Ende des Programmtextes stehen. Der Kommentar hat in der EBNF-Syntaxbeschreibung keine Entsprechung, sondern steht außerhalb der Syntax.

1.3.1 Beispiele

```
(* Ein einfacher Kommentar *)

(* Ein Kommentar wie (*dieser*) ist geschachtelt. *)

(*
  ---------------------------------
  Ein mehrzeiliger Kommentar
  ---------------------------------
*)

(** Das Schlüsselwort POINTER (und andere) haben in
 ** Kommentaren keine Funktion, sondern werden wie
** jeder andere Text behandelt.
**********************************************)
```

2 Module

Ein Modul ist in Oberon eine separat übersetzte Programmeinheit. Ein Modul kann von mehreren anderen Modulen verwendet werden. Ein Abhängigkeitszyklus, bei dem ein Modul A direkt oder indirekt von einem Modul B verwendet wird, das selbst das Modul A verwendet, ist verboten.

Außerhalb von Modulen gibt es keine weiteren Einheiten, die Definitionen oder Code enthalten können. Oberon kennt keine Pakete oder Supermodule.

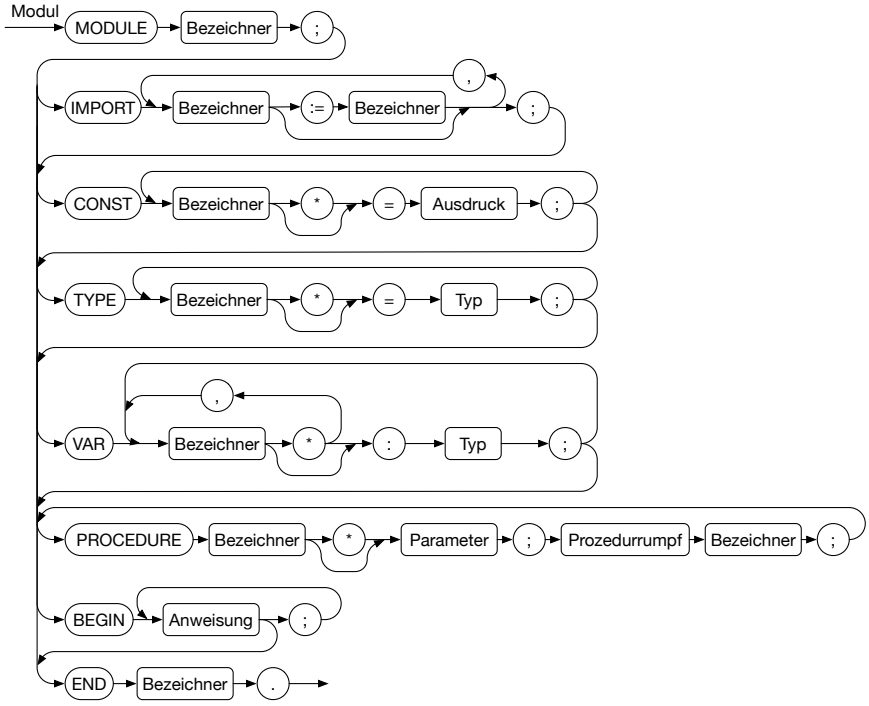

Abbildung 2: Syntaxdiagramm für Modul

2.1 Definition von Modulen

Die Definition eins Moduls erfolgt gemäß der folgenden Regel:

Modul = 'MODULE', Bezeichner, ';', [Importliste],Deklarationsfolge, ['BEGIN', Anweisungsfolge], 'END', Bezeichner, '.';

2.2 Importierte Module

Der Bezeichner eines Moduls ist mit denen anderer Module immer in einem gemeinsamen und nicht weiter strukturierten Namensraum. Dieser Umstand kann dazu führen, dass ein Modul umbenannt werden muss, wenn ein von einem Projekt in ein anderes übernommen wird, damit es im neuen Projekt nicht zu Bezeichnerkonflikten kommt. Das Refactoring ist bei einer Umbenennung allerdings nicht aufwendig, weil beim Import optional ein lokaler Bezeichner für das Modul angegeben werden kann. Lokale Bezeichner für Module werden auch oft verwendet, um Schreibarbeit zu sparen, bzw. den Quellcode übersichtlicher zu erscheinen zu lassen.

Importliste = 'IMPORT', Bezeichner, [':=', Bezeichner], {',', Bezeichner, [':=', Bezeichner]}, ';';

Exportierte Deklarationen eines importierten Moduls lassen sich nach dem Import im Programmtext über den Bezeichnerpfad nutzen.

2.3 Deklarationen

In einem Modul werden typischerweise eine Menge von Prozeduren (und Unterprozeduren) deklariert. Das Modul und die Prozeduren im Modul haben jeweils ihre eigene Deklarationsfolge, die immer dem gleichen Aufbau hat:

Deklarationsfolge = ['CONST', {Konstantendefinition, ';'}], ['TYPE', {Typdeklaration, ';'}], ['VAR', {Variablendeklaration, ';'}], {Prozedurdefinition, ';'};

Eine Unregelmäßigkeit innerhalb der Syntax ist, dass ein Semikolon **nach** einer Deklaration bzw. Konstantendefinition steht. Innerhalb einer Anweisungsfolge steht dieser Trenner **zwischen** den Anweisungen.

2.3.1 Export und Bezeichnerpfade

Bei der Deklaration können Konstanten, Typen, Variablen und Prozeduren aus der Modulebene exportiert werden, indem hinter dem betreffenden Bezeichner ein Stern (*) angegeben wird.

Lokal (also innerhalb von Prozeduren) deklarierte Bezeichner sind auf der Modulebene grundsätzlich nicht sichtbar und können nicht exportiert werden. Exportierte Bezeichner können hingegen in den Modulen, die das exportierende Modul importieren, an jeder Stelle genutzt werden. Bei der Nutzung wird dem importierten Bezeichner der Bezeichner des Moduls[1] vorangestellt, in dem ersterer Bezeichner ursrünglich definiert wurde:

Bezeichnerpfad = Bezeichner, ['.', Bezeichner];

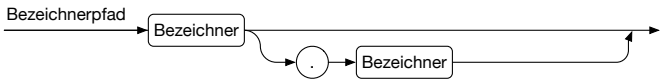

Abbildung 3: Syntaxdiagramm für Typ

2.4 Laden von Modulen und Ausführen von Programmen

Das Konzept eines Programms ist im Oberon System anders realisiert als unter anderen Betriebssystemen[2]. Ein Modul wird nur dann geladen, wenn eine exportierte Prozedur aufgerufen wird. Der äußere Block eines Moduls wird beim Laden dieses Moduls einmalig ausgeführt. Im Anschluss daran folgt die Ausführung des Codes der aufgerufenen Prozedur. Im Oberon System können exportierte Prozeduren, die keine formalen Parameter definieren und keinen Rückgabewert haben, direkt aus der User-Shell aufgerufen werden. (Wird der Bezeichnerpfad der Prozedur als Text in einem beliebigen Fenster mit der mittleren Maustaste angeklickt startet der Aufruf). Dieses Verfahren der Programmausführung wird von den Compilern auf anderen Systemen (zumindest bislang) nicht nachgebildet. Stattdessen wird zumeist bei der Übersetzung ein Modul ausgewählt, das als Hauptmodul betrachtet wird. Dessen äußerer Block fungiert als Hauptprogramm. Wir es beendet, so endet die gesamte Bearbeitung und alle bei der Ausführung geladenen Module werden wieder aus dem Speicher entfernt.

---

[1] Bzw. der lokale Modulbezeichner aus der IMPORT-Anweisung

[2] Beispiele wären AmigaOS, BeOS, CP/M, OS/2, RISCOS, TOS, Windows und UNIX.

3 Konstanten, Werte und Literale

Literale und Konstanten sind Daten, deren Wert zum Zeitpunkt der Programmübersetzung feststeht. Eine Konstante ist dabei ein Bezeichner, der aus einer Kombination anderer Konstanten und Literale gebildet werden kann.

3.1 Konstantendefinition

Eine Konstantendefinition enthält daher einen Ausdruck, der stärkeren Beschränkungen unterliegt als an anderen Programmstellen. Diese Einschränkungen sind allerdings überwiegend nicht syntaktisch. Folglich wird ein konstanter Ausdruck in den meisten Darstellungen mit einem gewöhnlichen Ausdruck gleichgesetzt, obwohl das eine Vereinfachung darstellt.

Konstantendefinition = Bezeichner, [''], '=', Ausdruck;*

Oberon unterstützt keine strukturierten Konstanten. Eine Datentypangabe für Konstanten ist ebenfalls nicht möglich. Da es außerdem keinen Aufzählungstypen (enum) oder Bereichstypen gibt, werden oft numerische Konstanten verwendet, die mit einem ganzzahligen Zahlenliteral definiert werden.

3.2 Ganzzahlige Zahlenliterale

Ganze Zahlen können in Oberon – je nach Bedarf - dezimal oder hexadezimal angegeben werden. Dabei enden alle hexadezimalen Zahlen als Postfix mit dem großen Buchstaben H.

ganze Zahl = Ziffer, ({Ziffer}|{hexadezimale Ziffer}, 'H');

hexadezimale Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

In Oberon sind für hexadezimale Zahlen keine Kleinbuchstaben zugelassen;

3.3 Dezimalbrüche

Oberon hat die Möglichkeit, Berechnungen im Raum der reellen Zahlen mit einer definierten Genauigkeit[1] durchzuführen. Literale für solche Operationen verwenden als Notation Dezimalbrüche. Bei Bedarf kann eine Darstellung mit Exponent (Basis 10) verwendet werden.

