O Oberon interface

OBERON WAS THE RESULT OF A CONCENTRATED EFFORT TO INCREASE THE POWER OF MODULA-2 AND SIMULTANEOUSLY TO REDUCE ITS COMPLEXITY.
— NIKLAUS WIRTH
Modula-2 and Oberon (2007)

Oberon is the final member of a family of similar programming languages designed by the late Swiss computer scientist, Niklaus Wirth (15 February 1934–1 January 2024), who was honored with the ACM Turing Award in 1984 for his influential works. Although there are variants Oberon-1, Object Oberon, Oberon-2, Oberon+, Oberon-07, and Component Pascal mentioned in the literature, in this appendix, we do not distinguish further among them.

Fifteen years of experience with Pascal, Modula, and Modula-2 led Niklaus Wirth to simplify those earlier languages, and he co-authored a pair of books, *Programming in Oberon*[1] and *Project Oberon: The Design of an Operating System and Compiler*[2] about his new, and simpler, language. He later updated one of his earlier books as *Algorithms and Data Structures: Oberon version* and made it freely downloadable.[3] In a 1997 interview[4] he stated

> *Keeping a language as simple and as regular as possible has always been a guideline in my work; the description of Pascal took some 50 pages, that of Modula 40, and Oberon's a mere 16 pages.*

[1] Martin Reiser and Niklaus Wirth, ACM Press, 1992, ISBN 0-201-56543-9.
[2] Niklaus Wirth and Jürg Gutknecht, Addison-Wesley, 1992. ISBN 0-201-54428-8.
[3] Published online in 2004, without ISBN, and available at `https://people.inf.ethz.ch/wirth/AD.pdf`.
[4] `http://pascal.hansotten.com/uploads/wirth/interview%20wirth%20June%201997%20carlio%20pescio.txt`

Interfaces to the mathcw library from the predecessor languages are described in Appendix M on page 989 and Appendix P on page 989. They may be helpful in understanding those for Oberon compilers.

This appendix would have appeared in this book at the location suggested by its page numbers, but was written after book publication, when its author undertook a detailed study of Oberon.

O.1 Compilers for Oberon

The literature on Oberon reports that several compilers have been developed for that language, some as commercial products, and others as open-source projects. For the purposes of this text, at least *six* different Oberon compilers have been installed by this author on multiple platforms:

- Karl Landstrom's *Oberon Compiler*, `obnc`.[5]
- The *Oxford Oberon Compiler*, `obc`.[6]
- The *Optimizing Oberon-2 Compiler*, `oo2c`.[7]
- The *Vishap Oberon Compiler*, `voc`.[8]
- The Oleg-n-Cher `ofront+`-based compiler, `o2c`.[9] The distribution does not supply an installer, or a wrapper script to handle translation, compilation, and linking, so this author wrote a private `o2c` script with a similar interface to that of `voc`. `ofront+` is an extension of Josef Templ's earlier `ofront` translator, which is available on the Web, but appears to no longer be buildable, because it requires a particular Oberon compiler to bootstrap it. That compiler is not available on modern systems.
- The Excelsior compilers, `xc` and `xm`.[10] Each compiler handles both Modula-2 and Oberon source code, and we treat them as equivalent in this appendix.

`o2c`, `obnc`, `oo2c`, `voc`, and `xm` compile Oberon code to C, and then silently invoke a second compiler to turn that into native code for the current platform. The resulting executable programs in Oberon should be about as efficient as code in C. For `xm`, with the `+GENCPP` option, translation is to C++.

The `xc` compiler translates Modula-2 and Oberon code to optimized x86 machine instructions in memory, and outputs `*.o` object files, without invoking an assembler. With both `xc` and `xm`, when linking is needed to produce executable programs, that job is handed off to the C compiler.

`obc` compiles Oberon code into platform-independent byte codes for a unique virtual machine that are interpreted at run time by a companion tool, `obxj`. On some systems, the `obc` byte codes can be translated to native machine code.

All of them were designed for cross-platform portability, and each should behave identically with their implementations on other systems. However, for the programmer, they are only superficially equivalent: they have different command line practices, and rather different system modules. Changing compilers generally requires revision of a significant portion of the I/O in user code, and may not be possible at all if the code uses module procedures that are absent in other compilers.

That situation is unlike that for software written in modern C, C++, C#, Fortran, and Java, for which it is possible, with suitable care, to write code that compiles and runs on most platforms, because the language implementations via compilers and libraries generally conform to international or industry standards.

---

[5] See `https://miasap.se/obnc`.
[6] See `https://github.com/Spivoxity/obc-3`.
[7] See `http://prdownloads.sourceforge.net/ooc`.
[8] See `https://github.com/vishaps/voc`.
[9] See `https://github.com/Oleg-N-Cher/OfrontPlus`.
[10] See `https://github.com/excelsior-oss/xds-2.60`.

O.2 Removal of Modula-2 features in Oberon

In designing Oberon, Niklaus Wirth removed several constructs that are available in Modula-2. Here is a partial list of his eliminations:

- concurrency and coroutines;
- enumeration types;
- subrange types;
- octal character constants (such as `11C` for ASCII horizontal tab);
- unsigned integers (the `CARDINAL` type family);
- the `SHORTREAL` floating-point type;
- variants in `RECORD` types (analogues of the C `union` type);
- index ranges in `ARRAY` declarations;
- selective `IMPORT` of functions from modules;
- `FOR` loop statement;
- `WITH` statement for avoiding module prefixes on procedure calls.

In addition, there is major revision and simplification of system modules in Oberon. The `xc` and `xm` compilers offer about 200 modules, `voc` about 140, `oo2c` about 120, `o2c` about 70, `obc` only 17, and `obnc` just 14. By comparison, there are more than 300 with the GNU `gm2` compiler for Modula-2. Of course, the age seniority of that language means that it has had more time to accumulate useful modules.

O.3 Filenames in Oberon

For the `voc` compiler, Oberon source files are, regrettably, named `*.mod` or `*.Mod`, instead of `*.obe`; their syntax is close to, but not identical to, that of Modula-2.

The `o2c` compiler source tree uses files named `*.Mod`, `*.ob2` and `*.ob3`, but the `ofront+` translator is not sensitive to filename extensions. For compatibility with other compilers, our `o2c` wrapper assumes `*.mod` extensions.

The `obc` compiler also accepts files named `*.m`, `*.ob2`, and `*.obn`.

The `obnc` compiler accepts only files named `*.mod`, `*.Mod`, and `*.obn`.

The `oo2c` compiler appears to accept any source file extension.

The `xc` and `xm` compilers use `*.ob2` for Oberon code, and `*.mod` for Modula-2 code, but have command-line options to choose the programming language if other extensions are used.

Output files from the compilers use extensions `.c`, `.d`, `.h`, `.imp`, `.k`, `.lst`, `.o`, `.oh`, `.Doc`, `.Sym`, and `.sym`.

O.4 Using the Oberon compilers

Compilation without linking with these compilers looks like this:

```
% o2c    hello.mod        # output to hello.{c,oh,sym}

% obc -c hello.mod        # output to byte-code file hello.k

% obnc-compile hello.mod  # output to .obnc/hello.{c,h,imp,sym}
```

```
% oo2c   hello.mod        # output to obj/hello.{c,d,oh} and sym/hello.{Doc,Sym}

% voc    hello.mod        # output to hello.{c,h,lst,o,sym}

% xc     hello.ob2        # output to hello.{o,sym}

% xm     hello.ob2        # output to hello.{c,h,o,sym}
```

Compilation and linking usually requires extra options:

```
% o2c     hello.mod -m    # output to hello

% obc     hello.mod       # output to hello.k and default a.out

% obc     hello.mod -o hi # output to hello.k and hi

% obnc    hello.mod       # output to .obnc/hello.{c,o} and hello

% oo2c -M hello.mod       # output to obj/hello.{c,d,o,oh}, obj/hello_.{c,o}, bin/hello,
                          # bin/.libs, and sym/hello.{Doc,Sym}

% voc  -m hello.mod       # output to hello.c and dynamically-linked hello
                          # (actually, the module name)

% voc  -M hello.mod       # output to hello.c and statically-linked hello

% xc   =make hello.ob2    # output to hello.{o,sym} and hello
% xm   =make hello.ob2    # output to hello.{o,sym} and hello
                          # NB: hello.ob2 must start with <* +MAIN *>
```

The `obc` compiler behaves like most UNIX compilers, but the command lines of the others are unusual. Even more peculiar is the `oo2c` practice of writing all of its output files to three subdirectories that are created automatically if they do not yet exist.

The compiler and linker invocation differences need not be memorized by the programmer: instead, they can be recorded in parametrized UNIX `Makefile` rules so that a simple command `make` can build and test the software. This author did exactly that with a collection of 2500 to 3500 lines of about 50 Oberon modules in directories for each compiler. Thus, a command `make TESTNAMES='hello eps32' check` in each compiler directory builds and validates the specified two modules, or all of them when the `TESTNAMES` setting is omitted.

O.5 Modules in Oberon

Oberon removes the separation of interface declarations and function implementations of modules in Modula-2. A file contains only a single module, and is a compilable object. If it has a final `BEGIN` block, it is compilable to an executable program with the same name as the module. Modules that supply functions for use by other code can, for example, use that code block to implement run-time checks that are normally not executed unless selected by an environment variable or command-line option.

There is no selective import of functions from modules: thus, the `IMPORT` statement is followed only by a simple list of module names. Calls to functions imported from modules must *always* be prefixed by the module name. That makes it easy to distinguish them from other functions in the same source file.

