Ofront ™

Oberon-2 to C Translator, Version 1.0
User Guide

SOFTWARE TEMPL

Author:
Dr. Josef Templ
Lüfteneggerstr. 8/44
4020 Linz, Austria
fon/fax: (++Austria) 732 / 77 89 54

This text has been produced with ETH-Oberon V4 compiled with Ofront 1.0.

Contents

1 Introduction 1
- 1.1 Typographic Conventions 2
- 1.2 What is *Ofront*? 2
- 1.3 Intended Use 4

2 Getting Started 7
- 2.1 Command-line Version 7
- 2.2 Integrated Version 11
- 2.3 Principles of Operation 11
- 2.4 Cross Translation 15

3 C-Compilation and Linking 17
- 3.1 SunOS 4.x (SPARC) 18
- 3.2 SunOS 5.x (SPARC) 19
- 3.3 DEC Ultrix (MIPS) 20
- 3.4 HP-UX (PA-RISC) 21
- 3.5 IBM AIX (RS/6000) 23
- 3.6 SGI IRIX 5 (MIPS) 24
- 3.7 Linux (i386) 25

4 Module *SYSTEM* 27
- 4.1 The Static Role 27
- 4.2 Interfacing with C 28
  - 4.2.1 Name Mapping 29
  - 4.2.2 Pointers 29
  - 4.2.3 Parameter Substitution inside Strings 30
  - 4.2.4 Exporting Code Procedures 30
  - 4.2.5 Return Types and Includes 31
  - 4.2.6 Debugging 33
- 4.3 The Dynamic Role 34

5 Module *Args* 35

6 Exception Handling 37

7 Module Loading 43

8 Garbage Collection and Finalization 47

Appendix

A Supported Architectures and Compilers 49

B Available Libraries 51

C The Programming Language Oberon-2 55

D Grammar of Oberon-2 85

E Limitations of the Implementation 89

F *Ofront* Error Messages 91

1 Introduction

This document serves as a guide for users of *Ofront*, the industry's leading Oberon-2 to C translator. The reader of this guide is expected to have at least a basic understanding of the programming language Oberon and to be able to use the C compiler and linkage editor on the respective target platform. It is also expected that the user of the integrated version (cf. 2.2) has some knowledge about the ETH Oberon system.

For an introduction to the programming language Oberon, please refer to

Wirth N. and Reiser M. (1992). *Programming in Oberon. Steps beyond Pascal and Modula-2*. Addison-Wesley, Wokingham, ISBN 0-201-56543-9.

This book is also available in German as

Wirth N. und Reiser M. (1994). *Programmieren in Oberon. Das neue Pascal.* Addison-Wesley, Bonn, ISBN 3-89319-675-9.

For a thorough description of the ETH Oberon system, consult

Reiser M. (1991). *The Oberon System. User Guide and Programmer's Manual.* Addison-Wesley, Wokingham, ISBN 0-201-54422-9.

For a description of object-oriented programming in general and object-oriented programming using Oberon-2 type-bound procedures in particular, refer to

Mössenböck H. (1993). *Object-oriented Programming in Oberon-2.* Springer Verlag, ISBN 3-540-56411-X.

This book is also available in German as

Mössenböck H. (1993). *Objektorientierte Programierung in Oberon-2.* Springer Verlag, ISBN 3-540-57789-0

1.1 Typographic Conventions

This document uses the following typographic conventions:

Table 1.1 Typographic Conventions

typeface/prefix	Meaning	Example
Abcdef	plain text	This document uses
Abcdef	name	*Ofront*
% Abcdef	C shell command	% strip oberon
$ Abcdef	grammar rule	$ options = ["–" {option}].
Abcdef	programs	PROCEDURE P;
	names from a program	SYSTEM_halt
	program output	pos 10 err 3
Abcdef	pseudo-code	*the Oberon main event loop*

1.2 What is Ofront?

Ofront is a tool that translates Oberon-2 programs into semantically equivalent C programs. Full error analysis is performed on the Oberon input program and in case of no errors up to three files are generated as output.

- an Oberon symbol file (suffix .sym)
- a C header file (suffix .h)
- a C body file (suffix .c)

The C header and body files follow widely used C programming conventions. *Ofront* is also capable of generating main programs by translating the body of a module into a C *main* function. In this case only the body file is generated.

Ofront does not invoke the C compiler or linkage editor. This may be done in separate shell scripts or make files and is inherently dependent on the C compiler and linkage editor being used.

Although normally not read by the user, the C code generated by *Ofront* is kept as readable as possible, nicely formatted, should not produce any C compiler error messages or warnings and is tuned for efficient execution. In fact, an Oberon program translated by *Ofront* can be expected to execute as fast and read as well as an equivalent hand-coded C program.

The following list and the subsequent explanations give an overview of the most important highlights of *Ofront*.

Highlights

- full Oberon-2 language support
- extensible module interfaces
- fast translation
- parameterization for arbitrary C compilers, ANSI and K&R
- highly compact and efficient run-time system
- automatic garbage collection
- advanced heap management (growth on demand, finalization)
- commands and modules preserved
- dynamic loading of modules or subsystems
- interfacing with C or other foreign languages
- clean and human-readable C code, no warnings
- information hiding preserved in the generated header files
- multiple libraries available
- command-line version and integrated development environment

Ofront supports the full Oberon-2 language standard as proposed by ETH Zurich. In addition to pure Oberon, this includes the FOR-statement, type-bound procedures, open arrays as pointer base types, ASSERT, more flexible string literals and nested comments.

Recent advances in compilation technology are included in *Ofront* to allow fine-grained interface checking rather than the more traditional per module interface checks. *Ofront* does not require recompilation of all clients of a module if the interface of a module changes. Only those clients which are actually affected by a change need to be recompiled. Since extending the interface of a module does not invalidate any clients, no recompilation is needed in this case.

Ofront provides fast translation of Oberon modules into C code. The additional *Ofront* step is therefore almost negligible when compared with C compilation and linking. Translation speed is more than 150,000 lines of Oberon code per minute on a 60 MHz HP 9000 Model 712, for example.

In order to support a wide range of C compilers, *Ofront* is parameterizable to support virtually all existing C compilers, be they ANSI or Kernighan/Ritchie (K&R) style compilers.

Ofront includes a highly compact and efficient run-time system that provides auxiliary routines for managing dynamic type information, exceptions, modules, commands, the Oberon heap, and an automatic garbage collector. Heap management and garbage collection are built upon C `malloc` and scale themselves to the actual memory requirements by means of an extensible heap. *Ofront*'s memory management is significantly more efficient than native `malloc`. For Sun/SPARC, for example, storage allocation is about a factor of 10 faster and the complete run-time system needs less than 10 KB of object code.

Ofront allows interfacing between Oberon and C or other foreign languages by means of in-lined code procedures. This provides for flexible parameter mapping and avoids any run-time overhead. Since *Ofront* programs are translated to C, they can be called from C or other foreign languages as well; thus interfacing actually works in both directions.

Ofront produces clean and human-readable C code by indentation of nested statements, a set of *Ofront*-related macros that define operations which are not directly available in C, and by avoidance of unnecessary parentheses. In many cases this allows the usage of C tools such as the debugger without getting lost in the generated code. The correspondence between the original Oberon code and the C code is always obvious.

Ofront follows widely used C programming conventions by producing two C files, a header and a body file. The header file contains the module interface to be used in `#include` control lines. Information hiding is preserved in the C header files; i.e., they contain only information about exported objects or record fields.

1.3 Intended Use

Ofront should be considered a complementary tool to already existing Oberon compilers which produce native machine code. It is intended to be used if at least one of the following properties is desired:

- Strong optimizations should be performed on the program.
- Oberon programs should be written for a computer which does not feature a native-code Oberon compiler (yet).

- Linking with existing C programs is essential.
- Stand-alone programs are desired.
- Shared object libraries should be generated.
- You know Oberon and have to learn C.

The translation of Oberon programs to C makes thousands of person years invested into aggressively optimizing C compilers available to the Oberon community. Oberon compilers that optimize equally well can only be expected when Oberon is as wide-spread as C.

Virtually all computers feature a descent C compiler; some machines still come with a C compiler bundled with the OS. The detour over C assures that portability of Oberon programs even to exotic machines is not an issue when selecting the programming language for a project.

The translation to C allows bidirectional interfacing with libraries written in C, i.e. calling C from Oberon and calling Oberon from C. This subject is not as trivial as might be expected, but with some experience and care, interfaces can be realized in a very short time.

Using the C compiler as a code generator assures that Oberon programs are translated into some standard object file format rather then into an Oberon-specific format, which requires a special environment to execute the programs. The standard object files can be linked with a standard linkage editor which produces stand-alone Oberon programs. The use of standard object files also supports interfacing with C libraries or other programming languages that follow an operating system’s standard object file format and calling conventions.

In addition to statically linked applications, many operating systems (e.g. SunOS, HPUX, AIX) support the use of shared object libraries. In this case object files can be shared between multiple applications, which reduces memory requirements and loading time. Using C as code generator enables the generation and use of shared object libraries for Oberon programs. Typically a set of modules (sometimes called a "subsystem") is linked into one shared object library. Since shared object libraries are usually mapped into memory by using demand paging, the loading time only depends on the amount of code accessed during startup of an application, not on the static size of a library.

More and more students are being educated with Oberon. Many of them have to use C in their industrial careers. *Ofront* can be immensely helpful to see the correspondence between Oberon programs and/or data types and their C counterparts. In effect, *Ofront* can be used as a sort of interactive electronic C teacher. This was also one of the reasons why readability of the generated code was one of *Ofront's* design goals.

2 Getting Started

Ofront comes in different flavors. For Unix platforms both an integrated environment and a command-line version is available. The integrated version runs inside ETH-Oberon and the command-line version runs as a stand-alone program that can be executed directly from a shell. For PCs and Macintoshes versions running as a subsystem of Oberon/F (TM) and as a separate environment are provided. These latter versions follow the respective platform's user interface conventions and provide online documentation for first-time users.

All versions have exactly the same features; in particular, they have the same set of options which we therefore describe only once in Section 2.1. *Ofront* is capable of generating code for different C compilers. Therefore it requires a file which parameterizes the translation process. This file is called *Ofront.par*. For details on parameterization and cross translation see Section 2.4.

The command-line version is provided for those users who prefer to use their traditional ASCII text editor (e.g., emacs or vi) or to embed Oberon in standard Unix programming tools such as shell scripts or make files. The integrated Unix versions allow using the standard Oberon text editors, which, although puristic at first glance, provide much more functionality than plain ASCII editors. The *Ofront* command-line version accepts source files either as plain ASCII texts, as Oberon V4 texts, or as Oberon System 3 texts.

2.1 Command-line Version

The command-line version of *Ofront* is represented by command ofront, which accepts an arbitrary number of input file parameters. Every file is expected to contain one Oberon module. Oberon source files typically (but not necessarily) have the file name extension .Mod. Options affecting the translation process may be specified immediately after the command name or after a file name. The former apply to all files, the latter only to the preceeding one; thus, the order in which file names and options are specified is important. Specifying an option twice nullifies the effect of the option. This might be used to override a global option for an individual file. The following EBNF grammar specifies *Ofront*'s command line syntax. Note that options must not contain whitespace.

```
$ command = "ofront" options {filename options}.
```

```
$ options = ["–" {option} ].
$ option = "m" | "s" | "e" | "i" | "r" | "x" | "a" | "p" | "t".
```

ofront performs full error analysis on the Oberon input modules and writes error and completion messages to the standard output device. No C code is generated if errors are detected. The exact meaning of the error numbers is listed in file OfrontErrors.Text and in the appendix.

ofront looks for its input files in the directories specified by the environment variable OBERON, which is expected to contain a colon-separated list of path names. The following example shows how to set the OBERON environment variable under the Unix C-shell. If the OBERON environment variable does not exist, files are looked up only in the current working directory. Files that contain a "/" character in their path name are always looked up relative to the current working directory and independent of the OBERON environment variable.

```
% setenv OBERON .:~:/usr/local/Oberon
```

The meaning of the individual options is defined as follows:

m *generate a main module (default off)*
This option signals *Ofront* that the module body should be translated into a C main function, which is the entry point of an application. Every application consists of exactly one main module. Modules which are intended to be included in a library should never be compiled with option m.

s *allow changing the symbol file (default off)*
The interface of an Oberon module is represented in a compact and efficient form in the module's symbol file (suffix .sym). Changing the symbol file of a module therefore means changing the interface of the module. Examples of such a change are to insert, rename, or delete an exported type, variable, or procedure. Those clients, which depend on the changed feature, have to be adapted to the new interface and recompiled. Note that, unlike earlier Modula-2 or Oberon-2 compilers, only those clients of the module that depend on the changed feature(s) need to be recompiled, not all modules which import the changed service module. The new fine-grained interface checking supports the evolution of software over time much better than its coarse-grained predecessor. To avoid unintended interface changes, this option is turned off by default.

e *allow extending the module interface (default off*)
This option is similar to s but restricts interface changes to extensions. For example, it is possible to export additional global variables or procedures if option e is specified. Renaming or deleting a procedure or variable is not allowed. To avoid unintended interface extensions, this option is turned off by default.

i *include header and body prefix files (default off)*
Specifying this option enables the inclusion of C code that is prepended to the generated header and body files. For a module M, the header and body prefix files are expected to be named M.h0 and M.c0 respectively. Non-existing prefix files are silently ignored.

r *check value ranges (default off)*
Specifying this option turns on value range checking such as checking if SHORT of a LONGINT variable is in the INTEGER value range. Since this option is not related to memory integrity, it is turned off by default.

x *check array indices (default on)*
Specifying this option turns off array index bounds checking. Since index checks are inlined and consist only of a single unsigned compare, they are very fast and it is normally not necessary to turn them off in order to get good performance. Furthermore, optimizers can remove index checks in many places without giving up security.

a *check assertions (default on)*
Specifying this option turns off run-time checking of ASSERT statements. Use this option only in carefully tested production code when utmost efficiency is required. An unchecked assertion is nothing but a comment.

p *pointer initialization (default on)*
Specifying this option turns off automatic pointer initialization. Note that Oberon does not specify the value of local pointer variables before they are assigned a value. Even with pointer initialization, it is not correct to make assumptions about the initial value of a pointer. In particular, it is not allowed to assume that they are set to NIL. Pointer initialization is a technique to ensure memory integrity even in case of erroneous programs and/or to detect uninitialized pointers as soon as possible.

t *check type guards (default on)*
Specifying this option turns off run-time type guard checking. Since type guard checks are very efficient anyway and undetected type guard failures can easily destroy memory integrity, we recommend using this option only in very rare cases such as low-level modules where every machine cycle counts.

Example

```
% ofront M1.Mod
M1.Mod  translating M1   298
```

The generated output contains the name of the file, the name of the module and the size in bytes of the generated C body file. In case of an error or if a warning is issued, the text position (not line number) and an error number are written to the standard output device. The meaning of the error number can be looked up in file OfrontErrors.Text or in the appendix.

```
% ofront –e M1.Mod M2.Mod M3.Mod –m
M1.Mod  translating M1   298
M2.Mod  translating M2  extended symbol file   340
M3.Mod  translating M3  main program   230
```

showdef The command showdef is provided to allow decoding *Ofront* symbol files. showdef xxx decodes the symbol file of module xxx and displays it in human-readable form on the standard output device.

Example

```
% showdef Args
DEFINITION Args;

 VAR
  argc–: LONGINT;
  argv–: LONGINT;

 PROCEDURE Get(pos: INTEGER; VAR val: ARRAY OF CHAR);
 PROCEDURE GetInt(pos: INTEGER; VAR val: LONGINT);
 PROCEDURE Pos(s: ARRAY OF CHAR): INTEGER;

END Args.
```

The showdef command corresponds to the Browser.Showdef command in the integrated version.

2.2 Integrated Version

The integrated version of *Ofront* is represented by command Ofront.Translate and accepts the same options and parameters as the command line version. Note that options are preceded by a "\" character on Unix platforms and that the "*" character refers to the star-marked text in Oberon. This text can be translated right from the editor without storing the text to a file first. The following EBNF grammar specifies the Ofront.Translate command.

```
$ command = "Ofront.Translate" options {("*" | filename) options } "~".
$ options = ["\" {option} ].
$ option = "s" | "e" | "m" | "i" | "a" | "p" | "x" | "t".
```

The input files are searched in exactly the same way as in the command-line version, i.e., using the environment variable OBERON. This environment variable is also used by the ETH-Oberon V4 system (activated by the command oberon), which (under Unix) runs as an X-client. As usual for X-clients, oberon supports the -d (display) and -g (geometry) command-line options and the DISPLAY environment variable. For more details about using the integrated version please refer to the oberon (1) manual page and the online documentation included in the distribution.

