Developing Applications With

Objective Caml

Emmanuel Chailloux Pascal Manoury Bruno Pagano

Developing Applications With Objective Caml

Translated by

Francisco Albacete • Mark Andrew • Martin Anlauf • Christopher Browne • David Casperson • Gang Chen • Harry Chomsky • Ruchira Datta • Seth Delackner • Patrick Doane • Andreas Eder • Manuel Fahndrich • Joshua Guttman • Theo Honohan • Xavier Leroy • Markus Mottl • Alan Schmitt • Paul Steckler • Perdita Stevens • François Thomasset

Éditions O'REILLY
18 rue Séguier
75006 Paris
FRANCE
france@oreilly.com
<url:http://www.editions-oreilly.fr/>

O'REILLY®

Cambridge • Cologne • Farnham • Paris • Pékin • Sebastopol • Taipei • Tokyo

The original edition of this book (ISBN 2-84177-121-0) was published in France by O'REILLY & Associates under the title *Dveloppement d'applications avec Objective Caml*.

Historique :

- Version 19990324???????????

Cover concept by Ellie Volckhausen.

Édition : Xavier Cazin.

ISBN

Preface

The desire to write a book on Objective Caml sprang from the authors' pedagogical experience in teaching programming concepts through the Objective Caml language. The students in various majors and the engineers in continuing education at Pierre and Marie Curie University have, through their dynamism and their critiques, caused our presentation of the Objective Caml language to evolve greatly. Several examples in this book are directly inspired by their projects.

The implementation of the Caml language has been ongoing for fifteen years. Its development comes from the Formel and then Cristal projects at INRIA, in collaboration with Denis Diderot University and the École Normale Supérieure. The continuous efforts of the researchers on these teams, as much to develop the theoretical underpinnings as the implementation itself, have produced over the span of years a language of very high quality. They have been able to keep pace with the constant evolution of the field while integrating new programming paradigms into a formal framework. We hope through this exposition to contribute to the widespread diffusion which this work deserves.

The form and the foundation of this book wouldn't be what they are without the help of numerous colleagues. They were not put off by rereading our first manuscripts. Their remarks and their comments have allowed this exposition to improve throughout the course of its development. We wish particularly to thank María-Virginia Aponte, Sylvain Baro, Christian Codognet, Hélène Cottier, Guy Cousineau, Pierre Crégut, Titou Durand, Christophe Gonzales, Michelle Morcrette, Christian Queinnec, Attila Raksany and Didier Rémy.

The HTML version of this book would not have seen the light of day without the tools `hevea` and `VideoC`. A big thank you to their respective authors, Luc Maranget and Christian Queinnec, who have always responded in the briefest intervals to our questions and our demands for changes.

Contents

Introduction

Objective Caml is a programming language. One might ask why yet another language is needed. Indeed there are already numerous existing languages with new ones constantly appearing. Beyond their differences, the conception and genesis of each one of them proceeds from a shared motivation: the desire to abstract.

To abstract from the machine In the first place, a programming language permits one to neglect the "mechanical" aspect of the computer; it even lets one forget the microprocessor model or the operating system on which the program will be executed.

To abstract from the operational model The notion of function which most languages possess in one form or another is borrowed from mathematics and not from electronics. In a general way, languages substitute formal models for purely computational viewpoints. Thus they gain *expressivity*.

To abstract errors This has to do with the attempt to guarantee execution safety; a program shouldn't terminate abruptly or become inconsistent in case of an error. One of the means of attaining this is strong static typing of programs and having an exception mechanism in place.

To abstract components (I) Programming languages make it possible to subdivide an application into different software components which are more or less independent and autonomous. Modularity permits higher-level structuring of the whole of a complex application.

To abstract components (II) The existence of programming units has opened up the possibility of their reuse in contexts other than the ones for which they were developed. *Object-oriented* languages constitute another approach to reusability permitting rapid prototyping.

Objective Caml is a recent language which takes its place in the history of programming languages as a distant descendant of Lisp, having been able to draw on the lessons

of its cousins while incorporating the principal characteristics of other languages. It is developed at INRIA[1] and is supported by long experience with the conception of the languages in the ML family. Objective Caml is a general-purpose language for the expression of symbolic and numeric algorithms. It is object-oriented and has a parameterized module system. It supports the development of concurrent and distributed applications. It has excellent execution safety thanks to its static typing, its exception mechanism and its garbage collector. It is high-performance while still being portable. Finally, a rich development environment is available.

Objective Caml has never been the subject of a presentation to the "general public". This is the task to which the authors have set themselves, giving this exposition three objectives:

1. To describe in depth the Objective Caml language, its libraries and its development environment.
2. To show and explain what are the concepts hidden behind the programming styles which can be used with Objective Caml.
3. To illustrate through numerous examples how Objective Caml can serve as the development language for various classes of applications.

The authors' goal is to provide insight into how to choose a programming style and structure a program, consistent with a given problem, so that it is maintainable and its components are reusable.

Description of the language

Objective Caml is a functional language: it manipulates functions as values in the language. These can in turn be passed as arguments to other functions or returned as the result of a function call.

Objective Caml is statically typed: verification of compatibility between the types of formal and actual parameters is carried out at program compilation time. From then on it is not necessary to perform such verification during the execution of the program, which increases its efficiency. Moreover, verification of typing permits the elimination of most errors introduced by typos or thoughtlessness and contributes to execution safety.

Objective Caml has parametric polymorphism: a function which does not traverse the totality of the structure of one of its arguments accepts that the type of this argument is not fully determined. In this case this parameter is said to be *polymorphic*. This feature permits development of generic code usable for different data structures,

1. Institut National de Recherche en Informatique et Automatique (National Institute for Research in Automation and Information Technology).

such that the exact representation of this structure need not be known by the code in question. The typing algorithm is in a position to make this distinction.

Objective Caml has type inference: the programmer need not give any type information within the program. The language alone is in charge of deducing from the code the most general type of the expressions and declarations therein. This *inference* is carried out jointly with verification, during program compilation.

Objective Caml is equipped with an exception mechanism: it is possible to interrupt the normal execution of a program in one place and resume at another place thanks to this facility. This mechanism allows control of exceptional situations, but it can also be adopted as a programming style.

Objective Caml has imperative features: I/O, physical modification of values and iterative control structures are possible without having recourse to functional programming features. Mixture of the two styles is acceptable, and offers great development flexibility as well as the possibility of defining new data structures.

Objective Caml executes (threads): the principal tools for creation, synchronization, management of shared memory, and interthread communication are predefined.

Objective Caml communicates on the Internet: the support functions needed to open communication channels between different machines are predefined and permit the development of client-server applications.

Numerous libraries are available for Objective Caml: classic data structures, I/O, interfacing with system resources, lexical and syntactic analysis, computation with large numbers, persistent values, etc.

A programming environment is available for Objective Caml: including interactive toplevel, execution trace, dependency calculation and profiling.

Objective Caml interfaces with the C language: by calling C functions from an Objective Caml program and vice versa, thus permitting access to numerous C libraries.

Three execution modes are available for Objective Caml: interactive by means of an interactive toplevel, compilation to bytecodes interpreted by a virtual machine, compilation to native machine code. The programmer can thus choose between

flexibility of development, portability of object code between different architectures, or performance on a given architecture.

Structure of a program

Development of important applications requires the programmer or the development team to consider questions of organization and structure. In Objective Caml, two models are available with distinct advantages and features.

The parameterized module model: data and procedures are gathered within a single entity with two facets: the code proper, and its interface. Communication between modules takes place via their interface. The description of a type may be hidden, not appearing in the module interface. These abstract data types facilitate modifications of the internal implementation of a module without affecting other modules which use it. Moreover, modules can be parameterized by other modules, thus increasing their reusability.

The object model: descriptions of procedures and data are gathered into entities called *classes*; an object is an instance (value) of a class. Interobject communication is implemented through "message passing", the receiving object determines upon execution (late binding) the procedure corresponding to the message. In this way, object-oriented programming is "data-driven". The program structure comes from the relationships between classes; in particular inheritance lets one class be defined by extending another. This model allows concrete, abstract and parameterized classes. Furthermore, it introduces polymorphism of inclusion by defining the subtyping relationship between classes.

The choice between these two models allows great flexibility in the logical organization of an application and facilitates its maintenance and evolution. There is a duality between these two models. One cannot add data fields to a module type (no extensibility of data), but one can add new procedures (extensibility of procedures) acting on data. In the object model, one can add subclasses of a class (extensibility of data) for dealing with new cases, but one cannot add new procedures visible from the ancestor class (no extensibility of procedures). Nevertheless the combination of the two offers new possibilities for extending data and procedures.

Safety and efficiency of execution

Objective Caml bestows excellent execution safety on its programs without sacrificing their efficiency. Fundamentally, static typing is a guarantee of the absence of run-time type errors and makes useful static information available to the compiler without burdening performance with dynamic type tests. These benefits also extend to the object-oriented language features. Moreover, the built-in garbage collector adds to the safety of the language system. Objective Caml's is particularly efficient. The exception

mechanism guarantees that the program will not find itself in an inconsistent state after a division by zero or an access outside the bounds of an array.

Outline of the book

The present work consists of four main parts, bracketed by two chapters and enhanced by two appendices, a bibliography, an index of language elements and an index of programming concepts.

Chapter 1 : This chapter describes how to install version `2.04` of the Objective Caml language on the most current systems (Windows, Unix and MacOS).

Part I: Core of the language The first part is a complete presentation of the basic elements of the Objective Caml language. Chapter 2 is a dive into the functional core of the language. Chapter 3 is a continuation of the previous one and describes the imperative part of the language. Chapter 4 compares the "pure" functional and imperative styles, then presents their joint use. Chapter 5 presents the graphics library. Chapter 6 exhibits three applications: management of a simple database, a mini-Basic interpreter and a well-known single-player game, minesweeper.

Part II: Development tools The second part of the book describes the various tools for application development. Chapter 7 compares the various compilation modes, which are the interactive toplevel and command-line bytecode and native code compilers. Chapter 8 presents the principal libraries provided with the language distribution. Chapter 9 explains garbage collection mechanisms and details the one used by Objective Caml. Chapter 10 explains the use of tools for debugging and profiling programs. Chapter 11 addresses lexical and syntactic tools. Chapter 12 shows how to interface Objective Caml programs with C. Chapter 13 constructs a library and an application. This library offers tools for the construction of GUIs. The application is a search for least-cost paths within a graph, whose GUI uses the preceding library.

Part III: Organization of applications The third part describes the two ways of organizing a program: with modules, and with objects. Chapter 14 is a presentation of simple and parameterized language modules. Chapter 15 introduces Objective Caml object-oriented extension. Chapter 16 compares these two types of organization and indicates the usefulness of mixing them to increase the extensibility of programs. Chapter 17 describes two substantial applications: two-player games which put to work several parameterized modules used for two different games, and a simulation of a robot world demonstrating interobject communication.

Part IV: Concurrence and distribution The fourth part introduces concurrent and distributed programs while detailing communication between processes, lightweight or not, and on the Internet. Chapter 18 demonstrates the direct link between the language and the system libraries, in particular the notions of process and

communication. Chapter 19 leads to the lack of determinism of concurrent programming while presenting Objective Caml's threads. Chapter 20 discusses interprocess communication via sockets in the distributed memory model. Chapter 21 presents first of all a toolbox for client-server applications. It is subsequently used to extend the robots of the previous part to the client-server model. Finally, we adapt some of the programs already encountered in the form of an HTTP server.

Chapter 22 This last chapter takes stock of application development in Objective Caml and presents the best-known applications of the ML language family.

Appendices The first appendix explains the notion of cyclic types used in the typing of objects. The second appendix describes the language changes present in the new version `3.00`. These have been integrated in all following versions of Objective Caml (`3.xx`).

Each chapter consists of a general presentation of the subject being introduced, a chapter outline, the various sections thereof, statements of exercises to carry out, a summary, and a final section entitled "To learn more" which indicates bibliographic references for the subject which has been introduced.

1

How to obtain Objective Caml

The various programs used in this work are "free" software [1]. They can be found either on the CD-ROM accompanying this work, or by downloading them from the Internet. This is the case for Objective Caml, developed at INRIA.

Description of the CD-ROM

The CD-ROM is provided as a hierarchy of files. At the root can be found the file `index.html` which presents the CD-ROM, as well as the five subdirectories below:

- book: root of the HTML version of the book along with the solutions to the exercises;
- apps: applications described in the book;
- exercises: independent solutions to the proposed exercises;
- distrib: set of distributions provided by INRIA, as described in the next section;
- tools: set of tools for development in Objective Caml;
- docs: online documentation of the distribution and the tools.

To read the CD-ROM, start by opening the file `index.html` in the root using your browser of choice. To access directly the hypertext version of the book, open the file `book/index.html`. This file hierarchy, updated in accordance with readers' remarks, can be found posted on the editor's site:

Link: http://www.oreilly.fr

1. "Free software" is not to be confused with "freeware". "Freeware" is software which costs nothing, whereas "free software" is software whose source is also freely available. In the present case, all the programs used cost nothing and their source is available.

Downloading

Objective Caml can be downloaded via web browser at the following address:

Link: http://caml.inria.fr/ocaml/distrib.html

There one can find binary distributions for Linux (Intel and PPC), for Windows (NT, 95, 98) and for MacOS (7, 8), as well as documentation, in English, in different formats (PDF, PostScript and HTML). The source code for the three systems is available for download as well. Once the desired distribution is copied to one's machine, it's time to install it. This procedure varies according to the operating system used.

Installation

Installing Objective Caml requires about 10MB of free space on one's hard disk drive. The software can easily be uninstalled without corrupting the system.

Installation under Windows

The file containing the binary distribution is called: `ocaml-2.04-win.zip`, indicating the version number (here `2.04`) and the operating system.

Warning Objective Caml only works under recent versions of Windows : Windows 95, 98 and NT. Don't try to install it under Windows 3.x or OS2/Warp.

1. The file is in compressed (`.zip`) format; the first thing to do is decompress it. Use your favorite decompression software for this. You obtain in this way a file hierarchy whose root is named `ocaml`. You can place this directory at any location on your hard disk. It is denoted by `<caml-dir>` in what follows.
2. This directory includes:
   - two subdirectories: `bin` for binaries and `lib` for libraries;
   - two "text" files: `License.txt` and `Changes.txt` containing the license to use the software and the changes relative to previous versions;
   - an application: `OCamlWin` corresponding to the main application;
   - a configuration file: `Ocamlwin.ini` which will need to be modified (see the following point);
   - two files of version notes: the first, `Readme.gen`, for this version and the second, `Readme.win`, for the version under Windows.
3. If you have chosen a directory other than `c:\ocaml` as the root of your file hierarchy, then it is necessary to indicate this in the configuration file. Edit it with `Wordpad` and change the line defining `CmdLine` which is of the form:
   `CmdLine=ocamlrun c:\ocaml\bin\ocaml.exe -I c:\ocaml\lib`
   to

`CmdLine=ocamlrun <caml-dir>\bin\ocaml.exe -I <caml-dir>\lib`
You have to replace the names of the search paths for binaries and libraries with the name of the Objective Caml root directory. If we have chosen `C:\Lang\ocaml` as the root directory (`<caml-dir>`), the modification becomes:
`CmdLine=ocamlrun C:\Lang\ocaml\bin\ocaml.exe -I C:\Lang\ocaml\lib`

4. Copy the file OCamlWin.ini to the main system directory, that is, `C:\windows` or `C:\win95` or `C:\winnt` according to the installation of your system.

Now it's time to test the `OCamlWin` application by double-clicking on it. You'll get the window in figure 1.1.

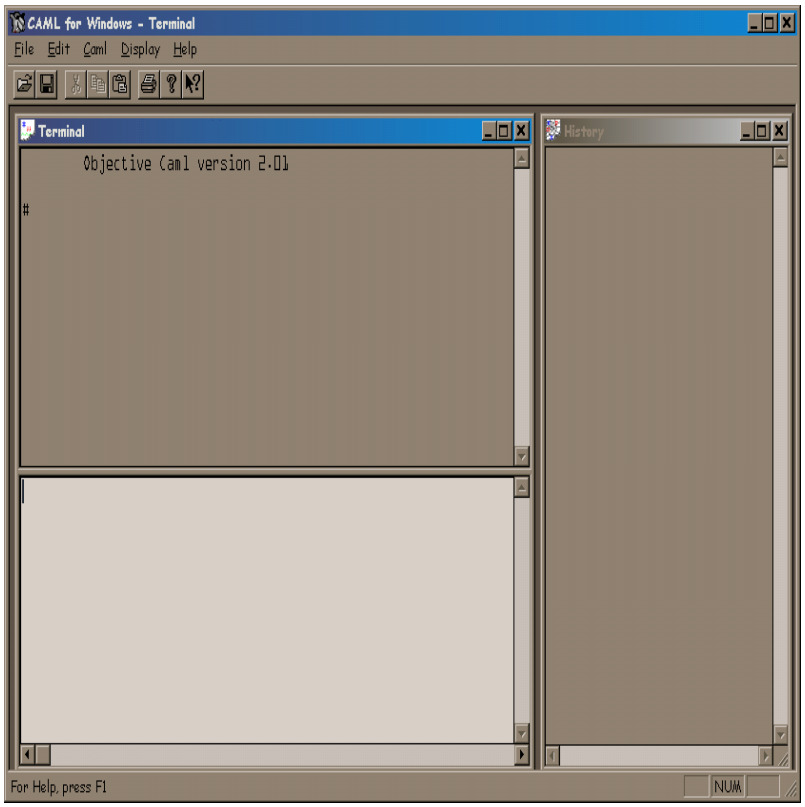


Figure 1.1: Objective Caml window under Windows.

The configuration of command-line executables, launched from a DOS window, is done by modifying the `PATH variable and the Objective Caml` library search path variable (`CAMLLIB`), as follows:

```
PATH=%PATH%;<caml-dir>\bin
set CAMLLIB=<caml-dir>\lib
```

where `<caml-dir>` is replaced by the path where Objective Caml is installed.

These two commands can be included in the `autoexec.bat` file which every good DOS has. To test the command-line executables, type the command `ocaml` in a DOS window. This executes the file:

```
<caml-dir>/bin/ocaml.exe
```

corresponding to the Objective Caml. text mode toplevel. To exit from this command, type `#quit;;`.

To install Objective Caml from source under Windows is not so easy, because it requires the use of commercial software, in particular the Microsoft C compiler. Refer to the file `Readme.win` of the binary distribution to get the details.

Installation under LINUX

The LINUX installation also has an easy-to-install binary distribution in the form of an `rpm`. package. Installation from source is described in section 1. The file to download is: `ocaml-2.04-2.i386.rpm` which will be used as follows with root privileges:

```
rpm -i ocaml-2.04-2.i386.rpm
```

which installs the executables in the `/usr/bin` directory and the libraries in the `/usr/lib/ocaml` directory.

To test the installation, type: `ocamlc -v` which prints the version of Objective Caml installed on the machine.

```
ocamlc -v
The Objective Caml compiler, version 2.04
Standard library directory: /usr/lib/ocaml
```

You can also execute the command `ocaml` which prints the header of the interactive toplevel.

```
        Objective Caml version 2.04

#
```

The `#` character is the prompt in the interactive toplevel. This interactive toplevel can be exited by the `#quit;;` directive, or by typing `CTRL-D`. The two semi-colons indicate the end of an Objective Caml phrase.

Installation under MacOS

The MacOS distribution is also in the form of a self-extracting binary. The file to download is: `ocaml-2.04-mac.sea.bin` which is compressed. Use your favorite software

to decompress it. Then all you have to do to install it is launch the self-extracting archive and follow the instructions printed in the dialog box to choose the location of the distribution. For the MacOS X server distribution, follow the installation from source under Unix.

Installation from source under Unix

Objective Caml can be installed on systems in the Unix family from the source distribution. Indeed it will be necessary to compile the Objective Caml system. To do this, one must either have a C compiler on one's Unix, machine, which is generally the case, or download one such as `gcc` which works on most Unix. systems. The Objective Caml distribution file containing the source is: `ocaml-2.04.tar.gz`. The file `INSTALL` describes, in a very clear way, the various stages of configuring, making, and then installing the binaries.

Installation of the HTML *documentation*

Objective Caml's English documentation is present also in the form of a hierarchy of HTML files which can be found in the `docs` directory of the CD-ROM.

This documentation is a reference manual. It is not easy reading for the beginner. Nevertheless it is quite useful as a description of the language, its tools, and its libraries. It will soon become indispensable for anyone who hopes to write a program of more than ten lines.

Testing the installation

Once installation of the Objective Caml development environment is done, it is necessary to test it, mainly to verify the search paths for executables and libraries. The simplest way is to launch the interactive toplevel of the system and write the first little program that follows:

```
String.concat "/" ["a"; "path"; "here"] ;;
```

This expression concatenates several character strings, inserting the "/" character between each word. The notation `String.concat` indicates use of the function `concat` from the `String`. If the library search path is not correct, the system will print an error. It will be noted that the system indicates that the computation returns a character string and prints the result.

The documentation of this function `String.concat` can be found in the online reference manual by following the links "The standard library" then "Module String: string operations".

To exit the interactive toplevel, the user must type the directive "`#quit ;;`".

Part I

Language Core

The first part of this book is a complete introduction to the core of the Objective Caml language, in particular the expression evaluation mechanism, static typing and the data memory model.

An expression is the description of a computation. Evaluation of an expression returns a value at the end of the computation. The execution of an Objective Caml program corresponds to the computation of an expression. Functions, program execution control structures, even conditions or loops, are themselves also expressions.

Static typing guarantees that the computation of an expression cannot cause a run-time type error. In fact, application of a function to some arguments (or actual parameters) isn't accepted unless they all have types compatible with the formal parameters indicated in the definition of the function. Furthermore, the Objective Caml language has type infererence: the compiler automatically determines the most general type of an expression.

Finally a minimal knowledge of the representation of data is indispensable to the programmer in order to master the effects of physical modifications to the data.

Outline

Chapter 2 contains a complete presentation of the purely functional part of the language and the constraints due to static typing. The notion of expression evaluation is illustrated there at length. The following control structures are detailed: conditional, function application and pattern matching. The differences between the type and the domain of a function are discussed in order to introduce the exception mechanism. This feature of the language goes beyond the functional context and allows management of computational breakdowns.

Chapter 3 exhibits the imperative style. The constructions there are closer to classic languages. Associative control structures such as sequence and iteration are presented there, as well as mutable data structures. The interaction between physical modifications and sharing of data is then detailed. Type inference is described there in the context of these new constructions.

Chapter 4 compares the two preceding styles and especially presents different mixed styles. This mixture supports in particular the construction of lazy data structures, including mutable ones.

Chapter 5 demonstrates the use of the `Graphics` library included in the language distribution. The basic notions of graphics programming are exhibited there and immediately put into practice. There's even something about GUI construction thanks to the minimal event control provided by this library.

These first four chapters are illustrated by a complete example, the implementation of a calculator, which evolves from chapter to chapter.

Chapter 6 presents three complete applications: a little database, a mini-BASIC interpreter and the game Minesweeper. The first two examples are constructed mainly in a functional style, while the third is done in an imperative style.

The rudiments of syntax

Before beginning we indicate the first elements of the syntax of the language. A program is a sequence of phrases in the language. A phrase is a complete, directly executable syntactic element (an expression, a declaration). A phrase is terminated with a double semi-colon (;;). There are three different types of declarations which are each marked with a different keyword:

value declaration	:	**let**
exception declaration	:	**exception**
type declaration	:	**type**

All the examples given in this part are to be input into the interactive toplevel of the language.

Here's a first (little) Objective Caml program, to be entered into the toplevel, whose prompt is the pound character (#), in which a function *fact* computing the factorial of a natural number, and its application to a natural number 8, are defined.

```
# let rec fact n = if n < 2 then 1 else n * fact(n-1) ;;
val fact : int -> int = <fun>
# fact 8 ;;
- : int = 40320
```

This program consists of two *phrases*. The first is the declaration of a function value and the second is an expression. One sees that the toplevel prints out three pieces of information which are: the name being declared, or a dash (-) in the case of an expression; the inferred type; and the return value. In the case of a function value, the system prints <fun>.

The following example demonstrates the manipulation of functions as values in the language. There we first of all define the function *succ* which calculates the successor of an integer, then the function *compose* which composes two functions. The latter will be applied to *fact* and *succ*.

```
# let succ x = x+1 ;;
val succ : int -> int = <fun>
# let compose f g x = f(g x)  ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# compose fact succ 8 ;;
- : int = 362880
```

This last call carries out the computation *fact*(*succ* 8) and returns the expected result. Let us note that the functions *fact* and *succ* are passed as parameters to compose in the same way as the natural number 8.

2

Functional programming

The first functional language, Lisp, appeared at the end of the 1950's. That is, at the same time as Fortran, the first representative of the imperative languages. These two languages still exist, although both have evolved greatly. They are used widely for numerical programming (in the case of Fortran) and symbolic applications in the case of Lisp. Interest in functional programming arises from the great ease of writing programs and specifying the values which they manipulate. A program is a function applied to its arguments. It computes a result which is returned (when the computation terminates) as the output of the program. In this way it becomes easy to combine programs: the output of one program becomes an input argument to another, in the sense of function composition.

Functional programming is based on a simple computation model with three constructions: variables, function definitions, and applications of a function to an argument. This model is called the λ-calculus and it was introduced by Alonzo Church in 1932, thus before the first computer. It was created to offer a general theoretical model of the notion of *computability*. In the λ-calculus, all functions are values which can be manipulated. They can be used as arguments to other functions, or returned as the result of a call to another function. The theory of λ-calculus asserts that everything which is computable (i.e., programmable) can be written in this formalism. Its syntax is too limited to make its use as a programming language practical, so primitive values (such as integers or character strings), operations on these primitive values, control structures, and declarations which allow the naming of values or functions and, in particular, recursive functions, have all been added to the λ-calculus to make it more palatable.

There are several classifications of functional languages. For our part, we will distinguish them according to two characteristics which seem to us most salient:

- Without side effects (pure) or with side effects (impure): a pure functional language is a language in which there is no change of state. There everything is simply a computation and the way it is carried out is unimportant. Impure func-

tional languages, such as Lisp or ML, integrate imperative traits such as change of state. They permit the writing of algorithms in a style closer to languages like Fortran, where the order of evaluation of expressions is significant.

- Dynamically typed or statically typed: typing permits verification of whether an argument passed to a function is indeed of the type of the function's formal parameter. This verification can be made during program execution. In that case this verification is called *dynamic typing*. If type errors occur the program will halt in a consistent state. This is the case in the language Lisp. This verification can also be done before program execution, that is, at compilation time. This *a priori* verification is called *static typing*. Having been carried out once and for all, it won't slow down program execution. This is the case in the ML language and its dialects such as Objective Caml. Only correctly typed programs, i.e., those accepted by the type verifier, will be able to be compiled and then executed.

Chapter outline

This chapter presents the basic elements of the functional part of the Objective Caml language, namely its syntactic elements, its language of types and its exception mechanism. This will lead us to the development of a first example of a complete program.

The first section describes the core of the language, beginning with primitive values and the functions which manipulate them. We then go on to structured values and to function values. The basic control structures are introduced as well as local and global value declarations. The second section deals with type definitions for the construction of structured values and with pattern matching to access these structures. The third section compares the inferred type of functions and their domain of definition, which leads us to introduce the exception mechanism. The fourth section illustrates all these notions put together, by describing a simple application: a desktop calculator.

Functional core of Objective Caml

Like all functional languages, Objective Caml is an expression oriented language, where programming consists mainly of creating functions and applying them. The result of the evaluation of one of these expressions is a value in the language and the execution of a program is the evaluation of all the expressions which comprise it.

Primitive values, functions, and types

Integers and floating-point numbers, characters, character strings, and booleans are predefined in Objective Caml.

Numbers

There are two kinds of numbers: integers[1] of type ***int*** and floating-point numbers of type ***float***. Objective Caml follows the IEEE 754 standard[2] for representing double-precision floating-point numbers. The operations on integers and floating-point numbers are described in figure 2.1. Let us note that when the result of an integer operation is outside the interval on which values of type ***int*** are defined, this does not produce an error, but the result is an integer within the system's interval of integers. In other words, all integer operations are operations *modulo* the boundaries of the interval.

integer numbers		floating-point numbers	
`+`	addition	`+.`	addition
`-`	subtraction and unary negation	`-.`	subtraction and unary negation
`*`	multiplication	`*.`	multiplication
`/`	integer division	`/.`	division
`mod`	remainder of integer division	`**`	exponentiation

```
# 1 ;;
- : int = 1
# 1 + 2 ;;
- : int = 3
# 9 / 2 ;;
- : int = 4
# 11 mod 3 ;;
- : int = 2
(* limits of the representation  *)
(* of integers                   *)
# 2147483650 ;;
- : int = 2
```

```
# 2.0 ;;
- : float = 2
# 1.1 +. 2.2 ;;
- : float = 3.3
# 9.1 /. 2.2 ;;
- : float = 4.13636363636
# 1. /. 0. ;;
- : float = inf
(* limits of the representation  *)
(* of floating-point numbers     *)
# 222222222222.11111 ;;
- : float = 222222222222
```

Figure 2.1: Operations on numbers.

Differences between integers and floating-point numbers Values having different types such as ***float*** and ***int*** can never be compared directly. But there are functions for conversion (`float_of_int` and `int_of_float`) between one and the other.

```
# 2 = 2.0 ;;
Characters 5-8:
This expression has type float but is here used with type int
# 3.0 = float_of_int 3 ;;
```

1. In the interval $[-2^{30}, 2^{30} - 1]$ on 32-bit machines and in the interval $[-2^{62}, 2^{62} - 1]$ on 64-bit machines

2. The floating point number $m \times 10^n$ is represented with a 53-bit mantissa m and an exponent n in the interval $[-1022, 1023]$.

```
- : bool = true
```

In the same way, operations on floating-point numbers are distinct from those on integers.

```
# 3 + 2 ;;
- : int = 5
# 3.0 +. 2.0 ;;
- : float = 5
# 3.0 + 2.0 ;;
Characters 0-3:
This expression has type float but is here used with type int
# sin 3.14159 ;;
- : float = 2.65358979335e-06
```

An ill-defined computation, such as a division by zero, will raise an exception (see page 54) which interrupts the computation. Floating-point numbers have a representation for infinite values (printed as `Inf`) and ill-defined computations (printed as `NaN`[3]). The main functions on floating-point numbers are described in figure 2.2.

functions on floats		trigonometric functions	
`ceil`		`cos`	cosine
`floor`		`sin`	sine
`sqrt`	square root	`tan`	tangent
`exp`	exponential	`acos`	arccosine
`log`	natural log	`asin`	arcsine
`log10`	log base 10	`atan`	arctangent

```
# ceil 3.4 ;;
- : float = 4
# floor 3.4 ;;
- : float = 3
# ceil (-.3.4) ;;
- : float = -3
# floor (-.3.4) ;;
- : float = -4
```

```
# sin 1.57078 ;;
- : float = 0.999999999867
# sin (asin 0.707) ;;
- : float = 0.707
# acos 0.0 ;;
- : float = 1.57079632679
# asin 3.14 ;;
- : float = nan
```

Figure 2.2: Functions on floats.

3. Not a Number

Characters and Strings

Characters, type *char*, correspond to integers between 0 and 255 inclusive, following the ASCII encoding for the first 128. The functions `char_of_int` and `int_of_char` support conversion between integers and characters. Character strings, type *string*, are sequences of characters of definite length (less than $2^{24} - 6$). The concatenation operator is `^` . The functions `int_of_string`, `string_of_int`, `string_of_float` and `float_of_string` carry out the various conversions between numbers and character strings.

```
# 'B' ;;
- : char = 'B'
# int_of_char 'B' ;;
- : int = 66
# "is a string" ;;
- : string = "is a string"
# (string_of_int 1987) ^ " is the year Caml was created" ;;
- : string = "1987 is the year Caml was created"
```

Even if a string contains the characters of a number, it won't be possible to use it in operations on numbers without carrying out an explicit conversion.

```
# "1999" + 1 ;;
Characters 1-7:
This expression has type string but is here used with type int
# (int_of_string "1999") + 1 ;;
- : int = 2000
```

Numerous functions on character strings are gathered in the `String` module (see page 217).

Booleans

Booleans, of type *bool*, belong to a set consisting of two values: `true` and `false`. The primitive operators are described in figure 2.3. For historical reasons, the "and" and "or" operators each have two forms.

`not`	negation	`&`	synonym for `&&`
`&&`	sequential and	`or`	synonym for `||`
`||`	sequential or		

Figure 2.3: Operators on booleans.

```
# true ;;
- : bool = true
# not true ;;
- : bool = false
```

```
# true && false ;;
- : bool = false
```

The operators `&&` and `||`, or their synonyms, evaluate their left argument and then, depending on its value, evaluate their right argument. They can be rewritten in the form of conditional constructs (see page 18).

`=`	structural equality	`<`	less than
`==`	physical equality	`>`	greater than
`<>`	negation of `=`	`<=`	less than or equal to
`!=`	negation of `==`	`>=`	greater than or equal to

Figure 2.4: Equality and comparison operators.

The equality and comparison operators are described in figure 2.4. They are polymorphic, i.e., they can be used to compare two integers as well as two character strings. The only constraint is that their two operands must be of the same type (see page 28).

```
# 1<=118 && (1=2 || not(1=2)) ;;
- : bool = true
# 1.0 <= 118.0 && (1.0 = 2.0 || not (1.0 = 2.0)) ;;
- : bool = true
# "one" < "two" ;;
- : bool = true
# 0 < '0' ;;
Characters 4-7:
This expression has type char but is here used with type int
```

Structural equality tests the equality of two values by traversing their structure, whereas physical equality tests whether the two values occupy the same region in memory. These two equality operators return the same result for simple values: booleans, characters, integers and constant constructors (page 45).

Warning Floating-point numbers and character strings are considered structured values.

Unit

The *unit* type describes a set which possesses only a single element, denoted: `()`.

```
# () ;;
- : unit = ()
```

This value will often be used in imperative programs (see chapter 3, page 67) for functions which carry out side effects. Functions whose result is the value `()` simulate the notion of procedure, which doesn't exist in Objective Caml, just as the type `void` does in the C language.

Cartesian product, tuple

Values of possibly different types can be gathered in pairs or more generally in tuples. The values making up a tuple are separated by commas. The type constructor `*` indicates a tuple. The type *`int * string`* is the type of pairs whose first element is an integer (of type *`int`*) and whose second is a character string (of type *`string`*).

```
# ( 12 , "October" ) ;;
- : int * string = 12, "October"
```

When there is no ambiguity, it can be written more simply:

```
# 12 , "October" ;;
- : int * string = 12, "October"
```

The functions `fst` and `snd` allow access to the first and second elements of a pair.

```
# fst ( 12 , "October" ) ;;
- : int = 12
# snd ( 12 , "October" ) ;;
- : string = "October"
```

These two functions accept pairs whose components are of any type whatsoever. They are polymorphic, in the same way as equality.

```
# fst;;
- : 'a * 'b -> 'a = <fun>
# fst ( "October", 12 ) ;;
- : string = "October"
```

The type *`int * char * string`* is that of triplets whose first element is of type *`int`*, whose second is of type *`char`*, and whose third is of type *`string`*. Its values are written

```
# ( 65 , 'B' , "ascii" ) ;;
- : int * char * string = 65, 'B', "ascii"
```

Warning The functions `fst` and `snd` applied to a tuple, other than a pair, result in a type error.

```
# snd ( 65 , 'B' , "ascii" ) ;;
Characters 7-25:
This expression has type int * char * string but is here used with type
  'a * 'b
```

There is indeed a difference between the type of a pair and that of a triplet. The type *`int * int * int`* is different from the types *`(int * int) * int`* and *`int * (int * int)`*. Functions to access a triplet (and other tuples) are not defined by the core library. One can use pattern matching to define them if need be (see page 34).

Lists

Values of the same type can be gathered into a list. A list can either be empty or consist of elements of the same type.

```
# [] ;;
- : 'a list = []
```

```
# [ 1 ; 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; "two" ; 3 ] ;;
Characters 14-17:
This expression has type int list but is here used with type string list
```

The function which adds an element at the head of a list is the infix operator :: . It is the analogue of Lisp's *cons*.

```
# 1 :: 2 :: 3 :: [] ;;
- : int list = [1; 2; 3]
```

Concatenation of two lists is also an infix operator `@`.

```
# [ 1 ]  @  [ 2 ; 3 ] ;;
- : int list = [1; 2; 3]
# [ 1 ; 2 ]  @  [ 3 ] ;;
- : int list = [1; 2; 3]
```

The other list manipulation functions are defined in the `List` library. The functions `hd` and `tl` from this library give respectively the head and the tail of a list when these values exist. These functions are denoted by `List.hd` and `List.tl` to indicate to the system that they can be found in the module `List`[4].

```
# List.hd [ 1 ; 2 ; 3 ] ;;
- : int = 1
# List.hd [] ;;
Uncaught exception: Failure("hd")
```

In this last example, it is indeed problematic to request retrieval of the first element of an empty list. It is for this reason that the system raises an *exception* (see page 54).

Conditional control structure

One of the indispensable control structures in any programming language is the structure called *conditional* (or branch) which guides the computation as a function of a condition.

Syntax : **if** $expr_1$ **then** $expr_2$ **else** $expr_3$

The expression $expr_1$ is of type `bool`. The expressions $expr_2$ and $expr_3$ must be of the same type, whatever it may be.

```
# if 3=4 then 0 else 4 ;;
- : int = 4
# if 3=4 then "0" else "4" ;;
```

4. The `List` module is presented on page 217.

```
- : string = "4"
# if 3=4 then 0 else  "4";;
Characters 20-23:
This expression has type string but is here used with type int
```

A conditional construct is itself an expression and its evaluation returns a value.

```
# (if 3=5 then 8 else 10) + 5 ;;
- : int = 15
```

Note

The **else** branch can be omitted, but in this case it is implicitly replaced by **else** (). Consequently, the type of the expression $expr_2$ must be ***unit*** (see page 79).

Value declarations

A declaration binds a name to a value. There are two types: *global declarations* and *local declarations*. In the first case, the declared names are known to all the expressions following the declaration; in the second, the declared names are only known to one expression. It is equally possible to *simultaneously* declare several name-value bindings.

Global declarations

Syntax : **let** *name* **=** *expr* **;;**

A global declaration defines the binding between the name *name* and the value of the expression *expr* which will be known to all subsequent expressions.

```
# let yr = "1999" ;;
val yr : string = "1999"
# let x = int_of_string(yr) ;;
val x : int = 1999
# x ;;
- : int = 1999
# x + 1 ;;
- : int = 2000
# let new_yr = string_of_int (x + 1) ;;
val new_yr : string = "2000"
```

Simultaneous global declarations

Syntax :

let $name_1$ **=** $expr_1$
and $name_2$ **=** $expr_2$
⋮
and $name_n$ **=** $expr_n$ **;;**

A simultaneous declaration declares different symbols at the same level. They won't be known until the end of all the declarations.

```
# let x = 1 and y = 2 ;;
val x : int = 1
val y : int = 2
# x + y ;;
- : int = 3
# let z = 3 and t = z + 2 ;;
Characters 18-19:
Unbound value z
```

It is possible to gather several global declarations in the same phrase; then printing of their types and their values does not take place until the end of the phrase, marked by double "**;;**". These declarations are evaluated sequentially, in contrast with a simultaneous declaration.

```
# let x = 2
  let y = x + 3  ;;
val x : int = 2
val y : int = 5
```

A global declaration can be masked by a new declaration of the same name (see page 26).

Local declarations

Syntax : **let** *name* **=** $expr_1$ **in** $expr_2$**;;**

The name *name* is only known during the evaluation of $expr_2$. The local declaration binds it to the value of $expr_1$.

```
# let xl = 3 in xl * xl ;;
- : int = 9
```

The local declaration binding `xl` to the value `3` is only in effect during the evaluation of `xl * xl`.

```
# xl ;;
Characters 1-3:
Unbound value xl
```

A local declaration masks all previous declarations of the same name, but the previous value is reinstated upon leaving the scope of the local declaration:

```
# let x = 2 ;;
```

```
val x : int = 2
# let x = 3 in x * x ;;
- : int = 9
# x * x ;;
- : int = 4
```

A local declaration is an expression and can thus be used to construct other expressions:

```
# (let x = 3 in x * x) + 1 ;;
- : int = 10
```

Local declarations can also be simultaneous.

Syntax :

```
let   name_1 = expr_1
and   name_2 = expr_2
 ⋮
and   name_n = expr_n
in    expr ;;
```

```
# let a = 3.0 and b = 4.0 in sqrt (a*.a +. b*.b) ;;
- : float = 5
# b ;;
Characters 0-1:
Unbound value b
```

Function expressions, functions

A function expression consists of a *parameter* and a *body*. The formal parameter is a variable name and the body an expression. The parameter is said to be *abstract*. For this reason, a function expression is also called an *abstraction*.

Syntax : **function** *p* -> *expr*

Thus the function which squares its argument is written:

```
# function x → x*x ;;
- : int -> int = <fun>
```

The Objective Caml system deduces its type. The *function type* ***int -> int*** indicates a function expecting a parameter of type ***int*** and returning a value of type ***int***.

Application of a function to an argument is written as the function followed by the argument.

```
# (function x → x * x) 5 ;;
- : int = 25
```

The evaluation of an application amounts to evaluating the body of the function, here

`x * x`, where the formal parameter, `x`, is replaced by the value of the argument (or the *actual parameter*), here `5`.

In the construction of a function expression, ***expr*** is any expression whatsoever. In particular, ***expr*** may itself be a function expression.

```
# function x → (function y → 3*x + y) ;;
- : int -> int -> int = <fun>
```

The parentheses are not required. One can write more simply:

```
# function x → function y → 3*x + y ;;
- : int -> int -> int = <fun>
```

The type of this expression can be read in the usual way as the type of a function which expects two integers and returns an integer value. But in the context of a functional language such as Objective Caml we are dealing more precisely with the type of a function which expects an integer and returns a *function value* of type ***int -> int***:

```
# (function x → function y → 3*x + y) 5 ;;
- : int -> int = <fun>
```

One can, of course, use the function expression in the usual way by applying it to two arguments. One writes:

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

When one writes ***f a b***, there is an implicit parenthesization on the left which makes this expression equivalent to: ***(f a) b***.

Let's examine the application

```
(function x → function y → 3*x + y) 4 5
```

in detail. To compute the value of this expression, it is necessary to compute the value of

```
(function x → function y → 3*x + y) 4
```

which is a *function expression* equivalent to

```
function y → 3*4 + y
```

obtained by replacing ***x*** by `4` in ***3*x + y***. Applying this value (which is a function) to `5` we get the final value `3*4+5 = 17`:

```
# (function x → function y → 3*x + y) 4 5 ;;
- : int = 17
```

Arity of a function

The number of arguments of a function is called its *arity*. Usage inherited from mathematics demands that the arguments of a function be given in parentheses after the name of the function. One writes: $f(4,5)$. We've just seen that in Objective Caml, one more usually writes: f 4 5. One can, of course, write a function expression in Objective Caml which can be applied to $(4,5)$:

```
# function (x,y) → 3*x + y ;;
- : int * int -> int = <fun>
```

But, as its type indicates, this last expression expects not two, but only one argument: a pair of integers. Trying to pass two arguments to a function which expects a pair or trying to pass a pair to a function which expects two arguments results in a type error:

```
# (function (x,y) → 3*x + y) 4 5 ;;
Characters 29-30:
This expression has type int but is here used with type int * int
# (function x → function y → 3*x + y) (4, 5) ;;
Characters 39-43:
This expression has type int * int but is here used with type int
```

Alternative syntax

There is a more compact way of writing function expressions with several parameters. It is a legacy of former versions of the Caml language. Its form is as follows:

Syntax : **fun** $p_1 \ldots p_n$ -> *expr*

It allows one to omit repetitions of the **function** keyword and the arrows. It is equivalent to the following translation:

function p_1 -> ...-> **function** p_n -> *expr*

```
# fun x y → 3*x + y ;;
- : int -> int -> int = <fun>
# (fun x y → 3*x + y) 4 5 ;;
- : int = 17
```

This form is still encountered often, in particular in the libraries provided with the Objective Caml distribution.

Closure

Objective Caml treats a function expression like any other expression and is able to compute its value. The value returned by the computation is a function expression and is called a *closure*. Every Objective Caml expression is evaluated in an *environment*

consisting of name-value bindings coming from the declarations preceding the expression being computed. A closure can be described as a triplet consisting of the name of the formal parameter, the body of the function, and the environment of the expression. This environment needs to be preserved because the body of a function expression may use, in addition to the formal parameters, every other variable declared previously. These variables are said to be "free" in the function expression. Their values will be needed when the function expression is applied.

```
# let m = 3 ;;
val m : int = 3
# function x → x + m ;;
- : int -> int = <fun>
# (function x → x + m) 5 ;;
- : int = 8
```

When application of a closure to an argument returns a new closure, the latter possesses within its environment all the bindings necessary for a future application. The subsection on the scope of variables (see page 26) details this notion. We will return to the memory representation of a closure in chapter 4 (page 103) as well as chapter 12 (page 332).

The function expressions used until now are *anonymous*. It is rather useful to be able to name them.

Function value declarations

Function values are declared in the same way as other language values, by the **let** construct.

```
# let succ = function x → x + 1 ;;
val succ : int -> int = <fun>
# succ 420 ;;
- : int = 421
# let g = function x → function y → 2*x + 3*y ;;
val g : int -> int -> int = <fun>
# g 1 2;;
- : int = 8
```

To simplify writing, the following notation is allowed:

Syntax : **let** *name* $p_1 \ldots p_n$ **=** *expr*

which is equivalent to the following form:

let *name* **= function** p_1 -> ...-> **function** p_n -> *expr*

The following declarations of `succ` and `g` are equivalent to their previous declaration.

```
# let succ x = x + 1 ;;
```

```
val succ : int -> int = <fun>
# let g x y = 2*x + 3*y ;;
val g : int -> int -> int = <fun>
```

The completely functional character of Objective Caml is brought out by the following example, in which the function `h1` is obtained by the application of `g` to a single integer. In this case one speaks of *partial application*:

```
# let h1 = g 1 ;;
val h1 : int -> int = <fun>
# h1 2 ;;
- : int = 8
```

One can also, starting from *g*, define a function *h2* by fixing the value of the second parameter, *y*, of *g*:

```
# let h2 = function x → g x 2 ;;
val h2 : int -> int = <fun>
# h2 1 ;;
- : int = 8
```

Declaration of infix functions

Certain functions taking two arguments can be applied in infix form. This is the case with addition of integers. One writes `3 + 5` for the application of `+` to `3` and `5`. To use the symbol `+` as a regular function value, this must be syntactically indicated by surrounding the infix symbol with parentheses. The syntax is as follows:

Syntax : `( op )`

The following example defines the function `succ` using `( + )`.

```
# ( + ) ;;
- : int -> int -> int = <fun>
# let succ = ( + ) 1 ;;
val succ : int -> int = <fun>
# succ 3 ;;
- : int = 4
```

It is also possible to define new operators. We define an operator `++`, addition on pairs of integers

```
# let ( ++ ) c1 c2 = (fst c1)+(fst c2), (snd c1)+(snd c2) ;;
val ++ : int * int -> int * int -> int * int = <fun>
# let c = (2,3) ;;
val c : int * int = 2, 3
# c ++ c ;;
- : int * int = 4, 6
```

There is an important limitation on the possible operators. They must contain only *symbols* (such as `*`, `+`, `@`, etc.) and not letters or digits. Certain functions predefined as infixes are exceptions to the rule. They are listed as follows: `or mod land lor lxor lsl lsr asr`.

Higher order functions

A function value (a closure) can be returned as a result. It can equally well be passed as an argument to a function. Functions taking function values as arguments or returning them as results are called *higher order*.

```
# let h = function f → function y → (f y) + y ;;
val h : (int -> int) -> int -> int = <fun>
```

Note

Application is implicitly parenthesized to the left, but function types are implicitly parenthesized to the right. Thus the type of the function `h` can be written

(int -> int) -> int -> int or *(int -> int) -> (int -> int)*

Higher order functions offer elegant possibilities for dealing with lists. For example the function `List.map` can apply a function to all the elements of a list and return the results in a list.

```
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# let square x = string_of_int (x*x) ;;
val square : int -> string = <fun>
# List.map square [1; 2; 3; 4] ;;
- : string list = ["1"; "4"; "9"; "16"]
```

As another example, the function `List.for_all` can find out whether all the elements of a list satisfy a given criterion.

```
# List.for_all ;;
- : ('a -> bool) -> 'a list -> bool = <fun>
# List.for_all (function n → n<>0) [-3; -2; -1; 1; 2; 3] ;;
- : bool = true
# List.for_all (function n → n<>0) [-3; -2; 0; 1; 2; 3] ;;
- : bool = false
```

Scope of variables

In order for it to be possible to evaluate an expression, all the variables appearing therein must be defined. This is the case in particular for the expression `e` in the dec-

laration **let** *p* = *e*. But since p is not yet known within this expression, this variable can only be present if it refers to another value issued by a previous declaration.

```
# let p = p ^ "-suffix" ;;
Characters 9-10:
Unbound value p
# let p = "prefix" ;;
val p : string = "prefix"
# let p = p ^ "-suffix" ;;
val p : string = "prefix-suffix"
```

In Objective Caml, variables are statically bound. The environment used to execute the application of a closure is the one in effect at the moment of its declaration (static scope) and not the one in effect at the moment of application (dynamic scope).

```
# let p = 10 ;;
val p : int = 10
# let k x = (x, p, x+p) ;;
val k : int -> int * int * int = <fun>
# k p ;;
- : int * int * int = 10, 10, 20
# let p = 1000 ;;
val p : int = 1000
# k p ;;
- : int * int * int = 1000, 10, 1010
```

The function k contains a free variable: p. Since the latter is defined in the global environment, the definition of k is legal. The binding between the name p and the value 10 in the environment of the closure k is static, i.e., does not depend on the most recent definition of p.

Recursive declarations

A variable declaration is called *recursive* if it uses its own identifier in its definition. This facility is used mainly for functions, notably to simulate a definition by recurrence. We have just seen that the **let** declaration does not support this. To declare a recursive function we will use a dedicated syntactic construct.

Syntax : **let rec** *name* **=** *expr* **;;**

We can equally well use the syntactic facility for defining function values while indicating the function parameters:

Syntax : **let rec** *name* $p_1 \ldots p_n$ **=** *expr* **;;**

By way of example, here is the function sigma which computes the sum of the (non-negative) integers between 0 and the value of its argument, inclusive.

```
# let rec sigma x = if x = 0 then 0 else x + sigma (x-1) ;;
val sigma : int -> int = <fun>
```

```
# sigma 10 ;;
- : int = 55
```
It may be noted that this function does not terminate if its argument is strictly negative.

A recursive value is in general a function. The compiler rejects some recursive declarations whose values are not functions:
```
# let rec x = x + 1 ;;
Characters 13-18:
This kind of expression is not allowed as right-hand side of 'let rec'
```
We will see however that in certain cases such declarations are allowed (see page 52).

The **let rec** declaration may be combined with the **and** construction for simultaneous declarations. In this case, all the functions defined at the same level are known within the bodies of each of the others. This permits, among other things, the declaration of *mutually recursive* functions.
```
# let rec even n = (n<>1) && ((n=0) or (odd (n-1)))
  and     odd  n = (n<>0) && ((n=1) or (even (n-1)))    ;;
val even : int -> bool = <fun>
val odd : int -> bool = <fun>
# even 4 ;;
- : bool = true
# odd 5 ;;
- : bool = true
```

In the same way, local declarations can be recursive. This new definition of `sigma` tests the validity of its argument before carrying out the computation of the sum defined by a local function `sigma_rec`.
```
# let sigma x =
    let rec sigma_rec x =  if x = 0 then 0 else x + sigma_rec (x-1) in
     if (x<0) then "error: negative argument"
     else "sigma = " ^ (string_of_int (sigma_rec x)) ;;
val sigma : int -> string = <fun>
```

Note

The need to give a return value of the same type, whether the argument is negative or not, has forced us to give the result in the form of a character string. Indeed, what value should be returned by `sigma` when its argument is negative? We will see the proper way to manage this problem, using exceptions (see page 54).