O.6 Input/Output in Oberon

As in Modula-2, input/output is not part of the Oberon language, but instead, supplied by library modules. For simple output, a single `IMPORT Out` statement may supply most of the needed functionality.

The `Args` module of `o2c`, `obc`, and `voc`, and the `ProgEnv` module for `xc` and `xm`, give command-line argument and environment variable access. `obnc` has no equivalent facility.

With `oo2c`, command-line access is more complex, because the arguments are mapped to lines of a pseudo-file that must be opened and read. The needed modules are `ProgramArgs` and `TextRider`. A rough outline of the access code looks like this fragment:

```
IMPORT ProgramArgs, TextRider;

VAR k : INTEGER;
    arg : ARRAY 4096 OF CHAR;
    argReader: TextRider.Reader;

BEGIN
    argReader := TextRider.ConnectReader(ProgramArgs.args);

    FOR k = 0 TO ProgramArgs.args.ArgNumber() DO
        argReader.ReadLine(arg);
        ...
    END;
    ...
END themodulename.
```

O.7 Identifiers in Oberon

Identifier names in Oberon begin with an ASCII letter, and are followed by one or more letters or digits. As in most modern programming languages, lettercase in names is *significant*. Unlike Modula-2, underscores are not permitted in such names. However, `obnc` allows underscores, except as the initial character, and `o2c`, `oo2c`, `xc`, and `xm` accept underscores as letters.

There appears to be no serious limit on the lengths of Oberon variable names: five of the test compilers accept 200-character names; `o2c` allows up to 47 characters. However, any underlying C compilers could impose length restrictions. Standards for that language require support for 63 significant initial characters for internal identifiers, and 31 for external ones.

O.8 Characters, logicals, and numbers in Oberon

`CHAR` values represent characters that, for most Oberon compilers, are stored as 8-bit bytes. If Unicode support is required, it must be in the UTF-8 encoding that uses just one byte for characters in the first 128 (the ASCII portion), or two to four bytes for characters with higher Unicode glyph numbers. The UTF-8 encoding guarantees that no zero byte can occur, so a `NUL` character can continue to be a valid string terminator in several programming languages.

There is no provision in Oberon strings for Unicode glyph numbers specified as hexadecimal escape sequences `\u`*hhhh* and `\U`*hhhhhhhh* that were introduced in C99, nor are any of the historical C-style escape sequences recognized.

The compilers `o2c`, `oo2c`, and `voc` all reject source code strings with ASCII control characters, including horizontal tab. Thus, our test suite programs that use tabs for output alignment must employ special handling of that character.

Octal character constants suffixed by the letter `C` in Modula-2 are replaced in Oberon by uppercase hexadecimal constants suffixed with `X`, and prefixed with a zero if the first digit is a letter: `0AX` is an ASCII linefeed character.

`BOOLEAN` values are either `FALSE` or `TRUE`. They cannot be coerced to numerical values by assignment or in arithmetic expressions, but they can be reinterpreted like this: `SYSTEM.VAL(INTEGER, FALSE)` evaluates to 0, and `SYSTEM.VAL(INTEGER, TRUE)` to 1.

The signed integer types are `SHORTINT`, `INTEGER`, `LONGINT`, and for `voc` only, `HUGEINT`. There are no unsigned integer types, except in `xc` and `xm`, which have `CARD8`, `CARD16`, `CARD32`, and `CARD64`. The architect wrote in defense of his design choice to drop unsigned integers:

> *With the prevalence of 32-bit addresses in modern processors, the need for unsigned arithmetic has practically vanished, and therefore the type CARDINAL has been eliminated. With it, the bothersome incompatibilities of operands of types CARDINAL and INTEGER have disappeared.*
>
> Niklaus Wirth
> *From Modula to Oberon* (1990), page 6

In this author's view, that decision is *incorrect*: it is not the *size* of integers that matters, but their *applications*. Address arithmetic, cryptography, hashing, interfacing with hardware devices, multiple precision arithmetic, networking, and operating systems are all application areas where the leading bit of a word has no meaning as a sign, and its treatment as such by the programming language can significantly complicate programming. Computer hardware designs of the past sixty years have almost always supported both signed and unsigned integer arithmetic, because both are needed. Programming languages that fail to supply unsigned integer arithmetic strongly discourage programmers from using those languages in some application areas.

Integer constants are written in decimal, or in uppercase hexadecimal suffixed by `H`, with a leading 0 if the first character would be a letter, However, a hexadecimal value must evaluate to a *positive* integer: the assignment `n := 7FFFFFFFH` to a value of type `INTEGER` is accepted, but `n := 8FFFFFFFH` is rejected with an error `incompatible assignment`. A workaround is to code the latter as `n := 8FFFH; n := 65536 * n + 0FFFFH`. `xc` and `xm` fix that blemish.

Modula-2 recognizes octal numbers suffixed with `B`, but Oberon has no octal support.

Oberon lacks the `SHORTREAL` type, and four of our test compilers map the data type `REAL` to C `float`, and the data type `LONGREAL` to C `double`, so Oberon has only two usable floating-point types. With `obnc`, there is only a `REAL` type, equivalent by default to the 64-bit IEEE 754 format, but buildable to the 80-bit or 128-bit longer formats.

Oberon expects either no exponent, or an `E`-type exponent, on `REAL` constants, and a `D`-type exponent on `LONGREAL` constants. Experiments with numerical constants set like this

```
CONST
     pid = 3.14159265358979323846264338327950382D+00;
     pie = 3.14159265358979323846264338327950382E+00;
     pif = 3.14159265358979323846264338327950382;
```

demonstrate that the last two are converted to 32-bit `REAL` values, whereas the first is converted to a 64-bit `LONGREAL` value. `o2c` and `obnc` do not recognize `D`-style exponents, but `o2c` uses them in output. Numerical programmers therefore need to exercise caution in the coding of such constants.

The numeric types form a hierarchy that allows assignment only to identical types, or to earlier types in this *ordered* list: `LONGREAL`, `REAL`, `LONGINT`, `INTEGER`, and `SHORTINT`. The `voc` compiler's choice of floating-point formats violates that hierarchy, which we can demonstrate with this program to display the sizes of standard types, followed by a test run:

```
% cat tsize.mod
MODULE tsize;

IMPORT Out;

CONST tab = CHR(9);

BEGIN
```

```
    Out.String ('SIZE(BOOLEAN)     ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(BOOLEAN), 0);
    Out.Ln;
    Out.Ln;

    Out.String ('SIZE(CHAR)        ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(CHAR), 0);
    Out.Ln;
    Out.Ln;

    Out.String ('SIZE(SHORTINT)    ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(SHORTINT), 0);
    Out.Ln;

    Out.String ('SIZE(INTEGER)     ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(INTEGER), 0);
    Out.Ln;

    Out.String ('SIZE(LONGINT)     ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(LONGINT), 0);
    Out.Ln;

    Out.String ('SIZE(HUGEINT)     ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(HUGEINT), 0);
    Out.Ln;
    Out.Ln;

    Out.String ('SIZE(REAL)        ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(REAL), 0);
    Out.Ln;

    Out.String ('SIZE(LONGREAL)    ');
    Out.Char(tab);
    Out.String (' = ');
    Out.Int (SIZE(LONGREAL), 0);
    Out.Ln

END tsize.
```

```
% voc tsize.mod -m && ./tsize
SIZE(BOOLEAN)            = 1

SIZE(CHAR)               = 1

SIZE(SHORTINT)           = 1
SIZE(INTEGER)            = 2
SIZE(LONGINT)            = 4
SIZE(HUGEINT)            = 8

SIZE(REAL)               = 4
SIZE(LONGREAL)           = 8
```

The size violation by `voc` is that a `LONGINT` is a 32-bit value, while the significand of a `REAL` is only a 23-bit field. Compilation with `voc -OC` changes the four integer sizes to 2, 4, 8, and 8 bytes.

The other compilers lack `HUGEINT`, and `obc` has integer sizes 2, 4, and 8, while `oo2c`, `xc`, and `xm` have only 1, 2, and 4. `o2c` is unusual: it has a 1-byte `SHORTCHAR` and a 2-byte `CHAR`, three integer types of lengths 2, 4, and 8 bytes, a 4-byte `SHORTREAL`, and an 8-byte `REAL`. Confusingly, `Out.Real()` takes a `SHORTREAL` argument, and `Out.LongReal()` expects a `REAL` argument!

`obnc` is based on a later Oberon language specification that reduces numeric types to just `BYTE` (an unsigned 8-bit integer), `INTEGER`, and `REAL`. A default build of that compiler maps the latter two to C's 32-bit `int` and 64-bit `double` types, but build options can choose any standard supported integer and floating-point types. In addition to the defaults, this author has built the compiler and tools for 64-bit integers, and either 80-bit or 128-bit IEEE 754 floating-point formats, on multiple operating systems for several different CPU families. Only two bugs were found in testing, each easily fixed with a one-line patch. Building on SOLARIS also required changing the shell in script files from `/bin/sh` to `/bin/bash`. One further set of changes was required for that compiler in this author's Oberon test suites: strings can only be delimited by quotation marks. The other compilers accept either apostrophe or quotation mark delimiters.