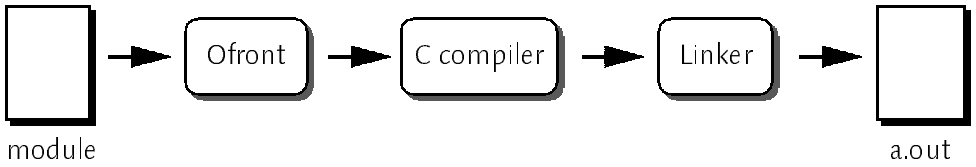
Examples

```
Ofront.Translate *\s ~
Ofront.Translate \er M1.Mod M2.Mod M3.Mod\m ~
```

2.3 Principles of Operation

Working with *Ofront* involves several steps to achieve a running application. The following gives an overview of this process. Chapter 3 goes into details of particular combinations of C compilers and operating systems.

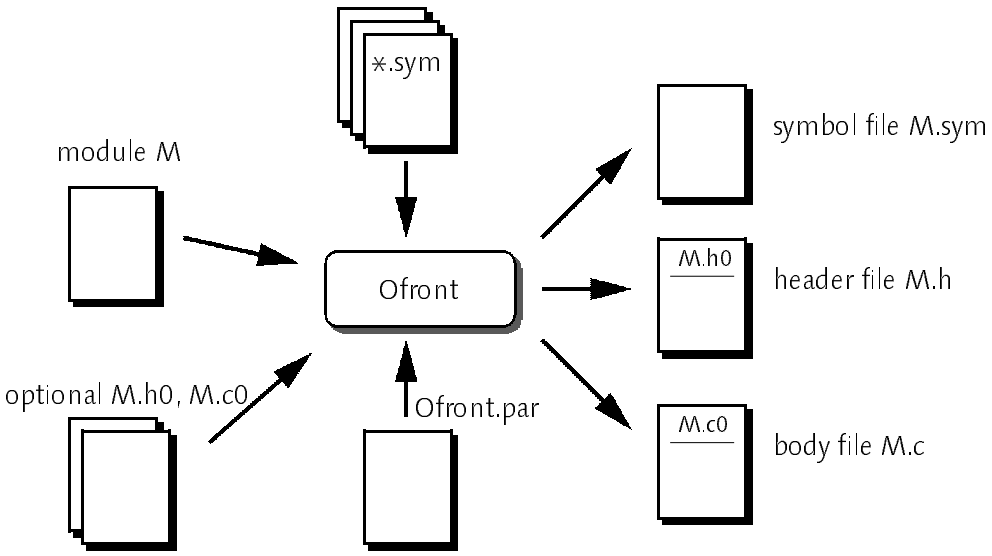
Figure 2.1 Sequence of processing steps



Ofront The first step is to run *Ofront*, which produces as its output the input to the C compiler. This process is shown in Figures 2.2 and 2.3 for a module M. In case of an error, no output files are written and existing files with the same names are preserved. In case of success, existing files with the same name are overwritten.

Ofront creates new files in the current working directory and looks up old files (e.g., symbol files) in all directories listed in the OBERON environment variable, which is expected to contain a list of path names delimited by colons.

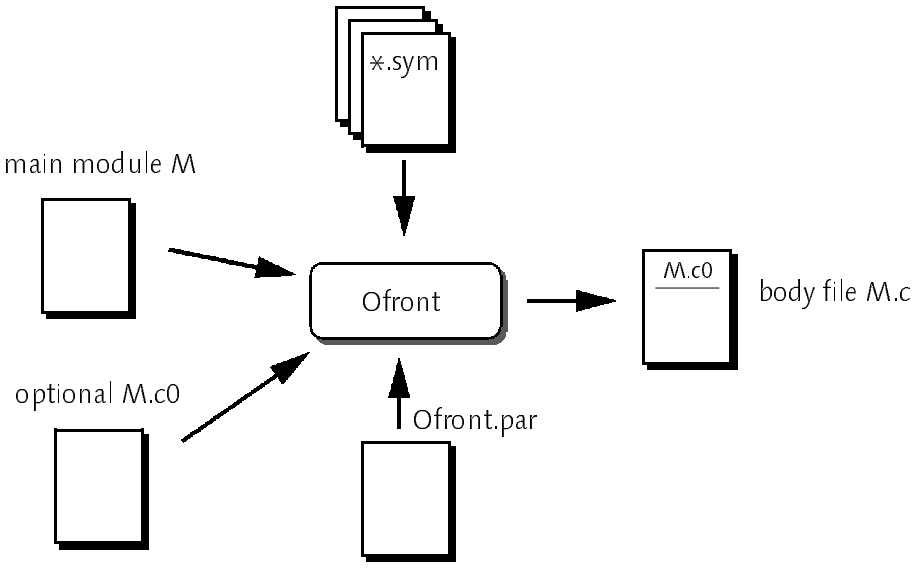
Figure 2.2 Translation of a normal module



In the case of translating a main module (by specifying option m), only a body file is generated. A main module cannot be imported by other modules since it is the distinguished root of a module hierarchy. Files M.sym and M.h are therefore deleted if they exist.

If option i is in effect, the files M.h0 and M.c0 would be used as a prefix of the output files M.h and M.c. Missing prefix files are treated as if they were empty.

Figure 2.3 Translation of a main module



C compiler The second step, C compilation, unavoidably needs some knowledge about a system's C compiler in order to fully exploit *Ofront*. One point is the optimization level that is desired for a given module. It is also possible to C-compile without optimizations but with debugging enabled. Selecting the appropriate debugging and/or optimization level is the first decision when C-compiling. The second decision is whether the module should be statically linked or put into a shared object library from where it can be linked dynamically. On a system featuring shared object libraries (sometimes also called dynamic link libraries), object files normally must be in a certain format, which is often called "position independent". This ensures that the generated object code can be mapped into the address space of a process at an arbitrary position. Most compilers must be instructed explicitly to generate position independent code. Due to the variety of the C compiler options, the *Ofront* and C steps are not integrated. However, the two steps (or three if you also consider linking) can be automated by writing appropriate shell scripts.

Linker Most systems allow putting object files into an archive or into a shared object library. Archives are used for static linking of applications, i.e. if the code of the archived modules should be copied into the executable of the application. Shared object libraries are used if the code of one or several modules should be shared among multiple applications. This leads to significantly reduced executables, but on the other hand to dependencies on the shared object library. The executable is no longer self-contained. Due to the reduced code size and the possibility of

creating truly extensible applications, shared object libraries are gaining importance nowadays while static archives are declining.

For shared object libraries, we distinguish two linking strategies: *dynamic* linking and *run-time* linking. Dynamic linking is equivalent to static linking except that some parts of an application reside outside the application's executable in a shared object library. As with static linking, all parts of the application must be known in advance. Run-time linking means that an arbitrary library which is not known in advance can be loaded at run time and thereby truly extends an application. Technically speaking, run-time linking is realized by providing a programmatic interface to the dynamic linker. Clearly, run-time linking is needed for Oberon in order to achieve the effect of dynamic module loading. Unfortunately, not all operating systems support run-time linking yet; most operating systems support dynamic linking, and all support static linking.

Subsystems

Typically, a group of modules that together provide an abstraction (e.g., a graphics editor) can be linked into a shared object library. In principle, every single module can be regarded as a shared object library, but this is not usually the granularity expected by an operating system's dynamic loader and might lead to inefficiencies. As an example, the libraries that contain all modules of the Oberon V4 system are provided as shared object libraries (e.g. libOberonV4.so for SunOS 5.x). For a complete Oberon system there must only be a simple main module which initializes some global data structures and starts the Oberon main event loop (cf. example module Configuration.Mod). The size of this main program is just a couple of lines and a few KB of object code if it is linked dynamically with the appropriate Oberon library.

When packing more than one module into a subsystem, the problem of retrieving such modules at run time arises. There must be a mapping from a module name to the package that contains this module. The simplest way of providing such a mapping is to follow an appropriate naming convention for all modules that together form a subsystem. All module names should start with a common prefix which identifies the subsystem. We propose to use the first transition from lower case to non-lower case characters in a module name as the end of this common prefix. If, for example, there is a package named Dialog that provides graphical user interface building blocks, all modules of the form DialogFrames, DialogBoxes, Dialog1, etc, would be recognized as belonging to this subsystem and can be loaded at runtime from the Dialog subsystem. Note that in a software system that might be

extended from different persons it is a good idea to use unique module name prefixes anyway in order to avoid name clashes. Chapter 7 gives a concrete example of a run-time linking strategy including source code.

On a platform that supports run-time linking, after the oberon application is started, the module list contains only those modules which are directly or indirectly imported by the main module or which are loaded explicitly during the initialization process. When activating a command, e.g. Edit.Open, module Edit will be loaded (mapped) as well and become a member of the module list. Thus dynamic loading of modules is preserved.

On a platform that only supports static or dynamic linking, all modules which are to be available to the application must be imported by the main module either directly or indirectly. After starting such an application, all modules are contained in the module list right from the beginning.

2.4 Cross Translation

Ofront allows cross-translation of Oberon modules to a variety of target systems by means of the parameter file Ofront.par. Parameter files for various C compilers are provided and named Ofront.xxx.par where xxx identifies the target platform. The parameter file for the Intel 960 architecture, for example, would be named Ofront.i960.par. A list of available parameter files is given in the appendix. If your C compiler is not listed, you have two possibilities:

- Use a prameterization file for a C compiler that has the same characteristics as yours. To generate code for a specific C compiler, simply rename the appropriate parameter file to Ofront.par.

- Generate a new parameter file. A C program ofrontparam.c is included, which, when compiled and executed, will output the characteristic attributes of the C compiler used to compile it. ofrontparam.c includes SYSTEM.h; thus you have to make sure that the C compiler uses the appropriate version of this include file.

```
% cc ofrontparam.c; a.out > Ofront.par; rm a.out
```

Note that for successful cross-translation all modules must be translated against the right symbol files, i.e., against symbol files that are generated while using the same parameter file. Otherwise the size and

alignment of various data types might be inconsistent.

If there is no precompiled run-time system provided for your target architecture, you will have to translate and cross-compile the module SYSTEM.Mod first. (If the source text of module SYSYEM is not included in your version of Ofront, contact your Ofront distributor.) Translation should be done using the –i option, since the body prefix file SYSTEM.c0 must be included. For maximum efficiency, run-time checks should be disabled for module SYSTEM. *Ofront* will neither produce a .sym nor a *.h* file when translating module SYSTEM. C compilation of SYSTEM.c should always be done using the highest optimization level available with a particular C compiler.

```
% ofront SYSTEM.Mod –iapx
% cc –O –c SYSTEM.c
```

For particular application domains, such as real time systems or multiprocessor environments, modifications of the standard run-time system might be necessary. Please note that it is not allowed to redistribute the original or modified SYSTEM module as source or object code without the prior written permission of SOFTWARE TEMPL. This limitation is intended to prevent incompatibilities between the original and possibly modified run-time system versions. It is of course allowed to link a modified SYSTEM object file statically with your application and distribute it.

3 C-Compilation and Linking

This chapter contains a description of C compilation and linking in selected combinations of C compilers and operating systems which might serve as prototypes for other combinations. The reader should be familiar with at least one C compiler and linker; i.e., this chapter is not an introduction to the field of compiling and linking programs for a particular platform. Most compilers and linkers provide an overwhelming number of arguments and options, some of them may be combined, others not. As a rule of thumb, however, most of these options can be ignored.

For every compiler/platform combination, an example is presented that shows how to compile and link the Oberon V4 module library as a shared library (if possible) and how to create a main executable program that is dynamically linked with the library. The starting point is that all library modules have been translated to C and are available in the current working directory. Module Configuration.Mod has not been translated yet and is intended to become the main executable program. Therefore this is the only module that will not be part of the generated library. On platforms that do not support run-time linking, the file Configuration.Max.Mod is used; it imports all modules which should be available in the executable.

hello world

The following program shows how the generated library libOberonV4 can be used to create a simple command-line program. Module Console is used to write output to the standard output device. The cc command requires specification of the referenced library, which is platform dependent in general. Options –L and –l are supported by most C-compilers, though.

```
MODULE hello;
IMPORT Console;
BEGIN Console.String("hello world"); Console.Ln
END hello.
```

```
% ofront hello.Mod –m; cc hello.c –L. –lOberonV4 –o hello
% hello
hello world
```

3.1 SunOS 4.x (SPARC)

SunOS 4.x does support dynamic linking and a limited form of run-time linking. The main restriction is that SunOS 4.x does not allow that shared object libraries which are loaded at runtime (by means of a dlopen call) depend on shared object libraries which are themselves loaded at runtime. Such cases can lead to loading shared libraries multiple times into main memory causing inconsistencies in a program due to multiple instances of global variables. The mentioned limitations are removed in SunOS 5.x (Solaris2). We deliberately refrain from using the run-time linking facilities of SunOS 4 but use dynamic linking to reduce the size of an application.

The following example shows how to create a shared library libOberonV4.so.1.0 and a dynamically linked executable named oberon.

```
% cc -O -PIC *.c -Qproduce .o
```

C compilation for shared object libraries requires option -pic or -PIC (position independent code). The difference is that -pic only works for small libraries. If debugging should be enabled rather than optimizations, specify option -g instead of -O. Option -Qproduce .o prevents invocation of the linker and leaves the output of the compiler in object files with suffix .o.

```
% ld *.o -lm -o libOberonV4.so.1.0
```

The system's linkage editor ld combines the *.o files to a shared object library named libOberonV4.so.1.0. External references that refer to statically bound objects (e.g., the functions of libm) must be resolved in the ld command; references to shared libraries (e.g., libX11) can be resolved when linking the main executable. The C library (/usr/lib/libc.so.x.y) is imported automatically and need not be specified explicitly.

C compilation for static archives or statically linked applications is normally done without option -PIC; although it would also work with -PIC, but the code is slightly faster without it.

A dynamically linked executable can be created by the cc command simply by replacing the -Qproduce .o option with a specification of the generated object file (-o oberon) as in

```
% ofront Configuration.Max.Mod -m
% cc Configuration.c -L. -lOberonV4 -lm -lX11 -o oberon
```

In this case, the executable is dynamically linked if shared versions of the referenced libraries are available. Option -L is used to specify additional directories that contain shared libraries or static archives. Option -l specifies libraries or archives to be used to resolve external references.

The environment variable LD_LIBRARY_PATH may be used to specify a colon-separated list of path names to be used by the dynamic linker to search for libraries at both linking and loading time. A typical setting for LD_LIBRARY_PATH would be

.:/usr/local/Oberon/lib:/usr/openwin/lib.

If object files (or static archives) are specified explicitly as in

```
% cc Configuration.c *.o -lm -lX11 -o fatoberon
```

the object files *.o would be statically linked into the executable fatoberon. Use options -Bstatic and -Bdynamic for controlling the static or dynamic linking of modules in SunOS 4.

