Oxford Oberon-2 Compiler

Users' Manual: version 2.9

Mike Spivey

Oxford University Computing Laboratory

Draft of 15.xii.2010

Contents

Chapter 1

Introduction

This manual explains how to use the Oxford Oberon–2 compiler *obc* to compile and run Oberon–2 programs. The content is largely extracted from the laboratory manual used by our undergraduates in Oxford for their second programming course (their first course is functional programming with Haskell).

The following typographical conventions are used throughout this manual:

- Names of programs are shown in *bold face* type.
- File names are shown in italic type.
- The text of Oberon programs is shown in sans serif type, except that identifiers in the running text are shown in *italic*.
- The text of UNIX commands is shown in `typewriter` type. Where an interaction with the computer is shown, the characters you type are shown in *`italic typewriter`* type, and the computer's responses are shown in `upright typewriter` type.

Chapter 2

Using Oberon

This chapter will tell you how to use the Oberon compiler *obc* to compile and run Oberon programs.

2.1 Introducing the Oberon compiler

Before your Oberon program can be run, it must first be translated by the Oberon compiler into a form that can be directly obeyed by a machine. For example, Figure 2.1 shows a little Oberon program for computing factorials. As you can see, the name of the file exactly matches the name of the module *Fac* that it contains, even down to the capitalization of the name. (In the software lab at Oxford, you can find a copy of this file at /usr/local/practicals/ip1/examples/Fac.m.) To translate this program with the Oberon compiler, give the command[1]

```
$ obc -o fac Fac.m
```

This command runs the Oberon compiler *obc*, asking it to translate the program found in Fac.m and place the translation ("`-o fac`") in the file fac.[2]

When you have done this, you can run the program by giving the UNIX command `./fac`, like this:

```
$ ./fac
Gimme a number: 4
The factorial of 4 is 24
Gimme a number: 5
The factorial of 5 is 120
Gimme a number: 6
The factorial of 6 is 720
Gimme a number: -1
$
```

[1] In this manual, I've shown the shell's prompt as $, but it may appear differently, depending on the shell that your account has been set up with. The characters you type are shown in *slanted type*.

[2] If you don't say "`-o fac`", the compiler rather unhelpfully puts the translation in a file called a.out, according to a long-standing UNIX tradition.

```
MODULE Fac;

IMPORT In, Out;

VAR n, i, f: INTEGER;

BEGIN
  LOOP
    Out.String("Gimme a number: "); In.Int(n);
    IF n < 0 THEN EXIT END;

    i := 0; f := 1;
    WHILE i < n DO i := i+1; f := f*i END;

    Out.String("The factorial of "); Out.Int(n, 0);
    Out.String(" is "); Out.Int(f, 0); Out.Ln
  END
END Fac.
```

Figure 2.1: *Contents of file* Fac.m

As you can see in Figure 2.1, the program contains a loop that repeatedly asks for a number and computes its factorial, until a negative number is input. With our usual setup at the Computing Laboratory, you have to type `./fac` as the UNIX command for starting *fac*, to reflect the fact that the file fac is in the current directory, not one of the directories where UNIX usually keeps executable programs.[3]

The file fac contains low-level instructions that are equivalent to the high-level program in Fac.m. Once the translation has been done, you could delete the file Fac.m and still run the low-level version in fac; or you could copy the file fac and give it to a customer, who could run the program without needing to have the source code.

There's something a little unusual about the Oberon compiler we'll be using, and that's the fact that (like most compilers for Java) it doesn't actually translate your program into instructions for the physical computer you are using. Instead, the designer of the compiler has invented a simple 'virtual' computer specially for implementing Oberon, and the file fac contains instructions for that computer instead. The gap between the virtual computer and the actual, physical computer is bridged in one of two ways: either by means of an *interpreter* program that uses the physical computer to carry out a simulation of the virtual computer, or by means of a *just-in-time translator* (JIT) that compiles code for the physical machine after the program is loaded, but before each piece of it starts to run. The biggest advantage of using a virtual machine is that the compiler can be *ported* to a new computer just by making a new simulator – and since the simulator is itself written in a highish-level language (C), that's usually just a matter of re-compiling it on the new computer.

