Die Programmiersprache Oberon

Chistoph Doninger
christoph@doninger.com

25. November 2004

Abstract

Oberon ist der Name für ein Betriebssystem und eine Programmiersprache - dieser Artikel beschäftigt sich insbesondere mit der Programmiersprache. Er gibt einen Überblick über die Entwicklung der Sprache aus Modula, liefert Programmierern hinreichend Informationen um einfache Programme in Oberon implementieren zu können sowie generelle Informationen über manche Konzepte und Besonderheiten der Sprache.

Schließlich werden zusammenfassend Aspekte der Sprache aus der Sicht eines Anwenders, der sich eher mit verbreiteteren Sprachen, wie dem Oberon-Vorgänger PASCAL und dem weit verbreiteten C/C++, beschäftigt hat, erläutert.

Motivation zur Entwicklung der Sprache

Oberon ist gleichzeitig der Name eines Betriebssystems (*Oberon System*) und einer Programmiersprache (*Oberon Language*), die beide dem Oberon Projekt entstammen. Ziel dieses Projektes, das 1985 von Niklaus Wirth und Jürg Gutknecht an der Eidgenössischen Technischen Hochschule Zürich initiiert wurde, war es, ein modernes, flexibles Betriebssystem für single-user Workstations zu entwerfen [BCF92].

Niklaus Wirth entwarf und implementierte bereits 1968 die Programmiersprache PASCAL, und später dessen Nachfolger Modula und Modula-2, die trotz ihrer klareren Modularisierung nicht den gleichen Erfolg erzielten wie ihr Vorgänger.

Anfangs war geplant das Oberon System in Modula-2 zu formulieren, da diese Sprache – wie der Name bereits andeutet – modulares Design effizient unterstützte und separat-kompilierbare Programmteile unter der Verwendung von Interfaces zuließ, aber Wirth und Gutknecht fanden einige Punkte, die ihnen wichtig genug waren um über eine Erweiterung von Modula-2 nachzudenken: Modula stellte zwar bereits zufriedenstellende Möglichkeiten im Bereich der Prozedurendeklaration und -Verwendbarkeit zur Verfügung hatte aber im Bereich des Datentypen-handlings einige Nachteile. So erlaubt es zum Beispiel Modula nicht, Datentypen von anderen benutzerdefinierten Typen abzuleiten oder diese zu erweitern. Diese Erweiterung führte allerdings dazu, dass zum Beispiel das Konzept der „variant records“ von Modula-2 für die erweiterte Sprache überflüssig wurde, weshalb die Entwickler beschlossen die Sprache komplett neu zu formulieren und sie die „Oberon langue“ zu nennen.

“The programming language Oberon is the result of a concentrated effort to increase the power of Modula-2 and simultaneously to reduce its complexity. Several features were eliminated, and a few were added in order to increase the expressive power and flexibility of the language.” N. Wirth [Wir88a]

Die Sprache Oberon ist also schlanker und dennoch im Funktionsumfang mächtiger als Modula-2, was zu schlankeren Compilern und präziseren und kürzen Definitionen in Sprachbeschreibungen – zum Beispiel dem *language defining document* [Wir88a] – führt.

So entstanden 1988, quasi als Nebenprodukt, aus dem Designvorgang des Betriebssystems Oberon - selbst zum größten Teil in Oberon geschrieben - die Programmiersprache Oberon und der erste Oberon Compiler.

Nachdem das System sehr positiv aufgenommen wurde, wurde der Oberon Compiler auch auf andere Hardwaresysteme als das von Wirth und Gutknecht verwendete Ceres System portiert und ist heute für viele Hardware-Plattformen erhältlich oder gerade in Entwicklung. Dazu haben einige Firmen bereits kommerzielle Oberon Implementationen entwickelt oder arbeiten gerade daran [BCF92].
Im folgenden wird die Programmiersprache Oberon einfach als Oberon und das Betriebssystem Oberon zur Unterscheidung als Oberon System bezeichnet.