Polymorphism and type constraints

Some functions execute the same code for arguments having different types. For example, creation of a pair from two values doesn't require different functions for each type

known to the system[5]. In the same way, the function to access the first field of a pair doesn't have to be differentiated according to the type of the value of this first field.

```
# let make_pair a b = (a,b) ;;
val make_pair : 'a -> 'b -> 'a * 'b = <fun>
# let p = make_pair "paper" 451 ;;
val p : string * int = "paper", 451
# let a = make_pair 'B' 65 ;;
val a : char * int = 'B', 65
# fst p ;;
- : string = "paper"
# fst a ;;
- : char = 'B'
```

Functions are called polymorphic if their return value or one of their parameters is of a type which need not be specified. The type synthesizer contained in the Objective Caml compiler finds the most general type for each expression. In this case, Objective Caml uses variables, here `'a` and `'b`, to designate these general types. These variables are instantiated to the type of the argument during application of the function.

With Objective Caml's polymorphic functions, we get the advantages of being able to write generic code usable for values of every type, while still preserving the execution safety of static typing. Indeed, although `make_pair` is polymorphic, the value created by `(make_pair 'B' 65)` has a well-specified type which is different from that of `(make_pair "paper" 451)`. Moreover, type verification is carried out on compilation, so the generality of the code does not hamper the efficiency of the program.

Examples of polymorphic functions and values

The following examples of polymorphic functions have functional parameters whose type is parameterized.

The `app` function applies a function to an argument.

```
# let app = function f → function x → f x ;;
val app : ('a -> 'b) -> 'a -> 'b = <fun>
```

So it can be applied to the function `odd` defined previously:

```
# app odd 2;;
- : bool = false
```

The identity function (`id`) takes a parameter and returns it as is.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# app id  1 ;;
- : int = 1
```

5. Fortunately since the number of types is only limited by machine memory

The `compose` function takes two functions and another value and composes the application of these two functions to this value.

```
# let compose f g  x = f (g x) ;;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
# let add1 x = x+1 and mul5 x = x*5 in compose  mul5 add1  9 ;;
- : int = 50
```

It can be seen that the result of `g` must be of the same type as the argument of `f`.

Values other than functions can be polymorphic as well. For example, this is the case for the empty list:

```
# let l = [] ;;
val l : 'a list = []
```

The following example demonstrates that type synthesis indeed arises from resolution of the constraints coming from function application and not from the value obtained upon execution.

```
# let t = List.tl [2] ;;
val t : int list = []
```

The type of `List.tl` is *`'a list -> 'a list`*, so this function applied to a list of integers returns a list of integers. The fact that upon execution it is the empty list which is obtained doesn't change its type at all.

Objective Caml generates parameterized types for every function which doesn't use the form of its arguments. This polymorphism is called *parametric polymorphism*[6].

Type constraint

As the Caml type synthesizer generates the most general type, it may be useful or necessary to specify the type of an expression.

The syntactic form of a type constraint is as follows:

Syntax : **`(`** *expr* **`:`** *`t`* **`)`**

When it runs into such a constraint, the type synthesizer will take it into account while constructing the type of the expression. Using type constraints lets one:

- make the type of the parameters of a function visible;
- forbid the use of a function outside its intended context;
- specify the type of an expression, which will be particularly useful for mutable values (see page 68).

The following examples demonstrate the use of such type constraints

```
# let add (x:int) (y:int) = x + y ;;
```

6. Some predefined functions do not obey this rule, in particular the structural equality function (`=`) which is polymorphic (its type is *`'a -> 'a -> bool`*) but which explores the structure of its arguments to test their equality.

```
val add : int -> int -> int = <fun>
# let make_pair_int (x:int) (y:int) = x,y;;
val make_pair_int : int -> int -> int * int = <fun>
# let compose_fn_int (f : int → int) (g : int → int) (x:int) =
    compose f g  x;;
val compose_fn_int : (int -> int) -> (int -> int) -> int -> int = <fun>
# let nil = ( [] : string list);;
val nil : string list = []
# 'H' :: nil;;
Characters 5-8:
This expression has type string list but is here used with type char list
```

Restricting polymorphism this way lets us control the type of an expression better by constraining the polymorphism of the type deduced by the system. Any defined type whatsoever may be used, including ones containing type variables, as the following example shows:

```
# let llnil = ( [] : 'a list list) ;;
val llnil : 'a list list = []
# [1;2;3]:: llnil ;;
- : int list list = [[1; 2; 3]]
```

The symbol `llnil` is a list of lists of any type whatsoever.

Here we are dealing with constraints, and not replacing Objective Caml's type synthesis with explicit typing. In particular, one cannot generalize types beyond what inference permits.

```
# let add_general (x:'a) (y:'b) = add x y ;;
val add_general : int -> int -> int = <fun>
```

Type constraints will be used in module interfaces (see chapter 14) as well as in class declarations (see chapter 15).

Examples

In this section we will give several somewhat elaborate examples of functions. Most of these functions are predefined in Objective Caml. We will redefine them for the sake of "pedagogy".

Here, the test for the terminal case of recursive functions is implemented by a conditional. Hence a programming style closer to Lisp. We will see how to give a more ML character to these definitions when we present another way of defining functions by case (see page 34).

Length of a list

Let's start with the function `null` which tests whether a list is empty.

```
# let null l =  (l = [] ) ;;
```

```
val null : 'a list -> bool = <fun>
```
Next, we define the function `size` to compute the length of a list (*i.e.*, the number of its elements).
```
# let rec size l =
    if null l then 0
    else 1 + (size (List.tl l)) ;;
val size : 'a list -> int = <fun>
# size [] ;;
- : int = 0
# size [1;2;18;22] ;;
- : int = 4
```
The function `size` tests whether the list argument is empty. If so it returns `0`, if not it returns `1` plus the value resulting from computing the length of the tail of the list.

Iteration of composition

The expression *iterate n f* computes the value `f` iterated n times.
```
# let rec iterate n f =
    if n = 0 then (function x → x)
    else compose f (iterate (n-1) f) ;;
val iterate : int -> ('a -> 'a) -> 'a -> 'a = <fun>
```
The `iterate` function tests whether *n* is 0, if yes it returns the identity function, if not it composes `f` with the iteration of `f n-1` times.

Using `iterate`, one can define exponentiation as iteration of multiplication.
```
# let rec power i n =
   let i_times = ( * ) i in
    iterate n i_times 1 ;;
val power : int -> int -> int = <fun>
# power 2 8 ;;
- : int = 256
```
The `power` function iterates n times the function expression `i_times`, then applies this result to 1, which does indeed compute the nth power of an integer.

Multiplication table

We want to write a function *multab* which computes the multiplication table of an integer passed as an argument.

First we define the function `apply_fun_list` such that, if *f_list* is a list of functions, *apply_fun_list x f_list* returns the list of results of applying each element of *f_list* to *x*.
```
# let rec apply_fun_list x f_list =
    if null f_list then []
    else ((List.hd f_list) x) :: (apply_fun_list x (List.tl f_list)) ;;
val apply_fun_list : 'a -> ('a -> 'b) list -> 'b list = <fun>
# apply_fun_list 1 [( + ) 1;( + ) 2;( + ) 3] ;;
```

```
- : int list = [2; 3; 4]
```

The function *mk_mult_fun_list* returns the list of functions multiplying their argument by *i*, for *i* varying from 0 to *n*.

```
# let mk_mult_fun_list n =
    let rec mmfl_aux p =
      if p = n then [ ( * ) n ]
      else (( * ) p) :: (mmfl_aux (p+1))
    in  (mmfl_aux 1) ;;
val mk_mult_fun_list : int -> (int -> int) list = <fun>
```

We obtain the multiplication table of 7 by:

```
# let multab n = apply_fun_list n (mk_mult_fun_list 10) ;;
val multab : int -> int list = <fun>
# multab 7 ;;
- : int list = [7; 14; 21; 28; 35; 42; 49; 56; 63; 70]
```

Iteration over lists

The function call *fold_left f a* [*e1*; *e2*; ... ; *en*] returns *f* ... (*f* (*f a e1*) *e2*) ... *en*. So there are n applications.

```
# let rec fold_left f a l =
    if null l then a
    else fold_left f ( f a (List.hd l)) (List.tl l) ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
```

The function `fold_left` permits the compact definition of a function to compute the sum of the elements of a list of integers:

```
# let sum_list = fold_left (+) 0 ;;
val sum_list : int list -> int = <fun>
# sum_list [2;4;7] ;;
- : int = 13
```

Or else, the concatenation of the elements of a list of strings:

```
# let concat_list = fold_left (^) "" ;;
val concat_list : string list -> string = <fun>
# concat_list ["Hello "; "world" ; "!"] ;;
- : string = "Hello world!"
```

Type declarations and pattern matching

Although Objective Caml's predefined types permit the construction of data structures from tuples and lists, one needs to be able to define new types to describe certain data structures. In Objective Caml, type declarations are recursive and may be parameterized by type variables, in the same vein as the type `'a list` already encountered. Type inference takes these new declarations into account to produce the type of an expression. The construction of values of these new types uses the *constructors* described in their definition. A special feature of languages in the ML family is *pattern matching*. It allows simple access to the components of complex data structures. A function definition most often corresponds to pattern matching over one of its parameters, allowing the function to be defined by cases.

First of all we present pattern matching over the predefined types, and then go on to describe the various ways to declare structured types and how to construct values of such types, as well as how to access their components through pattern matching.

Pattern matching

A pattern is not strictly speaking an Objective Caml expression. It's more like a correct (syntactically, and from the point of view of types) arrangement of elements such as constants of the primitive types (`int`, `bool`, `char`, ..), variables, constructors, and the symbol _ called the *wildcard pattern*. Other symbols are used in writing patterns. We will introduce them to the extent needed.

Pattern matching applies to values. It is used to recognize the form of this value and lets the computation be guided accordingly, associating with each pattern an expression to compute.

Syntax :

match *expr* **with**
| p_1 -> $expr_1$
⋮
| p_n -> $expr_n$

The expression *expr* is matched sequentially to the various patterns $p_1, \ldots, p_n$. If one of the patterns (for example p_i) is consistent with the value of *expr* then the corresponding computation branch ($expr_i$) is evaluated. The various patterns p_i are of the same type. The same goes for the various expressions $expr_i$. The vertical bar preceding the first pattern is optional.

Examples

Here are two ways to define by pattern matching a function `imply` of type `(bool * bool) -> bool` implementing logical implication. A pattern which matches pairs has the form `( , )`.

The first version of `imply` enumerates all possible cases, as a truth table would:

```
# let imply v = match v with
      (true,true)   → true
    | (true,false)  → false
    | (false,true)  → true
    | (false,false) → true;;
val imply : bool * bool -> bool = <fun>
```

By using variables which group together several cases, we obtain a more compact definition.

```
# let imply v = match v with
      (true,x)  → x
    | (false,x) → true;;
val imply : bool * bool -> bool = <fun>
```

These two versions of `imply` compute the same function. That is, they return the same values for the same inputs.

Linear pattern

A pattern must necessarily be *linear*, that is, no given variable can occur more than once inside the pattern being matched. Thus, we might have hoped to be able to write:

```
# let equal c = match c with
     (x,x) → true
   | (x,y) → false;;
Characters 35-36:
This variable is bound several times in this matching
```

But this would have required the compiler to know how to carry out equality tests. Yet this immediately raises numerous problems. If we accept physical equality between values, we get a system which is too weak, incapable of recognizing the equality between two occurrences of the list `[1; 2]`, for example. If we decide to use structural equality, we run the risk of having to traverse, ad infinitum, circular structures. Recursive functions, for example, are circular structures, but we can construct recursive, hence circular, values which are not functions as well (see page 52).

Wildcard pattern

The symbol _ matches all possible values. It is called a wildcard pattern. It can be used to match complex types. We use it, for example, to further simplify the definition of the function `imply`:

```
# let imply v = match v with
          (true,false) → false
        |   _          → true;;
val imply : bool * bool -> bool = <fun>
```

A definition by pattern matching must handle the entire set of possible cases of the values being matched. If this is not the case, the compiler prints a warning message:

```
# let is_zero n = match n with  0 → true ;;
Characters 17-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Indeed if the actual parameter is different from `0` the function doesn't know what value to return. So the case analysis can be completed using the wildcard pattern.

```
# let is_zero n = match n with
     0 → true
   | _ → false ;;
val is_zero : int -> bool = <fun>
```

If, at run-time, no pattern is selected, then an exception is raised. Thus, one can write:

```
# let f x = match x with 1 → 3 ;;
Characters 11-30:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
0
val f : int -> int = <fun>
# f 1 ;;
- : int = 3
# f 4 ;;
Uncaught exception: Match_failure("", 11, 30)
```

The `Match_Failure` exception is raised by the call to `f 4`, and if it is not handled induces the computation in progress to halt (see 54)

Combining patterns

Combining several patterns lets us obtain a new pattern which can match a value according to one or another of the original patterns. The syntactic form is as follows:

Syntax : p_1 **|** ... **|** p_n

It constructs a new pattern by combining the patterns p_1, ... and p_n. The only strong constraint is that all naming is forbidden within these patterns. So each one of them must contain only constant values or the wildcard pattern. The following example demonstrates how to verify that a character is a vowel.

```
# let is_a_vowel c = match c with
     'a' | 'e' | 'i' | 'o' | 'u' | 'y' → true
   | _ → false ;;
val is_a_vowel : char -> bool = <fun>
```

```
# is_a_vowel 'i' ;;
- : bool = true
# is_a_vowel 'j' ;;
- : bool = false
```

Pattern matching of a parameter

Pattern matching is used in an essential way for defining functions by cases. To make writing these definitions easier, the syntactic construct **`function`** allows pattern matching of a parameter:

Syntax :

`function`	**`|`**	p_1 `->` $expr_1$
	`|`	p_2 `->` $expr_2$
		$\vdots$
	`|`	p_n `->` $expr_n$

The vertical bar preceding the first pattern is optional here as well. In fact, like Mr. Jourdain, each time we define a function, we use pattern matching[7]. Indeed, the construction `function x -> expression`, is a definition by pattern matching using a single pattern reduced to one variable. One can make use of this detail with simple patterns as in:

```
# let f = function (x,y) → 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

In fact the form

`function` p_1 `->` $expr_1$ **`|`** $\ldots$ **`|`** p_n `->` $expr_n$

is equivalent to

`function` $expr$ `->` **`match`** `expr` **`with`** p_1 `->` $expr_1$ **`|`** $\ldots$ **`|`** p_n `->` $expr_n$

Using the equivalence of the declarations mentioned on page 24, we write:

```
# let f (x,y) = 2*x + 3*y + 4 ;;
val f : int * int -> int = <fun>
```

But this natural way of writing is only possible if the value being matched belongs to

7. Translator's note: In Molière's play *Le Bourgeois Gentilhomme* (*The Bourgeois Gentleman*), the character Mr. Jourdain is amazed to discover that he has been speaking prose all his life. The play can be found at
Link: http://www.site-moliere.com/pieces/bourgeoi.htm
and
Link: http://moliere-in-english.com/bourgeois.html
gives an excerpt from an English translation, including this part.

a type having only a single constructor. If such is not the case, the pattern matching is not exhaustive:

```
# let is_zero 0 = true ;;
Characters 13-21:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
1
val is_zero : int -> bool = <fun>
```

Naming a value being matched

During pattern matching, it is sometimes useful to name part or all of the pattern. The following syntactic form introduces the keyword **as** which binds a name to a pattern.

Syntax : **(** *p* **as** *name* **)**

This is useful when one needs to take apart a value while still maintaining its integrity. In the following example, the function `min_rat` gives the smaller rational of a pair of rationals. The latter are each represented by a numerator and denominator in a pair.

```
# let min_rat pr = match pr with
    ((_,0),p2) →  p2
  | (p1,(_,0)) →  p1
  | (((n1,d1) as r1), ((n2,d2) as r2)) →
          if (n1 *  d2 ) < (n2 * d1) then r1 else r2;;
val min_rat : (int * int) * (int * int) -> int * int = <fun>
```

To compare two rationals, it is necessary to take them apart in order to name their numerators and denominators (`n1`, `n2`, `d1` and `d2`), but the initial pair (`r1` or `r2`) must be returned. The **as** construct allows us to name the parts of a single value in this way. This lets us avoid having to reconstruct the rational returned as the result.

Pattern matching with guards

Pattern matching with guards corresponds to the evaluation of a conditional expression immediately after the pattern is matched. If this expression comes back `true`, then the expression associated with that pattern is evaluated, otherwise pattern matching continues with the following pattern.

Syntax :

match *expr* **with**
⋮
| p_i **when** $cond_i$ -> $expr_i$
⋮

The following example uses two guards to test equality of two rationals.

```
# let eq_rat cr = match cr with
```

```
    ((_,0),(_,0)) →  true
  | ((_,0),_) →  false
  | (_,(_,0)) →  false
  | ((n1,1), (n2,1)) when n1 = n2 → true
  | ((n1,d1), (n2,d2)) when ((n1 * d2) = (n2 * d1)) → true
  | _ → false;;
val eq_rat : (int * int) * (int * int) -> bool = <fun>
```

If the guard fails when the fourth pattern is matched, matching continues with the fifth pattern.

Note

The verification carried out by Objective Caml as to whether the pattern matching is exhaustive assumes that the conditional expression in the guard may be false. Consequently, it does not count this pattern since it is not possible to know, before execution, whether the guard will be satisfied or not.

It won't be possible to detect that the pattern matching in the following example is exhaustive.

```
#  let f = function x when x = x  → true;;
Characters 10-40:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:

-
val f : 'a -> bool = <fun>
```

Pattern matching on character intervals

In the context of pattern matching on characters, it is tedious to construct the combination of all the patterns corresponding to a character interval. Indeed, if one wishes to test a character or even a letter, one would need to write 26 patterns at a minimum and combine them. For characters, Objective Caml permits writing patterns of the form:

Syntax : `'c1' .. 'cn'`

It is equivalent to the combination: `'c1' | 'c2' | ...| 'cn'`.

For example the pattern `'0' .. '9'` corresponds to the pattern `'0' | '1' | '2' | '3' | '4' | '5' | '6' | '7' | '8' | '9'`. The first form is nicer to read and quicker to write.

Warning This feature is among the extensions to the language and may change in future versions.

Using combined patterns and intervals, we define a function categorizing characters according to several criteria.

```
# let char_discriminate c = match c with
```

```
      'a' | 'e' | 'i' | 'o' | 'u' | 'y'
    | 'A' | 'E' | 'I' | 'O' | 'U' | 'Y'  → "Vowel"
    | 'a'..'z' | 'A'..'Z' → "Consonant"
    | '0'..'9' → "Digit"
    |   _ → "Other" ;;
val char_discriminate : char -> string = <fun>
```

It should be noted that the order of the groups of patterns has some significance. Indeed, the second set of patterns includes the first, but it is not examined until after the check on the first.

Pattern matching on lists

As we have seen, a list can be:

- either empty (the list is of the form **`[]`**),
- or composed of a first element (its head) and a sublist (its tail). The list is then of the form `h::t`.

These two possible ways of writing a list can be used as patterns and allow pattern matching on a list.

```
# let rec size x = match x with
      [] → 0
    | _::tail_x → 1 + (size tail_x) ;;
val size : 'a list -> int = <fun>
# size [];;
- : int = 0
# size [7;9;2;6];;
- : int = 4
```

So we can redo the examples described previously (see page 31) using pattern matching, such as iteration over lists for example.

```
# let rec fold_left f a = function
     [] → a
   | head::tail → fold_left f (f a head) tail ;;
val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
# fold_left (+) 0 [8;4;10];;
- : int = 22
```

Value declaration through pattern matching

Value declaration in fact uses pattern matching. The declaration **`let`** *x* `=` `18` matches the value `18` with the pattern *x*. Any pattern is allowed as the left-hand side of a declaration; the variables in the pattern are bound to the values which they match.

```
# let (a,b,c) = (1, true, 'A');;
val a : int = 1
```

```
val b : bool = true
val c : char = 'A'
# let (d,c) = 8, 3 in d + c;;
- : int = 11
```

The scope of pattern variables is the usual static scope for local declarations. Here, `c` remains bound to the value `'A'`.

```
# a + (int_of_char c);;
- : int = 66
```

As with any kind of pattern matching, value declaration may not be exhaustive.

```
# let [x;y;z] = [1;2;3];;
Characters 5-12:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val x : int = 1
val y : int = 2
val z : int = 3
# let [x;y;z] = [1;2;3;4];;
Characters 4-11:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
Uncaught exception: Match_failure("", 4, 11)
```

Any pattern is allowed, including constructors, wildcards and combined patterns.

```
# let head :: 2 :: _ =  [1; 2; 3] ;;
Characters 5-19:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : int = 1
# let _ = 3. +. 0.14 in "PI" ;;
- : string = "PI"
```

This last example is of little use in the functional world insofar as the computed value `3.14` is not named and so is lost.

Type declaration

Type declarations are another possible ingredient in an Objective Caml phrase. They support the definition of new types corresponding to the original data structures used in a program. There are two major families of types: *product* types for tuples or records; and *sum* types for unions.

Type declarations use the keyword **type**.

Syntax : **type** *name* **=** *typedef* **;;**

In contrast with variable declarations, type declarations are recursive by default. That is, type declarations, when combined, support the declaration of mutually recursive types.

Syntax :

type $name_1$ **=** $typedef_1$
and $name_2$ **=** $typedef_2$
⋮
and $name_n$ **=** $typedef_n$ **;;**

Type declarations can be parameterized by type variables. A type variable name always begins with an apostrophe (the **'** character):

Syntax : **type** *'a name* **=** *typedef* **;;**

When there are several of them, the type parameters are declared as a tuple in front of the name of the type:

Syntax : **type (**$'a_1$ ... $'a_n$**)** *name* **=** *typedef* **;;**

Only the type parameters defined on the left-hand side of the declaration may appear on the right-hand side.

Note

Objective Caml's type printer renames the type parameters encountered; the first is called *'a*, the second *'b* and so forth.

One can always define a new type from one or more existing types.

Syntax : **type** *name* **=** *type expression*

This is useful for constraining a type which one finds too general.

```
# type 'param paired_with_integer = int * 'param ;;
type 'a paired_with_integer = int * 'a
# type specific_pair = float paired_with_integer ;;
type specific_pair = float paired_with_integer
```

Nevertheless without type constraints, inference will produce the most general type.

```
# let x = (3, 3.14) ;;
val x : int * float = 3, 3.14
```

But one can use a type constraint to see the desired name appear:

```
# let (x:specific_pair) = (3, 3.14) ;;
val x : specific_pair = 3, 3.14
```

Records

Records are tuples, each of whose fields is named in the same way as the Pascal *record* or the C *struct*. A record always corresponds to the declaration of a new type. A record type is defined by the declaration of its name and the names and types of each of its fields.

Syntax : **type** *name* **=** { *name*$_1$ **:** *t*$_1$**;** ... **;** *name*$_n$ **:** *t*$_n$ } **;;**

We can define a type representing complex numbers by:

```
# type complex = { re:float; im:float } ;;
type complex = { re: float; im: float }
```

The creation of a value of record type is done by giving a value to each of its fields (in arbitrary order).

Syntax : { *name*$_{i_1}$ **=** *expr*$_{i_1}$**;** ... **;** *name*$_{i_n}$ **=** *expr*$_{i_n}$ } **;;**

For example, we create a complex number with real part `2.` and imaginary part `3.`:

```
# let c = {re=2.;im=3.} ;;
val c : complex = {re=2; im=3}
# c = {im=3.;re=2.} ;;
- : bool = true
```

In the case where some fields are missing, the following error is produced:

```
# let d = { im=4. } ;;
Characters 9-18:
Some labels are undefined
```

A field can be accessed in two ways: by the dot notation or by pattern matching on certain fields.

The dot notation syntax is as usual:

Syntax : *expr.name*

The expression *expr* must be of a record type containing a field *name*.

Pattern matching a record lets one retrieve the value bound to several fields. A pattern to match a record has the following syntax:

Syntax : { *name*$_i$ **=** *p*$_i$ **;** ... **;** *name*$_j$ **=** *p*$_j$ }

The patterns are to the right of the **=** sign (p_i, ..., p_j). It is not necessary to make all the fields of a record appear in such a pattern.

The function `add_complex` accesses fields through the dot notation, while the function `mult_complex` accesses them through pattern matching.

```
# let add_complex c1 c2 = {re=c1.re+.c2.re; im=c1.im+.c2.im};;
val add_complex : complex -> complex -> complex = <fun>
# add_complex c c ;;
- : complex = {re=4; im=6}
# let mult_complex c1 c2 = match (c1,c2) with
    ({re=x1;im=y1},{re=x2;im=y2}) → {re=x1*.x2-.y1*.y2;im=x1*.y2+.x2*.y1} ;;
val mult_complex : complex -> complex -> complex = <fun>
# mult_complex c c ;;
- : complex = {re=-5; im=12}
```

The advantages of records, as opposed to tuples, are at least twofold:

- descriptive and distinguishing information thanks to the field names: in particular this allows pattern matching to be simplified;
- access in an identical way, by name, to any field of the record whatsoever: the order of the fields no longer has significance, only their names count.

The following example shows the ease of accessing the fields of records as opposed to tuples:

```
# let a = (1,2,3) ;;
val a : int * int * int = 1, 2, 3
# let f tr = match tr with  x,_,_ → x ;;
val f : 'a * 'b * 'c -> 'a = <fun>
# f a ;;
- : int = 1
# type triplet = {x1:int; x2:int; x3:int} ;;
type triplet = { x1: int; x2: int; x3: int }
# let b = {x1=1; x2=2; x3=3} ;;
val b : triplet = {x1=1; x2=2; x3=3}
# let g tr = tr.x1 ;;
val g : triplet -> int = <fun>
# g b ;;
- : int = 1
```

For pattern matching, it is not necessary to indicate all the fields of the record being matched. The inferred type is then that of the last field.

```
# let h tr = match tr with  {x1=x} → x;;
val h : triplet -> int = <fun>
# h b;;
- : int = 1
```

There is a construction which lets one create a record identical to another except for some fields. It is often useful for records containing many fields.

Syntax : { *name* **with** $name_i$**=** $expr_i$ **;** ... **;** $name_j$**=**$expr_j$}

```
# let c = {b with x1=0} ;;
val c : triplet = {x1=0; x2=2; x3=3}
```

A new copy of the value of `b` is created where only the field `x1` has a new value.

Warning This feature is among the extensions to the language and may change in future versions.

Sum types

In contrast with tuples or records, which correspond to a Cartesian product, the declaration of a sum type corresponds to a union of sets. Different types (for example integers or character strings) are gathered into a single type. The various members of the sum are distinguished by *constructors*, which support on the one hand, as their name indicates, *construction* of values of this type and on the other hand, thanks to pattern matching, *access* to the components of these values. To apply a constructor to an argument is to indicate that the value returned belongs to this new type.

A sum type is declared by giving the names of its constructors and the types of their eventual arguments.

Syntax :

type *name* **=** ...
| $Name_i$ ...
| $Name_j$ **of** t_j ...
| $Name_k$ **of** t_k ***** ...***** t_l ... **;;**

A constructor name is a particular identifier:

Warning The names of constructors always begin with a capital letter.

Constant constructors

A constructor which doesn't expect an argument is called a *constant constructor*. Constant constructors can subsequently be used directly as a value in the language, as a constant.

```
# type coin = Heads | Tails;;
type coin = | Heads | Tails
# Tails;;
- : coin = Tails
```

The type `bool` can be defined in this way.

Constructors with arguments

Constructors can have arguments. The keyword **of** indicates the type of the constructor's arguments. This supports the gathering into a single type of objects of different types, each one being introduced with a particular constructor.

Here is a classic example of defining a datatype to represent the cards in a game, here Tarot[8]. The types *suit* and *card* are defined in the following way:

```
# type suit = Spades | Hearts | Diamonds | Clubs ;;
# type card =
       King of suit
     | Queen of suit
     | Knight of suit
     | Knave of suit
     | Minor_card of suit * int
     | Trump of int
     | Joker ;;
```

The creation of a value of type *card* is carried out through the application of a constructor to a value of the appropriate type.

```
# King Spades ;;
- : card = King Spades
# Minor_card(Hearts, 10) ;;
- : card = Minor_card (Hearts, 10)
# Trump 21 ;;
- : card = Trump 21
```

And here, for example, is the function `all_cards` which constructs a list of all the cards of a suit passed as a parameter.

```
# let rec interval a b =  if a = b then [b] else a :: (interval (a+1) b) ;;
val interval : int -> int -> int list = <fun>
# let all_cards s =
     let face_cards = [ Knave s; Knight s; Queen s; King s ]
     and other_cards  = List.map (function n → Minor_card(s,n)) (interval 1 10)
     in face_cards @ other_cards ;;
val all_cards : suit -> card list = <fun>
# all_cards Hearts ;;
- : card list =
[Knave Hearts; Knight Hearts; Queen Hearts; King Hearts;
 Minor_card (Hearts, 1); Minor_card (Hearts, 2); Minor_card (Hearts, 3);
 Minor_card (Hearts, ...); ...]
```

8. Translator's note: The rules for French Tarot can be found, for example, at
Link: http://www.pagat.com/tarot/frtarot.html

To handle values of sum types, we use pattern matching. The following example constructs conversion functions from values of type *suit* and of type *card* to character strings (type *string*):

```
# let string_of_suit = function
        Spades   → "spades"
      | Diamonds → "diamonds"
      | Hearts   → "hearts"
      | Clubs    → "clubs"  ;;
val string_of_suit : suit -> string = <fun>
# let string_of_card = function
        King c            → "king of " ^ (string_of_suit c)
      | Queen c           → "queen of " ^ (string_of_suit c)
      | Knave c           → "knave of " ^ (string_of_suit c)
      | Knight c          → "knight of " ^ (string_of_suit c)
      | Minor_card (c, n) → (string_of_int n) ^ " of "^(string_of_suit c)
      | Trump n           → (string_of_int n) ^ " of trumps"
      | Joker             → "joker" ;;
val string_of_card : card -> string = <fun>
```

Lining up the patterns makes these functions easy to read.

The constructor `Minor_card` is treated as a constructor with *two* arguments. Pattern matching on such a value requires naming its two components.

```
# let is_minor_card c = match c with
      Minor_card v →  true
    | _ → false;;
Characters 41-53:
The constructor Minor_card expects 2 argument(s),
but is here applied to 1 argument(s)
```

To avoid having to name each component of a constructor, one declares it to have a single argument by parenthesizing the corresponding tuple type. The two constructors which follow are pattern-matched differently.

```
# type t =
        C of int * bool
      | D of (int * bool) ;;
# let access v = match v with
        C (i, b) → i,b
      | D x  → x;;
val access : t -> int * bool = <fun>
```

Recursive types

Recursive type definitions are indispensable in any algorithmic language for describing the usual data structures (lists, heaps, trees, graphs, etc.). To this end, in Objective Caml type definition is recursive by default, in contrast with value declaration (**let**).

Objective Caml's predefined type of lists only takes a single parameter. One may wish to store values of belonging to two different types in a list structure, for example, integers (*int*) or characters (*char*). In this case, one defines:

```
# type int_or_char_list =
        Nil
      | Int_cons of int * int_or_char_list
      | Char_cons of char * int_or_char_list ;;

# let l1 =  Char_cons ( '=', Int_cons(5, Nil) )  in
     Int_cons ( 2, Char_cons ( '+', Int_cons(3, l1) ) )  ;;
- : int_or_char_list =
Int_cons (2, Char_cons ('+', Int_cons (3, Char_cons ('=', Int_cons (...)))))
```

Parametrized types

A user can equally well declare types with parameters. This lets us generalize the example of lists containing values of two different types.

```
# type ('a, 'b) list2 =
        Nil
      | Acons of 'a * ('a, 'b) list2
      | Bcons of 'b * ('a, 'b) list2 ;;

# Acons(2, Bcons('+', Acons(3, Bcons('=', Acons(5, Nil))))) ;;
- : (int, char) list2 =
Acons (2, Bcons ('+', Acons (3, Bcons ('=', Acons (...)))))
```

One can, obviously, instantiate the parameters *'a* and *'b* with the same type.

```
# Acons(1, Bcons(2, Acons(3, Bcons(4, Nil)))) ;;
- : (int, int) list2 = Acons (1, Bcons (2, Acons (3, Bcons (4, Nil))))
```

This use of the type *list2* can, as in the preceding example, serve to mark even integers and odd integers. In this way we extract the sublist of even integers in order to construct an ordinary list.

```
# let rec extract_odd = function
       Nil → []
     | Acons(_, x) → extract_odd x
     | Bcons(n, x) → n :: (extract_odd x) ;;
val extract_odd : ('a, 'b) list2 -> 'b list = <fun>
```

The definition of this function doesn't give a single clue as to the nature of the values stored in the structure. That is why its type is parameterized.

Scope of declarations

Constructor names obey the same scope discipline as global declarations: a redefinition masks the previous one. Nevertheless values of the masked type still exist. The interactive toplevel does not distinguish these two types in its output. Whence some unclear error messages.

In this first example, the constant constructor `Nil` of type *int_or_char* has been masked by the constructor declarations of the type *('a, 'b) list2*.

```
# Int_cons(0, Nil) ;;
Characters 13-16:
This expression has type ('a, 'b) list2 but is here used with type
  int_or_char_list
```

This second example provokes a rather baffling error message, at least the first time it appears. Let the little program be as follows:

```
# type t1 = Empty | Full;;
type t1 = | Empty | Full
# let empty_t1 x = match x with Empty → true | Full → false ;;
val empty_t1 : t1 -> bool = <fun>
# empty_t1 Empty;;
- : bool = true
```

Then, we redeclare the type *t1*:

```
# type t1 = {u : int; v : int} ;;
type t1 = { u: int; v: int }
# let y = { u=2; v=3 } ;;
val y : t1 = {u=2; v=3}
```

Now if we apply the function `empty_t1` to a value of the new type *t1*, we get the following error message:

```
# empty_t1 y;;
Characters 10-11:
This expression has type t1 but is here used with type t1
```

The first occurrence of *t1* represents the first type defined, while the second corresponds to the second type.

Function types

The type of the argument of a constructor may be arbitrary. In particular, it may very well contain a function type. The following type constructs lists, all of whose elements except the last are function values.

```
# type 'a listf =
      Val of 'a
    | Fun of ('a → 'a) * 'a listf ;;
```

```
type 'a listf = | Val of 'a | Fun of ('a -> 'a) * 'a listf
```

Since function values are values which can be manipulated in the language, we can construct values of type *listf*:

```
# let eight_div = (/) 8 ;;
val eight_div : int -> int = <fun>
# let gl = Fun (succ, (Fun (eight_div, Val 4))) ;;
val gl : int listf = Fun (<fun>, Fun (<fun>, Val 4))
```

and functions which pattern-match such values:

```
# let rec compute = function
      Val v → v
    | Fun(f, x) → f (compute x) ;;
val compute : 'a listf -> 'a = <fun>
# compute gl;;
- : int = 3
```

Example: representing trees

Tree structures come up frequently in programming. Recursive types make it easy to define and manipulate such structures. In this subsection, we give two examples of tree structures.

Binary trees We define a binary tree structure whose nodes are labelled with values of a single type by declaring:

```
# type 'a bin_tree =
      Empty
    | Node of 'a bin_tree * 'a * 'a bin_tree ;;
```

We use this structure to define a little sorting program using binary search trees. A binary search tree has the property that all the values in the left branch are less than that of the root, and all those of the right branch are greater. Figure 2.5 gives an example of such a structure over the integers. The empty nodes (constructor `Empty`) are represented there by little squares; the others (constructor `Node`), by a circle in which is inscribed the stored value.

A sorted list is extracted from a binary search tree via an inorder traversal carried out by the following function:

```
# let rec list_of_tree = function
      Empty → []
    | Node(lb, r, rb) → (list_of_tree lb) @ (r :: (list_of_tree rb)) ;;
val list_of_tree : 'a bin_tree -> 'a list = <fun>
```

Figure 2.5: Binary search tree.

To obtain a binary search tree from a list, we define an insert function.

```
# let rec insert x = function
      Empty  →  Node(Empty, x, Empty)
    | Node(lb, r, rb)  →  if x < r then Node(insert x lb, r, rb)
                          else Node(lb, r, insert x rb) ;;
val insert : 'a -> 'a bin_tree -> 'a bin_tree = <fun>
```

The function to transform a list into a tree is obtained by iterating the function `insert`.

```
# let rec tree_of_list = function
        []    →  Empty
     | h :: t  →  insert h (tree_of_list t)  ;;
val tree_of_list : 'a list -> 'a bin_tree = <fun>
```

The sort function is then simply the composition of the functions `tree_of_list` and `list_of_tree`.

```
# let sort x = list_of_tree (tree_of_list x) ;;
val sort : 'a list -> 'a list = <fun>
# sort [5; 8; 2; 7; 1; 0; 3; 6; 9; 4] ;;
- : int list = [0; 1; 2; 3; 4; 5; 6; 7; 8; 9]
```

General planar trees In this part, we use the following predefined functions from the `List` module (see page 217):

- `List.map`: which applies a function to all the elements of a list and returns the list of results;

- `List.fold_left`: which is an equivalent version of the function `fold_left` defined on page 33;
- `List.exists`: which applies a boolean-valued function to all the elements of a list; if one of these applications yields `true` then the result is `true`, otherwise the function returns `false`.

A *general planar tree* is a tree whose number of branches is not fixed *a priori*; to each node is associated a list of branches whose length may vary.

```
# type 'a tree = Empty
                | Node of 'a * 'a tree list ;;
```

The empty tree is represented by the value `Empty`. A leaf is a node without branches either of the form `Node(x,[])`, or of the degenerate form `Node(x, [Empty;Empty; ..])`. It is then relatively easy to write functions to manipulate these trees, e.g., to determine whether an element belongs to a tree or compute the height of the tree.

To test membership of an element `e`, we use the following algorithm: if the tree is empty then `e` does not belong to this tree, otherwise `e` belongs to the tree if and only if either it is equal to the label of the root, or it belongs to one of its branches.

```
# let rec belongs e = function
     Empty → false
   | Node(v, bs) → (e=v) or (List.exists (belongs e) bs) ;;
val belongs : 'a -> 'a tree -> bool = <fun>
```

To compute the height of a tree, we use the following definition: an empty tree has height `0`, otherwise the height of the tree is equal to the height of its highest subtree plus `1`.

```
# let rec height =
   let max_list l = List.fold_left max 0 l in
    function
      Empty → 0
    | Node (_, bs) → 1 + (max_list (List.map height bs)) ;;
val height : 'a tree -> int = <fun>
```

Recursive values which are not functions

Recursive declaration of non-function values allows the construction of circular data structures.

The following declaration constructs a circular list with one element.

```
# let rec l = 1 :: l ;;
val l : int list =
  [1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; 1; ...]
```

Application of a recursive function to such a list risks looping until memory overflows.

```
# size l ;;
Stack overflow during evaluation (looping recursion?).
```

Structural equality remains usable with such lists only when physical equality is first verified:

```
# l=l ;;
- : bool = true
```

In short, if you define a new list, even an equal one, you must not use the structural equality test on pain of seeing your program loop indefinitely. So we don't recommend attempting to evaluate the following example:

```
let rec l2 = 1::l2 in l=l2 ;;
```

On the other hand, physical equality always remains possible.

```
# let rec l2 = 1 :: l2 in l==l2 ;;
- : bool = false
```

The predicate `==` tests equality of an immediate value or sharing of a structured object (equality of the address of the value). We will use it to verify that in traversing a list we don't retraverse a sublist which was already examined. First of all, we define the function `memq`, which verifies the presence of an element in the list by relying on physical equality. It is the counterpart to the function `mem` which tests structural equality; these two functions belong to the module `List`.

```
# let rec memq a l = match l with
      [] → false
    | b :: l → (a==b) or (memq a l) ;;
val memq : 'a -> 'a list -> bool = <fun>
```

The size computation function is redefined, storing the list of lists already examined and halting if a list is encountered a second time.

```
# let special_size l =
    let rec size_aux previous l = match l with
        [] → 0
      | _::l1 → if memq l previous then 0
                        else 1 + (size_aux (l :: previous) l1)
    in size_aux [] l ;;
val special_size : 'a list -> int = <fun>
# special_size [1;2;3;4] ;;
- : int = 4
# special_size l ;;
- : int = 1
# let rec l1 = 1 :: 2 :: l2 and l2 = 1 :: 2 :: l1 in special_size l1 ;;
- : int = 4
```

Typing, domain of definition, and exceptions

The inferred type of a function corresponds to a subset of its domain of definition. Just because a function takes a parameter of type *int* doesn't mean it will know how to compute a value for all integers passed as parameters. In general this problem is dealt with using Objective Caml's exception mechanism. Raising an exception results in a computational interruption which can be intercepted and handled by the program. For this to happen program execution must have registered an exception handler before the computation of the expression which raises this exception.

Partial functions and exceptions

The domain of definition of a function corresponds to the set of values on which the function carries out its computation. There are many mathematical functions which are partial; we might mention division or taking the natural log. This problem also arises for functions which manipulate more complex data structures. Indeed, what is the result of computing the first element of an empty list? In the same way, evaluation of the `factorial` function on a negative integer can lead to an infinite recursion.

Several exceptional situations may arise during execution of a program, for example an attempt to divide by zero. Trying to divide a number by zero will provoke at best a program halt, at worst an inconsistent machine state. The *safety* of a programming language comes from the guarantee that such a situation will not arise for these particular cases. Exceptions are a way of responding to them.

Division of `1` by `0` will cause a specific exception to be raised:

```
# 1/0;;
Uncaught exception: Division_by_zero
```

The message `Uncaught exception: Division_by_zero` indicates on the one hand that the `Division_by_zero` exception has been raised, and on the other hand that it has not been handled. This exception is among the core declarations of the language.

Often, the type of a function does not correspond to its domain of definition when a pattern-matching is not exhaustive, that is, when it does not match all the cases of a given expression. To prevent such an error, Objective Caml prints a message in such a case.

```
# let head l = match l with h::t -> h ;;
Characters 14-36:
Warning: this pattern-matching is not exhaustive.
Here is an example of a value that is not matched:
[]
val head : 'a list -> 'a = <fun>
```

If the programmer nevertheless keeps the incomplete definition, Objective Caml will use the exception mechanism in the case of an erroneous call to the partial function:

```
# head [] ;;
Uncaught exception: Match_failure("", 14, 36)
```

Finally, we have already met with another predefined exception: `Failure`. It takes an argument of type *string*. One can raise this exception using the function `failwith`. We can use it in this way to complete the definition of our `head`:

```
# let head = function
      [] → failwith "Empty list"
   | h :: t → h;;
val head : 'a list -> 'a = <fun>
# head [] ;;
Uncaught exception: Failure("Empty list")
```

Definition of an exception

In Objective Caml, exceptions belong to a predefined type *exn*. This type is very special since it is an *extensible* sum type: the set of values of the type can be extended by declaring new constructors[9]. This detail lets users define their own exceptions by adding new constructors to the type *exn*.

The syntax of an exception declaration is as follows:

Syntax : **exception** *Name* **;;**

or

Syntax : **exception** *Name* **of** *t* **;;**

Here are some examples of exception declarations:

```
# exception MY_EXN;;
exception MY_EXN
# MY_EXN;;
- : exn = MY_EXN
# exception Depth of int;;
exception Depth of int
# Depth 4;;
- : exn = Depth(4)
```

Thus an exception is a full-fledged language value.

9. Translator's note: Thanks to the new "polymorphic variants" feature of Objective Caml `3.00`, some other sum types can now be extended as well

Warning The names of exceptions are constructors. So they necessarily begin with a capital letter.

```
# exception lowercase ;;
Characters 11-20:
Syntax error
```

Warning Exceptions are monomorphic: they do not have type parameters in the declaration of the type of their argument.

```
# exception Value of 'a ;;
Characters 20-22:
Unbound type parameter 'a
```

A polymorphic exception would permit the definition of functions with an arbitrary return type as we will see further on, page 58.

Raising an exception

The function `raise` is a primitive function of the language. It takes an exception as an argument and has a completely polymorphic return type.

```
# raise ;;
- : exn -> 'a = <fun>
# raise MY_EXN;;
Uncaught exception: MY_EXN
# 1+(raise MY_EXN);;
Uncaught exception: MY_EXN
# raise (Depth 4);;
Uncaught exception: Depth(4)
```

It is not possible to write the function `raise` in Objective Caml. It must be predefined.

Exception handling

The whole point of raising exceptions lies in the ability to handle them and to direct the sequence of computation according to the value of the exception raised. The order of evaluation of an expression thus becomes important for determining which exception is raised. We are leaving the purely functional context, and entering a domain where the order of evaluation of arguments can change the result of a computation, as will be discussed in the following chapter (see page 85).

The following syntactic construct, which computes the value of an expression, permits the handling of an exception raised during this computation:

Syntax :

try *expr* **with**
| p_1 -> $expr_1$
⋮
| p_n -> $expr_n$

If the evaluation of *expr* does not raise any exception, then the result is that of the evaluation of *expr*. Otherwise, the value of the exception which was raised is pattern-matched; the value of the expression corresponding to the first matching pattern is returned. If none of the patterns corresponds to the value of the exception then the latter is propagated up to the next outer **try-with** entered during the execution of the program. Thus pattern matching an exception is always considered to be exhaustive. Implicitly, the last pattern is `| e -> raise e`. If no matching exception handler is found in the program, the system itself takes charge of intercepting the exception and terminates the program while printing an error message.

One must not confuse *computing* an exception (that is, a value of type ***exn***) with *raising* an exception which causes computation to be interrupted. An exception being a value like others, it can be returned as the result of a function.

```
# let return x = Failure x ;;
val return : string -> exn = <fun>
# return "test" ;;
- : exn = Failure("test")
# let my_raise x = raise (Failure x) ;;
val my_raise : string -> 'a = <fun>
# my_raise "test" ;;
Uncaught exception: Failure("test")
```

We note that applying `my_raise` does not return any value while applying `return` returns one of type ***exn***.

Computing with exceptions

Beyond their use for handling exceptional values, exceptions also support a specific programming style and can be the source of optimizations. The following example finds the product of all the elements of a list of integers. We use an exception to interrupt traversal of the list and return the value `0` when we encounter it.

```
# exception Found_zero ;;
exception Found_zero
# let rec mult_rec l = match l with
       [] → 1
    | 0 :: _ → raise Found_zero
    | n :: x → n * (mult_rec x) ;;
val mult_rec : int list -> int = <fun>
# let mult_list l =
     try mult_rec l with Found_zero → 0 ;;
val mult_list : int list -> int = <fun>
# mult_list [1;2;3;0;5;6] ;;
```

```
- : int = 0
```

So all the computations standing by, namely the multiplications by n which follow each of the recursive calls, are abandoned. After encountering **raise**, computation resumes from the pattern-matching under **with**.

Polymorphism and return values of functions

Objective Caml's parametric polymorphism permits the definition of functions whose return type is completely unspecified. For example:

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
```

However, the return type depends on the type of the argument. Thus, when the function `id` is applied to an argument, the type inference mechanism knows how to instantiate the type variable *'a*. So for each particular use, the type of `id` can be determined.

If this were not so, it would no longer make sense to use strong static typing, entrusted with ensuring execution safety. Indeed, a function of completely unspecified type such as *'a -> 'b* would allow any type conversion whatsoever, which would inevitably lead to a run-time error since the physical representations of values of different types are not the same.

Apparent contradiction

However, it is possible in the Objective Caml language to define a function whose return type contains a type variable which does not appear in the types of its arguments. We will consider several such examples and see why such a possibility is not contradictory to strong static typing.

Here is a first example:

```
# let f x = [] ;;
val f : 'a -> 'b list = <fun>
```

This function lets us construct a polymorphic value from anything at all:

```
# f () ;;
- : '_a list = []
# f "anything at all" ;;
- : '_a list = []
```

Nevertheless, the value obtained isn't entirely unspecified: we're dealing with a list. So it can't be used just anywhere.

Here are three examples whose type is the dreaded *'a -> 'b*:

```
# let rec f1 x = f1 x ;;
val f1 : 'a -> 'b = <fun>
# let f2 x = failwith "anything at all" ;;
val f2 : 'a -> 'b = <fun>
# let f3 x = List.hd [] ;;
val f3 : 'a -> 'b = <fun>
```

These functions are not, in fact, dangerous vis-a-vis execution safety, since it isn't possible to use them to construct a value: the first one loops forever, the latter two raise an exception which interrupts the computation.

Similarly, it is in order to prevent functions of type *'a -> 'b* from being defined that new exception constructors are forbidden from having arguments whose type contains a variable.

Indeed, if one could declare a polymorphic exception `Poly_exn` of type *'a -> exn*, one could then write the function:

```
let f = function
   0 → raise (Poly_exn false)
 | n → n+1 ;;
```

The function `f` being of type *int -> int* and `Poly_exn` being of type *'a -> exn*, one could then define:

```
let g n = try f n with Poly_exn x → x+1 ;;
```

This function is equally well-typed (since the argument of `Poly_exn` may be arbitrary) and now, evaluation of `(g 0)` would end up in an attempt to add an integer and a boolean!

Desktop Calculator

To understand how a program is built in Objective Caml, it is necessary to develop one. The chosen example is a desktop calculator—that is, the simplest model, which only works on whole numbers and only carries out the four standard arithmetic operations.

To begin, we define the type *key* to represent the keys of a pocket calculator. The latter has fifteen keys, namely: one for each operation, one for each digit, and the `=` key.

```
# type key = Plus | Minus | Times | Div | Equals | Digit of int ;;
```

We note that the numeric keys are gathered under a single constructor *Digit* taking an integer argument. In fact, some values of type *key* don't actually represent a key. For example, `(Digit 32)` is a possible value of type *key*, but doesn't represent any of the calculator's keys.

So we write a function `valid` which verifies that its argument corresponds to a calculator key. The type of this function is *key -> bool*, that is, it takes a value of type *key* as argument and returns a value of type *bool*.

The first step is to define a function which verifies that an integer is included between 0 and 9. We declare this function under the name `is_digit`:

```
# let is_digit = function x → (x>=0) && (x<=9) ;;
val is_digit : int -> bool = <fun>
```

We then define the function `valid` by pattern-matching over its argument of type ***key***:

```
#   let valid ky = match ky with
      Digit n → is_digit n
    | _ → true ;;
val valid : key -> bool = <fun>
```

The first pattern is applied when the argument of `valid` is a value made with the ***Digit*** constructor; in this case, the argument of ***Digit*** is tested by the function `is_digit`. The second pattern is applied to every other kind of value of type ***key***. Recall that thanks to typing, the value being matched is necessarily of type ***key***.

