The programming language Oberon-2 (2020 Edition)

Andreas Pirklbauer

15.9.2020

The programming language *Oberon-2 (2020 Edition)* is a revision of the original programming language Oberon-2[1]. The main difference to the original is that it implements a strict 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 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 case statement, exporting and importing of string constants, and no access to intermediate objects within nested scopes. For the remaining language features, the reader is referred to the official language report of *Oberon-07 (Revised Oberon)*[2].

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.

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*.

---

[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)*

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  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  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 statement

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.

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

Numeric case statements

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

---

[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

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
```

Type case statements

The type *T* of the case expression (case variable) may also be a record or pointer type. Then 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.

Example:

```
1  TYPE R   = RECORD a: INTEGER END ;
2     R0 = RECORD (R) b: INTEGER END ;
3     R1 = RECORD (R) b: REAL END ;
4     R2 = RECORD (R) b: SET END ;
5     P   = POINTER TO R;
6     P0 = POINTER TO R0;
7     P1 = POINTER TO R1;
8     P2 = POINTER TO R2;
9
10 VAR p: P;
11
12 CASE p OF
13    | P0: p.b := 10
14    | P1: p.b := 2.5
15    | P2: p.b := {0, 2}
16 END
```

The following rules and restrictions apply:

- Case labels of *numeric* case statements must have values between 0 and 255.
- Case variables of *type* case statements must be *simple* identifiers that cannot be followed by selectors, i.e. they cannot be elements of a structure (array elements or record fields).
- If the value of the case expression 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 no action is taken (in the case of a *type* case statement) or the program is aborted (in the case of a *numeric* case statement)[7].

The ELSE clause has been re-introduced even though it is not part of the Oberon-07 language definition. This was done mainly for backward compatibility reasons. In general, we recommend using the ELSE clause only in well-justified cases, for example if the index range far exceeds the label range. But even in that case, 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
  | 1:  S1
  | 3:  S3
```

```
CASE i OF
  | 1:  S1
  | 3:  S3
```

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

```
  | 7:  S7
  | 9:  S9
ELSE S0
END
```

is the same as

```
  | 7:  S7
  | 9:  S9
  | 0, 2, 4..6, 8, 10..255:  S0        (*preferred*)
END
```

Exporting and importing of string constants

The revised compiler brings the compiler in line with the official Oberon-07 language report, and now allows exporting and importing of string constants[8]. Exported *string constants* are treated like pre-initialized, immutable exported *variables*.

Example:

```
 1  MODULE M;
 2    CONST str* = "This is a sample string";          (*exported string constant*)
 3  END M.
 4
 5  MODULE N;
 6    IMPORT M, Out;
 7
 8    PROCEDURE P*;
 9    BEGIN Out.Str(M.str)                            (*print the imported string constant*)
10    END P;
11  END N.
```

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* within nested scopes, not just access to intermediate *variables*[9].

Like the official Oberon-07 compiler, the revised compiler implements *shadowing through sco*pe 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] *http://github.com/andreaspirklbauer/Oberon-importing-string-constants*
[9] *http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects*

```
 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 interm. 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 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").

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 variables *B* that may also exist (line 14). 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[10].

[10] *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[11]:

Compiler module	Oberon-07	Oberon-2 (2020 Edition)	Difference	Percent
ORS (scanner)	293	293	0	0 %
ORB (base)	394	461	67	+ 17.0 %
ORG (generator)	984	1108	124	+ 12.6 %
ORP (parser)	949	1094	145	+ 15.3 %
Total	**2620**	**2956**	**336**	**+ 12.8 %**

Feature	Source lines of code
Type-bound procedures	210
Dynamic heap allocation procedure for fixed-length and open arrays	40
Numeric case statement	65
All other features combined	21
Total	**336**

* * *

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