Enhanced Oberon-07 Compiler

(without Oberon-2 style type-bound procedures)

Andreas Pirklbauer

11.1.2020

The official Oberon-07 compiler, available at www.projectoberon.com, has been enhanced with various new features, including

- Dynamic heap allocation procedure for fixed-length and open arrays
- Numeric case statement
- Exporting and importing of string constants
- No access to intermediate objects within nested scopes
- Forward references and forward declarations of procedures (experimental feature)
- Module contexts (experimental feature)

In this document, the term “FPGA Oberon” refers to a re-implementation of the original *Project Oberon* on an FPGA development board around 2013, as published at www.projectoberon.com.

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 a: P; b: Q;
9
10   PROCEDURE New1*;
11   BEGIN NEW(a, 100);  a[53].x := 1
12   END New1;
13
14   PROCEDURE New2*;
15   BEGIN NEW(b);  b[3] := 2
16   END New2;
17  END Test.
```

The following rules and restrictions apply[1]:

- 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[2].

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[3] 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.

Example:

[1] *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.*

[2] *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 (but not anonymous) records.*

[3] *http://github.com/andreaspirklbauer/Oberon-numeric-case-statement*

```
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)[4].

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

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

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[5]. 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*[6].

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
 9    PROCEDURE P;               (*global procedure P*)
10
11      PROCEDURE Q;             (*intermediate procedure Q, contains shadowing elements C, T and B*)
```

[5] *http://github.com/andreaspirklbauer/Oberon-importing-string-constants*
[6] *http://github.com/andreaspirklbauer/Oberon-no-access-to-intermediate-objects*

```
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
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 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[7].

Forward references of procedures (experimental feature)

If a procedure Q which is local to another procedure P refers to the enclosing procedure P, as in

```
1   PROCEDURE P;
2
3     PROCEDURE Q;
4     BEGIN (*body of Q*) P          (*forward reference from Q to P, as the body of P is not compiled yet *)
5     END Q;
6
7   BEGIN (*body of P*) …
8   END P;
```

then the official Oberon-07 compiler generates the following code (on FPGA Oberon):

```
20   P'    BL  10          … forward branch to line 31 (across the body of Q to the body of P)
21   Q     body of Q
```

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

```
    ...                    … any calls from Q to P are BACKWARD jumps to line 20 and from there forward to line 31
31  P   body of P
```

whereas the revised compiler generates the following, more streamlined, code:

```
20  Q   body of Q
    ...                    … any calls from Q to P are FORWARD jumps to line 30, fixed up when P is compiled
30  P   body of P
```

i.e. it does not generate an extra forward jump in line 20 across the body of Q to the body of P and backward jumps from Q to line 20. With the official compiler, the extra BL instruction in line 20 is generated, so that Q can call P (as the body of Q is compiled before the body of P).

The motivation for implementing forward *references* of procedures was *not* primarily to improve the performance of nested procedure calls (the performance gain is negligible), but to allow for an easy implementation of forward *declarations* of procedures (see below). As a welcome side effect, generating forward calls for nested procedures actually makes the compiler *shorter*[8].

Forward declarations of procedures (experimental feature)

Forward declarations of procedures have been eliminated in Oberon-07, as they can always be eliminated from any program by an appropriate nesting or by introducing procedure variables[9].

Whether forward declarations of procedures *should* be re-introduced into the language can of course be debated. Here, we have experimented with them for 3 main reasons:

- Legacy programs containing forward declarations of procedures are now accepted again by the compiler, without the need to re-express them using procedure variables.
- Direct procedure calls are (slightly) more efficient than using procedure variables – although the performance gain is negligible and thus provides no strong argument in favor of them.
- Introducing forward *declarations* of procedures adds only about a dozen of lines of source code to the compiler – once the mechanism of forward *references* is in place (see above). However, simplicity of implementation is generally not a good reason for introducing a given feature. Nevertheless, it is comforting to know that it *could* be done with minimal effort.