Before setting out to code the calculator mechanism, we will specify a *model* allowing us to describe from a formal point of view the reaction to the activation of one of the device's keys. We will consider a pocket calculator to have four registers in which are stored respectively the last computation done, the last key activated, the last operator activated, and the number printed on the screen. The set of these four registers is called the state of the calculator; it is modified by each keypress on the keypad. This modification is called a transition and the theory governing this kind of mechanism is that of automata. A state will be represented in our program by a record type:

```
# type state = {
    lcd : int; (* last computation done   *)
    lka : key; (* last key activated      *)
    loa : key; (* last operator activated *)
    vpr : int  (* value printed           *)
  } ;;
```

Figure 2.6 gives an example of a sequence of transitions.

	state	key
	$(0, =, =, 0)$	3
$\longrightarrow$	$(0, 3, =, 3)$	$+$
$\longrightarrow$	$(3, +, +, 3)$	2
$\longrightarrow$	$(3, 2, +, 2)$	1
$\longrightarrow$	$(3, 1, +, 21)$	$\times$
$\longrightarrow$	$(24, *, *, 24)$	2
$\longrightarrow$	$(24, 2, *, 2)$	$=$
$\longrightarrow$	$(48, =, =, 48)$	

Figure 2.6: Transitions for $3 + 21 * 2 =$.

In what follows we need the function `evaluate` which takes two integers and a value of type `key` containing an operator and which returns the result of the operation corresponding to the key, applied to the integers. This function is defined by pattern-matching over its last argument, of type ***key***:

```
# let evaluate x y ky = match ky with
      Plus   → x + y
    | Minus  → x - y
    | Times  → x * y
    | Div    → x / y
    | Equals → y
    | Digit _ → failwith "evaluate : no op";;
val evaluate : int -> int -> key -> int = <fun>
```

Now we give the definition of the transition function by enumerating all possible cases. We assume that the current state is the quadruplet $(a, b, \oplus, d)$:

- a key with digit x is pressed, then there are two cases to consider:
  - the last key pressed was also a digit. So it is a number which the user of the pocket calculator is in the midst of entering; consequently the digit x must be affixed to the printed value, i.e., replacing it with $d \times 10 + x$. The new state is:
    $$(a, \texttt{(Digit x)}, \oplus, d \times 10 + x)$$
  - the last key pressed was not a digit. So it is the start of a new number which is being entered. The new state is:
    $$(a, \texttt{(Digit x)}, \oplus, x)$$
- a key with operator $\otimes$ has been pressed, the second operand of the operation has thus been completely entered and the calculator has to deal with carrying out this operation. It is to this end that the last operation (here $\oplus$) is stored. The new state is:
  $$(\oplus d, \otimes, \otimes, a \oplus d)$$

To write the function `transition`, it suffices to translate the preceding definition word for word into Objective Caml: the definition by cases becomes a definition by pattern-matching over the key passed as an argument. The case of a key, which itself is made up of two cases, is handled by the local function `digit_transition` by pattern-matching over the last key activated.

```
# let transition st ky =
    let digit_transition n = function
        Digit _ → { st with lka=ky; vpr=st.vpr*10+n }
      | _       → { st with lka=ky; vpr=n }
    in
       match ky with
           Digit p  →  digit_transition p st.lka
         | _        →  let res = evaluate st.lcd st.vpr st.loa
                       in { lcd=res; lka=ky; loa=ky; vpr=res }  ;;
```

```
val transition : state -> key -> state = <fun>
```
This function takes a *state* and a *key* and computes the new *state*.

We can now test this program on the previous example:
```
# let initial_state = { lcd=0; lka=Equals; loa=Equals; vpr=0 } ;;
val initial_state : state = {lcd=0; lka=Equals; loa=Equals; vpr=0}
# let state2 = transition initial_state (Digit 3) ;;
val state2 : state = {lcd=0; lka=Digit 3; loa=Equals; vpr=3}
# let state3 = transition state2 Plus ;;
val state3 : state = {lcd=3; lka=Plus; loa=Plus; vpr=3}
# let state4 = transition state3 (Digit 2) ;;
val state4 : state = {lcd=3; lka=Digit 2; loa=Plus; vpr=2}
# let state5 = transition state4 (Digit 1) ;;
val state5 : state = {lcd=3; lka=Digit 1; loa=Plus; vpr=21}
# let state6 = transition state5 Times ;;
val state6 : state = {lcd=24; lka=Times; loa=Times; vpr=24}
# let state7 = transition state6 (Digit 2) ;;
val state7 : state = {lcd=24; lka=Digit 2; loa=Times; vpr=2}
# let state8 = transition state7 Equals ;;
val state8 : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}
```

This run can be written in a more concise way using a function applying a sequence of transitions corresponding to a list of keys passed as an argument.

```
# let transition_list st ls = List.fold_left transition st ls ;;
val transition_list : state -> key list -> state = <fun>
# let example = [ Digit 3; Plus; Digit 2; Digit 1; Times; Digit 2; Equals ]
  in transition_list initial_state example ;;
- : state = {lcd=48; lka=Equals; loa=Equals; vpr=48}
```

Exercises

Merging two lists

1. Write a function `merge_i` which takes as input two integer lists sorted in increasing order and returns a new sorted list containing the elements of the first two.
2. Write a general function `merge` which takes as argument a comparison function and two lists sorted in this order and returns the list merged in the same order. The comparison function will be of type *'a → 'a → bool*.
3. Apply this function to two integer lists sorted in decreasing order, then to two string lists sorted in decreasing order.
4. What happens if one of the lists is not in the required decreasing order?

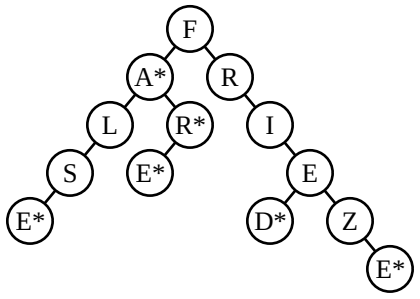
5. Write a new *list* type in the form of a record containing three fields: the conventional list, an order function and a boolean indicating whether the list is in that order.
6. Write the function `insert` which adds an element to a list of this type.
7. Write a function `sort` which insertion sorts the elements of a list.
8. Write a new function `merge` for these lists.

Lexical trees

Lexical trees (or *tries*) are used for the representation of dictionaries.

```
# type lex_node = Letter of char * bool * lex_tree
  and  lex_tree = lex_node  list;;
# type word = string;;
```

The boolean value in *lex_node* marks the end of a word when it equals **true**. In such a structure, the sequence of words "fa, false, far, fare, fried, frieze" is stored in the following way:



An asterisk (*) marks the end of a word.

1. Write the function `exists` which tests whether a word belongs to a dictionary of type *lex_tree*.
2. Write a function `insert` which takes a word and a dictionary and returns a new dictionary which additionally contains this word. If the word is already in the dictionary, it is not necessary to insert it.
3. Write a function `construct` which takes a list of words and constructs the corresponding dictionary.
4. Write a function `verify` which takes a list of words and a dictionary and returns the list of words not belonging to this dictionary.
5. Write a function `select` which takes a dictionary and a length and returns the set of words of this length.

Graph traversal

We define a type *'a graph* representing directed graphs by adjacency lists containing for each vertex the list of its successors:

```
# type 'a graph = ( 'a * 'a  list) list ;;
```

1. Write a function `insert_vtx` which inserts a vertex into a graph and returns the new graph.
2. Write a function `insert_edge` which adds an edge to a graph already possessing these two vertices.
3. Write a function `has_edges_to` which returns all the vertices following directly from a given vertex.
4. Write a function `has_edges_from` which returns the list of all the vertices leading directly to a given vertex.

Summary

This chapter has demonstrated the main features of functional programming and parametric polymorphism, which are two essential features of the Objective Caml language. The syntax of the expressions in the functional core of the language as well as those of the types which have been described allowed us to develop our first programs. Moreover, the profound difference between the type of a function and its domain of definition was underlined. Introducing the exception mechanism allowed us to resolve this problem and already introduces a new programming style in which one specifies how computations should unfold.

To learn more

The computation model for functional languages is λ-calculus, which was invented by Alonzo Church in 1932. Church's goal was to define a notion of effective computability through the medium of λ-definability. Later, it became apparent that the notion thus introduced was equivalent to the notions of computability in the sense of Turing (Turing machine) and Gödel-Herbrand (recursive functions). This coïncidence leads one to think that there exists a universal notion of computability, independent of particular formalisms: this is Church's thesis. In this calculus, the only two constructions are abstraction and application. Data structures (integers, booleans, pairs, ...) can be coded by λ-termes.

Functional languages, of which the first representative was Lisp, implement this model and extend it mainly with more efficient data structures. For the sake of efficiency, the first functional languages implemented physical modifications of memory, which among other things forced the evaluation strategy to be immediate, or strict, evaluation. In this strategy, the arguments of functions are evaluated before being passed to the

function. It is in fact later, for other languages such as Miranda, Haskell, or LML, that the strategy of delayed (lazy, or call-by-need) evaluation was implemented for pure functional languages.

Static typing, with type inference, was promoted by the ML family at the start of the 80's. The web page

Link: http://www.pps.jussieu.fr/~cousinea/Caml/caml_history.html

presents a historical overview of the ML language. Its computation model is typed λ-calculus, a subset of λ-calculus. It guarantees that no type error will occur during program execution. Nevertheless "completely correct" programs can be rejected by ML's type system. These cases seldom arise and these programs can always be rewritten in such a way as to conform to the type system.

The two most-used functional languages are Lisp and ML, representatives of impure functional languages. To deepen the functional approach to programming, the books [ASS96] and [CM98] each present a general programming course using the languages Scheme (a dialect of Lisp) and Caml-Light, respectively.

3

Imperative Programming

In contrast to functional programming, in which you calculate a value by applying a function to its arguments without caring how the operations are carried out, imperative programming is closer to the machine representation, as it introduces memory state which the execution of the program's actions will modify. We call these actions of programs *instructions*, and an imperative program is a list, or *sequence*, of instructions. The execution of each operation can alter the memory state. We consider input-output actions to be modifications of memory, video memory, or files.

This style of programming is directly inspired by assembly programming. You find it in the earliest general-purpose programming languages (Fortran, C, Pascal, etc.). In Objective Caml the following elements of the language fit into this model:

- modifiable data structures, such as arrays, or records with mutable fields;
- input-output operations;
- control structures such as loops and exceptions.

Certain algorithms are easier to write in this programming style. Take for instance the computation of the product of two matrices. Even though it is certainly possible to translate it into a purely functional version, in which lists replace vectors, this is neither natural nor efficient compared to an imperative version.

The motivation for the integration of imperative elements into a functional language is to be able to write certain algorithms in this style when it is appropriate. The two principal disadvantages, compared to the purely functional style, are:

- complicating the type system of the language, and rejecting certain programs which would otherwise be considered correct;
- having to keep track of the memory representation and of the order of calculations.

Nevertheless, with a few guidelines in writing programs, the choice between several programming styles offers the greatest flexibility for writing algorithms, which is the principal objective of any programming language. Besides, a program written in a style which is close to the algorithm used will be simpler, and hence will have a better chance of being correct (or at least, rapidly correctable).

For these reasons, the Objective Caml language has some types of data structures whose values are physically modifiable, structures for controlling the execution of programs, and an I/O library in an imperative style.

Plan of the Chapter

This chapter continues the presentation of the basic elements of the Objective Caml language begun in the previous chapter, but this time focusing on imperative constructions. There are five sections. The first is the most important; it presents the different modifiable data structures and describes their memory representation. The second describes the basic I/O of the language, rather briefly. The third section is concerned with the new iterative control structures. The fourth section discusses the impact of imperative features on the execution of a program, and in particular on the order of evaluation of the arguments of a function. The final section returns to the calculator example from the last chapter, to turn it into a calculator with a memory.

Modifiable Data Structures

Values of the following types: vectors, character strings, records with mutable fields, and references are the data structures whose parts can be physically modified.

We have seen that an Objective Caml variable bound to a value keeps this value to the end of its lifetime. You can only modify this binding with a redefinition—in which case we are not really talking about the "same" variable; rather, a new variable of the same name now masks the old one, which is no longer directly accessible, but which remains unchanged. With modifiable values, you can change the value associated with a variable without having to redeclare the latter. You have access to the value of a variable for writing as well as for reading.

Vectors

Vectors, or one dimensional arrays, collect a known number of elements of the same type. You can write a vector directly by listing its values between the symbols `[|` and `|]`, separated by semicolons as for lists.

```
# let v = [| 3.14; 6.28; 9.42 |] ;;
val v : float array = [|3.14; 6.28; 9.42|]
```

The creation function `Array.create` takes the number of elements in the vector and an initial value, and returns a new vector.

```
# let v = Array.create  3  3.14;;
val v : float array = [|3.14; 3.14; 3.14|]
```

To access or modify a particular element, you give the index of that element:

Syntax : *expr*$_1$ **.** **(** *expr*$_2$ **)**

Syntax : *expr*$_1$ **.** **(** *expr*$_2$ **)** **<-** *expr*$_3$

expr$_1$ should be a vector (type ***array***) whose values have type *expr*$_3$. The expression *expr*$_2$ must, of course, have type ***int***. The modification is an expression of type ***unit***. The first element of a vector has index 0 and the index of the last element is the length of the vector minus 1. The parentheses around the index expression are required.

```
# v.(1) ;;
- : float = 3.14
# v.(0) <- 100.0 ;;
- : unit = ()
# v ;;
- : float array = [|100; 3.14; 3.14|]
```

If the index used to access an element in an array is outside the range of indices of the array, an exception is raised at the moment of access.

```
# v.(-1) +. 4.0;;
Uncaught exception: Invalid_argument("Array.get")
```

This check is done during program execution, which can slow it down. Nevertheless it is essential, in order to avoid writing to memory outside the space allocated to a vector, which would cause serious execution errors.

The functions for manipulating arrays are part of the `Array` module in the standard library. We'll describe them in chapter 8 (page 217). In the examples below, we will use the following three functions from the `Array` module:

- `create` which creates an array of the given size with the given initial value;
- `length` which gives the length of a vector;
- `append` which concatenates two vectors.

Sharing of Values in a Vector

All the elements of a vector contain the value that was passed in when it was created. This implies a sharing of this value, if it is a structured value. For example, let's create a matrix as a vector of vectors using the function `create` from the `Array` module.

```
# let v = Array.create 3 0;;
val v : int array = [|0; 0; 0|]
# let m = Array.create 3 v;;
```

```
val m : int array array = [|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]
```

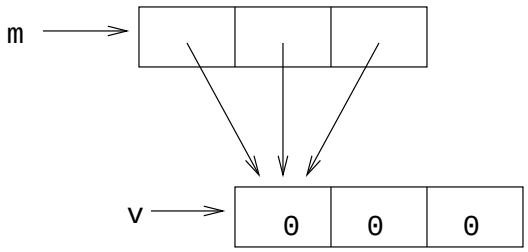


Figure 3.1: Memory representation of a vector sharing its elements.

If you modify one of the fields of vector `v`, which was used in the creation of `m`, then you automatically modify all the "rows" of the matrix together (see figures 3.1 and 3.2).

```
# v.(0) <- 1;;
- : unit = ()
# m;;
- : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
```

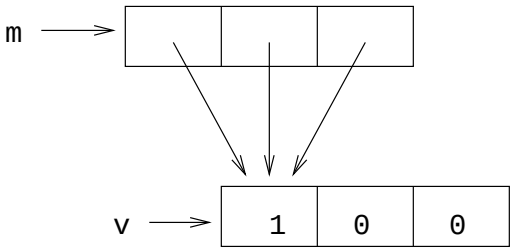


Figure 3.2: Modification of shared elements of a vector.

Duplication occurs if the initialization value of the vector (the second argument passed to `Array.create`) is an *atomic value* and there is sharing if this value is a structured value.

Values whose size does not exceed the standard size of Objective Caml values—that is, the memory word—are called atomic values. These are the integers, characters, booleans, and constant constructors. The other values—structured values—are represented by a pointer into a memory area. This distinction is detailed in chapter 9 (page 247).

Vectors of floats are a special case. Although floats are structured values, the creation of a vector of floats causes the the initial value to be copied. This is for reasons of optimization. Chapter 12, on the interface with the C language (page 315), describes this special case.

Non-Rectangular Matrices

A matrix, a vector of vectors, does not need not to be rectangular. In fact, nothing stops you from replacing one of the vector elements with a vector of a different length. This is useful to limit the size of such a matrix. The following value `t` constructs a triangular matrix for the coefficients of Pascal's triangle.

```
# let t = [|
            [|1|];
            [|1; 1|];
            [|1; 2; 1|];
            [|1; 3; 3; 1|];
            [|1; 4; 6; 4; 1|];
            [|1; 5; 10; 10; 5; 1|]
          |] ;;
val t : int array array =
  [|[|1|]; [|1; 1|]; [|1; 2; 1|]; [|1; 3; 3; 1|]; [|1; 4; 6; 4; ...|]; ...|]
# t.(3) ;;
- : int array = [|1; 3; 3; 1|]
```

In this example, the element of vector `t` with index i is a vector of integers with size $i + 1$. To manipulate such matrices, you have to calculate the size of each element vector.

Copying Vectors

When you copy a vector, or when you concatenate two vectors, the result obtained is a new vector. A modification of the original vectors does not result in the modification of the copies, unless, as usual, there are shared values.

```
# let v2 = Array.copy v ;;
val v2 : int array = [|1; 0; 0|]
# let m2 = Array.copy m ;;
val m2 : int array array = [|[|1; 0; 0|]; [|1; 0; 0|]; [|1; 0; 0|]|]
# v.(1)<- 352;;
- : unit = ()
# v2;;
- : int array = [|1; 0; 0|]
# m2 ;;
- : int array array = [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
```

We notice in this example that copying `m` only copies the pointers to `v`. If one of the elements of `v` is modified, `m2` is modified too.

Concatenation creates a new vector whose size is equal to the sum of the sizes of the two others.

```
# let mm = Array.append m m ;;
val mm : int array array =
  [|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
    [|1; 352; ...|]; ...|]
# Array.length mm ;;
```

```
- : int = 6
# m.(0) <- Array.create 3 0 ;;
- : unit = ()
# m ;;
- : int array array = [|[|0; 0; 0|]; [|1; 352; 0|]; [|1; 352; 0|]|]
# mm ;;
- : int array array =
[|[|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|]; [|1; 352; 0|];
  [|1; 352; ...|]; ...|]
```

On the other hand, modification of `v`, a value shared by `m` and `mm`, does affect both these matrices.

```
# v.(1) <- 18 ;;
- : unit = ()
# mm;;
- : int array array =
[|[|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; 0|]; [|1; 18; ...|];
  ...|]
```

Character Strings

Character strings can be considered a special case of vectors of characters. Nevertheless, for efficient memory usage[1] their type is specialized. Moreover, access to their elements has a special syntax:

Syntax : $expr_1$ **.** **[**$expr_2$**]**

The elements of a character string can be physically modified:

Syntax : $expr_1$ **.** **[**$expr_2$**]** **<-** $expr_3$

```
# let s = "hello";;
val s : string = "hello"
# s.[2];;
- : char = 'l'
# s.[2]<-'Z';;
- : unit = ()
# s;;
- : string = "heZlo"
```

1. A 32-bit word contains four characters coded as bytes

Mutable Fields of Records

Fields of a record can be declared mutable. All you have to do is to show this in the declaration of the type of the record using the keyword **mutable**.

Syntax : **type** *name* **=** { ... **; mutable** $name_i$ **:** t **;** ... }

Here is a small example defining a record type for points in the plane:

```
# type point = { mutable xc : float; mutable yc : float } ;;
type point = { mutable xc: float; mutable yc: float }
# let p = { xc = 1.0; yc = 0.0 } ;;
val p : point = {xc=1; yc=0}
```

Thus the value of a field which is declared **mutable** can be modified using the syntax:

Syntax : $expr_1$ **.** *name* **<-** $expr_2$

The expression $expr_1$ should be a record type which has the field *name*. The modification operator returns a value of type ***unit***.

```
# p.xc <- 3.0 ;;
- : unit = ()
# p ;;
- : point = {xc=3; yc=0}
```

We can write a function for moving a point by modifying its components. We use a local declaration with pattern matching in order to sequence the side-effects.

```
# let moveto p dx dy =
    let () = p.xc <- p.xc +. dx
    in p.yc <- p.yc +. dy ;;
val moveto : point -> float -> float -> unit = <fun>
# moveto p 1.1 2.2 ;;
- : unit = ()
# p ;;
- : point = {xc=4.1; yc=2.2}
```

It is possible to mix mutable and non-mutable fields in the definition of a record. Only those specified as **mutable** may be modified.

```
# type t = { c1 : int; mutable c2 : int } ;;
type t = { c1: int; mutable c2: int }
# let r = { c1 = 0; c2 = 0 } ;;
val r : t = {c1=0; c2=0}
# r.c1 <- 1 ;;
Characters 0-9:
The label c1 is not mutable
# r.c2 <- 1 ;;
- : unit = ()
```

```
# r ;;
- : t = {c1=0; c2=1}
```

On page 82 we give an example of using records with modifiable fields and arrays to implement a stack structure.

References

Objective Caml provides a polymorphic type `ref` which can be seen as the type of a pointer to any value; in Objective Caml terminology we call it a *reference* to a value. A referenced value can be modified. The type *`ref`* is defined as a record with one modifiable field:

```
type 'a ref = {mutable contents:'a}
```

This type is provided as a syntactic shortcut. We construct a reference to a value using the function `ref`. The referenced value can be reached using the prefix function **`(!)`**. The function modifying the content of a reference is the infix function **`(:=)`**.

```
# let x = ref 3 ;;
val x : int ref = {contents=3}
# x ;;
- : int ref = {contents=3}
# !x ;;
- : int = 3
# x := 4 ;;
- : unit = ()
# !x ;;
- : int = 4
# x := !x+1 ;;
- : unit = ()
# !x ;;
- : int = 5
```

Polymorphism and Modifiable Values

The type *`ref`* is parameterized. This is what lets us use it to create references to values of any type whatever. However, it is necessary to place certain restrictions on the type of referenced values; we cannot allow the creation of a reference to a value with a polymorphic type without taking some precautions.

Let us suppose that there were no restriction; then someone could declare:

```
let x = ref [] ;;
```

Then the variable x would have type *'a list ref* and its value could be modified in a way which would be inconsistent with the strong static typing of Objective Caml:

```
x := 1 :: !x ;;
x := true :: !x ;;
```

Thus we would have one and the same variable having type *int list* at one moment and *bool list* the next.

In order to avoid such a situation, Objective Caml's type inference mechanism uses a new category of type variables: *weak type variables*. Syntactically, they are distinguished by the underscore character which prefixes them.

```
# let x = ref [] ;;
val x : '_a list ref = {contents=[]}
```

The type variable *'_a* is not a type parameter, but an unknown type awaiting instantiation; the first use of x after its declaration fixes the value that *'_a* will take in all types that depend on it, permanently.

```
# x := 0::!x ;;
- : unit = ()
# x ;;
- : int list ref = {contents=[0]}
```

From here onward, the variable x has type *int list ref*.

A type containing an unknown is in fact monomorphic even though its type has not been specified. It is not possible to instantiate this unknown with a polymorphic type.

```
# let x = ref [] ;;
val x : '_a list ref = {contents=[]}
# x := (function y → ())::!x ;;
- : unit = ()
# x ;;
- : ('_a -> unit) list ref = {contents=[<fun>]}
```

In this example, even though we have instantiated the unknown type with a type which is *a priori* polymorphic (*'a -> unit*), the type has remained monomorphic with a new unknown type.

This restriction of polymorphism applies not only to references, but to any value containing a modifiable part: vectors, records having at least one field declared **mutable**, etc. Thus all the type parameters, even those which have nothing to do with a modifiable part, are weak type variables.

```
# type ('a,'b) t = { ch1 :'a list ; mutable ch2 : 'b list } ;;
type ('a, 'b) t = { ch1: 'a list; mutable ch2: 'b list }
# let x = { ch1 = [] ; ch2 = [] } ;;
val x : ('_a, '_b) t = {ch1=[]; ch2=[]}
```

Warning This modification of the typing of application has consequences for pure functional programs.

Likewise, when you apply a polymorphic value to a polymorphic function, you get a weak type variable, because you must not exclude the possibility that the function may construct physically modifiable values. In other words, the result of the application is always monomorphic.

```
# (function x → x) [] ;;
- : '_a list = []
```

You get the same result with partial application:

```
# let f a b = a ;;
val f : 'a -> 'b -> 'a = <fun>
# let g = f 1 ;;
val g : '_a -> int = <fun>
```

To get a polymorphic type back, you have to abstract the second argument of `f` and then apply it:

```
# let h x = f 1 x ;;
val h : 'a -> int = <fun>
```

In effect, the expression which defines `h` is the functional expression **function** *x* → *f* `1` *x*. Its evaluation produces a closure which does not risk producing a side effect, because the body of the function is not evaluated.

In general, we distinguish so-called "non-expansive" expressions, whose calculation we are sure carries no risk of causing a side effect, from other expressions, called "expansive." Objective Caml's type system classifies expressions of the language according to their syntactic form:

- "non-expansive" expressions include primarily variables, constructors of non-mutable values, and abstractions;
- "expansive" expressions include primarily applications and constructors of modifiable values. We can also include here control structures like conditionals and pattern matching.

Input-Output

Input-output functions do calculate a value (often of type *unit*) but during their calculation they cause a modification of the state of the input-output peripherals: modification of the state of the keyboard *buffer*, outputting to the screen, writing in a file, or modification of a read pointer. The following two types are predefined: *in_channel* and *out_channel* for, respectively, input channels and output channels. When an end of file is met, the exception `End_of_file` is raised. Finally, the following three constants correspond to the standard channels for input, output, and error in Unix fashion: `stdin`, `stdout`, and `stderr`.

Channels

The input-output functions from the Objective Caml standard library manipulate *communication channels*: values of type ***in_channel*** or ***out_channel***. Apart from the three standard predefined values, the creation of a channel uses one of the following functions:

```
# open_in;;
- : string -> in_channel = <fun>
# open_out;;
- : string -> out_channel = <fun>
```

open_in opens the file if it exists[2], and otherwise raises the exception Sys_error. open_out creates the specified file if it does not exist or truncates it if it does.

```
# let ic = open_in "koala";;
val ic : in_channel = <abstr>
# let oc = open_out "koala";;
val oc : out_channel = <abstr>
```

The functions for closing channels are:

```
# close_in ;;
- : in_channel -> unit = <fun>
# close_out ;;
- : out_channel -> unit = <fun>
```

Reading and Writing

The most general functions for reading and writing are the following:

```
# input_line ;;
- : in_channel -> string = <fun>
# input ;;
- : in_channel -> string -> int -> int -> int = <fun>
# output ;;
- : out_channel -> string -> int -> int -> unit = <fun>
```

- ***input_line ic***: reads from input channel ic all the characters up to the first carriage return or end of file, and returns them in the form of a list of characters (excluding the carriage return).
- ***input ic s p l***: attempts to read l characters from an input channel ic and stores them in the list s starting from the p$^{\text{th}}$ character. The number of characters actually read is returned.
- ***output oc s p l***: writes on an output channel oc part of the list s, starting at the p-th character, with length l.

2. With appropriate read permissions, that is.

The following functions read from standard input or write to standard output:

```
# read_line ;;
- : unit -> string = <fun>
# print_string ;;
- : string -> unit = <fun>
# print_newline ;;
- : unit -> unit = <fun>
```

Other values of simple types can also be read directly or appended. These are the values of types which can be converted into lists of characters.

Local declarations and order of evaluation We can simulate a sequence of printouts with expressions of the form `let x = e1 in e2`. Knowing that, in general, `x` is a local variable which can be used in `e2`, we know that `e1` is evaluated first and then comes the turn of `e2`. If the two expressions are imperative functions whose results are `()` but which have side effects, then we have executed them in the right order. In particular, since we know the return value of `e1`—the constant () of type *unit*—we get a sequence of printouts by writing the sequence of nested declarations which pattern match on ().

```
# let () = print_string "and one," in
  let () = print_string " and two," in
  let () = print_string " and three" in
     print_string " zero";;
and one, and two, and three zero- : unit = ()
```

Example: Higher/Lower

The following example concerns the game "Higher/Lower" which consists of choosing a number which the user must guess at. The program indicates at each turn whether the chosen number is smaller or bigger than the proposed number.

```
#  let rec hilo n =
      let () = print_string "type a number: " in
      let i = read_int ()
      in
        if i = n then
          let () = print_string "BRAVO" in
          let () = print_newline ()
          in print_newline ()
        else
          let () =
            if i < n then
```

```
              let () = print_string "Higher"
              in print_newline ()
            else
              let () = print_string "Lower"
              in print_newline ()
          in hilo n ;;
val hilo : int -> unit = <fun>
```

Here is an example session:

```
# hilo 64;;
type a number: 88
Lower
type a number: 44
Higher
type a number: 64
BRAVO

- : unit = ()
```

Control Structures

Input-output and modifiable values produce side-effects. Their use is made easier by an imperative programming style furnished with new control structures. We present in this section the sequence and iteration structures.

We have already met the conditional control structure on page 18, whose abbreviated form **if then** patterns itself on the imperative world. We will write, for example:

```
# let n = ref 1 ;;
val n : int ref = {contents=1}
# if !n > 0 then n := !n - 1 ;;
- : unit = ()
```

Sequence

The first of the typically imperative structures is the *sequence*. This permits the left-to-right evaluation of a sequence of expressions separated by semicolons.

Syntax : $expr_1$ **;** ... **;** $expr_n$

A sequence of expressions is itself an expression, whose value is that of the last expression in the sequence (here, $expr_n$). Nevertheless, all the expressions are evaluated, and in particular their side-effects are taken into account.

```
# print_string "2 = "; 1+1 ;;
```

```
2 = - : int = 2
```

With side-effects, we get back the usual construction of imperative languages.

```
# let x = ref 1 ;;
val x : int ref = {contents=1}
# x:=!x+1 ; x:=!x*4 ; !x ;;
- : int = 8
```

As the value preceding a semicolon is discarded, Objective Caml gives a warning when it is not of type *unit*.

```
# print_int 1; 2 ; 3 ;;
Characters 14-15:
Warning: this expression should have type unit.
1- : int = 3
```

To avoid this message, you can use the function `ignore`:

```
# print_int 1; ignore 2; 3 ;;
1- : int = 3
```

A different message is obtained if the value has a functional type, as Objective Caml suspects that you have forgotten a parameter of a function.

```
# let g x y = x := y ;;
val g : 'a ref -> 'a -> unit = <fun>
# let a = ref 10;;
val a : int ref = {contents=10}
# let u = 1 in g a ; g a u ;;
Characters 13-16:
Warning: this function application is partial,
maybe some arguments are missing.
- : unit = ()
# let u = !a in ignore (g a) ; g a u ;;
- : unit = ()
```

As a general rule we parenthesize sequences to clarify their scope. Syntactically, parenthesizing can take two forms:

Syntax : **(** *expr* **)**

Syntax : **begin** *expr* **end**

We can now write the Higher/Lower program from page 78 more naturally:

```
# let rec hilo n =
    print_string "type a number: ";
    let i = read_int () in
```

```
    if i = n then print_string "BRAVO\n\n"
     else
       begin
         if i < n then print_string "Higher\n" else print_string "Lower\n"  ;
         hilo n
       end ;;
val hilo : int -> unit = <fun>
```

Loops

The iterative control structures are also from outside the functional world. The conditional expression for repeating, or leaving, a loop does not make sense unless there can be a physical modification of the memory which permits its value to change. There are two iterative control structures in Objective Caml: the **for** loop for a bounded iteration and the **while** loop for a non-bounded iteration. The loop structures themselves are expressions of the language. Thus they return a value: the constant **()** of type *unit*.

The **for** loop can be rising (**to**) or falling (**downto**) with a step of one.

Syntax :

for *name* **=** $expr_1$ **to** $expr_2$ **do** $expr_3$ **done**
for *name* **=** $expr_1$ **downto** $expr_2$ **do** $expr_3$ **done**

The expressions $expr_1$ and $expr_2$ are of type *int*. If $expr_3$ is not of type *unit*, the compiler produces a warning message.

```
# for i=1 to 10 do  print_int i; print_string " " done; print_newline() ;;
1 2 3 4 5 6 7 8 9 10
- : unit = ()
# for i=10 downto 1 do print_int i; print_string " " done; print_newline() ;;
10 9 8 7 6 5 4 3 2 1
- : unit = ()
```

The non-bounded loop is the "while" loop whose syntax is:

Syntax : **while** $expr_1$ **do** $expr_2$ **done**

The expression $expr_1$ should be of type *bool*. And, as for the **for** loop, if $expr_2$ is not of type *unit*, the compiler produces a warning message.

```
# let r = ref 1
  in while !r < 11  do
       print_int !r ;
       print_string " " ;
       r := !r+1
     done ;;
1 2 3 4 5 6 7 8 9 10 - : unit = ()
```

It is important to understand that loops are expressions like the previous ones which calculate the value **()** of type *unit*.

```
# let f () = print_string "-- end\n" ;;
val f : unit -> unit = <fun>
# f (for i=1 to 10 do  print_int i; print_string " " done) ;;
1 2 3 4 5 6 7 8 9 10 -- end
- : unit = ()
```

Note that the string `"-- end\n"` is output after the integers from 1 to 10 have been printed: this is a demonstration that the arguments (here the loop) are evaluated before being passed to the function.

In imperative programming, the body of a loop ($expr_2$) does not calculate a value, but advances by side effects. In Objective Caml, when the body of a loop is not of type *unit* the compiler prints a warning, as for the sequence:

```
# let s = [5; 4; 3; 2; 1; 0] ;;
val s : int list = [5; 4; 3; 2; 1; 0]
#  for i=0 to 5 do List.tl s done ;;
Characters 17-26:
Warning: this expression should have type unit.
- : unit = ()
```

Example: Implementing a Stack

The data structure *'a stack* will be implemented in the form of a record containing an array of elements and the first free position in this array. Here is the corresponding type:

```
# type 'a stack = { mutable ind:int; size:int; mutable elts : 'a array } ;;
```

The field `size` contains the maximal size of the stack.

The operations on these stacks will be `init_stack` for the initialization of a stack, `push` for pushing an element onto a stack, and `pop` for returning the top of the stack and popping it off.

```
# let init_stack n = {ind=0; size=n; elts =[||]} ;;
val init_stack : int -> 'a stack = <fun>
```

This function cannot create a non-empty array, because you would have to provide it with the value with which to construct it. This is why the field `elts` gets an empty array.

Two exceptions are declared to guard against attempts to pop an empty stack or to add an element to a full stack. They are used in the functions `pop` and `push`.

```
# exception Stack_empty ;;
# exception Stack_full ;;

# let pop p =
    if p.ind = 0 then raise Stack_empty
    else (p.ind <- p.ind - 1; p.elts.(p.ind)) ;;
```

```
val pop : 'a stack -> 'a = <fun>
# let push e p =
    if p.elts = [||] then
      (p.elts <- Array.create p.size e;
       p.ind <- 1)
    else if p.ind >= p.size then raise Stack_full
    else (p.elts.(p.ind) <- e; p.ind <- p.ind + 1) ;;
val push : 'a -> 'a stack -> unit = <fun>
```

Here is a small example of the use of this data structure:

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[||]}
# push 1 p ;;
- : unit = ()
# for i = 2 to 5 do push i p done ;;
Uncaught exception: Stack_full
# p ;;
- : int stack = {ind=4; size=4; elts=[|1; 2; 3; 4|]}
# pop p ;;
- : int = 4
# pop p ;;
- : int = 3
```

If we want to prevent raising the exception `Stack_full` when attempting to add an element to the stack, we can enlarge the array. To do this the field `size` must be modifiable too:

```
# type 'a stack =
    {mutable ind:int ; mutable size:int ; mutable elts : 'a array} ;;
# let init_stack n = {ind=0; size=max n 1; elts = [||]} ;;
# let n_push e p =
    if p.elts = [||]
    then
      begin
        p.elts <- Array.create p.size e;
        p.ind <- 1
      end
    else if p.ind >= p.size then
      begin
        let nt = 2 * p.size in
        let nv = Array.create nt e in
        for j=0 to p.size-1 do nv.(j) <- p.elts.(j) done ;
        p.elts <- nv;
        p.size <- nt;
        p.ind <- p.ind + 1
      end
    else
```

```
      begin
        p.elts.(p.ind) <- e ;
        p.ind <- p.ind + 1
      end ;;
val n_push : 'a -> 'a stack -> unit = <fun>
```

All the same, you have to be careful with data structures which can expand without bound. Here is a small example where the initial stack grows as needed.

```
# let p = init_stack 4 ;;
val p : '_a stack = {ind=0; size=4; elts=[||]}
# for i = 1 to 5 do n_push i p done ;;
- : unit = ()
# p ;;
- : int stack = {ind=5; size=8; elts=[|1; 2; 3; 4; 5; 5; 5; 5|]}
# p.stack ;;
Characters 0-7:
Unbound label stack
```

It might also be useful to allow `pop` to decrease the size of the stack, to reclaim unused memory.

Example: Calculations on Matrices

In this example we aim to define a type for matrices, two-dimensional arrays containing floating point numbers, and to write some operations on the matrices. The monomorphic type *mat* is a record containing the dimensions and the elements of the matrix. The functions `create_mat`, `access_mat`, and `mod_mat` are respectively the functions for creation, accessing an element, and modification of an element.

```
# type mat = { n:int; m:int; t: float array array };;
type mat = { n: int; m: int; t: float array array }
# let create_mat n m = { n=n; m=m; t = Array.create_matrix n m 0.0 } ;;
val create_mat : int -> int -> mat = <fun>
# let access_mat m i j = m.t.(i).(j) ;;
val access_mat : mat -> int -> int -> float = <fun>
# let mod_mat m i j e = m.t.(i).(j) <- e ;;
val mod_mat : mat -> int -> int -> float -> unit = <fun>
# let a = create_mat 3 3 ;;
val a : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 0; 0|]; [|0; 0; 0|]|]}
# mod_mat a 1 1 2.0; mod_mat a 1 2 1.0; mod_mat a 2 1 1.0 ;;
- : unit = ()
# a ;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 2; 1|]; [|0; 1; 0|]|]}
```

The sum of two matrices a and b is a matrix c such that $c_{ij} = a_{ij} + b_{ij}$.

```
# let add_mat p q =
    if p.n = q.n && p.m = q.m then
      let r = create_mat p.n p.m  in
      for i = 0 to p.n-1 do
        for j = 0 to p.m-1 do
          mod_mat r i j  (p.t.(i).(j) +. q.t.(i).(j))
        done
      done ;
      r
    else failwith "add_mat : dimensions incompatible";;
val add_mat : mat -> mat -> mat = <fun>
# add_mat  a a ;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 4; 2|]; [|0; 2; 0|]|]}
```

The product of two matrices a and b is a matrix c such that $c_{ij} = \sum_{k=1}^{k=m_a} a_{ik}.b_{kj}$

```
# let mul_mat p q =
    if p.m = q.n then
      let r = create_mat p.n q.m in
      for i = 0 to p.n-1 do
        for j = 0 to q.m-1 do
          let c = ref 0.0 in
          for k = 0 to p.m-1 do
             c := !c +.   (p.t.(i).(k) *. q.t.(k).(j))
           done;
           mod_mat r i j !c
        done
      done;
      r
    else failwith "mul_mat : dimensions incompatible" ;;
val mul_mat : mat -> mat -> mat = <fun>
# mul_mat a a;;
- : mat = {n=3; m=3; t=[|[|0; 0; 0|]; [|0; 5; 2|]; [|0; 2; 1|]|]}
```

Order of Evaluation of Arguments

In a pure functional language, the order of evaluation of the arguments does not matter. As there is no modification of memory state and no interruption of the calculation, there is no risk of the calculation of one argument influencing another. On the other hand, in Objective Caml, where there are physically modifiable values and exceptions, there is a danger in not taking account of the order of evaluation of arguments. The following example is specific to version 2.04 of Objective Caml for Linux on Intel hardware:

```
# let new_print_string s = print_string s; String.length s ;;
val new_print_string : string -> int = <fun>
```

```
# (+) (new_print_string "Hello ") (new_print_string "World!") ;;
World!Hello - : int = 12
```
The printing of the two strings shows that the second string is output before the first.

It is the same with exceptions:
```
# try (failwith "function") (failwith "argument") with Failure s → s;;
- : string = "argument"
```

If you want to specify the order of evaluation of arguments, you have to make local declarations forcing this order before calling the function. So the preceding example can be rewritten like this:
```
# let e1 = (new_print_string "Hello ")
  in let e2 = (new_print_string "World!")
  in (+) e1 e2 ;;
Hello World!- : int = 12
```

In Objective Caml, the order of evaluation of arguments is not specified. As it happens, today all implementations of Objective Caml evaluate arguments from left to right. All the same, making use of this implementation feature could turn out to be dangerous if future versions of the language modify the implementation.

We come back to the eternal debate over the design of languages. Should certain features of the language be deliberately left unspecified—should programmers be asked not to use them, on pain of getting different results from their program according to the compiler implementation? Or should everything be specified—should programmers be allowed to use the whole language, at the price of complicating compiler implementation, and forbidding certain optimizations?

Calculator With Memory

We now reuse the calculator example described in the preceding chapter, but this time we give it a user interface, which makes our program more usable as a desktop calculator. This loop allows entering operations directly and seeing results displayed without having to explicitly apply a transition function for each keypress.

We attach four new keys: **C**, which resets the display to zero, **M**, which memorizes a result, **m**, which recalls this memory and **OFF**, which turns off the calculator. This corresponds to the following type:
```
# type key = Plus | Minus | Times | Div | Equals | Digit of int
           | Store | Recall | Clear | Off ;;
```

It is necessary to define a translation function from characters typed on the keyboard to values of type *key*. The exception `Invalid_key` handles the case of characters that do not represent any key of the calculator. The function `code` of module `Char` translates a character to its ASCII-code.

```
# exception Invalid_key ;;
exception Invalid_key
# let translation c = match c with
      '+' → Plus
    | '-' → Minus
    | '*' → Times
    | '/' → Div
    | '=' → Equals
    | 'C' | 'c' → Clear
    | 'M' → Store
    | 'm' → Recall
    | 'o' | 'O' → Off
    | '0'..'9' as c → Digit ((Char.code c) - (Char.code '0'))
    | _ → raise Invalid_key ;;
val translation : char -> key = <fun>
```

In imperative style, the translation function does not calculate a new state, but physically modifies the state of the calculator. Therefore, it is necessary to redefine the type `state` such that the fields are modifiable. Finally, we define the exception `Key_off` for treating the activation of the key **OFF**.

```
# type state = {
   mutable lcd : int;  (* last computation done   *)
   mutable lka : bool; (* last key activated      *)
   mutable loa : key;  (* last operator activated *)
   mutable vpr : int;  (* value printed           *)
   mutable mem : int   (* memory of calculator    *)
  };;
```

```
# exception Key_off ;;
exception Key_off
# let transition s key = match key with
      Clear →   s.vpr <- 0
    | Digit n → s.vpr <- ( if s.lka then s.vpr*10+n else n );
                s.lka <- true
    | Store   →  s.lka <- false ;
                s.mem <- s.vpr
    | Recall →  s.lka <- false ;
                s.vpr <- s.mem
    | Off → raise Key_off
    |  _  → let lcd = match s.loa with
                        Plus  → s.lcd + s.vpr
                      | Minus → s.lcd - s.vpr
                      | Times  → s.lcd * s.vpr
                      | Div   → s.lcd / s.vpr
                      | Equals  → s.vpr
```

```
                        | _ → failwith "transition: impossible match"
                    in
                s.lcd <- lcd ;
                s.lka <- false ;
                s.loa <- key ;
                s.vpr <- s.lcd;;
val transition : state -> key -> unit = <fun>
```

We define the function go, which starts the calculator. Its return value is `()`, because we are only concerned about effects produced by the execution on the environment (start/end, modification of state). Its argument is also the constant `()`, because the calculator is autonomous (it defines its own initial state) and interactive (the arguments of the computation are entered on the keyboard as required). The transitions are performed within an infinite loop (**while** `true` **do**) so we can quit with the exception `Key_off`.

```
# let go () =
    let state = { lcd=0; lka=false; loa=Equals; vpr=0; mem=0 }
    in try
         while true do
           try
             let input = translation (input_char stdin)
             in transition state input ;
                print_newline () ;
                print_string "result: " ;
                print_int state.vpr ;
                print_newline ()
           with
             Invalid_key → ()  (* no effect *)
         done
       with
         Key_off → () ;;
val go : unit -> unit = <fun>
```

We note that the initial state must be either passed as a parameter or declared locally within the function go, because it needs to be initialized at every application of this function. If we had used a value `initial_state` as in the functional program, the calculator would start in the same state as the one it had when it was terminated. This would make it difficult to use two calculators in the same program.

Exercises

Doubly Linked Lists

Functional programming lends itself well to the manipulation of non-cyclic data structures, such as lists for example. For cyclic structures, on the other hand, there are real implementation difficulties. Here we propose to define doubly linked lists, i.e., where each element of a list knows its predecessor and its successor.

1. Define a parameterized type for doubly linked lists, using at least one record with mutable fields.
2. Write the functions `add` and `remove` which add and remove an element of a doubly linked list.

Solving linear systems

This exercise has to do with matrix algebra. It solves a system of equations by Gaussian elimination (i.e., pivoting). We write the system of equations $A\ X = Y$ with A, a square matrix of dimension n, Y, a vector of constants of dimension n and X, a vector of unknowns of the same dimension.

This method consists of transforming the system $A\ X = Y$ into an equivalent system $C\ X = Z$ such that the matrix C is upper triangular. We diagonalize C to obtain the solution.

1. Define a type *`vect`*, a type *`mat`*, and a type *`syst`* .
2. Write utility functions for manipulating vectors: to display a system on screen, to add two vectors, to multiply a vector by a scalar.
3. Write utility functions for matrix computations: multiplication of two matrices, product of a matrix with a vector.
4. Write utility functions for manipulating systems: division of a row of a system by a pivot, (A_{ii}), swapping two rows.
5. Write a function to diagonalize a system. From this, obtain a function solving a linear system.
6. Test your functions on the following systems:

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8 & 7 \\ 7 & 5 & 6 & 5 \\ 8 & 6 & 10 & 9 \\ 7 & 5 & 9 & 10 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32.1 \\ 22.9 \\ 33.1 \\ 30.9 \end{pmatrix} = Y$$

$$AX = \begin{pmatrix} 10 & 7 & 8.1 & 7.2 \\ 7.08 & 5.04 & 6 & 5 \\ 8 & 5.98 & 9.89 & 9 \\ 6.99 & 4.99 & 9 & 9.98 \end{pmatrix} * \begin{pmatrix} x_1 \\ x_2 \\ x_3 \\ x_4 \end{pmatrix} = \begin{pmatrix} 32 \\ 23 \\ 33 \\ 31 \end{pmatrix} = Y$$

7. What can you say about the results you got?

Summary

This chapter has shown the integration of the main features of imperative programming (mutable values, I/O, iterative control structures) into a functional language. Only mutable values, such as strings, arrays, and records with mutable fields, can be physically modified. Other values, once created, are immutable. In this way we obtain read-only (RO) values for the functional part and read-write (RW) values for the imperative part.

It should be noted that, if we don't make use of the imperative features of the language, this extension to the functional core does not change the functional part, except for typing considerations which we can get around.

To Learn More

Imperative programming is the style of programming which has been most widely used since the first computer languages such as Fortran, C, or Pascal. For this reason numerous algorithms are described in this style, often using some kind of pseudo-Pascal. While they could be implemented in a functional style, the use of arrays promotes the use of an imperative style. The data structures and algorithms presented in classic algorithms books, such as [AHU83] and [Sed88], can be carried over directly in the appropriate style. An additional advantage of including these two styles in a single language is being able to define new programming models by mixing the two. This is precisely the subject of the next chapter.

4

Functional and Imperative Styles

Functional and imperative programming languages are primarily distinguished by the control over program execution and the data memory management.

- A functional program computes an expression. This computation results in a value. The order in which the operations needed for this computation occur does not matter, nor does the physical representation of the data manipulated, because the result is the same anyway. In this setting, deallocation of memory is managed implicitly by the language itself: it relies on an automatic garbage collector or *GC*; see chapter 9.
- An imperative program is a sequence of instructions modifying a memory state. Each execution step is enforced by rigid control structures that indicate the next instruction to be executed. Imperative programs manipulate pointers or references to values more often than the values themselves. Hence, the memory space needed to store values must be allocated and reclaimed explicitly, which sometimes leads to errors in accessing memory. Nevertheless, nothing prevents use of a GC.

Imperative languages provide greater control over execution and the memory representation of data. Being closer to the actual machine, the code can be more efficient, but loses in execution safety. Functional programming, offering a higher level of abstraction, achieves a better level of execution safety: Typing (dynamic or static) may be stricter in this case, thus avoiding operations on incoherent values. Automatic storage reclamation, in exchange for giving up efficiency, ensures the current existence of the values being manipulated.

Historically, the two programming paradigms have been seen as belonging to different universes: symbolic applications being suitable for the former, and numerical applications being suitable for the latter. But certain things have changed, especially techniques for compiling functional programming languages, and the efficiency of GCs. From another side, execution safety has become an important, sometimes the predominant criterion in the quality of an application. Also familiar is the “selling point” of

the *Java* language, according to which efficiency need not preempt assurance, especially if efficiency remains reasonably good. And this idea is spreading among software producers.

Objective Caml belongs to this class. It combines the two programming paradigms, thus enlarging its domain of application by allowing algorithms to be written in either style. It retains, nevertheless, a good degree of execution safety because of its static typing, its GC, and its exception mechanism. Exceptions are a first explicit execution control structure; they make it possible to break out of a computation or restart it. This trait is at the boundary of the two models, because although it does not replace the result of a computation, it can modify the order of execution. Introducing physically mutable data can alter the behavior of the purely functional part of the language. For instance, the order in which the arguments to a function are evaluated can be determined, if that evaluation causes side effects. For this reason, such languages are called "impure functional languages." One loses in level of abstraction, because the programmer must take account of the memory model, as well as the order of events in running the program. This is not always negative, especially for the efficiency of the code. On the other hand, the imperative aspects change the type system of the language: some functional programs, correctly typed in theory, are no longer in fact correctly typed because of the introduction of references. However, such programs can easily be rewritten.

Plan of the Chapter

This chapter provides a comparison between the functional and imperative models in the Objective Caml language, at the level both of control structure and of the memory representation of values. The mixture of these two styles allows new data structures to be created. The first section studies this comparison by example. The second section discusses the ingredients in the choice between composition of functions and sequencing of instructions, and in the choice between sharing and copying values. The third section brings out the interest of mixing these two styles to create mutable functional data, thus permitting data to be constructed without being completely evaluated. The fourth section describes *streams*, potentially infinite sequences of data, and their integration into the language via pattern-matching.

Comparison between Functional and Imperative

Character strings (of Objective Caml type `string`) and linked lists (of Objective Caml type `'a list`) will serve as examples to illustrate the differences between "functional" and "imperative."

The Functional Side

The function `map` (see page 26) is a classic ingredient in functional languages. In a purely functional style, it is written:

```
# let rec map f l = match l with
        []    →   []
    | h :: q  →   (f h) :: (map f q)  ;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

It recursively constructs a list by applying `f` to the elements of the list given as argument, independently specifying its head `(f h)` and its tail `(map f q)`. In particular, the program does not stipulate which of the two will be computed first.

Moreover, the physical representation of lists need not be known to the programmer to write such a function. In particular, problems of allocating and sharing data are managed implicitly by the system and not by the programmer. An example illustrating this follows:

```
# let example = [ "one" ; "two" ; "three" ] ;;
val example : string list = ["one"; "two"; "three"]
# let result = map (function x → x) example  ;;
val result : string list = ["one"; "two"; "three"]
```

The lists `example` and `result` contain equal values:

```
# example = result ;;
- : bool = true
```

These two values have exactly the same structure even though their representation in memory is different, as one learns by using the test for physical equality:

```
# example == result ;;
- : bool = false
# (List.tl example) == (List.tl result) ;;
- : bool = false
```

The Imperative Side

Let us continue the previous example, and modify a string in the list `result`.

```
# (List.hd result).[1] <- 's' ;;
- : unit = ()
# result ;;
- : string list = ["ose"; "two"; "three"]
# example ;;
- : string list = ["ose"; "two"; "three"]
```

Evidently, this operation has modified the list `example`. Hence, it is necessary to know the physical structure of the two lists being manipulated, as soon as we use imperative aspects of the language.