Kurzüberblick über Syntax und Semantik

Hier wird nur ein kurzer Überblick gegeben, die genaue Sprachspezifizierung ist [Wir88a] zu entnehmen.

Die Syntaxbeschreibungen erfolgen hier in einer EBNF(Erweiterte Backus-Naur Form)-ähnlichen Form – zur besseren Übersicht werden Terminalsymbole nicht in Anführungszeichen gesetzt sondern unterstrichen. (Zum Beispiel: der Buchstabe a wird in EBNF als 'a' angeführt, hier einfach als a , das Wort 'THEN' als THEN).

```
letter = a|b|c| ... |z|A|B| ... |Z|.
digit  = 0|1|2|3|4|5|6|7|8|9.
```

Alle gültigen (wohlgeformten) *Sätze* in Oberon – auch „compilation units" genannt – bestehen aus einer endlichen Sequenz von *Symbolen* aus dem *Vokabular* der Sprache.

Symbole werden aus einem endlichen *Alphabet*, dem ASCII-Zeichensatz, gebildet.

Vokabular

Das *Vokabular* von Oberon besteht aus Bezeichnern (*identifiers*), Zahlen (*numbers*), Zeichenketten (*strings*), Operatoren (*operators*), Trennsymbolen (*delimiters*) und Kommentaren (*comments*).

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

Zahl	`number = integer | real.`
	`integer = digit{digit} | digit{hexDigit}H.`
	`real = digit{digit}.{digit} [ScaleFactor].`
	`ScaleFactor = (E|D)[+|-]digit{digit}.`
	`ScaleFactor = (E|D)[+|-]digit{digit}.`				

Zeichenketten	`CharConstant = "character".` `string = "{character}".`				
Operatoren und Trennsymbole	+	:=	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	
Kommentare	`Comment = (*{character}*)`				

Deklarationen

Jeder *identifier* in einem Programm muss vorher durch eine Deklaration (*declaration*) eingeführt werden und referenziert dann für den aktuellen Gültigkeitsbereich (*scope*) das ihm durch die Deklaration zugewiesene Objekt. Der Gültigkeitsbereich des *identifiers* beginnt bei seiner Deklaration und reicht bis zum Ende des nächsten Blocks (dem Ende der Prozedur oder des Moduls), in dem das referenzierte Objekt dann als *lokales Objekt* bezeichnet wird.
Natürlich kann ein *identifier* innerhalb eines Gültigkeitsbereiches nur ein Objekt referenzieren.

Wird ein *identifier* in seiner Deklaration mit dem Exportzeichen * gekennzeichnet, kann er von anderen Modulen, die das deklarierende Modul importieren, verwendet werden, indem sein deklarierendes Modul als Prefix verwendet wird. Deklarierendes Modul und *identifier*, durch einen Punkt getrennt, nennt man dann *qualified identifier*.

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

Folgende Bezeichner sind in Oberon vordefiniert:

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

Konstantendeklarationen

Konstantendeklarationen weisen einem *identifier* bei der Deklaration einen konstanten Wert zu.

```
ConstantDeclaration = identdef ``='' ConstExpression.
ConstExpression = expression.
```

Typendeklarationen

Datentypen (*types*) definieren die Menge an Werten die eine Variable des Datentyps annehmen kann. Eine Typendeklaration assoziiert einen *identifier* mit einem Typ.

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

Basis Typen	`BOOLEAN,CHAR,SHORTINT,INTEGER,LONGINT,REAL,LONGREAL,SET`

Array-Typen	`ArrayType = ARRAY length {, length} OF type.`
	`length = ConstExpression.`
	
	Ein Array besteht aus einer fixen Anzahl (*length*) an Elementen desselben Types und wird mit Indices zwischen 0 und *length*-1 angesprochen: *arrayname*[0] bis *arrayname*[*length*-1]. Mehrdimensionale Arrays werden durch zusätzliche Längenangaben, getrennt durch Beistriche, erzeugt.