3.2 SunOS 5.x (SPARC)

SunOS 5.x (Solaris2) supports both dynamic linking and run-time linking. There is no need to explicitly instruct the compiler to generate position-independent code since this is the default mode. The system's linkage editor (ld) may be used to combine object files into shared object libraries as shown in the following example.

```
% cc -c -fast *.c
% ld -G *.o -o libOberonV4.so
% ofront Configuration.Mod -m
% cc Configuration.c -L. -lOberonV4 -lm -lX11 -ldl -o oberon
```

The first step runs the C compiler for all .c files but supresses linking (option -c). Optimizations for execution speed are turned on by option -fast. The second step invokes the linkage editor. Option -G specifies generation of a shared object library. The last step compiles the main executable and links it dynamically to libOberonV4. Additional libraries and directories can be specified with options -l and -L as usual. The environment variable LD_LIBRARY_PATH may be used to specify a colon separated list of path names to be used by the dynamic linker to search for libraries at both linking and execution time. In addition, a run path

may be written into the executable (use option -R or environment variable LD_RUN_PATH). The run path provides a default path list if LD_LIBRARY_PATH is not present at run time. A typical setting for LD_LIBRARY_PATH (or the -R option) would be
.:/opt/Oberon/lib:/usr/openwin/lib.

ocl In order to extend for example the Oberon V4 system by an additional module at runtime, the script ocl as shown below is provided for Solaris2. It translates an Oberon module to C, compiles it, and links it into a shared library consisting of exactly one Oberon module. The module may then be loaded at runtime at the first time a command of the module is to be executed. Note that you have to link additional libraries using the -l option in the ld command in case that the module references those libraries.

```
#!/bin/csh
#
# compile and link an Oberon module
#
# SYNOPSIS
#   ocl moduleName [ofrontOption [ccOption]]
#
# use the single character "-" to skip ofrontOption
# example: ocl hello - -g

# translate .Mod to .c
ofront $1.Mod $2

# compile .c to .o
cc -c $3 $1.c

# link .o into lib$1.so; use option -l to link appropriate libraries
ld -G -L. -lOberonV4 $1.o -o lib$1.so

# remove unnecessary files and show result
rm $1.c $1.o; ls -l lib$1.so
```

3.3 DEC Ultrix (MIPS)

DEC/Ultrix (V4.2A) only supports static linking. Object file archives can be created and maintained by means of the ar command. The

linkage editor ld combines object files and archives to self-contained executables. In order to generate an executable command oberon, it suffices to use the compile-and-link mode of the cc command as shown below:

```
% ofront Configuration.Max.Mod -m
% cc *.c -lm -lX11 -o oberon
```

To create a static archive named libOberonV4.a, the archiving tool ar may be used. The strip command strips off symbolic information contained in the executable, making it slightly smaller.

```
% ar qs libOberonV4.a *.o
% cc Configuration.c -L. -lOberonV4 -lm -lX11 -o oberon
% strip oberon
```

3.4 HP-UX (PA-RISC)

HP-UX allows both dynamic linking and run-time linking. In order to use dynamic or run-time linking, the C compiler (cc) must be instructed explicitly to generate position independent code (option +z or +Z for very large libraries). The linker (ld) must be instructed to generate a shared library instead of a normal executable (option -b). The following commands show how to create libOberonV4.sl and the executable oberon under HP-UX:

```
% cc -Aa +z -O -c *.c
% ld -b -z *.o -lc -lm -L/usr/lib/X11R5 -lX11 -o libOberonV4.sl
% ofront Configuration.Mod -m
% cc -Aa -Wl,+s Configuration.c -L. -lOberonV4 -ldld -o oberon
```

The first step compiles all .c files found in the working directory. Option -Aa specifies ANSI semantics to be used in cases where ANSI C differs from older HP C compilers. +z specifies generation of position-independent code. Option -O requires performing level-two optimizations (same as +O2). Option -c avoids invoking the linker after C compilation.

The second step generates a shared library (option -b) and enables NIL-checking (option -z) to be done by the hardware. Library inclusion is specified by means of the -L and -l options. -o specifies the name of the generated output file.

The third step compiles the main executable and generates an executable named oberon, which is dynamically linked with library libOberonV4.sl. It is important to include library libdld.sl (by means of option –ldld) in order to enable run-time linking. Option –Wl,+s indicates passing option +s to the linker. +s means that the environment variable SHLIB_PATH (a colon-separated list of path names) should be consulted for dynamic library loading.

ocl In order to extend for example the Oberon V4 system by an additional module at runtime, the script ocl as shown below is provided. It translates an Oberon module to C, compiles it, and links it into a shared library consisting of exactly one Oberon module. The module may then be loaded the first time a command of the module is to be executed. Note that you have to link additional libraries using the –l option in the ld command in case that the module references those libraries.

```
#!/bin/csh
#
# compile and link an Oberon module
#
# SYNOPSIS
#   ocl moduleName [ofrontOption [ccOption]]
#
# use the single character "–" to skip ofrontOption
# example: ocl hello – –g

# translate .Mod to .c
ofront $1.Mod $2

# compile .c to .o
cc –Aa –c +z $3  $1.c

# link .o into lib$1.sl; use option –l to link appropriate libraries
ld –b –z –L. –lOberonV4 $1.o –o lib$1.sl

# remove unnecessary files and show result
rm $1.c $1.o; ls –l lib$1.sl
```

3.5 IBM AIX (RS/6000)

AIX supports dynamic linking but not run-time linking. Note that the load system call is not sufficient for run-time linking. A separate library (libdl.a) is available from a third party vendor (HELIOS Software GmbH, Germany, e-mail: jum@helios.de) that provides run-time linking facilities similar to the one found in SunOS. The AIX linkage editor may either be invoked by the ld or the cc command.

```
% cc -c -O *.c
% cc -o libOberonV4.o -bM:SRE -bE:V4.exp -lX11 -lm -lc -L. -ldl \
-e _nostart *.o
% cc Configuration.c libOberonV4.o
```

The first step compiles all .c files with optimizations being switched on (-O) and supresses linking (option -c). There is no need to explicitly instruct the compiler to generate position independent code. The second step links all .o files to a shared library libOberonV4.o. Option -bM:SRE sets the shared object flag in the generated object file. Option -bE:V4.exp specifies file V4.exp to be used as an export list. Export lists start with the #! sign and list all names of exported objects. Options -lX11 -lm -lc specify that three additional libraries (libX11, libm, and libc) should be linked. Option -e _nostart specifies that the shared object does not have an entry point that is to be given control after loading it.

A separate tool (genexp) to generate the export lists for a set of modules is provided with *Ofront* for AIX. genexp takes an arbitrary number of module names (possibly including a filename extension) as input and writes the export list to the standard output device. For example

```
% genexp *.sym > V4.exp
```

ocl In order to extend for example the Oberon V4 system by an additional module at runtime, the script ocl as shown below is provided. It translates an Oberon module to C, compiles it, and links it into a shared library consisting of exactly one Oberon module. The module may then be loaded at runtime the first time a command of the module is to be executed. Note that you have to link additional libraries using the -o option in the second cc command in case that the module references those libraries.

```
#!/bin/csh
#
# compile and link an Oberon module
#
# SYNOPSIS
#   ocl moduleName [ofrontOption [ccOption]]
#
# use the single character "-" to skip ofrontOption
# example: ocl hello - -g

# translate .Mod to .c
ofront $1.Mod $2

# generate an export file
genexp $1 > $1.exp

# compile .c to .o
cc -c $3 $1.c

# compile .o to lib*.o; use option -o to link to appropriate libraries
cc $1.o -o lib$1.o -bM:SRE -bE:$1.exp -e _nostart libOberonV4.o

# remove unnecessary files and show result
rm $1.c $1.o $1.exp; ls -l lib$1.o
```

3.6 SGI IRIX 5 (MIPS)

IRIX 5.3 supports dynamic linking and a limited form of run-time linking. The programmatic interface to the runtime linker and the limitations are exactly the same as in SunOS 4.x. The main restriction is that IRIX 5.3 does not allow shared object libraries which are loaded at runtime (by means of a dlopen call) to depend on shared object libraries which are themselves loaded at runtime. Such cases can lead to loading shared libraries multiple times into main memory causing inconsistencies in a program due to multiple instances of global variables. We deliberately refrain from using the run-time linking facilities of IRIX 5.3 but show how to use dynamic linking to reduce the size of an application.

```
% cc -O -c *.c
```

C compilation for shared object libraries does not require a flag for position independent code since this is set implicitly. Option -O specifies level 2 optimizations and -c suppresses linking after C compilation.

```
% ld *.o -lc -lm -lX11 -o libOberonV4.so
```

The system's linkage editor ld combines the *.o files to a dynamically linked shared object library named libOberonV4.so. Additional libraries are referenced by means of the -l option. A dynamically linked executable can be created by the cc command as shown below.

```
% ofront Configuration.Max.Mod -m
% cc Configuration.c -L. -lOberonV4 -o oberon
```

At runtime, the environment variable LD_LIBRARY_PATH, a colon-separated list of path names, is used to specify the directories in which the dynamic linker looks for shared object libraries.

3.7 Linux (i386)

Linux (Yggdrasil, Fall 1994 distribution) supports dynamic linking similar to SunOS 4.x but with even more stringent limitations. All shared object libraries must be assigned a world-wide unique address range in order to avoid conflicts with other shared libraries. In addition, special tools are needed to generate shared libraries. The situation is expected to change with the introduction of the ELF object file format for Linux and version 2.7.0 of gcc, which supports generation of position independent code. So far, we deliberately refrain from using shared object libraries for Linux and show how to produce a static archive and a traditionally linked application only. Note that existing shared libraries, such as libc or libX11 will nevertheless be linked dynamically.

```
% cc -O -c *.c
% ar libOberonV4.a *.o
% ranlib libOberonV4.a
% cc Configuration.c -L. -lOberonV4 -lm -lX11 -o oberon
% strip oberon
```

4 Module SYSTEM

The pseudo module SYSTEM plays a dual role in *Ofront*. First, it serves as an escape mechanism to unsafe and system-dependent language constructs (the static role) and second, it serves as the container of run-time routines (the dynamic role). The following sections explain both roles of module SYSTEM together with a mechanism to interface Oberon with C or other foreign languages.

4.1 The Static Role

Ofront's SYSTEM module is identical to that described in "Programming in Oberon" (see Ch. 1) except that it does not allow direct access to registers or condition codes. If these features are desired in a program, external or in-line expanded assembly language routines have to be used. For interfacing with foreign language procedures, see Section 4.2. The following definition describes all objects exported by module SYSTEM. Note that type Any stands for an arbitrary type, type Int stands for an arbitrary integer type, and type Scalar for an arbitrary unstructured type.

```
DEFINITION SYSTEM;

 TYPE
  BYTE = Octet;
  PTR = POINTER TO Any;

 PROCEDURE ADR (x: Any): LONGINT;
 PROCEDURE BIT (adr, n: LONGINT): BOOLEAN;
 PROCEDURE GET (adr: LONGINT; VAR x: Scalar);
 PROCEDURE LSH (i: Int; n: LONGINT): Int;
 PROCEDURE MOVE (sadr, dadr, n: LONGINT);
 PROCEDURE NEW (VAR p: PTR; n: LONGINT);
 PROCEDURE PUT (adr: LONGINT; x: Scalar);
 PROCEDURE ROT (i: Int; n: LONGINT): Int;
 PROCEDURE VAL (T: Type; x: Any): T;

END SYSTEM.
```

In addition to these exported objects, import of module SYSTEM enables specification of various flags for types or procedures. A type flag

always follows the first keyword that is used to construct the type and consists of an integer constant enclosed in brackets, as in the following:

```
TYPE
  P = POINTER [1] TO PDesc;

PROCEDURE WriteString(s: ARRAY [1] OF CHAR);
```

The meaning of type flags is defined in Table 4.1.

Table 4.1 Type flags

type	flag	meaning
POINTER	1	untraced pointer, implied by RECORD [1]
ARRAY	1	do not copy value array parameters
RECORD	1	untagged record

For procedures, *Ofront* allows a "-" sign after the keyword PROCEDURE in a procedure declaration to indicate that this procedure is an in-lined C code sequence. The in-lined code is written in quotation marks after the procedure heading as in the following example:

```
PROCEDURE -malloc(size: LONGINT): LONGINT
  "((LONGINT)malloc(size))";
```

Ofront translates such procedures into macro definitions which are subject to C preprocessing.

```
#define   Mymodule_malloc(size)   ((LONGINT)malloc(size))
```

Obviously, this mechanism provides a way to interface Oberon with foreign languages such as C or assembly language, as explained in more detail in Section 4.2.

4.2 Interfacing with C

In order to allow reuse of existing libraries written in C or other foreign languages (including assembly language), *Ofront* provides a mechanism to interface with such languages. The requirements on this mechanism are at least that it should not introduce unjustified run-time overhead and that it should allow a flexible way of mapping Oberon to C parameters, which are not always identical. Realistically, it cannot be expected that mixing different languages is as simple as staying within

only one language. The general rule is that if you want to be 100% compatible with C, then you have to use C. Nevertheless, *Ofront*'s way of interfacing with C allows writing interfaces with very little programming effort.

The basic idea is to use in-lined code procedures implemented as C macro definitions as shown in Section 4.1. This mechanism has been successfully used to connect the X11 library, the C library and mathematical functions to Oberon.

Please observe the following pitfalls in order to succeed, and note that for inexperienced users it is a good idea to look at the generated macro or even at the preprocessed C code before trusting a code procedure. An important rule for writing a C macro is to put arguments in parentheses to avoid semantic changes due to the application of precedence rules after macro expansion has taken place.

4.2.1 Name Mapping

Somehow Oberon names must be mapped to C names used in the in-lined code procedures. Since code procedures translate to macros, the C macro preprocessor can be used to perform this name mapping on all parameters of the macro. Thus it is possible to use Oberon names inside the code procedures as far as parameters of the procedure are concerned. The following gives a nontrivial example:

```
PROCEDURE –externalFunction(if: BOOLEAN)    "externalFunction(if)"
```

Note that "if" is not a valid parameter name in C but a reserved keyword. *Ofront* translates the parameter to if_ and passes this symbol to the macro activation. The preprocessor then substitutes the if inside the code procedure with if_ in the in-lined C code and everything works as expected. Obviously the name if must not be used as the C keyword if inside the code procedure.

For names of objects not passed as parameters, please look in Section 4.2.6 and in the generated C code. The rule for global names is that they are always prefixed by the module name followed by an underscore.

4.2.2 Pointers

Dynamic data structures in Oberon and C differ in the way storage is released. Oberon uses automatic garbage collection, whereas C uses

explicit release of storage blocks. To avoid corrupting Oberon's or C's memory management, never assign a C pointer to an Oberon pointer unless the pointer points to a valid Oberon heap object or the Oberon pointer is a local variable. Assign an Oberon pointer to a C pointer only if you are sure that the C program does not release the storage block.

Note also the possibility of using pointer types with type flag [1] (cf. Section 4.1) for specifying pointers that are not traced by Oberon's garbage collector. These pointers cannot be used to anchor an Oberon heap object but can refer to an external storage block which is subject to explicit deallocation as it is the case in C.

4.2.3 Parameter Substitution inside Strings

The C preprocessor only works on whole tokens, and a string literal is considered a token. Thus macro substitution normally does not take place inside a string. There is, however, one noteworthy exception to this rule. Some pre-ANSI C preprocessors do perform parameter substitution in strings that are contained in a macro definition. For the not so rare case of calling the printf function with a string as argument, for example, one should always check that no macro parameter occurs as a token within the string. The following example is erroneous since the s inside the string literal might be substituted with the actual argument s.

```
PROCEDURE –error(s: ARRAY OF CHAR)    'printf("error: %s", s)';
```

A formulation that works for all compilers would be

```
PROCEDURE –error(x: ARRAY OF CHAR)    'printf("error: %s", x)';
```

4.2.4 Exporting Code Procedures

When exporting a code procedure, note that this might easily lead to name clashes at the point where the code procedure is used. The following example shows a problematic situation:

```
PROCEDURE P;
  VAR sin: LONGREAL;
BEGIN
  sin := Math.sin(x);
```

```
  ...
END P;
```

If Math.sin is an exported code procedure defined as

```
PROCEDURE -sin*(x: REAL): REAL    "sin(x)";
```

then preprocessing sin := Math.sin(x) would result in sin := sin(x), which clearly uses the name sin ambiguously. The quintessence is that code procedures should only be exported if the names involved are very unlikely to produce a name clash or if efficiency is of highest priority. Otherwise a wrapper procedure should be exported that passes the parameters to the code procedure as shown below:

```
PROCEDURE -Sin(x: REAL): REAL "sin(x)";

PROCEDURE sin*(x: REAL): REAL;
BEGIN RETURN Sin(x)
END sin;
```

The careful reader might have noticed that the above example would not work correctly were there not additional provisions to get the return type of the external function sin right. Section 4.2.5 deals with this problem.

4.2.5 Return Types and Includes

By definition, the return type of a C function that has no prototype in the current scope is int, which is incorrect in the above example (sin) since the return type of sin is float or double. We have to declare a function prototype that provides the correct return type. As an alternative, we could also include a header file which contains such a declaration.

Ofront allows specification of special files that are included at the beginning of the generated header and body files. Such files are called *prefix files* since they prefix the generated output files. The prefix files of a module M are expected to be called M.h0 and M.c0 for the header and the body, respectively. Inclusion of prefix files must be enabled explicitly by specifying option i.

Prefix files can be of particular importance if combined with conditional inclusion (#ifdef) since they allow keeping an Oberon

module portable even if it depends, for example, on particular definitions of the underlying Unix variant.