The biggest disadvantage of this scheme is that programs run more slowly than they would with a conventional compiler. The interpreter-based imple-

[3] I find this need to type `./fac` instead of just `fac` very annoying. If you hate it as much as I do, ask the demonstrator how to change it, after asking about the tiny security risk involved.

```
MODULE FibMain;

IMPORT In, Out, FibFun;

VAR n, i, f: INTEGER;

BEGIN
  LOOP
    Out.String("Gimme a number: "); In.Int(n);
    IF n < 0 THEN EXIT END;

    f := FibFun.Fib(n);

    Out.String("fib("); Out.Int(n, 0);
    Out.String(") = "); Out.Int(f, 0); Out.Ln
  END
END FibMain.
```

Figure 2.2: *Contents of file* FibMain.m

mentation is perhaps 5 to 10 times slower than a code from a conventional compiler, whilst the JIT is also slower, because it generates code in a simple way without a lot of optimization. But we won't be writing any programs that run for a very long time, so that won't be too much of a disadvantage.[4]

2.2 Programs with more than one module

Bigger programs are often divided into several modules for ease of understanding. If this division into modules is done judiciously, it should be possible to understand a lot about each module without looking at the other ones. A well-designed module has a small interface, and it is possible to say *what* the module does without giving the details of *how* it does it.

As a contrived example, let's look at a variation on our program for calculating factorials that calculates Fibonacci numbers instead. Calculating Fibonacci numbers is a simple task that can be done in several ways, some more complicated and efficient than others. That makes it sensible to introduce a procedure, so as to separate the *what* – the Fibonacci function – from the *how* – the algorithm for calculating it.

Putting the procedure in its own module is over-kill, since there are no details to hide that are not already hidden inside the procedure. But doing so will let us show how to split a program into modules, so let's do it anyway. Figure 2.2 shows a program like our earlier factorials program, but with the code to calculate the factorial replaced by a call to the function *FibFun.Fib*, imported from a module *FibFun*. Figure 2.3 shows the text of that module, including a procedure that calculates Fibonacci numbers. Note the export mark $*$ in the heading of this procedure.

To compile this program, we must separately translate the two modules into machine code, then combine the two files of machine code into one

[4] Actually, there's a hidden advantage: having an artificially slowed-down computer will encourage you to learn how to write programs that make economical use of computer time.

```
MODULE FibFun;

PROCEDURE Fib*(n: INTEGER): INTEGER;
  VAR a, b, c, i: INTEGER;
BEGIN
  IF n = 0 THEN
    RETURN 0
  ELSE
    a := 0; b := 1; i := 1;
    WHILE i < n DO
      c := a + b; a := b; b := c;
      i := i + 1
    END;
    RETURN b
  END
END Fib;

END FibFun.
```

Figure 2.3: *Contents of file* FibFun.m

program. Although all this can be done with the single command

```
$ obc -o fib FibFun.m FibMain.m
```

it is better to separate the three steps, so that we need not repeat all the steps if we change only some of the modules. With a small example like this, it doesn't make a significant difference: but as I've already admitted, using modules at all in this example is over-kill. To compile the two modules separately, we need the three commands

```
$ obc -c FibFun.m
$ obc -c FibMain.m
$ obc -o fib FibFun.k FibMain.k
```

The first command translates the *FibFun* module, producing a file FibFun.k that contains the virtual machine code, and also some information about the exported interface of the module. The second command translates the *FibMain* module into a file FibMain.k of virtual machine code; during the translation, the compiler reads the file FibFun.k to check that the function *FibFun.Fib* is used in a way that is consistent with its definition. The third command combines the two files of machine code with the code for the library modules *In* and *Out* to produce the executable file fib.

The files FibFun.k and FibMain.k containing the low-level instructions are in a textual format, so if you are curious you can examine them with a text editor. Don't try changing them, though: that will probably result in chaos. You will see that the files contain lines from your source program in the form of comments. These are completely ignored by the simulator, but are there to help us track down problems with the Oberon compiler.

```
1  # Makefile for fib
2
3  FIB = FibFun.k FibMain.k
4  fib: $(FIB)
5          obc -o fib $(FIB)
6
7  FibMain.k: FibFun.k
8
9  %.k: %.m
10         obc -c $<
```

Figure 2.4: *Contents of file* Makefile

2.3 Managing compilation with make

If you are writing a program, perhaps in several modules, and trying to get it to work, it becomes very boring to have to type out a long sequence of commands each time you want to compile it. Also, if we make a change to only one of the modules in a big program, we want to avoid recompiling all of it so as to save time.