Dezimalbruch = Ziffer, {Ziffer},'.' {Ziffer}, [Skalierung];

Skalierung = 'E', ['+' | '-'], Ziffer, {Ziffer};

3.4 Mengenliterale

Eine in Akkoladen[2] gefasste Liste von Elementen, die zu nicht-negativen ganzen Zahlen ausgewertet werden können, bildet einen Mengenliteral.

Menge = '{', [Element, {',', Element}], '}';

Element = Ausdruck, ['..' Ausdruck];

---

[1] Da klassisches Oberon zwischen *REAL* und *LONGREAL* Typen unterscheidet, gibt es dort die Möglichkeit ein Dezimalbruch-Literal mit einem *D* statt einem *E* zu notieren, um den Compiler anzuzeigen, dass eine höhere Genauigkeit benötigt wird. Oberon-07 enthält nur noch *REAL*.

[2] Akkoladen werden auch Geschweifte Klammern, geschwungene Klammern oder „Nasenklammern“ genannt.

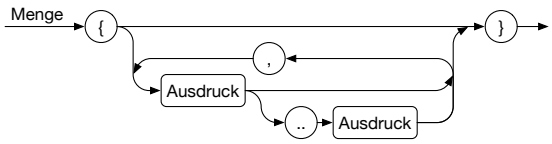

Abbildung 4: Syntaxdiagramm für Menge

Viele Compiler beschränken den maximalen Wert für ein Element einer Menge im Programmablauf auf 31 oder 63. Ein korrekt implementierter Compiler sollte keine Mengenliterale zulassen, die mit Variablen vom Datentyp *SET* nicht zuverlässig verwendet werden können. Leider ist dies nicht immer der Fall.

3.5 Wahrheitswerte und Referenzen ohne Ziel

Als boolesche Werte stehen die beiden Schlüsselwörter *TRUE* und *FALSE* zur Verfügung.

Variablen, die als *Zeigertyp* oder *Prozedurtyp* deklariert sind, haben einen speziellen Wert, der Anzeigt, dass die Variable aktuell nicht verwendet wird. Ein Zeiger der nicht auf Daten zeigt und eine Prozedurreferenz, die auf keine Prozedur verweist, können mithilfe des Schlüsselwortes *NIL*[1] diesen „Nullwert“ zugewiesen bekommen und mit ihm verglichen werden um sie zu identifizieren.

3.6 Zeichen- und Zeichenketten

Es gibt zwei Arten von Zeichenliteralen. Am häufigsten wird die Variante mit den *geraden Anführungszeichen*[2] (*"*) verwendet. Steuer- und Sonderzeichen können mit dieser Notation in vielen Fällen nicht angegeben werden[3].

Die zweite Variante ist eine hexadezimaler Zeichencode, der mit einem *X* statt einem *H* abgeschlossen wird. Die meisten Implementierungen erlauben hier nur Werte unter *100X* ohne explizite Nennung der Zeichenkodierung[4].

Zeichenketten sind eine (potenziell auch leere) Folge von beliebigen in Anführungszeichen eingeschlossenen Zeichen. Es gelten die oben genannten Einschränkungen an Zeichenliterale auch in der Zeichenkette. Die Anzahl der Zeichen in der Kette wird als Länge der Zeichenkette bezeichnet. Einzelne Zeichen sind immer auch Zeichenketten der Länge eins.

Zeichenkette = '"', {irgendein Zeichen - '"'}, '"' | (Ziffer, {hexadezimale Ziffer}, 'X');

Für das EBNF-Symbol *irgendein Zeichen* in der obigen Syntaxregel ist keine Definition vorgesehen. Ein Oberon Compiler kann hier also mehr Zeichen zulassen, als für die Grammatik sonst erforderlich wären.

---

[1] *NIL* ist kurz für „nihil“ (lat. nichts). Obwohl im Maschinencode vom Compiler oft mit der Null realisiert, hat *NIL* in Oberon keinen Zahlencharakter.

[2] Gemeint ist das Zeichen 22X (ASCII-Code 34); U+0022 *Quotation Mark.*

[3] Anders als in anderen Sprachen ist beispielsweise *"\n"* kein einzelnes Zeichen sondern eine Zeichenkette aus dem Zeichen *"\"* und dem Zeichen *"n"*.

[4] Für Werte bis 7FX in der Regel ASCII. Danach oft ISO/IEC 8859-1.

3.7 Beispiele für Konstantendefinitionen und Literale

```
CONST

(* Mit Literal *)
Kreiszahl = 3.1415926536; (* ca. Pi *)
r = 0.2025E4;

(* Mit Ausdruck (Multiplikation) *)
A = Kreiszahl * r * r; (* Flächeninhalt *)

(* Exportierte ASCII Konstanten *)
CR* = 0DX; LF* = 0AX;

Puffergroesse = 07FFFH + 1; (* > SHORTINT *)

Nachricht = "Hallo Welt!";

keineVier = {1,2,3,5 .. 31};

(* Auch exportiert *)
Ja * = TRUE;
Nein * = ~Ja;
```

4 Datentypen

Alle Variablen sind in Oberon-Programmen einem Datentyp fest zugeordnet. Durch den Compiler wird bei Operationen mit Variablen stets die Typkompatibilität sichergestellt. Für die meisten Fälle erfolgt die Prüfung der Kompatibilität zur Übersetzungszeit. Für Typerweiterungen und Wertebereichsverletzungen werden Überprüfungen in den ausführbaren Code eingefügt, die zur Laufzeit das Programm beenden, wenn ein Fehler festgestellt wird.

4.1 Einfache Datentypen

Oberon-07 verfügt über sechs eingebaute „einfache“ Datentypen: *BOOLEAN*, *BYTE*, *CHAR*, *INTEGER*, *REAL* und *SET*. Die Wertebereiche gibt der Sprachstandard nur für *BOOLEAN* (*TRUE* oder *FALSE*) und für *BYTE* (0 ... 255) vor. *BYTE* und die Elementwerte für *SET* sind vorzeichenlos. *INTEGER* und *REAL* sind vorzeichenbehaftet.

Ganzzahlige Werte können in Variablen vom Typ *INTEGER* oder *REAL* gespeichert werden. Für Werte und Berechnungen mit reellen Zahlen ist der Typ *REAL* vorgesehen. *SET* ist ein einfacher Mengentyp für ganze Zahlen. (Siehe Seite 9.)

Klassisches Oberon kennt darüber hinaus noch *SHORTINT*, *LONGINT* und *LONGREAL*, die sich von *INTEGER* bzw. *REAL* darin unterscheiden, dass Hardwareeigenschaften des genutzten Systems berücksichtigt werden: Die häufiger genutzten Typen werden mit weniger Bits[1] realisiert. Um nicht auf den erweiterten Wertebereich bzw. die erhöhte Genauigkeit verzichten zu müssen, stehen dann zusätzlich die Long-Varianten zur Verfügung. *BYTE* ist in klassischem Oberon kein vordeklarierter Datentyp, sondern nur als *SYSTEM.BYTE* verfügbar.

[1] Beispielsweise wäre auf einer 16-Bit-CPU ein *INTEGER* typischerweise zwischen -32.768 und +32.767, während ein *LONGINT* Werte zwischen -2.147.483.648 und 2.147.483.647 annehmen könnte.

4.2 Abgeleitete Datentypen

Auf der Basis existierender Datentypen können weitere Datentypen deklariert werden. Im einfachsten Fall wird nur ein neuer Name für einen bestehenden Typ eingeführt. Darüber hinaus können auch strukturierte Datentypen und Datentypen mit Referenzcharakter deklariert werden.

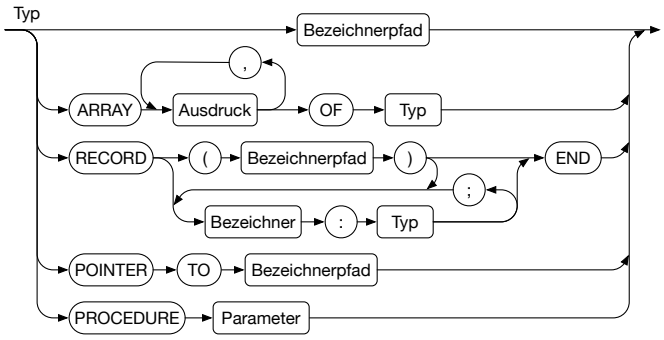

Abbildung 5: Syntaxdiagramm für Typ

Typen werden wie folgt deklariert:

Typdeklaration = Bezeichner, [''], '=' Typ;*

Typ = Bezeichnerpfad | ArrayTyp | Datensatztyp | Zeigertyp | Prozedurtyp;

4.2.1 Homogener Datenbehälter

Eine Datenstruktur, bei der alle Felder von gleicher Art sind, kann als Array-Typ realisiert werden. Ein Array-Typ hat in Oberon immer eine feste Größe[1].

ArrayTyp = 'ARRAY', Ausdruck, {',', Ausdruck}, 'OF', Typ;

Auf ein Array kann Feldweise über einen Index zugegriffen werden. (Siehe Seite 20.) Sind in der Deklaration mehrere Größenangaben (durch Komma getrennt) vorhanden, so handelt es sich um ein Mehrdimensionales Array. Die Dimensionen werden stets in der gleichen Reihenfolge behandelt.

4.2.2 Heterogene Datensätzen

Um verschiedengeartete Daten zu gruppieren und gemeinsam zu behandeln, bietet Oberon mit *RECORD* einen Typ für Datensätze an. Die Datenanteile werden ebenso wie beim Array als Felder bezeichnet. Die Felddefinitionen sind syntaktisch Variablendefinitionen. Erweiterte Typen können die Bezeichner von Feldern des Basistyps weder überlagern noch deklarieren.

Datensatztyp = 'RECORD', ['(', Bezeichnerpfad, ')'], [Variablendeklaration, {';', Variablendeklaration}], 'END';

Auf die Felder eines Datensatzes kann durch das Anhängen eines Punktes über ihren Bezeichner zugegriffen werden. (Siehe Seite 20.)