Example To get the return type of function Math.sin right, module Math must be translated with option i and a file Math.c0 must be provided that contains either an include control line for an appropriate header file (math.h) or that contains the extern declarations directly. The second form has the advantage that "name space pollution" is reduced to a small number of explicitly specified identifiers; it has the disadvantage of possible inconsistencies with math.h and it requires additional typing if more than one function is involved.

```
% Ofront Math.Mod –i

Math.c0:
#include <math.h>
```

or

```
Math.c0:
extern double sin();
```

Since it is sometimes inconvenient to provide an additional file for only one or two declarations, *Ofront* also allows specification of control lines and extern declarations directly in the Oberon source text. This is done by means of looking at the contents of a code procedure and translating those that start with # or extern directly into the corresponding control line or extern declaration. Note that such code procedures are useful only as declarations; they should never be called.

The code procedure

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

would be translated into the include control line

```
#include <math.h>
```

The code procedure

```
PROCEDURE –externsin()    "extern double sin();";
```

would be translated into the declaration

```
extern double sin();
```

It is highly recommended to use these facilities only in very few low-level modules (if at all) since they are inherently unportable and potentially unsafe. They also lead to name space pollution, i.e. they can produce name clashes. If code procedures are exported or if header prefix files are used, this may even affect client modules.

4.2.6 Debugging

Due to the translation of Oberon programs to C programs, any C debugging tool including core dump analyzers or fancy run-time debuggers can be used to inspect Oberon programs. However, debugging happens on the level of the generated C code, not at the Oberon language level. This fact is addressed by one of *Ofront*'s design goals, viz. to generate human readable C code. Provided some basic knowledge of *C*, it is always obvious which Oberon statement corresponds to which C statement. The following name mappings should be kept in mind when accessing Oberon objects from within a C debugging tool:

- The name of a global variable or global procedure is prefixed by the module name followed by an underscore.
- Constructs which have no direct counterpart in C such as NEW or COPY are expressed by macros or functions with the same name prefixed with __ (double underscore). These macros are defined in file SYSTEM.h.
- The names of predefined Oberon types are unchanged.
- The names of local variables and parameters are unchanged except for those which conflict with C keywords. These are postfixed with an underscore.
- Local types, local procedures and anonymous types receive unique names through appending serial numbers.

Examples

Oberon	C	meaning
Args.Get	Args_Get	global procedure
Args.argv	Args_argv	global variable
INTEGER	INTEGER	predefined type
i, j, k, if	i, j, k, if_	local variables
p, q, default	p, q, default_	value parameters

	*p, *q, *default_	VAR parameters
s	s, s__len	an array parameter s with its length in s__len
r	*r, r__typ	a VAR–parameter record r with dynamic type r__typ
s[i]	s[i]	array element with index i
r.f	r.f	record field f
	(*r).f	record field f of VAR parameter r
p↑	*p	dereferenced pointer
p.f, p↑.f	p–>f	field f of record p↑
NEW(p)	__NEW(p, T)	allocate variable p↑ of type T
M.P(x)	M_P(x)	call procedure M.P
o.P(x)	__M_T0_P(o, x)	macro to call type bound procedure P
o.P↑(x)	M_T0_P(o, x)	super call, statically bound

4.3 The Dynamic Role

Module SYSTEM plays a second role in *Ofront*. It acts as a run-time system for applications generated by *Ofront*. All run-time routines are contained in an object file called SYSTEM.o and all exported names in this file are prefixed with "SYSTEM_" in order to guarantee globally unique names. The contents of SYSTEM.o is not important for using *Ofront*. However, it is important that SYSTEM.o be linked either statically or dynamically to every application generated with *Ofront*. Due to the compactness of *Ofront*'s run-time system, this increases the size of statically linked applications by only about 10 KB of object code.

Run-time routines are provided for operations not directly available in C. Among them are: heap management, automatic garbage collection, a finalization registry, a registry for modules, commands and types, and primitives for exception handling (cf. Chapters 6, 7, 8).

5 Module Args

Module Args provides access to a program's command line arguments and environment variables.

```
DEFINITION Args;
 VAR argc–, argv–: LONGINT;
 PROCEDURE Get(n: INTEGER; VAR val: ARRAY OF CHAR);
 PROCEDURE GetInt(n: INTEGER; VAR val: LONGINT);
 PROCEDURE Pos(s: ARRAY OF CHAR): INTEGER;
 PROCEDURE GetEnv(var: ARRAY OF CHAR; VAR val: ARRAY OF CHAR);
END Args.
```

argc and argv provide direct access to the argument count and argument vector. Note that argc is at least 1 since by convention the first argument is the name by which the program was invoked. argv is defined as the C pointer `char *argv[];` i.e., it refers to an array of character pointers. Every character array is terminated by a null character.

Get(n, val) returns the nth argument as string val. val remains unchanged if the argument does not exist. Get(0, val) returns the name by which the program was invoked. The argument is silently truncated to the length of val.

GetInt(n, val) returns the nth argument as integer val. val remains unchanged if the argument does not exist or if it is not an integer number.

Pos(s) searches for argument s and returns its position if it exists; otherwise it returns argc. Pos is useful for looking up a particular option as shown below.

GetEnv(var, val) returns the value of environment variable var if it exists; otherwise val remains unchanged. val is silently truncated in case of overflow.

Example The following statement sequence looks for a display argument which is specified either by the environment variable DISPLAY or as the command line argument following –d or –display. The string "unix:0" is used as a default value.

```
DISPLAY := "unix:0";
Args.GetEnv("DISPLAY", DISPLAY);
Args.Get(Args.Pos("-d") + 1, DISPLAY);
Args.Get(Args.Pos("-display") + 1, DISPLAY);
```

6 Exception Handling

Oberon does not define exception handling in the language itself since this is highly platform- and/or application-specific. If appropriate libraries are used for developing Oberon programs with *Ofront*, the low level details of exception handling should be hidden from the programmer. Only if such libraries are not available or if you are going to develop such a library, the following is relevant.

Halt

In order to implement a particular exception handling mechanism, module SYSTEM provides a hook into a simple trap handling framework and two variables which hold additional information. Whenever an explicit run-time check fails or if HALT is called from a program, procedure SYSTEM_HALT gets called. Note that *Ofront* implements run-time checks simply by calling HALT with a negative parameter. In turn, this triggers an upcall of the procedure installed in SYSTEM_Halt, which represents a customizable trap dispatcher. If this procedure returns or if there is no such procedure installed, the process is exited with a call to exit(n) where n is the HALT parameter.

void (*SYSTEM-Halt)();	hook for a trap dispatcher
LONGINT SYSTEM_halt;	holds the value of x in HALT(x)
LONGINT SYSTEM_assert;	holds the value of x in ASSERT(cond, x)

On Unix platforms, for example, exception handling essentially means signal handling since an exception is communicated to a process by sending a signal. Unix programs may use this mechanism and install a SYSTEM_Halt procedure that turns the call into a signal 4 (illegal instruction) sent to the process. By installing a Unix signal handler and examining the two variables SYSTEM_halt and SYSTEM_assert listed below, exception handling can be realized. The advantage of such a solution in the realm of Unix systems is that exceptions which are not detected by explicit tests but generated directly by the CPU (e.g., a zero divide) or sent by another process (e.g. an interrupt signal) also result in sending a signal. In any case, if the procedure installed in SYSTEM_Halt returns, the process is exited with a call to exit(n) where n is the HALT parameter.

lock

Two additional variables allow to protect regions that are not reentrant from keyboard interrupts (e.g. by typing Ctrl-C or a Break character on the controlling terminal or by sending a signal 2 to a Unix process). Non-reentrant procedures can for example be found in the X11 library. Calls of functions of this library can easily kill the calling process if a

previous display operation has been interrupted and left the connection to the X-server in an inconsistent state. Other examples of non-reentrant procedures are *Ofront*'s heap management and garbage collection procedures since they work _ by definition _ on a global data structure.

```
LONGINT SYSTEM_lock;
BOOLEAN SYSTEM_interrupted;
```

The variable SYSTEM_lock represents a lock that is incremented whenever a critical region is entered and decremented when it is left. If a keyboard interrupt happens to occur and SYSTEM_lock is greater than zero, only the boolean flag SYSTEM_interrupted must be set to TRUE, no other action that possibly leads to reentering the critical region should occur. When the critical region is left and SYSTEM_lock decremented to zero, this flag might be checked to see if an interrupt is "pending" and if so, __HALT(–9) can be used to issue a "deferred" interrupt signal. It is important that SYSTEM_interrupted is set to FALSE (either before posting the delayed interrupt signal or inside the signal handler) in order to prevent unintended postings of deferred interrupt signals at the end of other critical regions.

Example The following example shows an exception handler for the ETH Oberon system. The call Kernel.siglongjmp(Kernel.trapEnv, 1) at the end of procedure Trap transfers control to Oberon's main event loop, provided that an appropriate context has been stored in variable Kernel.trapEnv. The mechanism used for saving the execution context and transferring control across procedure calls is Unix's sigsetjmp/siglongjmp. The example involves four modules: one that implements the signal handler (System), one that sets up the environment that is to be restored after handling a signal (Oberon), one that shows how to use the locking mechanism in order to safely access non-reentrant procedures (Display) and one that provides the low level facilities (Kernel).

```
MODULE System;
...
VAR trapLevel: INTEGER;

PROCEDURE –signal(sig: LONGINT; func: Unix.SignalHandler)
   "signal(sig, func)";
PROCEDURE –halt(): LONGINT "SYSTEM_halt";
PROCEDURE –assert(): LONGINT "SYSTEM_assert";
PROCEDURE –lock(): LONGINT "SYSTEM_lock";
```

```
PROCEDURE -resetHalt() "SYSTEM_halt = -128";
PROCEDURE -setIntd(v: BOOLEAN) "SYSTEM_interrupted = v";
PROCEDURE -FinalizeAll() "SYSTEM_FINALL()";

PROCEDURE Trap(sig, code: LONGINT; scp: Unix.SigCtxPtr);
BEGIN
  signal(sig, Trap);
  IF trapLevel = 0 THEN
    trapLevel := 1;
    CASE sig OF
    | 2:
        IF lock() > 0 THEN (* delay interrupt until lock = 0 *)
          setIntd(TRUE); trapLevel := 0; RETURN
        ELSE Out.String("INTERRUPT")
        END
    | 3:
        FinalizeAll(); Unix.Exit(0)
    | 4:
        CASE halt() OF
        |  0: (* silent halt *)
            resetHalt(); trapLevel := 0;
            Kernel.siglongjmp(Kernel.trapEnv, 1)
        | -1: Out.String("ASSERT(");
            Out.Int(assert(), 1); Out.String(") FAILED")
        | -2: Out.String("INDEX OUT OF RANGE")
        | -3: Out.String("FUNCTION WITHOUT RETURN")
        | -4: Out.String("INVALID CASE")
        | -5: Out.String("TYPE GUARD FAILED")
        | -6: Out.String("IMPLICIT TYPE GUARD FAILED")
        | -7: Out.String("WITH GUARD FAILED")
        | -8: Out.String("VALUE OUT OF RANGE")
        | -9: setIntd(FALSE); Out.String("DELAYED INTERRUPT")
        ELSE
          IF (halt() > 0) & (halt() < 256) THEN
            Out.String("HALT("); Out.Int(halt(), 1); Out.Char(")")
          ELSE Out.String("ILLEGAL INSTRUCTION")
          END
        END ;
        resetHalt()
    | 8:
        Out.String("ARITHMETIC EXCEPTION,  code = ");
        Out.Int(code, 1)
```

```
      |10:
          Out.String("BUS ERROR")
      |11:
          Out.String("SEGMENTATION VIOLATION")
      |13:
          Out.String("UNCONNECTED PIPE")
      ELSE
          Out.String("SIGNAL "); Out.Int(sig, 0)
      END ;
      Out.Ln
   END ;
   trapLevel := 0;
   Kernel.siglongjmp(Kernel.trapEnv, 1)
END Trap;
...
BEGIN
   trapLevel := 0;
   signal(2, Trap);    (* keyboard interrupt *)
   signal(3, Trap);    (* quit *)
   signal(4, Trap);    (* illegal instruction *)
   signal(8, Trap);    (* arithmetic error *)
   signal(10, Trap);    (* bus error *)
   signal(11, Trap);    (* segmentation violation *)
   signal(13, Trap)    (* unconnected pipe *)
END System.


MODULE Oberon;
...
PROCEDURE Loop*;
BEGIN
   res := Kernel.sigsetjmp(Kernel.trapEnv, 1);
   LOOP
      the Oberon main event loop, which is to be reentered upon a trap
   END
END Loop;

...
END Oberon.
```

```
MODULE Display;
...
PROCEDURE -Lock() "SYSTEM_lock++";
PROCEDURE -Unlock() "SYSTEM_lock--; if (SYSTEM_interrupted &&
SYSTEM_lock == 0) --HALT(-9)";

PROCEDURE CopyBlock*(SX, SY, W, H, DX, DY, mode: INTEGER);
BEGIN
   Lock();
   call of non-reentrant X-library routines
   Unlock()
END CopyBlock
...
END Display.


MODULE Kernel;
   ...

VAR trapEnv*: Unix.JmpBuf;

PROCEDURE -sigsetjmp*
   (VAR env: Unix.JmpBuf; savemask: LONGINT): LONGINT
      "sigsetjmp(env, savemask)";

PROCEDURE -siglongjmp*(VAR env:Unix. JmpBuf; val: LONGINT)
      "siglongjmp(env, val)";

PROCEDURE -SetHalt*(p: PROCEDURE(n: LONGINT));
      "SYSTEM-Halt = p";

PROCEDURE Halt(n: LONGINT);
   VAR res: LONGINT;
BEGIN res := Unix.Kill(Unix.Getpid(), 4)";
END Halt;

BEGIN SetHalt(Halt)
END Kernel.
```

7 Module Loading

Module SYSTEM provides a registration service for Oberon modules. It deliberately does not provide dynamic linking facilities itself since this is platform- and possibly application-specific. Dynamic loading as provided, for example, by module Modules in ETH Oberon systems can be built on top of this registration service. For every module, *Ofront* generates an exported function named after the module and followed by the suffix _init. This function is the C counterpart of an Oberon module body. Upon the first call of an init function, the module is registered together with its commands and exported types by means of the registration service of module SYSTEM. In addition, the module is initialized as required by Oberon (the part of the init function that corresponds to the Oberon module body is preceded by the comment /* BEGIN */). Thus, when implementing dynamic linking, the only task is to call a module's init function, which returns a pointer to the module.