A partial solution to this problem would be to write a *shell script* for compiling the program: that is, a sequence of UNIX commands that is stored in a file. This solves the problem of typing a long command or sequence of commands each time we want to compile our program; but there's a better way that uses a nifty program called *make*. This program lets you write a script that, like a shell script, contains the UNIX commands for compiling your program. Unlike a shell script, the *make* script associates each of these commands with a file that it produces. The *make* program uses the modification time that UNIX associates with each file to work out which files in your program have been changed, and issues only the commands that are needed to bring everything up to date. Usually, there's just one script that describes how to compile all the programs in a directory, and the script is stored in the same directory under the name Makefile.

For example, Figure 2.4 shows a makefile for our little Fibonacci program: I've numbered the lines for ease of reference. In this course, you will not need to know how to write makefiles, because a working makefile will be provided when it is needed. Nevertheless, *make* is such a useful program that it's good to understand something of what it can do.

Line 1 of the makefile is a comment: it starts with # and continues to the end of the line. Lines 3–4 describe the last of the three commands needed to build the *fib* program, using the list of files `FIB` defined on line 2. Line 3 records the fact that the file fib is made from the two files FibFun.k and FibMain.k, so that it should be rebuilt if either of these two files is newer than it; and line 4 (starting with a tab character) gives the UNIX command to execute in this case.

The other two commands needed to build *fib* are summarized in the 'pattern rule' on lines 6–7; this says that any file ???.k can be made from the corresponding file ???.m by running the command `obc -c ???.m`. Line 5 records the fact that rebuilding FibFun.k requires FibMain.k to be rebuilt too, because the interface of the *FibFun* module may have changed.

Suppose we were to modify our program, perhaps to make it more acceptable to English tastes by replacing the word "Gimme" by the polite phrase "Kindly enter". To do this, we would use a text editor on the file FibMain.m, and our final action with the editor would be to save the new file over the old one. This would cause UNIX to change the modification time of the file, making it newer than the file FibMain.k that was generated by the compiler last time we built the program. Now we simply give the command

```
$ make
```

The *make* program examines the rule that says how to make FibMain.k from FibMain.m and sees that the time-stamps are out of step. So it runs the Oberon compiler and creates an up-to-date version of FibMain.k. This makes FibMain.k newer than the final program fib, so *make* also runs the command to rebuild fib. Since we haven't touched the *FibFun* module, it does not need to be rebuilt, and *make* omits the command to build it. If we immediately run *make* again, it will do nothing the second time, because each file will already be up to date with respect to the files from which it is built. So it's quick and easy to use *make* simply to check that everything is up to date.

Two very common mistakes in using editors and *make* are these: forgetting to save a file in the editor before running *make*, and forgetting to run *make* after changing your program. The first kind of mistake is noticeable, because *make* does less work than you expected: if you change a source file, it ought to be recompiled. Both kinds of mistake become noticeable when your program continues behaving exactly as it did before you made your changes.

2.4 When things go wrong

There are many kinds of things that can go wrong in writing a program:

- You might write a program that does not conform to the grammatical rules of Oberon.
- You might write a program that, although grammatically correct, does not make any sense because you have (for example) forgotten to declare some of the variables you use.

Because of the way the Oberon compiler has been written, it checks that your program is grammatically correct before it begins to address the question of whether the program makes sense, and it checks the whole program for sense before it starts to generate the machine code translation. So if you write a grammatically incorrect program that also contains undeclared identifiers, then the compiler will not bother to mention the missing declarations until you have put the grammar right; and if the program contains grammatical errors, if declarations are missing or if the types do not match up properly in expressions, the compiler will not generate any machine code for your program until you have put it all right.[5]

After detecting one grammatical error in your program, the compiler tries to continue with its analysis of the rest of your program, so that you can

[5] This is a bit annoying when the only error is a missing semicolon, I admit.

correct more than one error each time you run the compiler. The compiler works by reading your program from beginning to end, and it produces its *first* error message when it reaches the first word or symbol in your program that could not be the next symbol in any valid Oberon program. So the first error message at least is reliable.