In the revised compiler, forward declarations of procedures are implemented in the same way as in the original Oberon implementation *before* the Oberon-07 language revision, i.e.,

- They are explicitly specified by ^ following the symbol PROCEDURE in the source text.
- The compiler processes the heading in the normal way, assuming its body to be missing. The newly generated object in the symbol table is marked as a forward declaration.
- When later in the source text the full declaration is encountered, the symbol table is first searched. If the given identifier is found and denotes a procedure, the full declaration is associated with the already existing entry in the symbol table and the parameter lists are compared. Otherwise a multiple definition of the same identifier is present.

The following rules and restrictions apply:

[8] *On the downside, it adds a hidden mechanism in the code generator to construct the fixup list of forward references (see procedures ORG.load, ORG.Call and ORP.ProcedureDecl).*
[9] *See section 2 of www.inf.ethz.ch/personal/wirth/Oberon/PortingOberon.pdf*

- In a forward declaration of a *type-bound* procedure the receiver parameter must be of the same type as in the actual procedure declaration, and the formal parameter lists of both declarations must be identical.
- Both global and local procedures can be declared forward.

Example:

```
 1  MODULE M;
 2    PROCEDURE^ P(x, y: INTEGER; z: REAL);          (*forward declaration of P*)
 3
 4    PROCEDURE Q*;
 5    BEGIN P(1, 2, 3.0)                             (*Q calls P which is declared forward*)
 6    END Q;
 7
 8    PROCEDURE P(x, y: INTEGER; z: REAL);           (*procedure body of P*)
 9    BEGIN …
10    END P;
11  END M.
```

Module contexts (experimental feature)

Module contexts have originally been proposed for the A2 operating system[10]. A module context acts as a single-level name space for modules and allows modules with the same name to co-exist within different contexts.

```
Module   =  MODULE ident [IN ident] ";".
Import   =  IMPORT ident [":=" ident ] [IN ident] ";".
```

In the first line, the optional identifier specifies the name of the context the module belongs to. In the second line, it tells the compiler in which context to look for when importing modules.

Module contexts are implemented as follows:

- Module contexts are specified within the *source* text of a module, as an optional feature. If a context is specified, the name of the source file typically (but not necessarily) contains a prefix indicating its module context, for example *Oberon.Texts.Mod* or *EO.Texts.Mod*.
- If a module context is specified within the source text of a module, the compiler generates the output files *contextname.modulename.smb* and *contextname.modulename.rsc*, i.e. the module context is encoded in the symbol and object file names.
- If no module context is specified within the source text of a module, the compiler generates the output files *modulename.smb* and *modulename.rsc*, i.e. no context is assumed.
- On an FPGA Oberon system, the module context "Oberon" is implicitly specified at run time, i.e. the module loader first looks for *Oberon.modulename.rsc*, then for *modulename.rsc*.
- A module cannot be loaded under more than one context on the same system.
- A module belonging to a context can only import modules belonging to the same context or no context (implementation restriction).

Example:

[10] *http://www.ocp.inf.ethz.ch/wiki/Documentation/Language?action=download&upname=contexts.pdf*

```
1  MODULE Test1 IN Oberon;              (*FPGA Oberon*)
2   IMPORT Out;
3
4   PROCEDURE Go1*;
5   BEGIN Out.Str("Hello from module Test1 in context Oberon (FPGA Oberon)"); Out.Ln
6   END Go1;
7
8  END Test1.
9
10
11 MODULE Test2 IN Oberon;              (*FPGA Oberon*)
12  IMPORT Out, Test1 IN EO;
13
14  PROCEDURE Go1*;
15  BEGIN Out.Str("Calling Test1.Go1.. "); Test1.Go1
16  END Go1;
17
18  PROCEDURE Go2*;
19  BEGIN Out.Str("Hello from module Test2 in context Oberon (FPGA Oberon)"); Out.Ln
20  END Go2;
21
22 END Test2.
```

* * *