The `SIZE()` function is one of about 30 built-ins that need no special `IMPORT` statement, except for `obnc`, which needs the prefix and module `SYSTEM`. Two other useful such functions are `MAX()` and `MIN()` that return the largest and smallest finite values in their type arguments. They are not recognized by `obnc`. Others can be found online.[11]

The symbol file produced by `voc` is not a readable text file, but there is a tool that can extract declarations from the symbol file of any local or system module, without requiring knowledge of the Oberon installation location:

```
% showdef Out
DEFINITION Out;

  VAR
    IsConsole-: BOOLEAN;

  PROCEDURE Char(ch: CHAR);
  PROCEDURE Flush;
  PROCEDURE Hex(x: INT64; n: INT64);
  PROCEDURE Int(x: INT64; n: INT64);
  PROCEDURE Ln;
  PROCEDURE LongReal(x: LONGREAL; n: INT16);
  PROCEDURE Open;
  PROCEDURE Real(x: REAL; n: INT16);
  PROCEDURE String(str: ARRAY OF CHAR);
  PROCEDURE Ten(e: INT16): LONGREAL;
```

[11] See `http://www.edm2.com/0608/oberon2.html`, `http://www.ethoberon.ethz.ch/ethoberon/defs`, and `https://github.com/vishaps/voc/blob/master/doc/Features.md`.

```
END Out.
```

Notice that the declarations use the largest supported integer types from the `voc` system types, `INT8`, `INT16`, `INT32`, and `INT64`, to avoid a proliferation of functions for each type.

The corresponding tools for module content listings for the other compilers are `obb` for `obc`, and `oob` for `oo2c`. `o2c`, `obnc`, `xc`, and `xm` lack such a tool.

O.9 Number ranges in Oberon

We can display the number ranges of the standard numeric scalar types in Oberon with a run of this test program:

```
% cat tmaxmin.mod
MODULE tmaxmin;

IMPORT Out;

CONST tab = CHR(9);

BEGIN
    Out.String('MIN(SET)       = ');
    Out.Int(MIN(SET), 0);
    Out.Char(tab); Out.Char(tab); Out.Char(tab);
    Out.String('MAX(SET)       = ');
    Out.Int(MAX(SET), 0);
    Out.Ln;
(*
    Out.String('MIN(BOOLEAN)   = ');
    Out.Int(MIN(BOOLEAN), 0);
    Out.Char(tab); Out.Char(tab); Out.Char(tab);
    Out.String('MAX(BOOLEAN)   = ');
    Out.Int(MAX(BOOLEAN), 0);
    Out.Ln;
*)
    Out.String('ORD(MIN(CHAR)) = ');
    Out.Int(ORD(MIN(CHAR)), 0);
    Out.Char(tab); Out.Char(tab); Out.Char(tab);
    Out.String('ORD(MAX(CHAR)) = ');
    Out.Int(ORD(MAX(CHAR)), 0);
    Out.Ln;

    Out.String('MIN(SHORTINT)  = ');
    Out.Int(MIN(SHORTINT), 0);
    Out.Char(tab); Out.Char(tab); Out.Char(tab);
    Out.String('MAX(SHORTINT)  = ');
    Out.Int(MAX(SHORTINT), 0);
    Out.Ln;

    Out.String('MIN(INTEGER)   = ');
    Out.Int(MIN(INTEGER), 0);
    Out.Char(tab); Out.Char(tab); Out.Char(tab);
    Out.String('MAX(INTEGER)   = ');
    Out.Int(MAX(INTEGER), 0);
```

```
    Out.Ln;

    Out.String('MIN(LONGINT)   = ');
    Out.Int(MIN(LONGINT), 0);
    Out.Char(tab); Out.Char(tab);
    Out.String('MAX(LONGINT)   = ');
    Out.Int(MAX(LONGINT), 0);
    Out.Ln;

    Out.String('MIN(HUGEINT)   = ');
    Out.Int(MIN(HUGEINT), 0);
    Out.Char(tab);
    Out.String('MAX(HUGEINT)   = ');
    Out.Int(MAX(HUGEINT), 0);
    Out.Ln;

    Out.String('MIN(REAL)      = ');
    Out.Real(MIN(REAL), 0);
    Out.Char(tab); Out.Char(tab);
    Out.String('MAX(REAL)      = ');
    Out.Real(MAX(REAL), 0);
    Out.Ln;

    Out.String('MIN(LONGREAL)  = ');
    Out.LongReal(MIN(LONGREAL), 0);
    Out.Char(tab);
    Out.String('MAX(LONGREAL)  = ');
    Out.LongReal(MAX(LONGREAL), 0);
    Out.Ln

END tmaxmin.
```

```
% voc tmaxmin.mod -m && ./tmaxmin
MIN(SET)       = 0                       MAX(SET)       = 31
ORD(MIN(CHAR)) = 0                       ORD(MAX(CHAR)) = 255
MIN(SHORTINT)  = -128                    MAX(SHORTINT)  = 127
MIN(INTEGER)   = -32768                  MAX(INTEGER)   = 32767
MIN(LONGINT)   = -2147483648             MAX(LONGINT)   = 2147483647
MIN(HUGEINT)   = -9223372036854775808    MAX(HUGEINT)   = 9223372036854775807
MIN(REAL)      = -3.40282E+38            MAX(REAL)      = 3.40282E+38
MIN(LONGREAL)  = -1.79769296342094D+308  MAX(LONGREAL)  = 1.79769296342094D+308
```

If the `-OC` compiler option is added to the last command, the range reports for the three standard integer types change to these:

```
MIN(SHORTINT)  = -32768                  MAX(SHORTINT)  = 32767
MIN(INTEGER)   = -2147483648             MAX(INTEGER)   = 2147483647
MIN(LONGINT)   = -9223372036854775808    MAX(LONGINT)   = 9223372036854775807
```

We treat the `SET` data type in Section O.11 on page O-12.

The `o2c`, `oo2c`, `voc`, `xc`, and `xm` compilers reject an attempt to use `BOOLEAN` as an argument of the `MAX()` and `MIN()` functions. The `obc` and `obnc` compilers accept that use, and report the expected limits of 0 and 1.

For the floating-point types, the `MIN()` function returns a negative large magnitude, consistent with its behavior for integer types: it does *not* return the number of smallest magnitude. For `voc`, the constant `Math.small` in module

`Math` holds the smallest normal number of type `REAL`. Alas, the corresponding value for the `LONGREAL` type is zero in module `MathL`: another implementation error.

O.10 Loops in Oberon

The original Oberon specification eliminated the `FOR` loop of the predecessor languages. Instead, it recommended use of a loop with a setup, an initial test, and a final index increment:

```
k := 1;
WHILE k <= n DO
    ... ;
    INC(k)
END;
```

As in Modula-3, the Oberon `DEC()` and `INC()` built-in functions accept an optional second argument that specifies the step: `INC(k, 3)` is equivalent to `k := k + 3`, and `INC(k, -3)` to `DEC(k, 3)`.

However, all of the test compilers recognize the `FOR` loop; it was restored by Niklaus Wirth in Oberon-2. Here is its syntax:

```
FOR k := 1 TO n BY 3 DO
    ...
END;
```

The index variable must be an integer type, and normally should not be altered in the loop body. Five of the compilers permit that, but `xc` and `xm` do not.

The `BY` clause value can be negative, and that clause is normally omitted when the step size is the default of 1. The range test is made *before* the loop body is entered, so if `n < 1`, the body is not executed.

A loop with a test at the end looks like this:

```
k := 3;
REPEAT
  ...
  INC(k, 5)
UNTIL k > 25;
```

`WHILE` loops execute *zero or more times*, and are far more common in computer code than `REPEAT` loops, which always execute *at least once*. The reason is that the common requirement for a loop is *while there is work to be done, do the work*. The other loop corresponds to *calculate until the result is accurate enough*.

More general loops with exits anywhere in the body take this form:

```
LOOP
  ...
  IF condition-1 THEN EXIT END;
  ...
  IF condition-2 THEN EXIT END;
  ...
  IF condition-3 THEN EXIT END;
  ...
END;
```

The `EXIT` statement can *only* be used in a `LOOP` statement: curiously, it is illegal in `FOR`, `REPEAT`, and `WHILE` statements.

Notice that neither Modula-2 nor Oberon has a statement corresponding to the `continue` of the C language, which transfers control to the next iteration of the loop. The `break` and `continue` statements are trivial to implement in a compiler, and long programming experience shows their great utility. It is likely that Niklaus Wirth found

them too much like the distasteful `goto` statement, and banished them in his language designs after Pascal. In their absence, the programmer is forced to introduce conditional code blocks and `BOOLEAN` flags that prevent remaining code from being executed.