After the first error, the compiler has the problem of re-construing your program in such a way that it can continue and look for other errors. The methods used for this are necessarily crude, because after all your program is wrong, and the compiler can only guess at what you meant. The compiler uses rules for recovery like "skip to the next semicolon, and see if what follows looks like the beginning of another statement". This means that the compiler can diagnose as another error in your program a problem that is in fact caused by the compiler's own state of confusion. For example, the compiler might skip over an **end** keyword as it looks for a place to restart, and that might cause it to think that a later procedure declaration appears in the middle of the current procedure. There's no way to avoid this problem in general, so it's best to be aware that only the first error message is really worth relying on.

When errors are reported by the compiler, the combination of the *obc* compiler, *make* and the *emacs* editor is particularly productive. *Emacs* and its X version *xemacs* provide a command that you can run by using the menus or toolbar under X, or by typing `M-x compile`;[6] this uses *make* to recompile your program, and displays the output of that process, including any messages from the compiler, in an *Emacs* buffer. The *obc* compiler puts its error messages in a standard format, like this:

```
"Planner.m", line 92: missing ';' at token 'IF'
>       IF verbose THEN ShowLink(u, "green") END;
>       ^^
```

Another *emacs* command `next-error` (bound to the key sequence[7] `C-x ‘`) recognizes these error messages and finds the relevant file and the relevant line, putting the editor's cursor there ready to correct the error. Subsequent invocations of `next-error` will take you to the locations of other errors reported by the compiler.

Other things can go wrong when you run your program:

- You might write a program that performs an illegal operation when it runs. Examples of such illegal operations are accessing an array with an index that is negative or greater than the length of the array, and following a pointer that has been set to **nil**.

Errors like these are detected when the program runs, because there is no generally applicable way for the compiler can work out in advance whether such errors can happen.[8] The advantage of a language like Oberon is that the compiler can generate code that checks for such errors, and stops the

[6] Hold down the Alt key and press x, then release both, type the word `compile`, and press Return.

[7] Hold down the control key while pressing letter x, then release both and press ‘ (grave accent).

[8] Actually, we can *prove mathematically* that no compiler could do this, because it is equivalent to solving the 'halting problem' for Turing machines. The intriguing theory of such things is covered in the second year course *Models of Computation*.

program immediately. For an array reference $a[i]$, the compiler generates a check that $0 \leq i < N$, where N is the length of a. If this is not true, then the program is stops with a message such as

```
Runtime error: array bound error
                on line 123 in module Directory
```

The Oberon language has been very cleverly designed so that errors of this kind can be detected immediately with simple compiler-generated checks. This property does not hold for C, for example, so C programmers are always trying to track down the cause of baffling problems.[9]

Some things that can go wrong do not result in any helpful message:

- Your program may run forever, without producing any output. A common cause of this is that you've written a loop

  ```
  WHILE i < n DO ...
  ```

  but have forgotten to put $i := i + 1$ in the loop body.

- Your program may produce output that is wrong. In this case, there's no substitute for careful thought, but you can gather evidence about what is happening either by adding code at suitable points in the program that prints relevant information about the value of variables, or by using a debugger (see the next section).

- Your program may produce the right output, but run too slowly to be useful. In this case, the solution is *not* to start madly 'optimizing' everything in sight in the hope of speeding things up. If you do that, you will probably spend most of your effort on parts of the program that have almost no effect on the overall speed, and you will certainly make your program more complicated, perhaps so much so that it stops working properly, and you end up in a worse position than when you started. A better alternative is to use *profiling* to find out where the program is spending its time: see Section 2.6.

2.5 A debugger

If a program produces the wrong output, then it's often hard to see from the output what could be going wrong. In order to investigate further, you can take some steps for yourself. Adding assertions to the program, particularly at the start of procedures, can help to pin down where the fault lies. You can also add printing code to the program that gives a kind of running commentary on what the program is doing. If the program has significant internal data structures, then it is well worth the effort of writing a general, robust print routine that prints out those data structures in human-intelligible form. A convenient alternative to instrumenting the program by hand is to use a debugger, a utility program that allows you to watch and control the actions of your program as it runs.

To start the debugger, use a command like

```
$ obdb ./total
```

[9] That's why debuggers are so popular with C programmers: when a program crashes, they use a debugger to try and work out what has happened.