Wird bei der Deklaration eines Datensatzes ein Bezeichnerpfad angegeben, so handelt es sich um eine Typerweiterung des bezeichneten Datensatzes. Den Typ der in Klammern angegeben

[1] In Oberon-2 kann die Größe zur Laufzeit bestimmt werden.

wird bezeichnet man als *Basistyp*. Hat eine Variable einen erweiterten Typ, dann kann sie an jeder Stelle verwendet werden, an der der Basistyp gültig ist. Zusätzlich zu den Feldern des Basistyps werden die Felder der neuen Deklaration für den Zugriff verfügbar, wenn die Variable auf diese Weise als Erweiterung deklariert ist.

4.2.3 Zeiger

Mithilfe von Zeigertypen werden dynamische Variablen realisiert, die zur Programmlaufzeit nach Bedarf erzeugt und durch die automatische Speicherbereinigung (garbage collection) wieder entfernt werden können, wenn kein Zeiger mehr auf die Variable verweist, der über einen noch aktiven Sichtbarkeitsbereich erreichbar ist. In Oberon-07 können dynamische Variablen nur als *RECORD* definiert werden. Zeiger für andere Typen sind nicht möglich.

Zeigertyp = 'POINTER', 'TO', Typ;

Um einem Zeiger zu folgen und Operationen auf seinem Ziel zu erreichen, wird ein Zirkumflex (^) verwendet. (Siehe Seite 20.)

4.3 Prozedurverweise

War es in Pascal noch so, dass Prozeduren (und Funktionen) zwar als Parameter an andere Prozeduren übergeben werden konnten, aber nicht in anderen Variablen für eine spätere Nutzung gespeichert werden konnten, sind in Oberon Prozeduren gleichzeitig auch vollwertige Datentypen.

Prozedurtyp = 'PROCEDURE', [Parameter];

Die (formalen) Parameter werden bei einem Prozedurverweis wie folgt angegeben. Diese Form entspricht der Angabe bei der Signatur, die zu der Prozedurdefinition gehört. (Siehe Seite 16.) Prozedurverweise können nur auf Prozeduren verweisen, deren Parameter semantisch identisch definiert sind. Die Bezeichner der einzelnen Komponenten sind dabei nicht erheblich.

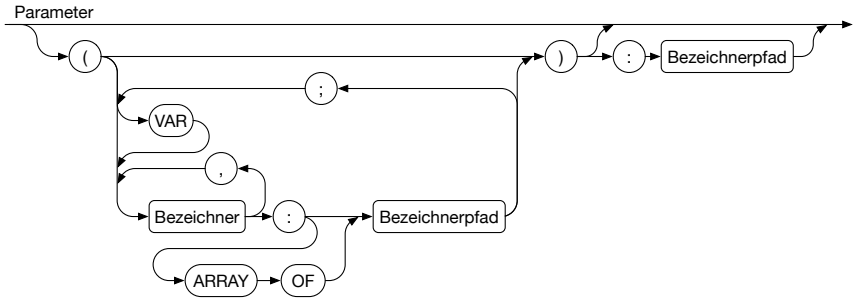

Abbildung 6: Syntaxdiagramm für Parameter

4.4 Beispiele

```
TYPE
Schachfigur* = INTEGER;
Feldfarbe* = INTEGER;
Spielfeld = POINTER TO SpielfeldDaten;
SpielfeldDaten = RECORD
        besetzt: BOOLEAN;
        Farbe: Feldfarbe
        END;
DameDaten = RECORD (SpielfeldDaten)
        AnzahlDerSteine: INTEGER
        END;
SchachDaten* = RECORD (SpielfeldDaten)
        Figur*: Schachfigur;
        aufGrundlinie: BOOLEAN
        END;
Spielbrett* = ARRAY 8, 8 OF Spielfeld;
```

4.5 Typregeln

Die Typkompatibilität von Prozedurtypen orientiert sich an der Signatur der Prozedur. D.h. wenn eine Prozedur die gleiche Anzahl an Parameter mit den gleichen Argumenttypen hat und sich auch im Rückgabetyp nicht unterscheidet, ist sie mit dem Prozedurtyp nutzbar.

Bei numerischen Typen gibt es im klassischen Oberon eine nicht-permutative Inklusion:

SHORTINT ⊆ *INTEGER* ⊆ *LONGINT* ⊆ *REAL* ⊆ *LONGREAL*

Ein enthaltener Typ kann also einer Variable vom Typ der Obermenge zugewiesen werden, während das Gegenteil nicht der Fall ist.

Diese Regel gilt in Oberon-07 nicht. Es ist keine direkte Zuweisung zwischen REAL und INTEGER möglich. Es stehen aber eingebaute Funktionsprozeduren für die Konversion zur Verfügung. (Siehe Seite 26.) Der Typ *BYTE* kann in Oberon-07 einem INTEGER direkt zugewiesen werden. Auch die entgegengesetzte Zuweisung ist erlaubt[1].

5 Variablen

Globale Variablen haben einen schlechten Ruf, weil ihre Änderungen so weit verstreut sein können, dass bestimmte Fehler schwer zu finden sind. Das was man in Oberon allerdings als global bezeichnet, ist in Wirklichkeit immer in einem Modul gekapselt und somit in einem eingeschränkten Sichtbarkeitsbereich. Mehr noch: in Oberon-07 sind selbst exportierte Variablen vor direkten Schreibzugriffen aus fremden Modulen geschützt. Die Unterscheidung von globalen Variablen und lokalen Variablen, deren Gültigkeit mit dem Verlassen einer Prozedur erlischt, ist dennoch erforderlich.

---

[1] Der Oberon-07 Compiler im Oberon System führt eine Modulo-Operation durch, wenn der Wertebereich nicht ausreicht.

Deklariert werden Variablen stets mit einer Typangabe:

Variablendeklaration = Bezeichner, [''], {',', Bezeichner, ['*']}, ':', Typ;*

5.1 Beispiele

VAR

i, j, k : INTEGER;

Compilerschalter : SET;*

Text: ARRAY laengenKonstante OF CHAR;

Affe: POINTER TO Baum.Knoten;

5.2 Sichtbarkeitsbereiche und Initialisierungen

Variablen, die im selben Sichtbarkeitsbereich deklariert werden, müssen unterschiedliche Bezeichner verwenden. Variablen, die im Rahmen einer Prozedur deklariert werden, können Bezeichner des sie umgebenden Bereichs für andere, lokale Variablen verwenden. Die Variable aus der Umgebung wird dabei dem Zugriff der Prozedur selbst und den Unterprozeduren entzogen.

Diese Regeln zu den Sichtbarkeitsbereichen gelten auch für die Bezeichner von Prozeduren Typen und Konstanten: Grundsätzlich können lokale Deklarationen in einem Sichtbarkeitsbereich bestehende Deklarationen aus anderen Sichtbarkeitsbereichen verdecken, wenn sie den gleichen Bezeichner haben. Es kann beispielsweise auch der Bezeichner einer Konstante durch einen Bezeichner eines Typs verdeckt werden.

Bei ihrer Deklaration haben Variablen grundsätzlich keinen festgelegten Wert. Die Initialisierung muss innerhalb des Sichtbarkeitsbereichs der Variablen in Form einer (oder mehrerer) Anweisung(en) erfolgen, bevor die Variable anderweitig verwendet wird. Bei strukturierten Variablen gilt dies auch für alle enthaltenen Felder.

6 Prozeduren

Wie schon im vorangegangenen Abschnitt angedeutet, gehört zu einer Prozedurdefinition eine Deklarationsfolge in der lokale Konstanten, Typen, Variablen, und Prozeduren enthalten sein können. Diese sind im Sichtbarkeitsbereich, den die Prozedur eröffnet. Aber auch die Parameter, die zusammen mit der Prozedur deklariert werden, sind Variablen, die in diesem Sichtbarkeitsbereich gültig sind.

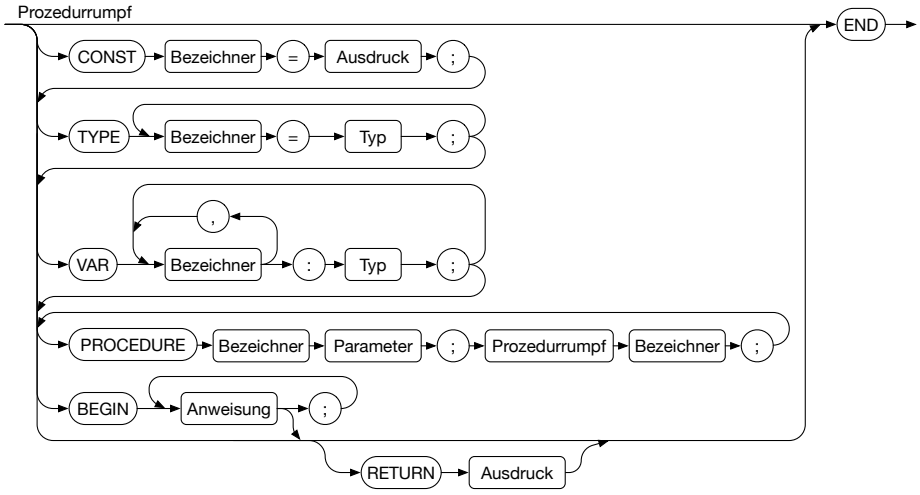

Abbildung 7: Syntaxdiagramme für den Prozedurrumpf

Prozedurdefinition = Prozedurkopf, ';', Prozedurrumpf, Bezeichner;

Prozedurkopf = 'PROCEDURE', Bezeichner, [''], [Parameter];*

Prozedurrumpf = Deklarationsfolge, ['BEGIN', Anweisungsfolge], ['RETURN' Ausdruck], 'END';

Prozeduren können mit einen Rückgabewert deklariert werden[1]. Prozeduren mit deklariertem Rückgabewert (Funktionsprozeduren) können in beliebigen Ausdrücken aufgerufen werden und haben am Ende ihrer Deklaration einen mit *RETURN* eingeleiteten Ausdruck, der den Rückgabewert festlegt. In klassischem Oberon ist *RETURN* eine Anweisung, die an beliebigen Stellen in der Funktion verwendet werden kann.

