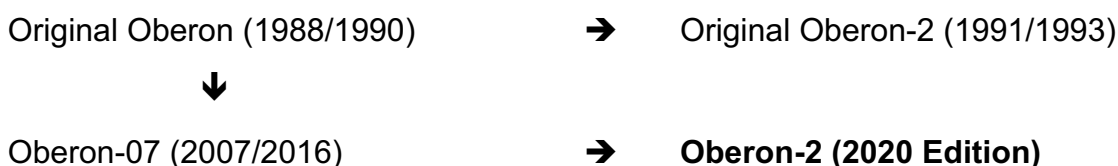# The programming language Oberon-2 (2020 Edition)

Andreas Pirklbauer

1.5.2023

The programming language *Oberon-2 (2020 Edition)* is a revision of the original programming language Oberon-2, as defined in 1991/1993[1]. The main difference to the original is that it is specified as a superset of *Oberon-07 (Revised Oberon)*, as defined in 2007/2016[2], rather than being based on the original language *Oberon,* as defined in 1988/1990[3].

Original Oberon (1988/1990) ➔ Original Oberon-2 (1991/1993)

⬇

Oberon-07 (2007/2016) ➔ **Oberon-2 (2020 Edition)**

This document is *not* a proper language report, as it describes only the *additions* to the *Oberon-07 (Revised Oberon)* programming language, namely: type-bound procedures, a dynamic heap allocation procedure for fixed-length and open arrays, a numeric and a revised type case statement, exporting and importing of string constants, no access to intermediate objects from within nested scopes and module finalization. For the remaining language features, the reader is referred to the official language report of *Oberon-07 (Revised Oberon)*[2]. For some language constructs, this document also provides some rationale for the design choices made.

Type-bound procedures

Globally declared procedures may be associated with a record type declared in the same module. The procedures are said to be *bound* to the record type. The binding is expressed by the type of the *receiver* in the heading of a procedure declaration. The receiver may be either a variable parameter of record type T or a value parameter of type POINTER TO T (where T is a record type). The procedure is bound to the type T and is considered local to it.

    ProcedureHeading = PROCEDURE [Receiver] IdentDef [FormalParameters].
    Receiver = "(" [VAR] ident ":" ident ")".

If a procedure P is bound to a type *T0*, it is implicitly also bound to any type *T1* which is an extension of *T0*. However, a procedure *P'* (with the same name as *P*) may be explicitly bound to *T1* in which case it overrides the binding of *P*. *P'* is considered a *redefinition* of *P* for *T1*. The formal parameters of *P* and *P'* must match. If *P* and *T1* are exported, *P'* must be exported too.

If *v* is a designator and *P* is a type-bound procedure, then *v.P* denotes that procedure *P* which is bound to the dynamic type of *v*. This may be a different procedure than the one bound to the static type of *v*. *v* is passed to *P's* receiver according to the standard parameter passing rules.

[1] *Mössenböck H., Wirth N.: The Programming Language Oberon-2. Structured Programming, 12(4):179-195, 1991*
[2] *http://www.inf.ethz.ch/personal/wirth/Oberon/Oberon07.Report.pdf (Revision 3.5.2016)*
[3] *http://inf.ethz.ch/personal/wirth/Oberon/Oberon.Report.pdf (Revision 1.10.1990)*

If *r* is a receiver parameter declared with type *T*, *r.P^* (pronounced *r.P-referenced*) denotes the (redefined) procedure *P* bound to the base type of *T*.

Example:

```
1    MODULE Trees;
2      IMPORT Out;
3
4      TYPE Tree = POINTER TO Node;
5        Node = RECORD key : INTEGER;
6          left, right: Tree
7        END ;
8
9        CenterTree = POINTER TO CenterNode;
10       CenterNode = RECORD (Node) width: INTEGER;
11         subnode: Tree
12       END ;
13
14       PROCEDURE (T: Tree) Insert (node: Tree);          (*procedure bound to Tree*)
15        VAR p, father: Tree;
16       BEGIN p := T;
17        REPEAT father := p;
18          IF node.key < p.key THEN p := p.left
19          ELSIF node.key > p.key THEN p := p.right
20          ELSE p := NIL
21          END
22        UNTIL p = NIL;
23        IF node.key < father.key THEN father.left := node ELSE father.right := node END ;
24        node.left := NIL; node.right := NIL
25       END Insert;
26
27       PROCEDURE (T: CenterTree) Insert (node: Tree);    (*redefinition of Insert bound to CenterTree*)
28       BEGIN Out.Int(node(CenterTree).width, 3);
29        T.Insert^(node)                                  (*calls the Insert procedure bound to Tree*)
30       END Insert;
31
32     END Trees.
```

Dynamic heap allocation procedure for fixed-length and open arrays

If *p* is a variable of type *P = POINTER TO T*, a call of the predefined procedure *NEW* allocates a variable of type *T* in free storage at run time. The type *T* can be a record *or* array type.