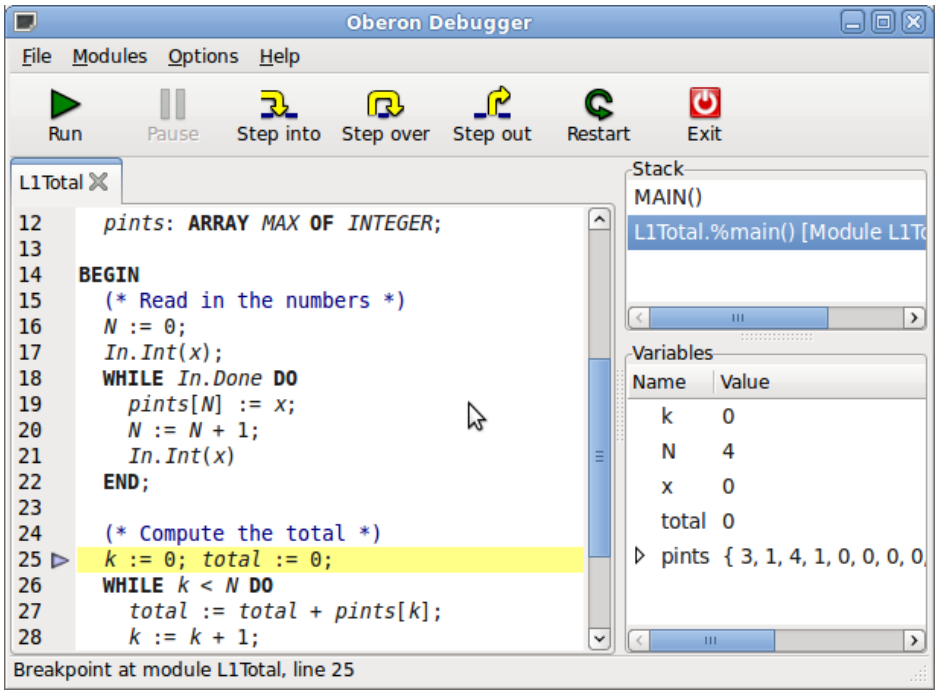

Figure 2.5: *Oberon debugger*

where total is a compiled Oberon program. Any arguments for the program can appear after the name of the program on the command line.

The debugger shows a window like Figure 2.5. On the left, you see a listing of the source code; the margin shows line numbers and a little triangle that indicates the next line to be executed. You can step through the program line by line by clicking on one of the three buttons 'Step into' (which stops inside any procedure that is called), 'Step over' (which continues until the next line of the current procedure) or 'Step out' (which continues until the current procedure returns). If the program consists of more than one module, then the 'notebook' of source listings will have several pages.

At the top right, you see a representation of the subroutine stack of the program. Since the program shown here contains no procedures, there is only one significant entry, the subroutine `L1Total.%main` that represents the body of the module *L1Total*. By clicking on different entries in the list, you can view the source code for different procedures in the left-hand pane, with the current location highlighted with the yellow bar. The little blue triangle always shows the current execution point, even if the yellow bar has been moved to another procedure.

At the bottom right, the local variables for the currently selected procedure are shown. Arrays and records are initially shown in a 'closed' state, with just the first few elements visible on one line, but you can open them by clicking on the little triangle, and see the contents listed with each element on a separate line. The display can be changed to different active procedures by clicking in the stack display.

If you click in the left margin of the source display, you can set and clear breakpoints, which will stop the execution of the program when they are reached. If you click on a line which does not correspond to any actual

instructions in the program, then the breakpoint may be created a couple of lines further down. Breakpoints are shown by a red circle, and execution always stops at a breakpoint, even if it was started with the 'Step over' or 'Step out' button and would otherwise continue further.

2.6 Profiling

Profiling is a process of gathering statistics about where a program spends its time, as the first step of speeding it up. Profiling makes for more effective optimization, because it lets you concentrate your effort on the things that will really make a difference. In most large programs, the majority of the code accounts for only a small fraction of the execution time – or to put it another way, a small part of the code consumes a big fraction of the time. If you can identify this small part of the code – sometimes called the *inner loop*, then you can devote all your energies to improving it. Perhaps there is a better algorithm or data structure that will speed up the time-consuming task. Or perhaps a different approach to the whole program will make the problem area go away. Time spent on optimizing the rest of the program, apart from the inner loop, may be entirely wasted, because the effect on the overall speed may be negligible.