Let us now observe how the order of evaluating the arguments of a function can amount to a trap in an imperative program. We define a mutable list structure with primitive functions for creation, modification, and access:

```
# type 'a ilist = { mutable c : 'a list } ;;
type 'a ilist = { mutable c: 'a list }
# let icreate () = { c = [] }
  let iempty l = (l.c = [])
  let icons x y = y.c <- x :: y.c ; y
  let ihd x = List.hd x.c
  let itl x = x.c <- List.tl x.c ; x ;;
val icreate : unit -> 'a ilist = <fun>
val iempty : 'a ilist -> bool = <fun>
val icons : 'a -> 'a ilist -> 'a ilist = <fun>
val ihd : 'a ilist -> 'a = <fun>
val itl : 'a ilist -> 'a ilist = <fun>
# let rec imap f l =
    if iempty l then icreate()
    else icons (f (ihd l))  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
```

Despite having reproduced the general form of the `map` of the previous paragraph, with `imap` we get a distinctly different result:

```
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
Uncaught exception: Failure("hd")
```

What has happened? Just that the evaluation of `(itl l)` has taken place before the evaluation of `(ihd l)`, so that on the last iteration of `imap`, the list referenced by `l` became the empty list before we examined its head. The list `example` is henceforth definitely empty even though we have not obtained any result:

```
# example ;;
- : string ilist = {c=[]}
```

The flaw in the function `imap` arises from a mixing of the genres that has not been controlled carefully enough. The choice of order of evaluation has been left to the system. We can reformulate the function `imap`, making explicit the order of evaluation, by using the syntactic construction **`let .. in ..`**

```
# let rec imap f l =
    if iempty l then icreate()
    else let h = ihd l  in  icons (f h)  (imap f (itl l)) ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
```

```
- : string ilist = {c=["one"; "two"; "three"]}
```

However, the original list has still been lost:

```
# example ;;
- : string ilist = {c=[]}
```

Another way to make the order of evaluation explicit is to use the sequencing operator and a looping structure.

```
# let imap f l =
    let l_res = icreate ()
    in while not (iempty l) do
         ignore (icons (f (ihd l)) l_res) ;
         ignore (itl l)
       done ;
       { l_res with c = List.rev l_res.c } ;;
val imap : ('a -> 'b) -> 'a ilist -> 'b ilist = <fun>
# let example = icons "one" (icons "two" (icons "three" (icreate()))) ;;
val example : string ilist = {c=["one"; "two"; "three"]}
# imap (function x → x) example ;;
- : string ilist = {c=["one"; "two"; "three"]}
```

The presence of `ignore` emphasizes the fact that it is not the result of the functions that counts here, but their side effects on their argument. In addition, we had to put the elements of the result back in the right order (using the function `List.rev`).

Recursive or Iterative

People often mistakenly associate recursive with functional and iterative with imperative. A purely functional program cannot be iterative because the value of the condition of a loop never varies. By contrast, an imperative program may be recursive: the original version of the function `imap` is an example.

Calling a function conserves the values of its arguments during its computation. If it calls another function, the latter conserves its own arguments in addition. These values are conserved on the *execution stack*. When the call returns, these values are popped from the stack. The memory space available for the stack being bounded, it is possible to encounter the limit when using a recursive function with calls too deeply nested. In this case, Objective Caml raises the exception `Stack_overflow`.

```
# let rec succ n = if n = 0 then 1 else 1 + succ (n-1) ;;
val succ : int -> int = <fun>
# succ 100000 ;;
Stack overflow during evaluation (looping recursion?).
```

In the iterative version `succ_iter`, the stack space needed for a call does not depend on its argument.

```
# let succ_iter n =
    let i = ref 0 in
      for j=0 to n do incr i done ;
      !i ;;
val succ_iter : int -> int = <fun>
# succ_iter 100000 ;;
- : int = 100001
```

The following recursive version has *a priori* the same depth of calls, yet it executes successfully with the same argument.

```
# let succ_tr n =
    let rec succ_aux n accu =
      if n = 0 then accu  else succ_aux (n-1) (accu+1)
    in
      succ_aux 1 n ;;
val succ_tr : int -> int = <fun>
# succ_tr 100000 ;;
- : int = 100001
```

This function has a special form of recursive call, called *tail recursion*, in which the result of this call will be the result of the function without further computation. It is therefore unnecessary to have stored the values of the arguments to the function while computing the recursive call. When Objective Caml can observe that a call is tail recursive, it frees the arguments on the stack before making the recursive call. This optimization allows recursive functions that do not increase the size of the stack.

Many languages detect tail recursive calls, but it is indispensable in a functional language, where naturally many tail recursive calls are used.

Which Style to Choose?

This is no matter of religion or esthetics; *a priori* neither style is prettier or holier than the other. On the contrary, one style may be more adequate than the other depending on the problem to be solved.

The first rule to apply is the rule of simplicity. Whether the algorithm to use implemented is written in a book, or whether its seed is in the mind of the programmer, the algorithm is itself described in a certain style. It is natural to use the same style when implementing it.

The second criterion of choice is the efficiency of the program. One may say that an imperative program (if well written) is more efficient that its functional analogue, but in very many cases the difference is not enough to justify complicating the code to

adopt an imperative style where the functional style would be natural. The function `map` in the previous section is a good example of a problem naturally expressed in the functional style, which gains nothing from being written in the imperative style.

Sequence or Composition of Functions

We have seen that as soon as a program causes side effects, it is necessary to determine precisely the order of evaluation for the elements of the program. This can be done in both styles:

functional: using the fact that Objective Caml is a *strict language*, which means that the argument is evaluated before applying the function. The expression `(f (g x))` is computed by first evaluating `(g x)`, and then passing the result as argument to `f`. With more complex expressions, we can name an intermediate result with the **`let in`** construction, but the idea remains the same: **`let`** *`aux`*`=(g x)` **`in`** `(f aux)`.

imperative: using sequences or other control structures (loops). In this case, the result is not the value returned by a function, but its side effects on memory: `aux:=(g x) ; (f !aux)`.

Let us examine this choice of style on an example. The *quick sort* algorithm, applied to a vector, is described recursively as follows:

1. Choose a pivot: This is the index of an element of the vector;
2. Permute around the pivot: Permute the elements of the vector so elements less than the value at the pivot have indices less than the pivot, and vice versa;
3. sort the subvectors obtained on each side of the pivot, using the same algorithm: The subvector preceding the pivot and the subvector following the pivot.

The choice of algorithm, namely to modify a vector so that its elements are sorted, incites us to use an imperative style at least to manipulate the data.

First, we define a function to permute two elements of a vector:

```
# let permute_element vec n p =
    let aux = vec.(n) in vec.(n) <- vec.(p) ; vec.(p) <- aux  ;;
val permute_element : 'a array -> int -> int -> unit = <fun>
```

The choice of a good pivot determines the efficiency of the algorithm, but we will use the simplest possible choice here: return the index of the first element of the (sub)vector.

```
# let choose_pivot vec start finish = start ;;
val choose_pivot : 'a -> 'b -> 'c -> 'b = <fun>
```

Let us write the algorithm that we would like to use to permute the elements of the vector around the pivot.

1. Place the pivot at the beginning of the vector to be permuted;
2. Initialize i to the index of the second element of the vector;
3. Initialize j to the index of the last element of the vector;
4. If the element at index j is greater than the pivot, permute it with the element at index i and increment i; otherwise, decrement j;
5. While $i < j$, repeat the previous operation;
6. At this stage, every element with index $< i$ (or equivalently, j) is less than the pivot, and all others are greater; if the element with index i is less than the pivot, permute it with the pivot; otherwise, permute its predecessor with the pivot.

In implementing this algorithm, it is natural to adopt imperative control structures.

```
# let permute_pivot vec start finish ind_pivot =
      permute_element vec start ind_pivot ;
      let i = ref (start+1) and j = ref finish and pivot = vec.(start) in
      while !i < !j do
        if vec.(!j) >= pivot then decr j
        else
          begin
            permute_element vec !i !j ;
            incr i
          end
      done ;
      if vec.(!i) > pivot then decr i ;
      permute_element vec start !i ;
      !i
   ;;
val permute_pivot : 'a array -> int -> int -> int -> int = <fun>
```

In addition to its effects on the vector, this function returns the index of the pivot as its result.

All that remains is to put together the different stages and add the recursion on the sub-vectors.

```
# let rec quick vec start finish =
      if start < finish
      then
        let pivot = choose_pivot vec start finish in
        let place_pivot = permute_pivot vec start finish pivot in
        quick (quick vec start (place_pivot-1)) (place_pivot+1) finish
      else vec ;;
val quick : 'a array -> int -> int -> 'a array = <fun>
```

We have used the two styles here. The chosen pivot serves as argument to the permutation around this pivot, and the index of the pivot after the permutation is an argument to the recursive call. By contrast, the vector obtained after the permutation is not returned by the `permute_pivot` function; instead, this result is produced by side

effect. However, the `quick` function returns a vector, and the sorting of sub-vectors is obtained by composition of recursive calls.

The main function is:

```
# let quicksort vec = quick vec 0 ((Array.length vec)-1) ;;
val quicksort : 'a array -> 'a array = <fun>
```

It is a polymorphic function because the order relation `<` on vector elements is itself polymorphic.

```
# let t1 = [|4;8;1;12;7;3;1;9|] ;;
val t1 : int array = [|4; 8; 1; 12; 7; 3; 1; 9|]
# quicksort t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# t1 ;;
- : int array = [|1; 1; 3; 4; 7; 8; 9; 12|]
# let t2 = [|"the"; "little"; "cat"; "is"; "dead"|] ;;
val t2 : string array = [|"the"; "little"; "cat"; "is"; "dead"|]
# quicksort t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
# t2 ;;
- : string array = [|"cat"; "dead"; "is"; "little"; "the"|]
```

Shared or Copy Values

When the values that we manipulate are not mutable, it does not matter whether they are shared or not.

```
# let id x = x ;;
val id : 'a -> 'a = <fun>
# let a = [ 1; 2; 3 ] ;;
val a : int list = [1; 2; 3]
# let b = id a ;;
val b : int list = [1; 2; 3]
```

Whether `b` is a copy of the list `a` or the very same list makes no difference, because these are intangible values anyway. But if we put modifiable values in place of integers, we need to know whether modifying one value causes a change in the other.

The implementation of polymorphism in Objective Caml causes immediate values to be copied, and structured values to be shared. Even though arguments are always passed by value, only the pointer to a structured value is copied. This is the case even in the function `id`:

```
# let a = [| 1 ; 2 ; 3 |] ;;
val a : int array = [|1; 2; 3|]
# let b = id a ;;
val b : int array = [|1; 2; 3|]
# a.(1) <- 4 ;;
- : unit = ()
# a ;;
```

```
- : int array = [|1; 4; 3|]
# b ;;
- : int array = [|1; 4; 3|]
```

We have here a genuine programming choice to decide which is the most efficient way to represent a data structure. On one hand, using mutable values allows manipulations in place, which means without allocation, but requires us to make copies sometimes when immutable data would have allowed sharing. We illustrate this here with two ways to implement lists.

```
# type 'a list_immutable = LInil | LIcons of 'a * 'a list_immutable ;;
# type 'a list_mutable = LMnil | LMcons of 'a * 'a list_mutable ref ;;
```

The immutable lists are strictly equivalent to lists built into Objective Caml, while the mutable lists are closer to the style of C, in which a cell is a value together with a reference to the following cell.

With immutable lists, there is only one way to write concatenation, and it requires duplicating the structure of the first list; by contrast, the second list may be shared with the result.

```
# let rec concat l1 l2 = match l1 with
      LInil → l2
    | LIcons (a,l11) → LIcons(a, (concat l11 l2)) ;;
val concat : 'a list_immutable -> 'a list_immutable -> 'a list_immutable =
  <fun>

# let li1 = LIcons(1, LIcons(2, LInil))
  and li2 =  LIcons(3, LIcons(4, LInil)) ;;
val li1 : int list_immutable = LIcons (1, LIcons (2, LInil))
val li2 : int list_immutable = LIcons (3, LIcons (4, LInil))
# let li3 = concat li1 li2 ;;
val li3 : int list_immutable =
  LIcons (1, LIcons (2, LIcons (3, LIcons (4, LInil))))
# li1==li3 ;;
- : bool = false
# let tlLI l =  match l with
    LInil       → failwith "Liste vide"
  | LIcons(_,x) → x ;;
val tlLI : 'a list_immutable -> 'a list_immutable = <fun>
# tlLI(tlLI(li3)) == li2 ;;
- : bool = true
```

From these examples, we see that the first cells of `li1` and `li3` are distinct, while the second half of `li3` is exactly `li2`.

With mutable lists, we have a choice between modifying arguments (function `concat_share`) and creating a new value (function `concat_copy`).

```
# let rec concat_copy l1 l2 =  match l1 with
      LMnil  → l2
    | LMcons (x,l11) → LMcons(x, ref (concat_copy !l11 l2)) ;;
```

```
val concat_copy : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

This first solution, `concat_copy`, gives a result similar to the previous function, `concat`. A second solution shares its arguments with its result fully:

```
# let concat_share l1 l2 =
    match l1 with
      LMnil → l2
    | _     → let rec set_last = function
                  LMnil       → failwith "concat_share : impossible case!!"
                | LMcons(_,l) → if !l=LMnil then l:=l2 else set_last !l
              in
                set_last l1 ;
                l1 ;;
val concat_share : 'a list_mutable -> 'a list_mutable -> 'a list_mutable =
  <fun>
```

Concatenation with sharing does not require any allocation, and therefore does not use the constructor `LMcons`. Instead, it suffices to cause the last cell of the first list to point to the second list. However, this version of concatenation has the potential weakness that it alters arguments passed to it.

```
# let lm1 = LMcons(1, ref (LMcons(2, ref LMnil)))
   and lm2 = LMcons(3, ref (LMcons(4, ref LMnil))) ;;
val lm1 : int list_mutable =
  LMcons (1, {contents=LMcons (2, {contents=LMnil})})
val lm2 : int list_mutable =
  LMcons (3, {contents=LMcons (4, {contents=LMnil})})
# let lm3 = concat_share lm1 lm2 ;;
val lm3 : int list_mutable =
  LMcons (1, {contents=LMcons (2, {contents=LMcons (...)})})
```

We do indeed obtain the expected result for `lm3`. However, the value bound to `lm1` has been modified.

```
# lm1 ;;
- : int list_mutable =
LMcons (1, {contents=LMcons (2, {contents=LMcons (...)})})
```

This may therefore have consequences on the rest of the program.

How to Choose your Style

In a purely functional program, side effects are forbidden, and this excludes mutable data structures, exceptions, and input/output. We prefer, though, a less restrictive definition of the functional style, saying that functions that do not modify their global environment may be used in a functional style. Such a function may manipulate mutable values locally, and may therefore be written in an imperative style, but must not modify global variables, nor its arguments. We permit them to raise exceptions in addition. Viewed from outside, these functions may be considered "black boxes." Their behavior matches a function written in a purely functional style, apart from being able of breaking control flow by raising an exception. In the same spirit, a mutable value which can no longer be modified after initialization may be used in a functional style.

On the other hand, a program written in an imperative style still benefits from the advantages provided by Objective Caml: static type safety, automatic memory management, the exception mechanism, parametric polymorphism, and type inference.

The choice between the imperative and functional styles depends on the application to be developed. We may nevertheless suggest some guidelines based on the character of the application, and the criteria considered important in the development process.

- **choice of data structures**: The choice whether to use mutable data structures follows from the style of programming adopted. Indeed, the functional style is essentially incompatible with modifying mutable values. By contrast, constructing and traversing objects are the same whatever their status. This touches the same issue as "modification in place *vs* copying" on page 99; we return to it again in discussing criteria of efficiency.
- **required data structures**: If a program must modify mutable data structures, then the imperative style is the only one possible. If, on the other hand, you just have to traverse values, then adopting the functional style guarantees the integrity of the data.
  Using recursive data structures requires the use of functions that are themselves recursive. Recursive functions may be defined using either of the two styles, but it is often easier to understand the creation of a value following a recursive definition, which corresponds to a functional approach, than to repeat the recursive processing on this element. The functional style allows us to define generic iterators over the structure of data, which factors out the work of development and makes it faster.
- **criteria of efficiency**: Modification in place is far more efficient than creating a value. When code efficiency is the preponderant criterion, it will usually tip the balance in favor of the imperative style. We note however that the need to avoid sharing values may turn out to be a very hard task, and in the end costlier than copying the values to begin with.
  Being purely functional has a cost. Partial application and using functions passed as arguments from other functions has an execution cost greater than total application of a function whose declaration is visible. Using this eminently functional feature must thus be avoided in those portions of a program where efficiency is crucial.
- **development criteria**: the higher level of abstraction of functional programs permits them to be written more quickly, leading to code that is more compact and contains fewer errors than the equivalent imperative code, which is generally more verbose. The functional style is better suited to the constraints imposed by developing substantial applications. Since each function is not dependent upon its evaluation context, functional can be easily divided into small units that can be examined separately; as a consequence, the code is easier to read.
  Programs written using the functional style are more easily reusable because of its better modularity, and because functions may be passed as arguments to other functions.

These remarks show that it is often a good idea to mix the two programming styles within the same application. The functional programming style is faster to develop and confers a simpler organization to an application. However, portions whose execution time is critical repay being developed in a more efficient imperative style.

Mixing Styles

As we have mentioned, a language offering both functional and imperative characteristics allows the programmer to choose the more appropriate style for each part of the implementation of an algorithm. One can indeed use both aspects in the same function. This is what we will now illustrate.

Closures and Side Effects

The convention, when a function causes a side effect, is to treat it as a procedure and to return the value `()`, of type *unit*. Nevertheless, in some cases, it can be useful to cause the side effect within a function that returns a useful value. We have already used this mixture of the styles in the function `permute_pivot` of quicksort.

The next example is a symbol generator that creates a new symbol each time that it is called. It simply uses a counter that is incremented at every call.

```
# let c = ref 0;;
val c : int ref = {contents=0}
# let reset_symb = function () → c:=0 ;;
val reset_symb : unit -> unit = <fun>
# let new_symb = function s →  c:=!c+1 ; s^(string_of_int !c) ;;
val new_symb : string -> string = <fun>
# new_symb "VAR" ;;
- : string = "VAR1"
# new_symb "VAR" ;;
- : string = "VAR2"
# reset_symb () ;;
- : unit = ()
# new_symb "WAR" ;;
- : string = "WAR1"
# new_symb "WAR" ;;
- : string = "WAR2"
```

The reference `c` may be hidden from the rest of the program by writing:

```
# let (reset_s , new_s) =
    let c = ref 0
    in let f1 () = c := 0
       and f2 s  = c := !c+1 ; s^(string_of_int !c)
    in (f1,f2) ;;
```

```
val reset_s : unit -> unit = <fun>
val new_s : string -> string = <fun>
```

This declaration creates a pair of functions that share the variable `c`, which is local to this declaration. Using these two functions produces the same behavior as the previous definitions.

```
# new_s "VAR";;
- : string = "VAR1"
# new_s "VAR";;
- : string = "VAR2"
# reset_s();;
- : unit = ()
# new_s "WAR";;
- : string = "WAR1"
# new_s "WAR";;
- : string = "WAR2"
```

This example permits us to illustrate the way that closures are represented. A closure may be considered as a pair containing the code (that is, the **`function`** part) as one component and the local envoronment containing the values of the free variables of the function. Figure 4.1 shows the memory representation of the closures `reset_s` and `new_s`.

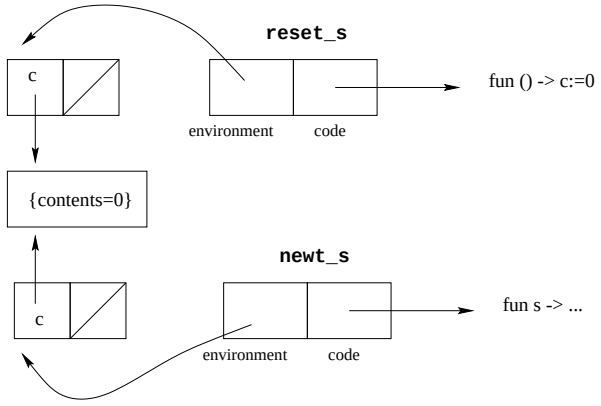


Figure 4.1: Memory representation of closures.

These two closures share the same environment, containing the value of `c`. When either one modifies the reference `c`, it modifies the contents of an area of memory that is shared with the other closure.

Physical Modifications and Exceptions

Exceptions make it possible to escape from situations in which the computation cannot proceed. In this case, an exception handler allows the calculation to continue, knowing that one branch has failed. The problem with side effects comes from the state of the modifiable data when the exception was raised. One cannot be sure of this state if there have been physical modifications in the branch of the calculation that has failed.

Let us define the increment function `(++)` analogous to the operator in C:

```
# let (++) x = x:=!x+1; x;;
val ++ : int ref -> int ref = <fun>
```

The following example shows a little computation where division by zero occurs together with

```
# let x = ref 2;;
val x : int ref = {contents=2}
(* 1 *)
# !((++) x) * (1/0) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=2}
(* 2 *)
# (1/0) * !((++) x) ;;
Uncaught exception: Division_by_zero
# x;;
- : int ref = {contents=3}
```

The variable `x` is not modified during the computation of the expression in (∗1∗), while it is modified in the computation of (∗2∗). Unless one saves the initial values, the form **`try .. with ..`** must not have a **`with ..`** part that depends on modifiable variables implicated in the expression that raised the exception.

Modifiable Functional Data Structures

In functional programming a program (in particular, a function expression) may also serve as a data object that may be manipulated, and one way to see this is to write association lists in the form of function expressions. In fact, one may view association lists of type *`('a * 'b) list`* as partial functions taking a key chosen from the set *`'a`* and returning a value in the set of associated values *`'b`*. Each association list is then a function of type *`'a -> 'b`*.

The empty list is the everywhere undefined function, which one simulates by raising an exception:

```
# let nil_assoc = function x → raise Not_found ;;
val nil_assoc : 'a -> 'b = <fun>
```

We next write the function `add_assoc` which adds an element to a list, meaning that it extends the function for a new entry:

```
# let add_assoc (k,v) l = function x → if x = k then v else l x ;;
val add_assoc : 'a * 'b -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = add_assoc ('1', 1) (add_assoc ('2', 2) nil_assoc) ;;
val l : char -> int = <fun>
# l '2' ;;
- : int = 2
# l 'x' ;;
Uncaught exception: Not_found
```

We may now re-write the function mem_assoc:

```
# let mem_assoc k l = try  (l k) ; true  with  Not_found → false ;;
val mem_assoc : 'a -> ('a -> 'b) -> bool = <fun>
# mem_assoc '2' l ;;
- : bool = true
# mem_assoc 'x' l ;;
- : bool = false
```

By contrast, writing a function to remove an element from a list is not trivial, because one no longer has access to the values captured by the closures. To accomplish the same purpose we mask the former value by raising the exception Not_found.

```
# let rem_assoc k l = function x → if x=k then raise Not_found else l x ;;
val rem_assoc : 'a -> ('a -> 'b) -> 'a -> 'b = <fun>
# let l = rem_assoc '2' l ;;
val l : char -> int = <fun>
# l '2' ;;
Uncaught exception: Not_found
```

Clearly, one may also create references and work by side effect on such values. However, one must take some care.

```
# let add_assoc_again (k,v) l =  l := (function x → if x=k then v else !l x) ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
```

The resulting value for l is a function that points at itself and therefore loops. This annoying side effect is due to the fact that the dereferencing !l is within the scope of the closure **function** *x* →. The value of !l is not evaluated during compilation, but at run-time. At that time, l points to the value that has already been modified by add_assoc. We must therefore correct our definition using the closure created by our original definition of add_assoc:

```
# let add_assoc_again (k, v) l =  l := add_assoc (k, v) !l ;;
val add_assoc_again : 'a * 'b -> ('a -> 'b) ref -> unit = <fun>
# let l = ref nil_assoc ;;
val l : ('_a -> '_b) ref = {contents=<fun>}
# add_assoc_again ('1',1) l ;;
- : unit = ()
```

```
# add_assoc_again ('2',2) l ;;
- : unit = ()
# !l '1' ;;
- : int = 1
# !l 'x' ;;
Uncaught exception: Not_found
```

Lazy Modifiable Data Structures

Combining imperative characteristics with a functional language produces good tools for implementing computer languages. In this subsection, we will illustrate this idea by implementing data structures with deferred evaluation. A data structure of this kind is not completely evaluated. Its evaluation progresses according to the use made of it.

Deferred evaluation, which is often used in purely functional languages, is simulated using function values, possibly modifiable. There are at least two purposes for manipulating incompletely evaluated data structures: first, so as to calculate only what is effectively needed in the computation; and second, to be able to work with potentially infinite data structures.

We define the type `vm`, whose members contain either an already calculated value (constructor `Imm`) or else a value to be calculated (constructor `Deferred`):

```
# type 'a v =
      Imm of 'a
    | Deferred of (unit → 'a);;
# type 'a vm = {mutable c : 'a v };;
```

A computation is deferred by encapsulating it in a closure. The evaluation function for deferred values must return the value if it has already been calculated, and otherwise, if the value is not already calculated, it must evaluate it and then store the result.

```
# let eval e = match e.c with
    Imm a → a
  | Deferred f → let u = f () in e.c <- Imm u ; u ;;
val eval : 'a vm -> 'a = <fun>
```

The operations of deferring evaluation and activating it are also called *freezing* and *thawing* a value.

We could also write the conditional control structure in the form of a function:

```
# let if_deferred c e1 e2 =
    if eval c then eval e1 else eval e2;;
val if_deferred : bool vm -> 'a vm -> 'a vm -> 'a = <fun>
```

Here is how to use it in a recursive function such as factorial:

```
# let rec facr n =
    if_deferred
          {c=Deferred(fun () → n = 0)}
          {c=Deferred(fun () →  1)}
          {c=Deferred(fun () → n*(facr(n-1)))};;
val facr : int -> int = <fun>
# facr 5;;
- : int = 120
```

The classic form of **if** can not be written in the form of a function. In fact, if we define a function `if_function` this way:

```
# let if_function c e1 e2 = if c then e1 else e2;;
val if_function : bool -> 'a -> 'a -> 'a = <fun>
```

then the three arguments of `if_function` are evaluated at the time they are passed to the function. So the function `fact` loops, because the recursive call *fact*(*n*-1) is always evaluated, even when `n` has the value `0`.

```
# let rec fact n = if_function (n=0) 1 (n*fact(n-1)) ;;
val fact : int -> int = <fun>
# fact 5 ;;
Stack overflow during evaluation (looping recursion?).
```

Module Lazy

The implementation difficulty for frozen values is due to the conflict between the eager evaluation strategy of Objective Caml and the need to leave expressions unevaluated. Our attempt to redefine the conditional illustrated this. More generally, it is impossible to write a function that freezes a value in producing an object of type *vm*:

```
# let freeze e = { c = Deferred (fun () → e) };;
val freeze : 'a -> 'a vm = <fun>
```

When this function is applied to arguments, the Objective Caml evaluation strategy evaluates the expression `e` passed as argument before constructing the closure **fun** () → *e*. The next example shows this:

```
# freeze (print_string "trace"; print_newline(); 4*5);;
trace
- : int vm = {c=Deferred <fun>}
```

This is why the following syntactic form was introduced.

Syntax : **lazy** *expr*

Warning This form is a language extension that may evolve in future versions.

When the keyword **lazy** is applied to an expression, it constructs a value of a type declared in the module Lazy:

```
# let x = lazy (print_string "Hello"; 3*4) ;;
val x : int Lazy.status ref = {contents=Lazy.Delayed <fun>}
```

The expression (*print_string* "Hello") has not been evaluated, because no message has been printed. The function force of module Lazy allows one to force evaluation:

```
# Lazy.force x ;;
Hello- : int = 12
```

Now the value x has altered:

```
# x ;;
- : int Lazy.t = {contents=Lazy.Value 12}
```

It has become the value of the expression that had been frozen, namely 12.

For another call to the function force, it's enough to return the value already calculated:

```
# Lazy.force x ;;
- : int = 12
```

The string "Hello" is no longer prefixed.

"Infinite" Data Structures

The second reason to defer evaluation is to be able to construct potentially infinite data structures such as the set of natural numbers. Because it might take a long time to construct them all, the idea here is to compute only the first one and to know how to pass to the next element.

We define a generic data structure *'a enum* which will allow us to enumerate the elements of a set.

```
# type 'a enum = { mutable i : 'a; f :'a → 'a } ;;
type 'a enum = { mutable i: 'a; f: 'a -> 'a }
# let next e = let x = e.i in e.i <- (e.f e.i) ; x ;;
val next : 'a enum -> 'a = <fun>
```

Now we can get the set of natural numbers by instantiating the fields of this structure:

```
# let nat = { i=0; f=fun x → x + 1 };;
val nat : int enum = {i=0; f=<fun>}
# next nat;;
- : int = 0
# next nat;;
- : int = 1
# next nat;;
- : int = 2
```

Another example gives the elements of the Fibonnacci sequence, which has the defini-

tion:

$$\begin{cases} u_0 = 1 \\ u_1 = 1 \\ u_{n+2} = u_n + u_{n+1} \end{cases}$$

The function to compute the successor must take account of the current value, (u_{n-1}), but also of the preceding one (u_{n-2}). For this, we use the state `c` in the following closure:

```
# let fib = let fx = let c = ref 0 in fun v → let r = !c + v in c:=v ; r
            in { i=1 ; f=fx } ;;
val fib : int enum = {i=1; f=<fun>}
# for i=0 to 10 do  print_int (next fib); print_string "  " done ;;
1  1  2  3  5  8  13  21  34  55  89  - : unit = ()
```

Streams of Data

Streams are (potentially infinite) sequences containing elements of the same kind. The evaluation of a part of a stream is done on demand, whenever it is needed by the current computation. A stream is therefore a *lazy* data structure.

The *stream* type is an abstract data type; one does not need to know how it is implemented. We manipulate objects of this type using constructor functions and destructor (or selector) functions. For the convenience of the user, Objective Caml has simple syntactic constructs to construct streams and to access their elements.

Warning Streams are an extension of the language, not part of the stable core of Objective Caml.

Construction

The syntactic sugar to construct streams is inspired by that for lists and arrays. The empty stream is written:

```
# [< >] ;;
- : 'a Stream.t = <abstr>
```

One may construct a stream by enumerating its elements, preceding each one with an with a single quote (character `'`):

```
# [< '0; '2; '4 >] ;;
- : int Stream.t = <abstr>
```

Expressions not preceded by an apostrophe are considered to be sub-streams:

```
# [< '0; [< '1; '2; '3 >]; '4 >] ;;
- : int Stream.t = <abstr>
# let s1 = [< '1; '2; '3 >] in [< s1; '4 >] ;;
```

```
- : int Stream.t = <abstr>
# let concat_stream a b = [< a ; b >] ;;
val concat_stream : 'a Stream.t -> 'a Stream.t -> 'a Stream.t = <fun>
# concat_stream [< '"if"; '"c";'"then";'"1" >] [< '"else";'"2" >] ;;
- : string Stream.t = <abstr>
```

The `Stream` module also provides other construction functions. For instance, the functions `of_channel` and `of_string` return a stream containing a sequence of characters, received from an input stream or a string.

```
# Stream.of_channel ;;
- : in_channel -> char Stream.t = <fun>
# Stream.of_string ;;
- : string -> char Stream.t = <fun>
```

The deferred computation of streams makes it possible to manipulate infinite data structures in a way similar to the type *'a enum* defined on page 109. We define the stream of natural numbers by its first element and a function calculating the stream of elements to follow:

```
# let rec nat_stream n = [< 'n ; nat_stream (n+1) >] ;;
val nat_stream : int -> int Stream.t = <fun>
# let nat = nat_stream 0 ;;
val nat : int Stream.t = <abstr>
```

Destruction and Matching of Streams

The primitive `next` permits us to evaluate, retrieve, and remove the first element of a stream, all at once:

```
# for i=0 to 10 do
    print_int (Stream.next nat) ;
    print_string "  "
    done ;;
0  1  2  3  4  5  6  7  8  9  10  - : unit = ()
# Stream.next nat ;;
- : int = 11
```

When the stream is exhausted, an exception is raised.

```
# Stream.next [< >] ;;
Uncaught exception: Stream.Failure
```

To manipulate streams, Objective Caml offers a special-purpose matching construct called *destructive matching*. The value matched is calculated and removed from the stream. There is no notion of exhaustive match for streams, and, since the data type is lazy and potentially infinite, one may match less than the whole stream. The syntax for matching is:

Syntax : **match** *expr* **with parser [< '**p_1 ... **>] ->** $expr_1$ **|** ...

The function `next` could be written:

```
# let next s = match s with parser [< 'x >] → x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat;;
- : int = 12
```

Note that the enumeration of natural numbers picks up where we left it previously.

As with function abstraction, there is a syntactic form matching a function parameter of type *Stream.t*.

Syntax : **parser** *p -¿ ...*

The function `next` can thus be rewritten:

```
# let next = parser [<'x>] → x ;;
val next : 'a Stream.t -> 'a = <fun>
# next nat ;;
- : int = 13
```

It is possible to match the empty stream, but take care: the stream pattern `[<>]` matches every stream. In fact, a stream `s` is always equal to the stream `[< [<>]; s >]`. For this reason, one must reverse the usual order of matching:

```
# let rec it_stream f s =
    match s with parser
      [< 'x ; ss >] → f x ; it_stream f ss
    | [<>] → () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# let print_int1 n = print_int n ; print_string" " ;;
val print_int1 : int -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Since matching is destructive, one can equivalently write:

```
# let rec it_stream f s =
    match s with parser
      [< 'x >] → f x ; it_stream f s
    | [<>] → () ;;
val it_stream : ('a -> 'b) -> 'a Stream.t -> unit = <fun>
# it_stream print_int1 [<'1; '2; '3>] ;;
1 2 3 - : unit = ()
```

Although streams are lazy, they want to be helpful, and never refuse to furnish a first element; when it has been supplied once it is lost. This has consequences for matching. The following function is an attempt (destined to fail) to display pairs from a stream of integers, except possibly for the last element.

```
# let print_int2 n1
  n2 =
```

```
    print_string "(" ; print_int n1 ; print_string "," ;
    print_int n2 ; print_string ")" ;;
val print_int2 : int -> int -> unit = <fun>
# let rec print_stream s =
    match s with parser
      [< 'x; 'y >] → print_int2 x y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)Uncaught exception: Stream.Error("")
```

The first two two members of the stream were displayed properly, but during the evaluation of the recursive call (*print_stream* `[<3>]`), the first pattern found a value for `x`, which was thereby consumed. There remained nothing more for `y`. This was what caused the error. In fact, the second pattern is useless, because if the stream is not empty, then first pattern always begins evaluation.

To obtain the desired result, we must sequentialize the matching:

```
# let rec print_stream s =
    match s with parser
      [< 'x >]
       → (match s with parser
             [< 'y >] → print_int2 x y; print_stream s
           | [<>] → print_int1 x; print_stream s)
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>];;
(1,2)3
- : unit = ()
```

If matching fails on the first element of a pattern however, then we again have the familiar behavior of matching:

```
# let rec print_stream s =
    match s with parser
      [< '1; 'y >] → print_int2 1 y; print_stream s
    | [< 'z >] → print_int1 z; print_stream s
    | [<>] → print_newline() ;;
val print_stream : int Stream.t -> unit = <fun>
# print_stream [<'1; '2; '3>] ;;
(1,2)3
- : unit = ()
```

The Limits of Matching

Because it is destructive, matching streams differs from matching on sum types. We will now illustrate how radically different it can be.

We can quite naturally write a function to compute the sum of the elements of a stream:

```
# let rec sum s =
    match s with parser
      [< 'n; ss >] → n+(sum ss)
    | [<>] → 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

However, we can just as easily consume the stream from the inside, naming the partial result:

```
# let rec sum s =
    match s with parser
      [< 'n; r = sum >] → n+r
    | [<>] → 0 ;;
val sum : int Stream.t -> int = <fun>
# sum [<'1; '2; '3; '4>] ;;
- : int = 10
```

We will examine some other important uses of streams in chapter 11, which is devoted to lexical and syntactic analysis. In particular, we will see how consuming a stream from the inside may be profitably used.

Exercises

Binary Trees

We represent binary trees in the form of vectors. If a tree a has height h, then the length of the vector will be $2^{(h+1)}-1$. If a node has position i, then the left subtree of this node lies in the interval of indices $[i+1\ ,\ i+1+2^h]$, and its right subtree lies in the interval $[i+1+2^h+1\ ,\ 2^{(h+1)}-1]$. This representation is useful when the tree is almost completely filled. The type `'a` of labels for nodes in the tree is assumed to contain a special value indicating that the node does not exist. Thus, we represent labeled trees by the by vectors of type `'a array`.

1. Write a function , taking as input a binary tree of type `'a bin_tree` (defined on page 50) and an array (which one assumes to be large enough). The function stores the labels contained in the tree in the array, located according to the discipline described above.
2. Write a function to create a leaf (tree of height 0).

3. Write a function to construct a new tree from a label and two other trees.
4. Write a conversion function from the type *'a bin_tree* to an array.
5. Define an infix traversal function for these trees.
6. Use it to display the tree.
7. What can you say about prefix traversal of these trees?

Spelling Corrector

The exercise uses the lexical tree , from the exercise of chapter 2, page 63, to build a spelling corrector.

1. Construct a dictionary from a file in ASCII in which each line contains one word. For this, one will write a function which takes a file name as argument and returns the corresponding dictionary.
2. Write a function `words` that takes a character string and constructs the list of words in this string. The word separators are space, tab, apostrophe, and quotation marks.
3. Write a function `verify` that takes a dictionary and a list of words, and returns the list of words that do not occur in the dictionary.
4. Write a function `occurrences` that takes a list of words and returns a list of pairs associating each word with the number of its occurrences.
5. Write a function `spellcheck` that takes a dictionary and the name of a file containing the text to analyze. It should return the list of incorrect words, together with their number of occurrences.

Set of Prime Numbers

We would like now to construct the infinite set of prime numbers (without calculating it completely) using lazy data structures.

1. Define the predicate `divisible` which takes an integer and an initial list of prime numbers, and determines whether the number is divisible by one of the integers on the list.
2. Given an initial list of prime numbers, write the function `next` that returns the smallest number not on the list.
3. Define the value *setprime* representing the set of prime numbers, in the style of the type *'a enum* on page 109. It will be useful for this set to retain the integers already found to be prime.

Summary

This chapter has compared the functional and imperative programming styles. They differ mainly in the control of execution (implicit in functional and explicit in impera-

tive programming), and in the representation in memory of data (sharing or explicitly copied in the imperative case, irrelevant in the functional case). The implementation of algorithms must take account of these differences. The choice between the two styles leads in fact to mixing them. This mixture allows us to clarify the representation of closures, to optimize crucial parts of applications, and to create mutable functional data. Physical modification of values in the environment of a closure permits us to better understand what a functional value is. The mixture of the two styles gives powerful implementation tools. We used them to construct potentially infinite values.

To Learn More

The principal consequences of adding imperative traits to a functional language are:

- To determine the evaluation strategy (strict evaluation);
- to add implementation constraints, especially for the GC (see Chapter 9);
- For statically typed languages, to make their type system more complex;
- To offer different styles of programming in the same language, permitting us to program in the style appropriate to the algorithm at hand, or possibly in a mixed style.

This last point is important in Objective Caml where we need the same parametric polymorphism for functions written in either style. For this, certain purely functional programs are no longer typable after the addition. Wright's article ([Wri95]) explains the difficulties of polymorphism in languages with imperative aspects. Objective Caml adopts the solution that he advocates. The classification of different kinds of polymorphism in the presence of physical modification is described well in the thesis of Emmanuel Engel ([Eng98]).

These consequences make the job of programming a bit harder, and learning the language a bit more difficult. But because the language is richer for this reason and above all offers the choice of style, the game is worth the candle. For example, strict evaluation is the rule, but it is possible to implement basic mechanisms for lazy evaluation, thanks to the mixture of the two styles. Most purely functional languages use a lazy evaluation style. Among languages close to ML, we would mention Miranda, LazyML, and Haskell. The first two are used at universities for teaching and research. By contrast, there are significant applications written in Haskell. The absence of controllable side effects necessitates an additional abstraction for input/output called *monads*. One can read works on Haskell (such as [Tho99]) to learn more about this subject. Streams are a good example of the mixture of functional and imperative styles. Their use in lexical and syntactic analysis is described in Chapter 11.

5

The Graphics Interface

This chapter presents the `Graphics` library, which is included in the distribution of the Objective Caml-language. This library is designed in such a way that it works identically under the main graphical interfaces of the most commonly used operating systems: Windows, MacOS, Unix with X-Windows. `Graphics` permits the realization of drawings which may contain text and images, and it handles basic events like mouse clicks or pressed keys.

The model of programming graphics applied is the "painter's model:" the last touch of color erases the preceding one. This is an imperative model where the graphics window is a table of points which is physically modified by each graphics primitive. The interactions with the mouse and the keyboard are a model of event-driven programming: the primary function of the program is an infinite loop waiting for user interaction. An event starts execution of a special handler, which then returns to the main loop to wait for the next event.

Although the `Graphics` library is very simple, it is sufficient for introducing basic concepts of graphical interfaces, and it also contains basic elements for developing graphical interfaces that are rich and easy to use by the programmer.

Chapter overview

The first section explains how to make use of this library on different systems. The second section introduces the basic notions of graphics programming: reference point, plotting, filling, colors, bitmaps. The third section illustrates these concepts by describing and implementing functions for creating and drawing "boxes." The fourth section demonstrates the animation of graphical objects and their interaction with the background of the screen or other animated objects. The fifth section presents event-driven programming, in other terms the skeleton of all graphical interfaces. Finally, the last

section uses the library `Graphics` to construct a graphical interface for a calculator (see page 86).

Using the `Graphics` *Module*

Utilization of the library `Graphics` differs depending on the system and the compilation mode used. We will not cover applications other than usable under the interactive toplevel of Objective Caml. Under the Windows and MacOS systems the interactive working environment already preloads this library. To make it available under Unix, it is necessary to create a new toplevel. This depends on the location of the X11 library. If this library is placed in one of the usual search paths for C language libraries, the command line is the following:

```
ocamlmktop -custom -o mytoplevel graphics.cma -cclib -lX11
```

It generates a new executable`mytoplevel` into which the library `Graphics` is integrated. Starting the executable works as follows:

```
./mytoplevel
```

If, however, as under Linux, the library X11 is placed in another directory, this has to be indicated to the command `ocamlmktop`:

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

In this example, the file `libX11.a` is searched in the directory `/usr/X11/lib`.

A complete description of the command `ocamlmktop` can be found in chapter 7.

Basic notions

Graphics programming is tightly bound to the technological evolution of hardware, in particular to that of screens and graphics cards. In order to render images in sufficient quality, it is necessary that the drawing be refreshed (redrawn) at regular and short intervals, somewhat like in a cinema. There are basically two techniques for drawing on the screen: the first makes use of a list of visible segments where only the useful part of the drawing is drawn, the second displays all points of the screen (bitmap screen). It is the last technique which is used on ordinary computers.

Bitmap screens can be seen as rectangles of accessible, in other terms, displayable points. These points are called *pixels*, a word derived from *picture element*. They are the basic elements for constructing images. The height and width of the main bitmap

is the resolution of the screen. The size of this bitmap therefore depends on the size of each pixel. In monochrome (black/white) displays, a pixel can be encoded in one bit. For screens that allow gray scales or for color displays, the size of a pixel depends on the number of different colors and shades that a pixel may take. In a bitmap of 320x640 pixels with 256 colors per pixel, it is therefore necessary to encode a pixel in 8 bits, which requires video memory of: $480 * 640$ bytes $= 307200$ bytes $\simeq$ 300KB. This resolution is still used by certain MS-DOS programs.

The basic operations on bitmaps which one can find in the `Graphics` library are:

- coloration of pixels,
- drawing of pixels,
- drawing of forms: rectangles, ellipses,
- filling of closed forms: rectangles, ellipses, polygons,
- displaying text: as bitmap or as vector,
- manipulation or displacement of parts of the image.

All these operations take place at a *reference point*, the one of the bitmap. A certain number of characteristics of these graphical operations like the width of strokes, the joints of lines, the choice of the character font, the style and the motive of filling define what we call a *graphical context*. A graphical operation always happens in a particular graphical context, and its result depends on it. The graphical context of the `Graphics` library does not contain anything except for the current point, the current color, the current font and the size of the image.

Graphical display

The elements of the graphical display are: the reference point and the graphical context, the colors, the drawings, the filling pattern of closed forms, the texts and the bitmaps.

Reference point and graphical context

The `Graphics` library manages a unique main window. The coordinates of the reference point of the window range from point $(0, 0)$ at the bottom left to the upper right corner of the window. The main functions on this window are:

- `open_graph`, of type *`string -> unit`*, which opens a window;
- `close_graph`, of type *`unit -> unit`*, which closes it;
- `clear_graph`, of type *`unit -> unit`*, which clears it.

The dimensions of the graphical window are given by the functions `size_x` and `size_y`.

The string argument of the function `open_graph` depends on the window system of the machine on which the program is executed and is therefore not platform independent. The empty string, however, opens a window with default settings. It is possible to

specify the size of the window: under X-Windows, `" 200x300"` yields a window which is 200 pixels wide and 300 pixels high. Beware, the space at the beginning of the string `" 200x300"` is required!

The graphical context contains a certain number of readable and/or modifiable parameters:

the current point:	`current_point` : *unit -> int * int*
	`moveto` : *int -> int -> unit*
the current color:	`set_color` : *color -> unit*
the width of lines:	`set_line_width` : *int -> unit*
the current character font:	`set_font` : *string -> unit*
the size of characters:	`set_text_size` : *int -> unit*

Colors

Colors are represented by three bytes: each stands for the intensity value of a main color in the RGB-model (red, green, blue), ranging from a minimum of 0 to a maximum of 255. The function `rgb` (of type *int -> int -> int -> color*) allows the generation of a new color from these three components. If the three components are identical, the resulting color is a gray which is more or less intense depending on the intensity value. Black corresponds to the minimum intensity of each component (`0 0 0`) and white is the maximum (`255 255 255`). Certain colors are predefined: `black`, `white`, `red`, `green`, `blue`, `yellow`, `cyan` and `magenta`.

The variables `foreground` and `background` correspond to the color of the fore- and the background respectively. Clearing the screen is equivalent to filling the screen with the background color.

A color (a value of type *color*) is in fact an integer which can be manipulated to, for example, decompose the color into its three components (`from_rgb`) or to apply a function to it that inverts it (`inv_color`).

```
(* color == R * 256 * 256  +  G * 256  +  B  *)
# let from_rgb (c : Graphics.color)  =
    let r = c / 65536  and  g = c / 256 mod 256  and  b = c mod 256
    in (r,g,b);;
val from_rgb : Graphics.color -> int * int * int = <fun>
# let inv_color (c : Graphics.color) =
     let (r,g,b) = from_rgb c
     in Graphics.rgb (255-r) (255-g) (255-b);;
val inv_color : Graphics.color -> Graphics.color = <fun>
```

The function `point_color`, of type *int -> int -> color*, returns the color of a point when given its coordinates.

Drawing and filling

A drawing function draws a line on the screen. The line is of the current width and color. A filling function fills a closed form with the current color. The various line- and filling functions are presented in figure 5.1.

drawing	filling	type
`plot`		*int -> int -> unit*
`lineto`		*int -> int -> unit*
	`fill_rect`	*int -> int -> int -> int -> unit*
	`fill_poly`	*(int * int) array -> unit*
`draw_arc`	`fill_arc`	*int -> int -> int -> int -> int -> unit*
`draw_ellipse`	`fill_ellipse`	*int -> int -> int -> int -> unit*
`draw_circle`	`fill_circle`	*int -> int -> int -> unit*

Figure 5.1: Drawing- and filling functions.

Beware, the function `lineto` changes the position of the current point to make drawing of vertices more convenient.

Drawing polygons To give an example, we add drawing primitives which are not predefined. A polygon is described by a table of its vertices.

```
# let draw_rect x0 y0 w h =
    let (a,b) = Graphics.current_point()
    and x1 = x0+w and y1 = y0+h
    in
      Graphics.moveto x0 y0;
      Graphics.lineto x0 y1; Graphics.lineto x1 y1;
      Graphics.lineto x1 y0; Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_rect : int -> int -> int -> int -> unit = <fun>

# let draw_poly r =
    let (a,b) = Graphics.current_point () in
    let (x0,y0) = r.(0) in Graphics.moveto x0 y0;
      for i = 1 to (Array.length r)-1 do
        let (x,y) = r.(i) in Graphics.lineto x y
      done;
      Graphics.lineto x0 y0;
      Graphics.moveto a b;;
val draw_poly : (int * int) array -> unit = <fun>
```

Please note that these functions take the same arguments as the predefined ones for filling forms. Like the other functions for drawing forms, they do not change the current point.

Illustrations in the painter's model This example generates an illustration of a token ring network (figure 5.2). Each machine is represented by a small circle. We place the set of machines on a big circle and draw a line between the connected machines. The current position of the token in the network is indicated by a small black disk.

The function `net_points` generates the coordinates of the machines in the network. The resulting data is stored in a table.

```
# let pi = 3.1415927;;
val pi : float = 3.1415927
# let net_points (x,y) l n =
    let a = 2. *. pi /. (float n) in
    let rec aux (xa,ya) i =
      if i > n then []
      else
        let na = (float i) *. a in
        let x1 = xa + (int_of_float ( cos(na) *. l))
        and y1 = ya + (int_of_float ( sin(na) *. l))  in
        let np = (x1,y1) in
          np :: (aux np (i+1))
    in Array.of_list (aux (x,y) 1);;
val net_points : int * int -> float -> int -> (int * int) array = <fun>
```

The function `draw_net` displays the connections, the machines and the token.

```
# let draw_net (x,y) l n sc st =
   let r = net_points (x,y) l n in
     draw_poly r;
     let draw_machine (x,y) =
       Graphics.set_color Graphics.background;
       Graphics.fill_circle x y sc;
       Graphics.set_color Graphics.foreground;
       Graphics.draw_circle x y sc
     in
       Array.iter draw_machine r;
       Graphics.fill_circle x y st;;
val draw_net : int * int -> float -> int -> int -> int -> unit = <fun>
```

The following function call corresponds to the left drawing in figure 5.2.