In order to get access to the list of loaded modules, module SYSTEM provides the anchor of the module list in variable SYSTEM_modules. Module nodes are defined as follows:

```
TYPE
  Module = POINTER TO ModuleDesc;
  Cmd = POINTER TO CmdDesc;

  ModuleDesc = RECORD
    next: Module;
    name: ARRAY 20 OF CHAR;
    refcnt: LONGINT;
    cmds: POINTER TO CmdDesc;
    types: LONGINT;
    enumPtrs: PROCEDURE (P: PROCEDURE(p: LONGINT))
    reserved1, reserved2: LONGINT
  END ;

  CmdDesc = RECORD
    next: Cmd;
    name: ARRAY 24 OF CHAR;
    cmd: Command
  END ;
```

Example An example implementation of module Modules (a component of the ETH Oberon V4 module library) for HP-UX with a particular shared

library search strategy is given below. Modules which are loaded at runtime are looked up

1. in the program's executable
2. in the previously loaded shared libraries
3. in a shared library named after the module to be loaded
4. in a subsytem named after the module name prefix; digits treated as upper case letters
5. in a subsytem named after the module name prefix; digits treated as lower case letters
6. in the shared library libOberonV4.sl.

```
MODULE Modules;
...
PROCEDURE -include() "#include <dl.h>";

PROCEDURE -dlopen(name: ARRAY OF CHAR): LONGINT
  "(long)shl-load(name, BIND-DEFERRED, 0)";

PROCEDURE -dlsym(VAR h: LONGINT; name: Name; VAR p: Command)
  "if (shl-findsym((shl-t*)h, (const char*)name, TYPE-PROCEDURE, p) ==
-1) *p = 0";

PROCEDURE -Prog(): LONGINT "(long)PROG-HANDLE";

PROCEDURE -modules*(): Module
  "(Modules_Module)SYSTEM_modules";

PROCEDURE Concat(s1, s2: ARRAY OF CHAR; VAR d : ARRAY OF CHAR);
  VAR i, j: INTEGER;
BEGIN i := 0; j := 0;
  WHILE s1[i] # 0X DO d[i] := s1[i]; INC(i); END;
  WHILE s2[j] # 0X DO d[i] := s2[j]; INC (i); INC(j); END;
  d[i] := 0X;
END Concat;

PROCEDURE GetSubsys1(n: ARRAY OF CHAR; VAR s: ARRAY OF CHAR);
  VAR i: INTEGER; ch: CHAR;
BEGIN
  ch := n[0]; i := 0;
  WHILE (ch # 0X) & ((ch < "a") OR (ch > "z")) DO
    s[i] := ch; INC(i); ch := n[i]
  END ;
```

```
  WHILE (ch >= "a") & (ch <= "z") DO s[i] := ch; INC(i); ch := n[i] END ;
  IF ch = 0X THEN s[0] := 0X ELSE s[i] := 0X END
END GetSubsys1;

PROCEDURE GetSubsys2(n: ARRAY OF CHAR; VAR s: ARRAY OF CHAR);
  VAR i: INTEGER; ch: CHAR;
BEGIN
  ch := n[0]; i := 0;
  WHILE (ch >= "A") & (ch <= "Z") DO s[i] := ch; INC(i); ch := n[i] END ;
  WHILE (ch # 0X) & ((ch < "A") OR (ch > "Z")) DO
    s[i] := ch; INC(i); ch := n[i]
  END ;
  IF ch = 0X THEN s[0] := 0X ELSE s[i] := 0X END
END GetSubsys2;

PROCEDURE ThisMod* (name: ARRAY OF CHAR): Module;
  VAR m: Module; bodyname, libname, libname2: ARRAY 64 OF CHAR;
    body: Command; lib, prog, all: LONGINT;
BEGIN m := modules();
  WHILE (m # NIL) & (m.name # name) DO m := m.next END ;
  IF m = NIL THEN
    all := 0; prog := Prog(); Concat(name, "--init", bodyname);
    dlsym(prog, bodyname, body);
    IF body = NIL THEN dlsym(all, bodyname, body) END ;
    IF body = NIL THEN
      Concat("lib", name, libname); Concat(libname, ".sl", libname);
      lib := dlopen(libname);
      IF lib # 0 THEN dlsym(lib, bodyname, body) END
    END ;
    IF body = NIL THEN
      GetSubsys1(name, libname);
      IF libname[0] # 0X THEN
        Concat("lib", libname, libname); Concat(libname, ".sl", libname);
        lib := dlopen(libname);
        IF lib # 0 THEN dlsym(lib, bodyname, body) END
      END
    END ;
    IF body = NIL THEN
      GetSubsys2(name, libname2);
      IF libname2[0] # 0X THEN
        Concat("lib", libname2, libname2);
        Concat(libname2, ".so", libname2);
```

```
            IF (libname2 # libname) THEN
              lib := dlopen(libname2);
              IF lib # 0 THEN dlsym(lib, bodyname, body) END
            END
          END
        END ;
        IF body = NIL THEN lib := dlopen("libOberonV4.sl");
          IF lib # 0 THEN dlsym(lib, bodyname, body) END
        END ;
        IF body # NIL THEN
          body(); m := modules();
          WHILE (m # NIL) & (m.name # name) DO m := m.next END
        END
      END ;
      IF m # NIL THEN res := 0 ELSE res := 1; COPY(name, importing) END ;
      RETURN m
    END ThisMod;

    PROCEDURE ThisCommand*
        (mod: Module; name: ARRAY OF CHAR): Command;
      VAR c: Cmd;
    BEGIN c := mod.cmds;
      WHILE (c # NIL) & (c.name # name) DO c := c.next END ;
      IF c = NIL THEN res := 2; RETURN NIL
      ELSE res := 0; RETURN c.cmd
      END
    END ThisCommand;

    END Modules.
```

8 Garbage Collection and Finalization

Conceptually speaking, the programming language Oberon is based on an infinite heap since there is no way to explicitly deallocate dynamic data structures. Automatic garbage collection in combination with dynamic heap expansion is a techique to implement this feature on today's finite hardware. Garbage collection is performed implicitly whenever the heap storage is exhausted. Only if there is not enough storage to be reclaimed, is the heap extended. In addition, *Ofront's* run-time system provides the procedure SYSTEM_GC to call the garbage collector explicitly as shown by the following code procedure:

```
PROCEDURE –GC(markStack: BOOLEAN)
  "SYSTEM_GC(markStack)";
```

The parameter markStack specifies whether the run-time stack should be consulted for the reachability analysis. As an optimization, markStack may be set to FALSE if it can be guaranteed that no objects are anchored on the stack, which is the case, for example, in the main Oberon loop of the ETH Oberon system. If in doubt, always pass TRUE for markStack.

gclock

Module SYSTEM provides the variable SYSTEM_gclock for controlling the activities of the garbage collector.

```
SHORTINT SYSTEM_gclock;
```

A value of 0 means default behavior, i.e., garbage collection before expanding the heap, value 1 means no garbage collection with markStack set to TRUE as it is the case if the heap storage is exhausted, and value 2 means no garbage collection at all even if SYSTEM_GC is called with markStack set to FALSE.

Finalization

Systems based on automatic garbage collection rather than explicit release of unused memory blocks require a mechanism to release external resources that are connected with released objects. This mechanism is usually called *finalization*. Examples are Unix file descriptors, which could be connected with Oberon file objects. Whenever Oberon's garbage collector detects that a file object is no longer used, it releases this object. Closing the associated Unix file descriptor directly by the garbage collector would imply that the garbage collector knows about file objects and Unix file descriptors. Since it is not possible for the garbage collector to know in advance all kinds of objects that have external resources associated with them, there

must be an extensible mechanism that allows performing such cleanup operations.

For this purpose, *Ofront* provides a registration service that associates an object with a finalization procedure. The finalization procedure is called when the object its released. If an object is registered n times, there are exactly n finalization procedures to be called. No assumption about the finalization order must be made. Although possible, a finalization procedure should never establish new references to the finalized object since this would prevent such objects from eventually being reclaimed by the garbage collector. Otherwise there are no restrictions to the finalization procedure.

The following declarations show the programmatic interface of the finalization mechanism. Procedure SYSTEM_FINALL is implicitly called at the end of a main program.

```
TYPE
   Finalizer = PROCEDURE(obj: SYSTEM.PTR);

PROCEDURE –RegisterFin(obj: SYSTEM.PTR; finalize: Finalizer)
   "SYSTEM_REGFIN(obj, finalize)";

PROCEDURE –FinalizeAll()
   "SYSTEM_FINALL()";
```

Appendix A

Supported Architectures

The following Ofront.par parameterization files have been prepared for your convenience. Additional architectures can easily be supported by compiling and executing the program ofrontparam.c as described in Section 2.4.

Ofront.aix.par
: RS6000/IBM AIX

Ofront.hpux.par
: PA-RISC/HP-UX Series 700, 800

Ofront.i960.par
: Intel i960 embedded controller with natural alignment

Ofront.irix5.par
: MIPS R4000 in 32 bit mode/Silicon Graphics IRIX 5.x

Ofront.linux386.par
: Intel 386/Linux

Ofront.sunos.par
: SPARC V7, V8/Solaris 1 and 2

Ofront.ultrix.par
: MIPS R2000/Ultrix

Supported Compilers

The following SYSTEM.h include files have been prepared and tested for the specified compilers and platforms. Additional platforms can be supported on demand. Please contact your *Ofront* distributor.

SYSTEM.cc.aix.h
: the IBM XlC compiler for RS6000/AIX machines

SYSTEM.cc.hpux.h

the C compiler bundled with HP-UX Series 700/800 machines

SYSTEM.cc.irix5.h
: the standard C compiler for MIPS R4000 based SGI IRIX 5.x machines

SYSTEM.cc.sunos4.h
: the C compiler bundled with SPARC/Solaris 1

SYSTEM.cc.sunos5.h
: the Sun C compiler for SPARC/Solaris 2

SYSTEM.cc.ultrix.h
: the C compiler bundled with DEC/Ultrix (MIPS based)

SYSTEM.gcc.i960.h
: the gnu C compiler for cross development of i960-based embedded systems

SYSTEM.gcc.linux.h
: the gnu C compiler for Linux based PCs.

SYSTEM.gcc.sunos4.h
: the gnu C compiler for SPARC/Solaris 1

Appendix B

Available Libraries

Unix platforms only

libOberonV4:

the ETH Oberon V4 module library consisting of a tiled window system, an extensible text and graphics editor and a number of extensions and utilities. Module CmdlnTexts may be used much like module Texts, but does not import modules Display and Fonts, thus CmdlnTexts may be used in command line programs that deal with texts but do not open a window. libOberonV4 may be distributed freely as long as the ETH copyright restrictions are observed.

SYSTEM
Args
Console
Modules
Unix
Kernel
Files
X11
Display
Input
Math
MathL
Fonts
Viewers
Reals
Texts
CmdlnTexts
Oberon
MenuViewers
TextFrames
In
Out
Printer
TextPrinter

ParcElems
System
Edit
EdiT
EditTools
MenuElems
IconElems
ClockElems
TextPFrames
TextPreview
TableElems
StyleElems
FoldElems
Folds
ErrorElems
ChartElems
LineElems
PopupElems
StampElems
AsciiCoder
Miscellaneous
FKeys
Colors
FontToBDF
Browser
Types
Display1
KeplerPorts
KeplerGraphs
KeplerFrames
Kepler
Kepler1
Kepler2
Kepler4
Kepler5
Kepler6
Kepler7
Kepler8
Kepler9
KeplerElems
Mailer

libOberonS3

The ETH Oberon System 3 module library (Release 2.0) consists of a tiled and an overlapping window system, text and graphics editors, graphical end-user objects (including buttons, text boxes, lists and panels) together with user interface construction tools, and ready-to-use components for accessing Internet services such as a world-wide-web browser, an electronic mail facility, and clients for ftp, gopher, finger, news, and telnet.
[not yet released]

Further libraries are in preparation.

Appendix C

The Programming Language Oberon-2

H. Mössenböck, N. Wirth
Institut für Computersysteme, ETH Zürich
October 1993

1 Introduction

Oberon-2 is a general-purpose programming language in the tradition of Pascal and Modula-2. Its most important features are block structure, modularity, separate compilation, static typing with strong type checking (also across module boundaries), and type extension with type-bound procedures.

Type extension makes Oberon-2 an object-oriented language. An object is a variable of an abstract data type consisting of private data (its state) and procedures that operate on this data. Abstract data types are declared as extensible records. Oberon-2 covers most terms of object-oriented languages by the established vocabulary of imperative languages in order to minimize the number of notions for similar concepts.

This report is not intended as a programmer's tutorial. It is intentionally kept concise. Its function is to serve as a reference for programmers, implementors, and manual writers. What remains unsaid is mostly left so intentionally, either because it can be derived from stated rules of the language, or because it would require to commit the definition when a general commitment appears as unwise.

Chapter 12 defines some terms that are used to express the type checking rules of Oberon-2. Where they appear in the text, they are written in italics to indicate their special meaning (e.g. the *same* type).

2 Syntax

An extended Backus-Naur Formalism (EBNF) is used to describe the syntax of Oberon-2: Alternatives are separated by |. Brackets [and] denote optionality of the enclosed expression, and braces { and } denote its repetition (possibly 0 times). Non-terminal symbols start with an upper-case letter (e.g. Statement). Terminal symbols either start with a

lower-case letter (e.g. ident), or are written all in upper-case letters (e.g. BEGIN), or are denoted by strings (e.g. ":=").

3 Vocabulary and Representation

The representation of (terminal) symbols in terms of characters is defined using the ASCII set. Symbols are identifiers, numbers, strings, operators, and delimiters. The following lexical rules must be observed: Blanks and line breaks must not occur within symbols (except in comments, and blanks in strings). They are ignored unless they are essential to separate two consecutive symbols. Capital and lower-case letters are considered as distinct.

1. *Identifiers* are sequences of letters and digits. The first character must be a letter.

```
$ ident = letter {letter | digit}.
```

Examples: x Scan Oberon2 GetSymbol firstLetter

2. *Numbers* are (unsigned) integer or real constants. The type of an integer constant is the minimal type to which the constant value belongs (see 6.1). If the constant is specified with the suffix H, the representation is hexadecimal otherwise the representation is decimal.

A real number always contains a decimal point. Optionally it may also contain a decimal scale factor. The letter E (or D) means "times ten to the power of". A real number is of type REAL, unless it has a scale factor containing the letter D. In this case it is of type LONGREAL.

```
$ number = integer | real.
$ integer = digit {digit} | digit {hexDigit} "H".
$ real = digit {digit} "." {digit} [ScaleFactor].
$ ScaleFactor = ("E" | "D") ["+" | "−"] digit {digit}.
$ hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
$ digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
```

Examples:

1991	INTEGER	1991
0DH	SHORTINT	13
12.3	REAL	12.3
4.567E8	REAL	456700000

0.57712566D–6 LONGREAL 0.00000057712566

3. *Character* constants are denoted by the ordinal number of the character in hexadecimal notation followed by the letter X.

```
$ character = digit {hexDigit} "X".
```

4. *Strings* are sequences of characters enclosed in single (’) or double (") quote marks. The opening quote must be the same as the closing quote and must not occur within the string. The number of characters in a string is called its *length*. A string of length 1 can be used wherever a character constant is allowed and vice versa.

```
$ string = '"' {char} '"' | "'" {char} "'".
```

Examples: "Oberon–2" "Don't worry!" "x"

5. *Operators* and *delimiters* are the special characters, character pairs, or reserved words listed below. The reserved words consist exclusively of capital letters and cannot be used as identifiers.

+	:=	ARRAY	IMPORT	RETURN
–	↑	BEGIN	IN	THEN
*	=	BY	IS	TO
/	#	CASE	LOOP	TYPE
~	<	CONST	MOD	UNTIL
&	>	DIV	MODULE	VAR
.	<=	DO	NIL	WHILE
,	>=	ELSE	OF	WITH
;	..	ELSIF	OR	
|	:	END	POINTER	
(	)	EXIT	PROCEDURE	
[	]	FOR	RECORD	
{	}	IF	REPEAT	

6. *Comments* may be inserted between any two symbols in a program. They are arbitrary character sequences opened by the bracket (* and closed by *). Comments may be nested. They do not affect the meaning of a program.

4 Declarations and scope rules

Every identifier occurring in a program must be introduced by a declaration, unless it is a predeclared identifier. Declarations also specify certain permanent properties of an object, such as whether it is a constant, a type, a variable, or a procedure. The identifier is then used to refer to the associated object.

The *scope* of an object *x* extends textually from the point of its declaration to the end of the block (module, procedure, or record) to which the declaration belongs and hence to which the object is *local*. It excludes the scopes of equally named objects which are declared in nested blocks. The scope rules are:

1. No identifier may denote more than one object within a given scope (i.e. no identifier may be declared twice in a block);
2. An object may only be referenced within its scope;
3. A type *T* of the form POINTER TO *T1* (see 6.4) can be declared at a point where *T1* is still unknown. The declaration of *T1* must follow in the same block to which *T* is local;
4. Identifiers denoting record fields (see 6.3) or type-bound procedures (see 10.2) are valid in record designators only.