Die Parameter einer Prozedur werden wie folgt spezifiziert.

Parameter = '(', [Parameterabschnitt, {';', Parameterabschnitt}], ')', [':', Bezeichnerpfad];

Parameterabschnitt = ['VAR'], Bezeichner, {',', Bezeichner}, ':', {'ARRAY OF'}, Bezeichnerpfad;

Parameter, die mit *VAR* deklariert werden, können verwendet werden, um die Werte von Variablen des aufrufenden Sichtbarkeitsbereichs zu ändern. Oberon-07 und klassisches Oberon unterscheiden sich bei der Behandlung der Parameter ohne *VAR*: Während in klassischem Oberon für diese Parameter stets eine lokale Variable geführt wird, die mit dem Wert des übergebenen Arguments initialisiert wird, verwendet Oberon-07 für Record- und Array-Ty-

[1] In Pascal werden solche Prozeduren als *Funktionen* bezeichnet und mit dem Schlüsselwort *function* deklariert.

pen keine zusätzlichen Variablen, sondern verhindert nur Schreibzugriffe über den Parameter-Bezeichner.

Wird ein Parameter mit *ARRAY OF* ohne Dimensionsgröße angegeben, so können als Argument gleich typisierte Arrays beliebiger Größe verwendet werden. Im anderen Fall gilt die Größenangabe in der Typdeklaration (*ArrayTyp*).

Für den hinter dem Doppelpunkt angegebenen Rückgabetyp einer Funktionsprozedur sind Array- und Record-Typen ausgeschlossen und bei Bedarf muss ersatzweise ein Zeigertyp verwendet werden.

Die Deklarationsstelle einer Prozedur bestimmt deren Sichtbarkeitsbereich in der gleichen Weise, wie die Deklarationsstelle einer Variablen deren Sichtbarkeitsbereich bestimmt.

Prozeduren ohne Rückgabewert werden nur in Anweisungen aufgerufen, Funktionsprozeduren werden aus Ausdrücken heraus aufgerufen. (Siehe Seite 23.) Alle Prozeduren können über ihren Bezeichner in Ausdrücken verwendet werden, ohne dass ein Aufruf erfolgt. In diesem Fall wird ein Prozedurverweis verwendet. (Siehe Seite 13.) Um diesen Fall eindeutig zu machen, wird für den Aufruf in Ausdrücken grundsätzlich ein folgendes Klammerpaar gefordert, selbst wenn die Funktionsprozedur keine Argumente hat bzw. ohne Parameter deklariert wurde.

Das überladen von Prozedurbezeichnern ist nicht möglich. Es können also im gleichen Sichtbarkeitsbereich nicht zwei Prozeduren mit gleichen Namen und unterschiedlichen Parametertypen deklariert werden[1].

Prozeduren rufen sich unter Umständen gegenseitig auf, aber nur bereits deklarierte Prozeduren können direkt aufgerufen werden. In Oberon-07 müssen Prozedurverweise verwendet werden um das resultierende Henne-Ei-Problem zu lösen. (Siehe Seite 13.) Im klassischen Oberon gibt es die Form einer Vorwärtsdeklaration, bei der eine Prozedur bekannt gemacht werden kann, obwohl die Deklaration erst weiter unten im Quelltext erfolgt.

Vorwärtsdeklaration = 'PROCEDURE', ,'^' , Bezeichner, [''], [Parameter];*

[1] Polymorphie im Sinne der Objektorientierung ist aber durch Prozedurverweise indirekt realisierbar.

6.1 Beispiele

Die in den Beispielen enthaltenen Anweisungen werden im nächsten Abschnitt behandelt.

```
PROCEDURE Maximum (a, b: REAL): REAL;
BEGIN
        IF a < b THEN b := a END
RETURN b END Maximum;

PROCEDURE inGrossbuchstaben(VAR k: ARRAY OF CHAR);
VAR pos: INTEGER;
BEGIN pos := 0;
        WHILE pos < LEN(k) DO
                CASE k[pos] OF
                0X .. "@": (* NOP *)
                |"a" .. "z": k[pos]:= CHR(ORD(k[pos])-32)
                |"ä": k[pos]:="Ä" |"ö": k[pos]:="Ö" |"ü": k[pos]:="Ü"
                |"ß": k[pos]:="ẞ" (* ist nicht in Latin1 enthalten *)
                | "{" .. 1FFFFFX: (* Im Beispiel gilt: CHAR  hat 21 Bit *)
                END;
        INC(pos) END
END inGrossbuchstaben;
```

7 Anweisungen

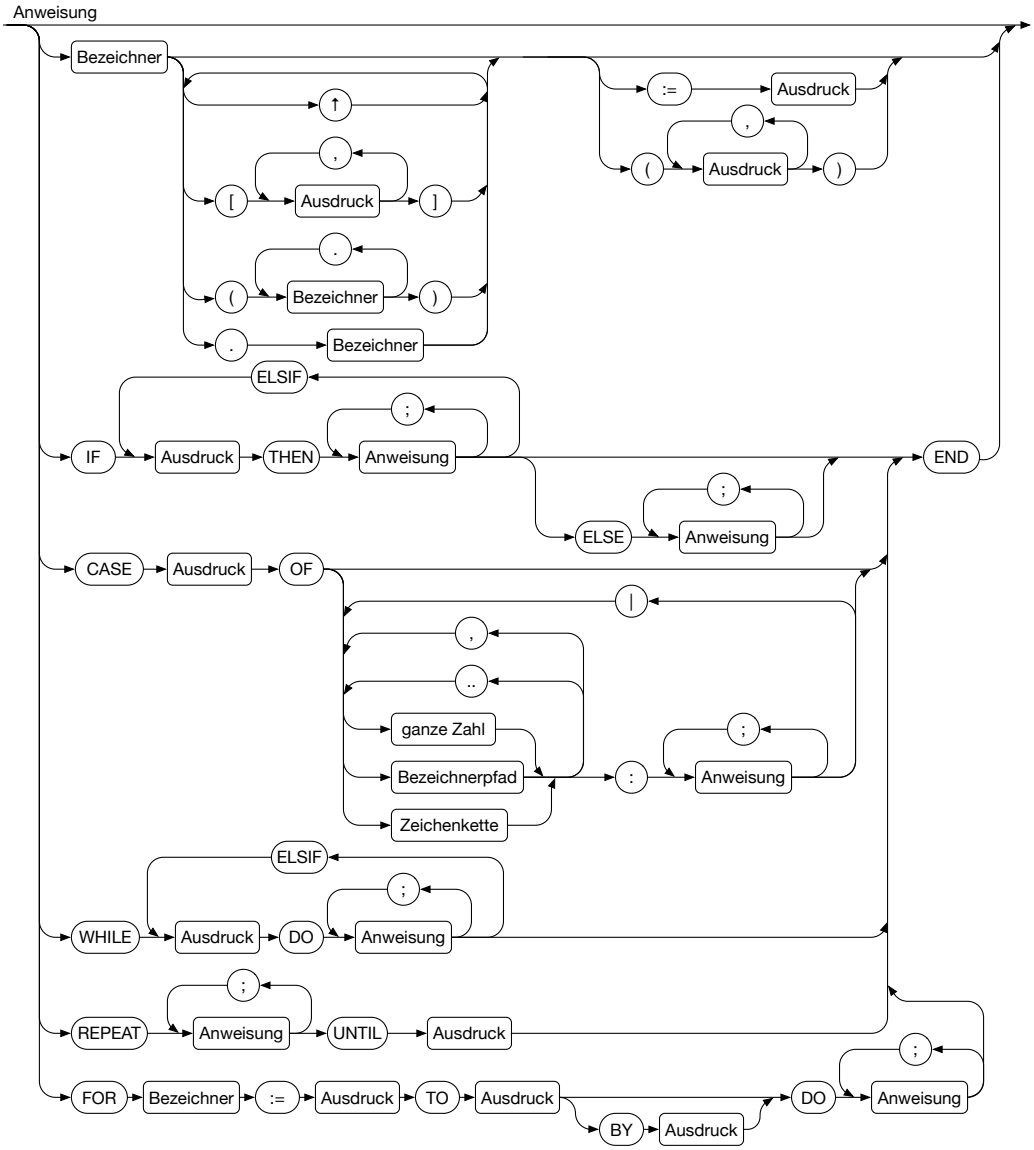

Abbildung 8: Syntaxdiagramm Anweisung

In einer Anweisungsfolge ist das Trenner-Zeichen zwischen zwei Anweisungen das Semikolon[1]. Oberon-07 kennt sieben Arten von Anweisungen:

Anweisungsfolge = Anweisung, {';', Anweisung};

Anweisung = [Zuweisung | Prozeduraufruf | IfAnweisung | CaseAnweisung | WhileAnweisung | RepeatAnweisung | ForAnweisung];

Darüber hinaus hat klassisches Oberon noch eine Return-Anweisung und eine Exit-Anweisung. (Siehe Seite 25.)

[1] Ein beliebter Stil ist es, als letzte Anweisung einer Anweisungsfolge eine leere Anweisung zu wählen, so dass im Quelltext immer ein Semikolon am Ende steht.