O.11 Bit sets in Oberon

The `SET` data type occupies a single default integer word, and its elements are not set with bit masks, but rather with braced lists of bit numbers. Here is a test program that shows set assignment, intersection, union, difference, symmetric difference, and membership testing:

```
MODULE tset2;

IMPORT Out;

CONST tab = CHR(9);

VAR k : INTEGER;
    ones, w, zeros : SET;

PROCEDURE showset(w : SET);
VAR k : INTEGER;
BEGIN
    k := MAX(SET);

    WHILE k >= MIN(SET) DO
        IF (k < MAX(SET)) & ((k MOD 4) = 3) THEN
            Out.Char('_')
        END;

        IF k IN w THEN                  (* set membership test *)
            Out.Char('1')
        ELSE
            Out.Char('0')
        END;

        DEC(k)
    END
END showset;

BEGIN
    ones := {MIN(SET) .. MAX(SET)};
    zeros := { };

    w := {1, 3, 5, 7, 28, 29, 30, 31};  (* set assignment *)
    w := w + {10, 11, 12};              (* set union      *)
    w := w - {1, 2, 3};                 (* set difference *)

    Out.String('w = {1, 3, 5, 7, 28, 29, 30, 31} + {10, 11, 12} - {1, 2, 3}');
    Out.Ln;
    Out.Ln;

    k := MIN(SET);
```

```
    WHILE k <= MAX(SET) DO

        IF (k MOD 4) # 0 THEN
            Out.Char(tab)
        END;

        Out.String('Bit ');
        Out.Int(k, 2);
        Out.String(' = ');

        IF k IN w THEN                  (* set membership test *)
            Out.Char('1')
        ELSE
            Out.Char('0')
        END;

        IF (k MOD 4) = 3 THEN
            Out.Ln
        END;

        INC(k)
    END;

    Out.Ln;
    Out.String('zeros         = ');
    showset(zeros);
    Out.Ln;

    Out.String('ones          = ');
    showset(ones);
    Out.Ln;

    Out.String('w             = ');
    showset(w);
    Out.Ln;

    Out.String('{1, 5, 9}     = ');
    showset({1, 5, 9});
    Out.Ln;

    Out.Ln;
    Out.String('w * {1, 5, 9} = ');
    showset(w * {1, 5, 9});             (* set intersection *)
    Out.Ln;

    Out.String('w + {1, 5, 9} = ');
    showset(w + {1, 5, 9});             (* set union *)
    Out.Ln;

    Out.String('w - {1, 5, 9} = ');
    showset(w - {1, 5, 9});             (* set difference *)
```

```
    Out.Ln;

    Out.String('w / {1, 5, 9} = ');
    showset(w / {1, 5, 9});             (* symmetric set difference *)
    Out.Ln;

    Out.Ln;
    Out.String('w * {}        = ');
    showset(w * zeros);                 (* set intersection *)
    Out.Ln;

    Out.String('w + {}        = ');
    showset(w + zeros);                 (* set union *)
    Out.Ln;

    Out.String('w - {}        = ');
    showset(w - zeros);                 (* set difference *)
    Out.Ln;

    Out.String('w / {}        = ');
    showset(w / zeros);                 (* symmetric set difference *)
    Out.Ln;

    Out.Ln;
    Out.String('w * ones      = ');
    showset(w * ones);                  (* set intersection *)
    Out.Ln;

    Out.String('w + ones      = ');
    showset(w + ones);                  (* set union *)
    Out.Ln;

    Out.String('w - ones      = ');
    showset(w - ones);                  (* set difference *)
    Out.Ln;

    Out.String('w / ones      = ');
    showset(w / ones);                  (* symmetric set difference *)
    Out.Ln

END tset2.

% voc tset.mod -m && ./tset
w = {1, 3, 5, 7, 28, 29, 30, 31} + {10, 11, 12} - {1, 2, 3}

Bit  0 = 0      Bit  1 = 0      Bit  2 = 0      Bit  3 = 0
Bit  4 = 0      Bit  5 = 1      Bit  6 = 0      Bit  7 = 1
Bit  8 = 0      Bit  9 = 0      Bit 10 = 1      Bit 11 = 1
Bit 12 = 1      Bit 13 = 0      Bit 14 = 0      Bit 15 = 0
Bit 16 = 0      Bit 17 = 0      Bit 18 = 0      Bit 19 = 0
Bit 20 = 0      Bit 21 = 0      Bit 22 = 0      Bit 23 = 0
Bit 24 = 0      Bit 25 = 0      Bit 26 = 0      Bit 27 = 0
```

```
Bit 28 = 1      Bit 29 = 1      Bit 30 = 1      Bit 31 = 1

zeros         = 0000_0000_0000_0000_0000_0000_0000_0000
ones          = 1111_1111_1111_1111_1111_1111_1111_1111
w             = 1111_0000_0000_0000_0001_1100_1010_0000
{1, 5, 9}     = 0000_0000_0000_0000_0000_0010_0010_0010

w * {1, 5, 9} = 0000_0000_0000_0000_0000_0000_0010_0000
w + {1, 5, 9} = 1111_0000_0000_0000_0001_1110_1010_0010
w - {1, 5, 9} = 1111_0000_0000_0000_0001_1100_1000_0000
w / {1, 5, 9} = 1111_0000_0000_0000_0001_1110_1000_0010

w * {}        = 0000_0000_0000_0000_0000_0000_0000_0000
w + {}        = 1111_0000_0000_0000_0001_1100_1010_0000
w - {}        = 1111_0000_0000_0000_0001_1100_1010_0000
w / {}        = 1111_0000_0000_0000_0001_1100_1010_0000

w * ones      = 1111_0000_0000_0000_0001_1100_1010_0000
w + ones      = 1111_1111_1111_1111_1111_1111_1111_1111
w - ones      = 0000_0000_0000_0000_0000_0000_0000_0000
w / ones      = 0000_1111_1111_1111_1110_0011_0101_1111
```

Close examination of the bit patterns shows these correspondences with bit operations:

- *Assignment* with member ranges sets the member bits to 1.
- *Assignment* of an empty set zeros all bits.
- *Assignment* of a full set makes all bits 1.
- *Intersection* with * is a bitwise *AND*.
- *Union* with + is a bitwise *OR*.
- *Difference* with - is a bitwise *AND NOT*.
- *Symmetric difference* with / is a bitwise *XOR*.
- *Symmetric difference* with a full set using / is a bitwise *NOT*, or one's complement.

To complement a set in Oberon, prefix it with a unary minus.

Viewing a set as a 32-bit integer shows that set membership numbering increases from right to left, corresponding to *little-endian* bit numbering. It is not clear from Oberon documentation whether this would change on a platform with *big-endian* addressing and bit numbering. At least for `obnc`, bit numbering remains little endian on both big- and little-endian byte addressing CPU architectures.

The `voc` compiler translates Oberon code to C, rather than to native assembly code, and examination of the translation shows efficient handling of the set operations. The first five assignments in our test program are turned into this C fragment:

```
tset2_ones = 0xffffffff;
tset2_zeros = 0x0;
tset2_w = 0xf00000aa;
tset2_w = tset2_w | 0x1c00;
tset2_w = (tset2_w & ~0x0e);
```

Each of those translates into just one or two CPU instructions, depending on compiler optimization levels.

Further probing into the contents of all of the Oberon modules supplied with the `voc` compiler reveals that module `SYSTEM` defines a `SET64` type, so a module name change, extending the `IMPORT` statement, and replacing `SET` by `SYSTEM.SET64` produces a new test program that works without further changes, but not correctly.

Unfortunately, the test run revealed a compiler bug: the initial assignment of the variable `ones` sets only the rightmost 32 bits! Using the hard-coded value `{ 0 .. 63 }` behaves the same way. This new code solves the initialization problem:

```
FOR k := MIN(SYSTEM.SET64) TO MAX(SYSTEM.SET64) DO
    ones := ones + { k }
END;
```

The later assignments were then augmented to set a few bits in the range `32 .. 63`, and all of the output is now correct.

Use of the `SYSTEM` module is generally discouraged in most Modula-2 and Oberon programs, because it introduces system-dependent code. However, that also makes such code likely to be less-well tested, and to harbor bugs that might not be uncovered for a long time, as our discovery shows! Examination of comments in the `voc` compiler source code shows that it was last worked on in the years 1992–1996.

O.12 Arrays in Oberon

Pascal, Modula-2, and Oberon use slightly different syntax for declarations of arrays with more than one dimension:

```
VAR a : array [1 .. MX,   1 .. MY] of Real;         (* Pascal *)
VAR a : ARRAY [1 .. MX], [1 .. MY] OF REAL;         (* Modula-2 *)
VAR a : ARRAY [1 .. MX] OF ARRAY [1 .. MY] OF REAL; (* variant Modula-2 *)
VAR a : ARRAY MX, MY OF REAL;                       (* Oberon *)
VAR a : ARRAY MX OF ARRAY MY OF REAL;               (* some Oberon compilers *)
```

In Oberon, array indexes always start at zero and increase up to one less than the declared dimension. The built-in function `LEN(a, n)` returns the length of dimension `n`, counting dimensions from 0, and `LEN(a)` is equivalent to `LEN(a, 0)`. The index ranges are 0 to `LEN(a, n) - 1`. Here is a function that carries out the matrix operation $\boldsymbol{C} = \boldsymbol{A} \times \boldsymbol{B}$, for matrices of arbitrary, but conformant, dimensions:

```
PROCEDURE matmul(VAR c, a, b : ARRAY OF ARRAY OF REAL);
VAR i, j, k : INTEGER;
    sum : REAL;
BEGIN
    FOR i := 0 TO LEN(a, 0) - 1 DO
        FOR j := 0 TO LEN(b, 1) - 1 DO
            sum := 0.0;
            FOR k := 0 TO LEN(a, 1) - 1 DO
                sum := sum + a[i][k] * b[k][j]
            END;
            c[i][j] := sum
        END
    END
END matmul;
```

O.13 A numeric example in Oberon

Here is the computation of the machine epsilon in Oberon:

```
% cat eps32.mod
MODULE eps32;

IMPORT Out;

VAR eps, temp: REAL;
    k: INTEGER;

BEGIN
    Out.String ('SIZE(REAL) = ');
    Out.Int (SIZE(REAL), 0);
    Out.Ln;
    Out.Ln;

    eps := 1;
    k := 0;
    temp := 1 + eps * 0.5;

    WHILE temp > 1 DO
        DEC(k);
        eps := 0.5 * eps;
        temp := 1 + eps * 0.5
    END;

    Out.String ('eps32 = ');
    Out.Real(eps, 9);
    Out.String (' = 2**(');
    Out.Int (k, 0);
    Out.String (')');
    Out.Ln
END eps32.

% voc eps32.mod -m && ./eps32
SIZE(REAL) = 4

eps32 = 1.19209E-07 = 2**(-23)
```

The subterfuge of the temporary variable, `temp`, is needed here, because otherwise, the loop condition is evaluated in type `LONGREAL`. We simplified `1.0` to `1` to illustrate the numeric type promotion in Oberon.

