

Part II

Development Tools

We describe the set of elements of the environment included in the language distribution. There one finds different compilers, numerous libraries, program analysis tools, lexical and syntactic analysis tools, and an interface with the C language.

Objective Caml is a compiled language offering two types of code generation:

1. *bytecode* to be executed by a *virtual machine*;
2. *native code* to be executed directly by a microprocessor.

The Objective Caml toplevel uses bytecode to execute the phrases submitted to it. It constitutes the primary development aid, offering the possibility of rapid typing, compilation and testing of function definitions. Moreover, it offers a trace mechanism visualizing parameter values and return values of functions.

The other usual development tools are supplied by the distribution as well: file dependency computation, debugging and profiling. The debugger allows one to execute programs step-by-step, use breakpoints and inspect values. The profiling tool gives measurements of the number of calls or the amount of time spent in a particular function or a particular part of the code. These two tools are only available for Unix platforms.

The richness of a language derives from its core but also from the libraries, sets of reusable programs, which come with it. Objective Caml is no exception to the rule. We have already portrayed to a large extent the graphical library that comes with the distribution. There are many others which we will describe. Libraries bring new functionality to the language, but they are not without drawbacks. In particular, they can present some difficulty vis-a-vis the type discipline.

However rich a language's set of libraries may be, it will always be necessary that it be able to communicate with another language. The Objective Caml distribution includes an interface with the C language allowing Objective Caml to call C functions or be called by them. The difficulty of understanding and implementing this interface lies in the fact that the memory models of Objective Caml and C are different. The essential reason for this difference is that an Objective Caml program includes a garbage collection mechanism.

C as well as Objective Caml allow dynamic memory allocation, and thus fine control over space according to the needs of a program. This only makes sense if unused space can be reclaimed for other use during the course of execution. Garbage collection frees the programmer from responsibility for managing deallocation, a frequent source of execution errors. This feature constitutes one of the safety elements of the Objective Caml language.

However, this mechanism has an impact on the representation of data. Also, knowledge of the guiding principles of memory management is indispensable in order to use communication between the Objective Caml world and the C world correctly.

Chapter 7 presents the basic elements of the Objective Caml system: virtual machine, compilers, and execution library. It describes the language's different compilation modes and compares their portability and efficiency.

Chapter 8 gives a bird's-eye view of the set of predefined types, functions, and exceptions that come with the system distribution. It does not do away with the need to read the reference manual ([LRVD99]) which describes these libraries very well. On the contrary it focuses on the new functionalities supplied by some of them. In particular we may mention output formatting, persistence of values and interfacing with the operating system.

Chapter 9 presents different garbage collection methods in order to then describe the mechanism used by Objective Caml.

Chapter 10 presents debugging tools for Objective Caml programs. Although still somewhat frustrating in some respects, these tools quite often allow one to understand why a program does not work.

Chapter 11 describes the language's different approaches to lexical and syntactic analysis problems: a regular expression library, the `ocamllex` and `ocamlyacc` tools, but also the use of streams.

Chapter 12 describes the interface with the C language. It is no longer possible for a language to be completely isolated from other languages. This interface lets an Objective Caml program call a C function, while passing it values from the Objective Caml world, and vice-versa. The main difficulty with this interface stems from the memory model. For this reason it is recommended that you read the 9 chapter beforehand.

Chapter 13 covers two applications: an improved graphics library based on a hierarchical model of graphical components inspired by the JAVA AWT²; and a classic program to find least-cost paths in a graph using our new graphical interface as well as a cache memory mechanism.

7

Compilation and Portability

The transformation from human readable source code to an executable requires a number of steps. Together these steps constitute the process of *compilation*. The compilation process produces an abstract syntax tree (for an example, see page 159) and a sequence of instructions for a cpu or virtual machine. In Objective Caml, the product of compilation is linked with the Objective Caml *runtime library*. The library is provided with the compiler distribution and is adapted to different host environments (operating system and CPU). The runtime library contains primitive functions such as operations over numbers, the interface to the operating system, and memory management.

Objective Caml has two compilers. The first compiler produces *bytecode* for the Objective Caml virtual machine. The second compiler generates instructions for a number of “real” processors, such as the INTEL, MOTOROLA, SPARC, HP-PA, POWER-PC and ALPHA CPUs. The Objective Caml bytecode compiler produces compact portable code, while the native-code compiler generates high performance architecture dependent code. The Objective Caml toplevel system, which appeared in the first part of this book, uses the bytecode compiler; each user input is compiled and executed in the symbolic environment defined by the current interactive session.