7.1 Zuweisungen

Die vielleicht grundlegendste Anweisung ist die Zuweisung. Um einer Variablen einen neuen Wert zuzuweisen, muss sie in der Regel vollständig spezifiziert sein: Wenn sie innerhalb eines *ARRAY* oder einem *RECORD* liegt, ist ein Selektor[1] erforderlich. Das gleiche gilt, wenn einem Zeiger gefolgt wird.

Zuweisung = Instanz, ':=', Ausdruck;

Instanz = Bezeichnerpfad, {Selektor};

Ein *ARRAY OF CHAR* kann über eine Zuweisung mit dem Inhalt einer Zeichenkette gefüllt werden. Dabei muss die Größe so dimensioniert sein, dass jedes Zeichen der Zeichenkette und ein Nullzeichen am Ende (*0X*) Platz finden.

Sind Records oder Arrays vom gleichen Typ, können ihre Variablen in Gänze zugewiesen werden. Daten eines erweiterten Datensatzes können einer Variablen eines seiner Basistypen zugewiesen werden, aber nicht umgekehrt. Zusätzlich ist es in Oberon-07 möglich Arrays, deren Typen kompatibel sind, mit einer einfachen Zuweisung zu kopieren, wenn das Ziel nicht kürzer als die rechte Seite der Zuweisung ist.

Für das Ziel einer Zuweisung ist die Angabe eines Moduls im Bezeichnerpfad in Oberon-07 nicht zulässig, da exportierte Variablen nur gelesen und nicht verändert werden dürfen. Diese Beschränkung besteht in klassischem Oberon nicht.

7.1.1 Selektoren

Um eine Instanz genau spezifizieren und auf die Felder eines Arrays oder Records zugreifen zu können ist in vielen Fällen ein Selektor erforderlich. Aber auch ein sogenannter Typwächter wird über eine solche Angabe realisiert, die dem Variablenbezeicher hintangestellt wird.

Selektor = ('.', Bezeichner) | ('[', Ausdruck, {',',Ausdruck}, ']') | '^' | ('(', Bezeichnerpfad, ')');

Wrid einem Datensatzbezeichner ein Selektor nachgestellt, der mit einem Punkt beginnt, so soll der folgende Bezeichner dem Namen eines Felds in diesem Record entsprechen. Ist ein Selektor ein Ausdruck in eckigen Klammern, wird der Index einer Array-Variablen spezifiziert. Besteht der Selektor nur aus einem Zirkumflex, so wird eine dynamische Variable durch ihren Zeiger gefunden. Wenn ein Zeiger auf einen Record verwendet wird[2] und ein Zugriff auf dessen Felder erfolgen soll, wird kein Zirkumflex verwendet. In diesem Fall wird direkt mit dem Punkt auf das Feld zugegriffen. Die Dereferenzierung erfolgt also automatisch.

Ein Typwächter, bei dem der Name eines Datensatztypen in runden Klammern den Selektor bildet, ermöglicht innerhalb eines Ausdrucks den dynamischen Typ einer Datensatzvariablen vorauszusetzen. Dabei führt eine fehlerhafte Typangabe zu einem Abbruch wie bei *ASSERT*. (Siehe Seite 26.) Mit einem Typwächter werden sogenannte Downcasts realisiert. Dies ist allerdings auch mit einer anderen Konstrollstruktur möglich. (Siehe Seite 23.)

---

[1] Ein Selektor kann syntaktisch auch ein Funktionsaufruf sein, dies ist aber nur in einem Ausdruck sinnvoll. Auf der linken Seite einer Zuweisung würde die Änderung einer temporären Variablen verpuffen

[2] In Oberon-07 können Zeiger grundsätzlich nur für Records deklariert werden.

7.2 Ausdrücke und Operatoren

Obwohl Ausdrücke auch bei der Definition von Konstanten vorkommen, sind sie dort nicht uneingeschränkt nutzbar. (Siehe Seite 9.)

In der folgenden Abbildung kann man, ebenso wie in der EBNF schreibweise, nachvollziehen, welche Operatoren stärker binden als andere. Die jeweils engere Schleife hat die höhere Präzidenz.

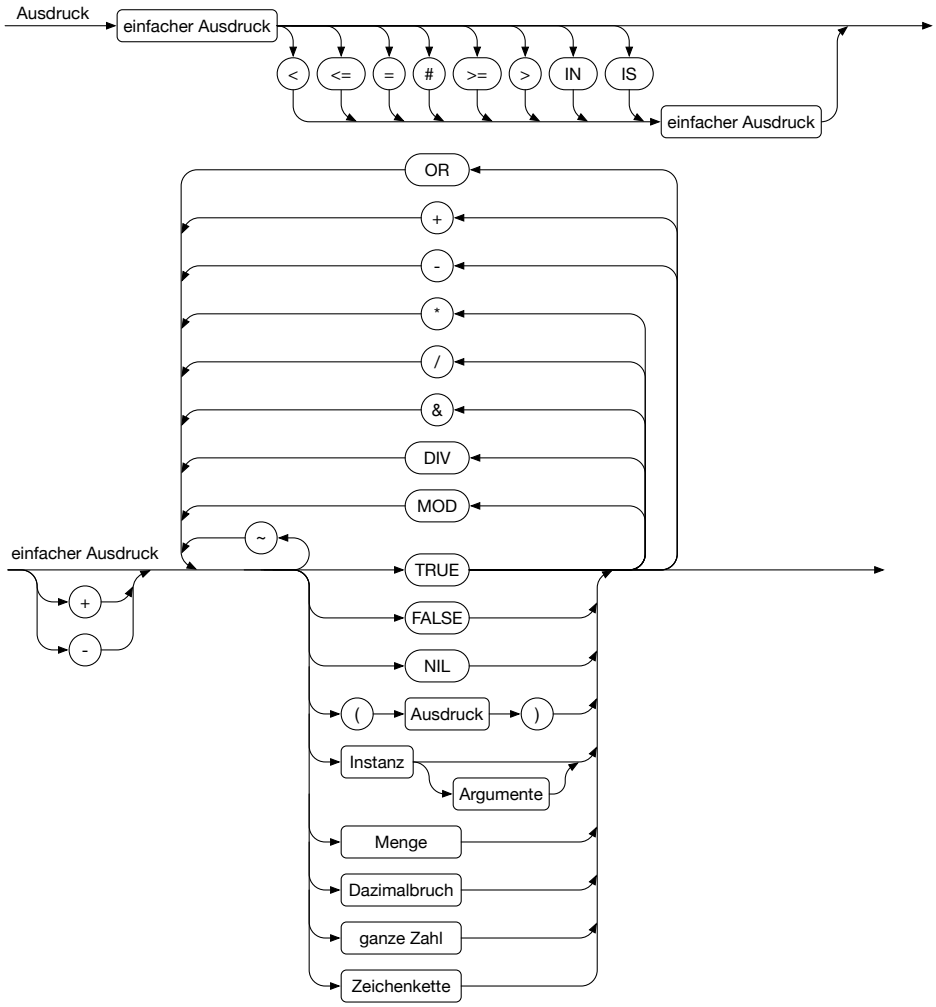

Abbildung 9: Syntaxdiagramm Ausdruck

Für die Definitionen der EBNF gilt hat die Präzidenz der Operator, der mit geringerer Verschachtelungstiefe erreicht wird.

Ausdruck = einfacher Ausdruck, [Vergleichsoperator, einfacher Ausdruck];

einfacher Ausdruck = ['+' | '-'], Term, {('+' | '-' | 'OR'), Term};

Vergleichsoperator = '<' | '<=' | '=' | '#' | '>=' | '>' | 'IN' | 'IS';

Term = Faktor, {('' | '/' | 'DIV' | 'MOD' | '&'), Faktor};*

Faktor = ganze Zahl | Dezimalbruch | Zeichenkette | 'NIL' | 'TRUE' | 'FALSE' | Menge | (Instanz, [Argumente]) | ('(', Ausdruck, ')') | ('~', Faktor);

Enthält ein Ausdruck einen Vergleichsoperator, so wird er immer zu einem booleschen Ergebnis ausgewertet. Die booleschen Operatoren *&* und *OR* werten ihren rechten Operanden nur aus, wenn das Ergebnis nicht bereits durch den linken Operanden eindeutig bestimmt ist (lazy evaluation). Die syntaktische Aufteilung in *Term* und *Faktor* spiegelt die Hierarchie und Auswertungsreihenfolge der Operatoren wider[1].

In der folgenden Tabelle gelten die Angaben für *REAL* und *INTEGER* in klassischem Oberon auch für *LONGREAL* und *LONGINT*. In Oberon-07 ist es nicht zulässig einen Operanden vom Typ *REAL* mit einem Operanden eines anderen Typs in zusammen mit einem dieser Operatoren zu verwenden. In beiden Varianten von Oberon kann der Typ *CHAR* ebenfalls nicht mit anderen Typen für diese Operatoren verwendet werden.

Operator	**Datentypen**	**Bedeutung**
+	*BYTE, INTEGER, REAL*	Summe (oder als Vorzeichen)
+	*SET*	Vereinigungsmenge
-	*BYTE, INTEGER, REAL*	Differenz (oder Vorzeichenwechsel)
-	*SET*	Differenz (z. B. {1,2}-{2,3}={1})
OR	*BOOLEAN*	Disjunktion/Oder (*TRUE* gdw[2]. mindestens einer der beiden Operanden ist *TRUE*)
>	*BYTE, CHAR, INTEGER, REAL*	Größer (*TRUE* gdw. linker Operand ist größer)
>=	*BYTE, CHAR, INTEGER, REAL*	Größer oder gleich (*TRUE* gdw. rechter Operand ist kleiner)
=	alle zueinander kompatiblen Typen	Äquivalenz (*TRUE* gdw. gleich)
<=	*BYTE, CHAR, INTEGER, REAL*	Kleiner oder gleich (*TRUE* gdw. rechter Operand ist größer)
<	*BYTE, CHAR, INTEGER, REAL*	Kleiner (*TRUE* gdw. linker Operand ist kleiner)
#	alle zueinander kompatiblen Typen	Divergenz (*TRUE* gdw. ungleich)
IS	links: *POINTER* oder *RECORD* rechts: Typ	Typprüfung (*TRUE* gdw. der dynamische oder statische Typ der Variablen entspricht dem rechten Operanden oder ist eine Erweiterung dieses Typs)

[1] Es gilt also beispielsweise: Punktrechnung vor Strichrechnung.