If T is a record type or an array type with *fixed* length, the allocation has to be done with

NEW(p)

If T is an *open* array type, the allocation has to be done with

NEW(p, len)

where *T* is allocated with the length given by the expression *len*, which must be an integer type.

In either case, a pointer to the allocated variable is assigned to *p*. This pointer *p* is of type *P*, while the referenced variable *p^* (pronounced *p-referenced*) is of type *T*.

If *T* is a record type, a field *f* of an allocated record *p^* can be accessed as *p^.f* or as *p.f*. If *T* is an array type, the elements of an allocated array *p^* can be accessed as *p^[0]* to *p^[len-1]* or as *p[0]* to *p[len-1]*, i.e. record and array selectors imply dereferencing.

If *T* is an array type, its element type can be a *record*, *pointer*, *procedure* or a *basic* type (BYTE, BOOLEAN, CHAR, INTEGER, REAL, SET), but not an *array* type (no multi-dimensional arrays).

Example:

```
1    MODULE Test;
2      TYPE R = RECORD x, y: INTEGER END ;
3        A = ARRAY OF R;                (*open array*)
4        B = ARRAY 20 OF INTEGER;       (*fixed-length array*)
5        P = POINTER TO A;              (*pointer to open array*)
6        Q = POINTER TO B;              (*pointer to fixed-length array*)
7
8      VAR p: P; q: Q;
9
10     PROCEDURE New1*;
11     BEGIN NEW(p, 100);  p[53].x := 1
12     END New1;
13
14     PROCEDURE New2*;
15     BEGIN NEW(q);  q[3] := 2
16     END New2;
17
18   END Test.
```

The following rules and restrictions apply[4]:

- Bounds checks on *fixed-length* arrays are performed at *compile* time.
- Bounds checks on *open* arrays are performed at *run* time.
- If *P* is of type *P = POINTER TO T*, the type *T* must be a *named* record or array type[5].

Numeric case statements and type case statements

The revised compiler brings the compiler in line with the official Oberon-07 language report, and now also allows *numeric* case statements[6] in addition to *type* case statements.

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then the statement sequence is executed whose case label list contains the obtained value.

Numeric case statements

If the case expression is of type INTEGER or CHAR, all case labels must be integers or single-character strings, respectively[7].

---

[4] Note that allocating dynamic arrays requires a modified version of the inner core module Kernel, which introduces a new kind of heap block – a r r a y block in addition to r e c o r d block. In some implementations of the Oberon system, an additional kind of heap block describing a storage block of n bytes ("sysblk") exists, which is typically allocated by a special low-level procedure SYSTEM.NEW(p, n). In our implementation, no such special procedure is necessary, as it is covered by a call to NEW(p, n), where p is a pointer to an array of BYTE. Array blocks allocated using NEW(p, len) or NEW(p) are garbage-collected in the same way as regular record blocks. The implementation of garbage collection on fixed-length and open arrays is similar to other implementations of the Oberon system. See, for example, "Oberon Technical Notes: Garbage collection on open arrays", J. Templ, ETH technical report, March 1991.
[5] Restricting pointers to n a m e d arrays is consistent with the official Oberon-07 compiler, which restricts pointers to point to n a m e d records.
[6] http://github.com/andreaspirklbauer/Oberon-numeric-case-statement
[7] In a typical implementation, case labels are restricted to values between 0 and 255.

```
CaseStatement  =  CASE expression OF case {"|" case} [ELSE StatementSequence] END.
case           =  [CaseLabelList ":" StatementSequence].
CaseLabelList  =  LabelRange {"," LabelRange}.
LabelRange     =  label [".." label].
label          =  integer | string | qualident.
```

Example:

```
1   CASE k OF
2     | 0: x := x + y
3     | 1: x := x − y
4     | 2: x := x * y
5     | 3: x := x / y
6   END
```

If the value of the case expression of a *numeric* case statement does not correspond to any case label in the source text, the statement sequence following the symbol ELSE is selected, if there is one, otherwise the program is aborted[8].