Chapter Overview

This chapter presents the different ways to compile an Objective CAML program and compares their portability and efficiency. The first section explains the different steps of Objective Caml compilation. The second section describes the different types of compilation and the syntax for the production of executables. The third section shows how to construct standalone executables - programs which are independent of an installation of the Objective Caml system. Finally the fourth section compares the different types of compilation with respect to portability and efficiency of execution.

Steps of Compilation

An executable file is obtained by translating and linking as described in figure 7.1.

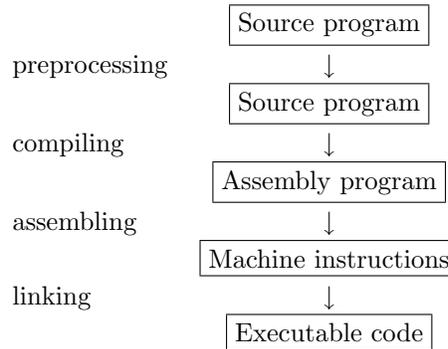


Figure 7.1: Steps in the production of an executable.

To start off, preprocessing replaces certain pieces of text by other text according to a system of macros. Next, compilation translates the source program into assembly instructions, which are then converted to machine instructions. Finally, the linking process establishes a connection to the operating system for primitives. This includes adding the runtime library, which mainly consists of memory management routines.

The Objective Caml Compilers

The code generation phases of the Objective Caml compiler are detailed in figure 7.2. The internal representation of the code generated by the compiler is called an intermediate language (IL).

The lexical analysis stage transforms a sequence of characters to a sequence of lexical elements. These lexical entities correspond principally to integers, floating point numbers, characters, strings of characters and identifiers. The message `Illegal character` might be generated by this analysis.

The parsing stage constructs a syntax tree and verifies that the sequence of lexical elements is correct with respect to the grammar of the language. The message `Syntax error` indicates that the phrase analyzed does not follow the grammar of the language.

The semantic analysis stage traverses the syntax tree, checking another aspect of program correctness. The analysis consists principally of type inference, which if successful, produces the *most general type* of an expression or declaration. Type error messages may occur during this phase. This stage also detects whether any members of a sequence are not of type *unit*. Other warnings may result, including pattern matching analy-

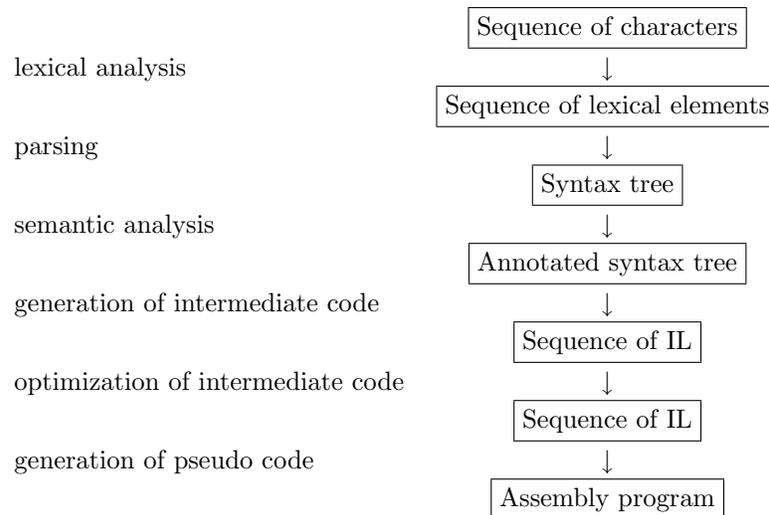


Figure 7.2: Compilation stages.

sis (e.g pattern matching is not exhaustive, part of pattern matching will not be used).

Generation and the optimization of intermediate code does not produce errors or warning messages.

The final step in the compilation process is the generation of a program binary. Details differ from compiler to compiler.

Description of the Bytecode Compiler

The Objective Caml virtual machine is called *Zinc* (“*Zinc Is Not Caml*”). Originally created by Xavier Leroy, *Zinc* is described in ([Ler90]). *Zinc*’s name was chosen to indicate its difference from the first implementation of Caml on the virtual machine CAM (Categorical Abstract Machine, see [CCM87]).