Record-Typen	`RecordType = RECORD [( BaseType )] FieldListSequence END.`
	`BaseType = qualident.`
	`FieldListSequence = FieldList {; FieldList}.`
	`FieldList = [IdentList : type].`
	`IdentList = identdef {, identdef}.`
	
	Elemente können als *public* oder *private* für diesen Typ deklariert werden und Records können sich aus anderen Records herleiten (*basetype*). Auf ein Element eines Records wird mittels *recordvariable.fieldname* zugegriffen.

Zeigertypen	`PointerType = POINTER TO type.` Zeigertypen des Typs P haben als Wert einen Zeiger auf Variablen eines bestimmten Types T, welcher bei der Deklaration angegeben werden muss. Die Prozedur `NEW(p)` erzeugt eine Variable des Typs T und weist dem Zeiger p dessen Adresse zu. Zeiger können ausserdem den Wert `NIL` annehmen und zeigen somit auf keine Variable. Über Zeigervariablen kann mittels des Dereferenzators ^ auf den Inhalt der Variable auf die der Zeiger zeigt zugegriffen werden: `p^.fieldname`.
Prozeduren-Typen	`ProcedureType = PROCEDURE [FormalParameters].` Variablen des Prozedurtypen T können Prozeduren oder NIL als Wert annehmen. Wenn eine Prozedur einer Prozedurvariablen zugeweisen wird, müssen die Parameter der Prozedur und des Prozedurtypes der Variablen übereinstimmen.

Variablendeklaration

Variablendeklarationen erzeugen Variablen und assoziieren Bezeichener mit fixen Datentypen.

```
VariableDeclaration = IdentList : type.
```

Ausdrücke

Ausdrücke (e*xpressions*) bestehen aus Operanden (*operands*) und Operatoren (*operators*) und verknüpfen Variablenwerte und Konstanten (=Operanden) durch Operatoren oder Funktionsaufrufe.

Operanden

```
designator = qualident {. ident | [ ExpList ] | ( qualident ) | ^}.
ExpList = expression {, expression}.
```

Operatoren

Logische Operatoren	OR	logische Disjunktion	~Negation	
	&	logische Konjunktion		
Arithmetische Operatoren	+	Summe	/	Quotient
	-	Differenz	DIV	Ganzzahliger Quotiont
	*	Produkt	MOD	Modulus
Mengenoperatoren	+	Vereinigung	/	Symetrische Differenz
	-	Differenz	*	Schnitt
Relationen	=	Gleich	#	Ungleich
	<	Kleiner	<=	Kleiner oder gleich
	>	Größer	>=	Größer oder gleich
	IN	Element von	IS	Test auf Typengleichheit

Anweisungen

Aktionen im Programm erfolgen durch Anweisungen (*Statements*) .

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

Zuweisungen	`assignment = operand := expression.` Der aktuelle Wert einer Variable wird durch Zuweisungen (*assignments*) verändert.
Prozeduraufrufe	`ProcedureCall = designator [ActualParameters].`
Anweisungssequenzen	`StatementSequence = statement {; statement}.` Anweisungen werden durch Semikolons (;) getrennt.

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

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

WHILE Anweisungen	`WhileStatement = WHILE expression DO StatementSequence END.`
REPEAT Anweisungen	`RepeatStatement = REPEAT StatementSequence UNTIL expression.`
LOOP Anweisungen	`LoopStatement = LOOP Statementsequence END.` Loop-Anweisungen terminieren nach einer EXIT Anweisung in der Anweisungssequenz.
RETURN Anweisungen	`ReturnStatement = RETURN {expression}.` Return Anweisungen terminieren eine Prozedur.
EXIT Anweisungen	`EXIT` Anweisungen beenden eine LOOP-Schleife.
WITH Anweisungen	`WithStatement = WITH qualident : qualident DO StatementSequence END.` Allerdings erlaubt diese Anweisung nicht, wie gewohnt, das Argument im folgenden Code Block einfach weglassen zu können, sondern bewirkt lediglich eine temporäre Umbennung eines Bezeichners.