An identifier declared in a module block may be followed by an export mark (" * " or " - ") in its declaration to indicate that it is exported. An identifier *x* exported by a module *M* may be used in other modules, if they import *M* (see Ch. 11). The identifier is then denoted as *M.x* in these modules and is called a *qualified identifier*. Identifiers marked with " - " in their declaration are *read-only* in importing modules.

```
$ Qualident = [ident "."] ident.
$ IdentDef = ident [" * " | " – "].
```

The following identifiers are predeclared; their meaning is defined in the indicated sections:

ABS	(10.3)	LEN	(10.3)
ASH	(10.3)	LONG	(10.3)
BOOLEAN	(6.1)	LONGINT	(6.1)
CAP	(10.3)	LONGREAL	(6.1)
CHAR	(6.1)	MAX	(10.3)
CHR	(10.3)	MIN	(10.3)
COPY	(10.3)	NEW	(10.3)

DEC	(10.3)	ODD	(10.3)
ENTIER	(10.3)	ORD	(10.3)
EXCL	(10.3)	REAL	(6.1)
FALSE	(6.1)	SET	(6.1)
HALT	(10.3)	SHORT	(10.3)
INC	(10.3)	SHORTINT	(6.1)
INCL	(10.3)	SIZE	(10.3)
INTEGER	(6.1)	TRUE	(6.1)

5 Constant declarations

A constant declaration associates an identifier with a constant value.

```
$ ConstantDeclaration = IdentDef "=" ConstExpression.
$ ConstExpression = Expression.
```

A constant expression is an expression that can be evaluated by a mere textual scan without actually executing the program. Its operands are constants (Ch.8) or predeclared functions (Ch.10.3) that can be evaluated at compile time. Examples of constant declarations are:

```
N = 100
limit = 2*N - 1
fullSet = {MIN(SET) .. MAX(SET)}
```

6 Type declarations

A data type determines the set of values which variables of that type may assume, and the operators that are applicable. A type declaration associates an identifier with a type. In the case of structured types (arrays and records) it also defines the structure of variables of this type. A structured type cannot contain itself.

```
$ TypeDeclaration = IdentDef "=" Type.
$ Type = Qualident | ArrayType | RecordType | PointerType | ProcedureType.
```

Examples:

```
Table = ARRAY N OF REAL
Tree = POINTER TO Node
```

```
Node =  RECORD
   key : INTEGER;
   left, right: Tree
END
CenterTree = POINTER TO CenterNode
CenterNode = RECORD (Node)
   width: INTEGER;
   subnode: Tree
END
Function = PROCEDURE(x: INTEGER): INTEGER
```

6.1 Basic types

The basic types are denoted by predeclared identifiers. The associated operators are defined in 8.2 and the predeclared function procedures in 10.3. The values of the given basic types are the following:

1. BOOLEAN	the truth values TRUE and FALSE
2. CHAR	the characters of the extended ASCII set (0X .. 0FFX)
3. SHORTINT	the integers between MIN(SHORTINT) and MAX(SHORTINT)
4. INTEGER	the integers between MIN(INTEGER) and MAX(INTEGER)
5. LONGINT	the integers between MIN(LONGINT) and MAX(LONGINT)
6. REAL	the real numbers between MIN(REAL) and MAX(REAL)
7. LONGREAL	the real numbers between MIN(LONGREAL) and MAX(LONGREAL)
8. SET	the sets of integers between 0 and MAX(SET)

Types 3 to 5 are *integer types*, types 6 and 7 are *real types*, and together they are called *numeric types*. They form a hierarchy; the larger type *includes* (the values of) the smaller type:

```
LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT
```

6.2 Array types

An array is a structure consisting of a number of elements which are all of the same type, called the *element type*. The number of elements of an array is called its *length*. The elements of the array are designated by indices, which are integers between 0 and the length minus 1.

```
$ ArrayType = ARRAY [Length {"," Length}] OF Type.
$ Length = ConstExpression.
```

A type of the form

```
ARRAY L0, L1, ..., Ln OF T
```

is understood as an abbreviation of

```
ARRAY L0 OF
  ARRAY L1 OF
  ...
    ARRAY Ln OF T
```

Arrays declared without length are called *open arrays*. They are restricted to pointer base types (see 6.4), element types of open array types, and formal parameter types (see 10.1). Examples:

```
ARRAY 10, N OF INTEGER
ARRAY OF CHAR
```

6.3 Record types

A record type is a structure consisting of a fixed number of elements, called *fields*, with possibly different types. The record type declaration specifies the name and type of each field. The scope of the field identifiers extends from the point of their declaration to the end of the record type, but they are also visible within designators referring to elements of record variables (see 8.1). If a record type is exported, field identifiers that are to be visible outside the declaring module must be marked. They are called *public fields*; unmarked elements are called *private fields*.

```
$ RecordType = RECORD ["("BaseType")"] FieldList {";" FieldList} END.
$ BaseType = Qualident.
$ FieldList = [IdentList ":" Type ].
```

Record types are extensible, i.e. a record type can be declared as an extension of another record type. In the example

```
T0 = RECORD x: INTEGER END
T1 = RECORD (T0) y: REAL END
```

T1 is a (direct) *extension* of *T0* and *T0* is the (direct) *base type* of *T1* (see Ch. 12). An extended type *T1* consists of the fields of its base type and of the fields which are declared in *T1*. All identifiers declared in the extended record must be different from the identifiers declared in its base type record(s).

Examples of record type declarations:

```
RECORD
  day, month, year: INTEGER
END

RECORD
  name, firstname: ARRAY 32 OF CHAR;
  age: INTEGER;
  salary: REAL
END
```

6.4 Pointer types

Variables of a pointer type *P* assume as values pointers to variables of some type *T*. *T* is called the pointer base type of *P* and must be a record or array type. Pointer types adopt the extension relation of their pointer base types: if a type *T1* is an extension of *T*, and *P1* is of type POINTER TO *T1*, then *P1* is also an extension of *P*.

```
$ PointerType = POINTER TO Type.
```

If *p* is a variable of type *P* = POINTER TO *T*, a call of the predeclared procedure *NEW(p)* (see 10.3) allocates a variable of type *T* in free storage. If *T* is a record type or an array type with fixed length, the allocation has to be done with *NEW(p)*; if T is an n-dimensional open array type the allocation has to be done with *NEW(p, e0, ..., en-1)* where *T* is allocated with lengths given by the expressions *e0, ..., en-1*. In either case a pointer to the allocated variable is assigned to *p*. *p* is of

type *P*. The *referenced* variable *p*↑ (pronounced as *p-referenced*) is of type *T*. Any pointer variable may assume the value NIL, which points to no variable at all.

6.5 Procedure types

Variables of a procedure type *T* have a procedure (or NIL) as value. If a procedure *P* is assigned to a variable of type *T*, the formal parameter lists (see Ch. 10.1) of *P* and *T* must *match* (see Ch. 12). *P* must not be a predeclared or type-bound procedure nor may it be local to another procedure.

```
$ ProcedureType = PROCEDURE [FormalParameters].
```

7 Variable declarations

Variable declarations introduce variables by defining an identifier and a data type for them.

```
$ VariableDeclaration = IdentList ":" Type.
```

Record and pointer variables have both a *static type* (the type with which they are declared - simply called their type) and a *dynamic type* (the type of their value at run time). For pointers and variable parameters of record type the dynamic type may be an extension of their static type. The static type determines which fields of a record are accessible. The dynamic type is used to call type-bound procedures (see 10.2).

Examples of variable declarations (refer to examples in Ch. 6):

```
i, j, k: INTEGER
x, y: REAL
p, q: BOOLEAN
s: SET
F: Function
a: ARRAY 100 OF REAL
w: ARRAY 16 OF RECORD
      name: ARRAY 32 OF CHAR;
      count: INTEGER
```

```
  END
t, c: Tree
```

8 Expressions

Expressions are constructs denoting rules of computation whereby constants and current values of variables are combined to compute other values by the application of operators and function procedures. Expressions consist of operands and operators. Parentheses may be used to express specific associations of operators and operands.

8.1 Operands

With the exception of set constructors and literal constants (numbers, character constants, or strings), operands are denoted by *designators*. A designator consists of an identifier referring to a constant, variable, or procedure. This identifier may possibly be qualified by a module identifier (see Ch. 4 and 11) and may be followed by *selectors* if the designated object is an element of a structure.

```
$ Designator = Qualident {"." ident | "[" ExpressionList "]" | "↑" |
        "(" Qualident ")"}.
$ ExpressionList = Expression {"," Expression}.
```

If *a* designates an array, then *a[e]* denotes that element of *a* whose index is the current value of the expression *e*. The type of *e* must be an integer type. A designator of the form *a[e0, e1, ..., en]* stands for *a[e0][e1]...[en]*. If *r* designates a record, then *r.f* denotes the field *f* of *r* or the procedure *f* bound to the dynamic type of *r* (Ch. 10.2). If *p* designates a pointer, *p↑* denotes the variable which is referenced by *p*. The designators *p↑.f* and *p↑[e]* may be abbreviated as *p.f* and *p[e]*, i.e. record and array selectors imply dereferencing. If *a* or *r* are read-only, then also *a[e]* and *r.f* are read-only.

A *type guard v(T)* asserts that the dynamic type of *v* is *T* (or an extension of *T*), i.e. program execution is aborted, if the dynamic type of *v* is not *T* (or an extension of *T*). Within the designator, *v* is then regarded as having the static type *T*. The guard is applicable, if

1. *v* is a variable parameter of record type or *v* is a pointer, and if
2. *T* is an extension of the static type of *v*

If the designated object is a constant or a variable, then the designator refers to its current value. If it is a procedure, the designator refers to that procedure unless it is followed by a (possibly empty) parameter list in which case it implies an activation of that procedure and stands for the value resulting from its execution. The actual parameters must correspond to the formal parameters as in proper procedure calls (see 10.1).

Examples of designators (refer to examples in Ch.7):

```
i                       (INTEGER)
a[i]                    (REAL)
w[3].name[i]            (CHAR)
t.left.right            (Tree)
t(CenterTree).subnode   (Tree)
```

8.2 Operators

Four classes of operators with different precedences (binding strengths) are syntactically distinguished in expressions. The operator ~ has the highest precedence, followed by multiplication operators, addition operators, and relations. Operators of the same precedence associate from left to right. For example, x-y-z stands for (x-y)-z.

```
$ Expression = SimpleExpression [Relation SimpleExpression].
$ SimpleExpression = ["+" | "-"] Term {AddOperator Term}.
$ Term = Factor {MulOperator Factor}.
$ Factor = Designator [ActualParameters] | number | character |
        string | NIL | Set | "(" Expression ")" | "~" Factor.
$ Set = "{" [Element {"," Element}] "}".
$ Element = Expression [".." Expression].
$ ActualParameters = "(" [ExpressionList] ")".
$ Relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
$ AddOperator = "+" | "-" | OR.
$ MulOperator = "*" | "/" | DIV | MOD | "&".
```

The available operators are listed in the following tables. Some operators are applicable to operands of various types, denoting different operations. In these cases, the actual operation is identified by the type of the operands. The operands must be *expression compatible* with respect to the operator (see Ch. 12).

8.2.1 Logical operators

OR	logical disjunction	p OR q	"if p then TRUE, else q"
&	logical conjunction	p & q	"if p then q, else FALSE"
~	negation	~ p	"not p"

These operators apply to BOOLEAN operands and yield a BOOLEAN result.

8.2.2 Arithmetic operators

+	sum
-	difference
*	product
/	real quotient
DIV	integer quotient
MOD	modulus

The operators +, -, *, and / apply to operands of numeric types. The type of the result is the type of that operand which includes the type of the other operand, except for division (/), where the result is the smallest real type which includes both operand types. When used as monadic operators, - denotes sign inversion and + denotes the identity operation. The operators DIV and MOD apply to integer operands only. They are related by the following formulas defined for any *x* and positive divisors *y*:

```
x = (x DIV y) * y + (x MOD y)
0 <= (x MOD y) < y
```

Examples:

x	y	x DIV y	x MOD y
5	3	1	2
−5	3	−2	1

8.2.3 Set Operators

+	union
-	difference (x - y = x * (-y))
*	intersection
/	symmetric set difference (x / y = (x-y) + (y-x))

Set operators apply to operands of type SET and yield a result of type SET. The monadic minus sign denotes the complement of *x*, i.e. -*x* denotes the set of integers between 0 and MAX(SET) which are not elements of *x*. Set operators are not associative ((a+b)-c # a+(b-c)).
A set constructor defines the value of a set by listing its elements between curly brackets. The elements must be integers in the range 0..MAX(SET). A range *a..b* denotes all integers in the interval [*a*, *b*].

8.2.4 Relations

=	equal
#	unequal
<	less
<=	less or equal
>	greater
>=	greater or equal
IN	set membership
IS	type test

Relations yield a BOOLEAN result. The relations =, #, <, <=, >, and >= apply to the numeric types, CHAR, strings, and character arrays containing 0X as a terminator. The relations = and # also apply to BOOLEAN and SET, as well as to pointer and procedure types (including the value NIL). *x IN s* stands for "*x* is an element of *s*". *x* must be of an integer type, and *s* of type SET. *v IS T* stands for "the dynamic type of *v* is *T* (or an extension of *T*)" and is called a *type test*. It is applicable if

1. *v* is a variable parameter of record type or *v* is a pointer, and if
2. *T* is an extension of the static type of *v*

Examples of expressions (refer to examples in Ch.7):

1991	INTEGER
i DIV 3	INTEGER
~p OR q	BOOLEAN
(i+j) * (i-j)	INTEGER
s - {8, 9, 13}	SET
i + x	REAL
a[i+j] * a[i-j]	REAL
(0<=i) & (i<100)	BOOLEAN
t.key = 0	BOOLEAN

```
k IN {i..j-1}              BOOLEAN
w[i].name <= "John"        BOOLEAN
t IS CenterTree            BOOLEAN
```

9 Statements

Statements denote actions. There are elementary and structured statements. Elementary statements are not composed of any parts that are themselves statements. They are the assignment, the procedure call, the return, and the exit statement. Structured statements are composed of parts that are themselves statements. They are used to express sequencing and conditional, selective, and repetitive execution. A statement may also be empty, in which case it denotes no action. The empty statement is included in order to relax punctuation rules in statement sequences.

```
$ Statement = [ Assignment | ProcedureCall | IfStatement | CaseStatement |
      WhileStatement | RepeatStatement | LoopStatement |
      ForStatement | WithStatement | EXIT | RETURN [Expression] ].
```

9.1 Assignments

Assignments replace the current value of a variable by a new value specified by an expression. The expression must be *assignment compatible* with the variable (see Ch. 12). The assignment operator is written as ":=" and pronounced as *becomes*.

```
$ Assignment = Designator ":=" Expression.
```

If an expression *e* of type *Te* is assigned to a variable *v* of type *Tv*, the following happens:

1. if *Tv* and *Te* are record types, only those fields of *Te* are assigned which also belong to *Tv* (*projection*); the dynamic type of *v* must be the *same* as the static type of *v* and is not changed by the assignment;
2. if *Tv* and *Te* are pointer types, the dynamic type of *v* becomes the dynamic type of *e*;
3. if *Tv* is ARRAY n OF CHAR and *e* is a string of length *m*<*n*, *v[i]* becomes e_i for *i* = 0..*m*-1 and *v[m]* becomes 0X.

Examples of assignments (refer to examples in Ch.7):

```
i := 0
p := i = j
x := i + 1
k := log2(i+j)
F := log2                                   (* see 10.1 *)
s := {2, 3, 5, 7, 11, 13}
a[i] := (x+y) * (x-y)
t.key := i
w[i+1].name := "John"
t := c
```

9.2 Procedure calls

A procedure call activates a procedure. It may contain a list of actual parameters which replace the corresponding formal parameters defined in the procedure declaration (see Ch. 10). The correspondence is established by the positions of the parameters in the actual and formal parameter lists. There are two kinds of parameters: *variable* and *value parameters*.

If a formal parameter is a variable parameter, the corresponding actual parameter must be a designator denoting a variable. If it denotes an element of a structured variable, the component selectors are evaluated when the formal/actual parameter substitution takes place, i.e. before the execution of the procedure. If a formal parameter is a value parameter, the corresponding actual parameter must be an expression. This expression is evaluated before the procedure activation, and the resulting value is assigned to the formal parameter (see also 10.1).

```
$ ProcedureCall = Designator [ActualParameters].
```

Examples:

```
WriteInt(i*2+1)      (* see 10.1 *)
INC(w[k].count)
t.Insert("John")     (* see 11 *)
```

9.3 Statement sequences

Statement sequences denote the sequence of actions specified by the

component statements which are separated by semicolons.

```
$ StatementSequence = Statement {";" Statement}.
```

9.4 If statements

```
$ IfStatement = IF Expression THEN StatementSequence
    {ELSIF Expression THEN StatementSequence}
    [ELSE StatementSequence]
    END.
```

If statements specify the conditional execution of guarded statement sequences. The Boolean expression preceding a statement sequence is called its *guard*. The guards are evaluated in sequence of occurrence, until one evaluates to TRUE, whereafter its associated statement sequence is executed. If no guard is satisfied, the statement sequence following the symbol ELSE is executed, if there is one.