Alternatively, instead of a local variable, we can introduce a function, preferably defined elsewhere, that forces storage of the test expression in memory, and then returns the stored value:

```
PROCEDURE store(VAR result : REAL; value : REAL) : REAL;
BEGIN
    result := value;
    RETURN result
END store;
...
    WHILE (store(temp, 1 + 0.5 * eps) > 1)
    DO
        DEC(k);
        eps := 0.5 * eps
    END;
```

O.14 The Oakwood guidelines

Years of experience with Modula-2 using different compiler implementations on multiple platforms showed that there was considerable divergence in module support, because external modules were not sufficiently specified in the language definitions. That situation was later improved with the language standardization in ISO/IEC 10514-1:1996, 10514-2:1998, and 10514-3:1998. However, the complexity, and long development time, of the ISO standards for Modula-2 discouraged Niklaus Wirth and other developers from pursuing ISO standardization for Oberon.

A similar divergence in module support happened in the early years of Oberon implementations, so a conference of leading developers was held at the Oakwood Hotel in the London suburb of Croydon, UK, on 21–23 June 1993 to reach an agreement on a common baseline for language extensions and module support. The outcome of that meeting is an informal document called the *Oakwood Guidelines for Oberon-2 Compiler Developers*.[12]

Apart from module requirements, here are some notable things that the Oakwood guidelines recommend about the Oberon language itself:

- Strings are *always* `NUL` terminated.
- A string of length 1 can be used in any context where a character constant is allowed, and vice versa. Thus, `61X` and `"a"` are both characters and 1-character strings that represent the first letter of many alphabets.
- The type inclusion hierarchy may infer an implicit truncation of precision between `REAL` and `LONGINT`. For example, if both types are represented in 32 bits, then the `REAL` significand precision is likely to be only 24 bits. An assignment from a `LONGINT` to a `REAL` will therefore involve a truncation of precision of the value assigned. [Acceptance of this limitation seems unfortunate: it provides a silent hole for digit loss, unless compilers implement a range test in the assignment of integers to floating-point values.]
- Identifiers may contain the additional character, underscore, with the lexical grammar rule

  ```
  ident = (letter | "_") { letter | digit | "_" }.
  ```

  The syntax extension allows for identifiers to begin with the underscore character. That character is widely used in identifier names in many other programming languages, and supporting it is *essential* for interlanguage calls.
- The exponentiation operator ** provides a convenient notation for arithmetic expressions, is part of Fortran since its introduction in 1956, and has been adopted by several other languages. The operator has higher precedence that any other binary operator, so `w + x**y - z` is equivalent to `w + Math.power(x,y) - z`.
- Identifiers of at least 23 significant characters are possible.
- An exported comment is denoted using two consecutive asterisks after the opening parenthesis, for example, `(** this is an exported comment *)`. It signals to a browser that the comment should be included in a `DEFINITION` module being derived from the module being processed. It is a *convention*, rather than a language issue.

While the *Oakwood Guidelines* are frequently cited in the Oberon literature, experience with the compilers treated in this appendix demonstrates that the Oakwood recommendations have not been widely adopted, although `xc` and `xm` support most of the Oakwood extensions.

O.15 Programming the Oberon MathCW interface

Interfacing to C code requires different syntax with the six Oberon compilers, so we treat them separately in the following subsections.

[12] See `http://www.math.bas.bg/bantchev/place/oberon/oakwood-guidelines.pdf`. That file is easily found at many other Web sites.

O.15.1 The obc MathCW interface

With obc, the interface to C is delightfully simple:

```
MODULE MathCW;
...
PROCEDURE erff*    (x : REAL)          : REAL     IS "erff";    (* C float *)
PROCEDURE erf*     (x : LONGREAL)      : LONGREAL IS "erf";     (* C double *)
...
PROCEDURE isqnanf* (x : REAL)          : INTEGER  IS "isqnanf"; (* C int *)
PROCEDURE isqnan*  (x : LONGREAL)      : INTEGER  IS "isqnan";  (* C int *)
...
PROCEDURE qnanf*   (s : ARRAY OF CHAR) : REAL     IS "qnanf";   (* C float *)
PROCEDURE qnan*    (s : ARRAY OF CHAR) : LONGREAL IS "qnan";    (* C double *)
...
END MathCW.
```

For this compiler, the asterisk on procedure names means that they are visible outside the module. The only addition from a normal prototype is the `IS "name"` suffix. For occasional use in your code, there is no need for a separate interface module: prototypes like those shown can be used directly, and the asterisk suffix is then not needed.

Some Modula-2 and Oberon compilers pass composite arguments, like `ARRAY` and `RECORD` variables, as a pair of arguments: an unsigned integer length in elements or bytes, and a pointer to the first element. That requires introducing an extra interface function in C to capture, and discard, the length argument. `obc`, however, does not pass the length to an external function, so no intermediary function is needed. The Oberon programmer just has to ensure that character string arguments are `NUL`-terminated, as they guaranteed to be if they are constant strings.

As with the interfaces to other programming languages treated in this book, we retain the `isxxx()` property-test functions as *integer* functions, rather than converting them to *Boolean* return values when the master language supports such a type. For the comfort of Oberon programmers, the module `MathCW.mod` has therefore been extended with additional public functions like these:

```
PROCEDURE IsQNaNf* (x : REAL)        : BOOLEAN; BEGIN RETURN isqnanf(x) # 0 END IsQNaNf;
PROCEDURE IsQNaN*  (x : LONGREAL)    : BOOLEAN; BEGIN RETURN isqnan(x)  # 0 END IsQNaN;

PROCEDURE IsSNaNf* (x : REAL)        : BOOLEAN; BEGIN RETURN issnanf(x) # 0 END IsSNaNf;
PROCEDURE IsSNaN*  (x : LONGREAL)    : BOOLEAN; BEGIN RETURN issnan(x)  # 0 END IsSNaN;
```

To use the interface, its byte-code file, or source file, must be supplied to the compiler *ahead of* user modules. In addition, if the referenced link library is not installed in standard system directories, then the path to its location must also be supplied. Here is a typical example:

```
% obc -c MathCW.mod                                              # precompile interface

% obc -C MathCW.k   myprog.mod -L$prefix/lib64 -lmcw -o myprog  # compile and link user code

% obc -C MathCW.mod myprog.mod -L$prefix/lib64 -lmcw -o myprog  # compile both and link user code

% ./myprog                                                      # run user program
```

The `-C` option is essential for cross-language linking.

It appears that `obc` has compiled-in knowledge of standard system modules, because copying the `MathCW.k` file into the library directory, and omitting it from the second command line, is not enough for it to be found.

There is one additional step that may need to be taken. On most of this author's systems, the `mathcw` library is built with `dgcc`, a version of `gcc` that has been patched to support decimal floating-point arithmetic. The `obc` command is a shell script that contains a hard-coded choice of the C compiler with this line:

```
CC="gcc -std=gnu99"
```

However, that compiler does not recognize the decimal extensions: we need to use `dgcc` instead. There are three reasonable solutions:

- Change the file `/usr/bin/obc` to replace `gcc` by `dgcc`.
- Change the file `/usr/bin/obc` so that the C compiler is set by `CC=${CC-"gcc -std=gnu99"}`, allowing the user to override the default compiler via the `CC` environment variable, and to supply alternate compiler flags.
- Leave `/usr/bin/obc` intact, and create a symbolic link from the pathname of `dgcc` to a program `gcc` in a `PATH` directory that precedes `/usr/bin`.

The first two of those require administrator access, and have to be repeated if the obc package is later updated.

The third choice requires no extra privileges, but has to be done by each user of the interface.

The best choice is the second, because it gives the Oberon user flexibility, and does not change the C compiler for other programs, or for other users.

The number-to-string functions in the mathcw library return a pointer to a static internal buffer. Oberon does not support functions that return `ARRAY` or `RECORD` results, so to make those functions accessible, we need a wrapper function in C that looks like this:

```
% cat wrapntos.c
#include <mathcw.h>
#include <string.h>

void
wrapntos(char s[], double x)
{
    (void)strcpy(s, ntos(x));
}
```

The Oberon program has this code fragment that prototypes the wrapper, then uses it to convert a number to a string:

```
MODULE myprog;
...
PROCEDURE wrapntos (s : ARRAY OF CHAR; x : LONGREAL) IS "wrapntos";
...
VAR s : ARRAY 25 OF CHAR;
    x : LONGREAL;
...
x := 1.125;
wrapntosf(s, x);
Out.String('s = "');
Out.String(s);
Out.String('"');
Out.Ln;
...
END myprog.
```

Finally, these steps compile, link, and run the program:

```
% cc -c -I$prefix/include wrapntos.c

% obc -C myprog.mod wrapntos.o -L$prefix/lib64 -lmcw && ./a.out
s = "                   +1.125 "
```

Having a single module provide an interface to the mathcw library has a side effect that we have not mentioned: the interface contains references to a large number of library functions. Historically, UNIX linkers treat the object file compiled from a single source file as an unbreakable unit. That means that all of the functions referenced in the interface must be linked into the executable program, even if they are not called by the user program.