```
# draw_net (140,20) 60.0 10 10 3;;
- : unit = ()

# save_screen "IMAGES/tokenring.caa";;
```

```
- : unit = ()
```

We note that the order of drawing objects is important. We first plot the connections

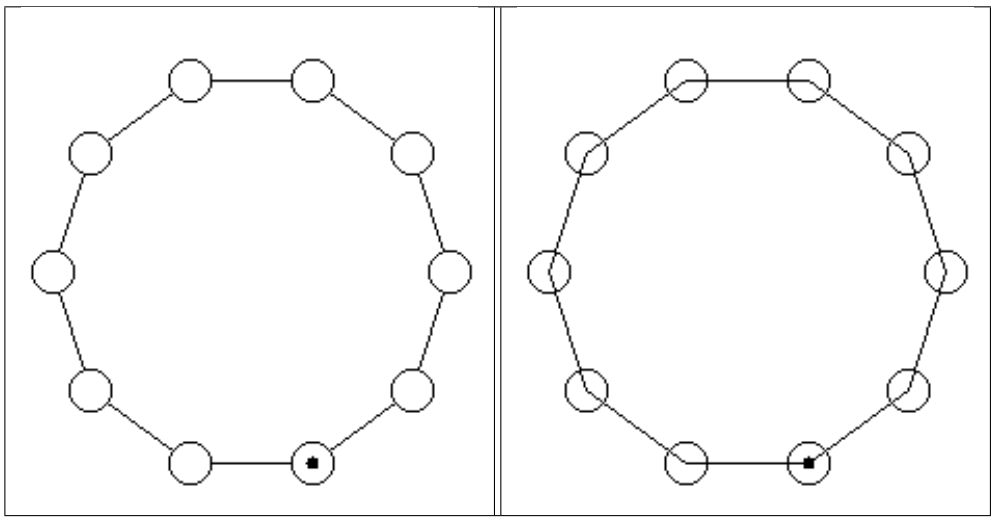


Figure 5.2: Tokenring network.

then the nodes. The drawing of network nodes erases some part of the connecting lines. Therefore, there is no need to calculate the point of intersection between the connection segments and the circles of the vertices. The right illustration of figure 5.2 inverts the order in which the objects are displayed. We see that the segments appear inside of the circles representing the nodes.

Text

The functions for displaying texts are rather simple. The two functions `draw_char` (of type *`char -> unit`*) and `draw_string` (of type *`string -> unit`*) display a character and a character string respectively at the current point. After displaying, the latter is modified. These functions do not change the current font and its current size.

Note

The displaying of strings may differ depending on the graphical interface.

The function `text_size` takes a string as input and returns a pair of integers that correspond to the dimensions of this string when it is displayed in the current font and size.

Displaying strings vertically This example describes the function `draw_string_v`, which displays a character string vertically at the current point. It is used in figure 5.3. Each letter is displayed separately by changing the vertical coordinate.

```
# let draw_string_v  s =
```

```
    let (xi,yi) = Graphics.current_point()
    and l = String.length s
    and (_,h) = Graphics.text_size s
    in
       Graphics.draw_char s.[0];
       for i=1 to l-1 do
         let (_,b) = Graphics.current_point()
         in Graphics.moveto xi (b-h);
            Graphics.draw_char s.[i]
        done;
      let (a,_) = Graphics.current_point() in Graphics.moveto a yi;;
val draw_string_v : string -> unit = <fun>
```

This function modifies the current point. After displaying, the point is placed at the initial position offset by the width of one character.

The following program permits displaying a legend around the axes (figure 5.3)

```
#
  Graphics.moveto 0 150; Graphics.lineto 300 150;
  Graphics.moveto 2 130; Graphics.draw_string "abscissa";
  Graphics.moveto 150 0; Graphics.lineto 150 300;
  Graphics.moveto 135 280; draw_string_v "ordinate";;
- : unit = ()
```

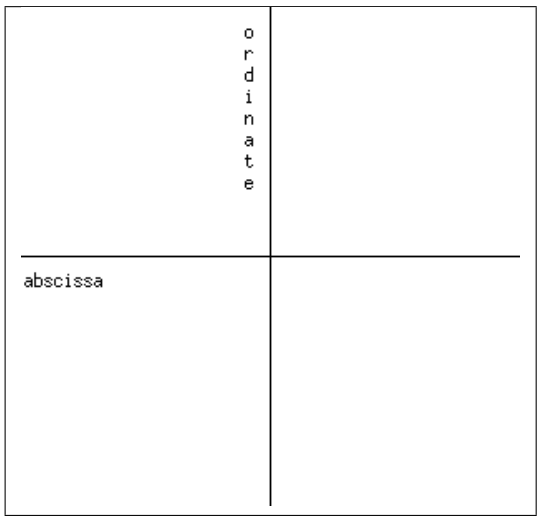


Figure 5.3: Legend around axes.

If we wish to realize vertical displaying of text, it is necessary to account for the fact that the current point is modified by the function `draw_string_v`. To do this, we define the function `draw_text_v`, which accepts the spacing between columns and a list of words as parameters.

```
# let draw_text_v n l =
    let f s = let (a,b) = Graphics.current_point()
              in draw_string_v s;
                 Graphics.moveto (a+n) b
    in List.iter f l;;
val draw_text_v : int -> string list -> unit = <fun>
```

If we need further text transformations like, for example, rotation, we will have to take the *bitmap* of each letter and perform the rotation on this set of pixels.

Bitmaps

A bitmap may be represented by either a color matrix (*color array array*) or a value of abstract type [1] *image*, which is declared in library `Graphics`. The names and types of the functions for manipulating bitmaps are given in figure 5.4.

function	type
make_image	*color array array -> image*
dump_image	*image -> color array array*
draw_image	*image -> int -> int -> unit*
get_image	*int -> int -> int -> int -> image*
blit_image	*image -> int -> int -> unit*
create_image	*int -> int -> image*

Figure 5.4: Functions for manipulating bitmaps.

The functions `make_image` and `dump_image` are conversion functions between types *image* and *color array array*. The function `draw_image` displays a bitmap starting at the coordinates of its bottom left corner.

The other way round, one can capture a rectangular part of the screen to create an image using the function `get_image` and by indicating the bottom left corner and the upper right one of the area to be captured. The function `blit_image` modifies its first parameter (of type *image*) and captures the region of the screen where the lower left corner is given by the point passed as parameter. The size of the captured region is the one of the image argument. The function `create_image` allows initializing images by specifying their size to use them with `blit_image`.

The predefined color `transp` can be used to create transparent points in an image. This makes it possible to display an image within a rectangular area only; the transparent points do not modify the initial screen.

1. Abstract types hide the internal representation of their values. The declaration of such types will be presented in chapter 14.

Polarization of Jussieu This example inverts the color of points of a bitmap. To do this, we use the function for color inversion presented on page 120, applying it to each pixel of a bitmap.

```
# let inv_image i =
    let inv_vec = Array.map (fun c → inv_color c) in
    let inv_mat = Array.map inv_vec in
    let inverted_matrix = inv_mat (Graphics.dump_image i) in
    Graphics.make_image inverted_matrix;;
val inv_image : Graphics.image -> Graphics.image = <fun>
```

Given the bitmap `jussieu`, which is displayed in the left half of figure 5.5, we use the function `inv_image` and obtain a new "solarized" bitmap, which is displayed in the right half of the same figure.

```
# let f_jussieu2 () = inv_image jussieu1;;
val f_jussieu2 : unit -> Graphics.image = <fun>
```

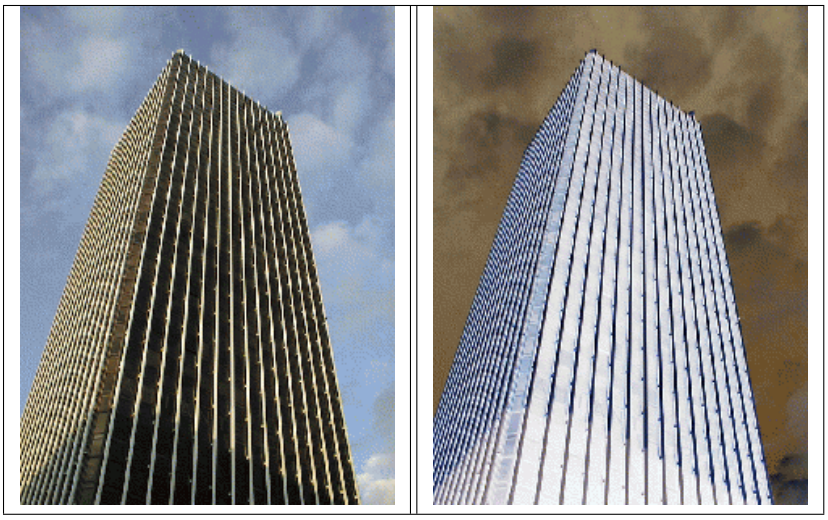
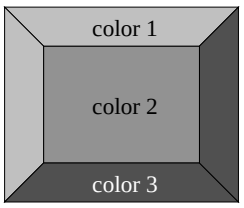


Figure 5.5: Inversion of Jussieu.

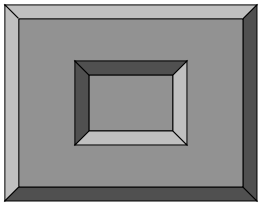
Example: drawing of boxes with relief patterns

In this example we will define a few utility functions for drawing boxes that carry relief patterns. A box is a generic object that is useful in many cases. It is inscribed in a rectangle which is characterized by a point of origin, a height and a width.

To give an impression of a box with a relief pattern, it is sufficient to surround it with two trapezoids in a light color and two others in a somewhat darker shade.



Inverting the colors, one can give the impression that the boxes are on top or at the bottom.



Implementation We add the border width, the display mode (top, bottom, flat) and the colors of its edges and of its bottom. This information is collected in a record.

```
# type relief = Top | Bot | Flat;;
# type box_config =
    { x:int; y:int; w:int; h:int; bw:int; mutable r:relief;
      b1_col : Graphics.color;
      b2_col : Graphics.color;
      b_col : Graphics.color};;
```

Only field `r` can be modified. We use the function `draw_rect` defined at page 121, which draws a rectangle.

For convenience, we define a function for drawing the outline of a box.

```
# let draw_box_outline bcf col =
    Graphics.set_color col;
    draw_rect bcf.x bcf.y bcf.w  bcf.h;;
val draw_box_outline : box_config -> Graphics.color -> unit = <fun>
```

The function of displaying a box consists of three parts: drawing the first edge, drawing the second edge and drawing the interior of the box.

```
# let draw_box bcf =
     let x1 = bcf.x and y1 = bcf.y in
     let x2 = x1+bcf.w and y2 = y1+bcf.h in
     let ix1 = x1+bcf.bw and ix2 = x2-bcf.bw
     and iy1 = y1+bcf.bw and iy2 = y2-bcf.bw in
     let border1 g =
       Graphics.set_color g;
       Graphics.fill_poly
         [| (x1,y1);(ix1,iy1);(ix2,iy1);(ix2,iy2);(x2,y2);(x2,y1) |]
```

```
    in
    let border2 g =
      Graphics.set_color g;
      Graphics.fill_poly
        [| (x1,y1);(ix1,iy1);(ix1,iy2);(ix2,iy2);(x2,y2);(x1,y2) |]
    in
    Graphics.set_color bcf.b_col;
    ( match bcf.r with
          Top  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b1_col;
            border2 bcf.b2_col
        | Bot  →
            Graphics.fill_rect ix1 iy1 (ix2-ix1) (iy2-iy1);
            border1 bcf.b2_col;
            border2 bcf.b1_col
        | Flat →
            Graphics.fill_rect x1 y1 bcf.w bcf.h );
    draw_box_outline bcf Graphics.black;;
val draw_box : box_config -> unit = <fun>
```

The outline of boxes is highlighted in black. Erasing a box fills the area it covers with the background color.

```
# let erase_box bcf =
     Graphics.set_color bcf.b_col;
     Graphics.fill_rect (bcf.x+bcf.bw) (bcf.y+bcf.bw)
                        (bcf.w-(2*bcf.bw)) (bcf.h-(2*bcf.bw));;
val erase_box : box_config -> unit = <fun>
```

Finally, we define a function for displaying a character string at the left, right or in the middle of the box. We use the type *position* to describe the placement of the string.

```
# type position = Left | Center | Right;;
type position = | Left | Center | Right
# let draw_string_in_box pos str bcf col =
      let (w, h) = Graphics.text_size str in
      let ty = bcf.y + (bcf.h-h)/2 in
      ( match pos with
            Center → Graphics.moveto (bcf.x + (bcf.w-w)/2) ty
          | Right  → let tx = bcf.x + bcf.w - w - bcf.bw - 1 in
                     Graphics.moveto tx ty
          | Left   → let tx = bcf.x + bcf.bw + 1 in Graphics.moveto tx ty  );
      Graphics.set_color col;
      Graphics.draw_string str;;
val draw_string_in_box :
  position -> string -> box_config -> Graphics.color -> unit = <fun>
```

Example: drawing of a game We illustrate the use of boxes by displaying the position of a game of type "tic-tac-toe" as shown in figure 5.6. To simplify the creation of boxes, we predefine colors.

```
# let set_gray x =  (Graphics.rgb x x x);;
val set_gray : int -> Graphics.color = <fun>
# let gray1= set_gray 100 and gray2= set_gray 170 and gray3= set_gray 240;;
val gray1 : Graphics.color = 6579300
val gray2 : Graphics.color = 11184810
val gray3 : Graphics.color = 15790320
```

We define a function for creating a grid of boxes of same size.

```
# let rec create_grid nb_col n sep b  =
    if n < 0 then []
    else
      let px = n mod nb_col and py = n / nb_col in
      let nx = b.x +sep + px*(b.w+sep)
      and ny = b.y +sep + py*(b.h+sep) in
      let b1 = {b with x=nx; y=ny} in
      b1 :: (create_grid nb_col (n-1) sep b);;
val create_grid : int -> int -> int -> box_config -> box_config list = <fun>
```

And we create the vector of boxes:

```
# let vb =
    let b =  {x=0; y=0; w=20;h=20; bw=2;
              b1_col=gray1; b2_col=gray3; b_col=gray2; r=Top} in
    Array.of_list (create_grid 5 24 2  b);;
val vb : box_config array =
  [|{x=90; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=15790320;
     b_col=11184810};
    {x=68; y=90; w=20; h=20; bw=2; r=Top; b1_col=6579300; b2_col=15790320;
     b_col=...};
    ...|]
```

Figure 5.6 corresponds to the following function calls:

```
# Array.iter draw_box vb;
  draw_string_in_box Center "X" vb.(5) Graphics.black;
  draw_string_in_box Center "X" vb.(8) Graphics.black;
  draw_string_in_box Center "O" vb.(12) Graphics.yellow;
  draw_string_in_box Center "O" vb.(11) Graphics.yellow;;
- : unit = ()
```

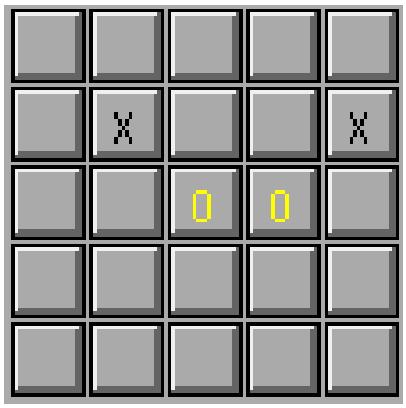


Figure 5.6: Displaying of boxes with text.

Animation

The animation of graphics on a screen reuses techniques of animated drawings. The major part of a drawing does not change, only the animated part must modify the color of its constituent pixels. One of the immediate problems we meet is the speed of animation. It can vary depending on the computational complexity and on the execution speed of the processor. Therefore, to be portable, an application containing animated graphics must take into account the speed of the processor. To get smooth rendering, it is advisable to display the animated object at the new position, followed by the erasure of the old one and taking special care with the intersection of the old and new regions.

Moving an object We simplify the problem of moving an object by choosing objects of a simple shape, namely rectangles. The remaining difficulty is knowing how to redisplay the background of the screen once the object has been moved.

We try to make a rectangle move around in a closed space. The object moves at a certain speed in directions X and Y. When it encounters a border of the graphical window, it bounces back depending on the angle of impact. We assume a situation without overlapping of the new and old positions of the object. The function `calc_pv` computes the new position and the new velocity from an old position `(x,y)`, the size of the object `(sx,sy)` and from the old speed `(dx,dy)`, taking into account the borders of the window.

```
# let calc_pv (x,y) (sx,sy) (dx,dy) =
    let nx1 = x+dx     and  ny1 = y + dy
    and nx2 = x+sx+dx  and  ny2 = y+sy+dy
    and ndx = ref dx   and  ndy = ref dy
    in
      ( if (nx1 < 0) || (nx2 >= Graphics.size_x()) then ndx := -dx );
```

```
      ( if (ny1 < 0) || (ny2 >= Graphics.size_y()) then ndy := -dy );
      ((x+ !ndx, y+ !ndy), (!ndx, !ndy));;
val calc_pv :
  int * int -> int * int -> int * int -> (int * int) * (int * int) = <fun>
```

The function `move_rect` moves the rectangle given by `pos` and `size n` times, the trajectory being indicated by its `speed` and by taking into account the borders of the space. The trace of movement which one can see in figure 5.7 is obtained by inversion of the corresponding bitmap of the displaced rectangle.

```
# let move_rect pos size speed n =
    let (x, y) = pos and (sx,sy) = size in
    let mem = ref (Graphics.get_image x y sx sy) in
    let rec move_aux x y speed n =
     if n = 0 then Graphics.moveto x y
     else
      let ((nx,ny),n_speed) = calc_pv (x,y) (sx,sy) speed
      and old_mem = !mem in
       mem := Graphics.get_image nx ny sx sy;
       Graphics.set_color Graphics.blue;
       Graphics.fill_rect nx ny sx sy;
       Graphics.draw_image (inv_image old_mem) x y;
       move_aux nx ny n_speed (n-1)
    in move_aux x y speed n;;
val move_rect : int * int -> int * int -> int * int -> int -> unit = <fun>
```

The following code corresponds to the drawings in figure 5.7. The first is obtained on a uniformly red background, the second by moving the rectangle across the image of Jussieu.

```
# let anim_rect () =
    Graphics.moveto 105 120;
    Graphics.set_color Graphics.white;
    Graphics.draw_string "Start";
    move_rect (140,120) (8,8) (8,4) 150;
    let (x,y) = Graphics.current_point() in
     Graphics.moveto (x+13) y;
    Graphics.set_color Graphics.white;
    Graphics.draw_string "End";;
val anim_rect : unit -> unit = <fun>
# anim_rect();;
- : unit = ()
```

The problem was simplified, because there was no intersection between two successive positions of the moved object. If this is not the case, it is necessary to write a function that computes this intersection, which can be more or less complicated depending on

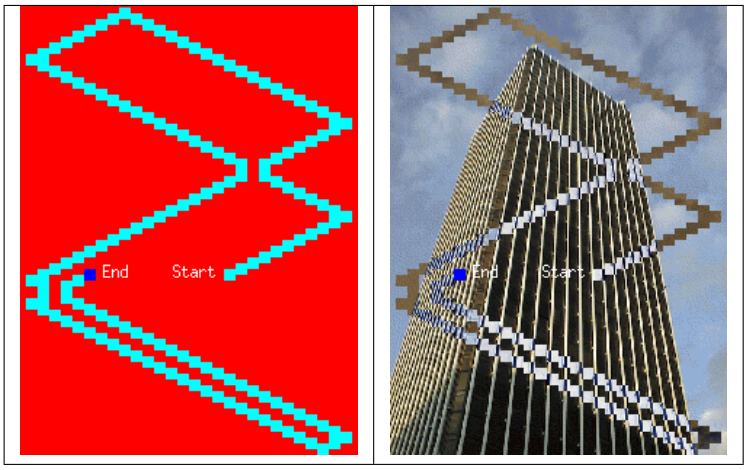


Figure 5.7: Moving an object.

the form of the object. In the case of a square, the intersection of two squares yields a rectangle. This intersection has to be removed.

Events

The handling of events produced in the graphical window allows interaction between the user and the program. `Graphics` supports the treating of events like keystrokes, mouse clicks and movements of the mouse.

The programming style therefore changes the organization of the program. It becomes an infinite loop waiting for events. After handling each newly triggered event, the program returns to the infinite loop except for events that indicate program termination.

Types and functions for events

The main function for waiting for events is `wait_next_event` of type *`event list -> status`*.

The different events are given by the sum type *`event`*.

```
type event = Button_down | Button_up | Key_pressed | Mouse_motion | Poll;;
```

The four main values correspond to pressing and to releasing a mouse button, to movement of the mouse and to keystrokes. Waiting for an event is a blocking operation except if the constructor `Poll` is passed in the event list. This function returns a value of type *`status`*:

```
type status =
  { mouse_x : int;
    mouse_y : int;
```

```
    button : bool;
    keypressed : bool;
    key : char};;
```

This is a record containing the position of the mouse, a Boolean which indicates whether a mouse button is being pressed, another Boolean for the keyboard and a character which corresponds to the pressed key. The following functions exploit the data contained in the event record:

- `mouse_pos`: *`unit -> int * int`*: returns the position of the mouse with respect to the window. If the mouse is placed elsewhere, the coordinates are outside the borders of the window.
- `button_down`: *`unit -> bool`*: indicates pressing of a mouse button.
- `read_key`: *`unit -> char`*: fetches a character typed on the keyboard; this operation blocks.
- `key_pressed`: *`unit -> bool`*: indicates whether a key is being pressed on the keyboard; this operation does not block.

The handling of events supported by `Graphics` is indeed minimal for developing interactive interfaces. Nevertheless, the code is portable across various graphical systems like Windows, MacOS or X-Windows. This is the reason why this library does not take into account different mouse buttons. In fact, the Mac does not even possess more than one. Other events, such as exposing a window or changing its size are not accessible and are left to the control of the library.

Program skeleton

All programs implementing a graphical user interface make use of a potentially infinite loop waiting for user interaction. As soon as an action arrives, the program executes the job associated with this action. The following function possesses five parameters of functionals. The first two serve for starting and closing the application. The next two arguments handle keyboard and mouse events. The last one permits handling of exceptions that escape out of the different functions of the application. We assume that the events associated with terminating the application raise the exception `End`.

```
# exception End;;
exception End
# let skel f_init f_end f_key f_mouse f_except =
    f_init ();
    try
        while true do
          try
            let s = Graphics.wait_next_event
                      [Graphics.Button_down; Graphics.Key_pressed]
            in if s.Graphics.keypressed then f_key s.Graphics.key
               else if s.Graphics.button
                    then f_mouse s.Graphics.mouse_x s.Graphics.mouse_y
```

```
          with
               End → raise End
             | e   → f_except e
        done
    with
        End  → f_end ();;
val skel :
  (unit -> 'a) ->
  (unit -> unit) ->
  (char -> unit) -> (int -> int -> unit) -> (exn -> unit) -> unit = <fun>
```

Here, we use the skeleton to implement a mini-editor. Touching a key displays the typed character. A mouse click changes the current point. The character '&' exits the program. The only difficulty in this program is line breaking. We assume as simplification that the height of characters does not exceed twelve pixels.

```
# let next_line () =
    let (x,y) = Graphics.current_point()
    in if y>12 then Graphics.moveto 0 (y-12)
       else Graphics.moveto 0 y;;
val next_line : unit -> unit = <fun>
# let handle_char c = match c with
      '&'  → raise End
    | '\n' → next_line ()
    | '\r' → next_line ()
    |  _   → Graphics.draw_char c;;
val handle_char : char -> unit = <fun>
# let go () = skel
    (fun () → Graphics.clear_graph ();
              Graphics.moveto 0 (Graphics.size_y() -12) )
    (fun () → Graphics.clear_graph())
    handle_char
    (fun x y → Graphics.moveto x y)
    (fun e → ());;
val go : unit -> unit = <fun>
```

This program does not handle deletion of characters by pressing the key `DEL`.

Example: telecran

Telecran is a little drawing game for training coordination of movements. A point appears on a slate. This point can be moved in directions X and Y by using two control buttons for these axes without ever releasing the pencil. We try to simulate this behavior to illustrate the interaction between a program and a user. To do this we reuse the previously described skeleton. We will use certain keys of the keyboard to indicate movement along the axes.

We first define the type *state*, which is a record describing the size of the slate in terms of the number of positions in X and Y, the current position of the point and the scaling factor for visualization, the color of the trace, the background color and the color of the current point.

```
# type state = {maxx:int; maxy:int; mutable x : int; mutable y :int;
                scale:int;
                bc : Graphics.color;
                fc: Graphics.color; pc : Graphics.color};;
```

The function `draw_point` displays a point given its coordinates, the scaling factor and its color.

```
# let draw_point x y s c =
    Graphics.set_color c;
    Graphics.fill_rect (s*x) (s*y) s s;;
val draw_point : int -> int -> int -> Graphics.color -> unit = <fun>
```

All these functions for initialization, handling of user interaction and exiting the program receive a parameter corresponding to the state. The first four functions are defined as follows:

```
# let t_init s () =
    Graphics.open_graph (" " ^ (string_of_int (s.scale*s.maxx)) ^
      "x" ^ (string_of_int (s.scale*s.maxy)));
    Graphics.set_color s.bc;
    Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
    draw_point s.x s.y s.scale s.pc;;
val t_init : state -> unit -> unit = <fun>
# let t_end s () =
    Graphics.close_graph();
    print_string "Good bye..."; print_newline();;
val t_end : 'a -> unit -> unit = <fun>
# let t_mouse s x y = ();;
val t_mouse : 'a -> 'b -> 'c -> unit = <fun>
# let t_except s ex = ();;
val t_except : 'a -> 'b -> unit = <fun>
```

The function `t_init` opens the graphical window and displays the current point, `t_end` closes this window and displays a message, `t_mouse` and `t_except` do not do anything. The program handles neither mouse events nor exceptions which may accidentally arise during program execution. The important function is the one for handling the keyboard `t_key`:

```
# let t_key s c =
   draw_point s.x s.y s.scale s.fc;
    (match c with
    '8' → if s.y < s.maxy then  s.y <- s.y + 1;
  | '2' → if s.y > 0 then s.y <- s.y - 1
```

```
  | '4' → if s.x > 0 then s.x <- s.x - 1
  | '6' → if s.x < s.maxx then s.x <- s.x + 1
  | 'c' →   Graphics.set_color s.bc;
            Graphics.fill_rect 0 0 (s.scale*s.maxx+1) (s.scale*s.maxy+1);
            Graphics.clear_graph()
  | 'e' → raise End
  | _ → ());
   draw_point s.x s.y s.scale s.pc;;
val t_key : state -> char -> unit = <fun>
```

It displays the current point in the color of the trace. Depending on the character passed, it modifies, if possible, the coordinates of the current point (characters: '2', '4', '6', '8'), clears the screen (character: 'c') or raises the exception `End` (character: 'e'), then it displays the new current point. Other characters are ignored. The choice of characters for moving the cursor comes from the layout of the numeric keyboard: the chosen keys correspond to the indicated digits and to the direction arrows. It is therefore useful to activate the numeric keyboard for the ergonomics of the program.

We finally define a state and apply the skeleton function in the following way:

```
# let stel = {maxx=120; maxy=120; x=60; y=60;
              scale=4; bc=Graphics.rgb 130 130 130;
              fc=Graphics.black; pc=Graphics.red};;
val stel : state =
  {maxx=120; maxy=120; x=60; y=60; scale=4; bc=8553090; fc=0; pc=16711680}
# let slate () =
    skel (t_init stel)  (t_end stel) (t_key stel)
     (t_mouse stel) (t_except stel);;
val slate : unit -> unit = <fun>
```

Calling function `slate` displays the graphical window, then it waits for user interaction on the keyboard. Figure 5.8 shows a drawing created with this program.

A Graphical Calculator

Let's consider the calculator example as described in the preceding chapter on imperative programming (see page 86). We will give it a graphical interface to make it more usable as a desktop calculator.

The graphical interface materializes the set of keys (digits and functions) and an area for displaying results. Keys can be activated using the graphical interface (and the mouse) or by typing on the keyboard. Figure 5.9 shows the interface we are about to construct.

We reuse the functions for drawing boxes as described on page 126. We define the following type:

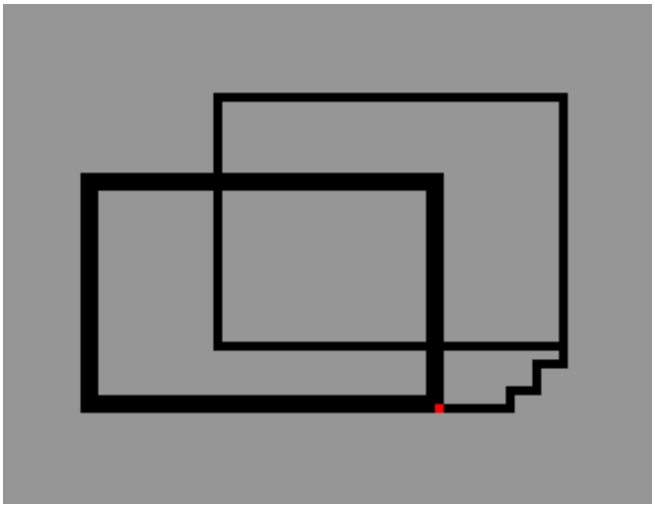


Figure 5.8: Telecran.

Figure 5.9: Graphical calculator.

```
# type calc_state =
    { s : state; k : (box_config * key * string ) list; v : box_config } ;;
```

It contains the state of the calculator, the list of boxes corresponding to the keys and the visualization box. We plan to construct a calculator that is easily modifiable. Therefore, we parameterize the construction of the interface with an association list:

```
# let descr_calc =
    [  (Digit 0,"0");  (Digit 1,"1");  (Digit 2,"2");  (Equals, "=");
       (Digit 3,"3");  (Digit 4,"4");  (Digit 5,"5");  (Plus, "+");
```

```
    (Digit 6,"6");  (Digit 7,"7");  (Digit 8,"8");  (Minus, "-");
    (Digit 9,"9");  (Recall,"RCL"); (Div, "/");     (Times, "*");
    (Off,"AC");     (Store, "STO"); (Clear,"CE/C")
  ] ;;
```

Generation of key boxes At the beginning of this description we construct a list of key boxes. The function `gen_boxes` takes as parameters the description (`descr`), the number of the column (`n`), the separation between boxes (`wsep`), the separation between the text and the borders of the box (`wsepint`) and the size of the board (`wbord`). This function returns the list of key boxes as well as the visualization box. To calculate these placements, we define the auxiliary functions `max_xy` for calculating the maximal size of a list of complete pairs and `max_lbox` for calculating the maximal positions of a list of boxes.

```
# let gen_xy vals comp o  =
    List.fold_left (fun  a (x,y) → comp (fst a) x,comp (snd a) y) o vals ;;
val gen_xy : ('a * 'a) list -> ('b -> 'a -> 'b) -> 'b * 'b -> 'b * 'b = <fun>
# let max_xy vals = gen_xy vals max (min_int,min_int);;
val max_xy : (int * int) list -> int * int = <fun>
# let max_boxl l =
    let bmax (mx,my) b = max mx b.x, max my b.y
    in List.fold_left bmax  (min_int,min_int) l ;;
val max_boxl : box_config list -> int * int = <fun>
```

Here is the principal function `gen_boxes` for creating the interface.

```
# let gen_boxes descr n wsep wsepint wbord  =
    let l_l = List.length descr in
    let nb_lig = if l_l mod n = 0 then l_l / n else l_l / n + 1 in
    let ls = List.map (fun (x,y) → Graphics.text_size y) descr in
    let sx,sy = max_xy  ls in
    let sx,sy= sx+wsepint ,sy+wsepint  in
      let r = ref [] in
        for i=0 to l_l-1 do
          let px = i mod n and py = i / n in
            let b = { x = wsep * (px+1) + (sx+2*wbord) * px  ;
                      y = wsep * (py+1) + (sy+2*wbord) * py ;
                      w = sx; h = sy ; bw = wbord;
                      r=Top;
                      b1_col = gray1;  b2_col = gray3; b_col =gray2}
            in r:= b::!r
        done;
        let mpx,mpy = max_boxl !r in
        let upx,upy = mpx+sx+wbord+wsep,mpy+sy+wbord+wsep in
        let (wa,ha) = Graphics.text_size "        0" in
          let v = { x=(upx-(wa+wsepint +wbord))/2 ; y= upy+ wsep;
                    w=wa+wsepint; h = ha +wsepint; bw = wbord *2; r=Flat ;
```

```
                    b1_col = gray1;  b2_col = gray3; b_col =Graphics.black}
          in
            upx,(upy+wsep+ha+wsepint+wsep+2*wbord),v,
            List.map2 (fun b (x,y) → b,x,y ) (List.rev !r) descr;;
val gen_boxes :
  ('a * string) list ->
  int ->
  int ->
  int -> int -> int * int * box_config * (box_config * 'a * string) list =
  <fun>
```

Interaction Since we would also like to reuse the skeleton proposed on page 133 for interaction, we define the functions for keyboard and mouse control, which are integrated in this skeleton. The function for controlling the keyboard is very simple. It passes the translation of a character value of type ***key*** to the function `transition` of the calculator and then displays the text associated with the calculator state.

```
# let f_key cs c =
    transition cs.s (translation c);
    erase_box cs.v;
    draw_string_in_box Right (string_of_int cs.s.vpr) cs.v Graphics.white ;;
val f_key : calc_state -> char -> unit = <fun>
```

The control of the mouse is a bit more complex. It requires verification that the position of the mouse click is actually in one of the key boxes. For this we first define the auxiliary function `mem`, which verifies membership of a position within a rectangle.

```
# let mem (x,y) (x0,y0,w,h) =
      (x >= x0) && (x< x0+w) && (y>=y0) && ( y<y0+h);;
val mem : int * int -> int * int * int * int -> bool = <fun>
# let f_mouse cs x y =
    try
      let b,t,s =
        List.find (fun (b,_,_) →
                     mem (x,y) (b.x+b.bw,b.y+b.bw,b.w,b.h)) cs.k
      in
        transition cs.s t;
        erase_box cs.v;
        draw_string_in_box Right (string_of_int cs.s.vpr ) cs.v Graphics.white
    with Not_found → ();;
val f_mouse : calc_state -> int -> int -> unit = <fun>
```

The function `f_mouse` looks whether the position of the mouse during the click is really-dwell within one of the boxes corresponding to a key. If it is, it passes the corresponding key to the transition function and displays the result, otherwise it will not do anything.

The function `f_exc` handles the exceptions which can arise during program execution.

```
# let f_exc cs ex =
    match ex with
     Division_by_zero →
       transition cs.s Clear;
       erase_box cs.v;
       draw_string_in_box Right "Div 0" cs.v (Graphics.red)
  | Invalid_key → ()
  | Key_off → raise End
  | _  → raise ex;;
val f_exc : calc_state -> exn -> unit = <fun>
```

In the case of a division by zero, it restarts in the initial state of the calculator and displays an error message on its screen. Invalid keys are simply ignored. Finally, the exception `Key_off` raises the exception `End` to terminate the loop of the skeleton.

Initialization and termination The initialization of the calculator requires calculation of the window size. The following function creates the graphical information of the boxes from a key/text association and returns the size of the principal window.

```
# let create_e k =
       Graphics.close_graph ();
       Graphics.open_graph " 10x10";
       let mx,my,v,lb = gen_boxes k 4 4 5 2 in
       let s = {lcd=0; lka = false; loa = Equals; vpr = 0; mem = 0}   in
         mx,my,{s=s; k=lb;v=v};;
val create_e : (key * string) list -> int * int * calc_state = <fun>
```

The initialization function makes use of the result of the preceding function.

```
# let f_init mx my cs () =
         Graphics.close_graph();
         Graphics.open_graph (" "^(string_of_int mx)^"x"^(string_of_int my));
         Graphics.set_color gray2;
         Graphics.fill_rect 0 0 (mx+1) (my+1);
         List.iter (fun (b,_,_) → draw_box b) cs.k;
         List.iter
           (fun (b,_,s) → draw_string_in_box Center s b Graphics.black) cs.k ;
         draw_box cs.v;
         erase_box cs.v;
         draw_string_in_box Right "hello" cs.v (Graphics.white);;
val f_init : int -> int -> calc_state -> unit -> unit = <fun>
```

Finally the termination function closes the graphical window.

```
# let f_end e () = Graphics.close_graph();;
val f_end : 'a -> unit -> unit = <fun>
```

The function `go` is parameterized by a description and starts the interactive loop.

```
# let go descr =
    let mx,my,e = create_e descr in
      skel (f_init mx my e) (f_end e) (f_key e) (f_mouse e) (f_exc e);;
val go : (key * string) list -> unit = <fun>
```

The call to `go descr_calc` corresponds to the figure 5.9.

Exercises

Polar coordinates

Coordinates as used in the library `Graphics` are Cartesian. There a line segment is represented by its starting point `(x0,y0)` and its end point `(x1,y1)`. It can be useful to use polar coordinates instead. Here a line segment is described by its point of origin `(x0,y0)`, a length (radius) `(r)` and an angle `(a)`. The relation between Cartesian and Polar coordinates is defined by the following equations:

$$\begin{cases} x_1 &= x_0 + r * \cos(a) \\ y_1 &= y_0 + r * \sin(a) \end{cases}$$

The following type defines the polar coordinates of a line segment:

```
# type seg_pol = {x:float; y:float; r:float; a:float};;
type seg_pol = { x: float; y: float; r: float; a: float }
```

1. Write the function `to_cart` that converts polar coordinates to Cartesian ones.
2. Write the function `draw_seg` which displays a line segment defined by polar coordinates in the reference point of `Graphics`.
3. One of the motivations behind polar coordinates is to be able to easily apply transformations to line segments. A translation only modifies the point of origin, a rotation only affects the angle field and modifying the scale only changes the length field. Generally, one can represent a transformation as a triple of floats: the first represents the translation (we do not consider the case of translating the second point of the line segment here), the second the rotation and the third the scaling factor. Define the function `app_trans` which takes a line segment in polar coordinates and a triple of transformations and returns the new segment.
4. One can construct recursive drawings by iterating transformations. Write the function `draw_r` which takes as arguments a line segment `s`, a number of iterations `n`, a list of transformations and displays all the segments resulting from the transformations on `s` iterated up to `n`.
5. Verify that the following program does produce the images in figure 5.10.

   ```
   let pi = 3.1415927 ;;
   ```

```
let s = {x=100.; y= 0.; a= pi /. 2.; r = 100.} ;;
draw_r s 6 [ (-.pi/.2.),0.6,1.; (pi/.2.), 0.6,1.0] ;;
Graphics.clear_graph();;
draw_r s 6 [(-.pi /. 6.), 0.6, 0.766;
            (-.pi /. 4.), 0.55, 0.333;
            (pi /. 3.), 0.4, 0.5 ] ;;
```

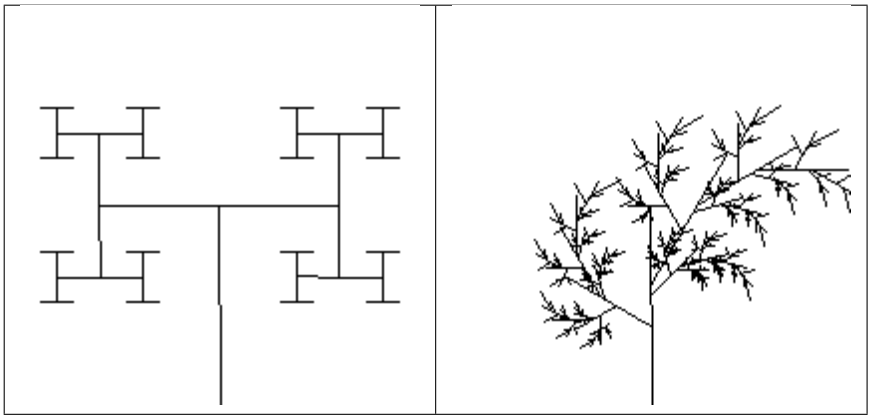


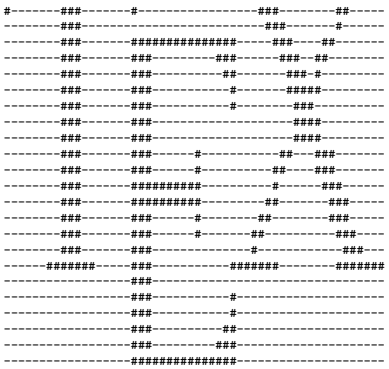
Figure 5.10: Recursive drawings.

Bitmap editor

We will attempt to write a small bitmap editor (similar to the command `bitmap` in X-window). For this we represent a bitmap by its dimensions (width and height), the pixel size and a two-dimensional table of booleans.

1. Define a type *bitmap_state* describing the information necessary for containing the values of the pixels, the size of the bitmap and the colors of displayed and erased points.
2. Write a function for creating bitmaps (`create_bitmap`) and for displaying bitmaps (`draw_bitmap`) .
3. Write the functions `read_bitmap` and `write_bitmap` which respectively read and write in a file passed as parameter following the ASCII format of X-window. If the file does not exist, the function for reading creates a new bitmap using the function `create_bitmap`. A displayed pixel is represented by the character `#`, the absence of a pixel by the character `-`. Each line of characters represents a line of the bitmap. One can test the program using the functions `atobm` and `bmtoa` of X-window, which convert between this ASCII format and the format of bitmaps created by the command `bitmap`. Here is an example.

```
##################------------#######--------######
##################--------------###------------##--
###-----###-----###--------------###------------#---
##------###------##--------------###-----------##---
#-------###------#---------------###---------##----
```



4. We reuse the skeleton for interactive loops on page 133 to construct the graphical interface of the editor. The human-computer interface is very simple. The bitmap is permanently displayed in the graphical window. A mouse click in one of the slots of the bitmap inverts its color. This change is reflected on the screen. Pressing the key 'S' saves the bitmap in a file. The key 'Q' terminates the program.
   - Write a function `start` of type *`bitmap_state -> unit -> unit`* which opens a graphical window and displays the bitmap passed as parameter.
   - Write a function `stop` that closes the graphical window and exits the program.
   - Write a function `mouse` of type *`bitmap_state -> int -> int -> unit`* which modifies the pixel state corresponding to the mouse click and displays the change.
   - Write a function `key` of type *`string -> bitmap_state -> char -> unit`* which takes as arguments the name of a file, a bitmap and the char of the pressed key and executes the associated actions: saving to a file for the key 'S' and raising of the exception `End` for the key 'Q'.
5. Write a function `go` which takes the name of a file as parameter, loads the bitmap, displays it and starts the interactive loop.

Earth worm

The earth worm is a small, longish organism of a certain size which grows over time while eating objects in a world. The earth worm moves constantly in one direction. The only actions allowing a player to control it are changes in direction. The earth worm vanishes if it touches a border of the world or if it passes over a part of its body. It is most often represented by a vector of coordinates with two principal indices: its head and its tail. A move will therefore be computed from the new coordinates of its head, will display it and erase the tail. A growth step only modifies its head without affecting the tail of the earth worm.

1. Write the Objective Caml type or types for representing an earth worm and the world where it evolves. One can represent an earth worm by a queue of its coordinates.

2. Write a function for initialization and displaying an earth worm in a world.
3. Modify the function `skel` of the skeleton of the program which causes an action at each execution of the interactive loop, parameterized by a function. The treatment of keyboard events must not block.
4. Write a function `run` which advances the earth worm in the game. This function raises the exception `Victory` (if the worm reaches a certain size) and `Loss` if it hits a full slot or a border of the world.
5. Write a function for keyboard interaction which modifies the direction of the earth worm.
6. Write the other utility functions for handling interaction and pass them to the new skeleton of the program.
7. Write the initiating function which starts the application.

Summary

This chapter has presented the basic notions of graphics programming and event-driven programming using the `Graphics` library in the distribution of Objective Caml. After having explained the basic graphical elements (colors, drawing, filling, text and bitmaps) we have approached the problem of animating them. The mechanism of handling events in `Graphics` was then described in a way that allowed the introduction of a general method of handling user interaction. This was accomplished by taking a game as model for event-driven programming. To improve user interactions and to provide interactive graphical components to the programmer, we have developed a new library called `Awi`, which facilitates the construction of graphical interfaces. This library was used for writing the interface to the imperative calculator.

To learn more

Although graphics programming is naturally event-driven, the associated style of programming being imperative, it is not only possible but also often useful to introduce more functional operators to manipulate graphical objects. A good example comes from the use of the `MLgraph` library,

Link: http://www.pps.jussieu.fr/~cousinea/MLgraph/mlgraph.html

which implements the graphical model of PostScript and proposes functional operators to manipulate images. It is described in [CC92, CS94] and used later in [CM98] for the optimized placement of trees to construct drawings in the style of Escher.

One interesting characteristic of the `Graphics` library is that it is portable to the graphical interfaces of Windows, MacOS and Unix. The notion of virtual bitmaps can be found in several languages like Le_Lisp and more recently in Java. Unfortunately, the `Graphics` library in Objective Caml does not possess interactive components for

the construction of interfaces. One of the applications described in part II of this book contains the first bricks of the `Awi` library. It is inspired by the *Abstract Windowing Toolkit* of the first versions of Java. One can perceive that it is relatively easy to extend the functionality of this library thanks to the existence of functional values in the language. Therefore chapter 16 compares the adaptation of object oriented programming and functional and modular programming for the construction of graphical interfaces. The example of `Awi` is functional and imperative, but it is also possible to only use the functional style. This is typically the case for purely functional languages. We cite the systems `Fran` and `Fudget` developed in Haskell and derivatives. The system `Fran` permits construction of interactive animations in 2D and 3D, which means with events between animated objects and the user.

Link: http://www.research.microsoft.com/~conal/fran/

The `Fudget` library is intended for the construction of graphical interfaces.

Link: http://www.cs.chalmers.se/ComputingScience/Research/Functional/Fudgets/

One of the difficulties when one wants to program a graphical interface for ones application is to know which of the numerous existing libraries to choose. It is not sufficient to determine the language and the system to fix the choice of the tool. For Objective Caml there exist several more or less complete ones:

- the encapsulation of `libX`, for X-Windows;
- the `librt` library, also for X-Windows;
- `ocamltk`, an adaptation of `Tcl/Tk`, portable;
- `mlgtk`, an adaptation of `Gtk`, portable.

We find the links to these developments in the "Caml Hump":

Link: http://caml.inria.fr/hump.html

Finally, we have only discussed programming in 2D. The tendency is to add one dimension. Functional languages must also respond to this necessity, perhaps in the model of VRML or the Java 3D-extension. In purely functional languages the system `Fran` offers interesting possibilities of interaction between *sprites*. More closely to Objective Caml one can use the `VRcaML` library or the development environment SCOL.

The `VRcaML` library was developed in the manner of `MLgraph` and integrates a part of the graphical model of VRML in Objective Caml.

Link: http://www.pps.jussieu.fr/~emmanuel/Public/enseignement/VRcaML

One can therefore construct animated scenes in 3D. The result is a VRML-file that can be directly visualized.

Still in the line of Caml, the language SCOL is a functional communication language with important libraries for 2D and 3D manipulations, which is intended as environment for people with little knowledge in computer science.

Link: http://www.cryo-networks.com

The interest in the language SCOL and its development environment is to be able to create distributed applications, e.g. client-server, thus facilitating the creation of Internet sites. We present distributed programming in Objective Caml in chapter 20.

6

Applications

The reason to prefer one programming language over another lies in the ease of developing and maintaining robust applications. Therefore, we conclude the first part of this book, which dealt with a general presentation of the Objective Caml language, by demonstrating its use in a number of applications.

The first application implements a few functions which are used to write database queries. We emphasize the use of list manipulations and the functional programming style. The user has access to a set of functions with which it is easy to write and run queries using the Objective Caml language directly. This application shows the programmer how he can easily provide the user with most of the query tools that the user should need.

The second application is an interpreter for a tiny BASIC[1]. This kind of imperative language fueled the success of the first microcomputers. Twenty years later, they seem to be very easy to design. Although BASIC is an imperative language, the implementation of the interpreter uses the functional features of Objective Caml, especially for the evaluation of commands. Nevertheless, the lexer and parser for the language use a mutable structure.

The third application is a one-player game, Minesweeper, which is fairly well-known since it is bundled with the standard installation of Windows systems. The goal of the game is to uncover a bunch of hidden mines by repeatedly uncovering a square, which then indicates the number of mines around itself. The implementation uses the imperative features of the language, since the data structure used is a two-dimensional array which is modified after each turn of the game. This application uses the `Graphics` module to draw the game board and to interact with the player. However, the automatic uncovering of some squares will be written in a more functional style.
This latter application uses functions from the `Graphics` module described in chapter

1. which means “Beginner’s All purpose Symbolic Instruction Code”.

5 (see page 117) as well as some functions from the `Random` and `Sys` modules (see chapter 8, pages 216 and 234).

Database queries

The implementation of a database, its interface, and its query language is a project far too ambitious for the scope of this book and for the Objective Caml knowledge of the reader at this point. However, restricting the problem and using the functional programming style at its best allows us to create an interesting tool for query processing. For instance, we show how to use iterators as well as partial application to formulate and execute queries. We also show the use of a data type encapsulating functional values.

For this application, we use as an example a database on the members of an association. It is presumed to be stored in the file `association.dat`.

Data format

Most database programs use a "proprietary" format to store the data they manipulate. However, it is usually possible to store the data as some text that has the following structure:

- the database is a list of *cards* separated by carriage-returns;
- each card is a list of *fields* separated by some given character, `':'` in our case;
- a field is a string which contains no carriage-return nor the character `':'`;
- the first card is the list of the names associated with the fields, separated by the character `'|'`.

The association data file starts with:

```
Num|Lastname|Firstname|Address|Tel|Email|Pref|Date|Amount
0:Chailloux:Emmanuel:Université P6:0144274427:ec@lip6.fr:email:25.12.1998:100.00
1:Manoury:Pascal:Laboratoire PPS::pm@lip6.fr:mail:03.03.1997:150.00
2:Pagano:Bruno:Cristal:0139633963::mail:25.12.1998:150.00
3:Baro:Sylvain::0144274427:baro@pps.fr:email:01.03.1999:50.00
```

The meaning of the fields is the following:

- `Num` is the member number;
- `Lastname`, `Firstname`, `Address`, `Tel`, and `Email` are obvious;
- `Pref` indicates the means by which the member wishes to be contacted: by mail (`mail`), by email (`email`), or by phone (`tel`);
- `Date` and `Amount` are the date and the amount of the last membership fee received, respectively.

We need to decide what represention the program should use internally for a database. We could use either a list of cards or an array of cards. On the one hand, a list has the nice property of being easily modified: adding and removing a card are simple operations. On the other hand, an array allows constant access time to any card. Since our goal is to work on all the cards and not on some of them, each query accesses all the cards. Thus a list is a good choice. The same issue arises concerning the cards themselves: should they be lists or arrays of strings? This time an array is a good choice, since the format of a card is fixed for the whole database. It not possible to add a new field. Since a query might access only a few fields, it is important for this access to be fast.

The most natural solution for a card would be to use an array indexed by the names of the fields. Since such a type is not available in Objective Caml, we can use an array (indexed by integers) and a function associating a field name with the array index corresponding to the field.

```
# type data_card = string array ;;
# type data_base = { card_index : string → int ; data : data_card list } ;;
```

Access to the field named `n` of a card `dc` of the database `db` is implemented by the function:

```
# let field db n (dc : data_card) = dc.(db.card_index n) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

The type of `dc` has been set to *data_card* to constrain the function `field` to only accept string arrays and not arrays of other types.

Here is a small example:

```
# let base_ex =
    { data = [ [|"Chailloux"; "Emmanuel"|] ; [|"Manoury"; "Pascal"|] ]  ;
      card_index = function "Lastname"→0 | "Firstname"→1
                            | _->raise Not_found  } ;;
val base_ex : data_base =
  {card_index=<fun>;
   data=[[|"Chailloux"; "Emmanuel"|]; [|"Manoury"; "Pascal"|]]}
# List.map (field base_ex "Lastname") base_ex.data ;;
- : string list = ["Chailloux"; "Manoury"]
```

The expression *field base_ex* `"Lastname"` evaluates to a function which takes a card and returns the value of its `"Lastname"` field. The library function `List.map` applies the function to each card of the database `base_ex`, and returns the list of the results: a list of the `"Lastname"` fields of the database.

This example shows how we wish to use the functional style in our program. Here, the partial application of `field` allows us to define an access function for a given field, which we can use on any number of cards. This also shows us that the implementation of the `field` function is not very efficient, since although we are always accessing the same field, its index is computed for each access. The following implementation is better:

```
# let field base name =
    let i = base.card_index name in fun (card : data_card) → card.(i) ;;
val field : data_base -> string -> data_card -> string = <fun>
```

Here, after applying the function to two arguments, the index of the field is computed and is used for any subsequent application.