Besonderheiten und Anwendungsbeispiele

Objektorientierte Programmierung in Oberon

Bei näherer Betrachtung bietet Oberon - obwohl es aus der Syntaktischen Spezifikation nicht unmittelbar hervorgeht und Oberon nicht als objektorientierte Sprache gilt – alle Grundlagen die für einen objektorientierten Programmierstil benötigt werden[Mar96]:

- Erweiterbare *abstrakte Datentypen* tauchen bereits in Oberon Standardmodulen auf, und eröffnen vorher ungeahnte Möglichkeiten: Ohne die Module zu ändern sind immer neue – speziell angepasste – Lösungen implementierbar.

- Im Gegensatz zur prozeduralen Programmierung – im einfachsten Fall eine Schritt für Schritt Festlegung des Programmablaufes – wird bei objektorientierter Programmierung davon ausgegangen dass ein Objekt spezifische Methoden, mit dem es auf Anforderungen reagiert, hat. Die Reaktion der Objekte kann, je nach tatsächlicher Implementierung, für dieselbe Anforderung (denselben Aufruf) unterschiedlich sein. Diese Eigenschaft nennt man *Polymorphismus* – eine weitere Grundlage der objektorientierten Programmierung. Zustandsinformationen und Methoden zur Abarbeitung können in dem Objekt *verkapselt* sein.

- Die Möglichkeit der Typenerweiterung (*Type extensions*) schließlich ermöglich die *Vererbung* von *private*- oder *public fields*, später - mit der Extension Oberon-2 - kommt noch die Möglichkeit hinzu, ein *field* als *protected* zu deklarieren[MoW91].

Oberon als Lehrsprache

"C++ is essentially a combination of C - which is nothing much more than a structured assembly language - and a conglomerate of different concepts such as functions, files, types, classes, inheritance, reusability etc.. Hence it is difficult to teach individual concepts with C++. C++ can be compared to different meals, all made with the same sauce." [Gut95]

Im Gegensatz zu anderen objektorientierten Programmiersprachen trennt Oberon verschiedene Konzepte strikt und vereinigt ähnliche oder gleichwertige Konstrukte. Zum Beispiel kann eine FOR-Schleife immer durch eine WHILE-Schleife ersetzt werden, weshalb Oberon keine FOR-Anweisung kennt.

Eine Eignung von Oberon als Lehrsprache lässt sich schon aus dem Erfolg der Vorgänger PASCAL und Modula-2 als Lehrsprachen und dem -in Kapitel Motivation zur Entwicklung der Sprache- erwähntem Motto, das Oberon Modula-2 mächtiger und einfacher machen soll – herleiten.

Vorteile von Oberon als Lehrsprache sind unter anderen:

- Modularisation: Jedes Modul ist statisch aber dennoch erweiterbar. Statische Strukturen erlauben bessere Integritätstest und erhöhen die Übersichtlichkeit und Lesbarkeit des Programmcodes.

- Klassendesign: Klassen in Oberon werden immer einfach durch RECORD-Typen repräsentiert, während zum Beispiel in C++ zwischen Klassen und Typen unterschieden wird – oft zur Verwirrung der Sprachanfänger.

Genaueres zu diesem Thema kann man in [Gut95] nachlesen.

Anwendungen in Oberon

Software für das Oberon System ist natürlich meist selbst in Oberon geschrieben und reicht von dem Betriebsystem selbst über Gerätetreiber, Text- und Grafikeditoren und Internetbrowsern bis zu Spielen und Unterhaltungssoftware. Aber auch für andere Betriebssysteme finden sich Applikationen, die – als Alternative zu C – komplett in Oberon implementiert sind.