Figure 7.3 depicts the bytecode compiler. The first part of this figure shows the Zinc machine interpreter, linked to the runtime library. The second part corresponds to the Objective Caml bytecode compiler which produces instructions for the Zinc machine. The third part contains the set of libraries that come with the compiler. They will be described in Chapter 8. Standard compiler graphical notation is used for describing the components in figure 7.3. A simple box represents a file written in the language indicated in the box. A double box represents the interpretation of a language by a program written in another language. A triple box indicates that a source language is compiled to a machine language by using a compiler written in a third language. Figure 7.4 gives the legend of each box.

The legend of figure 7.3 is as follows:

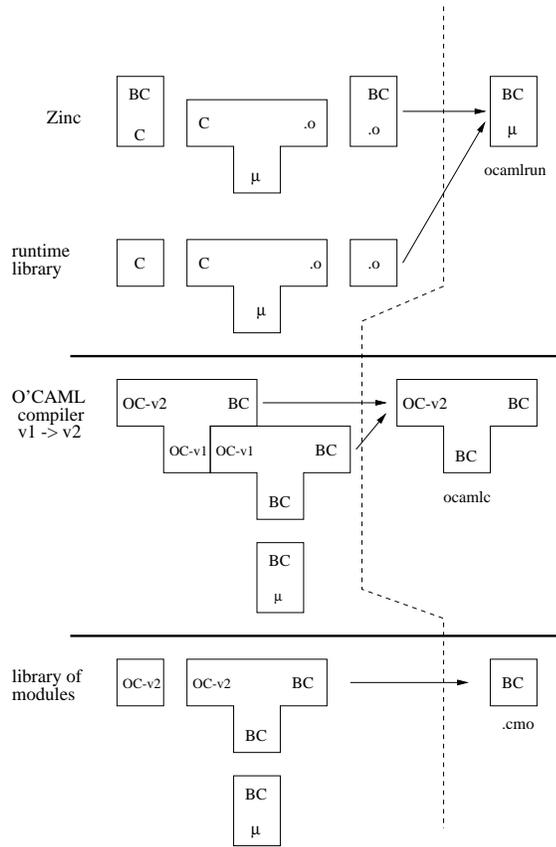


Figure 7.3: Virtual machine.

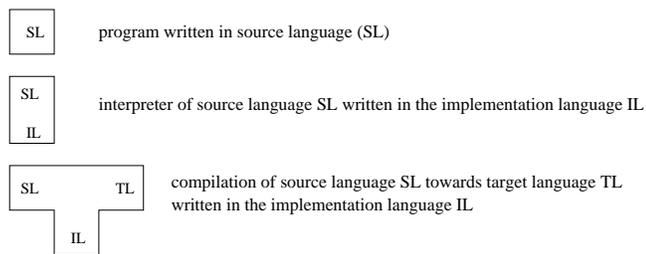


Figure 7.4: Graphical notation for interpreters and compilers.

- BC : Zinc bytecode;
- C : C code;
- .o : object code
- μ : micro-processor;

- OC (v1 or v2) : Objective Caml code.

Note

The majority of the Objective Caml compiler is written in Objective Caml. The second part of figure 7.3 shows how to pass from version v1 of a compiler to version v2.

Compilation

The distribution of a language depends on the processor and the operating system. For each architecture, a distribution of Objective Caml contains the toplevel system, the bytecode compiler, and in most cases a native compiler.

Command Names

The figure 7.5 shows the command names of the different compilers in the various Objective Caml distributions. The first four commands are available for all distributions.

<code>ocaml</code>	toplevel loop
<code>ocamlrun</code>	bytecode interpreter
<code>ocamlc</code>	bytecode batch compiler
<code>ocamlopt</code>	native code batch compiler
<code>ocamlc.opt</code>	optimized bytecode batch compiler
<code>ocamlopt.opt</code>	optimized native code batch compiler
<code>ocamlmktop</code>	new toplevel constructor

Figure 7.5: Commands for compiling.

The optimized compilers are themselves compiled with the Objective Caml native compiler. They compile faster but are otherwise identical to their unoptimized counterparts.