[2] gdw. = genau dann wenn

Operator	Datentypen	Bedeutung
IN	links: *INTEGER* rechts: *SET*	Enthalten in (TRUE gdw. linker Operand ist im rechten enthalten)
*	*BYTE, INTEGER, REAL*	Produkt
*	*SET*	Schnittmenge
/	*BYTE, INTEGER, REAL*	Quotient
/	*SET*	Symmetrische Differenz (z. B. {1,2}/{2,3}={1,3})
DIV	*BYTE, INTEGER, REAL*	Ganzzahliger Quotient
MOD	*BYTE, INTEGER, REAL*	Divisionsrest
&	*BOOLEAN*	Konjunktion/Und (*TRUE* gdw. beide Operanden sind *TRUE*)
~	*BOOLEAN*	Negation/Nicht (*TRUE* gdw. Operand ist *FALSE*)

7.3 Unterprogrammaufruf

Eine Prozedur wird immer mit den bei ihrer Deklaration festgelegten Parametern aufgerufen. Die Argumente beim Aufruf sind unter den Bezeichnern der Parameter in der Prozedur verfügbar. Handelt es sich um einen Parameter, der mit *VAR* deklariert wurde, ist es erforderlich, dass das Argument zu einer Variablen ausgewertet werden kann. Alle Argumente werden vollständig ausgewertet, bevor der Aufruf erfolgt.

Nach der Abarbeitung einer Prozedur im Rahmen einer einer Anweisung wird das Programm hinter dem Prozeduraufruf fortgesetzt. Beim Aufruf einer Funktionsprozedur innerhalb eines Ausdrucks wird die Auswertung des Ausdrucks mit dem Rückgabewert der Prozedur fortgesetzt.

Prozeduraufruf = Instanz, [Argumente];

Argumente = '(', [Ausdruck, {',', Ausdruck}], ')';

7.4 Kontrollstrukturen

Die einfachste Anweisung, um in Oberon eine Verzweigung zu programmieren ist die If-Anweisung.

IfAnweisung = 'IF', Ausdruck, 'THEN', Anweisungsfolge, {'ELSIF', Ausdruck, 'THEN', Anweisungsfolge}, ['ELSE' Anweisungsfolge], 'END';

Es wird immer nur die Anweisungsfolge ausgeführt, die zu dem Ausdruck gehört, der als erster als *TRUE* ausgewertet werden kann. Eine Else-Klausel am Ende der Anweisung wird ausgeführt, wenn alle Ausdrücke zu *FALSE* ausgewertet werden. Fehlt die Else-Klausel, so wird das Programm hinter der If-Anweisung normal fortgesetzt.

Bei einer Case-Anweisung wird nur ein einziger Ausdruck verwendet, um zwischen mehreren Fällen zu unterscheiden. Der Wert des Ausdrucks legt den Fall fest, der die passende An-

weisungsfolge enthält. Das Trenner-Symbol zwischen den Fällen ist der senkrechte Strich[1]. Eine Else- oder Otherwise-Klausel kennt Oberon für Case-Anweisungen nicht[2].

CaseAnweisung = 'CASE', Ausdruck, 'OF', Fall, {'|', Fall}, 'END';

Fall = [Bereich, {',', Bereich}, ':', Anweisungsfolge];

Bereich = (ganze Zahl | Zeichenkette | Bezeichnerpfad), ['..', (ganze Zahl | Zeichenkette | Bezeichnerpfad)];

Der Typ des Ausdrucks hinter *CASE* ist eingeschränkt: Es können nur Konstanten des Typs *INTEGER* und *CHAR* verwendet werden. In Oberon-07 wird zusätzlich gefordert, dass in der Fallunterscheidung eine lückenlose Abdeckung eines Wertebereichs von 0 bis zu einem gegebenen Maximum enthalten sein soll. Negative Werte sind folglich nicht zulässig. In klassischem Oberon lässt die Syntax im Bereich konstante Ausdrücke aller Art zu.

Aber der neuere Standard erlaubt zusätzlich das syntaktische Konstrukt für eine Fallunterscheidung bezüglich des dynamischen Typs zu verwenden. In diesem Fall wird der Ausdruck hinter *CASE* (ein Zeiger oder RECORD) mit den Typbezeichnern in den Fällen verglichen. Es sind hierbei nur Typen möglich, die Erweiterungen des Typs der Case-Variablen sind. In der Anweisungsfolge, die mit dem dynamischen Typ übereinstimmt wird die Variable dann als vom erweiterten Typ behandelt und dessen Felder sind somit sichtbar.

Letztere Funktion ist in klassischem Oberon mit der With-Anweisung realisierbar:

WithAnweisung = 'WITH', Bezeichnerpfad, ':', Bezeichnerpfad , 'DO', Anweisungsfolge, 'END';

Auch hier wird in der Anweisungsfolge die Variable links vom Doppelpunkt als vom Typ rechts vom Doppelpunkt behandelt. Sind Variable und Typ nicht passend, so handelt sich das um einen Fehler. Daher wird *WITH* immer erst nach einem Typtest verwendet.

7.4.1 Schleifen

Wiederholungen können in Oberon mit verschiedenen Schleifenkonstrukten realisiert werden. Soll nach dem Ausführen eines Programmteils überprüft werden soll, ob eine Wiederholung erforderlich ist, ist eine Repeat-Anweisung angemessen:

RepeatAnweisung = 'REPEAT', Anweisungsfolge, 'UNTIL', Ausdruck;

Die Anwendungsfolge wird also solange ausgeführt, bis der angegebene Ausdruck zu *TRUE* ausgewertet wird.

Wenn schon zu Beginn einer Schleife überprüft werden kann, ob die Anweisungsfolge ausgeführt werden soll, bietet sich eine While-Schleife an:

WhileAnweisung = 'WHILE', Ausdruck, 'DO', Anweisungsfolge, {'ELSIF', Ausdruck, 'DO', Anweisungsfolge}, 'END';

---

[1] Einige Programmierer lassen den erste Fall leer und setzen im Quelltext am Anfang jeder inneren Zeile einen senkrechten Strich, um eine optische Symmetrie herzustellen.

[2] Der Sprachstandard legt nicht eindeutig fest, wie ein Compiler den Fall behandeln soll, bei dem keine der Bereich-Angaben zutreffend ist. Ein Programmbeispiel in [Wirth 2015] legt aber nahe, dass das Programm zumindest bei einigen Compilern einfach hinter der Case-Anweisung fortgesetzt wird.

Die Verwendung von *ELSIF* innerhalb einer While-Schleife ist eine Neuerung[1] von Oberon-07. Eine While-Schleife wird solange ausgeführt, wie einer der angegeben Ausdrücke zu TRUE ausgewertet wird. Es wird immer nur die Anweisungsfolge ausgeführt, die zu dem Ausdruck gehört, der als erster wahr ist.

7.4.1.1 Endlosschleifen und vorzeitiger Ausstieg

In klassischem Oberon gibt es eine Endlosschleife, die in ihrer Syntaxregel zunächst kein Abbruchkriterium enthält.

Endlosschleife = 'LOOP', Anweisungsfolge, 'END';

Die Anweisung *EXIT* kann in klassischem Oberon als Anweisung verwendet werden, um die Ausführung einer Endlosschleife zu beenden[2]. Außerhalb einer Schleife mit *LOOP* kann diese Anweisung nicht verwendet werden. Bei Oberon-07 wurde diese Schleifenart entfernt, weil die Erfahrung zeigte, dass sie vergleichsweise selten verwendet wurde.

Aber *EXIT* ist nicht einzige Anweisung mit der eine Endlosschleife verlassen werden kann: *RETURN* ist in klassischen Oberon eine Anweisung[3], die eine aufgerufene Prozedur sofort verlässt. Innerhalb von Prozeduren mit Rückgabewert muss hinter jedem *RETURN* ein Ausdruck folgen. In klassischem Oberon ist es möglich, an mehreren Stellen innerhalb einer Prozedur die Return-Anweisung zu verwenden.

7.4.1.2 Abgezählte Wiederholung

Oberon-07 hat die auch in Oberon-2, Modula-2 und Pascal vorhandene For-Anweisung, die im klassischen Oberon zur Vereinfachung der Sprache eliminiert wurde, wieder eingeführt:

ForAnweisung = 'FOR', Bezeichner, ':=', Ausdruck, 'TO', Ausdruck, ['BY', Ausdruck], 'DO', Anweisungsfolge, 'END'};

Die bezeichnete Variable muss vom Typ *INTEGER* sein und wird als *Kontrollvariable* bezeichnet. Das Ändern des Wertes der Kontrollvariablen in der Anweisungsfolge einer For-Schleife ist unzulässig. Die Kontrollvariable wird nach jedem Durchlauf des Schleifenkörpers gemäß dem Ausdruck hinter *BY*[4] erhöht, oder erniedrigt. Die Ausführung entspricht ansonsten einer While-Schleife. Der Schleifenkörper wird so oft ausgeführt, wie der Wert der Kontrollvariablen innerhalb des angegebenen (geschlossenen) Intervalls[5] ist.