Zusammenfassung: Vor- und Nachteile

Übersichtlichkeit, Lesbarkeit, Schlankheit und Geschwindigkeit durch Minimalismus:
"*Make it as simple as possible, but not simpler*" (Albert Einstein) ist ein in Literatur über Oberon oft gelesenes Zitat. N. Wirths Bemühungen Modula-2 noch simpler und effizienter zu Gestalten, haben eine schlanke, gut strukturierte und leicht erlernbare Programmiersprache hervorgebracht.

Diese Form der Vereinfachung bringt allerdings auch Nachteile mit sich [Del89]:

- FOR-Schleifen, ansonsten weit verbreitet, müssen in Oberon durch WHILE-Schleifen ersetzt werden. (ab Oberon-2 sind FOR-Schleifen wieder verfügbar[MoW91])
- Oberon erlaubt keine Arrays oder Records als Rückgabewert von Prozeduren.
- Schon in Modula wurden die komfortablen, von PASCAL her bekannten, Stringoperationen +,=,<,<=,>,>= aus der Sprachspezifikation entfernt.
- Prozedurendeklarationen in RECORDS sind nicht erlaubt und müssen mit Prozedurzeigern umgangen werden.
- ARRAYS können nicht mit variabler Länge deklariert werden.

Ein großer Vorteil gegenüber PASCAL und Modula besteht in der Neuerung der *Type-extensions* und den damit verbundenen neuen Möglichkeiten. Weiters wurde die Modularisierung seit PASCAL konsequent verbessert, das Modulsystem ähnelt nun dem Klassensystem von z.B. C++ oder Java.

Obwohl nicht als solche angesehen, erfüllt Oberon die Anforderungen an eine objektorientierte Programmiersprache, wenn mir auch der Umgang mit diesen Konzepten in anderen Sprachen intuitiv einfacher erscheint : Oberon bietet keine Schlüsselwörter hierfür an, wie z.B. C++ und DELPHI, typgebundene Prozeduren – in C und PASCAL Funktionen genannt – werden erst ab Oberon-2 unterstützt und Prozeduren können nicht in RECORDS deklariert werden. Hinzuzufügen ist noch, dass Oberon-2 speziell zur Vereinfachung der Implementierung objektorientierter Konzepte entwickelt wurde und diese Nachteile teilweise aufhebt.

Einen Vergleich von Oberon-2, C++ und Fortran findet man in [Mos95].

Generell bietet sich die Oberon Sprache sowohl als Lehrsprache als auch für den täglichen Einsatz an, allerdings ist sie verhältnismäßig wenig verbreitet und bietet daher weniger verfügbare Bibliotheken, weniger unterstützende Tools und generell weniger Präsenz im World Wide Web als andere, gängigere Sprachen.

Literaturliste

[BCF92] M. Brandis, R. Crelier, M. Franz, J. Templ: The Oberon System Family.
Software - Practice and Experience, 25:12, 1331-1366, Dec. 1995

[Wir88a] N. Wirth: From Modula to Oberon.
Software - Practice and Experience, 18:7, 661-670, 1988

[Wir88b] N. Wirth: The Programming Language Oberon.
Software - Practice and Experience, 18:7,671-690, 1988

[MoW91] H. Mössenböck and N. Wirth: Differences between Oberon and Oberon-2.
Structured Programming, 12:4, 175-177, 1991

[Mar96] Hannes Marais: Extensible software systems in Oberon.
Journal of Computational and Graphical Statistics, 5(3):284-298, September 1996.

[Del89] A.J.E van Delft: Comments on Oberon.
SIGPLAN Notices, Vol. 24 Iss. 3 p. 23-30, 1989

[Gut95] J. Gutknecht - Oberon in Education.
Modulator Tech Journal, 5:8, Jul. 1995

[Mos95] Bernd Mösli: A Comparison of C++, FORTRAN 90 and Oberon-2 for Scientific Programming
First Joint Conference of GI and SI, GISI'95, Zürich, 1995