Compilation Unit

A compilation unit corresponds to the smallest piece of an Objective Caml program that can be compiled. For the interactive system, the unit of compilation corresponds to a phrase of the language. For the batch compiler, the unit of compilation is two files: the source file, and the interface file. The interface file is optional - if it does not exist, then all global declarations in the source file will be visible to other compilation units. The construction of interface files is described in the chapter on module programming (see chapter 14). The two file types (source and interface) are differentiated by separate file extensions.

Naming Rules for File Extensions

Figure 7.6 presents the extensions of different files used for Objective CAML and C programs.

extension	meaning
<code>.ml</code>	source file
<code>.mli</code>	interface file
<code>.cmo</code>	object file (bytecode)
<code>.cma</code>	library object file (bytecode)
<code>.cmi</code>	compiled interface file
<code>.cmx</code>	object file (native)
<code>.cmxa</code>	library object file (native)
<code>.c</code>	C source file
<code>.o</code>	C object file (native)
<code>.a</code>	C library object file (native)

Figure 7.6: File extensions.

The files `example.ml` and `example.mli` form a compilation unit. The compiled interface file (`example.cmi`) is used for both the bytecode and native code compiler. The C language related files are used when integrating C code with Objective Caml code. (see chapter 12).

The Bytecode Compiler

The general form of the batch compiler commands are:

```
command options file_name
```

For example:

```
ocamlc -c example.ml
```

The command-line options for both the native and bytecode compilers follow typical Unix conventions. Each option is prefixed by the character `-`. File extensions are interpreted in the manner described by figure 7.6. In the above example, the file `example.ml` is considered an Objective Caml source file and is compiled. The compiler will produce the files `example.cmo` and `example.cmi`. The option `-c` informs the compiler to generate individual object files, which may be linked at a later time. Without this option, the compiler will produce an executable file named `a.out`.

The table in figure 7.7 describes the principal options of the bytecode compiler. The table in figure 7.8 indicates other possible options.

Principal options	
<code>-a</code>	construct a runtime library
<code>-c</code>	compile without linking
<code>-o <i>name_of_executable</i></code>	specify the name of the executable
<code>-linkall</code>	link with all libraries used
<code>-i</code>	display all compiled global declarations
<code>-pp <i>command</i></code>	uses <i>command</i> as preprocessor
<code>-unsafe</code>	turn off index checking
<code>-v</code>	display the version of the compiler
<code>-w <i>list</i></code>	choose among the <i>list</i> the level of warning message (see fig. 7.9)
<code>-impl <i>file</i></code>	indicate that <i>file</i> is a Caml source (.ml)
<code>-intf <i>file</i></code>	indicate that <i>file</i> is a Caml interface (.mli)
<code>-I <i>directory</i></code>	add <i>directory</i> in the list of directories

Figure 7.7: Principal options of the bytecode compiler.

Other options	
light process	<code>-thread</code> (see chapter 19, page 599)
linking	<code>-g</code> , <code>-noassert</code> (see chapter 10, page 271)
standalone executable	<code>-custom</code> , <code>-cclib</code> , <code>-ccopt</code> , <code>-cc</code> (see page 207)
<i>runtime</i>	<code>-make-runtime</code> , <code>-use-runtime</code>
C interface	<code>-output-obj</code> (see chapter 12, page 315)

Figure 7.8: Other options for the bytecode compiler.

To display the list of bytecode compiler options, use the option `-help`.

The different levels of warning message are described in figure 7.9. A message level is a switch (enable/disable) represented by a letter. An upper case letter activates the level and a lower case letter disables it.

Principal levels	
A/a	enable/disable all messages
F/f	partial application in a sequence
P/p	for incomplete pattern matching
U/u	for missing cases in pattern matching
X/x	enable/disable all other messages
for hidden object	M/m and V/v (see chapter 15)

Figure 7.9: Description of compilation warnings.

By default, the highest level (A) is chosen by the compiler.

Example usage of the bytecode compiler is given in figure 7.10.

```

❏ xterm
bou: cat t.ml
let f x = x + 1;;
print_int (f 18);;
print_newline();;
bou: ocamlc -i -custom -o tb.exe t.ml
val f : int -> int

bou: ./tb.exe
19

```

Figure 7.10: Session with the bytecode compiler.

Native Compiler

The native compiler has behavior similar to the bytecode compiler, but produces different types of files. The compilation options are generally the same as those described in figures 7.7 and 7.8. It is necessary to take out the options related to *runtime* in figure 7.8. Options specific to the native compiler are given in figure 7.11. The different *warning* levels are same.