Our Oberon system provides a profiling tool called *obprof* that is used as follows: if you would normally run your program through the command

```
$ prog arg1 ... argn
```

then you should use the command

```
$ obprof prog arg1 ... argn
```

instead.[10] The *obprof* program is actually an augmented version of the usual Oberon run-time system; it runs your program exactly as usual, but it keeps statistics about what happens as the program runs. Specifically, *obprof* keeps track of what procedure in your program is active, and measures how much time is spent in each procedure. When your program finishes, *obprof* summarizes the information it has gathered by listing the procedures in your program, how often each one was called, and the total time spent in the procedure. Figure 2.6 shows the output that is produced for a trial run of the *fac* program. In this profile, the notation *Fac*.?*main* refers to the main body of the *Fac* module; similarly, *Files*.?*main* refers to the main body of the *Files* module from the standard library, which is executed before our program itself starts.

This profile is from too small an example program and too short a run to draw any firm conclusions, as you can see from the fact that *Out*.*Int* was called only twice. But you can see that a lot of the time was consumed in the process of reading numbers (procedure *In.Int*, which also calls *In.IsDigit*). Since *IsDigit* is such a simple procedure – it just tests whether a character is a digit – perhaps it would be better to do these tests where they are needed, rather than call a procedure to do them.

---

[10] In the Windows version, you have to type something like

```
C:\> obprof prog.exe arg1 ... argn
```

and specify the `.exe` suffix explicitly.

```
Execution profile:
    Ticks    Frac     Cumul   Calls   Procedure
 ------------------------------------------------
      174    31.4%    31.4%       2   In.Int
      155    28.0%    59.4%       1   Fac.%main
       78    14.1%    73.5%       6   In.Char
       56    10.1%    83.6%       7   In.IsDigit
       28     5.1%    88.6%       4   Out.String
       24     4.3%    93.0%       2   In.IsSpace
       14     2.5%    95.5%       2   Out.Int
       10     1.8%    97.3%       1   %Main.%main
        7     1.3%    98.6%       1   Files.%main
        5     0.9%    99.5%       1   Out.Ln
        3     0.5%   100.0%       1   In.%main
        0     0.0%   100.0%       2   Conv.IntVal
        0     0.0%   100.0%       6   Files.Eof
        0     0.0%   100.0%       1   Files.Init
        0     0.0%   100.0%       6   Files.ReadChar
        0     0.0%   100.0%       2   Files.WriteInt
        0     0.0%   100.0%       1   Files.WriteLn
        0     0.0%   100.0%       4   Files.WriteString
Total of 554 clock ticks
```

Figure 2.6: *Execution profile*

The profile data is obtained by counting execution cycles of the simulated virtual machine, and this is not quite the same thing as elapsed execution time. For one thing, the profiler uses an interpreter for the virtual machine code, rather than the JIT translator, so the whole program goes much more slowly under profiling (see the explanation on page 5). Another difference is that some library procedures are actually implemented in C, so no cycles of the virtual machine are needed to execute them. You can see the effect in Figure 2.6, because the procedure *Out.Int* is implemented as a C primitive, and appears to take much less time than *In.Int*, which is implemented in Oberon using the primitive *In.Char*. Apart from these primitives, the other operations of the virtual machine each take a small, fixed time that has the same order of magnitude for each operation, so that the number of clock ticks is a good guide to the actual execution time. Typical operations, each counted as one clock tick, are fetching the value of a variable, or adding two numbers together, or comparing two numbers and jumping if they are equal.

The *obprof* program also provides a more sophisticated form of profiling that takes into account the directed graph of calls between one procedure and another in the program. In such a *call graph profile*, the time spent in each procedure is also charged to its callers, so that you can build up a fuller picture of where time is being spent. In order to select call graph profiling, you need to give the `-g` switch to *obprof*, like this:

```
$ obprof -g prog arg1 ... argn
```

For further details, see the *obprof* manual page.