Type case statements

The type *T* of the case expression (case variable) may also be a record or pointer type. Then the case variable must be a *simple* identifier that cannot be followed by a selector for structured variables, i.e. it cannot be an array element or a record field. The case labels must be extensions of *T*, and in the statements S_i labelled by T_i, the case variable is considered as of type T_i.

```
CaseStatement  =  CASE qualident OF case {"|" case} [ELSE StatementSequence] END.
case           =  [qualident ":" StatementSequence].
```

Example:

```
1    MODULE M;
2     TYPE R = RECORD a: INTEGER END ;
3       R0 = RECORD (R) b: INTEGER END ;
4       R1 = RECORD (R) b: REAL END ;
5       R2 = RECORD (R) b: SET END ;
6       P  = POINTER TO R;
7       P0 = POINTER TO R0;
8       P1 = POINTER TO R1;
9       P2 = POINTER TO R2;
10
11    VAR p: P; p1: P1;
12
13    PROCEDURE A*;
14     VAR q: P;
15    BEGIN q := p;
16     CASE q OF
17       | P2: q.b := {0, 2}
18       | P1: q.b := 2.5
19       | P0: q.b := 10
20     END
21    END A;
22
23    BEGIN NEW(p1); p := p1
24    END M.
```

---

[8] If one wants to treat such events as "empty" actions, an empty ELSE clause can be used.

If the type of the case variable of a *type* case statement does not correspond to any case label type in the source text, the statement sequence following the symbol ELSE is selected, if there is one, otherwise no action is taken.

Note that the ELSE clause is *not* part of the official Oberon-07 language definition! Here, it has been re-introduced *only* for backward compatibility reasons. In general, we do *not* recommend using the ELSE clause, as it potentially obfuscates the thinking of the programmer. If used, it should be reserved only for exceptional cases, i.e. those that are neither numerous among the possible cases nor do occur frequently at the time of program execution.

But even in cases where the index range far exceeds the label range, one should first try to find a representation using *explicit* case label ranges, as shown in the example below (which assumes an index range of 0..255).

```
CASE i OF                                              CASE i OF
 | 1:  S1                                               | 1:  S1
 | 3:  S3                          is preferred to      | 3:  S3
 | 7:  S7                                               | 7:  S7
 | 9:  S9                                               | 9:  S9
 | 0, 2, 4..6, 8, 10..255:  S0                         ELSE S0
END                                                    END
```

Exporting and importing of string constants

The revised compiler now allows exporting and importing of string constants[9]. Exported *string constants* are treated like pre-initialized, immutable exported *variables*.

Example:

```
 1   MODULE M;
 2     CONST str* = "This is a sample string";      (*export a string constant*)
 3   END M.
 4
 5   MODULE Test;
 6     IMPORT Texts, Oberon, M;
 7
 8     VAR W: Texts.Writer;
 9
10     PROCEDURE P*;
11     BEGIN Texts.WriteString(W, M.str);           (*print the imported string constant*)
12       Texts.Append(Oberon.Log, W.buf)
13     END P;
14
15   BEGIN Texts.OpenWriter(W)
16   END Test.
```

No access to intermediate objects within nested scopes

The revised compiler brings the compiler in line with the official Oberon-07 language report, and now also disallows access to intermediate *constants* and *types* from within nested scopes, not

---

[9] http://github.com/andreaspirklbauer/Oberon-importing-string-constants

just access to intermediate *variables*[10]. With this change, constants, types and variables can be referenced only if they are global or strictly local.

Like the official Oberon-07 compiler, the revised compiler implements *shadowing through scope* when accessing named objects. This means when two objects share the same name, the one declared at the narrower scope hides, or shadows, the one declared at the wider scope. In such a situation, the *shadowed* element is not available in the narrower scope. If the *shadowing* element is itself declared at an intermediate scope, it is only available at *that* scope level, but *not* in narrower scopes (as access to intermediate objects is disallowed).

The official Oberon-07 compiler already issues an error message, if intermediate *variables* are accessed within nested scopes (line 25 of the program below), *regardless* of whether a global variable with the same name exists (line 7) or not. With the revised compiler, the same error message is now *also* issued for intermediate *constants* (line 21) and *types* (lines 16 and 18).