Reading a database from a file

As seen from Objective Caml, a file containing a database is just a list of lines. The first work that needs to be done is to read each line as a string, split it into smaller parts according to the separating character, and then extract the corresponding data as well as the field indexing function.

Tools for processing a line

We need a function `split` that splits a string at every occurrence of some separating character. This function uses the function `suffix` which returns the suffix of a string `s` after some position `i`. To do this, we use three predefined functions:

- `String.length` returns the length of a string;
- `String.sub` returns the substring of `s` starting at position `i` and of length `l`;
- `String.index_from` computes the position of the first occurrence of character `c` in the string `s`, starting at position `n`.

```
# let suffix s i = try String.sub s i ((String.length s)-i)
                   with Invalid_argument("String.sub") → "" ;;
val suffix : string -> int -> string = <fun>
# let split c s =
    let rec split_from n =
      try  let p = String.index_from s n c
           in (String.sub s n (p-n)) :: (split_from (p+1))
      with Not_found → [ suffix s n ]
    in if s="" then [] else split_from 0 ;;
val split : char -> string -> string list = <fun>
```

The only remarkable characteristic in this implementation is the use of exceptions, specifically the exception `Not_found`.

Computing the *data_base* structure There is no difficulty in creating an array of strings from a list of strings, since this is what the `of_list` function in the `Array` module does. It might seem more complicated to compute the index function from a list of field names, but the `List` module provides all the needed tools.

Starting from a list of strings, we need to code a function that associates each string with an index corresponding to its position in the list.

```
# let mk_index list_names =
    let rec make_enum a b = if a > b then [] else a :: (make_enum (a+1) b) in
    let list_index =  (make_enum 0 ((List.length list_names) - 1)) in
    let assoc_index_name = List.combine list_names list_index  in
      function name → List.assoc name assoc_index_name  ;;
val mk_index : 'a list -> 'a -> int = <fun>
```

To create the association function between field names and indexes, we combine the list of indexes and the list of names to obtain a list of associations of the type *string * int list*. To look up the index associated with a name, we use the function `assoc` from the `List` library. The function `mk_index` returns a function that takes a name and calls `assoc` on this name and the previously built association list.

It is now possible to create a function that reads a file of the given format.

```
# let read_base filename =
    let channel = open_in filename in
    let split_line = split ':' in
    let list_names = split '|' (input_line channel) in
    let rec read_file () =
      try
        let data = Array.of_list (split_line (input_line channel )) in
          data :: (read_file ())
      with End_of_file →  close_in channel ; []
    in
      { card_index = mk_index list_names ; data = read_file () } ;;
val read_base : string -> data_base = <fun>
```

The auxiliary function `read_file` reads records from the file, and works recursively on the input channel. The base case of the recursion corresponds to the end of the file, signaled by the `End_of_file` exception. In this case, the empty list is returned after closing the channel.

The association's file can now be loaded:

```
# let base_ex = read_base "association.dat" ;;
val base_ex : data_base =
  {card_index=<fun>;
   data=
    [[|"0"; "Chailloux"; "Emmanuel"; "Universit\233 P6"; "0144274427";
       "ec@lip6.fr"; "email"; "25.12.1998"; "100.00"|];
     [|"1"; "Manoury"; "Pascal"; "Laboratoire PPS"; ...|]; ...]}
```

General principles for database processing

The effectiveness and difficulty of processing the data in a database is proportional to the power and complexity of the query language. Since we want to use Objective Caml as query language, there is no limit *a priori* on the requests we can express! However,

we also want to provide some simple tools to manipulate cards and their data. This desire for simplicity requires us to limit the power of the Objective Caml language, through the use of general goals and principles for database processing.

The goal of database processing is to obtain a *state* of the database. Building such a state may be decomposed into three steps:

1. selecting, according to some given criterion, a set of cards;
2. processing each of the selected cards;
3. processing all the data collected on the cards.

Figure 6.1 illustrates this decomposition.

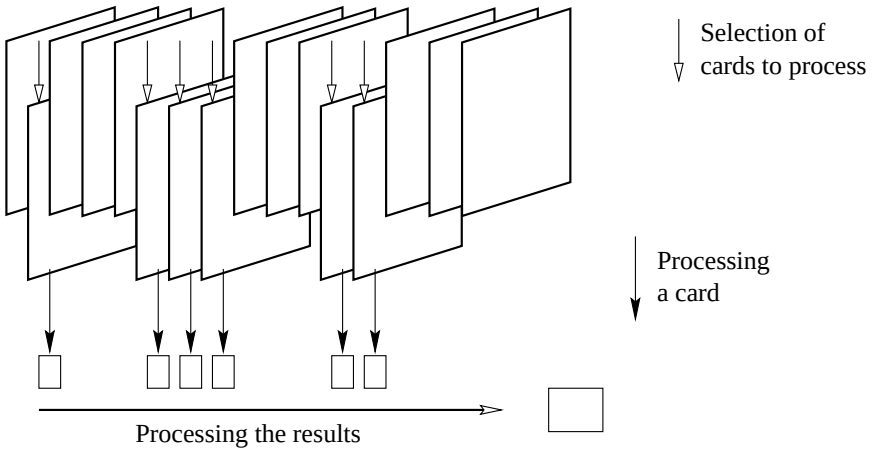


Figure 6.1: Processing a request.

According to this decomposition, we need three functions of the following types:

1. `(data_card -> bool) -> data_card list -> data_card list`
2. `(data_card -> 'a) -> data_card list -> 'a list`
3. `('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

Objective Caml provides us with three higher-order function, also known as iterators, introduced page 219, that satisfy our specification:

```
# List.find_all ;;
- : ('a -> bool) -> 'a list -> 'a list = <fun>
# List.map ;;
- : ('a -> 'b) -> 'a list -> 'b list = <fun>
# List.fold_right ;;
- : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b = <fun>
```

We will be able to use them to implement the three steps of building a state by choosing the functions they take as an argument.

For some special requests, we will also use:

```
# List.iter ;;
- : ('a -> unit) -> 'a list -> unit = <fun>
```

Indeed, if the required processing consists only of displaying some data, there is nothing to compute.

In the next paragraphs, we are going to see how to define functions expressing simple selection criteria, as well as simple queries. We conclude this section with a short example using these functions according to the principles stated above.

Selection criteria

Concretely, the boolean function corresponding to the selection criterion of a card is a boolean combination of properties of some or all of the fields of the card. Each field of a card, even though it is a string, can contain some information of another type: a float, a date, etc.

Selection criteria on a field

Selecting on some field is usually done using a function of the type *data_base -> 'a -> string -> data_card -> bool*. The *'a* type parameter corresponds to the type of the information contained in the field. The *string* argument corresponds to the name of the field.

String fields We define two simple tests on strings: equality with another string, and non-emptiness.

```
# let eq_sfield db s n dc =  (s = (field db n dc)) ;;
val eq_sfield : data_base -> string -> string -> data_card -> bool = <fun>
# let nonempty_sfield db n dc = ("" <> (field db n dc)) ;;
val nonempty_sfield : data_base -> string -> data_card -> bool = <fun>
```

Float fields To implement tests on data of type float, it is enough to translate the *string* representation of a decimal number into its *float* value. Here are some examples obtained from a generic function `tst_ffield`:

```
# let tst_ffield r db v n dc = r v (float_of_string (field db n dc)) ;;
val tst_ffield :
  ('a -> float -> 'b) -> data_base -> 'a -> string -> data_card -> 'b = <fun>
# let eq_ffield = tst_ffield (=) ;;
# let lt_ffield = tst_ffield (<) ;;
# let le_ffield = tst_ffield (<=) ;;
(* etc. *)
```

These three functions have type:

data_base -> float -> string -> data_card -> bool.

Dates This kind of information is a little more complex to deal with, as it depends on the representation format of dates, and requires that we define date comparison.

We decide to represent dates in a card as a string with format `dd.mm.yyyy`. In order to be able to define additional comparisons, we also allow the replacement of the day, month or year part with the underscore character (`'_'`). Dates are compared according to the lexicographic order of lists of integers of the form *[year; month; day]*. To express queries such as: "is before July 1998", we use the *date pattern*: `"_.07.1998"`. Comparing a date with a pattern is accomplished with the function `tst_dfield` which analyses the pattern to create the *ad hoc* comparison function. To define this generic test function on dates, we need a few auxiliary functions.

We first code two conversion functions from dates (`ints_of_string`) and date patterns (`ints_of_dpat`) to lists of ints. The character `'_'` of a pattern will be replaced by the integer 0:

```
# let split_date = split '.' ;;
val split_date : string -> string list = <fun>
# let ints_of_string d =
    try  match split_date d with
            [d;m;y] → [int_of_string y; int_of_string m; int_of_string d]
          |    _   → failwith "Bad date format"
    with Failure("int_of_string") → failwith "Bad date format" ;;
val ints_of_string : string -> int list = <fun>

# let ints_of_dpat d =
    let int_of_stringpat = function "_" → 0 | s → int_of_string s
    in try  match split_date d with
                [d;m;y] → [ int_of_stringpat y; int_of_stringpat m;
                            int_of_stringpat d ]
              | _ → failwith "Bad date format"
       with Failure("int_of_string") → failwith "Bad date pattern" ;;
val ints_of_dpat : string -> int list = <fun>
```

Given a relation `r` on integers, we now code the test function. It simply consists of implementing the lexicographic order, taking into account the particular case of 0:

```
# let rec app_dtst r d1 d2 = match d1, d2 with
     [] , [] → false
   | (0 :: d1) , (_::d2) → app_dtst r d1 d2
   | (n1 :: d1) , (n2 :: d2) → (r n1 n2) || ((n1 = n2) && (app_dtst r d1 d2))
   | _, _ → failwith "Bad date pattern or format" ;;
val app_dtst : (int -> int -> bool) -> int list -> int list -> bool = <fun>
```

We finally define the generic function `tst_dfield` which takes as arguments a relation `r`, a database `db`, a pattern `dp`, a field name `nm`, and a card `dc`. This function checks that the pattern and the field from the card satisfy the relation.

```
# let tst_dfield r db dp nm dc =
```

```
    r (ints_of_dpat dp) (ints_of_string (field db nm dc)) ;;
val tst_dfield :
  (int list -> int list -> 'a) ->
  data_base -> string -> string -> data_card -> 'a = <fun>
```

We now apply it to three relations.

```
# let eq_dfield = tst_dfield (=) ;;
# let le_dfield = tst_dfield (<=) ;;
# let ge_dfield = tst_dfield (>=) ;;
```

These three functions have type:
data_base -> string -> string -> data_card -> bool.

Composing criteria

The tests we have defined above all take as first arguments a database, a value, and the name of a field. When we write a query, the value of these three arguments are known. For instance, when we work on the database `base_ex`, the test "is before July 1998" is written

```
# ge_dfield base_ex "_.07.1998" "Date" ;;
- : data_card -> bool = <fun>
```

Thus, we can consider a test as a function of type *data_card -> bool*. We want to obtain boolean combinations of the results of such functions applied to a given card. To this end, we implement the iterator:

```
# let fold_funs b c fs dc =
    List.fold_right (fun f → fun r → c (f dc) r) fs b ;;
val fold_funs : 'a -> ('b -> 'a -> 'a) -> ('c -> 'b) list -> 'c -> 'a = <fun>
```

Where `b` is the base value, the function `c` is the boolean operator, `fs` is the list of test functions on a field, and `dc` is a card.

We can obtain the conjunction and the disjunction of a list of tests with:

```
# let and_fold fs = fold_funs true (&) fs ;;
val and_fold : ('a -> bool) list -> 'a -> bool = <fun>
# let or_fold fs = fold_funs false (or) fs ;;
val or_fold : ('a -> bool) list -> 'a -> bool = <fun>
```

We easily define the negation of a test:

```
# let not_fun f dc = not (f dc) ;;
val not_fun : ('a -> bool) -> 'a -> bool = <fun>
```

For instance, we can use these combinators to define a selection function for cards whose date field is included in a given range:

```
# let date_interval db d1 d2 =
    and_fold [(le_dfield db d1 "Date"); (ge_dfield db d2 "Date")] ;;
val date_interval : data_base -> string -> string -> data_card -> bool =
```

```
  <fun>
```

Processing and computation

It is difficult to guess how a card might be processed, or the data that would result from that processing. Nevertheless, we can consider two common cases: numerical computation and data formatting for printing. Let's take an example for each of these two cases.

Data formatting

In order to print, we wish to create a string containing the name of a member of the association, followed by some information.

We start with a function that reverses the splitting of a line using a given separating character:

```
# let format_list c =
    let s = String.make 1 c in
    List.fold_left (fun x y → if x="" then y else x^s^y) "" ;;
val format_list : char -> string list -> string = <fun>
```

In order to build the list of fields we are interested in, we code the function `extract` that returns the fields associated with a given list of names in a given card:

```
# let extract db ns dc =
     List.map (fun n → field db n dc) ns  ;;
val extract : data_base -> string list -> data_card -> string list = <fun>
```

We can now write the line formatting function:

```
# let format_line db ns dc =
    (String.uppercase (field db "Lastname" dc))
    ^" "^(field db "Firstname" dc)
    ^"\t"^(format_list '\t' (extract db ns dc))
    ^"\n" ;;
val format_line : data_base -> string list -> data_card -> string = <fun>
```

The argument `ns` is the list of requested fields. In the resulting string, fields are separated by a tab (`'\t'`) and the string is terminated with a newline character.

We display the list of last and first names of all members with:

```
# List.iter print_string (List.map (format_line base_ex []) base_ex.data) ;;
CHAILLOUX Emmanuel
MANOURY Pascal
PAGANO Bruno
BARO Sylvain
- : unit = ()
```

Numerical computation

We want to compute the total amount of received fees for a given set of cards. This is easily done by composing the extraction and conversion of the correct field with the addition. To get nicer code, we define an infix composition operator:

```
# let (++) f g x = g (f x) ;;
val ++ : ('a -> 'b) -> ('b -> 'c) -> 'a -> 'c = <fun>
```

We use this operator in the following definition:

```
# let total db dcs =
    List.fold_right ((field db "Amount") ++ float_of_string ++ (+.)) dcs 0.0 ;;
val total : data_base -> data_card list -> float = <fun>
```

We can now apply it to the whole database:

```
# total base_ex base_ex.data ;;
- : float = 450
```

An example

To conclude, here is a small example of an application that uses the principles described in the paragraphs above.

We expect two kinds of queries on our database:

- a query returning two lists, the elements of the first containing the name of a member followed by his mail address, the elements of the other containing the name of the member followed by his email address, according to his preferences.
- another query returning the state of received fees for a given period of time. This state is composed of the list of last and first names, dates and amounts of the fees as well as the total amount of the received fees.

List of addresses

To create these lists, we first select the relevant cards according to the field `"Pref"`, then we use the formatting function `format_line`:

```
# let mail_addresses db =
   let dcs = List.find_all (eq_sfield db "mail" "Pref") db.data in
    List.map (format_line db ["Mail"]) dcs ;;
val mail_addresses : data_base -> string list = <fun>

# let email_addresses db =
   let dcs = List.find_all (eq_sfield db "email" "Pref") db.data in
    List.map (format_line db ["Email"]) dcs ;;
val email_addresses : data_base -> string list = <fun>
```

State of received fees

Computing the state of the received fees uses the same technique: selection then processing. In this case however the processing part is twofold: line formatting followed by the computation of the total amount.

```
# let fees_state db d1 d2 =
    let dcs = List.find_all (date_interval db d1 d2) db.data in
    let ls = List.map (format_line db ["Date";"Amount"]) dcs in
    let t = total db dcs in
     ls, t ;;
val fees_state : data_base -> string -> string -> string list * float = <fun>
```

The result of this query is a tuple containing a list of strings with member information, and the total amount of received fees.

Main program

The main program is essentially an interactive loop that displays the result of queries asked by the user through a menu. We use here an imperative style, except for the display of the results which uses an iterator.

```
# let main() =
    let db = read_base "association.dat" in
    let finished = ref false in
     while not !finished do
      print_string" 1: List of mail addresses\n";
      print_string" 2: List of email addresses\n";
      print_string" 3: Received fees\n";
      print_string" 0: Exit\n";
      print_string"Your choice: ";
      match read_int() with
        0 → finished := true
      | 1 → (List.iter print_string (mail_addresses db))
      | 2 → (List.iter print_string (email_addresses db))
      | 3
       → (let d1 = print_string"Start date: "; read_line() in
           let d2 = print_string"End date: "; read_line() in
           let ls, t = fees_state db d1 d2 in
            List.iter print_string ls;
            print_string"Total: "; print_float t; print_newline())
      | _ → ()
     done;
     print_string"bye\n" ;;
val main : unit -> unit = <fun>
```

This example will be extended in chapter 21 with an interface using a web browser.

Further work

A natural extension of this example would consist of adding type information to every field of the database. This information would be used to define generic comparison operators with type `data_base -> 'a -> string -> data_card -> bool` where the name of the field (the third argument) would trigger the correct conversion and test functions.

BASIC interpreter

The application described in this section is a program interpreter for Basic. Thus, it is a program that can run other programs written in Basic. Of course, we will only deal with a restricted language, which contains the following commands:

- **PRINT** *expression*
  Prints the result of the evaluation of the expression.
- **INPUT** *variable*
  Prints a *prompt* (`?`), reads an integer typed in by the user, and assigns its value to the variable.
- **LET** *variable* = *expression*
  Assigns the result of the evaluation of *expression* to the variable.
- **GOTO** *line number*
  Continues execution at the given line.
- **IF** *condition* **THEN** *line number*
  Continues execution at the given line if the *condition* is true.
- **REM** *any string*
  One-line comment.

Every line of a Basic program is labelled with a line number, and contains only one command. For instance, a program that computes and then prints the factorial of an integer given by the user is written:

```
 5  REM inputting the argument
10  PRINT " factorial of:"
20  INPUT A
30  LET B = 1
35  REM beginning of the loop
40  IF A <= 1 THEN 80
50  LET B = B * A
60  LET A = A - 1
70  GOTO 40
75  REM prints the result
80  PRINT B
```

We also wish to write a small text editor, working as a toplevel interactive loop. It should be able to add new lines, display a program, execute it, and display the result.

Execution of the program is started with the `RUN` command. Here is an example of the evaluation of this program:

```
> RUN
 factorial of: ? 5
120
```

The interpreter is implemented in several distinct parts:

Description of the abstract syntax : describes the definition of data types to represent Basic programs, as well as their components (lines, commands, expressions, etc.).

Program pretty printing : consists of transforming the internal representation of Basic programs to strings, in order to display them.

Lexing and parsing : accomplish the inverse transformation, that is, transform a string into the internal representation of a Basic program (the abstract syntax).

Evaluation : is the heart of the interpreter. It controls and runs the program. As we will see, functional languages, such as Objective Caml, are particularly well adapted for this kind of problem.

Toplevel interactive loop : glues together all the previous parts.

Abstract syntax

Figure 6.2 introduces the concrete syntax, as a BNF grammar, of the Basic we will implement. This kind of description for language syntaxes is described in chapter 11, page 295.

We can see that the way expressions are defined does not ensure that a *well formed* expression can be evaluated. For instance, `1+"hello"` is an expression, and yet it is not possible to evaluate it. This deliberate choice lets us simplify both the abstract syntax and the parsing of the Basic language. The price to pay for this choice is that a syntactically correct Basic program may generate a runtime error because of a type mismatch.

Defining Objective Caml data types for this abstract syntax is easy, we simply translate the concrete syntax into a sum type:

```
# type unr_op = UMINUS | NOT  ;;
# type bin_op = PLUS | MINUS | MULT | DIV | MOD
                | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF
                | AND | OR  ;;
# type expression =
       ExpInt of int
     | ExpVar of string
     | ExpStr of string
```

UNARY_OP	::=	− | !
BINARY_OP	::=	+ | − | * | / | %
	|	= | < | > | <= | >= | <>
	|	& | '|'
EXPRESSION	::=	*integer*
	|	*variable*
	|	`"string"`
	|	UNARY_OP EXPRESSION
	|	EXPRESSION BINARY_OP EXPRESSION
	|	(EXPRESSION)
COMMAND	::=	**REM** *string*
	|	**GOTO** *integer*
	|	**LET** *variable* = EXPRESSION
	|	**PRINT** EXPRESSION
	|	**INPUT** *variable*
	|	**IF** EXPRESSION **THEN** *integer*
LINE	::=	*integer* COMMAND
PROGRAM	::=	LINE
	|	LINE PROGRAM
PHRASE	::=	LINE | **RUN** | **LIST** | **END**

Figure 6.2: BASIC Grammar.

```
    | ExpUnr of unr_op * expression
    | ExpBin of expression * bin_op * expression  ;;
# type command =
      Rem of string
    | Goto of int
    | Print of expression
    | Input of string
    | If of expression * int
    | Let of string * expression  ;;
# type line = { num : int ; cmd : command }  ;;
# type program = line list  ;;
```

We also define the abstract syntax for the commands for the small program editor:

```
# type phrase =  Line of line | List | Run | PEnd  ;;
```

It is convenient to allow the programmer to skip some parentheses in arithmetic expressions. For instance, the expression $1+3*4$ is usually interpreted as $1+(3*4)$. To this end, we associate an integer with each operator of the language:

```
# let priority_uop = function NOT → 1 | UMINUS → 7
  let priority_binop = function
      MULT | DIV  → 6
    | PLUS | MINUS → 5
    | MOD → 4
    | EQUAL | LESS | LESSEQ | GREAT | GREATEQ | DIFF → 3
    | AND | OR → 2 ;;
val priority_uop : unr_op -> int = <fun>
val priority_binop : bin_op -> int = <fun>
```

These integers indicate the *priority* of the operators. They will be used to print and parse programs.

Program pretty printing

To print a program, one needs to be able to convert abstract syntax program lines into strings.

Converting operators is easy:

```
# let pp_binop = function
      PLUS → "+" | MULT → "*" | MOD   → "%" | MINUS → "-"
    | DIV → "/"  | EQUAL → " = " | LESS → " < "
    | LESSEQ → " <= "   | GREAT   → " > "
    | GREATEQ → " >= "   | DIFF  → " <> " | AND → " & " | OR → " | "
  let pp_unrop = function UMINUS → "-"  | NOT → "!"  ;;
val pp_binop : bin_op -> string = <fun>
val pp_unrop : unr_op -> string = <fun>
```

Expression printing needs to take into account operator priority to print as few parentheses as possible. For instance, parentheses are put around a subexpression at the right of an operator only if the subexpression's main operator has a lower priority that the main operator of the whole expression. Also, arithmetic operators are left-associative, thus the expression $1-2-3$ is interpreted as $(1-2)-3$.

To deal with this, we use two auxiliary functions `ppl` and `ppr` to print left and right subtrees, respectively. These functions take two arguments: the tree to print and the priority of the enclosing operator, which is used to decide if parentheses are necessary. Left and right subtrees are distinguished to deal with associativity. If the current operator priority is the same than the enclosing operator priority, left trees do not need parentheses whereas right ones may require them, as in $1-(2-3)$ or $1-(2+3)$.

The initial tree is taken as a left subtree with minimal priority (0). The expression pretty printing function `pp_expression` is:

```
# let parenthesis x = "(" ^ x ^ ")";;
val parenthesis : string -> string = <fun>
# let pp_expression =
    let rec ppl pr = function
        ExpInt n → (string_of_int n)
      | ExpVar v → v
      | ExpStr s → "\"" ^ s ^ "\""
      | ExpUnr (op,e) →
          let res = (pp_unrop op)^(ppl (priority_uop op) e)
          in if pr=0 then res else parenthesis res
      | ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          (* parenthesis if priority is not greater *)
          in if pr2 >= pr then res else parenthesis res
    and ppr pr exp = match exp with
      (*  right subtrees only differ for binary operators *)
        ExpBin (e1,op,e2) →
          let pr2 = priority_binop op
          in let res = (ppl pr2 e1)^(pp_binop op)^(ppr pr2 e2)
          in if pr2 > pr then res else parenthesis res
      | _ → ppl pr exp
    in ppl 0 ;;
val pp_expression : expression -> string = <fun>
```

Command pretty printing uses the expression pretty printing function. Printing a line consists of printing the line number before the command.

```
# let pp_command = function
      Rem s     →  "REM " ^ s
    | Goto n    →  "GOTO " ^ (string_of_int n)
    | Print e   →  "PRINT " ^ (pp_expression e)
    | Input v   →  "INPUT " ^ v
    | If (e,n)  →  "IF "^(pp_expression e)^" THEN "^(string_of_int n)
    | Let (v,e) →  "LET " ^ v ^ " = " ^ (pp_expression e)  ;;
val pp_command : command -> string = <fun>
# let pp_line l = (string_of_int l.num) ^ "  " ^ (pp_command l.cmd)  ;;
val pp_line : line -> string = <fun>
```

Lexing

Lexing and parsing do the inverse transformation of printing, going from a string to a syntax tree. Lexing splits the text of a command line into independent lexical units called *lexemes*, with Objective Caml type:

```
# type lexeme = Lint of int
              | Lident of string
```

```
              | Lsymbol of string
              | Lstring of string
              | Lend ;;
```

A particular lexeme denotes the end of an expression: `Lend`. It is not present in the text of the expression, but is created by the lexing function (see the `lexer` function, page 165).

The string being lexed is kept in a record that contains a mutable field indicating the position after which lexing has not been done yet. Since the size of the string is used several times and does not change, it is also stored in the record:

```
# type string_lexer = {string:string; mutable current:int; size:int } ;;
```

This representation lets us define the lexing of a string as the application of a function to a value of type *string_lexer* returning a value of type *lexeme*. Modifying the current position in the string is done as a side effect.

```
# let init_lex s = { string=s; current=0 ; size=String.length s } ;;
val init_lex : string -> string_lexer = <fun>
# let forward cl = cl.current <- cl.current+1  ;;
val forward : string_lexer -> unit = <fun>
# let forward_n cl n = cl.current <- cl.current+n ;;
val forward_n : string_lexer -> int -> unit = <fun>
# let extract pred cl =
    let st = cl.string and pos = cl.current in
    let rec ext n = if n<cl.size && (pred st.[n]) then ext (n+1) else n in
    let res = ext pos
    in cl.current <- res ; String.sub cl.string pos (res-pos)  ;;
val extract : (char -> bool) -> string_lexer -> string = <fun>
```

The following functions extract a lexeme from the string and modify the current position. The two functions `extract_int` and `extract_ident` extract an integer and an identifier, respectively.

```
# let extract_int =
     let is_int = function '0'..'9' → true | _ → false
     in function cl → int_of_string (extract is_int cl)
  let extract_ident =
     let is_alpha_num = function
       'a'..'z' | 'A'..'Z' | '0' .. '9' | '_' → true
     | _ → false
     in extract is_alpha_num ;;
val extract_int : string_lexer -> int = <fun>
val extract_ident : string_lexer -> string = <fun>
```

The `lexer` function uses the two previous functions to extract a lexeme.

```
# exception LexerError ;;
exception LexerError
```

```
# let  rec lexer cl =
    let lexer_char c = match c with
        ' '
      | '\t'     → forward cl ; lexer cl
      | 'a'..'z'
      | 'A'..'Z' → Lident (extract_ident cl)
      | '0'..'9' → Lint (extract_int cl)
      | '"'      → forward cl ;
                   let res = Lstring (extract ((<>) '"') cl)
                   in forward cl ; res
      | '+' | '-' | '*' | '/' | '%' | '&' | '|' | '!' | '=' | '(' | ')'  →
                   forward cl; Lsymbol (String.make 1 c)
      | '<'
      | '>'      → forward cl;
                   if cl.current >= cl.size then Lsymbol (String.make 1 c)
                   else  let cs = cl.string.[cl.current]
                         in ( match (c,cs) with
                                ('<','=') → forward cl; Lsymbol "<="
                              | ('>','=') → forward cl; Lsymbol ">="
                              | ('<','>') → forward cl; Lsymbol "<>"
                              |     _     → Lsymbol (String.make 1 c) )
      | _ → raise LexerError
    in
       if cl.current >= cl.size then Lend
       else lexer_char cl.string.[cl.current] ;;
val lexer : string_lexer -> lexeme = <fun>
```

The `lexer` function is very simple: it matches the current character of a string and, based on its value, extracts the corresponding lexeme and modifies the current position to the start of the next lexeme. The code is simple because, for all characters except two, the current character defines which lexeme to extract. In the more complicated cases of `'<'`, we need to look at the next character, which might be a `'='` or a `'>'`, producing two different lexemes. The same problem arises with `'>'`.

Parsing

The only difficulty in parsing our language comes from expressions. Indeed, knowing the beginning of an expression is not enough to know its structure. For instance, having parsed the beginning of an expression as being $1 + 2 + 3$, the resulting syntax tree for this part depends on the rest of the expression: its structure is different when it is followed by $+4$ or $*4$ (see figure 6.3). However, since the tree structure for $1 + 2$ is the same in both cases, it can be built. As the position of $+3$ in the structure is not fully known, it is temporarily stored.

To build the abstract syntax tree, we use a *pushdown automaton* similar to the one built by *yacc* (see page 303). Lexemes are read one by one and put on a stack until

Figure 6.3: Basic: abstract syntax tree examples.

there is enough information to build the expression. They are then removed from the stack and replaced by the expression. This latter operation is called *reduction*.

The stack elements have type:

```
# type exp_elem =
      Texp of expression   (* expression       *)
    | Tbin of bin_op       (* binary operator  *)
    | Tunr of unr_op       (* unary operator   *)
    | Tlp                  (* left parenthesis *) ;;
```

Right parentheses are not stored on the stack as only left parentheses matter for reduction.

Figure 6.4 illustrates the way the stack is used to parse the expression $(1 + 2 * 3) + 4$. The character above the arrow is the current character of the string.

We define an exception for syntax errors.

```
# exception ParseError ;;
```

The first step consists of transforming symbols into operators:

```
# let unr_symb = function
    "!" → NOT | "-" → UMINUS | _ → raise ParseError
  let bin_symb = function
      "+" → PLUS | "-" → MINUS | "*" → MULT | "/" → DIV | "%" → MOD
    | "=" → EQUAL | "<" → LESS | "<=" → LESSEQ | ">" → GREAT
    | ">=" → GREATEQ | "<>" → DIFF | "&" → AND | "|" → OR
    | _ → raise ParseError
  let tsymb s = try Tbin (bin_symb s) with ParseError → Tunr (unr_symb s) ;;
val unr_symb : string -> unr_op = <fun>
val bin_symb : string -> bin_op = <fun>
val tsymb : string -> exp_elem = <fun>
```

The `reduce` function implements stack reduction. There are two cases to consider, whether the stack starts with:

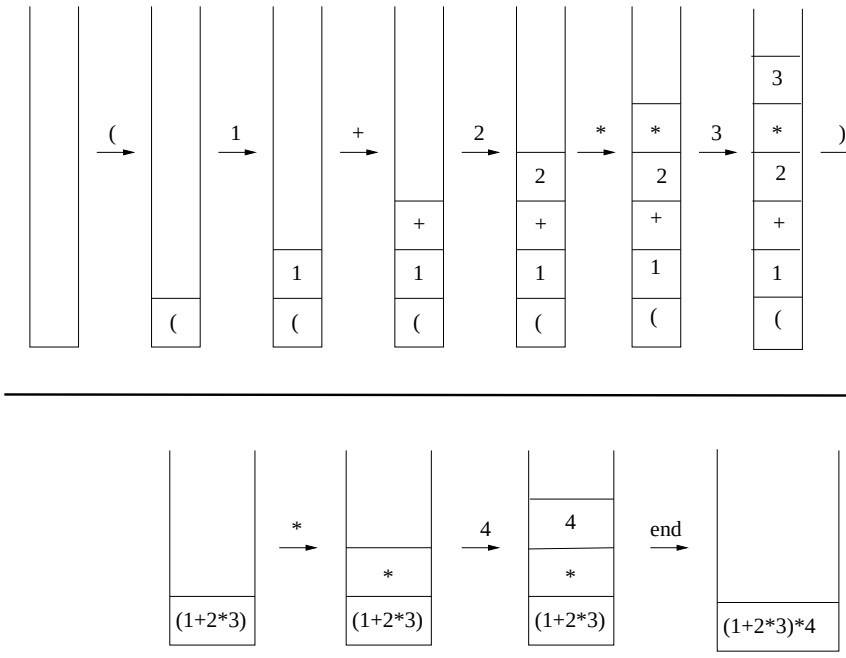


Figure 6.4: Basic: abstract syntax tree construction example.

- an expression followed by a unary operator,
- an expression followed by a binary operator and an expression.

Moreover, `reduce` takes an argument indicating the minimal priority that an operator should have to trigger reduction. To avoid this reduction condition, it suffices to give the minimal value, zero, as the priority.

```
# let reduce pr = function
      (Texp e) :: (Tunr op) :: st  when  (priority_uop op) >= pr
         → (Texp (ExpUnr (op,e))) :: st
    | (Texp e1) :: (Tbin op) :: (Texp e2) :: st  when  (priority_binop op) >= pr
         → (Texp (ExpBin (e2,op,e1))) :: st
    | _ → raise ParseError ;;
val reduce : int -> exp_elem list -> exp_elem list = <fun>
```

Notice that expression elements are stacked as they are read. Thus it is necessary to swap them when they are arguments of a binary operator.

The main function of our parser is `stack_or_reduce` that, according to the lexeme given in argument, puts it on the stack or triggers a reduction.

```
# let rec stack_or_reduce lex stack = match lex , stack with
      Lint n , _      →  (Texp (ExpInt n)) :: stack
    | Lident v , _    →  (Texp (ExpVar v)) :: stack
```

```
    | Lstring s , _    →  (Texp (ExpStr s)) :: stack
    | Lsymbol "(" , _  →  Tlp :: stack
    | Lsymbol ")" , (Texp e) :: Tlp :: st  →  (Texp e) :: st
    | Lsymbol ")" , _ → stack_or_reduce lex (reduce 0 stack)
    | Lsymbol s , _
        → let symbol =
            if s<>"-" then tsymb s
            (* remove the ambiguity of the ``-'' symbol           *)
            (* according to the last exp element put on the stack *)
            else match stack
                 with (Texp _)::_  →  Tbin MINUS
                               | _ →  Tunr UMINUS
          in ( match symbol with
                 Tunr op  →  (Tunr op) :: stack
               | Tbin op  →
                   ( try stack_or_reduce lex (reduce (priority_binop op)
                                                     stack )
                     with ParseError → (Tbin op) :: stack )
               | _ → raise ParseError )
    | _ , _ → raise ParseError ;;
val stack_or_reduce : lexeme -> exp_elem list -> exp_elem list = <fun>
```

Once all lexemes are defined and stacked, the function `reduce_all` builds the abstract syntax tree with the elements remaining in the stack. If the expression being parsed is well formed, only one element should remain in the stack, containing the tree for this expression.

```
# let rec reduce_all = function
    | [] → raise ParseError
    | [Texp x] → x
    | st → reduce_all (reduce 0 st) ;;
val reduce_all : exp_elem list -> expression = <fun>
```

The `parse_exp` function is the main expression parsing function. It reads a string, extracts its lexemes and passes them to the `stack_or_reduce` function. Parsing stops when the current lexeme satisfies a predicate that is given as an argument.

```
# let parse_exp stop cl =
    let p = ref 0 in
    let rec parse_one stack  =
      let l = ( p:=cl.current ; lexer cl)
      in if not (stop l) then parse_one (stack_or_reduce l stack)
         else ( cl.current <- !p ; reduce_all stack )
    in parse_one []  ;;
val parse_exp : (lexeme -> bool) -> string_lexer -> expression = <fun>
```

Notice that the lexeme that made the parsing stop is not used to build the expression. It is thus necessary to modify the current position to its beginning (variable `p`) to parse it later.

We can now parse a command line:

```
# let parse_cmd cl = match lexer cl with
      Lident s → ( match s with
          "REM" → Rem (extract (fun _ → true) cl)
        | "GOTO" → Goto (match lexer cl with
                           Lint p → p
                         | _ → raise ParseError)
        | "INPUT" → Input (match lexer cl with
                             Lident v → v
                           | _ → raise ParseError)
        | "PRINT" → Print (parse_exp ((=) Lend) cl)
        | "LET" →
            let l2 = lexer cl and l3 = lexer cl
            in ( match l2 ,l3 with
                     (Lident v,Lsymbol "=") → Let (v,parse_exp ((=) Lend) cl)
                   | _ → raise ParseError )
        | "IF" →
            let test = parse_exp ((=) (Lident "THEN")) cl
            in ( match ignore (lexer cl) ; lexer cl with
                    Lint n  →  If (test,n)
                  | _ → raise ParseError )
        | _ → raise ParseError )
    | _ → raise ParseError  ;;
val parse_cmd : string_lexer -> command = <fun>
```

Finally, we implement the function to parse commands typed by the user:

```
# let parse str =
    let cl = init_lex str
    in match lexer cl with
        Lint n → Line { num=n ; cmd=parse_cmd cl }
      | Lident "LIST" → List
      | Lident "RUN" → Run
      | Lident "END" → PEnd
      | _ → raise ParseError  ;;
val parse : string -> phrase = <fun>
```

Evaluation

A Basic program is a list of lines. Execution starts at the first line. Interpreting a program line consists of executing the task corresponding to its command. There are three different kinds of commands: input-output (**PRINT** and **INPUT**), variable declaration or modification (**LET**), and flow control (**GOTO** and **IF... THEN**). Input-output commands interact with the user and use the corresponding Objective Caml functions.

Variable declaration and modification commands need to know how to compute the value of an arithmetic expression and the memory location to store the result. Expression evaluation returns an integer, a boolean, or a string. Their type is `value`.

```
# type value = Vint of int | Vstr of string | Vbool of bool  ;;
```

Variable declaration should allocate some memory to store the associated value. Similarly, variable modification requires the modification of the associated value. Thus, evaluation of a Basic program uses an *environment* that stores the association between a variable name and its value. It is represented by an association list of tuples (name,value):

```
# type environment = (string * value) list ;;
```

The variable name is used to access its value. Variable modification modifies the association.

Flow control commands, conditional or unconditional, specify the number of the next line to execute. By default, it is the next line. To do this, it is necessary to remember the number of the current line.

The list of commands representing the program being edited under the toplevel is not an efficient data structure for running the program. Indeed, it is then necessary to look at the whole list of lines to find the line indicated by a flow control command (`If` and `goto`). Replacing the list of lines with an array of commands allows direct access to the command following a flow control command, using the array index instead of the line number in the flow control command. This solution requires some preprocessing called *assembly* before executing a `RUN` command. For reasons that will be detailed shortly, a program after assembly is not represented as an array of commands but as an array of lines:

```
# type code = line array ;;
```

As in the calculator example of previous chapters, the interpreter uses a state that is modified for each command evaluation. At each step, we need to remember the whole program, the next line to interpret and the values of the variables. The program being interpreted is not exactly the one that was entered in the toplevel: instead of a list of commands, it is an array of commands. Thus the state of a program during execution is:

```
# type state_exec = { line:int ; xprog:code ; xenv:environment } ;;
```

Two different reasons may lead to an error during the evaluation of a line: an error while computing an expression, or branching to an absent line. They must be dealt with so that the interpreter exits nicely, printing an error message. We define an exception as well as a function to raise it, indicating the line where the error occurred.

```
# exception RunError of int
  let runerr n = raise (RunError n) ;;
exception RunError of int
val runerr : int -> 'a = <fun>
```

Assembly Assembling a program that is a list of numbered lines (type *program*) consists of transforming this list into an array and modifying the flow control commands. This last modification only needs an association table between line numbers and array indexes. This is easily provided by storing lines (with their line numbers), instead of commands, in the array: to find the association between a line number and the index in the array, we look the line number up in the array and return the corresponding index. If no line is found with this number, the index returned is `-1`.

```
# exception Result_lookup_index of int ;;
exception Result_lookup_index of int
# let lookup_index tprog num_line =
    try
      for i=0 to (Array.length tprog)-1 do
        let num_i = tprog.(i).num
        in if num_i=num_line then raise (Result_lookup_index i)
           else if num_i>num_line then raise (Result_lookup_index (-1))
      done ;
      (-1 )
    with Result_lookup_index i → i ;;
val lookup_index : line array -> int -> int = <fun>

# let assemble prog =
    let tprog = Array.of_list prog in
      for i=0 to (Array.length tprog)-1 do
        match tprog.(i).cmd with
            Goto n → let index = lookup_index tprog n
                     in tprog.(i) <- { tprog.(i) with cmd = Goto index }
          | If(c,n) → let index = lookup_index tprog n
                      in tprog.(i) <- { tprog.(i) with cmd = If (c,index) }
          | _ → ()
      done ;
    tprog ;;
val assemble : line list -> line array = <fun>
```

Expression evaluation The evaluation function does a depth-first traversal on the abstract syntax tree, and executes the operations indicated at each node.

The `RunError` exception is raised in case of type inconsistency, division by zero, or an undeclared variable.

```
# let rec eval_exp n envt expr = match expr with
      ExpInt p  →  Vint p
    | ExpVar v  → ( try List.assoc v envt with Not_found → runerr n )
    | ExpUnr (UMINUS,e) →
        ( match eval_exp n envt e with
            Vint p → Vint (-p)
          | _ → runerr n )
    | ExpUnr (NOT,e) →
```

```
        ( match eval_exp n envt e with
            Vbool p → Vbool (not p)
          | _ → runerr n )
    | ExpStr s → Vstr s
    | ExpBin (e1,op,e2)
        → match eval_exp n envt e1 , op , eval_exp n envt e2 with
                Vint v1 , PLUS   , Vint v2  →  Vint (v1 + v2)
              | Vint v1 , MINUS  , Vint v2  →  Vint (v1 - v2)
              | Vint v1 , MULT   , Vint v2  →  Vint (v1 * v2)
              | Vint v1 ,  DIV   , Vint v2  when v2<>0 →  Vint (v1 / v2)
              | Vint v1 ,  MOD   , Vint v2  when v2<>0 →  Vint (v1 mod v2)

              | Vint v1 , EQUAL   , Vint v2  →  Vbool (v1 = v2)
              | Vint v1 , DIFF    , Vint v2  →  Vbool (v1 <> v2)
              | Vint v1 ,  LESS   , Vint v2  →  Vbool (v1 < v2)
              | Vint v1 ,  GREAT  , Vint v2  →  Vbool (v1 > v2)
              | Vint v1 , LESSEQ  , Vint v2  →  Vbool (v1 <= v2)
              | Vint v1 , GREATEQ , Vint v2  →  Vbool (v1 >= v2)

              | Vbool v1 , AND , Vbool v2  →  Vbool (v1 && v2)
              | Vbool v1 , OR , Vbool v2  →  Vbool (v1 || v2)

              | Vstr v1 , PLUS , Vstr v2 → Vstr (v1 ^ v2)
              | _ , _ , _  → runerr n  ;;
val eval_exp : int -> (string * value) list -> expression -> value = <fun>
```

Command evaluation To evaluate a command, we need a few additional functions.

We add an association to an environment by removing a previous association for the same variable name if there is one:

```
# let rec add v e env = match env with
      []  → [v,e]
    | (w,f) :: l → if w=v then (v,e) :: l else (w,f) :: (add v e l) ;;
val add : 'a -> 'b -> ('a * 'b) list -> ('a * 'b) list = <fun>
```

A function that prints the value of an integer or string is useful for evaluation of the PRINT command.

```
# let print_value v = match v with
      Vint n → print_int n
    | Vbool true → print_string "true"
    | Vbool false → print_string "false"
    | Vstr s → print_string s ;;
val print_value : value -> unit = <fun>
```

The execution of a command corresponds to a *transition* from one state to another. More precisely, the environment is modified if the command is an assignment. Furthermore, the next line to execute is always modified. As a convention, if the next line to execute does not exist, we set its value to `-1`

```
# let next_line state =
    let n = state.line+1 in
     if n < Array.length state.xprog then n else -1 ;;
val next_line : state_exec -> int = <fun>
# let eval_cmd state =
     match state.xprog.(state.line).cmd with
       Rem _     →  { state with line = next_line state }
     | Print e   →  print_value (eval_exp state.line state.xenv e) ;
                    print_newline () ;
                    { state with line = next_line state }
     | Let(v,e)  →  let ev = eval_exp state.line state.xenv e
                      in { state with line = next_line state ;
                                      xenv = add v ev state.xenv }
     | Goto n    →  { state with line = n }
     | Input v   →  let x = try read_int ()
                            with Failure "int_of_string" → 0
                    in { state with line = next_line state;
                                    xenv = add v (Vint x) state.xenv }
     | If (t,n)  →  match eval_exp state.line state.xenv t with
                      Vbool true  → { state with line = n }
                    | Vbool false → { state with line = next_line state }
                    | _ → runerr state.line  ;;
val eval_cmd : state_exec -> state_exec = <fun>
```

On each call of the transition function `eval_cmd`, we look up the current line, run it, then set the number of the next line to run as the current line. If the last line of the program is reached, the current line is given the value `-1`. This will tell us when to stop.

Program evaluation We recursively apply the transition function until we reach a state where the current line number is `-1`.

```
# let rec run state =
    if state.line = -1 then state else run (eval_cmd state) ;;
val run : state_exec -> state_exec = <fun>
```

Finishing touches

The only thing left to do is to write a small editor and to plug together all the functions we wrote in the previous sections.

The `insert` function adds a new line in the program at the requested place.

```
# let rec insert line p = match p with
       [] → [line]
    | l :: prog →
        if l.num < line.num then l :: (insert line prog)
        else if l.num=line.num then line :: prog
        else line :: l :: prog ;;
val insert : line -> line list -> line list = <fun>
```

The `print_prog` function prints the source code of a program.

```
# let print_prog prog =
    let print_line x = print_string (pp_line x) ; print_newline () in
     print_newline () ;
     List.iter print_line prog ;
     print_newline () ;;
val print_prog : line list -> unit = <fun>
```

The `one_command` function processes the insertion of a line or the execution of a command. It modifies the state of the toplevel loop, which consists of a program and an environment. This state, represented by the *loop_state* type, is different from the evaluation state.

```
# type loop_state = { prog:program; env:environment } ;;
# exception End ;;

# let one_command state =
    print_string "> " ; flush stdout ;
    try
      match parse (input_line stdin) with
         Line l → { state with prog = insert l state.prog }
       | List  → (print_prog state.prog ; state )
       | Run
        → let tprog = assemble state.prog in
           let xstate = run { line = 0; xprog = tprog; xenv = state.env } in
            {state with env = xstate.xenv }
       | PEnd → raise End
    with
        LexerError → print_string "Illegal character\n"; state
      | ParseError → print_string "syntax error\n"; state
      | RunError n →
          print_string "runtime error at line ";
          print_int n ;
          print_string "\n";
          state ;;
val one_command : loop_state -> loop_state = <fun>
```

The main function is the `go` function, which starts the toplevel loop of our Basic.

```
# let go () =
  try
    print_string "Mini-BASIC version 0.1\n\n";
    let rec loop state = loop (one_command state)  in
      loop { prog = []; env = [] }
  with End → print_string "See you later...\n";;
val go : unit -> unit = <fun>
```

The loop is implemented by the local function `loop`. It stops when the `End` exception is raised by the `one_command` function.

Example: C+/C-

We return to the example of the C+/C- game described in chapter 3, page 78. Here is the Basic program corresponding to that Objective Caml program:

```
10 PRINT "Give the hidden number: "
20 INPUT N
30 PRINT "Give a number: "
40 INPUT R
50 IF R = N THEN 110
60 IF R < N THEN 90
70 PRINT "C-"
80 GOTO 30
90 PRINT "C+"
100 GOTO 30
110 PRINT "CONGRATULATIONS"
```

And here is a sample run of this program.

```
> RUN
Give the hidden number:
64
Give a number:
88
C-
Give a number:
44
C+
Give a number:
64
CONGRATULATIONS
```

Further work

The Basic we implemented is minimalist. If you want to go further, the following exercises hint at some possible extensions.

1. *Floating-point numbers*: as is, our language only deals with integers, strings and booleans. Add floats, as well as the corresponding arithmetic operations in the language grammar. We need to modify not only parsing, but also evaluation, taking into account the implicit conversions between integers and floats.
2. *Arrays*: Add to the syntax the command `DIM var[x]` that declares an array `var` of size `x`, and the expression `var[i]` that references the `i`th element of the array `var`.
3. *Toplevel directives*: Add the toplevel directives `SAVE "file_name"` and `LOAD "file_name"` that save a Basic program to the hard disk, and load a Basic program from the hard disk, respectively.
4. *Sub-program*: Add sub-programs. The `GOSUB line number` command calls a sub-program by branching to the given line number while storing the line from where the call is made. The `RETURN` command resumes execution at the line following the last `GOSUB` call executed, if there is one, or exits the program otherwise. Adding sub-programs requires evaluation to manage not only the environement but also a stack containing the return addresses of the current `GOSUB` calls. The `GOSUB` command adds the possibility of defining recursive sub-programs.

Minesweeper

Let us briefly recall the object of this game: to explore a mine field without stepping on one. A mine field is a two dimensional array (a matrix) where some cells contain hidden mines while others are empty. At the beginning of the game, all the cells are closed and the player must open them one after another. The player wins when he opens all the cells that are empty.

Every turn, the player may open a cell or flag it as containing a mine. If he opens a cell that contains a mine, it blows up and the player loses. If the cell is empty, its appearance is modified and the number of mines in the 8 neighbor cells is displayed (thus at most 8). If the player decides to flag a cell, he cannot open it until he removes the flag.

We split the implementation of the game into three parts.

1. The abstract game, including the internal representation of the mine field as well as the functions manipulating this representation.
2. The graphical part of the game, including the function for displaying cells.
3. The interaction between the program and the player.

Figure 6.5: Screenshot.

The abstract mine field

This part deals with the mine field as an abstraction only, and does not address its display.

Configuration A mine field is defined by its dimensions and the number of mines it contains. We group these three pieces of data in a record and define a default configuration: 10×10 cells and 15 mines.

```
# type config = {
    nbcols  : int ;
    nbrows : int ;
    nbmines : int };;
# let default_config = { nbcols=10; nbrows=10; nbmines=15 } ;;
```

The mine field It is natural to represent the mine field as a two dimensional array. However, it is still necessary to specify what the cells are, and what information their encoding should provide. The state of a cell should answer the following questions:

- is there a mine in this cell?
- is this cell opened (has it been seen)?
- is this cell flagged?
- how many mines are there in neighbor cells?

The last item is not mandatory, as it is possible to compute it when it is needed. However, it is simpler to do this computation once at the beginning of the game.

We represent a cell with a record that contains these four pieces of data.

```
# type cell = {
    mutable mined : bool ;
    mutable seen : bool ;
    mutable flag : bool ;
    mutable nbm : int
  } ;;
```

The two dimensional array is an array of arrays of cells:

```
# type board = cell array array  ;;
```

An iterator In the rest of the program, we often need to iterate a function over all the cells of the mine field. To do it generically, we define the operator `iter_cells` that applies the function `f`, given as an argument, to each cell of the board defined by the configuration `cf`.

```
# let iter_cells cf f =
    for i=0 to cf.nbcols-1 do for j=0 to cf.nbrows-1 do f (i,j) done done ;;
val iter_cells : config -> (int * int -> 'a) -> unit = <fun>
```

This is a good example of a mix between functional and imperative programming styles, as we use a higher order function (a function taking another function as an argument) to iterate a function that operates through side effects (as it returns no value).

Initialization We randomly choose which cells are mines. If c and r are respectively the number of columns and rows of the mine field, and m the number of mines, we need to generate m different numbers between 1 and $c \times r$. We suppose that $m \leq c \times r$ to define the algorithm, but the program using it will need to check this condition.