With shared libraries, that may not matter, because only one copy of the library is in memory, and is available to multiple processes. With statically linked libraries, a copy of *every* library function named in the interface is present in the executable program.

When memory use must be minimized, Oberon users who need only a few functions from a C library are advised to create private subsets of the `MathCW.mod` and `MathCW.c` interface files with prototypes and functions for just the user-referenced ones.

O.15.2 The obnc MathCW interface

Interfacing Oberon code to C code with the `obnc` compiler is more complex than with `obc` because of the need for interface functions. The `MathCW` module looks like this:

```
MODULE MathCW;
...
PROCEDURE erf*    (x : REAL) : REAL;          RETURN 0.0 END erf;    (* C double *)
...
PROCEDURE isqnan* (x : REAL) : INTEGER;       RETURN 0   END isqnan;
PROCEDURE issnan* (x : REAL) : INTEGER;       RETURN 0   END issnan;
...
PROCEDURE qnan*   (s : ARRAY OF CHAR) : REAL: RETURN 0.0 END qnan;   (* C double *)
PROCEDURE snan*   (s : ARRAY OF CHAR) : REAL: RETURN 0.0 END snan;   (* C double *)
...
END MathCW.
```

The asterisk suffix makes the function names and their argument lists known outside the module, but their bodies are dummies, because they are never executed. Instead, the interface is supplemented with C code that looks like this:

```
% cat MathCW.c
#include <obnc/OBNC.h>
#include ".obnc/MathCW.h"
#include <mathcw.h>

#define OBERON_SOURCE_FILENAME "MathCW.mod"

void MathCW__Init(void)                                                 { }
...
OBNC_REAL oberonobnc__MathCW__erf_  (OBNC_REAL x)                       { return erf(x); }
...
OBNC_REAL oberonobnc__MathCW__qnan_ (const char * s, OBNC_INTEGER len) { return qnan(s); }
OBNC_REAL oberonobnc__MathCW__snan_ (const char * s, OBNC_INTEGER len) { return snan(s); }
...
OBNC_INTEGER oberonobnc__MathCW__isqnan_ (OBNC_REAL x)                  { return isqnan(x); }
OBNC_INTEGER oberonobnc__MathCW__issnan_ (OBNC_REAL x)                  { return issnan(x); }
...
```

`obnc` passes one-dimensional array arguments as a pair consisting of a pointer to the first element, followed by an integer length. Thus, the wrapper functions for `qnan()` and `snan()` have two arguments, but the second is not referenced in the function bodies.

`obnc` uses a C compiler that is by default `gcc`, but that choice can be overridden by the environment variable `CC`. We set that variable to point to this script that supplies locations of the mathcw header files and library:

```
% cat gccwrap
#! /bin/sh -
$prefix/bin/dgcc -I$prefix/include "$¨ $prefix/lib64/libmcw.a -lm
```

We can now compile and link a test program like this:

```
% obnc MathCWTest.mod && ./MathCWTest
Test of MathCW library interface in Oberon for obnc compiler

erf(5.00000000000000000E-01) = 5.20499877813046520E-01
...
```

Notice that `MathCW.c` and `MathCW.mod` are *not* mentioned in the test command. `obnc` knows about them because of the `IMPORT` statement in `MathCWTest.mod`, and automatically compiles `MathCW.c` if that has not already been done, creating files in the `.obnc` directory.

O.15.3 The o2c MathCW interface

The `ofront+` translator behind the `o2c` compiler wrapper has language extensions to support an interface to C code. The key features are illustrated with these Oberon declarations:

```
PROCEDURE -include()                            "#include <math.h>";
PROCEDURE -erf*  (x : REAL)      : REAL         "erf(x)";       (* C double *)
PROCEDURE -erff* (x : SHORTREAL) : SHORTREAL    "erff(x)";      (* C float *)
```

The leading hyphen on function names is a shorthand that tells the translator that the function are defined elsewhere in C, and have no Oberon bodies. The asterisk suffixes are needed only if the definitions are exported from the module. The quoted string that follows the argument list, and any return type, expands to a macro in the generated C code. The special function `include()` supplies an additional header file that is needed in the C code. The Oberon code can then invoke, for example, `erf(0.5)` to call the C code version, either provided as an extra object file for `o2c`, or loaded from a link-time library.

The interface from `o2c` to the `mathcw` library then looks like these fragments:

```
MODULE MathCW;
PROCEDURE -include()                             "#include <mathcw.h>";
...
PROCEDURE -erff*  (x : SHORTREAL) : SHORTREAL    "erff(x)";   (* C float *)
PROCEDURE -erf*   (x : REAL)      : REAL         "erf(x)";    (* C double *)
...
PROCEDURE isqnan* (x : REAL) : INTEGER           "isqnan(x)"; (* C double *)
PROCEDURE isqnan* (x : REAL) : INTEGER           "isqnan(x)"; (* C double *)
...
PROCEDURE qnanf*  (s : ARRAY OF CHAR) : REAL     "qnanf";     (* C float *)
PROCEDURE qnan*   (s : ARRAY OF CHAR) : LONGREAL "qnan";      (* C double *)
...
END MathCW.
```

The `ofront+` translator passes n-dimensional arrays as $n + 1$ consecutive arguments: the first is a pointer to the first element, and the next n are `INTEGER` values corresponding to the dimensions, listed from left to right. Thus, the Oberon function `qnan()` is translated to this wrapper macro

```
#define MathCW_qnan(s, s__len)  qnan(s)
```

that, when invoked, discards the string length argument in the call to the C function `qnan()`.

An Oberon module fragment that uses this interface looks like this:

```
MODULE testmcw;
IMPORT MathCW;
...
VAR x : REAL;
x := MathCW.erf(0.5);
...
END testmcw.
```

Compilation, linking, and execution is then simple:

```
env LIBS=-lmcw o2c testmcw.mod -m && ./testmcw
```

O.15.4 The voc MathCW interface

The voc documentation does not discuss how Oberon code can call C code, but experiments demonstrate that it behaves almost identically to o2c, so the same MathCW.mod file can be used for both compilers.

Unfortunately, while voc imports the environment variable CFLAGS to supply additional compiler flags, it has no support for changing the default compiler, gcc, nor for adding linker flags and library options.

The workaround is to supply a private wrapper script, gcc, in a directory that *precedes* system directories in the UNIX PATH variable, the colon-separated list of directories that are searched in order to find executable programs. The wrapper script is simple:

```
% cat gcc
#! /bin/sh -
$prefix/bin/dgcc -I$prefix/include "$¨ $prefix/lib64/libmcw.a -lm
```

A simple change in MathCW.mod that instead uses

```
PROCEDURE -include() "#include <math.h>";
```

can produce a similar interface, MathClm.mod, for the standard C library.

For both o2c and voc, there can be only one instance of the include() function. Thus, if multiple header files, or declarations of functions and variables, are needed, they must be supplied in a single private header file.

O.15.5 The oo2c MathCW interface

The oo2c compiler's interface to C differs from those of the other test compilers. The interface is defined in a module that has a special header, and no final BEGIN block.

An interface to functions in the standard C mathematical function library looks like this:

```
MODULE MathClm [INTERFACE "C"];
...
PROCEDURE erf*  (x : LONGREAL) : LONGREAL; (* C double *)
PROCEDURE erff* (x : REAL)     : REAL;     (* C float *)
...
END MathClm.
```

As with the interfaces for other Oberon compilers, the asterisk suffix identifies an exported symbol. With the asterisk, the translated C code function prototype begins with extern, and without it, with static.

To use the interface in Oberon code, we just need to import the module, and prefix function calls with the module name:

```
IMPORT MathClm;
...
VAR y  : REAL;
    yy : LONGREAL;
```

```
...
y  := MathClm.erfff(0.5);
yy := MathClm.erf(0.5);
```

C prototypes for the library functions in the translated file `obj/MathClm.oh` are automatically generated from the Oberon function declarations, so no `#include <math.h>` directive is needed.

The interface module must be compiled before it can be used, but need not be mentioned thereafter. We can run a test like this:

```
% oo2c    MathClm.mod
% oo2c -M TestClm.mod && bin/TestClm
```

For the interface to the mathcw library, only one minor change is needed, a specification of an additional link library:

```
MODULE MathCW [INTERFACE "C"; LINK LIB "mcw" END];
...
END MathCW.
```

If the library directory is not already known to the linker, then program execution requires it to be supplied via a standard environment variable:

```
% setenv LD_LIBRARY_PATH $prefix/lib64: # C-shell family syntax

% LD_LIBRARY_PATH=$prefix/lib64:        # Bourne-shell family syntax
% export LD_LIBRARY_PATH
```

Alternatively, the library path can be specified via a C compiler option in a wrapper script named `gcc` that is found before the system version:

```
% cat gcc
#! /bin/sh -
$prefix/bin/dgcc -I$prefix/include "$¨ -Wl,-rpath,$prefix/lib64 -lmcw -lm
```

Because the `mathcw.h` header file is not visible to the compiler, any C compiler could have been used in that script in place of `dgcc`.

O.15.6 The xc and xm MathCW interface

The `xc` and `xm` compilers are of considerable interest because they support two programming languages, Modula-2 and Oberon, with documented interfaces between those languages, and from them to C, C++, and Pascal, with preliminary work on an interface to Java.