<code>-compact</code>	optimize the produced code for space
<code>-S</code>	keeps the assembly code in a file
<code>-inline level</code>	set the aggressiveness of inlining

Figure 7.11: Options specific to the native compiler.

Inlining is an elaborated version of macro-expansion in the preprocessing stage. For functions whose arguments are fixed, inlining replaces each function call with the body of the function called. Several different calls produce several copies of the function body. Inlining avoids the overhead that comes with function call setup and return, at the expense of object code size. Principal inlining levels are:

- 0 : The expansion will be done only when it will not increase the size of the object code.
- 1 : This is the default value; it accepts a light increase on code size.
- $n > 1$: Raise the tolerance for growth in the code. Higher values result in more inlining.

Toplevel Loop

The toplevel loop provides only two command line options.

- `-I directory`: adds the indicated directory to the list of search paths for compiled source files.
- `-unsafe`: instructs the compiler not to do bounds checking on array and string accesses.

The toplevel loop provides several directives which can be used to interactively modify its behavior. They are described in figure 7.12. All these directives begin with the character `#` and are terminated by `;;`.

<code>#quit ;;</code>	quit from the toplevel interaction
<code>#directory <i>directory</i> ;;</code>	add the directory to the search path
<code>#cd <i>directory</i> ;;</code>	change the working directory
<code>#load <i>object_file</i> ;;</code>	load an object file (<code>.cmo</code>)
<code>#use <i>source_file</i> ;;</code>	compile and load a source file
<code>#print_depth <i>depth</i> ;;</code>	modify the depth of printing
<code>#print_length <i>width</i> ;;</code>	modify the length of printing
<code>#install_printer <i>function</i> ;;</code>	specify a printing function
<code>#remove_printer <i>function</i> ;;</code>	remove a printing function
<code>#trace <i>function</i> ;;</code>	trace the arguments of the function
<code>#untrace <i>function</i> ;;</code>	stop tracing the function
<code>#untrace_all ;;</code>	stop all tracing

Figure 7.12: Toplevel loop directives.

The directives dealing with directories respect the conventions of the operating system used.

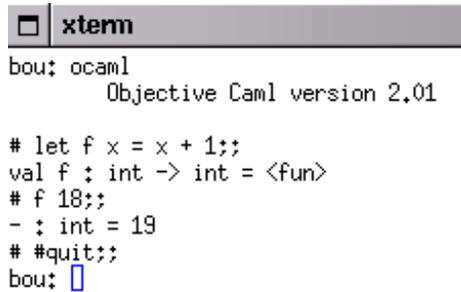
The loading directives do not have exactly the same behavior. The directive `#use` reads the source file as if it was typed directly in the toplevel loop. The directive `#load` loads the file with the extension `.cmo`. In the later case, the global declarations of this file are not directly accessible. If the file `example.ml` contains the global declaration `f`, then once the bytecode is loaded (`#load "example.cmo";;`), it is assumed that the value of `f` could be accessed by `Example.f`, where the first letter of the file is capitalized. This notation comes from the module system of Objective Caml (see chapter 14, page 405).

The directives for the depth and width of printing are used to control the display of values. This is useful when it is necessary to display the contents of a value in detail.

The directives for printer redefinition are used to install or remove a user defined printing function for values of a specified type. In order to integrate these printer functions into the default printing procedure, it is necessary to use the `Format` library(see chapter 8) for the definition.

The directives for tracing arguments and results of functions are particularly useful for debugging programs. They will be discussed in the chapter on program analysis (see chapter 10).

Figure 7.13 shows a session in the toplevel loop.



```

xterm
bou: ocaml
      Objective Caml version 2.01

# let f x = x + 1;;
val f : int -> int = <fun>
# f 18;;
- : int = 19
# #quit;;
bou: 

```

Figure 7.13: Session with the toplevel loop.

Construction of a New Interactive System

The command `ocamlmktop` can be used to construct a new toplevel executable which has specific library modules loaded by default. For example, `ocamlmktop` is often used for pulling native object code libraries (typically written in C) into a new toplevel.