Example:

```
 1   MODULE Test;
 2     CONST C = 10;              (*global constant C, shadowed in Q and therefore not available in R*)
 3
 4     TYPE G = REAL;             (*global type G, not shadowed in Q and therefore available in R*)
 5       T = REAL;                (*global type T, shadowed in Q and therefore not available in R*)
 6     VAR A,                     (*global variable A, not shadowed in Q and therefore available in R*)
 7       B: INTEGER;             (*global variable B, shadowed in Q and therefore not available in R*)
 8
 9     PROCEDURE P;               (*global procedure P*)
10
11      PROCEDURE Q;              (*intermediate procedure Q, contains shadowing elements C, T and B*)
12        CONST C = 20;          (*intermediate constant C which shadows the global constant C*)
13        TYPE T = INTEGER;      (*intermediate type T which shadows the global type T*)
14        VAR B: INTEGER;        (*intermediate variable B which shadows the global variable B*)
15
16        PROCEDURE R(x: T): T;   (*access to intermediate type T allowed in original, not allowed in modified compiler*)
17         VAR i: INTEGER;
18          q: T;                (*access to intermediate type T allowed in original, not allowed in modified compiler*)
19          g: G;                (*access to global type G (not shadowed) allowed in both compilers*)
20        BEGIN (*R*)
21         i := C;               (*access to intermed. constant C allowed in original, not allowed in modified compiler*)
22         P;                    (*access to global (unshadowed) procedure P allowed in both compilers*)
23         Q;                    (*access to intermediate procedure Q allowed in both compilers*)
24         i := A;               (*access to global (unshadowed) variable A allowed in both compilers*)
25         i := B;               (*access to intermediate variable B not allowed in both compilers*)
26         RETURN i
27        END R;
28      END Q;
29     END P;
30
31   END Test.
```

Disallowing access to intermediate objects from within nested scopes while at the same time implementing *shadowing through scope* raises the question whether one should *relax* the shadowing rules somewhat and *allow* access to the *global* scope level, when an object with the same name as a global object is re-declared at an *intermediate* level, but *not* at the strictly local level ("piercing through the shadow").

---

[10] http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects

In the above example, such an approach would allow access to the global variable *B* (line 7) in procedure R (line 25), effectively *ignoring* any intermediate-level variable *B* (line 14) that may also exist. It would make nested procedures "self-contained" in the sense that they can be moved around freely. For example, procedure R can be made local to procedure Q *without* having to be concerned about whether one can still access the global variable *B* (line 7).

We have opted not to adopt this approach for two main reasons. First, a nested procedure may also call the *surrounding* procedure that contains it (a frequent case) and is thus not necessarily self-contained anyway. Second, we didn't want to break with a long language tradition[11].

Module finalization

The statement sequence following the symbol FINAL is executed when the module is removed from the system. A standalone program cannot set a module finalization sequence.

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

Example:

```
 1   MODULE M;
 2     IMPORT Texts, Oberon;
 3
 4     VAR W: Texts.Writer;
 5
 6     PROCEDURE Start*;
 7     BEGIN  (*load module*)
 8     END Start;
 9
10   BEGIN
11     Texts.OpenWriter(W)
12   FINAL
13     Texts.WriteString(W, "Finalizing module M"); Texts.WriteLn(W); Texts.Append(Oberon.Log, W.buf)
14   END M.
```

* * *

---

[11] *In the* appendix *of http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects, a possible implementation of such relaxed shadowing rules is provided.*

Appendix: Implementation cost of the Oberon-2 (2020 Edition) language additions

The total aggregate implementation cost of the *Oberon-2 (2020 Edition)* language additions relative to *Oberon-07 (Revised Oberon)* in source lines of code (sloc) is as follows[12]:

Compiler module	Oberon-07	Oberon-2 (2020 Edition)	Difference	Percent
ORS (scanner)	293	294	1	+ 0.3 %
ORB (base)	394	458	64	+ 16.2 %
ORG (generator)	984	1118	134	+ 13.6 %
ORP (parser)	949	1115	166	+ 17.5 %
Total	**2620**	**2985**	**365**	**+ 13.9 %**

Feature	Source lines of code
Type-bound procedures	200
Dynamic heap allocation procedure for fixed-length and open arrays	40
Numeric case statement	60
All other features combined	65
Total	**365**

* * *

---

[12] *Not counting empty lines and about 100-150 additional lines of source code in modules Kernel, Modules, System to complement the implementation of the Oberon-2 (2020 Edition) language.*