The compilers run on GNU/LINUX and Microsoft WINDOWS, but alas, only for 32-bit environments. They are a substantial effort, with almost 1.2 million lines of code in five programming languages, and hundreds of pages of typeset documentation. As long as data types are compatible, their admirable goal is to permit interlanguage calling for all of their supported languages.

The `SYSTEM` module supplies additional data types `BOOL8`, `BOOL16`, `BOOL32`, `BOOL64`, `CARD8`, `CARD16`, `CARD32`, `CARD64`, `INT8`, `INT16`, `INT32`, and `INT64` in support of interlanguage communication. However, the 64-bit types are incompletely implemented, and code with integer divisions fails to link because of a missing library function. That function is present in the compiler source tree, but is not in the link libraries.

`xc` and `xm` also have a `SET64` type, but it is unusable, because it is type incompatible with the 32-bit `SET` type, and braced lists of set members are always treated as the latter type.

The default integer sizes of 1, 2, and 4 bytes are small compared to other modern languages, so the 4-byte `LONGINT` type is likely to be preferred over the 2-byte `INTEGER` type, and requires type changes when Oberon code is adapted from use with other compilers.

More seriously for numerical programming, xc and xm set traps for Infinity, NaN, and zero-divide, so occurrences of such values are run-time fatal errors, even when they occur in external libraries. That is in complete opposition to the nonstop computing goal of IEEE 754 arithmetic. Tests show that subnormals are properly supported, but negative zeros are converted to positive zeros, another defect of those compilers' arithmetic.

The interface to code written in C differs from the practices of the other test compilers, but is simple to use:

```
PROCEDURE ["C"] / erff (x : REAL)     : REAL;         (* C float *)
PROCEDURE ["C"] / erf  (x : LONGREAL) : LONGREAL;     (* C double *)
```

The bracketed language name identifies the required calling sequence, and the slash indicates that the following name is an external function, so the code body is omitted. No C header files are mentioned, or needed, because the translated C code has function prototypes generated from the Oberon declarations.

For the mathcw library, all that is needed is for the compiler wrapper script to supply options for the library name and location.

The mathcw library interface file for xc and xm only requires a few additional lines beyond the function prototypes, plus an asterisk suffixed on each function name to make it visible outside the module:

```
MODULE MathCW;

IMPORT SYSTEM;  (* Needed for C-style declarations! *)

TYPE CARDINAL = SYSTEM.CARD32;
     LONGCARD = SYSTEM.CARD32;

PROCEDURE ["C"] / acosdegf* (x : REAL)     : REAL;
...
PROCEDURE ["C"] / acosdeg*  (x : LONGREAL) : LONGREAL;
...
END MathCW.
```

O.16 Decimal arithmetic in Oberon

The interfaces from Oberon to functions in the mathcw library inspire an additional module that makes high-precision decimal floating-point arithmetic available. The version that we supply here is for the voc compiler, but with modest changes, the code can be adapted for the other compilers. Because Oberon supports only return of scalars from functions, we need to supply each of them with an additional argument to hold the function result, an array of four 32-bit integers that contains the 128-bit decimal value. We also need to simplify calls to functions in C that support variable argument lists, so that their Oberon counterparts can have fixed argument counts.

We leverage the C preprocessor to simplify the programming via this header file:

```
% cat DecOberon.h
#include <mathcw.h>
#include <stdint.h>
#include <stdio.h>
#include <string.h>

extern int (__isoc99_sscanf)    (const char * s, const char * fmt, void * presult);
// extern int sscanf            (const char * s, const char * fmt, void * presult);

#define CONS(a,b)       (a##b)
#define FP(x)           CONS(x,DL)

typedef decimal_long_double TDEC;
```

```
#define DLD(t)          (*(TDEC *)&t[0]) /* coerce Tdec to C TDEC */
#define INT32(t)        ((int)(t))      /* coerce Oberon integer types to C int */
#define INT64(t)        ((int)(t))      /* coerce Oberon integer types to C int */

#define FUN1(result, fun, x)       \
        do { TDEC r_; r_ = (fun) (DLD(x));           RESULT(result, r_); } while (0)
#define FUN2(result, fun, x, y)    \
        do { TDEC r_; r_ = (fun) (DLD(x), DLD(y));   RESULT(result, r_); } while (0)
#define FUN2I(result, fun, x, n)   \
        do { TDEC r_; r_ = (fun) (DLD(x), (int)(n)); RESULT(result, r_); } while (0)
#define FUN3(result, fun, x, y, z) \
        do { TDEC r_; r_ = (fun) (DLD(x), DLD(y), DLD(z)); RESULT(result, r_); } while (0)

#define RESULT(result, y)       (void)memcpy(&result, &y, 16)
```

That header file is needed by the Oberon module with this code:

```
% cat DecCW.mod
MODULE DecMCW   (*INTERFACE "C"; LINK LIB "mcw" ADD_OPTION LibMCWPrefix END*);

IMPORT SYSTEM;

TYPE Tdec* = ARRAY 4 OF INTEGER; (* 4 32-bit integers hold 128-bit decimal_long_double *)

PROCEDURE -include()    '#include "DecOberon.h"';

(* assignment operations via compiler type conversion: TDEC x <- (TDEC)(x) *)
PROCEDURE -set*     (VAR result : Tdec; x : LONGREAL)
                    "{TDEC r_; r_ = (TDEC)(x); RESULT(result, r_); }";
PROCEDURE -seti*    (VAR result : Tdec; x : HUGEINT)
                    "{TDEC r_; r_ = (TDEC)(x); RESULT(result, r_); }";

(* IEEE 754 basic operations *)

PROCEDURE -add*     (VAR result : Tdec; VAR x, y : Tdec)
                    "do {TDEC r_; r_ = DLD(x) + DLD(y); RESULT(result, r_); } while (0)";
PROCEDURE -div*     (VAR result : Tdec; VAR x, y : Tdec)
                    "do {TDEC r_; r_ = DLD(x) / DLD(y); RESULT(result, r_); } while (0)";
PROCEDURE -mul*     (VAR result : Tdec; VAR x, y : Tdec)
                    "do {TDEC r_; r_ = DLD(x) * DLD(y); RESULT(result, r_); } while (0)";
PROCEDURE -sub*     (VAR result : Tdec; VAR x, y : Tdec)
                    "do {TDEC r_; r_ = DLD(x) - DLD(y); RESULT(result, r_); } while (0)";

PROCEDURE -divbyint*(VAR result : Tdec; VAR x : Tdec; n : INTEGER)
                    "do {TDEC r_; r_ = DLD(x) / (int)(n); RESULT(result, r_); } while (0)";
PROCEDURE -mulbyint*(VAR result : Tdec; VAR x : Tdec; n : INTEGER)
                    "do {TDEC r_; r_ = DLD(x) * (int)(n); RESULT(result, r_); } while (0)";

PROCEDURE -rsqrt*   (VAR result, x : Tdec) "FUN1(result, rsqrtdl, x)";
PROCEDURE -sqrt*    (VAR result, x : Tdec) "FUN1(result, sqrtdl,  x)";

PROCEDURE -fma*     (VAR result, x, y, z : Tdec) "FUN3(fmadl, x, y, z)";
```

```
  (* Elementary and special functions, ordered alphanumerically in functional groups *)

  PROCEDURE -acos*    (VAR result, x : Tdec) "FUN1(result, acosdl, x)";
  PROCEDURE -asin*    (VAR result, x : Tdec) "FUN1(result, asindl, x)";
  PROCEDURE -atan*    (VAR result, x : Tdec) "FUN1(result, atandl, x)";
  PROCEDURE -atan2*   (VAR result, y, x : Tdec) "FUN2(result, atan2dl, y, x)";

  PROCEDURE -cos*     (VAR result, x : Tdec) "FUN1(result, cosdl, x)";
  PROCEDURE -sin*     (VAR result, x : Tdec) "FUN1(result, sindl, x)";
  PROCEDURE -tan*     (VAR result, x : Tdec) "FUN1(result, tandl, x)";

  PROCEDURE -erf*     (VAR result, x : Tdec) "FUN1(result, erfdl,  x)";
  PROCEDURE -ierf*    (VAR result, x : Tdec) "FUN1(result, ierfdl, x)";

  PROCEDURE -exp*     (VAR result, x : Tdec) "FUN1(result, expdl,   x)";
  PROCEDURE -exp2*    (VAR result, x : Tdec) "FUN1(result, exp2dl,  x)";
  PROCEDURE -exp10*   (VAR result, x : Tdec) "FUN1(result, exp10dl, x)";

  PROCEDURE -expm1*   (VAR result, x : Tdec) "FUN1(result, expm1dl,   x)";
  PROCEDURE -exp2m1*  (VAR result, x : Tdec) "FUN1(result, exp2m1dl,  x)";
  PROCEDURE -exp10m1* (VAR result, x : Tdec) "FUN1(result, exp10m1dl, x)";

  PROCEDURE -log*     (VAR result, x : Tdec) "FUN1(result, logdl,   x)";
  PROCEDURE -log2*    (VAR result, x : Tdec) "FUN1(result, log2dl,  x)";
  PROCEDURE -log10*   (VAR result, x : Tdec) "FUN1(result, log10dl, x)";

  PROCEDURE -log1p*   (VAR result, x : Tdec) "FUN1(result, log1pdl,   x)";
  PROCEDURE -log21p*  (VAR result, x : Tdec) "FUN1(result, log21pdl,  x)";
  PROCEDURE -log101p* (VAR result, x : Tdec) "FUN1(result, log101pdl, x)";

  PROCEDURE -ipow*    (VAR result : Tdec; VAR x : Tdec; n : INTEGER) "FUN2I(result,ipowdl,x, n)";
  PROCEDURE -pow*     (VAR result : Tdec; VAR x, y : Tdec) "FUN2(result, powdl, x, y)";

  PROCEDURE -lgamma*  (VAR result, x : Tdec) "FUN1(result, lgammadl, x)";
  PROCEDURE -tgamma*  (VAR result, x : Tdec) "FUN1(result, tgammadl, x)";

  (* Conversion functions: decimal <-> string *)

  PROCEDURE -sscanf1*   (s : ARRAY OF CHAR; fmt : ARRAY OF CHAR; VAR result : Tdec) : INTEGER
                        "(sscanf)(s, fmt, &result)";
  PROCEDURE -sprintf1*  (VAR s : ARRAY OF CHAR; fmt : ARRAY OF CHAR; x : Tdec)
                        "(void)(sprintf)(s, fmt, DLD(x))";
  PROCEDURE -sprintf1i* (VAR s : ARRAY OF CHAR; fmt : ARRAY OF CHAR; x : INTEGER)
                        "(void)(sprintf)(s, fmt, INT32(x))";

  END DecMCW.
```

The module provides the basic floating-point operations required by IEEE 754 arithmetic, plus a subset of elementary and special functions that are typical of the C mathematical library. In the quoted C fragments, function names are generally parenthesized to prevent macro substitution.