Profiling fits in well with a style of programming that begins by building a simple program that computes the right answers, preferring simple but perhaps inefficient algorithms and data structures to more complex ones. Thus we might use linear search instead of a hash table, or insertion sort instead of quicksort, just as a way of getting the program working sooner. When the program is working correctly, we can use profiling to identify the places where these simplified design decisions actually have a significant effect on performance, and revise just these decisions to make the program faster. This method leads to programs that are simple, compact and reliable whilst still having good performance. A wise programmer knows when to stop looking for further improvements: there is no point working for days to shave a minuscule amount of runtime from a program that will only be used occasionally.

2.7 Language differences

The Oberon language accepted by the *obc* compiler is that described in the document "Oberon–2 Language Definition", an extension of the document "Oberon Language Definition" that appears as Appendix A of the book

> M. Reiser and N. Wirth, *Programming in Oberon: steps beyond Pascal and Modula–2*, Addison-Wesley, 1992,

with the following exceptions:

- The numeric types are *SHORTINT* (16–bit integers), *INTEGER* (32–bit integers), *LONGINT* (64–bit integers), *REAL* (single-precision floating point) and *LONGREAL* (double-precision floating point). The built-in function *ENTIER* returns *INTEGER* instead of *LONGINT*.
- Within each module or procedure, it is not necessary to order the declarations so that (nested) procedure declarations follow all others. Instead, declarations of constants, types, variables and procedures may appear in any order.
- The scope of a procedure includes the bodies of all procedures defined at the same nesting level, without the need for 'forward declarations'.

Chapter 3

OBC Library Reference

This chapter contains a brief description of the library modules that are supplied with the *obc* compiler. The source code to these modules can be found in the directory /usr/local/lib/obc.

3.1 Module *In*: Standard input

This module provides simple input operations on the standard input channel.

PROCEDURE Char(VAR c: CHAR);

- Input one character.

PROCEDURE Int(VAR n: INTEGER);

- Input a decimal integer.

PROCEDURE Real(VAR x: REAL);

- Input a real number.

PROCEDURE Line(VAR s: ARRAY OF CHAR);

- Input one line of text and store it in *s*, removing the terminating newline character.

VAR Done: BOOLEAN;

- This read-only variable is set to *FALSE* when an input operation reaches the end of the input file.

3.2 Module *Out*: Standard output

This module provides simple output operations on the standard output channel.

PROCEDURE Int(n: INTEGER, width: INTEGER);

- Output an integer in decimal, in a field of at least *width* characters.

PROCEDURE Real(x: REAL);

- Output a real number, using scientific notation if it is very large or very small.

PROCEDURE Fixed(x: REAL; width, dec: INTEGER);

- Output a real number in decimal notation, in a field of at least *width* characters, with *dec* digits after the decimal point.

PROCEDURE Char(c: CHAR);

- Output a character.

PROCEDURE String(s: ARRAY OF CHAR);

- Output a string.

PROCEDURE Ln;

- Output a newline character.

3.3 Module *Err*: Standard error

This module provides simple output operations on the standard error channel, which remains connected to the user's display even when the standard output channel is redirected elsewhere. The interface is identical with that of the module *Out*: thus you can write *Out.Int*(*n*, *w*) to output an integer on the standard output, and *Err.Int*(*n*, *w*) to output it on the standard error channel.

3.4 Module *Math*: Mathematical functions

This module contains various mathematical functions on floating-point numbers.

PROCEDURE Sqrt(x: REAL): REAL;

- Compute the square root of the argument.

PROCEDURE Sin(x: REAL): REAL;

- The sine function.

PROCEDURE Cos(x: REAL): REAL;

- The cosine function.

PROCEDURE Tan(x: REAL): REAL;

- The tangent function.

PROCEDURE Arctan2(y, x: REAL): REAL;

- The function $Arctan2(y, x) = tan^{-1}(y/x)$. It is defined even when one of the arguments is zero, and uses the signs of both arguments to determine the quadrant of the result. The order of the arguments is traditional.

PROCEDURE Exp(x: REAL): REAL;

- The exponential function e^x.

PROCEDURE Ln(x: REAL): REAL;

- The natural logarithm function.

CONST pi = 3.1415927;

- (Approximately.)

All angles are expressed in radians.

3.5 Module *Args*: Program arguments

This module provides access to the command-line arguments given when the Oberon program was started.

VAR argc: INTEGER;

- This read-only variable gives the number of arguments, including the program name.

PROCEDURE GetArg(n: INTEGER; VAR s: ARRAY OF CHAR);

- Copy the *n*'th argument into the string variable *s*. The arguments are numbered from 0 to *argc* − 1, with the name of the program being argument 0.