Hier ein Beispiel:

```
FOR i := 1 TO 11 BY 2 DO Texts.WriteInt(Out,i,3) END
(* Ergebnis im Text "Out":   1  3  5  7  9 11 *)
```

---

[1] Wirth nennt dieses Detail „Dijkstras Form“ - nach Edsger Wybe Dijkstra - mit Bezug auf dessen Fassung des Euklidischen Algorithmus (Bestimmung des größten gemeinsamen Teilers): *do* $x > y \rightarrow x := x - y \; [] \; y > x \rightarrow y := y - x$ *od*. (vgl. [Dijkstra 1985])

[2] Insofern ist die Bezeichnung evtl. irreführend.

[3] Oberon-07 hat keine *RETURN*-Anweisung. Stattdessen steht das *RETURN* mit einem Ausdruck vor dem *END* der Prozedurdefinition, wenn es sich um eine Prozedur mit Rückgabewert handelt. (Siehe Seite 16.)

[4] Dieser Ausdruck muss als Konstante auswertbar sein.

[5] Ist das Intervall mit einer Variablen angegeben, gilt der Wert der Variablen, der bei der ersten Auswertung vorliegt. Nachträgliche Änderungen des Variablenwerts beeinflussen die Anzahl der Wiederholungen nicht.

8 Im Sprachstandard vorhandene Deklarationen

Die Unterscheidung zwischen Schlüsselwörtern und vordeklarierten Bezeichnern mag an einigen Stellen akademisch erscheinen. *TRUE*, *FALSE* und *NIL* beispielsweise hätten - wie in der Programmiersprache C - theoretisch auch Bezeichner mit den Werten 0 und 1 realisiert werden können[1]. Ein wichtiger Unterschied ist, dass vordeklarierte Bezeichner in einer Deklaration für einen anderen Zweck verwendet werden können. Dies erfolgt nach dem gleichen Prinzip, wie lokale Deklarationen die Bezeichner aus globalen Deklarationen neu belegen können. Für Schlüsselwörter ist das nicht möglich.

8.1 Vordeklarierte Prozeduren

Oberon enthält einige Prozeduren, die immer deklariert sind. Einige davon lassen sich in der Sprache selbst nicht spezifizieren. Trotzdem werden im folgenden Deklarationen zu allen vordeklarierten Prozeduren angegeben[2] und mit Kommentaren fehlende Sprachmittel und Funktionsweisen dargestellt.

8.1.1 Oberon-07

Die vordeklarierten Prozeduren sind in beiden Sprachvarianten sehr ähnlich. Die hier verwendeten Typen und Konstanten dienen nur der Erläuterung und sind keine Deklarationen, die in Oberon Programmen so verwendet werden können.

```
CONST kleinsterInt = 80000000H; (*32 Bit-Computer; negativ; Minimum*)

TYPE
ZahlT = INTEGER (* oder *) REAL;
        (* bzw. auch LONGINT und LONGREAL in klassischem Oberon *)
ArrayT = ARRAY (* Länge *) OF (* beliebiger Typ *);
PointerT = POINTER TO (* irgendein Record *)
OrdinalT = CHAR (* oder *) SET (* oder *) BOOLEAN;

PROCEDURE ABS(x: ZahlT): ZahlT;
BEGIN IF x<0 THEN x:= -x END RETURN x END ABS;

PROCEDURE ASR(k, n: INTEGER): INTEGER;
BEGIN (*Verschiebe jedes Bit in k, das nicht das Vorzeichen ist, um n
Positionen nach rechts. k / 2^n*)
END RETURN k END ASR;

PROCEDURE ASSERT(t: BOOLEAN);
BEGIN IF ~t THEN (* Abbruch des Nutzerkommandos im Oberon-System[3];
Programmende *) END END ASSERT;

PROCEDURE CHR(k: INTEGER): CHAR;
RETURN (* <Num>H → <Num>X *) END CHR;

PROCEDURE FLOOR(x: REAL): INTEGER;
VAR k: INTEGER;
```

---

[1] Die strikte Typisierung von Oberon steht dem aber entgegen.

[2] Wie überall in diesem Dokument wird auf eine gut strukturierte Einrückung verzichtet um die Darstellung kompakter zu machen. Da hier nur einfache Strukturen vorkommen erscheint dies als ein geeigneter Kompromiss.

[3] TRAP 7

BEGIN k:= kleinsterInt; WHILE k<x DO INC(k) END
RETURN k-1 END FLOOR;

PROCEDURE ***FLT(k: INTEGER): REAL;***
RETURN k END FLT;

PROCEDURE ***LEN(v: ArrayT): INTEGER****;*
RETURN (Länge des Array v *) END LEN;*

PROCEDURE ***LSL(k, n: INTEGER): INTEGER;***
*BEGIN (*Verschiebe jedes Bit in x um n Positionen nach links. $k \times 2^n$ *)*
END RETURN k END LSL;

PROCEDURE ***NEW(VAR p: PointerT);*** *BEGIN*
*p:=(*Gültiger Zeiger auf eine neue dynamischen Variablen passenden Typs *)*
END NEW;

PROCEDURE ***ODD(k: INTEGER): BOOLEAN****;*
RETURN k MOD 2 END ODD;

PROCEDURE ***ORD(c: OrdinalT): INTEGER;***
*RETURN (*Zahlenwert von c*) END ORD;*

PROCEDURE ***PACK(VAR x: REAL;n: INTEGER);***
BEGIN x := ($x \times 2^n$ *) END PACK;*

PROCEDURE ***ROR(k, n: INTEGER): INTEGER;***
*BEGIN (*Verschiebe jedes Bit in k um n Positionen nach rechts. Setze die n ehemals am weitesten rechts stehenden Bitwerte auf die links frei gewordenen Positionen.*)*
END RETURN k END ASR;

PROCEDURE ***UNPK(VAR x: REAL; VAR n: INTEGER);***
VAR xNeu: REAL; nNeu: INTEGER;
BEGIN (Bestimme $1.0 \leq xNeu < 20.0$ und nNeu so, dass gilt:*
*$x := xNeu \times 2^{nNeu}$ *) x := xNeu; n:= nNeu END UNPK;*

Es ist nicht garantiert, dass *ASR,LSL* und *ROR* für negative *n* funktionieren.

Zusätzlich gibt es noch die Prozeduren *INC*, *DEC*, *INCL* und *EXCL*, die auch mit dem Pluszeichen bzw. Minuszeichen als Operator und einer Zuweisung erreicht werden können:

INC(k,i) → k:=k+i	*(Fehlt Argument i, dann wird 1 angenommen)*
DEC(k,i) → k:=k-i	*(Fehlt Argument i, dann wird 1 angenommen)*
INCL(m,e) → m:=m+{e}	
EXCL(m,e) → m:=m-{e}	

8.1.2 Klassisches Oberon

Satt *ASR* und *LSL* bietet klassisches Oberon die Funktion ***ASH***, deren Argumente und Rückgabewert vom Typ *LONGINT* sind. Das Vorzeichen von *n* gibt die Schieberichtung an.

Die Funktionsprozedur *LEN* kann zusätzlich mit einem zweiten Parameter aufgerufen werden, der bei mehrdimensionalen Arrays die zu bestimmende Dimension angibt. Der Rückgabewert ist in jedem Fall *LONGINT.*

Für die Umwandlung zwischen verschiedenen Bitbreiten stehen die Funktionsprozeduren ***SHORT*** und ***LONG*** zur Verfügung. *FLOOR* heißt im klassischen Oberon ***ENTIER*** und liefert *LONGINT*.

Die Funktionsprozedur ***SIZE***, die in Oberon-07 in das Modul System verschoben wurde, ist im klassischen Oberon noch eine vordeklarierte Prozedur.

Die Funktionsprozedur ***CAP*** liefert den Großbuchstaben zu einem übergebenen Buchstaben zurück und ***MIN*** und ***MAX*** stellen die jeweiligen Grenzwerte zu dem als Argument übergebenen Datentypen zur Verfügung.

Für *NEW* dürfen auch Zeiger auf andere Typen verwendet werden.

Statt *ASSERT* stellt klassisches Oberon die Prozedur ***HALT*** zur Verfügung, die in Kombination mit einer IF-Anweisung das Gleiche erreicht. *HALT* wird eine Integer-Konstante übergeben, deren Nutzen aber nicht zum Sprachstandard gehört.

Die Prozeduren *PACK*, *UNPK* und *ROR* hat klassisches Oberon nicht. Für *ROR* gibt es im Modul System eine Entsprechung in der Prozedur *ROT*.

Klassisches Oberon hat noch eine Prozedur *COPY*, die sich wie folgt schreiben lässt:

```
PROCEDURE COPY(Q: ARRAY OF CHAR; VAR Z: ARRAY OF CHAR);
VAR k: INTEGER;
BEGIN k:=0;
        WHILE (k<LEN(Q))&(k<LEN(Z)) DO Z[k]:=Q[k]; INC(k) END;
        Z[k-1]:= 0X
END COPY;
```

8.2 Das Modul SYSTEM

Das Modul *SYSTEM* ist eine optionale Komponente der Sprache Oberon. Module, die *SYSTEM* verwenden gelten als nicht portabel und die Deklarationen in *SYSTEM* können je nach Plattform variieren. Schranken, die Fehler vermeiden und Schaden abwenden sollen, werden mit den Prozeduren in *SYSTEM* teilweise aufgehoben. Über die im folgenden beschriebenen Deklarationen hinaus ist bei klassischem Oberon der Typ *BYTE* enthalten, der kompatibel zu *CHAR* und *SHORT* ist.

Name	***Variante***	***Parameter: Typ***	***Rückgabetyp***	***Beschreibung***
ADR	*beide*	*v: beliebige Variable*	*INTEGER (LONGINT)*	*Speicheradresse der Variablen v.*
BIT	beide	*a, n: INTEGER*	*BOOLEAN*	Bitzustand von Bit n an der Speicheradresse a.
CC	klassisch	*n: INTEGER*	*BOOLEAN*	Zustand von Prozessorflag *n.*
COPY	07	*q, z, n: INTEGER*	-	Kopiert *n* Worte[1] aus dem Speicher ab Adresse *q* in den Bereich ab Adresse *z*.