Example:

```
IF (ch >= "A") & (ch <= "Z") THEN ReadIdentifier
ELSIF (ch >= "0") & (ch <= "9") THEN ReadNumber
ELSIF (ch = " ' ") OR (ch = ' " ') THEN ReadString
ELSE SpecialCharacter
END
```

9.5 Case statements

Case statements specify the selection and execution of a statement sequence according to the value of an expression. First the case expression is evaluated, then that statement sequence is executed whose case label list contains the obtained value. The case expression must either be of an *integer type* that *includes* the types of all case labels, or both the case expression and the case labels must be of type CHAR. Case labels are constants, and no value must occur more than once. If the value of the expression does not occur as a label of any case, the statement sequence following the symbol ELSE is selected, if there is one, otherwise the program is aborted.

```
$ CaseStatement = CASE Expression OF Case {"|" Case}
        [ELSE StatementSequence] END.
$ Case = [CaseLabelList ":" StatementSequence].
$ CaseLabelList = CaseLabels {"," CaseLabels}.
$ CaseLabels = ConstExpression [".." ConstExpression].
```

Example:

```
CASE ch OF
   "A" .. "Z": ReadIdentifier
|  "0" .. "9": ReadNumber
|  " ' ", ' " ': ReadString
ELSE SpecialCharacter
END
```

9.6 While statements

While statements specify the repeated execution of a statement sequence while the Boolean expression (its *guard*) yields TRUE. The guard is checked before every execution of the statement sequence.

```
$ WhileStatement = WHILE Expression DO StatementSequence END.
```

Examples:

```
WHILE i > 0 DO i := i DIV 2; k := k + 1 END
WHILE (t # NIL) & (t.key # i) DO t := t.left END
```

9.7 Repeat statements

A repeat statement specifies the repeated execution of a statement sequence until a condition specified by a Boolean expression is satisfied. The statement sequence is executed at least once.

```
$ RepeatStatement = REPEAT StatementSequence UNTIL Expression.
```

9.8 For statements

A for statement specifies the repeated execution of a statement sequence while a progression of values is assigned to an integer

variable called the *control variable* of the for statement.

```
$ ForStatement = FOR ident ":=" Expression TO Expression
      [BY ConstExpression] DO StatementSequence END.
```

The statement

```
FOR v := beg TO end BY step DO statements END
```

is equivalent to

```
temp := end; v := beg;
IF step > 0 THEN
  WHILE v <= temp DO statements; v := v + step END
ELSE
  WHILE v >= temp DO statements; v := v + step END
END
```

temp has the *same* type as *v*. *step* must be a nonzero constant expression. If *step* is not specified, it is assumed to be 1.

Examples:

```
FOR i := 0 TO 79 DO k := k + a[i] END
FOR i := 79 TO 1 BY –1 DO a[i] := a[i–1] END
```

9.9 Loop statements

A loop statement specifies the repeated execution of a statement sequence. It is terminated upon execution of an exit statement within that sequence (see 9.10).

```
$ LoopStatement = LOOP StatementSequence END.
```

Example:

```
LOOP
  ReadInt(i);
  IF i < 0 THEN EXIT END;
  WriteInt(i)
END
```

Loop statements are useful to express repetitions with several exit

points or cases where the exit condition is in the middle of the repeated statement sequence.

9.10 Return and exit statements

A return statement indicates the termination of a procedure. It is denoted by the symbol RETURN, followed by an expression if the procedure is a function procedure. The type of the expression must be *assignment compatible* (see Ch. 12) with the result type specified in the procedure heading (see Ch. 10).

Function procedures require the presence of a return statement indicating the result value. In proper procedures, a return statement is implied by the end of the procedure body. Any explicit return statement therefore appears as an additional (probably exceptional) termination point.

An exit statement is denoted by the symbol EXIT. It specifies termination of the enclosing loop statement and continuation with the statement following that loop statement. Exit statements are contextually, although not syntactically associated with the loop statement which contains them.

9.11 With statements

With statements execute a statement sequence depending on the result of a type test and apply a type guard to every occurrence of the tested variable within this statement sequence.

```
$ WithStatement = WITH Guard DO StatementSequence
        {"|" Guard DO StatementSequence} [ELSE StatementSequence] END.
$ Guard = Qualident ":" Qualident.
```

If *v* is a variable parameter of record type or a pointer variable, and if it is of a static type *T0*, the statement

```
WITH v: T1 DO S1 | v: T2 DO S2 ELSE S3 END
```

has the following meaning: if the dynamic type of *v* is *T1*, then the statement sequence *S1* is executed where *v* is regarded as if it had the static type *T1*; else if the dynamic type of *v* is *T2*, then *S2* is executed where *v* is regarded as if it had the static type *T2*; else *S3* is executed.

T1 and *T2* must be extensions of *T0*. If no type test is satisfied and if an else clause is missing the program is aborted.

Example:

```
WITH t: CenterTree DO i := t.width; c := t.subnode END
```

10 Procedure declarations

A procedure declaration consists of a *procedure heading* and a *procedure body*. The heading specifies the procedure identifier and the *formal parameters*. For type-bound procedures it also specifies the receiver parameter. The body contains declarations and statements. The procedure identifier is repeated at the end of the procedure declaration.

There are two kinds of procedures: *proper procedures* and *function procedures*. The latter are activated by a function designator as a constituent of an expression and yield a result that is an operand of the expression. Proper procedures are activated by a procedure call. A procedure is a function procedure if its formal parameters specify a result type. The body of a function procedure must contain a return statement which defines its result.

All constants, variables, types, and procedures declared within a procedure body are *local* to the procedure. Since procedures may be declared as local objects too, procedure declarations may be nested. The call of a procedure within its declaration implies recursive activation.

Objects declared in the environment of the procedure are also visible in those parts of the procedure in which they are not concealed by a locally declared object with the same name.

```
$ ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
$ ProcedureHeading =
        PROCEDURE [Receiver] IdentDef [FormalParameters].
$ ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
$ DeclarationSequence = {CONST {ConstantDeclaration ";"} |
        TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"} }
        {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
$ ForwardDeclaration =
        PROCEDURE "↑" [Receiver] IdentDef [FormalParameters].
```

If a procedure declaration specifies a *receiver* parameter, the procedure is considered to be bound to a type (see 10.2). A *forward declaration* serves to allow forward references to a procedure whose actual

declaration appears later in the text. The formal parameter lists of the forward declaration and the actual declaration must *match* (see Ch. 12).

10.1 Formal parameters

Formal parameters are identifiers declared in the formal parameter list of a procedure. They correspond to actual parameters specified in the procedure call. The correspondence between formal and actual parameters is established when the procedure is called. There are two kinds of parameters, *value* and *variable parameters*, indicated in the formal parameter list by the absence or presence of the keyword VAR. Value parameters are local variables to which the value of the corresponding actual parameter is assigned as an initial value. Variable parameters correspond to actual parameters that are variables, and they stand for these variables. The scope of a formal parameter extends from its declaration to the end of the procedure block in which it is declared. A function procedure without parameters must have an empty parameter list. It must be called by a function designator whose actual parameter list is empty too. The result type of a procedure can be neither a record nor an array.

```
$ FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" Qualident].
$ FPSection = [VAR] ident {"," ident} ":" Type.
```

Let *Tf* be the type of a formal parameter *f* (not an open array) and *Ta* the type of the corresponding actual parameter *a*. For variable parameters, *Ta* must be the *same* as *Tf*, or *Tf* must be a record type and *Ta* an extension of *Tf*. For value parameters, *a* must be *assignment compatible* with *f* (see Ch. 12).

If *Tf* is an open array , then *a* must be *array compatible* with *f* (see Ch. 12). The lengths of *f* are taken from *a*.

Examples of procedure declarations:

```
PROCEDURE ReadInt(VAR x: INTEGER);
  VAR i: INTEGER; ch: CHAR;
BEGIN i := 0; Read(ch);
  WHILE ("0" <= ch) & (ch <= "9") DO
    i := 10*i + (ORD(ch)-ORD("0")); Read(ch)
  END;
  x := i
```

```
END ReadInt

PROCEDURE WriteInt(x: INTEGER); (*0 <= x <100000*)
  VAR i: INTEGER; buf: ARRAY 5 OF INTEGER;
BEGIN i := 0;
  REPEAT buf[i] := x MOD 10; x := x DIV 10; INC(i) UNTIL x = 0;
  REPEAT DEC(i); Write(CHR(buf[i] + ORD("0"))) UNTIL i = 0
END WriteInt

PROCEDURE WriteString(s: ARRAY OF CHAR);
  VAR i: INTEGER;
BEGIN i := 0;
  WHILE (i < LEN(s)) & (s[i] # 0X) DO Write(s[i]); INC(i) END
END WriteString;

PROCEDURE log2(x: INTEGER): INTEGER;
  VAR y: INTEGER; (*assume x>0*)
BEGIN
  y := 0; WHILE x > 1 DO x := x DIV 2; INC(y) END;
  RETURN y
END log2
```

10.2 Type-bound procedures

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

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

If a procedure *P* is bound to a type *T0*, it is implicitly also bound to any type *T1* which is an extension of *T0*. However, a procedure *P'* (with the same name as *P*) may be explicitly bound to *T1* in which case it overrides the binding of *P*. *P'* is considered a *redefinition* of *P* for *T1*. The formal parameters of *P* and *P'* must *match* (see Ch. 12). If *P* and *T1*

are exported (see Chapter 4) *P'* must be exported too.

If *v* is a designator and *P* is a type-bound procedure, then *v.P* denotes that procedure *P* which is bound to the dynamic type of *v*. Note, that this may be a different procedure than the one bound to the static type of *v*. *v* is passed to *P*'s receiver according to the parameter passing rules specified in Chapter 10.1.

If *r* is a receiver parameter declared with type *T*, *r.P*↑ denotes the procedure *P* bound to the base type of *T*.

In a forward declaration of a type-bound procedure the receiver parameter must be of the *same* type as in the actual procedure declaration. The formal parameter lists of both declarations must *match* (Ch. 12).

Examples:

```
PROCEDURE (t: Tree) Insert (node: Tree);
  VAR p, father: Tree;
BEGIN p := t;
  REPEAT father := p;
    IF node.key = p.key THEN RETURN END;
    IF node.key < p.key THEN p := p.left ELSE p := p.right END
  UNTIL p = NIL;
  IF node.key < father.key THEN father.left := node
  ELSE father.right := node
  END;
  node.left := NIL; node.right := NIL
END Insert;

PROCEDURE (t: CenterTree) Insert (node: Tree);  (*redefinition*)
BEGIN
  WriteInt(node(CenterTree).width);
  t.Insert↑ (node)  (* calls the Insert procedure bound to Tree *)
END Insert;
```

10.3 Predeclared procedures

The following table lists the predeclared procedures. Some are generic procedures, i.e. they apply to several types of operands. *v* stands for a variable, *x* and *n* for expressions, and *T* for a type.

Function procedures

Name	Argument type	Result type	Function
ABS(x)	numeric type	type of x	absolute value
ASH(x, n)	x, n: integer type	LONGINT	arithmetic shift ($x * 2^n$)
CAP(x)	CHAR	CHAR	if x is a letter, the corresponding capital letter
CHR(x)	integer type	CHAR	character with ordinal number x
ENTIER(x)	real type	LONGINT	largest integer not greater than x
LEN(v, n)	v: array; n: integer const.	LONGINT	length of v in dimension n (first dimension = 0)
LEN(v)	v: array	LONGINT	equivalent to LEN(v, 0)
LONG(x)	SHORTINT	INTEGER	identity
	INTEGER	LONGINT	
	REAL	LONGREAL	
MAX(T)	T = basic type	T	maximum value of type T
	T = SET	INTEGER	maximum element of a set
MIN(T)	T = basic type	T	minimum value of type T
	T = SET	INTEGER	0
ODD(x)	integer type	BOOLEAN	x MOD 2 = 1
ORD(x)	CHAR	INTEGER	ordinal number of x
SHORT(x)	LONGINT	INTEGER	identity
	INTEGER	SHORTINT	identity
	LONGREAL	REAL	identity (truncation possible)
SIZE(T)	any type	integer type	number of bytes required by T

Proper procedures

Name	Argument types	Function
ASSERT(x)	x: Boolean expression	terminate program if not x
ASSERT(x, n)	x: Boolean expression; n: integer constant	terminate program if not x
COPY(x, v)	x: character array, string; v: character array	v := x truncation possible
DEC(v)	integer type	v := v – 1
DEC(v, n)	v, n: integer type	v := v – n
EXCL(v, x)	v: SET; x: integer type	v := v – {x}
HALT(n)	integer constant	terminate program
INC(v)	integer type	v := v + 1
INC(v, n)	v, n: integer type	v := v + n
INCL(v, x)	v: SET; x: integer type	v := v + {x}
NEW(v)	pointer to record or fixed size array	allocate v↑

```
NEW(v, x0, ..., xn)   v: pointer to open array;      allocate v↑ with
                      xi: integer type               lengths x0.. xn
```

COPY allows the assignment of a string or a character array containing a terminating 0X to another character array. If necessary, the assigned value is truncated to the target length minus one. The target will always contain 0X as a terminator. In ASSERT(x, n) and HALT(n), the interpretation of *n* is left to the underlying system implementation.

11 Modules

A module is a collection of declarations of constants, types, variables, and procedures, together with a sequence of statements for the purpose of assigning initial values to the variables. A module constitutes a text that is compilable as a unit.

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

The import list specifies the names of the imported modules. If a module *A* is imported by a module *M* and *A* exports an identifier *x*, then *x* is referred to as *A.x* within *M*. If *A* is imported as *B := A*, the object *x* must be referenced as *B.x*. This allows short alias names in qualified identifiers. A module must not import itself. Identifiers that are to be exported (i.e. that are to be visible in client modules) must be marked by an export mark in their declaration (see Chapter 4).

The statement sequence following the symbol BEGIN is executed when the module is added to a system (loaded), which is done after the imported modules have been loaded. It follows that cyclic import of modules is illegal. Individual (parameterless and exported) procedures can be activated from the system, and these procedures serve as *commands*.

```
MODULE Trees;
  IMPORT Texts, Oberon;
  (* exports: Tree, Node, Insert, Search, Write, Init;
    exports read-only: Node.name *)
  TYPE
    Tree* = POINTER TO Node;
```

```
  Node* = RECORD
    name-: POINTER TO ARRAY OF CHAR;
    left, right: Tree
  END;

VAR w: Texts.Writer;

PROCEDURE (t: Tree) Insert* (name: ARRAY OF CHAR);
  VAR p, father: Tree;
BEGIN p := t;
  REPEAT father := p;
    IF name = p.name↑ THEN RETURN END;
    IF name < p.name↑ THEN p := p.left ELSE p := p.right END
  UNTIL p = NIL;
  NEW(p); p.left := NIL; p.right := NIL;
  NEW(p.name, LEN(name)+1); COPY(name, p.name↑);
  IF name < father.name↑ THEN father.left := p
  ELSE father.right := p
  END
END Insert;

PROCEDURE (t: Tree) Search* (name: ARRAY OF CHAR): Tree;
  VAR p: Tree;
BEGIN p := t;
  WHILE (p # NIL) & (name # p.name↑) DO
    IF name < p.name↑ THEN p := p.left ELSE p := p.right END
  END;
  RETURN p
END Search;

PROCEDURE (t: Tree) Write*;
BEGIN
  IF t.left # NIL THEN t.left.Write END;
  Texts.WriteString(w, t.name↑); Texts.WriteLn(w);
  Texts.Append(Oberon.Log, w.buf);
  IF t.right # NIL THEN t.right.Write END
END Write;

PROCEDURE Init* (t: Tree);
BEGIN NEW(t.name, 1); t.name[0] := 0X; t.left := NIL; t.right := NIL
END Init;
```

```
BEGIN Texts.OpenWriter(w)
END Trees.
```

12 Definition of terms

Integer types	SHORTINT, INTEGER, LONGINT
Real types	REAL, LONGREAL
Numeric types	integer types, real types

Same types
Two variables *a* and *b* with types *Ta* and *Tb* are of the *same* type if
1. *Ta* and *Tb* are both denoted by the same type identifier, or
2. *Ta* is declared to equal *Tb* in a type declaration of the form *Ta = Tb*, or
3. *a* and *b* appear in the same identifier list in a variable, record field, or formal parameter declaration and are not open arrays.

Equal types
Two types *Ta* and *Tb* are *equal* if
1. *Ta* and *Tb* are the *same* type, or
2. *Ta* and *Tb* are open array types with *equal* element types, or
3. *Ta* and *Tb* are procedure types whose formal parameter lists *match*.