The straightforward algorithm consists of starting with an empty list, picking a random number and putting it in the list if it is not there already, and repeating this until the list contains m numbers. We use the following functions from the `Random` and `Sys` modules:

- `Random.int`: *int -> int*, picks a number between 0 and $n-1$ (n is the argument) according to a random number generator;
- `Random.init`: *int -> unit*, initializes the random number generator;

- `Sys.time`: *`unit -> float`*, returns the number of milliseconds of processor time the program used since it started. This function will be used to initialize the random number generator with a different seed for each game.

The modules containing these functions are described in more details in chapter 8, pages 216 and 234.

The random mine placement function receives the number of cells (`cr`) and the number of mines to place (`m`), and returns a list of linear positions for the `m` mines.

```
# let random_list_mines cr m =
    let cell_list = ref []
    in while (List.length !cell_list) < m do
         let n = Random.int cr in
          if not (List.mem n !cell_list) then cell_list := n :: !cell_list
       done ;
      !cell_list ;;
val random_list_mines : int -> int -> int list = <fun>
```

With such an implementation, there is no upper bound on the number of steps the function takes to terminate. If the random number generator is reliable, we can only insure that the probability it does not terminate is zero. However, all experimental uses of this function have never failed to terminate. Thus, even though it is not guaranteed that it will terminate, we will use it to generate the list of mined cells.

We need to initialize the random number generator so that each run of the game does not use the same mine field. We use the processor time since the beginning of the program execution to initialize the random number generator.

```
# let generate_seed () =
    let t = Sys.time () in
    let n = int_of_float (t*.1000.0)
    in Random.init(n mod 100000) ;;
val generate_seed : unit -> unit = <fun>
```

In practice, a given program very often takes the same execution time, which results in a similar result for `generate_seed` for each run. We ought to use the `Unix.time` function (see chapter 18).

We very often need to know the neighbors of a given cell, during the initialization of the mine field as well as during the game. Thus we write a `neighbors` function. This function must take into account the side and corner cells that have fewer neighbors than the middle ones (function `valid`).

```
# let valid cf (i,j) = i>=0 && i<cf.nbcols && j>=0 && j<cf.nbrows  ;;
val valid : config -> int * int -> bool = <fun>
# let neighbors cf (x,y) =
    let ngb = [x-1,y-1; x-1,y; x-1,y+1; x,y-1; x,y+1; x+1,y-1; x+1,y; x+1,y+1]
    in List.filter (valid cf) ngb ;;
val neighbors : config -> int * int -> (int * int) list = <fun>
```

The `initialize_board` function creates the initial mine field. It proceeds in four steps:

1. generation of the list of mined cells;
2. creation of a two dimensional array containing different cells;
3. setting of mined cells in the board;
4. computation of the number of mines in neighbor cells for each cell that is not mined.

The function `initialize_board` uses a few local functions that we briefly describe.

`cell_init` : creates an initial cell value;

`copy_cell_init` : puts a copy of the initial cell value in a cell of the board;

`set_mined` : puts a mine in a cell;

`count_mined_adj` : computes the number of mines in the neighbors of a given cell;

`set_count` : updates the number of mines in the neighbors of a cell if it is not mined.

```
# let initialize_board cf =
    let cell_init () = { mined=false; seen=false; flag=false; nbm=0 } in
    let copy_cell_init b (i,j) = b.(i).(j) <- cell_init() in
    let set_mined b n = b.(n / cf.nbrows).(n mod cf.nbrows).mined <- true
    in
    let count_mined_adj b (i,j) =
      let x = ref 0 in
      let inc_if_mined (i,j) = if b.(i).(j).mined then incr x
      in List.iter inc_if_mined (neighbors cf (i,j)) ;
         !x
    in
    let set_count b (i,j) =
      if not b.(i).(j).mined
      then b.(i).(j).nbm <- count_mined_adj b (i,j)
    in
    let list_mined = random_list_mines (cf.nbcols*cf.nbrows) cf.nbmines in
    let board = Array.make_matrix cf.nbcols cf.nbrows (cell_init ())
    in iter_cells cf (copy_cell_init board) ;
       List.iter (set_mined board) list_mined ;
       iter_cells cf (set_count board) ;
       board ;;
val initialize_board : config -> cell array array = <fun>
```

Opening a cell During a game, when the player opens a cell whose neighbors are empty (none contains a mine), he knows that he can open the neighboring cells without risk, and he can keep opening cells as long as he opens cells without any mined neighbor. In order to relieve the player of this boring process (as it is not challenging at all), our

Minesweeper opens all these cells itself. To this end, we write the function `cells_to_see` that returns a list of all the cells to open when a given cell is opened.

The algorithm needed is simple to state: if the opened cell has some neighbors that contain a mine, then the list of cells to see consists only of the opened cell; otherwise, the list of cells to see consists of the neighbors of the opened cell, as well as the lists of cells to see of these neighbors. The difficulty is in writing a program that does not loop, as every cell is a neighbor of any of its neighbors. We thus need to avoid processing the same cell twice.
To remember which cells were processed, we use the array of booleans `visited`. Its size is the same as the mine field. The value **`true`** for a cell of this array denotes that it was already visited. We recurse only on cells that were not visited.

We use the auxiliary function `relevant` that computes two sublists from the list of neighbors of a cell. Each one of these lists only contains cells that do not contain a mine, that are not opened, that are not flagged by the player, and that were not visited. The first sublist is the list of neighboring cells who have at least one neighbor containing a mine; the second sublist is the list of neighboring cells whose neighbors are all empty. As these lists are computed, all these cells are marked as visited. Notice that flagged cells are not processed, as a flag is meant to prevent opening a cell.

The local function `cells_to_see_rec` implements the recursive search loop. It takes as an argument the list of cells to visit, updates it, and returns the list of cells to open. This function is called with the list consisting only of the cell being opened, after it is marked as visited.

```
# let cells_to_see bd cf (i,j) =
    let visited = Array.make_matrix cf.nbcols cf.nbrows false in
    let rec relevant = function
        [] → ([],[])
      | ((x,y) as c) :: t →
          let cell=bd.(x).(y)
          in if cell.mined || cell.flag || cell.seen || visited.(x).(y)
             then relevant t
             else let (l1,l2) = relevant t
                  in visited.(x).(y) <- true ;
                     if cell.nbm=0 then (l1,c :: l2) else (c :: l1,l2)
    in
    let rec cells_to_see_rec = function
        [] → []
      | ((x,y) as c) :: t →
          if bd.(x).(y).nbm<>0 then c :: (cells_to_see_rec t)
          else let (l1,l2) = relevant (neighbors cf c)
               in (c :: l1) @ (cells_to_see_rec (l2 @ t))
    in visited.(i).(j) <- true ;
       cells_to_see_rec [(i,j)] ;;
val cells_to_see :
  cell array array -> config -> int * int -> (int * int) list = <fun>
```

At first sight, the argument of `cells_to_see_rec` may grow between two consecutive calls, although the recursion is based on this argument. It is legitimate to wonder if this function always terminates.
The way the `visited` array is used guarantees that a visited cell cannot be in the result of the `relevant` function. Also, all the cells to visit come from the result of the `relevant` function. As the `relevant` function marks as visited all the cells it returns, it returns each cell at most once, thus a cell may be added to the list of cells to visit at most once. The number of cells being finite, we deduce that the function terminates.

Except for graphics, we are done with our Minesweeper. Let us take a look at the programming style we have used. Mutable structures (arrays and mutable record fields) make us use an imperative style of loops and assignments. However, to deal with auxiliary issues, we use lists that are processed by functions written in a functional style. Actually, the programming style is a consequence of the data structure that it manipulates. The function `cells_to_see` is a good example: it processes lists, and it is natural to write it in a functional style. Nevertheless, we use an array to remember the cells that were already processed, and we update this array imperatively. We could use a purely functional style by using a list of visited cells instead of an array, and check if a cell is in the list to see if it was visited. However, the cost of such a choice is important (looking up an element in a list is linear in the size of the list, whereas accessing an array element takes constant time) and it does not make the program simpler.

Displaying the Minesweeper game

This part depends on the data structures representing the state of the game (see page 177). It consists of displaying the different components of the Minesweeper window, as shown in figure 6.6. To this end, we use the box drawing functions seen on page 126.

The following parameters characterize the components of the graphical window.

```
# let b0 = 3 ;;
# let w1 = 15 ;;
# let w2 = w1 ;;
# let w4 = 20 + 2*b0 ;;
# let w3 = w4*default_config.nbcols + 2*b0 ;;
# let w5 = 40 + 2*b0 ;;
```

```
# let h1 = w1 ;;
# let h2 = 30 ;;
# let h3 = w5+20 + 2*b0 ;;
# let h4 = h2 ;;
# let h5 = 20 + 2*b0 ;;
# let h6 = w5 + 2*b0 ;;
```

We use them to extend the basic configuration of our Minesweeper board (value of type *config*). Below, we define a record type *window_config*. The `cf` field contains the basic configuration. We associate a box with every component of the display: main window (field `main_box`), mine field (field `field_box`), dialog window (field `dialog_box`) with two sub-boxes (fields `d1_box` and `d2_box`), flagging button (field `flag_box`) and current cell (field `current_box`).

```
# type window_config = {
    cf : config ;
```

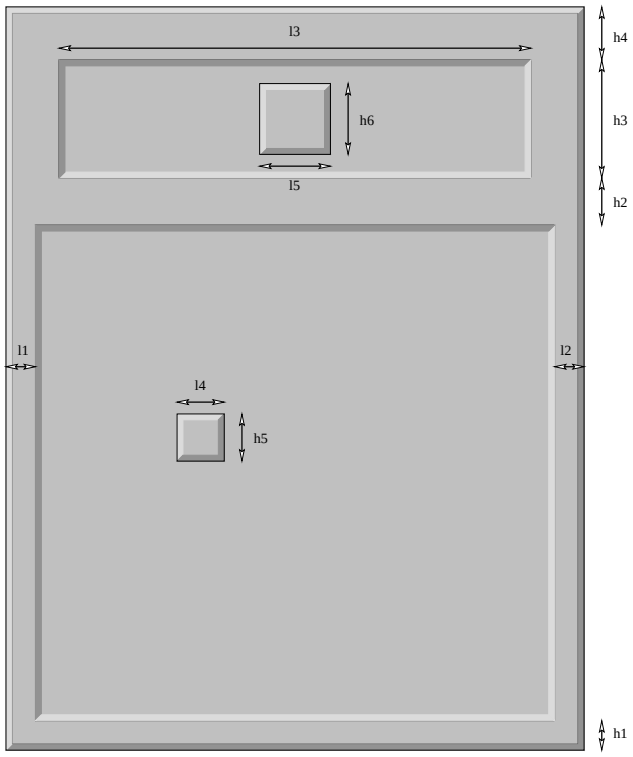


Figure 6.6: The main window of Minesweeper.

```
    main_box : box_config ;
    field_box : box_config ;
    dialog_box : box_config ;
    d1_box : box_config ;
    d2_box : box_config ;
    flag_box : box_config ;
    mutable current_box : box_config ;
    cell : int*int → (int*int) ;
    coor : int*int → (int*int)
  } ;;
```

Moreover, a record of type *window_config* contains two functions:

- `cell`: takes the coordinates of a cell and returns the coordinates of the corresponding box;
- `coor`: takes the coordinates of a pixel of the window and returns the coordinates of the corresponding cell.

Configuration We now define a function that builds a graphical configuration (of type *window_config*) according to a basic configuration (of type *config*) and the parameters above. The values of the parameters of some components depend on the value of the parameters of other components. For instance, the global box width depends on the mine field width, which, in turn, depends on the number of columns. To avoid computing the same value several times, we incrementally create the components. This initialization phase of a graphical configuration is always a little tedious when there is no adequate primitive or tool available.

```
# let make_box x y w h bw r =
    { x=x; y=y; w=w; h=h; bw=bw; r=r; b1_col=gray1; b2_col=gray3; b_col=gray2 } ;;
val make_box : int -> int -> int -> int -> int -> relief -> box_config =
  <fun>
# let make_wcf cf =
    let wcols =  b0 + cf.nbcols*w4 + b0
    and hrows =  b0 + cf.nbrows*h5 + b0  in
    let main_box =  let gw = (b0 + w1 + wcols + w2 + b0)
                    and gh = (b0 + h1 + hrows + h2 + h3 + h4 + b0)
                    in make_box 0 0 gw gh b0 Top
    and field_box = make_box w1 h1 wcols hrows b0 Bot  in
    let dialog_box = make_box  ((main_box.w - w3) / 2)
                               (b0+h1+hrows+h2)
                               w3 h3 b0 Bot
    in
    let d1_box = make_box (dialog_box.x + b0) (b0 + h1 + hrows + h2)
                          ((w3-w5)/2-(2*b0)) (h3-(2*b0)) 5 Flat  in
    let flag_box = make_box (d1_box.x + d1_box.w)
                         (d1_box.y + (h3-h6) / 2) w5 h6 b0 Top  in
    let d2_box = make_box (flag_box.x + flag_box.w)
                          d1_box.y d1_box.w d1_box.h 5 Flat  in
    let current_box = make_box 0 0 w4 h5 b0 Top
    in { cf = cf;
         main_box = main_box; field_box=field_box; dialog_box=dialog_box;
         d1_box=d1_box;
         flag_box=flag_box; d2_box=d2_box; current_box = current_box;
         cell = (fun (i,j) → ( w1+b0+w4*i , h1+b0+h5*j)) ;
         coor = (fun (x,y) → ( (x-w1)/w4 , (y-h1)/h5 )) } ;;
val make_wcf : config -> window_config = <fun>
```

Cell display We now need to write the functions to display the cells in their different states. A cell may be open or closed and may contain some information. We always display (the box corresponding with) the current cell in the game configuration (field `cc_bcf`).

We thus write two functions modifying the configuration of the current cell; one closing it, the other opening it.

```
# let close_ccell wcf i j =
```

```
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Top} ;;
val close_ccell : window_config -> int -> int -> unit = <fun>
# let open_ccell wcf i j =
    let x,y = wcf.cell (i,j)
    in wcf.current_box <- {wcf.current_box with x=x; y=y; r=Flat} ;;
val open_ccell : window_config -> int -> int -> unit = <fun>
```

Depending on the game phase, we may need to display some information on the cells. We write, for each case, a specialized function.

- Display of a closed cell:

```
# let draw_closed_cc wcf i j =
    close_ccell wcf i j;
    draw_box wcf.current_box ;;
val draw_closed_cc : window_config -> int -> int -> unit = <fun>
```

- Display of an opened cell with its number of neighbor mines:

```
# let draw_num_cc wcf i j n =
     open_ccell wcf i j ;
     draw_box wcf.current_box ;
     if n<>0 then draw_string_in_box Center (string_of_int n)
                                        wcf.current_box Graphics.white ;;
val draw_num_cc : window_config -> int -> int -> int -> unit = <fun>
```

- Display of a cell containing a mine:

```
# let draw_mine_cc wcf i j =
     open_ccell wcf i j ;
     let cc = wcf.current_box
     in draw_box wcf.current_box ;
        Graphics.set_color Graphics.black ;
        Graphics.fill_circle (cc.x+cc.w/2) (cc.y+cc.h/2) (cc.h/3) ;;
val draw_mine_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a flagged cell containing a mine:

```
# let draw_flag_cc wcf i j =
     close_ccell wcf i j ;
     draw_box wcf.current_box ;
     draw_string_in_box Center "!" wcf.current_box Graphics.blue ;;
val draw_flag_cc : window_config -> int -> int -> unit = <fun>
```

- Display of a wrongly flagged cell:

```
# let draw_cross_cc wcf i j =
    let x,y = wcf.cell (i,j)
    and w,h = wcf.current_box.w, wcf.current_box.h in
    let a=x+w/4 and b=x+3*w/4
```

```
        and c=y+h/4 and d=y+3*h/4
        in Graphics.set_color Graphics.red ;
           Graphics.set_line_width 3 ;
           Graphics.moveto a d ; Graphics.lineto b c ;
           Graphics.moveto a c ; Graphics.lineto b d ;
           Graphics.set_line_width 1 ;;
    val draw_cross_cc : window_config -> int -> int -> unit = <fun>
```

During the game, the choice of the display function to use is done by:

```
# let draw_cell wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.seen , cell.mined ) with
           (true,_,_)  → draw_flag_cc wcf i j
         | (_,false,_) → draw_closed_cc wcf i j
         | (_,_,true)  → draw_mine_cc wcf i j
         | _           → draw_num_cc wcf i j cell.nbm  ;;
val draw_cell : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

A specialized function displays all the cells at the end of the game. It is slightly different from the previous one as all the cells are taken as opened. Moreover, a red cross indicates the empty cells where the player wrongly put a flag.

```
# let draw_cell_end wcf bd i j =
    let cell = bd.(i).(j)
    in match (cell.flag, cell.mined ) with
           (true,true) → draw_flag_cc wcf i j
         | (true,false) → draw_num_cc wcf i j cell.nbm; draw_cross_cc wcf i j
         | (false,true) → draw_mine_cc wcf i j
         | (false,false) → draw_num_cc wcf i j cell.nbm ;;
val draw_cell_end : window_config -> cell array array -> int -> int -> unit =
  <fun>
```

Display of the other components The state of the flagging mode is indicated by a box that is either at the bottom or on top and that contain either the word `ON` or `OFF`:

```
# let draw_flag_switch wcf on =
    if on  then wcf.flag_box.r <- Bot  else wcf.flag_box.r <- Top ;
    draw_box wcf.flag_box ;
    if on  then draw_string_in_box Center "ON" wcf.flag_box Graphics.red
    else draw_string_in_box Center "OFF" wcf.flag_box Graphics.blue ;;
val draw_flag_switch : window_config -> bool -> unit = <fun>
```

We display the purpose of the flagging button above it:

```
# let draw_flag_title wcf =
    let m = "Flagging" in
    let w,h = Graphics.text_size m in
    let x = (wcf.main_box.w-w)/2
    and y0 = wcf.dialog_box.y+wcf.dialog_box.h in
    let y = y0+(wcf.main_box.h-(y0+h))/2
    in Graphics.moveto x y ;
       Graphics.draw_string m ;;
val draw_flag_title : window_config -> unit = <fun>
```

During the game, the number of empty cells left to be opened and the number of cells to flag are displayed in the dialog box, to the left and right of the flagging mode button.

```
# let print_score wcf nbcto nbfc =
    erase_box wcf.d1_box ;
    draw_string_in_box Center (string_of_int nbcto) wcf.d1_box Graphics.blue ;
    erase_box wcf.d2_box ;
    draw_string_in_box Center (string_of_int (wcf.cf.nbmines-nbfc)) wcf.d2_box
     ( if nbfc>wcf.cf.nbmines then Graphics.red else Graphics.blue ) ;;
val print_score : window_config -> int -> int -> unit = <fun>
```

To draw the initial mine field, we need to draw (number of rows) × (number of columns) times the same closed cell. It is always the same drawing, but it may take a long time, as it is necessary to draw a rectangle as well as four trapezoids. To speed up this initialization, we draw only one cell, take the bitmap corresponding to this drawing, and paste this bitmap into every cell.

```
# let draw_field_initial wcf =
    draw_closed_cc wcf 0 0 ;
    let cc = wcf.current_box  in
    let bitmap = draw_box cc ; Graphics.get_image cc.x cc.y cc.w cc.h  in
    let draw_bitmap (i,j) = let x,y=wcf.cell (i,j)
                            in Graphics.draw_image bitmap x y
    in iter_cells wcf.cf draw_bitmap  ;;
val draw_field_initial : window_config -> unit = <fun>
```

At the end of the game, we open the whole mine field while putting a red cross on cells wrongly flagged:

```
# let draw_field_end wcf bd =
    iter_cells wcf.cf (fun (i,j) → draw_cell_end wcf bd i j)  ;;
val draw_field_end : window_config -> cell array array -> unit = <fun>
```

Finally, the main display function called at the beginning of the game opens the graphical context and displays the initial state of all the components.

```
# let open_wcf wcf =
    Graphics.open_graph ( " " ^ (string_of_int wcf.main_box.w) ^ "x" ^
                          (string_of_int wcf.main_box.h)               ) ;
    draw_box wcf.main_box ;
    draw_box wcf.dialog_box ;
    draw_flag_switch wcf false ;
    draw_box wcf.field_box ;
    draw_field_initial wcf ;
    draw_flag_title wcf ;
    print_score wcf ((wcf.cf.nbrows*wcf.cf.nbcols)-wcf.cf.nbmines) 0 ;;
val open_wcf : window_config -> unit = <fun>
```

Notice that all the display primitives are parameterized by a graphical configuration of type *window_config*. This makes them independent of the layout of the components of our Minesweeper. If we wish to modify the layout, the code still works without any modification, only the configuration needs to be updated.

Interaction with the player

We now list what the player may do:

- he may click on the mode box to change mode (opening or flagging),
- he may click on a cell to open it or flag it,
- he may hit the `'q'` key to quit the game.

Recall that a `Graphic` event (*Graphics.event*) must be associated with a record (*Graphics.status*) that contains the current information on the mouse and keyboard when the event occurs. An interaction with the mouse may happen on the mode button, or on a cell of the mine field. Every other mouse event must be ignored. In order to differentiate these mouse events, we create the type:

```
# type clickon = Out | Cell of (int*int) | SelectBox  ;;
```

Also, pressing the mouse button and releasing it are two different events. For a click to be valid, we require that both events occur on the same component (the flagging mode button or a cell of the mine field).

```
# let locate_click wcf st1 st2 =
    let clickon_of st =
      let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y
      in if x>=wcf.flag_box.x && x<=wcf.flag_box.x+wcf.flag_box.w &&
            y>=wcf.flag_box.y && y<=wcf.flag_box.y+wcf.flag_box.h
         then SelectBox
         else let (x2,y2) = wcf.coor (x,y)
              in if x2>=0 && x2<wcf.cf.nbcols && y2>=0 && y2<wcf.cf.nbrows
                 then Cell (x2,y2) else Out
    in
```

```
    let r1=clickon_of st1 and r2=clickon_of st2
    in if r1=r2 then r1 else Out ;;
val locate_click :
  window_config -> Graphics.status -> Graphics.status -> clickon = <fun>
```

The heart of the program is the event waiting and processing loop defined in the function `loop`. It is similar to the function `skel` described page 133, but specifies the mouse events more precisely. The loop ends when:

- the player presses the `q` or `Q` key, meaning that he wants to end the game;
- the player opens a cell containing a mine, then he loses;
- the player has opened all the cell that are empty, then he wins the game.

We gather in a record of type *minesw_cf* the information useful for the interface:

```
# type minesw_cf =
    { wcf : window_config; bd : cell array array;
      mutable nb_flagged_cells : int;
      mutable nb_hidden_cells : int;
      mutable flag_switch_on : bool } ;;
```

The meaning of the fields is:

- `wcf`: the graphical configuration;
- `bd`: the board;
- `flag_switch_on`: a boolean indicating whether flagging mode or opening mode is on;
- `nb_flagged_cells`: the number of flagged cells;
- `nb_hidden_cells`: the number of empty cells left to open;

The main loop is implemented this way:

```
# let loop d f_init f_key f_mouse f_end =
    f_init ();
    try
      while true do
        let st = Graphics.wait_next_event
                   [Graphics.Button_down;Graphics.Key_pressed]
        in if st.Graphics.keypressed  then f_key st.Graphics.key
           else let st2 = Graphics.wait_next_event [Graphics.Button_up]
                in f_mouse (locate_click d.wcf st st2)
      done
    with End → f_end ();;
val loop :
  minesw_cf ->
  (unit -> 'a) -> (char -> 'b) -> (clickon -> 'b) -> (unit -> unit) -> unit =
  <fun>
```

The initialization function, cleanup function and keyboard event processing function are very simple.

```
# let d_init d () = open_wcf d.wcf
  let d_end  () = Graphics.close_graph()
  let d_key c = if c='q' || c='Q' then raise End;;
val d_init : minesw_cf -> unit -> unit = <fun>
val d_end : unit -> unit = <fun>
val d_key : char -> unit = <fun>
```

However, the mouse event processing function requires the use of some auxiliary functions:

- `flag_cell`: when clicking on a cell with flagging mode on.
- `ending`: when ending the game. The whole mine field is revealed, we display a message indicating whether the game was won or lost, and we wait for a mouse or keyboard event to quit the application.
- `reveal`: when clicking on a cell with opening mode on (*i.e.* flagging mode off).

```
# let flag_cell d i j =
    if d.bd.(i).(j).flag
    then ( d.nb_flagged_cells <- d.nb_flagged_cells -1;
           d.bd.(i).(j).flag <- false )
    else ( d.nb_flagged_cells <- d.nb_flagged_cells +1;
           d.bd.(i).(j).flag <- true );
    draw_cell d.wcf d.bd i j;
    print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;;
val flag_cell : minesw_cf -> int -> int -> unit = <fun>

# let ending d str =
    draw_field_end d.wcf d.bd;
    erase_box  d.wcf.flag_box;
    draw_string_in_box Center str d.wcf.flag_box Graphics.black;
    ignore(Graphics.wait_next_event
             [Graphics.Button_down;Graphics.Key_pressed]);
    raise End;;
val ending : minesw_cf -> string -> 'a = <fun>

# let reveal d i j =
    let reveal_cell (i,j) =
      d.bd.(i).(j).seen <- true;
      draw_cell d.wcf d.bd i j;
      d.nb_hidden_cells <- d.nb_hidden_cells -1
    in
      List.iter reveal_cell (cells_to_see d.bd d.wcf.cf (i,j));
      print_score d.wcf d.nb_hidden_cells d.nb_flagged_cells;
      if d.nb_hidden_cells = 0 then ending d "WON";;
```

```
val reveal : minesw_cf -> int -> int -> unit = <fun>
```

The mouse event processing function matches a value of type *clickon*.

```
# let d_mouse d click = match click with
      Cell (i,j) →
        if d.bd.(i).(j).seen then ()
        else if d.flag_switch_on then flag_cell d i j
        else if d.bd.(i).(j).flag then ()
        else if d.bd.(i).(j).mined then ending d "LOST"
        else reveal d i j
    | SelectBox →
        d.flag_switch_on <- not d.flag_switch_on;
        draw_flag_switch d.wcf d.flag_switch_on
    | Out → () ;;
val d_mouse : minesw_cf -> clickon -> unit = <fun>
```

To create a game configuration, three parameters are needed: the number of columns, the number of rows, and the number of mines.

```
# let create_minesw nb_c nb_r nb_m =
    let nbc = max default_config.nbcols nb_c
    and nbr = max default_config.nbrows nb_r in
    let nbm = min (nbc*nbr) (max 1 nb_m) in
    let cf = { nbcols=nbc ; nbrows=nbr ; nbmines=nbm } in
    generate_seed () ;
    let wcf = make_wcf cf in
    {  wcf = wcf ;
       bd = initialize_board wcf.cf;
       nb_flagged_cells = 0;
       nb_hidden_cells = cf.nbrows*cf.nbcols-cf.nbmines;
       flag_switch_on = false } ;;
val create_minesw : int -> int -> int -> minesw_cf = <fun>
```

The launch function creates a configuration according to the numbers of columns, rows, and mines, before calling the main event processing loop.

```
# let go nbc nbr nbm =
    let d = create_minesw nbc nbr nbm in
    loop d (d_init d) d_key (d_mouse d) (d_end);;
val go : int -> int -> int -> unit = <fun>
```

The function call `go 10 10 10` builds and starts a game of the same size as the one depicted in figure 6.5.

Exercises

This program can be built as a standalone executable program. Chapter 7 explains how to do this. Once it is done, it is useful to be able to specify the size of the game on the command line. Chapter 8 describes how to get command line arguments in an Objective Caml program, and applies it to our minesweeper (see page 236).

Another possible extension is to have the machine play to discover the mines. To do this, one needs to be able to find the safe moves and play them first, then compute the probabilities of presence of a mine and open the cell with the smallest probability.

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 `ocamlex` 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 AWT[2]; and a classic program to find least-cost paths in a graph using our new graphical interface as well as a cache memory mechanism.

2. *Abstract Windowing Toolkit*

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.

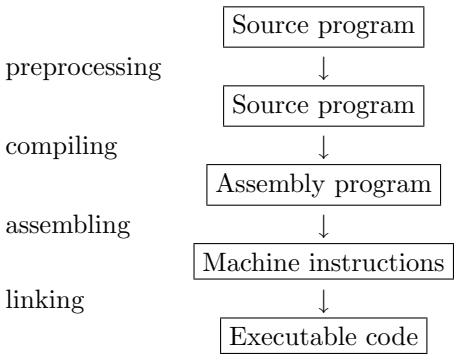


Figure 7.1: Steps in the production of an executable.

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-

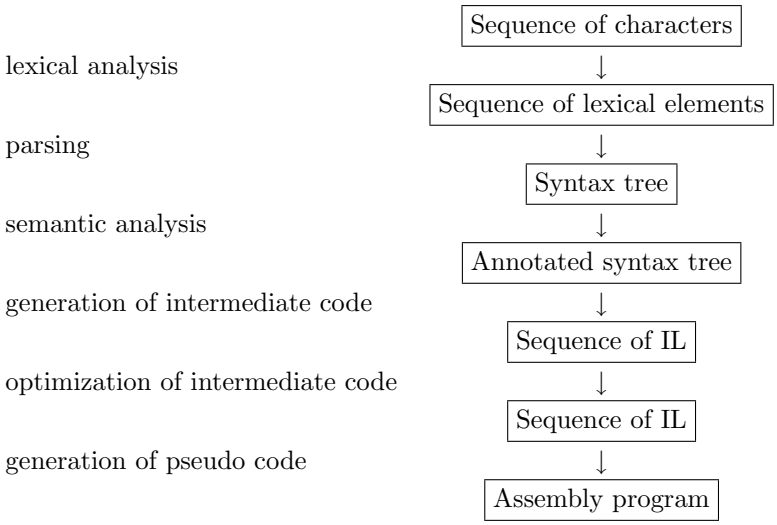


Figure 7.2: Compilation stages.

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:

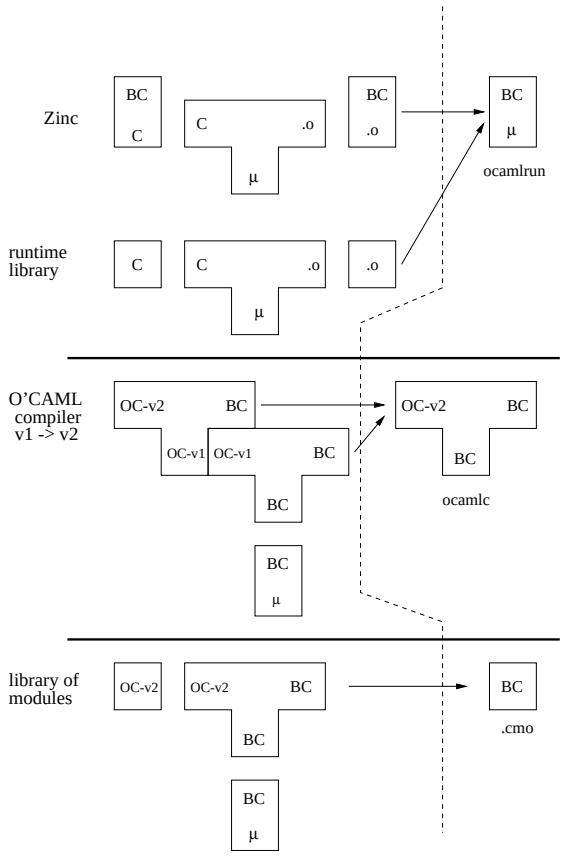


Figure 7.3: Virtual machine.

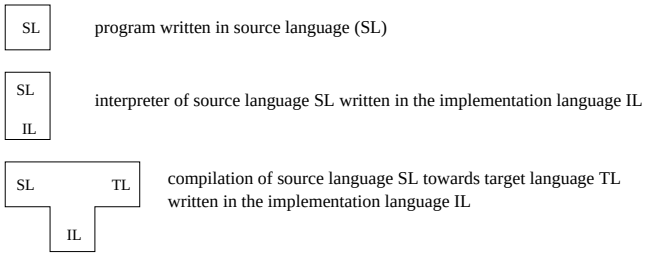


Figure 7.4: Graphical notation for interpreters and compilers.

- 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.

`ocaml`	toplevel loop
`ocamlrun`	bytecode interpreter
`ocamlc`	bytecode batch compiler
`ocamlopt`	native code batch compiler
`ocamlc.opt`	optimized bytecode batch compiler
`ocamlopt.opt`	optimized native code batch compiler
`ocamlmktop`	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
`.ml`	source file
`.mli`	interface file
`.cmo`	object file (bytecode)
`.cma`	library object file (bytecode)
`.cmi`	compiled interface file
`.cmx`	object file (native)
`.cmxa`	library object file (native)
`.c`	C source file
`.o`	C object file (native)
`.a`	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	
`-a`	construct a runtime library
`-c`	compile without linking
`-o` *name_of_executable*	specify the name of the executable
`-linkall`	link with all libraries used
`-i`	display all compiled global declarations
`-pp` *command*	uses *command* as preprocessor
`-unsafe`	turn off index checking
`-v`	display the version of the compiler
`-w` *list*	choose among the *list* the level of warning message (see fig. 7.9)
`-impl` *file*	indicate that *file* is a Caml source (.ml)
`-intf` *file*	indicate that *file* is a Caml interface (.mli)
`-I` *directory*	add *directory* in the list of directories

Figure 7.7: Principal options of the bytecode compiler.

Other options	
light process	`-thread` (see chapter 19, page 599)
linking	`-g`, `-noassert` (see chapter 10, page 271)
standalone executable	`-custom`, `-cclib`, `-ccopt`, `-cc` (see page 207)
runtime	`-make-runtime` , `-use-runtime`
C interface	`-output-obj` (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.

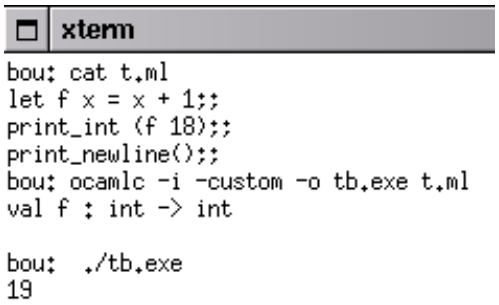


Figure 7.10: Session with the bytecode compiler.

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.

`-compact`	optimize the produced code for space
`-S`	keeps the assembly code in a file
`-inline` *level*	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 `;;`.

`#quit ;;`	quit from the toplevel interaction
`#directory` *directory* `;;` `#cd` *directory* `;;`	add the directory to the search path change the working directory
`#load` *object_file* `;;` `#use` *source_file* `;;`	load an object file (`.cmo`) compile and load a source file
`#print_depth` *depth* `;;` `#print_length` *width* `;;`	modify the depth of printing modify the length of printing
`#install_printer` *function* `;;` `#remove_printer` *function* `;;`	specify a printing function remove a printing function

`#trace` *function* `;;`
`#untrace` *function* `;;`
`#untrace_all ;;`
	trace the arguments of the function
	stop tracing the function
	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.

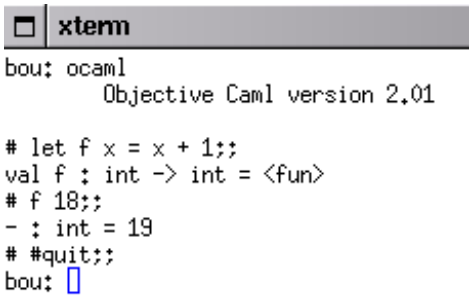


Figure 7.13: Session with the toplevel loop.

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*, `-ccopt` *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 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
$ ocamlopt -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:

`fib.exe` (bytecode)	`fibopt.exe` (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 (`alexsynt.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 it's 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.

8

Libraries

Every language comes with collections of programs that are reusable by the programmer, called *libraries*. The quality and diversity of these programs are often some of the criteria one uses to assess the ease of use of a language. You could separate libraries into two categories: those that offer types and functions that are often useful but could be written in the language, and those that offer functionality that cannot be defined in the language. The first group saves the programmer the effort of redefining utilities such as stacks, lists, etc. The second group extends the possible uses of the language by incorporating new functionality into it.

The Objective Caml language distribution comes with many precompiled libraries. For the curious reader, the uncompiled version of these libraries comes packaged with the source code distribution for the language.

In Objective Caml, all the libraries are organized into *modules* that are also compilation units. Each one contains declarations of globals and types, exceptions and values that can be used in programs. In this chapter we are not interested in how to create new modules; we just want to use the existing ones. Chapter 14 will revisit the concepts of the module and the compilation unit while describing the module language of Objective Caml, including parameterized modules. Regarding the creation of libraries that incorporate code that is not written in Objective Caml, chapter 12 will describe how to integrate Objective Caml programs with code written in C.

The Objective Caml distribution contains a preloaded library (the `Pervasives` module), a collection of basic modules called the *standard library*, and many other libraries adding functionality to the language. Some of the libraries are briefly shown in this chapter while others are described in later chapters.

Chapter Outline

This chapter describes the collection of libraries in the Objective Caml distribution. Some have been used in previous chapters, such as the `Graphics` library (see chapter 5), or the `Array` library. The first section shows the organization of the various libraries. The second section finishes describing the preloaded `Pervasives` module. The third section classifies the set of modules found in the standard library. The fourth section examines the high precision math libraries and the libraries for dynamically loading code.

Categorization and Use of the Libraries

The libraries in the Objective Caml distribution fall into three categories. The first contains preloaded global declarations. The second is called the standard library and is subdivided into four parts:

- data structures;
- input/output
- system interface;
- lexical and syntactic analysis.

Finally there are the libraries in the third group that generally extend the language, such as the `Graphics` library (see chapter 5). In this last group you will find libraries dealing with the following areas: regular expressions (`Str`), arbitrary-precision math (`Num`), Unix system calls (`Unix`), lightweight processes (`Threads`) and dynamic loading of bytecode (`Dynlink`).

The I/O and the system interface portions of the standard library are compatible with different operating systems such as Unix, Windows and MacOS. This is not always the case with the libraries in the third group (those that extend the language). There are also many independently written libraries that are not part of the Objective Caml distribution.

Usage and naming To use modules or libraries in a program, one has to use dot notation to specify the module name and the object to access. For example if one wants to use a function `f` in a library called `Name`, one qualifies it as `Name.f`. To avoid having to prefix everything with the name of the library, it is possible to open the library and use `f` directly.

Syntax : `open Name`

From then on, all the global declarations of the library `Name` will be considered as if they belonged to the global environment. If two declarations have the same name in two distinct open libraries, then only the last declaration is visible. To be able to call the first, it would be necessary to use the point notation.

Preloaded Library

The `Pervasives` library is always preloaded so that it will be available at the toplevel (interactive) loop or for inline compilation. It is always linked and is the initial environment of the language. It contains the declarations of:

- **type**: basic types (*int*, *char*, *string*, *float*, *bool*, *unit*, *exn*, *'a array*, *'a list*) and the types *'a option* (see page 223) and *('a, 'b, 'c) format* (see page 265).
- **exceptions**: A number of exceptions are raisable by the execution library. Some of the more common ones are the following:
  - *Failure* **of** *string* that is raised by the function `failwith` applied to a string.
  - *Invalid_argument* **of** *string* that indicates that an argument cannot be handled by the function having raised the exception. The function `invalid_arg` applied to a string starts this exception.
  - *Sys_error* **of** *string*, for the input/output, typically in attempting to open a nonexistent file for reading.
  - *End_of_file* for detecting the end of a file.
  - *Division_by_zero* for zero divide errors between integers.

  As well as internal exceptions like:
  - *Out_of_memory* and *Stack_overflow* for going beyond the memory of the heap or the stack. It should be noted that a program cannot recover from the `Out_of_memory` exception. In effect, when it is raised it is too late to allocate new memory space to continue functioning.
    Handling the `Stack_Overflow` exception differs depending on whether the program was compiled in byte code or native code. In the latter case, it is not possible to recover.
- **functions**: there are roughly 140, half of which correspond to the C functions of the execution library. There you may find mathematical and comparison operators, functions on integer and floating-point numbers, functions on character strings, on references and input-output. It should be noted that a certain number of these declarations are in fact synonyms for declarations defined in other modules. They are nevertheless declared here for historical and implementation reasons.

Standard Library

The standard library contains a group of stable modules. These are operating system independent. There are currently 29 modules in the standard library containing 400 functions, 30 types of which half are abstract, 8 exceptions, 10 sub-modules, and 3 parameterized modules. Clearly we will not describe all of the declarations in all of these modules. Indeed, the reference manual [LRVD99] already does that quite well. Only those modules presenting a new concept or a real difficulty in use will be detailed.

The standard library can be divided into four distinct parts:

- **linear data structures** (15 modules), some of which have already appeared in the first part;
- **input-output** (4 modules), for the formatting of output, the persistence and creation of cryptographic keys;
- **parsing and lexical analysis** (4 modules). They are described in chapter 11 (page 287);
- **system interface** that permit communication and examination of parameters passed to a command, directory navigation and file access.

To these four groups we add a fifth containing some utilities for handling or creating structures such as functions for text processing or generating pseudo-random numbers, etc.

Utilities

The modules that we have named "utilities" concern:

- characters: the `Char` module primarily contains conversion functions;
- object cloning: `OO` will be presented in chapter 15 (page 435), on object oriented programming
- lazy evaluation: `Lazy` is first presented on page 107;
- random number generator: `Random` will be described below.

Generation of Random Numbers

The `Random` module is a pseudo-random number generator. It establishes a random number generation function starting with a number or a list of numbers called a *seed*. In order to ensure that the function does not always return the same list of numbers, the programmer must give it a different seed each time the generator is initialized.

From this seed the function generates a succession of seemingly random numbers. Nevertheless, an initialization with the same seed will create the same list. To correctly initialize the generator, you need to find some outside resource, like the date represented in milliseconds, or the length of time since the start of the program.

The functions of the module:

- initialization: `init` of type *`int -> unit`* and `full_init` of type *`int array -> unit`* initialize the generator. The second function takes an array of seeds.
- generate random numbers: `bits` of type *`unit -> int`* returns a positive integer, `int` of type *`int -> int`* returns a positive integer ranging from 0 to a limit given as a parameter, and `float` returns a float between 0. and a limit given as a parameter.

Linear Data Structures

The modules for linear data structures are:

- simple modules: `Array`, `String`, `List`, `Sort`, `Stack`, `Queue`, `Buffer`, `Hashtbl` (that is also parameterized) and `Weak`;
- parameterized modules: `Hashtbl` (of `HashedType` parameters), `Map` and `Set` (of `OrderedType` parameters).

The parameterized modules are built from the other modules, thus making them more generic. The construction of parameterized modules will be presented in chapter 14, page 418.

Simple Linear Data Structures

The name of the module describes the type of data structures manipulated by the module. If the type is abstract, that is to say, if the representation is hidden, the current convention is to name it `t` inside the module. These modules establish the following structures:

- module `Array`: vectors;
- module `List`: lists;
- module `String`: character strings;
- module `Hashtbl`: hash tables (abstract type);
- module `Buffer`: extensible character strings (abstract type);
- module `Stack`: stacks (abstract type);
- module `Queue`: queues or *FIFO* (abstract type);
- module `Weak`: vector of weak pointers (abstract type).

Let us mention one last module that implements linear data structures:

- module `Sort`: sorting on lists and vectors, merging of lists.

Family of common functions Each of these modules (with the exception of `Sort`), has functions for defining structures, creating/accessing elements (such as handler functions), and converting to other types. Only the `List` module is not physically modifiable. We will not give a complete description of all these functions. Instead, we will focus on families of functions that one finds in these modules. Then we will detail the `List` and `Array` modules that are the most commonly used structures in functional and imperative programming.

One finds more or less the following functionality in all these modules:

- a `length` function that takes the value of a type and calculates an integer corresponding to its length;
- a `clear` function that empties the linear structure, if it is modifiable;

- a function to add an element, `add` in general, but sometimes named differently according to common practice, (for example, `push` for stacks);
- a function to access the n-th element, often called `get`;
- a function to remove an element (often the first) `remove` or `take`.

In the same way, in several modules the names of functions for traversal and processing are the same:

- `map`: applies a function on all the elements of the structure and returns a new structure containing the results of these calls;
- `iter`: like `map`, but drops successive results, and returns `()`.

For the structures with indexed elements we have:

- `fill`: replaces (modifies in place) a part of the structure with a value;
- `blit`: copies a part of one structure into another structure of the same type;
- `sub`: copies a part of one structure into a newly created structure.

Modules `List` *and* `Array`

We describe the functions of the two libraries while placing an emphasis on the similarities and the particularities of each one. For the functions common to both modules, *`t`* designates either the *`'a list`* or *`'a array`* type. When a function belongs to one module, we will use the dot notation.

Common or analogous functionality The first of them is the calculation of length.

```
List.length  :  'a t -> int
```

Two functions permitting the concatenation of two structures or all the structures of a list.

```
List.append  :  'a t -> 'a t -> 'a t
List.concat  :  'a t list -> 'a t
```

Both modules have a function to access an element designated by its position in the structure.

```
List.nth    :  'a list -> int -> 'a
Array.get   :  'a array -> int -> 'a
```

The function to access an element at index `i` of a vector `t`, which is frequently used, has a syntactic shorthand: `t.(i)`.

Two functions allow you to apply an operation to all the elements of a structure.

```
iter  :  ('a -> unit) -> 'a t -> unit
map   :  ('a -> 'b) -> 'a t -> 'b t
```

You can use `iter` to print the contents of a list or a vector.

```
# let print_content iter print_item xs =
    iter (fun x → print_string"("; print_item x; print_string")") xs;
    print_newline() ;;
val print_content : (('a -> unit) -> 'b -> 'c) -> ('a -> 'd) -> 'b -> unit =
  <fun>
# print_content List.iter print_int [1;2;3;4;5] ;;
(1)(2)(3)(4)(5)
- : unit = ()
# print_content Array.iter print_int [|1;2;3;4;5|] ;;
(1)(2)(3)(4)(5)
- : unit = ()
```

The `map` function builds a new structure containing the result of the application. For example, with vectors whose contents are modifiable:

```
# let a = [|1;2;3;4|] ;;
val a : int array = [|1; 2; 3; 4|]
# let b = Array.map succ a ;;
val b : int array = [|2; 3; 4; 5|]
# a, b;;
- : int array * int array = [|1; 2; 3; 4|], [|2; 3; 4; 5|]
```

Two iterators can be used to compose successive applications of a function on all elements of a structure.

```
fold_left   :  ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a
fold_right  :  ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

You have to give these iterators a base case that supplies a default value when the structure is empty.

```
fold_left f r [v1; v2; ...;  vn]   =  f ... ( f  (f r v1) v2 ) ... vn
fold_right f [v1; v2; ...; vn] r   =  f v1 ( f v2   ... (f vn r) ... )
```

These functions allow you to easily transform binary operations into n-ary operations. When the operation is commutative and associative, left and right iteration are indistinguishable:

```
# List.fold_left (+) 0 [1;2;3;4] ;;
- : int = 10
# List.fold_right (+) [1;2;3;4] 0 ;;
```

```
- : int = 10
# List.fold_left List.append [0] [[1];[2];[3];[4]] ;;
- : int list = [0; 1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [0] ;;
- : int list = [1; 2; 3; 4; 0]
```

Notice that, for binary concatenation, an empty list is a neutral element to the left and to the right. We find thus, in this specific case, the equivalence of the two expressions:

```
# List.fold_left List.append [] [[1];[2];[3];[4]] ;;
- : int list = [1; 2; 3; 4]
# List.fold_right List.append [[1];[2];[3];[4]] [] ;;
- : int list = [1; 2; 3; 4]
```

We have, in fact, found the `List.concat` function.

Operations specific to lists. It is useful to have the following list functions that are provided by the `List` module:

`List.hd`	:	*'a list -> 'a* first element of the list
`List.tl`	:	*'a list -> 'a* the list, without its first element
`List.rev`	:	*'a list -> 'a list* reversal of a list
`List.mem`	:	*'a -> 'a list -> bool* membership test
`List.flatten`	:	*'a list list -> 'a list* flattens a list of lists
`List.rev_append`	:	*'a list -> 'a list -> 'a list* is the same as `append (rev l1) l2`

The first two functions are partial. They are not defined on the empty list and raise a `Failure` exception. There is a variant of `mem`: `memq` that uses physical equality.

```
# let c = (1,2) ;;
val c : int * int = 1, 2
# let l = [c] ;;
val l : (int * int) list = [1, 2]
# List.memq (1,2) l ;;
- : bool = false
# List.memq c l ;;
- : bool = true
```

The `List` module provides two iterators that generalize boolean conjunction and disjunction (and / or): `List.for_all` and `List.exists` that are defined by iteration:

```
# let for_all f xs = List.fold_right (fun x → fun b → (f x) & b) xs true ;;
val for_all : ('a -> bool) -> 'a list -> bool = <fun>
# let exists f xs = List.fold_right (fun x → fun b → (f x) or b) xs false ;;
val exists : ('a -> bool) -> 'a list -> bool = <fun>
```

There are variants of the iterators in the `List` module that take two lists as arguments and traverse them in parallel (`iter2`, `map2`, etc.). If they are not the same size, the `Invalid_argument` exception is raised.

The elements of a list can be searched using the criteria provided by the following boolean functions:

`List.find`	:	*`('a -> bool) -> 'a list -> 'a`*
`List.find_all`	:	*`('a -> bool) -> 'a list -> 'a list`*

The `find_all` function has an alias: `filter`.

A variant of the general search function is the partitioning of a list:

`List.partition`	:	*`('a -> bool) -> 'a list -> 'a list * 'a list`*

The `List` module has two often necessary utility functions permitting the division and creation of lists of pairs:

`List.split`	:	*`('a * 'b) list -> 'a list * 'b list`*
`List.combine`	:	*`'a list -> 'b list -> ('a * 'b) list`*

Finally, a structure combining lists and pairs is often used: *association lists*. They are useful to store values associated to keys. These are lists of pairs such that the first entry is a key and the second is the information associated to the key. One has these data structures to deal with pairs:

`List.assoc`	:	*`'a -> ('a * 'b) list -> 'b`* extract the information associated to a key
`List.mem_assoc`	:	*`'a -> ('a * 'b) list -> bool`* test the existence of a key
`List.remove_assoc`	:	*`'a -> ('a * 'b) list -> ('a * 'b) list`* deletion of an element corresponding to a key

Each of these functions has a variant using physical equality instead of structural equality: `List.assq`, `List.mem_assq` and `List.remove_assq`.

Handlers specific to Vectors. The vectors that imperative programmers often use are physically modifiable structures. The `Array` module furnishes a function to change the value of an element:

```
Array.set  :  'a array -> int -> 'a -> unit
```

Like `get`, the `set` function has a syntactic shortcut: `t.(i) <- a`.

There are three vector allocation functions:

`Array.create`	:	*`int -> 'a -> 'a array`* creates a vector of a given size whose elements are all initialized with the same value
`Array.make`	:	*`int -> 'a -> 'a array`* alias for `create`
`Array.init`	:	*`int -> (int -> 'a) -> 'a array`* creates a vector of a given size whose elements are each initialized with the result of the application of a function to the element's index

Since they are frequently used, the `Array` module has two functions for the creation of matrices (vectors of vectors):

```
Array.create_matrix  :  int -> int -> 'a -> 'a array array
Array.make_matrix    :  int -> int -> 'a -> 'a array array
```

The `set` function is generalized as a function modifying the values on an interval described by a starting index and a length:

```
Array.fill  :  'a array -> int -> int -> 'a -> unit
```

One can copy a whole vector or extract a sub-vector (described by a starting index and a length) to obtain a new structure:

```
Array.copy  :  'a array -> 'a array
Array.sub   :  'a array -> int -> int -> 'a array
```

The copy or extraction can also be done towards another vector:

```
Array.blit  :  'a array -> int -> 'a array -> int -> int -> unit
```

The first argument is the index into the first vector, the second is the index into the second vector and the third is the number of values copied. The three functions `blit`, `sub` and `fill` raise the `Invalid_argument` exception.

The privileged use of indices in the vector manipulation functions leads to the definition of two specific iterators:

```
Array.iteri  :  (int -> 'a -> unit) -> 'a array -> unit
Array.mapi   :  (int -> 'a -> 'b) -> 'a array -> 'b array
```

They apply a function whose first argument is the index of the affected element.