`ocamlmktop` options are a subset of those used by the bytecode compiler (`ocamlc`):

```
-cclib libname, -ccoct option, -custom, -I directory -o executable_name
```

The chapter on graphics programming (see chapter 5, page 117) uses this command for constructing a toplevel system containing the `Graphics` library in the following manner:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib \
-I/usr/X11/lib -cclib -lX11
```

This command constructs an executable with the name `mytoplevel`, containing the bytecode library `graphics.cma`. This standalone executable (`-custom`, see the following section) will be linked to the library `X11 (libX11.a)` which in turn will be looked up in the path `/usr/X11/lib`.

Standalone Executables

A standalone executable is a program that does not depend on an Objective Caml installation to run. This facilitates the distribution of binary applications and robustness against runtime library changes across Objective Caml versions.

The Objective Caml native compiler produces standalone executables by default. But without the `-custom` option, the bytecode compiler produces an executable which requires the bytecode interpreter `ocamlrun`. Imagine the file `example.ml` is as follows:

```
let f x = x + 1;;
print_int (f 18);;
print_newline();;
```

Then the following command produces the (approximately 8k) file `example.exe`:

```
ocamlc -o example.exe example.ml
```

This file can be executed by the Objective Caml bytecode interpreter:

```
$ ocamlrun example.exe
19
```

The interpreter executes the Zinc machine instructions contained in the file `example.exe`.

Under Unix, the first line of the file `example.exe` contains the location of the interpreter, for example:

```
#!/usr/local/bin/ocamlrun
```

This means the file can be executed directly (without using `ocamlrun`). Like a shell-script, executing the file in turn runs the program specified on the first line, which is then used to interpret the remainder of the file. If `ocamlrun` can't be found, execution will fail and the error message `Command not found` will be displayed.

The same compilation with the option `-custom` produces a standalone executable with name `exauto.exe`:

```
ocamlc -custom -o exauto.exe example.ml
```

This time the file is about 85K, as it contains the Zinc interpreter as well as the program bytecode. This file can be executed directly or copied to another machine (using the same CPU/Operating System) for execution.

Portability and Efficiency

One reason to compile to an abstract machine is to produce an executable independent of the architecture of the real machine where it runs. A native compiler will produce more efficient code, but the binary can only be executed on the architecture it was compiled for.

Standalone Files and Portability

To produce a standalone executable, the bytecode compiler links the bytecode object file `example.cmo` with the runtime library, the bytecode interpreter and some C code. It is assumed that there is a C compiler on the host system. The inclusion of machine code means that stand-alone bytecode executables are not portable to other systems or other architectures.

This is not the case for the non-standalone version. Since the Zinc machine is not included, the only things generated are the platform independent bytecode instructions. Bytecode programs will run on any platform that has the interpreter. `Ocamlrun` is part of the default Objective Caml distribution for Sparc running SOLARIS, INTEL running Windows, etc. It is always preferable to use the same version of interpreter and compiler.

The portability of bytecode object files makes it possible to directly distribute Objective Caml libraries in bytecode form.

Efficiency of Execution

The bytecode compiler produces a sequence of instructions for the Zinc machine, which at the moment of the execution, will be interpreted by `ocamlrun`. Interpretation has a moderately negative linear effect on speed of execution. It is possible to view Zinc's bytecode interpretation as a big pattern matching machine (matching `match ... with`) where each instruction is a trigger and the computation branch modifies the stack and the counter (address of the next instruction).

Without testing all parts of the language, the following small example which computes Fibonacci numbers shows the difference in execution time between the bytecode compiler and the native compiler. Let the program `fib.ml` as follows:

```
let rec fib n =
  if n < 2 then 1
  else (fib (n-1)) + (fib(n-2));;
```

and the following program `main.ml` as follows:

```
for i = 1 to 10 do
```

```
    print_int (Fib.fib 30);  
    print_newline()  
done;;
```

Their compilation is as follows:

```
$ ocamlc -o fib.exe fib.ml main.ml  
$ ocamlc -o fibopt.exe fib.ml main.ml
```

These commands produce two executables: `fib.exe` and `fibopt.exe`. Using the Unix command `time` in Pentium 350 under Linux, we get the following data:

<code>fib.exe</code> (bytecode)	<code>fibopt.exe</code> (native)
7 s	1 s

This corresponds to a factor 7 between the two versions of the same program. This program does not test all characteristics of the language. The difference depends heavily on the type of application, and is typically much smaller.

Exercises

Creation of a Toplevel and Standalone Executable

Consider again the Basic interpreter. Modify it to make a new toplevel.

1. Split the Basic application into 4 files, each with the extension `.ml`. The files will be organized like this: abstract syntax (`syntax.ml`), printing (`pprint.ml`), parsing (`alxsynt.ml`) and evaluation of instructions (`eval.ml`). The head of each file should contain the open statements to load the modules required for compilation.
2. Compile all files separately.
3. Add a file `mainbasic.ml` which contains only the statement for calling the main function.
4. Create a new toplevel with the name `topbasic`, which starts the Basic interpreter.
5. Create a standalone executable which runs the Basic interpreter.

Comparison of Performance

Try to compare the performance of code produced by the bytecode compiler and by the native compiler. For this purpose, write an application for sorting lists and arrays.

1. Write a polymorphic function for sorting lists. The order relation should be passed as an argument to the sort function. The sort algorithm can be selected by the reader. For example: bubble sort, or quick sort. Write this function as `sort.ml`.
2. Create the main function in the file `trilist.ml`, which uses the previous function and applies it to a list of integers by sorting it in increasing order, then in decreasing order.
3. Create two standalone executables - one with the bytecode compiler, and another with the native compiler. Measure the execution time of these two programs. Choose lists of sufficient size to get a good idea of the time differences.
4. Rewrite the sort program for arrays. Continue using an order function as argument. Perform the test on arrays filled in the same manner as for the lists.
5. What can we say about the results of these tests?

Summary

This chapter has shown the different ways to compile an Objective Caml program. The bytecode compiler is favorable for portable code, allowing for the system independent distribution of programs and libraries. This property is lost in the case of standalone bytecode executables. The native compiler trades producing efficient architecture dependent code for a loss of portability.

To Learn More

The techniques to compile for abstract machines were used in the first generation of SmallTalk, then in the functional languages LISP and ML. The argument that the use of abstract machines will hinder performance has put a shadow on this technique for a long time. Now, the JAVA language has shown that the opposite is true. An abstract machine provides several advantages. The first is to facilitate the porting of a compiler to different architectures. The part of the compiler related to portability has been well defined (the abstract machine interpreter and part of runtime library). Another benefit of this technique is portable code. It is possible to compile an application on one architecture and execute it on another. Finally, this technique simplifies compiler construction by adding specific instructions for the type of language to compile. In the case of functional languages, the abstract machines make it easy to create the closures (packing environment and code together) by adding the notion of execution environment to the abstract machine.

To compensate for the loss in efficiency caused by the use of the bytecode interpreter, one can expand the set of abstract machine instructions to include those of a real machine at runtime. This type of expansion has been found in the implementation of Lisp (llm3) and JAVA (JIT). The performance increases, but does not reach the level of a native C compiler.

One difficulty of functional language compilation comes from closures. They contain both the executable code and execution environment (see page 23).

The choice of implementation for the environment and the access of values in the environment has a significant influence on the performance of the code produced. An important function of the environment consists of obtaining access to values in constant time; the variables are viewed as indexes in an array containing their values. This requires the preprocessing of functional expressions. An example can be found in L. Cardelli's book - *Functional Abstract Machine*. Zinc uses this technique. Another crucial optimization is to avoid the construction of useless closures. Although all functions in ML can be viewed as functions with only one argument, it is necessary to not create intermediate closures in the case of application on several arguments. For example, when the function `add` is applied with two integers, it is not useful to create the first closure corresponding to the function of applying `add` to the first argument. It is necessary to note that the creation of a closure would allocate certain memory space for the environment and would require the recovery of that memory space in the future (see chapter 9). Automatic memory recovery is the second major performance concern, along with environment.

Finally, bootstrapping allows us to write the majority of a compiler with the same language which it is going to compile. For this reason, like the chicken and the egg, it is necessary to define the minimal part of the language which can be expanded later. In fact, this property is hardly appreciable for classifying the languages and their implementations. This property is also used as a measure of the capability of a language to be used in the implementation of a compiler. A compiler is a large program, and bootstrapping is a good test of its correctness and performance. The following are links to the references:

Link: <http://caml.inria.fr/camlstone.txt>

At that time, Caml was compiled over fifty machines, these were antecedent versions of Objective Caml. We can get an idea of how the present Objective Caml has been improved since then.