3.6 Module *Random*: Random numbers

This module provides a pseudo-random number generator. Unless the procedure *Randomize* is called, the sequence of numbers will be the same on each run of the program. Actually, this is quite useful, because it makes the results reproducible.

PROCEDURE Random(): INTEGER;

- Generate a random integer, between 0 and *MAXRAND* inclusive.

PROCEDURE Roll(n: INTEGER): INTEGER;

- Generate a random integer, uniformly distributed between 0 and $n - 1$ inclusive. The result is accurately random only if n is fairly small.

PROCEDURE Uniform(): REAL;

- Generate a random real, uniformly distributed on [0, 1].

PROCEDURE Randomize;

- Initialize the random number generator so that it gives a different sequence of pseudo-random numbers on each run of the program.

CONST MAXRAND = 07FFFFFFFH;

- This constant (equal to $2^{31} - 1$) is the largest number that can be returned by *Random*.

3.7 Module *XYplane*: Simple bitmap graphics

Under X windows, this module provides a very simple monochrome graphics facility.

```
CONST W = 640; H = 480;
```

- These constants give the width and height of the graphics window in pixels. The origin is in the bottom left hand corner.

```
PROCEDURE Open;
```

- Open the graphics window.

```
PROCEDURE Clear
```

- Clear the graphics window to all white.

```
CONST erase = 0; draw = 1;
```

- These constants are used as the *mode* parameter of *Dot*.

```
PROCEDURE Dot(x, y, mode: INTEGER);
```

- Draw (*mode* = *draw*) or erase (*mode* = *erase*) a single pixel at coordinates (*x*, *y*).

```
PROCEDURE IsDot(x, y: INTEGER): BOOLEAN;
```

- Test whether a pixel has been drawn at coordinates (*x*, *y*).

```
PROCEDURE Key(): CHAR;
```

- Test whether a key has been pressed in the graphics window. If so, return the character that was typed; otherwise, return the null character 0_X.

The procedure *Key* allows simple keyboard interaction. It also handles the events generated by X when the graphics window is uncovered, so as to fill in the newly-exposed region; this means that a graphics application should call *Key* in each iteration of its main loop.

3.8 Module *Conv*: Numerical conversions

This module provides an interface to the procedures for converting between numbers and strings that are used for input/output.

```
PROCEDURE IntVal(s: ARRAY OF CHAR): INTEGER;
```

- Return the integer value of a string.

```
PROCEDURE RealVal(s: ARRAY OF CHAR): REAL;
```

- Return the real value of a string.

```
PROCEDURE ConvInt(n: INTEGER; VAR s: ARRAY OF CHAR);
```

- Convert an integer into a decimal string.

3.9 Module *String*: Operations on strings

Here is the place where many useful operations on strings will shortly appear. Only one is provided so far.

PROCEDURE Length(s: ARRAY OF CHAR): INTEGER;

- Return the length of *s* up to the first null character.

3.10 Module *Bit*: Bitwise operations on integers

This module provides various operations that treat integers as arrays of 32 bits.

PROCEDURE And(x, y: INTEGER): INTEGER;

- Bitwise AND: bit *i* of the result is 1 if bit *i* is 1 in both *x* and *y*.

PROCEDURE Or(x, y: INTEGER): INTEGER;

- Bitwise OR: bit *i* of the result is 1 if bit *i* is 1 in either *x* or *y* or both.

PROCEDURE Xor(x, y: INTEGER): INTEGER;

- Bitwise XOR: bit *i* of the result is 1 if bit *i* is 1 in either *x* or *y*, but not both.

PROCEDURE Not(x: INTEGER): INTEGER;

- Bitwise NOT: bit *i* of the result is 1 if bit *i* of *x* is 0.

Chapter 4

Language extensions

The -x flag enables a number of language extensions:

(1) A function may be called as a proper procedure; the result returned by the function is discarded.

(2) Enumeration types are supported, with the syntax

```
TYPE colour = (red, blue, green);
```

(3) CASE statements are allowed for any discrete type (including enumeration types), not just *INTEGER* and *CHAR*.

(4) FOR stamtements are also allowed for any discrete type.

(5) The function *ORD* may be applied to any discrete type, not just *CHAR*.