Type inclusion
Numeric types *include* (the values of) smaller numeric types according to the following hierarchy:
LONGREAL >= REAL >= LONGINT >= INTEGER >= SHORTINT

Type extension (base type)
Given a type declaration *Tb = RECORD (Ta) ... END*, *Tb* is a *direct extension* of *Ta*, and *Ta* is a *direct base type* of *Tb*. A type *Tb* is an *extension* of a type *Ta* (*Ta* is a *base type* of *Tb*) if
1. *Ta* and *Tb* are the *same* types, or
2. *Tb* is a *direct extension* of an *extension* of Ta

If *Pa = POINTER TO Ta* and *Pb = POINTER TO Tb*, *Pb* is an *extension* of *Pa* (*Pa* is a *base type* of *Pb*) if *Tb* is an *extension* of *Ta*.

Assignment compatible
An expression *e* of type *Te* is *assignment compatible* with a variable *v* of type *Tv* if one of the following conditions hold:

1. *Te* and *Tv* are the *same* type;
2. *Te* and *Tv* are numeric types and *Tv includes Te*;
3. *Te* and *Tv* are record types and *Te* is an *extension* of *Tv* and the dynamic type of *v* is *Tv* ;
4. *Te* and *Tv* are pointer types and *Te* is an *extension* of *Tv*;
5. *Tv* is a pointer or a procedure type and *e* is NIL;
6. *Tv* is ARRAY n OF CHAR, *e* is a string constant with *m* characters, and $m < n$;
7. *Tv* is a procedure type and *e* is the name of a procedure whose formal parameters *match* those of *Tv*.

Array compatible

An actual parameter *a* of type *Ta* is *array compatible* with a formal parameter *f* of type *Tf* if

1. *Tf* and *Ta* are the *same* type, or
2. *Tf* is an open array, *Ta* is any array, and their element types are *array compatible*, or
3. *Tf* is ARRAY OF CHAR and *a* is a string.

Expression compatible

For a given operator, the types of its operands are *expression compatible* if they conform to the following table (which shows also the result type of the expression). Character arrays that are to be compared must contain 0X as a terminator. Type T1 must be an extension of type T0, P0 and P1 denote pointer types bound to T0 and T1 respectively and Q stands for a procedure type. S stands for a character array or a string literal.

operator	*first operand*	*second operand*	*result type*
+ − *	*numeric*	*numeric*	smallest *numeric* type including both operands
/	*numeric*	*numeric*	smallest *real* type including both operands
+ − * /	SET	SET	SET
DIV MOD	*integer*	*integer*	smallest *integer* type including both operands
OR & ~	BOOLEAN	BOOLEAN	BOOLEAN
= # < <= > >=	*numeric*	*numeric*	BOOLEAN
	CHAR	CHAR	BOOLEAN
	S	S	BOOLEAN
= #	BOOLEAN	BOOLEAN	BOOLEAN
	SET	SET	BOOLEAN

	NIL, P0 or P1	NIL, P0 or P1	BOOLEAN
	Q, NIL	Q, NIL	BOOLEAN
IN	*integer*	SET	BOOLEAN
IS	type T0	type T1	BOOLEAN

Matching formal parameter lists

Two formal parameter lists *match* if

1. they have the same number of parameters, and
2. they have either the *same* function result type or none, and
3. parameters at corresponding positions have *equal* types, and
4. parameters at corresponding positions are both either value or variable parameters.

Appendix D

Grammar of Oberon-2

```
module = MODULE ident ";" [ImportList] DeclarationSequence
    [BEGIN StatementSequence] END ident "." .
ImportList = IMPORT import {"," import} ";" .
import = ident [":=" ident].
DeclarationSequence = {CONST {ConstantDeclaration ";"} |
    TYPE {TypeDeclaration ";"} | VAR {VariableDeclaration ";"}}
    {ProcedureDeclaration ";" | ForwardDeclaration ";"}.
ConstantDeclaration = identdef "=" ConstExpression.
identdef = ident ["*" | "-"].
ConstExpression = expression.
TypeDeclaration = identdef "=" type.
type = qualident | ArrayType | RecordType | PointerType | ProcedureType.
qualident = [ident "."] ident.
ArrayType = ARRAY [length {"," length}] OF type.
length = ConstExpression.
RecordType = RECORD ["(" BaseType ")"] FieldListSequence END.
BaseType = qualident.
FieldListSequence = FieldList {";" FieldList}.
FieldList = [IdentList ":" type].
IdentList = identdef {"," identdef}.
PointerType = POINTER TO type.
ProcedureType = PROCEDURE [FormalParameters].
VariableDeclaration = IdentList ":" type.
ProcedureDeclaration = ProcedureHeading ";" ProcedureBody ident.
ProcedureHeading = PROCEDURE [Receiver] ["*"] identdef
    [FormalParameters].
Receiver = "(" [VAR] ident ":" ident ")".
ProcedureBody = DeclarationSequence [BEGIN StatementSequence] END.
FormalParameters = "(" [FPSection {";" FPSection}] ")" [":" qualident].
FPSection = [VAR] ident {"," ident} ":" FormalType.
FormalType = type.
ForwardDeclaration = PROCEDURE [Receiver] "↑" identdef
    [FormalParameters].
StatementSequence = statement {";" statement}.
statement = [assignment | ProcedureCall | IfStatement |
    CaseStatement | WhileStatement | RepeatStatement | LoopStatement |
    WithStatement | ForStatement | EXIT | RETURN [expression] ].
```

```
assignment = designator ":=" expression.
designator = qualident {"." ident | "[" ExpList "]" | "(" qualident ")" | "↑" }.
ExpList = expression {"," expression}.
expression = SimpleExpression [relation SimpleExpression].
relation = "=" | "#" | "<" | "<=" | ">" | ">=" | IN | IS.
SimpleExpression = ["+"|"−"] term {AddOperator term}.
AddOperator = "+" | "−" | OR .
term = factor {MulOperator factor}.
MulOperator = "*" | "/" | DIV | MOD | "&" .
factor = number | CharConstant | string | NIL | set |
    designator [ActualParameters] | "(" expression ")" | "~" factor.
set = "{" [element {"," element}] "}".
element = expression [".." expression].
ProcedureCall = designator [ActualParameters].
ActualParameters = "(" [ExpList] ")" .
IfStatement = IF expression THEN StatementSequence
    {ELSIF expression THEN StatementSequence}
    [ELSE StatementSequence] END.
CaseStatement = CASE expression OF case {"|" case}
    [ELSE StatementSequence] END.
case = [CaseLabelList ":" StatementSequence].
CaseLabelList = CaseLabels {"," CaseLabels}.
CaseLabels = ConstExpression [".." ConstExpression].
WhileStatement = WHILE expression DO StatementSequence END.
RepeatStatement = REPEAT StatementSequence UNTIL expression.
LoopStatement = LOOP StatementSequence END.
WithStatement = WITH guard DO StatementSequence
    {"|" guard DO StatementSequence} [ELSE StatementSequence] END.
guard = qualident ":" qualident.
ForStatement = FOR ident ":=" expression TO expression
    [BY ConstExpression] DO StatementSequence END.
```

Lexical structure

ident = letter {letter | digit}.
number = integer | real.
integer = digit {digit} | digit {hexDigit} "H".
real = digit {digit} "." {digit} [ScaleFactor].
ScaleFactor = ("E" | "D") ["+" | "−"] digit {digit}.
hexDigit = digit | "A" | "B" | "C" | "D" | "E" | "F".
digit = "0" | "1" | "2" | "3" | "4" | "5" | "6" | "7" | "8" | "9".
CharConstant = "'" character "'" | "'" character "'" | digit {hexDigit} "X".
string = "'" {character} "'" | "'" {character} "'".

The ASCII-Code

	0	1	2	3	4	5	6	7
0	nul	dle		0	@	P	`	p
1	soh	dc1	!	1	A	Q	a	q
2	stx	dc2	"	2	B	R	b	r
3	etx	dc3	#	3	C	S	c	s
4	eot	dc4	$	4	D	T	d	t
5	enq	nak	%	5	E	U	e	u
6	ack	syn	&	6	F	V	f	v
7	bel	etb	'	7	G	W	g	w
8	bs	can	(	8	H	X	h	x
9	ht	em	)	9	I	Y	i	y
A	lf	sub	*	:	J	Z	j	z
B	vt	esc	+	;	K	[	k	{
C	ff	fs	,	<	L	\	l	|
D	cr	gs	−	=	M	]	m	}
E	so	rs	.	>	N	↑	n	~
F	si	us	/	?	O	-	o	del

Appendix E

Limitations of the Implementation (Version 1.0)

Only one-dimensional open arrays are currently supported.

Ofront assumes that variables of type LONGINT, pointers and procedure variables are the same size, for example all 32 bit.

The implementation of intermediate-level variables is not reentrant; i.e., procedures that access intermediate-level variables cannot be used in a multithread program.

Range checks are not always emitted, in particular, they are not emitted for SET operations.

Untagged records (RECORD [1]) are only rudimentarily supported and should not be used currently.

No link-time interface checking is performed.

There is no translator option to generate code for NIL checks because it is assumed that NIL checks are always done by the hardware (e.g., by protecting low memory pages from read and write access). However, this might not be the case in embedded systems and there are even Unix systems that allow reading (or even writing) the zero page.

The browsing facilities using the showdef or Browser.ShowDef commands are not as elaborate as they could be. In particular, currently they do not allow showing the definition of individual exported objects but always decode the complete interface of a module.

There are only a few shell scripts prepared that support automation of the multiple steps necessary to create an application or library.

There is no make file generator included since due to the fine grained interface checks employed by *Ofront* a simple file-based time stamping technique seems to be inappropriate.

HP-UX, IRIX 5: Open array value parameters may confuse the

conservative stack collection phase of the garbage collector if they contain pointers. The reason is that, due to the missing alloca function in HP-UX and IRIX 5, such arrays are currently copied onto the Unix heap by means of malloc. As a consequence, pointers inside these arrays are not seen by the garbage collector when inspecting the procedure activation stack.

Appendix F

Ofront Error Messages

NW, RC, JT / 16.1.95

1. Incorrect use of the language Oberon

0 undeclared identifier
1 multiply defined identifier
2 illegal character in number
3 illegal character in string
4 identifier does not match procedure name
5 comment not closed
6
7
8
9 "=" expected
10
11
12 type definition starts with incorrect symbol
13 factor starts with incorrect symbol
14 statement starts with incorrect symbol
15 declaration followed by incorrect symbol
16 MODULE expected
17
18 "." missing
19 "," missing
20 ":" missing
21
22 ")" missing
23 "]" missing
24 "}" missing
25 OF missing
26 THEN missing
27 DO missing
28 TO missing
29
30 "(" missing
31
32

33
34 ":=" missing
35 "," or OF expected
36
37
38 identifier expected
39 ";" missing
40
41 END missing
42
43
44 UNTIL missing
45
46 EXIT not within loop statement
47 illegally marked identifier
48
49
50 expression should be constant
51 constant not an integer
52 identifier does not denote a type
53 identifier does not denote a record type
54 result type of procedure is not a basic type
55 procedure call of a function
56 assignment to non–variable
57 pointer not bound to record or array type
58 recursive type definition
59 illegal open array parameter
60 wrong type of case label
61 inadmissible type of case label
62 case label defined more than once
63 illegal value of constant
64 more actual than formal parameters
65 fewer actual than formal parameters
66 element types of actual array and formal open array differ
67 actual parameter corresponding to open array is not an array
68 control variable must be integer
69 parameter must be an integer constant
70 pointer or VAR record required as formal receiver
71 pointer expected as actual receiver
72 procedure must be bound to a record of the same scope
73 procedure must have level 0
74 procedure unknown in base type

75 invalid call of base procedure
76 this variable (field) is read only
77 object is not a record
78 dereferenced object is not a variable
79 indexed object is not a variable
80 index expression is not an integer
81 index out of specified bounds
82 indexed variable is not an array
83 undefined record field
84 dereferenced variable is not a pointer
85 guard or test type is not an extension of variable type
86 guard or testtype is not a pointer
87 guarded or tested variable is neither a pointer nor a VAR–parameter record
88 open array not allowed as variable, record field or array element
89
90
91
92 operand of IN not an integer, or not a set
93 set element type is not an integer
94 operand of & is not of type BOOLEAN
95 operand of OR is not of type BOOLEAN
96 operand not applicable to (unary) +
97 operand not applicable to (unary) –
98 operand of ~ is not of type BOOLEAN
99 ASSERT fault
100 incompatible operands of dyadic operator
101 operand type inapplicable to *
102 operand type inapplicable to /
103 operand type inapplicable to DIV
104 operand type inapplicable to MOD
105 operand type inapplicable to +
106 operand type inapplicable to –
107 operand type inapplicable to = or #
108 operand type inapplicable to relation
109 overriding method must be exported
110 operand is not a type
111 operand inapplicable to (this) function
112 operand is not a variable
113 incompatible assignment
114 string too long to be assigned
115 parameter doesn't match

116 number of parameters doesn't match
117 result type doesn't match
118 export mark doesn't match with forward declaration
119 redefinition textually precedes procedure bound to base type
120 type of expression following IF, WHILE, UNTIL or ASSERT is not BOOLEAN
121 called object is not a procedure (or is an interrupt procedure)
122 actual VAR–parameter is not a variable
123 type of actual parameter is not identical with that of formal VAR–parameter
124 type of result expression differs from that of procedure
125 type of case expression is neither INTEGER nor CHAR
126 this expression cannot be a type or a procedure
127 illegal use of object
128 unsatisfied forward reference
129 unsatisfied forward procedure
130 WITH clause does not specify a variable
131 LEN not applied to array
132 dimension in LEN too large or negative
135 SYSTEM not imported

150 key inconsistency of imported module
151 incorrect symbol file
152 symbol file of imported module not found
153 object or symbol file not opened (disk full?)
154 recursive import not allowed
155 generation of new symbol file not allowed
156 parameter file not found
157 syntax error in parameter file

2. Limitations of the implementation

200 not yet implemented
201 lower bound of set range greater than higher bound
202 set element greater than MAX(SET) or less than 0
203 number too large
204 product too large
205 division by zero
206 sum too large
207 difference too large
208 overflow in arithmetic shift

209 case range too large
213 too many cases in case statement
218 illegal value of parameter (0 <= p < 256)
219 machine registers cannot be accessed
220 illegal value of parameter
221 too many pointers in a record
222 too many global pointers
223 too many record types
224 too many pointer types
225 address of pointer variable too large (move forward in text)
226 too many exported procedures
227 too many imported modules
228 too many exported structures
229 too many nested records for import
230 too many constants (strings) in module
231 too many link table entries (external procedures)
232 too many commands in module
233 record extension hierarchy too high
234 export of recursive type not allowed
240 identifier too long
241 string too long
242 address overflow
244 cyclic type definition not allowed
245 guarded pointer variable may be manipulated by non–local operations; use auxiliary pointer variable

3. Compiler Warnings

301 implicit type cast
306 inappropriate symbol file ignored

4. Run–time Error Messages

SYSTEM-halt
0 silent HALT(0)
1..255 HALT(n), cf. SYSTEM-halt
–1 assertion failed, cf. SYSTEM-assert
–2 invalid array index
–3 function procedure without RETURN statement
–4 invalid case in CASE statement

–5	type guard failed
–6	implicit type guard in record assignment failed
–7	invalid case in WITH statement
–8	value out of range
–9	(delayed) interrupt
–10	NIL access
–11	alignment error
–12	zero divide
–13	arithmetic overflow/underflow
–14	invalid function argument
–15	internal error

5. Unix signals

1	
2	interrupt signal
3	quit signal
4	invalid instruction, HALT
5	
6	
7	
8	arithmetic exception: division by zero, overflow, fpu error
9	
10	bus error, unaligned data access
11	segmentation violation, NIL–access
12	
13	access to closed pipe

```
Edit.Print "1:lp" *\p 1\s 1 4\p n ~
Edit.Print "none" *\p f\p -3\a\f Times12.Scn.Fnt~
Edit.Print none  *\p f\p -3\a\s -3 96\f Times12.Scn.Fnt~

Edit.Print "1:lp" *\p 1\s 1 4\i 1\p n ~
Edit.Print "2:lp" *\p 1\s 2 4\i 2\p n ~
Edit.Print "1:lp" *\p f\p -3\a\s 81 96\i 1\f Times12.Scn.Fnt~
Edit.Print "2:lp"  *\p f\p -3\a\s 82 96\i 2\f Times12.Scn.Fnt~
```