The special handling of the `sscanf()` function deals with its redefinition by a header file on the test system.

The C compiler handles the conversion of scalar integer and floating-point values to decimal equivalents via the `seti()` and `set()` functions. A lengthy test module exercises the decimal interface, and one function from it

demonstrates two ways to compute the exponential function, $\exp(x)$, by a library call, and by the standard series expansion:

$$\begin{aligned}
\exp(x) &= \sum_{k=0}^{\infty} x^k/k!, && \textit{defining series for exponential function,}\\
&= 1 + x + x^2/2! + x^3/3! + \cdots, && \textit{initial terms of series,}\\
t_0 &= 1, && \textit{zeroth term,}\\
t_1 &= x, && \textit{first term,}\\
t_k &= x t_{k-1}/k, && \textit{general term recurrence.}
\end{aligned}$$

The initial part of the test module, and the test of the exponential function, looks like this:

```
% cat TestDecMCW.mod
MODULE TestDecMCW;

IMPORT DecMCW, Out, SYSTEM, Utah;

CONST DEC128LEN = 80;
      fmtin = "%DLe";
      fmtout = "%# .34..3DLg";
      PiString = "3.14159_26535_89793_23846_26433_83279_50288_41971_69399_37510_58210";
      tab = 9X;

TYPE Ttmpbuf = ARRAY 8192 OF CHAR;
TYPE Ttmpfmt = ARRAY 20 OF CHAR;

PROCEDURE -include()    '#include "DecOberon.h"';

PROCEDURE TestExp ();
VAR k, retval : INTEGER;
    s : Ttmpbuf;
    sum, t0, t1, tk, x, y : DecMCW.Tdec;
BEGIN
    retval := DecMCW.sscanf1("0.1", fmtin, x);
    CheckRetval(retval);

    Out.String("Test of DecMCW.exp(0.1)");
    Out.Ln;
    Out.Ln;

    DecMCW.exp(y, x);
    DecMCW.sprintf1(s, fmtout, y);

    Out.String("DecMCW.exp (0.1) = ");
    Out.String(s);
    Out.Ln;

    DecMCW.seti(sum, 0);
    DecMCW.seti(t0, 1);
    t1 := x;
    tk := t1;
```

```
    FOR k := 2 TO 30 DO            (* tk = x**k / k! = x * tkm1 / k *)
        DecMCW.divbyint(tk, tk, k);
        DecMCW.mul(tk, x, tk);
        DecMCW.add(sum, sum, tk)
    END;

    DecMCW.add(sum, t1, sum);
    DecMCW.add(sum, t0, sum);

    DecMCW.sprintf1(s, fmtout, sum);
    Out.String("sum for exp(0.1) = ");
    Out.String(s);
    Out.Ln;
    Out.Ln

END TestExp;
```

Here is a run of the test program:

```
% voc -OC TestDecMCW.mod -m && ./TestDecMCW
...
Test of DecMCW.exp(0.1)

DecMCW.exp (0.1) =  1.105_170_918_075_647_624_811_707_826_490_247
sum for exp(0.1) =  1.105_170_918_075_647_624_811_707_826_490_247
```

Programming with the `DecCW` module lacks the ability to use normal mathematical unary and binary operators, and requires function calls to do the job, similar to what the C programmer has to do for a multiple-precision arithmetic library. Nevertheless, the Oberon code for the summation is easy to read, and the `voc` compiler generates C code that carries the overhead of two function calls, and the copying of a 128-bit result into the first argument of the Oberon function. However, that is modest compared to the work inside a software implementation of decimal arithmetic.

O.17 Summary

For numerical programming in Oberon, considerably more work needs to be done to provide a satisfactory, portable, and accurate set of module functions. For programmer familiarity, they should be based on functions recommended in the IEEE 754 Floating-Point Standards, as well as those of recent C and Fortran standards. Functions are needed for both floating-point sizes supported by the Oberon language for at least these operations:

- exact scaling;
- extraction of sign, exponent, and significand;
- reconstruction from sign, exponent, and significand;
- access to IEEE 754 rounding modes and exception flags;
- access to IEEE 754 precision control for CPUs that support it;
- the common elementary and special functions supplied by the older languages;
- precise control of output formatting;
- correct default digit counts, using the Matula formula: of the test compilers, only `oo2c` has correct counts, but `obnc`, `xc`, and `xm` have been patched locally for correctness;

- input, output, getting, and setting of payloads in quiet and signaling NaNs;
- hexadecimal floating-point constants, and binary, octal, and hexadecimal integer constants in code, input, and output;
- property test functions (Infinity, NaN, finite, normal, subnormal, negative zero, exactly representable as integer, ...).

The `MathCW.mod` interface files provide Oberon programs with solutions for most of those points, but it would be desirable to have the operations available as part of *standard* library modules, ideally a single one with a short name, like `FP.mod`, because the module name must be prefixed to all function calls.

Oberon compilers must handle IEEE 754 arithmetic in nonstop mode, and not terminate on division-by-zero, or on encountering an Infinity or NaN.

Compiler and language changes are needed to remedy the sloppy handling of floating-point constants: any such value specified with a large number of digits should automatically be given the highest available precision, *independent* of any exponent letter.

Hexadecimal integers with the high bit set should *not* be rejected by compilers. Of the test compilers, only `o2c`, `xc`, and `xm` accept hexadecimal constants with a high 1-bit, but the `o2c Out.Hex()` function botches their output.

Support for *unsigned* bit logical operations, getting and setting of subfields, complementing, masking, shifting, and rotations is needed, even if the numbers themselves are stored as signed integers. While Oberon sets are one way of representing bit operations, they are inadequate for representing multibit masks, and provide no efficient way of supporting shifts and rotations, even though hardware is almost universally capable of doing those operations in single instructions.

Escape sequences should be available in strings, including for Unicode characters.

Improved support is needed for strings, including searching for characters and substrings, safe concatenation, and efficient substring extraction. Input/output operations from and to strings is also required.

Predictable integer sizes are needed, and the longer sizes of 64 and 128 bits that are supported by Rust and compilers for some other languages should be added to Oberon. The rising use of various 8- and 16-bit floating-point formats in neural networks, and in image and signal processing, suggests that they too should be supported, as should the IEEE 754 128-bit format, even if it has to be implemented in software.

Sets and set operations can be a useful programming tool, but only if their sizes depend on *need*, not on the *size* of a default integer word. Because sets are declared at compile time, when they are small enough, compilers can use efficient inline bit instructions on single words; handling of larger sets can be done with compiler-generated calls to library functions, or simply unrolled inline.

Interfaces from other programming languages to C are critically important capabilities, because of the enormous existing base of code in that language in all modern operating systems, including libraries for filesystems, networks, operating systems, Web tools, and window systems of desktops and mobile devices. It would therefore have made sense for Oberon compiler developers to adopt a uniform interface syntax, but that clearly did not happen.

Some of the test compilers supply the standard mathematical functions with simplistic code of dubious accuracy and robustness, rather than simply invoking their generally reliable implementations in the C library. For example, one of them has a `sincos()` function that computes the sine, s, and then sets the cosine to $\sqrt{1 - s^2}$. At the very least, it should have used $\sqrt{(1 - s)(1 + s)}$ to avoid accuracy loss from underflow, and it still fails to correctly handle negative zero, and does not check for Infinity and NaN arguments.

Software packaging of most of the test compilers is antiquated and does not follow modern practices of a `configure` script and a top-level `Makefile` that guides the building, validation, and installation of the compilers and their libraries and modules in site-dependent locations. Instead, compiler-specific UNIX shell scripts are used to carry out the builds in ways that are hard to trace, alter, and repair. The shell scripts are sometimes not portable across systems, because they exploit extended features of particular UNIX shells, and they sometimes contain hard-coded pathnames and compiler settings that prevent the Oberon programmer from changing compilers, or adding options for debugging or optimization, and for supplying additional locations of header files and libraries.

Finally, this author's experience with the Oberon compilers demonstrates that they are *not* compilers for a single programming language, but instead represent *six distinct language dialects* because of their different choices of numeric data types, and their rather different module offerings. About a third of the code lines in a test suite of more than 3000 lines have to be changed to move from one compiler to another.