[1] Es bleibt hierbei unklar, welche Wortgröße anzunehmen ist und ob es Überlappungen von Quelle und Zielbereich geben darf.

Name	*Variante*	*Parameter: Typ*	*Rückgabetyp*	*Beschreibung*
GET	beide	*a: INTEGER (LONGINT), VAR v:* einfacher Typ	-	Setzt den Wert von v gemäß dem Inhalt des Speichers ab Adresse *a.*
LSH	klassisch	*v: SET/INTEGER, n: INTEGER*	wie *v*	Bitverschiebung nach links für positive *n*. Verschiebung nach rechts für negative *n*.
MOVE	klassisch	*q, z, n: INTEGER*	-	Kopiert *n* Bytes aus dem Speicher ab Adresse *q* in den Bereich ab Adresse *z*.
NEW	klassisch	*VAR v:* Zeigertyp, *n: INTEGER*	-	Reserviert n Bytes an Speicher und setzt den Zeiger *v* auf die Startadresse.
PUT	beide	*a: INTEGER (LONGINT), v:* einfacher Typ	-	Speichert den Wert von v im Speicher ab Adresse *a.*
ROT	klassisch	*v: SET/INTEGER, n: INTEGER*	wie *v*	Bitrotation nach links für positive *n*. Rotation nach rechts für negative *n*.
SIZE	07	*T:* Typ	*INTEGER*	Anzahl der Bytes, die für die Speicherung einer Variablen vom Typ *T* benötigt wird.
VAL	klassisch	*T:* Typ, *v: beliebige* Variable	*T*	Lässt die Variable *v* innerhalb dieses Ausdrucks so erscheinen, als wäre sie vom Typ *T*.

9 Literatur

[Dijkstra 1985]
Methodik des Programmierens
E. W. Dijkstra, W.H.J. Feijen, 1985
Addison Wesley Verlag (Deutschland) GmbH
ISBN 3-925118-18-7

[Wirth 1990]
The Programming Language Oberon (Revision 1.10.90)
Niklaus Wirth, 1990
Institut für Informatik, Fachgruppe Computer-System
ETH-Zentrum, Zürich

[Wirth 2015]
Programming - A Tutorial (Revision 5.10.2015)
Programming in Oberon
Niklaus Wirth, 2015
ETH Zürich, Department Informatik, Fakultät für Computersysteme
Campus Zentrum, Zürich

[Wirth 2016]
The Programming Language Oberon (Revision 1.10.2013 / 3.5.2016)
Niklaus Wirth, 2016
ETH Zürich, Department Informatik, Fakultät für Computersysteme
Campus Zentrum, Zürich

10 Vollständige Grammatik

Die Syntaktischen Unterschiede zwischen Oberon und Oberon-07 sind überschaubar. Im Folgenden sind die Syntaxregeln alphabetisch nach dem Namen des definierten Symbols sortiert. Für das Symbol *irgendein Zeichen* wird keine Definition angegeben. Für Portabilität ist dafür als Minimum die Menge der druckbaren Zeichen des ASCII-Zeichensatzes empfehlenswert.

10.1 Oberon-07

Anweisung = [Zuweisung | Prozeduraufruf | IfAnweisung | CaseAnweisung | WhileAnweisung | RepeatAnweisung | ForAnweisung];

Anweisungsfolge = Anweisung, {';', Anweisung};

Argumente = '(', [Ausdruck, {',', Ausdruck}], ')';

ArrayTyp = 'ARRAY', Ausdruck, {',', Ausdruck}, 'OF', Typ;

Ausdruck = einfacher Ausdruck, [Vergleichsoperator, einfacher Ausdruck];

Bereich = (ganze Zahl | Zeichenkette | Bezeichnerpfad), ['..', (ganze Zahl | Zeichenkette | Bezeichnerpfad)];

Bezeichner = (Großbuchstabe | Kleinbuchstabe), {Großbuchstabe | Kleinbuchstabe | Ziffer};

Bezeichnerpfad = Bezeichner, ['.', Bezeichner];

CaseAnweisung = 'CASE', Ausdruck, 'OF', Fall, {'|', Fall}, 'END';

Datensatztyp = 'RECORD', ['(', Bezeichnerpfad, ')'], [Variablendeklaration, {';', Variablendeklaration}], 'END';

Deklarationsfolge = ['CONST', {Konstantendefinition, ';'}], ['TYPE', {Typdeklaration, ';'}], ['VAR', {Variablendeklaration, ';'}], {Prozedurdefinition, ';'};

Dezimalbruch = Ziffer, {Ziffer},'.' {Ziffer}, [Skalierung];

einfacher Ausdruck = ['+' | '-'], Term, {('+' | '-' | 'OR'), Term};

Element = Ausdruck, ['..' Ausdruck];

Faktor = ganze Zahl | Dezimalbruch | Zeichenkette | 'NIL' | 'TRUE' | 'FALSE' | Menge | (Instanz, [Argumente]) | ('(', Ausdruck, ')') | ('~', Faktor);

Fall = [Bereich, {',', Bereich}, ':', Anweisungsfolge];

ForAnweisung = 'FOR', Bezeichner, ':=', Ausdruck, 'TO', Ausdruck, ['BY', Ausdruck], 'DO', Anweisungsfolge, 'END'};

ganze Zahl = Ziffer, ({Ziffer}|{hexadezimale Ziffer}, 'H');

Großbuchstabe = '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';

hexadezimale Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9' | 'A' | 'B' | 'C' | 'D' | 'E' | 'F';

IfAnweisung = 'IF', Ausdruck, 'THEN', Anweisungsfolge, {'ELSIF', Ausdruck, 'THEN', Anweisungsfolge}, ['ELSE' Anweisungsfolge], 'END';

Importliste = 'IMPORT', Bezeichner, [':=', Bezeichner], {',', Bezeichner, [':=', Bezeichner]}, ';';

Instanz = Bezeichnerpfad, {Selektor};

Kleinbuchstabe = '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';

Konstantendefinition = Bezeichner, [''], '=', Ausdruck;*

Menge = '{', [Element, {',', Element}], '}';

Modul = 'MODULE', Bezeichner, ';', [Importliste], Deklarationsfolge, ['BEGIN', Anweisungsfolge], 'END', Bezeichner, '.';

Parameterabschnitt = ['VAR'], Bezeichner, {',', Bezeichner}, ':', {'ARRAY OF'}, Bezeichnerpfad;

Parameter = '(', [Parameterabschnitt, {';', Parameterabschnitt}], ')', [':', Bezeichnerpfad];

Prozeduraufruf = Instanz, [Argumente];

Prozedurdefinition = Prozedurkopf, ';', Prozedurrumpf, Bezeichner;

Prozedurkopf = 'PROCEDURE', Bezeichner, [''], [Parameter];*

Prozedurrumpf = Deklarationsfolge, ['BEGIN', Anweisungsfolge], ['RETURN' Ausdruck], 'END';

Prozedurtyp = 'PROCEDURE', [Parameter];

RepeatAnweisung = 'REPEAT', Anweisungsfolge, 'UNTIL', Ausdruck;

Selektor = ('.', Bezeichner) | ('[', Ausdruck, {',', Ausdruck}, ']') | '^' | ('(', Bezeichnerpfad, ')');

Skalierung = 'E', ['+' | '-'], Ziffer, {Ziffer};

Term = Faktor, {('' | '/' | 'DIV' | 'MOD' | '&'), Faktor};*

Typ = Bezeichnerpfad | ArrayTyp | Datensatztyp | Zeigertyp | Prozedurtyp;

Typdeklaration = Bezeichner, [''], '=' Typ;*

Variablendeklaration = Bezeichner, [''], {',', Bezeichner, ['*']}, ':', Typ;*

Vergleichsoperator = '<' | '<=' | '=' | '#' | '>=' | '>' | 'IN' | 'IS';

WhileAnweisung = 'WHILE', Ausdruck, 'DO', Anweisungsfolge, {'ELSIF', Ausdruck, 'DO', Anweisungsfolge}, 'END';

Zeichenkette = '"', {irgendein Zeichen - '"'}, '"' | (Ziffer, {hexadezimale Ziffer}, 'X');

Zeigertyp = 'POINTER', 'TO', Typ;

Ziffer = '0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9';

Zuweisung = Instanz, ':=', Ausdruck;

10.2 Klassisches Oberon

Mit Ausnahme von *Anweisung*, Bereich, *Prozedurrumpf*, *Skalierung* und *WhileAnweisung*, gelten die Syntaxregeln von Oberon-07 auch für klassisches Oberon. Ferner gelten die folgenden Regeln.

Anweisung = [Zuweisung | Prozeduraufruf | IfAnweisung | CaseAnweisung | WhileAnweisung | RepeatAnweisung | Endlosschleife | WithAnweisung | ('RETURN', Ausdruck) | 'EXIT'];

Bereich = Ausdruck, ['..', Ausdruck];

Endlosschleife = 'LOOP', Anweisungsfolge, 'END';

Prozedurrumpf = Deklarationsfolge, ['BEGIN', Anweisungsfolge], 'END';

Skalierung = ('E' | 'D'), ['+' | '-'], Ziffer, {Ziffer};

Vorwärtsdeklaration = 'PROCEDURE', ,'^' , Bezeichner, [''], [Parameter];*

WhileAnweisung = 'WHILE', Ausdruck, 'DO', Anweisungsfolge, 'END';

WithAnweisung = 'WITH', Bezeichnerpfad, ':', Bezeichnerpfad , 'DO', Anweisungsfolge, 'END';