```
#   let f i a = (string_of_int i) ^ ":" ^ (string_of_int a) in
    Array.mapi f  [| 4; 3; 2; 1; 0 |] ;;
- : string array = [|"0:4"; "1:3"; "2:2"; "3:1"; "4:0"|]
```

Although the `Array` module does not have a function to modify the contents of all the elements in a vector, this effect can be easily obtained using `iteri`:

```
# let iter_and_set f t =
   Array.iteri (fun i → fun x → t.(i) <- f x) t ;;
val iter_and_set : ('a -> 'a) -> 'a array -> unit = <fun>
# let v = [|0;1;2;3;4|] ;;
val v : int array = [|0; 1; 2; 3; 4|]
# iter_and_set succ v ;;
- : unit = ()
# v ;;
- : int array = [|1; 2; 3; 4; 5|]
```

Finally, the `Array` module provides two list conversion functions:

```
Array.of_list  :  'a list -> 'a array
Array.to_list  :  'a array -> 'a list
```

Input-output

The standard library has four input-output modules:

- module `Printf`: for the formatting of output;
- `Format`: pretty-printing facility to format text within "pretty-printing boxes". The pretty-printer breaks lines at specified break hints, and indents lines according to the box structure.
- module `Marshal`: implements a mechanism for persistent values;
- module `Digest`: for creating unique keys.

The description of the `Marshal` module will be given later in the chapter when we begin to discuss persistent data structures (see page 228).

Module `Printf`

The `Printf` module formats text using the rules of the `printf` function in the C language library. The display format is represented as a character string that will be

decoded according to the conventions of `printf` in C, that is to say, by specializing the `%` character. This character followed by a letter indicates the type of the argument at this position. The following format `"(x=%d, y=%d)"` indicates that it should put two integers in place of the `%d` in the output string.

Specification of formats. A format defines the parameters for a printed string. Those, of basic types: *int*, *float*, *char* and *string*, will be converted to strings and will replace their occurrence in the printed string. The values 77 and 43 provided to the format `"(x=%d, y=%d)"` will generate the complete printed string `"(x=77, y=43)"`. The principal letters indicating the type of conversion to carry out are given in figure 8.1.

Type	Letter	Result
integer	`d` or `i`	signed decimal
	`u`	unsigned decimal
	`x`	unsigned hexadecimal, lower case form
	`X`	same, with upper case letters
character	`c`	character
string	`s`	string
float	`f`	decimal
	`e` or `E`	scientific notation
	`g` or `G`	same
boolean	`b`	`true` or `false`
special	`a` or `t`	functional parameter
		of type *(out_channel -> 'a -> unit) -> 'a -> unit*
		or *out_channel -> unit*

Figure 8.1: Conversion conventions.

The format also allows one to specify the justification of the conversion, which allows for the alignment of the printed values. One can indicate the size in conversion characters. For this one places between the `%` character and the type of conversion an integer number as in `%10d` that indicates a conversion to be padded on the right to ten characters. If the size of the result of the conversion exceeds this limit, the limit will be discarded. A negative number indicates left justification. For conversions of floating point numbers, it is helpful to be able to specify the printed precision. One places a decimal point followed by a number to indicate the number of characters after the decimal point as in `%.5f` that indicates five characters to the right of the decimal point.

There are two specific format letters: `a` and `t` that indicate a functional argument. Typically, a print function defined by the user. This is specific to Objective Caml.

Functions in the module The types of the five functions in this module are given in figure 8.2.

```
fprintf  :  out_channel -> ('a, out_channel, unit) format -> 'a
printf   :   ('a, out_channel, unit) format -> 'a
eprintf  :  ('a, out_channel, unit) format -> 'a
sprintf  :  ('a, unit, string) format -> 'a
bprintf  :  Buffer.t -> ('a, Buffer.t, string) format -> 'a
```

Figure 8.2: `Printf` formatting functions.

The `fprintf` function takes a channel, a format and arguments of types described in the format. The `printf` and `eprintf` functions are specializations on standard output and standard error. Finally, `sprintf` and `bprintf` do not print the result of the conversion, but instead return the corresponding string.

Here are some simple examples of the utilization of formats.

```
# Printf.printf "(x=%d, y=%d)" 34 78 ;;
(x=34, y=78)- : unit = ()
# Printf.printf "name = %s, age = %d" "Patricia" 18 ;;
name = Patricia, age = 18- : unit = ()
# let s = Printf.sprintf "%10.5f\n%10.5f\n" (-.12.24) (2.30000008) ;;
val s : string = " -12.24000\n   2.30000\n"
# print_string s ;;
-12.24000
   2.30000
- : unit = ()
```

The following example builds a print function from a matrix of floats using a given format.

```
# let print_mat m =
    Printf.printf "\n" ;
    for i=0 to (Array.length m)-1 do
      for j=0 to (Array.length m.(0))-1 do
        Printf.printf "%10.3f" m.(i).(j)
      done ;
      Printf.printf "\n"
    done ;;
val print_mat : float array array -> unit = <fun>
# print_mat (Array.create 4  [| 1.2; -.44.22; 35.2 |]) ;;

     1.200   -44.220    35.200
     1.200   -44.220    35.200
     1.200   -44.220    35.200
     1.200   -44.220    35.200
- : unit = ()
```

Note on the *format* type. The description of a format adopts the syntax of character strings, but it is not a value of type *string*. The decoding of a format, according to the preceding conventions, builds a value of type *format* where the *'a* parameter is instantiated either with *unit* if the format does not mention a parameter, or by a functional type corresponding to a function able to receive as many arguments as are mentioned and returning a value of type *unit*.

One can illustrate this process by partially applying the `printf` function to a format:

```
# let p3 =
    Printf.printf "begin\n%d is val1\n%s is val2\n%f is val3\n" ;;
begin
val p3 : int -> string -> float -> unit = <fun>
```

One obtains thus a function that takes three arguments. Note that the word `begin` had already been printed. Another format would have given another type of function:

```
# let p2 =
    Printf.printf "begin\n%f is val1\n%s is val2\n";;
begin
val p2 : float -> string -> unit = <fun>
```

In providing arguments one by one to `p3`, one progressively obtains the output.

```
# let p31 = p3 45 ;;
45 is val1
val p31 : string -> float -> unit = <fun>
# let p32 = p31 "hello" ;;
hello is val2
val p32 : float -> unit = <fun>
# let p33 = p32 3.14 ;;
3.140000 is val3
val p33 : unit = ()
# p33 ;;
- : unit = ()
```

From the last obtained value, nothing is printed: it is the value `()` of type *unit*.

One cannot build a format using values of type *string*:

```
# let f d =
   Printf.printf (d^d);;
Characters 27-30:
This expression has type string but is here used with type
  ('a, out_channel, unit) format
```

The compiler cannot know the value of the string passed as an argument. It thus cannot know the type that instantiates the *'a* parameter of type `format`.

On the other hand, strings are physically modifiable values, it would thus be possible to replace, for example, the `%d` part with another letter, thus dynamically changing the print format. This conflicts with the static generation of the conversion function.

`Digest` *Module*

A hash function converts a character string of unspecified size into a character string of fixed length, most often smaller. Hashing functions return a *fingerprint* (*digest*) of their entry.

Such functions are used for the construction of hash tables, as in the `Hashtbl` module, permitting one to rapidly test if an element is a member of such a table by directly accessing the fingerprint. For example the function `f_mod_n`, that generates the modulo n sum of the ASCII codes of the characters in a string, is a hashing function. If one creates an n by n table to arrange the strings, from the fingerprint one obtains direct access. Nevertheless two strings can return the same fingerprint. In the case of collisions, one adds to the hash table an extension to store these elements. If there are too many collisions, then access to the hash table is not very effective. If the fingerprint has a length of n bits, then the probability of collision between two different strings is $1/2^n$.

A *non-reversible* hash function has a very weak probability of collision. It is thus difficult, given a fingerprint, to construct a string with this fingerprint. The preceding function `f_mod_n` is not, based on the evidence, such a function. One way hash functions permit the authentification of a string, that it is for some text sent over the Internet, a file, etc.

The `Digest` module uses the *MD5* algorithm, short for *Message Digest 5*. It returns a 128 bit fingerprint. Although the algorithm is public, it is impossible (today) to carry out a reconstruction from a fingerprint. This module defines the ***`Digest.t`*** type as an abbreviation of the ***`string`*** type. The figure 8.3 details the main functions of this module.

`string`	:	*`string -> t`* returns the fingerprint of a string
`file`	:	*`string -> t`* returns the fingerprint of a file

Figure 8.3: Functions of the `Digest` module.

We use the `string` function in the following example on a small string and on a large one built from the first. The fingerprint is always of fixed length.

```
# let s = "The small cat is dead...";;
val s : string = "The small cat is dead..."
# Digest.string s;;
- : Digest.t = "xr6\127\171(\134=\238‘\252F\028\t\210$"

# let r = ref s in
    for i=1 to 100 do r:= s^ !r done;
    Digest.string !r;;
- : Digest.t = "\232\197|C]\137\180{>\224QX\155\131D\225"
```

The creation of a fingerprint for a program allows one to guarantee the contents and thus avoids the use of a bad version. For example, when code is dynamically loaded (see page 241), a fingerprint is used to select the binary file to load.

```
# Digest.file "basic.ml" ;;
- : Digest.t = "\179\026\191\137\157Ly|^w7\183\164:\167q"
```

Persistence

Persistence is the conservation of a value outside the running execution of a program. This is the case when one writes a value in a file. This value is thus accessible to any program that has access to the file. Writing and reading persistent values requires the definition of a format for representing the coding of data. In effect, one must know how to go from a complex structure stored in memory, such as a binary tree, to a *linear* structure, a list of bytes, stored in a file. This is why the coding of persistent values is called *linearization* [1].

Realization and Difficulties of Linearization

The implementation of a mechanism for the linearization of data structures requires choices and presents difficulties that we describe below.

- **read-write of data structures.** Since memory can always be viewed as a vector of words, one value can always correspond to the memory that it occupies, leaving us to preserve the useful part by then compacting the value.
- **share or copy.** Must the linearization of a data structure conserve sharing? Typically a binary tree having two identical children (in the sense of physical equality) can indicate, for the second child, that it has already saved the first. This characteristic influences the size of the saved value and the time taken to do it. On the other hand, in the presence of physically modifiable values, this could change the behavior of this value after a recovery depending on whether or not sharing was conserved.
- **circular structures.** In the case of a circular value, linearization without sharing is likely to loop. It will be necessary to conserve sharing.
- **functional values.** Functional values, or closures, are composed of an environment part and a code part. The code part corresponds to the entry point (address) of the code to execute. What must thus be done with code? It is possible to uniquely store this address, but thus only the same program will find the correct meaning of this address. It is also possible to save the list of machine instructions of this function, but that would require having a mechanism to dynamically load code.
- **guaranteeing the type when reloading.** This is the main difficulty of this mechanism. Static typing guarantees that typed values will not generate type

1. JAVA uses the term *serialization*

errors at execution time. But this is not true except for values belonging to the program during the course of execution. What type can one give to a value outside the program, that was not seen by the type verifier? Just to verify that the re-read value has the monomorphic type generated by the compiler, the type would have to be transmitted at the moment the value was saved, then the type would have to be checked when the value was loaded. Additionally, a mechanism to manage the versions of types would be needed to be safe in case a type is redeclared in a program.

`Marshal` *Module*

The linearization mechanism in the `Marshal` module allows you to choose to keep or discard the sharing of values. It also allows for the use of closures, but in this case, only the pointer to the code is saved.

This module is mainly comprised of functions for linearization via a channel or a string, and functions for recovery via a channel or a string. The linearization functions are parameterizable. The following type declares two possible options:

```
type external_flag =
  No_sharing
| Closures;;
```

The `No_sharing` constant constructor indicates that the sharing of values is not to be preserved, though the default is to keep sharing. The `Closures` constructor allows the use of closures while conserving its pointer to the code. Its absence will raise an exception if one tries to store a functional value.

Warning The `Closures` constructor is inoperative in interactive mode. It can only be used in command line mode.

The reading and writing functions in this module are gathered in figure 8.4.

```
to_channel    :  out_channel -> 'a -> extern_flag list -> unit
to_string     :  'a -> extern_flag list -> string
to_buffer     :  string -> int -> int -> 'a -> extern_flag list -> unit
from_channel  :  in_channel -> 'a
from_string   :  string -> int -> 'a
```

Figure 8.4: Functions of the `Marshal` module.

The `to_channel` function takes an output channel, a value, and a list of options and writes the value to the channel. The `to_string` function produces a string corresponding to the linearized value, whereas `to_buffer` accomplishes the same task by modifying part of a string passed as an argument. The `from_channel` function reads a linearized value from a channel and returns it. The `from_string` variant takes as input a string

and the position of the first character to read in the string. Several linearized values can be stored in the same file or in the same string. For a file, they can be read sequentially. For a string, one must specify the right offset from the beginning of the string to decode the desired value.

```
# let s = Marshal.to_string [1;2;3;4] [] in String.sub s 0 10;;
- : string = "\132\149\166\190\000\000\000\t\000\000"
```

Warning Using this module one loses the safety of static typing (see *infra*, page 233).

Loading a persistent object creates a value of indeterminate type:

```
# let x = Marshal.from_string (Marshal.to_string [1; 2; 3; 4] [] ) 0;;
val x : '_a = <poly>
```

This indetermination is denoted in Objective Caml by the weakly typed variable *'_a*. You should specify the expected type:

```
# let l =
    let s = (Marshal.to_string [1; 2; 3; 4] [] ) in
      (Marshal.from_string s 0 : int list) ;;
val l : int list = [1; 2; 3; 4]
```

We return to this topic on page 233.

Note

The `output_value` function of the preloaded library corresponds to calling `to_channel` with an empty list of options. The `input_value` function in the `Pervasives` module directly calls the `from_channel` function. These functions were kept for compatibility with old programs.

Example: Backup Screens

We want to save the *bitmap*, represented as a matrix of colors, of the whole screen. The `save_screen` function recovers the *bitmap*, converts it to a table of colors and saves it in a file whose name is passed as a parameter.

```
# let save_screen name =
   let i = Graphics.get_image 0 0 (Graphics.size_x ())
                                  (Graphics.size_y ())  in
    let j = Graphics.dump_image i in
    let oc = open_out name in
      output_value oc j;
      close_out oc;;
val save_screen : string -> unit = <fun>
```

The `load_screen` function does the reverse operation. It opens the file whose name is passed as a parameter, restores the value stored inside, converts this color matrix into a *bitmap*, then displays the bitmap.

```
# let load_screen name =
```

```
    let ic = open_in name in
    let image = ((input_value ic) : Graphics.color array array) in
      close_in ic;
      Graphics.close_graph();
      Graphics.open_graph (" "^(string_of_int(Array.length image.(0)))
                           ^"x"^(string_of_int(Array.length image)));
     let image2 = Graphics.make_image image in
       Graphics.draw_image image2 0 0; image2 ;;
val load_screen : string -> Graphics.image = <fun>
```

Warning Abstract typed values cannot be made persistent.

It is for this reason that the preceding example does not use the abstract *Graphics.image* type, but instead uses the concrete *color array array* type. The abstraction of types is presented in chapter 14.

Sharing

The loss of sharing in a data structure can make the structure completely lose its intended behavior. Let us revisit the example of the symbol generator from page 103. For whatever reason, we want to save the functional values `new_s` and `reset_s`, and thereafter use the current value of their common counter. We thus write the following program:

```
# let reset_s,new_s =
    let c = ref 0 in
    ( function () → c := 0 ) ,
    ( function s →  c:=!c+1; s^(string_of_int !c) ) ;;

# let save =
    Marshal.to_string (new_s,reset_s) [Marshal.Closures;Marshal.No_sharing] ;;

# let (new_s1,reset_s1) =
   (Marshal.from_string save 0 : ((string → string ) * (unit → unit))) ;;

# (* 1 *)
  Printf.printf "new_s : \%s\n" (new_s "X");
  Printf.printf "new_s : \%s\n" (new_s "X");
  (* 2 *)
  Printf.printf "new_s1 : \%s\n" (new_s1 "X");
  (* 3 *)
  reset_s1();
  Printf.printf "new_s1 (after reset_s1) : \%s\n" (new_s1 "X") ;;
Characters 148-154:
Unbound value new_s1
```

The first two outputs in `(* 1 *)` comply with our intent. The output obtained in `(* 2 *)` after re-reading the closures also appears correct (after `X2` comes `X3`). But, in fact, the sharing of the `c` counter between the re-read functions `new_s1` and `reset_s1` is lost, as the output of `X4` attests that one of them set the counter to zero. Each closure has a copy of the counter and the call to `reset_s1` does not reset the `new_s1` counter to zero. Thus we should not have used the `No_sharing` option during the linearization.

It is generally necessary to conserve sharing. Nevertheless in certain cases where execution speed is important, the absence of sharing speeds up the process of saving. The following example demonstrates a function that copies a matrix. In this case it might be preferable to break the sharing:

```
# let copy_mat_f (m : float array array) =
    let s = Marshal.to_string m [Marshal.No_sharing] in
      (Marshal.from_string s 0 : float array array);;
val copy_mat_f : float array array -> float array array = <fun>
```

One can also use it to create a matrix without sharing:

```
# let create_mat_f n m v =
     let m = Array.create n (Array.create m v) in
       copy_mat_f m;;
val create_mat_f : int -> int -> float -> float array array = <fun>
# let a = create_mat_f 3 4 3.14;;
val a : float array array =
  [|[|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 3.14; 3.14|];
    [|3.14; 3.14; 3.14; 3.14|]|]
# a.(1).(2) <- 6.28;;
- : unit = ()
# a;;
- : float array array =
[|[|3.14; 3.14; 3.14; 3.14|]; [|3.14; 3.14; 6.28; 3.14|];
  [|3.14; 3.14; 3.14; 3.14|]|]
```

Which is a more common behavior than that of `Array.create`, and resembles that of `Array.create_matrix`.

Size of Values

It may be useful to know the size of a persistent value. If sharing is conserved, this size also reflects the amount of memory occupied by a value. Although the encoding sometimes optimizes the size of atomic values[2], knowing the size of their respective encodings permits us to compare different implementations of a data structure. In addition, for programs that will never stop themselves, like embedded systems or even network servers; watching the size of data structures can help detect memory leaks.

2. Arrays of characters, for example.

The `Marshal` module has two functions to calculate the size of a constant. They are described in figure 8.5. The total size of a persistent value is the same as the size of its

`header_size`	:	*int*
`data_size`	:	*string -> int -> int*
`total_size`	:	*string -> int -> int*

Figure 8.5: Size functions of `Marshal`.

data structures plus the size of its header.

Below is a small example of the use of MD5 encoding to compare two representations of binary trees:

```
# let size x = Marshal.data_size (Marshal.to_string x []) 0;;
val size : 'a -> int = <fun>
# type 'a bintree1 = Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1 ;;
type 'a bintree1 = | Empty1 | Node1 of 'a * 'a bintree1 * 'a bintree1
# let s1 =
    Node1(2, Node1(1, Node1(0, Empty1, Empty1), Empty1),
             Node1(3, Empty1, Empty1)) ;;
val s1 : int bintree1 =
  Node1
   (2, Node1 (1, Node1 (0, Empty1, Empty1), Empty1),
    Node1 (3, Empty1, Empty1))
# type 'a bintree2 =
    Empty2 | Leaf2 of 'a | Node2 of 'a * 'a bintree2 * 'a bintree2 ;;
type 'a bintree2 =
  | Empty2
  | Leaf2 of 'a
  | Node2 of 'a * 'a bintree2 * 'a bintree2
# let s2 =
    Node2(2, Node2(1, Leaf2 0, Empty2), Leaf2 3) ;;
val s2 : int bintree2 = Node2 (2, Node2 (1, Leaf2 0, Empty2), Leaf2 3)
# let s1, s2 = size s1, size s2 ;;
val s1 : int = 13
val s2 : int = 9
```

The values given by the `size` function reflect well the intuition that one might have of the size of `s1` and `s2`.

Typing Problem

The real problem with persistent values is that it is possible to break the type system of Objective Caml. The creation functions return a monomorphic type (*unit* or *string*). On the other hand unmarshalling functions return a polymorphic type *'a*. From the point of view of types, you can do anything with a persistent value. Here is the usage that can be done with it (see chapter 2, page 58): create a function `magic_copy` of type

```
'a -> 'b.
# let magic_copy a =
    let s = Marshal.to_string a [Marshal.Closures] in
      Marshal.from_string s 0;;
val magic_copy : 'a -> 'b = <fun>
```

The use of such a function causes a brutal halt in the execution of the program.

```
#  (magic_copy 3 : float) +. 3.1;;
Segmentation fault
```

In interactive mode (under Linux), we even leave the toplevel (interactive) loop with a system error signal corresponding to a memory violation.

Interface with the System

The standard library has six system interface modules:

- module `Sys`: for communication between the operating system and the program;
- module `Arg`: to analyze parameters passed to the program from the command line;
- module `Filename`: operations on file names
- module `Printexc`: for the interception and printing of exceptions;
- module `Gc`: to control the mechanism that automatically deallocates memory, described in chapter 9;
- module `Callback`: to call Objective Caml functions from C, described in chapter 12.

The first four modules are described below.

Module `Sys`

This module provides quite useful functions for communication with the operating system, such as handling the signals received by a program. The values in figure 8.6 contain information about the system.

Communication between the program and the system can go through the command line, the value of an environmental variable, or through running another program. These functions are described in figure 8.7.

The functions of the figure 8.8 allow us to navigate in the file hierarchy.

Finally, the management of signals will be described in the chapter on system programming (see chapter 18).

`OS_type`	:	*string* type of system
`interactive`	:	*bool ref* true if executing at the *toplevel*
`word_size`	:	*string* size of a word (32 or 64 bits)
`max_string_length`	:	*int* maximum size of a string
`max_array_length`	:	*int* maximum size of a vector
`time`	:	*unit -> float* gives the time in seconds since the start of the program

Figure 8.6: Information about the system.

`argv`	:	*string array* contains the vector of parameters
`getenv`	:	*string -> string* retrieves the value of a variable
`command`	:	*string -> int* executes the command passed as an argument

Figure 8.7: Communication with the system.

`file_exists`	:	*string -> bool* returns `true` if the file exists
`remove`	:	*string -> unit* destroys a file
`rename`	:	*string -> string -> unit* renames a file
`chdir`	:	*string -> unit* change the current directory
`getcwd`	:	*unit -> string* returns the name of the current directory

Figure 8.8: File manipulation.

Here is a small program that revisits the example of saving a graphics window as an array of colors. The `main` function verifies that it is not started from the interactive loop, then reads from the command line the names of files to display, then tests if they exist, then displays them (with the `load_screen` function). We wait for a key to be pressed between displaying two images.

```
# let main () =
    if not (!Sys.interactive) then
      for i = 0 to Array.length(Sys.argv) -1 do
        let name = Sys.argv.(i) in
          if Sys.file_exists name then
          begin
            ignore(load_screen name);
            ignore(Graphics.read_key)
          end
      done;;
val main : unit -> unit = <fun>
```

Module `Arg`

The `Arg` module defines a small syntax for command line arguments. With this module, you can parse arguments and associate actions with them. The various elements of the command line are separated by one or more spaces. They are the values stored in the `Sys.argv` array. In the syntax provided by `Arg`, certain elements are distinguished by starting with the minus character (`-`). These are called command line *keywords* or *switches*. One can associate a specific action with a keyword or take as an argument a value of type *string*, *int* or *float*. The value of these arguments is initialized with the value found on the command line just after the keyword. In this case one can call a function that converts character strings into the expected type. The other elements on the command line are called *anonymous arguments*. One associates an action with them that takes their value as an argument. An undefined option causes the display of some short documentation on the command line. The documentation's contents are defined by the user.

The actions associated with keywords are encapsulated in the type:

```
type spec =
  | Unit of (unit → unit)     (* Call the function with unit argument*)
  | Set of bool ref           (* Set the reference to true*)
  | Clear of bool ref         (* Set the reference to false*)
  | String of (string → unit) (* Call the function with a string
                                 argument *)
  | Int of (int → unit)       (* Call the function with an int
                                 argument *)
  | Float of (float → unit)   (* Call the function with a float
                                 argument *)
  | Rest of (string → unit)   (* Stop interpreting keywords and call the
```

```
                                    function with each remaining argument*)
```

The command line parsing function is:

```
# Arg.parse ;;
- : (string * Arg.spec * string) list -> (string -> unit) -> string -> unit =
<fun>
```

Its first argument is a list of triples of the form `(key, spec, doc)` such that:

- `key` is a character string corresponding to the keyword. It starts with the reserved character `'_'`.
- `spec` is a value of type *`spec`* specifying the action associated with `key`.
- `doc` is a character string describing the option `key`. It is displayed upon a syntax error.

The second argument is the function to process the anonymous command line arguments. The last argument is a character string displayed at the beginning of the command line documentation.

The `Arg` module also includes:

- `Bad`: an exception taking as its argument a character string. It can be used by the processing functions.
- `usage`: of type *`(string * Arg.spec * string) list -> string -> unit`*, this function displays the command line documentation. One preferably provides it with the same arguments as those of `parse`.
- `current`: of type *`int ref`* that contains a reference to the current value of the index in the `Sys.argv` array. One can therefore modify this value if necessary.

By way of an example, we show a function `read_args` that initializes the configuration of the Minesweeper game seen in chapter 6, page 176. The possible options will be `-col`, `-lin` and `-min`. They will be followed by an integer indicating, respectively: the number of columns, the number of lines and the number of mines desired. These values must not be less than the default values, respectively `10`, `10` and `15`.

The processing functions are:

```
# let set_nbcols cf n = cf := {!cf with nbcols = n} ;;
# let set_nbrows cf n = cf := {!cf with nbrows = n} ;;
# let set_nbmines cf n = cf := {!cf with nbmines = n} ;;
```

All three are of type *`config ref -> int -> unit`*. The command line parsing function can be written:

```
# let read_args() =
    let cf = ref default_config in
    let speclist =
     [("-col", Arg.Int (set_nbcols cf), "number of columns (>=10)");
```

```
      ("-lin", Arg.Int (set_nbrows cf), "number of lines (>=10)");
      ("-min", Arg.Int (set_nbmines cf), "number of mines (>=15)")]
    in
    let usage_msg = "usage : minesweep [-col n] [-lin n] [-min n]" in
     Arg.parse speclist (fun s → ()) usage_msg; !cf ;;
val read_args : unit -> config = <fun>
```

This function calculates a configuration that will be passed as arguments to `open_wcf`, the function that opens the main window when the game is started. Each option is, as its name indicates, optional. If it does not appear on the command line, the corresponding parameter keeps its default value. The order of the options is unimportant.

Module `Filename`

The `Filename` module has operating system independant functions to manipulate the names of files. In practice, the file and directory naming conventions differ greatly between Windows, Unix and MacOS.

Module `Printexc`

This very short module (three functions described in figure 8.9) provides a general exception handler. This is particularly useful for programs executed in command mode[3] to be sure not to allow an exception to escape that would stop the program.

`catch`	:	*`('a -> 'b) -> 'a -> 'b`* general exception handler
`print`	:	*`('a -> 'b) -> 'a -> 'b`* print and re-raise the exception
`to_string`	:	*`exn -> string`* convert an exception to a string

Figure 8.9: Handling exceptions.

The `catch` function applies its first argument to its second. This launches the main function of the program. If an exception arrives at the level of `catch`, that is to say that if it is not handled inside the program, then `catch` will print its name and exit the program. The `print` function has the same behavior as `catch` but re-raises the exception after printing it. Finally the `to_string` function converts an exception into a character string. It is used by the two preceding functions. If we look again at the `main` function for displaying *bitmaps*, we might thus write an encapsulating function

3. The interactive mode has a general exception handler that prints a message signaling that an exception was not handled.

go in the following manner:

```
# let go () =
    Printexc.catch main ();;
val go : unit -> unit = <fun>
```

This permits the normal termination of the program by printing the value of the uncaptured exception.

Other Libraries in the Distribution

The other libraries provided with the Objective Caml language distribution relate to the following extensions:

- **graphics**, with the portable `Graphics` module that was described in chapter 5;
- **exact math**, containing many modules, and allowing the use of exact calculations on integers and rational numbers. Numbers are represented using Objective Caml integers whenever possible;
- **regular expression filtering**, allowing easier string and text manipulations. The `Str` module will be described in chapter 11;
- **Unix system calls**, with the `Unix` module allowing one to make unix system calls from Objective Caml. A large part of this library is nevertheless compatible with Windows. This bibliography will be used in chapters 18 and 20;
- **light-weight processes**, comprising many modules that will largely be described and used in chapter 19;
- **access to NDBD databases**, works only in Unix and will not be described;
- **dynamic loading of bytecode**, implemented by the `Dynlink` module.

We will describe the big integer and dynamic loading libraries by using them.

Exact Math

The big numbers library provides exact math functions using integers and rational numbers. Values of type *int* and *float* have two limitations: calculations on integers are done *modulo* the greatest positive integer, which can cause unperceived overflow errors; the results of floating point calculations are rounded, which by propagation can lead to errors. The library presented here mitigates these defects.

This library is written partly in C. For this reason, you have to build an interactive loop that includes this code using the command:

```
ocamlmktop -custom -o top nums.cma -cclib -lnums
```

The library contains many modules. The two most important ones are `Num` for all the operations and `Arith_status` for controlling calculation options. The general type *num*

is a variant type gathering three basic types:

```
type num = Int of int
         | Big_int of big_int
         | Ratio of ratio
```

The types *big_int* and *ratio* are abstract.

The operations on values of type *num* are followed by the symbol /. For example the addition of two *num* variables is written +/ and will be of type *num -> num -> num*. It will be the same for comparisons. Here is the first example that calculates the factorial:

```
# let rec fact_num n =
    if  Num.(<=/) n  (Num.Int 0) then (Num.Int 1)
    else  Num.( */ ) n  (fact_num ( Num.(-/) n (Num.Int 1)));;
val fact_num : Num.num -> Num.num = <fun>
# let r = fact_num (Num.Int 100);;
val r : Num.num = Num.Big_int <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "..." ;;
- : string = "933262154439441526816992388562667004907159682643 81..."
```

Opening the Num module makes the code of fact_num easier to read:

```
# open Num ;;
# let rec fact_num n =
    if  n <=/  (Int 0) then (Int 1)
    else  n */   (fact_num ( n -/ (Int 1))) ;;
val fact_num : Num.num -> Num.num = <fun>
```

Calculations using rational numbers are also exact. If we want to calculate the number e by following the following definition:

$$e = lim_{m\to\infty}\left(1+\frac{1}{m}\right)^m$$

We should write a function that calculates this limit up to a certain m.

```
# let  calc_e m =
    let a = Num.(+/) (Num.Int 1) ( Num.(//) (Num.Int 1) m) in
      Num.( **/ ) a  m;;
val calc_e : Num.num -> Num.num = <fun>
# let r = calc_e (Num.Int 100);;
val r : Num.num = Ratio <abstr>
# let n = Num.string_of_num r in (String.sub n 0 50) ^ "..." ;;
- : string = "270481382942152609326719471080753083367793838278 10..."
```

The Arith_status module allows us to control some calculations such as the normalization of rational numbers, approximation for printing, and processing null denominators. The arith_status function prints the state of these indicators.

```
# Arith_status.arith_status();;
```

```
Normalization during computation --> OFF
     (returned by get_normalize_ratio ())
     (modifiable with set_normalize_ratio <your choice>)

Normalization when printing --> ON
     (returned by get_normalize_ratio_when_printing ())
     (modifiable with set_normalize_ratio_when_printing <your choice>)

Floating point approximation when printing rational numbers --> OFF
     (returned by get_approx_printing ())
     (modifiable with set_approx_printing <your choice>)

Error when a rational denominator is null --> ON
     (returned by get_error_when_null_denominator ())
     (modifiable with set_error_when_null_denominator <your choice>)
- : unit = ()
```

They can be modified according to the needs of a calculation. For example, if we want to print an approximate value for a rational number, we can obtain, for the preceding calculation:

```
# Arith_status.set_approx_printing true;;
- : unit = ()
# Num.string_of_num (calc_e (Num.Int 100));;
- : string = "0.270481382942e1"
```

Calculations with big numbers take longer than those with integers and the values occupy more memory. Nevertheless, this library tries to use the most economical representations whenever possible. In any event, the ability to avoid the propagation of rounding errors and to do calculations on big numbers justifies the loss of efficiency.

Dynamic Loading of Code

The `Dynlink` module offers the ability to dynamically load programs in the form of bytecode. The dynamic loading of code provides the following advantages:

- reduces the size of a program's code. If certain modules are not used, they are not loaded.
- allows the choice at execution time of which module to load. According to certain conditions at execution time you choose to load one module rather than another.
- allows the modification of the behavior of a module during execution. Here again, under some conditions the program can load a new module and hide the old code.

The interactive loop of Objective Caml already uses such a mechanism. It is convenient to let the programmer have access to it as well.

During the loading of an object file (with the `.cmo` extension), the various expressions are evaluated. The main program, that initiated the dynamic loading of the code does not have access to the names of declarations. Therefore it is up to the dynamically loaded module to update a table of functions used by the main program.

Warning The dynamic loading of code only works for object files in bytecode.

Description of the Module

For dynamic loading of a bytecode file `f.cmo`, we need to know the access path to the file and the names of the modules that it uses. By default, dynamically loaded bytecode files do not have access to the paths and modules of the libraries in the distribution. Thus we have to add the path and the name of the required modules to the dynamic loading of the module.

`init`	:	*`unit -> unit`* initialize dynamic loading
`add_interfaces`	:	*`string list -> string list -> unit`* add the names of modules and paths for loading
`loadfile`	:	*`string -> unit`* load a bytecode file
`clear_avalaible_units`	:	*`unit -> unit`* empty the names of loadable modules and paths
`add_avalaible_units`	:	*`(string * Digest.t) list -> unit`* add the name of a module and a checksum† for loading without needing the interface file
`allow_unsafe_modules`	:	*`bool -> unit`* allow the loading of files containing **`external`** declarations
`loadfile_private`	:	*`string -> unit`* the loaded module is not accessible to modules loaded later

† The checksum of an interface `.cmi` can be obtained from the `extract_crc` command found in the catalog of libraries in the distribution.

Figure 8.10: Functions of the `Dynlink` module.

Many errors can occur during a request to load a module. Not only must the file exist with the right interface in one of the paths, but the bytecode must also be correct and loadable. These errors are gathered in the type *`error`* used as an argument to the

`Error` exception and to the `error` function of type *`error -> string`* that allows the conversion of an error into a clear description.

Example

To write a small program that allows us to illustrate dynamic loading of bytecode, we provide three modules:

- `F` that contains the definition of a reference to a function `f`;
- `Mod1` and `Mod2` that modify in different ways the function referenced by `F.f`.

The `F` module is defined in the file `f.ml`:

```
let g () =
  print_string "I am the 'f' function by default\n" ; flush stdout  ;;
let f = ref g ;;
```

The `Mod1` module is defined in the file `mod1.ml`:

```
print_string "The 'Mod1' module modifies the value of 'F.f'\n" ; flush stdout ;;
let g () =
  print_string "I am the 'f' function of module 'Mod1'\n" ;
  flush stdout ;;
F.f := g ;;
```

The `Mod2` module is defined in the file `mod2.ml`:

```
print_string "The 'Mod2' module modifies the value of 'F.f'\n" ; flush stdout ;;
let g () =
  print_string "I am the 'f' function of module 'Mod2'\n" ;
  flush stdout ;;
F.f := g ;;
```

Finally we define in the file `main.ml`, a main program that calls the original function referenced by `F.f`, loads the `Mod1` module, calls `F.f` again, then loads the `Mod2` module and calls the `F.f` function one last time:

```
let main () =
  try
    Dynlink.init () ;
    Dynlink.add_interfaces [ "Pervasives"; "F" ; "Mod1" ; "Mod2" ]
                           [ Sys.getcwd() ; "/usr/local/lib/ocaml/" ] ;
    !(F.f) () ;
    Dynlink.loadfile "mod1.cmo" ;  !(F.f) () ;
    Dynlink.loadfile "mod2.cmo" ;  !(F.f) ()
  with
    Dynlink.Error e → print_endline (Dynlink.error_message e) ; exit 1 ;;

main () ;;
```

The main program must, in addition to initializing the dynamic loading, declare by a call to `Dynlink.add_interfaces` the interface used.

We compile all of these modules:

```
$ ocamlc -c f.ml
$ ocamlc -o main dynlink.cma f.cmo main.ml
$ ocamlc -c f.cmo mod1.ml
$ ocamlc -c f.cmo mod2.ml
```

If we execute program `main`, we obtain:

```
$ main
I am the 'f' function by default
The 'Mod1' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod1'
The 'Mod2' module modifies the value of 'F.f'
I am the 'f' function of module 'Mod2'
```

Upon the dynamic loading of a module, its code is executed. This is demonstrated in our example, with the outputs beginning with `The 'Mod...`. The possible side effects that it contains are therefore reflected at the level of the program that caused the code to be loaded. This is why the different calls to `F.f` call different functions.

The `Dynlink` library offers the basic mechanism for dynamically loading bytecode. The programmer still has to manage tables such that the loading will really be effective.

Exercises

Resolution of Linear Systems

This exercise revisits the resolution of linear systems presented as an exercise in the chapter on imperative programming (see chapter 3).

1. By using the `Printf` module, write a function `print_system` that aligns the columns of the system.
2. Test this function on the examples given on page 89.

Search for Prime Numbers

The Sieve of Eratosthenes is an easily programmed algorithm that searches for prime numbers in a range of integers, given that the lower limit is a prime number. The method is:

1. Enumerate, in a list, all the values on the range.

2. Remove from the list all the values that are multiples of the first element.
3. Remove this first element from the list, and keep it as a prime.
4. Restart at step 2 as long as the list is not empty.

Here are the steps to create a program that implements this algorithm:

1. Write a function `range` that builds a range of integers represented in the form of a list.
2. Write a function `eras` that calculates the prime numbers on a range of integers starting with 2, according to the algorithm of the Sieve of Eratosthenes.
   Write a function `era_go` that takes an integer and returns a list of all the prime numbers smaller than this integer.
3. We want to write an executable `primes` that one will launch by typing the command `primes n`, where `n` is an integer. This executable will print the prime numbers smaller than `n`. For this we must use the `Sys` module and check whether a parameter was passed.

Displaying *Bitmaps*

Bitmaps saved as `color array array` are bulky. Since 24 bits of color are rarely used, it is possible to encode a *bitmap* in less space. For this we will analyze the number of colors in a *bitmap*. If the number is small (for example less than 256) we can encode each pixel in 1 byte, representing the number of the color in the table of colors of this *bitmap*.

1. Write a function `analyze_colors` exploring a value of type `color array array` and that returns a list of all the colors found in this image.
2. From this list, construct a palette. We will take a vector of colors. The index in the table will correspond to the order of the color, and the contents are the color itself. Write the function `find_index` that returns the index of a value stored in the array.
3. From this table, write a conversion function, `encode`, that goes from a `color array array` to a `string`. Each pixel is thus represented by a character.
4. Define a type `image_tdc` comprising a table that matches colors to a vector of strings, allowing the encoding of a *bitmap* (or color array) using a smaller method.
5. Write the function `to_image_tdc` to convert a `color array array` to this type.
6. Write the function `save_image_tdc` to save the values to a file.
7. Compare the size of the file obtained with the saved version of an equivalent palette.
8. Write the function `from_image_tdc` to do the reverse conversion.
9. Use it to display an image saved in a file. The file will be in the form of a value of type `bitmap_tdc`.

Summary

This chapter gave an overview of the different Objective Caml libraries presented as a set of simple modules (or compilation units). The modules for output formatting (`Printf`), persistant values (`Marshal`), the system interface (`Sys`) and the handling of exceptions (module `Printexc`) were detailed. The modules concerning parsing, memory management, system and network programming and light-weight processes will be presented in the following chapters.

To Learn More

The overview of the libraries in the distribution of the language showed the richness of the basic environment. For the `Printf` module nothing is worth more than reading a work on the C language, such as [HS94]. In [FW00] a solution is proposed for the typing of intput-output of values (module `Marshal`). The MD5 algorithm of the `Digest` module is described on the web page of its designer:

Link: http://theory.lcs.mit.edu/~rivest/homepage.html

In the same way you may find many articles on exact arithmetic used by the `num` library on the web page of Valérie Ménissier-Morain :

Link: http://www-calfor.lip6.fr/~vmm/

There are also other libraries than those in the distribution, developed by the community of Objective Caml programmers. Objective Caml. The majority of them are listed on the "Camel's hump" site:

Link: http://caml.inria.fr/hump.html

Some of them will be presented and discussed in the chapter on applications development (see chapter 22).

To know the exact contents of the various modules, don't hesitate to read the description of the libraries in the reference manual [LRVD99] or consult the online version in HTML format (see chapter 1). To enter into the details of the implementations of these libraries, nothing is better than reading the source code, available in the distribution of the language (see chapter 1).
Chapter 14 presents the language of Objective Caml modules. This allows you to build simple modules seen as independent compilation units, which will be similar to the modules presented in this chapter.

9

Garbage Collection

The execution model of a program on a microprocessor corresponds to that of imperative programming. More precisely, a program is a series of instructions whose execution modifies the *memory state* of the machine. Memory consists mainly of values created and manipulated by the program. However, like any computer resource, available memory has a finite size; a program trying to use more memory than the system provides will be in an incoherent state. For this reason, it is necessary to reuse the space of values that are at a given moment no longer used by future computations during continued execution. Such memory management has a strong influence on program execution and its efficiency.

The action of reserving a block of memory for a certain use is called *allocation*. We distinguish *static allocation*, which happens at program load time, *i.e.* before execution starts, from *dynamic allocation*, which happens during program execution. Whereas statically allocated memory is never reclaimed during execution, dynamically allocated regions are susceptible to being freed, or to being reused during execution.

Explicit memory management is risky for two reasons:

- if a block of memory is freed while it contains a value still in use, this value may become corrupted before being accessed. References to such values are called *dangling pointers*;
- if the address of a memory block is no longer known to the program, then the corresponding block cannot be freed before the end of program execution. In such cases, we speak of a *memory leak*.

Explicit memory management by the programmer requires much care to avoid these two possibilities. This task becomes rather difficult if programs manipulate complicated data structures, and in particular if data structures share common regions of memory.

To free the programmer from this difficult exercise, automatic memory management mechanisms have been introduced into numerous programming languages. The main

idea is that at any moment during execution, the only dynamically allocated values potentially useful to the program are those whose addresses are known by the program, directly or indirectly. All values that can no longer be reached at that moment cannot be accessed in the future and thus their associated memory can be reclaimed. This deallocation can be effected either immediately when a value becomes unreachable, or later when the program requires more free space than is available.

Objective Caml uses a mechanism called *garbage collection* (GC) to perform automatic memory management. Memory is allocated at value construction (*i.e.,* when a constructor is applied) and it is freed implicitly. Most programs do not have to deal with the garbage collector directly, since it works transparently behind the scenes. However, garbage collection can have an effect on efficiency for allocation-intensive programs. In such cases, it is useful to control the GC parameters, or even to invoke the collector explicitly. Moreover, in order to interface Objective Caml with other languages (see chapter 12), it is necessary to understand what constraints the garbage collector imposes on data representations.

Chapter Overview

This chapter presents dynamic memory allocation strategies and garbage collection algorithms, in particular the one used by Objective Caml which is a combination of the presented algorithms. The first section provides background on different classes of memory and their characteristics. The second section describes memory allocation and compares implicit and explicit deallocation. The third section presents the major GC algorithms. The fourth section details Objective Caml's algorithm. The fifth section uses the `Gc` module to control the heap. The sixth section introduces the use of *weak pointers* from the `Weak` module to implement caches.

Program Memory

A machine code program is a sequence of instructions manipulating values in memory. Memory consists generally of the following elements:

- processor registers (for direct and fast access),
- the stack,
- a data segment (static allocation region),
- the heap (dynamic allocation region).

Only the stack and the dynamic allocation region can change in size during the execution of a program. Depending on the programming language used, some control over these classes of memory can be exercised. Whereas the program instructions (code) usually reside in static memory, dynamic linking (see page 241) makes use of dynamic memory.

Allocation and Deallocation of Memory

Most languages permit dynamic memory allocation, among them C, Pascal, Lisp, ML, SmallTalk, C++, Java, ADA.

Explicit Allocation

We distinguish two types of allocation:

- a simple allocation reserving a block of memory of a certain size without concern of its contents;
- an allocation combining the reservation of space with its initialization.

The first case is illustrated by the function `new` in Pascal or `malloc` in C. These return a pointer to a memory block (*i.e.* its address), through which the value stored in memory can be read or modified. The second case corresponds to the construction of values in Objective Caml, Lisp, or in object-oriented languages. Class instances in object-oriented languages are constructed by combining `new` with the invocation of a constructor for the class, which usually expects a number of parameters. In functional languages, constructor functions are called in places where a structural value (tuple, list, record, vector, or closure) is defined.

Let's examine an example of value construction in Objective Caml. The representation of values in memory is illustrated in Figure 9.1.

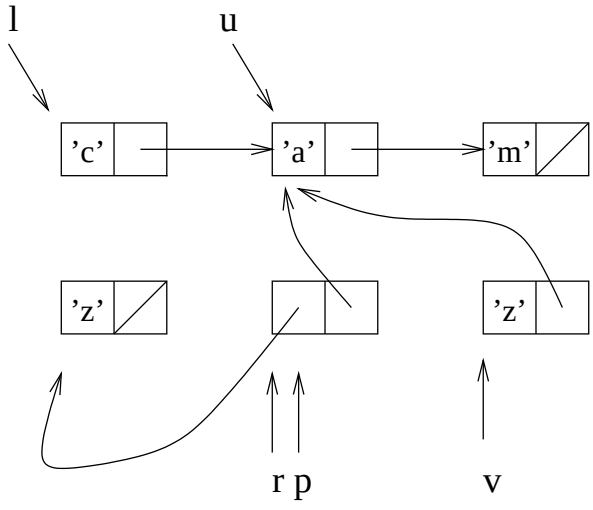


Figure 9.1: Memory representation of values.

```
# let u = let l = ['c'; 'a'; 'm'] in List.tl l ;;
val u : char list = ['a'; 'm']
```

```
# let v = let r = ( ['z'] , u )
          in match r with p → (fst p) @ (snd p) ;;
val v : char list = ['z'; 'a'; 'm']
```

A list element is represented by a tuple of two words, the first containing a character and the second containing a pointer to the next element of the list. The actual runtime representation differs slightly and is described in the chapter on interfacing with C (see page 315).

The first definition constructs a value named `l` by allocating a cell (constructor ::) for each element of the list `['c';'a';'m']`. The global declaration `u` corresponds to the tail of `l`. This establishes a sharing relationship between `l` and `u`, *i.e.* between the argument and the result of the function call to *`List.tl`*.

Only the declaration `u` is known after the evaluation of this first statement.

The second statement constructs a list with only one element, then a pair called `r` containing this list and the list `u`. This pair is pattern matched and renamed `p` by the matching. Next, the first element of `p` is concatenated with its second element, which creates a value `['z';'a';'m']` tied to the global identifier `v`. Notice that the result of `snd` (the list `['a';'m']`) is shared with its argument `p` whereas the result of `fst` (the character `'z'`) is copied.

In each case memory allocation is explicit, meaning that it is requested by the programmer (by a language command or instruction).

Note

Allocated memory stores information on the size of the object allocated in order to be able to free it later.

Explicit Reclamation

Languages with explicit memory reclamation possess a freeing operator (`free` in C or `dispose` in Pascal) that take the address (a pointer) of the region to deallocate. Using the information stored at allocation time, the program frees this region and may re-use it later.

Dynamic allocation is generally used to manipulate data structures that evolve, such as lists, trees *etc.*. Freeing the space occupied by such data is not done in one fell swoop, but instead requires a function to traverse the data. We call such functions destructors.

Although correctly defining destructors is not too difficult, their use is quite delicate. In fact, in order to free the space occupied by a structure, it is necessary to traverse the structure's pointers and apply the language's freeing operator. Leaving the responsibility of freeing memory to the programmer has the advantage that the latter is sure of the actions taken. However, incorrect use of these operators can cause an error during the execution of the program. The principal dangers of explicit memory reclamation are:

- dangling pointers: a memory region has been freed while there are still pointers pointing at it. If the region is reused, access to the region by these pointers risks being incoherent.
- Inaccessible memory regions (a memory "leak"): a memory region is still allocated, but no longer referenced by any pointer. There is no longer any possibility of freeing the region. There is a clear loss of memory.

The entire difficulty with explicit memory reclamation is that of knowing the lifetime of the set of values of a program.

Implicit Reclamation

Languages with implicit memory reclamation do not possess memory-freeing operators. It is not possible for the programmer to free an allocated value. Instead, an automatic reclamation mechanism is engaged when a value is no longer referenced, or at the time of an allocation failure, that is to say, when the heap is full.

An automatic memory reclamation algorithm is in some ways a global destructor. This characteristic makes its design and implementation more difficult than that of a destructor dedicated to a particular data structure. But, once this difficulty is overcome, the memory reclamation function obtained greatly enhances the safety of memory management. In particular, the risk of dangling pointers disappears.

Furthermore, an automatic memory reclamation mechanism may bring good properties to the heap:

- *compaction*: all the recovered memory belongs to a single block, thereby avoiding fragmentation of the heap, and allowing allocation of objects of the size of the free space on the heap;
- *localization*: the different parts of the same value are close to one another from the point of view of memory address, permitting them to remain in the same memory pages during use, and thereby avoiding their erasure from cache memory.

Design choices for a garbage collector must take certain criteria and constraints into account:

- reclamation factor: what percentage of unused memory is available?
- memory fragmentation: can one allocate a block the size of the free memory?
- the slowness of allocation and collection;
- what freedom do we have regarding the representation of values?

In practice, the safety criterion remains primordial, and garbage collectors find a compromise among the other constraints.

Automatic Garbage Collection

We classify automatic memory reclamation algorithms into two classes:

- reference counters: each allocated region knows how many references there are to it. When this number becomes zero, the region is freed.
- sweep algorithms: starting from a set of *roots*, the collection of all accessible values is traversed in a way similar to the traversal of a directed graph.

Sweep algorithms are more commonly used in programming languages. In effect, reference counting garbage collectors increase the processing costs (through counter updating) even when there is no need to reclaim anything.

Reference Counting

Each allocated region (object) is given a counter. This counter indicates the number of pointers to the object. It is incremented each time a reference to the object is shared. It is decremented whenever a pointer to the object disappears. When the counter becomes zero, the object is garbage collected.

The advantage of such a system comes from the immediate freeing of regions that are no longer used. Aside from the systematic slowdown of computations, reference counting garbage collectors suffer from another disadvantage: they do not know how to process circular objects. Suppose that Objective Caml had such a mechanism. The following example constructs a temporary value `l`, a list of characters of where the last element points to the cell containing `'c'`. This is clearly a circular value (figure 9.2).

```
# let rec l = 'c' :: 'a' :: 'm' :: l in List.hd l ;;
- : char = 'c'
```

At the end of the calculation of this expression each element of the list `l` has a counter

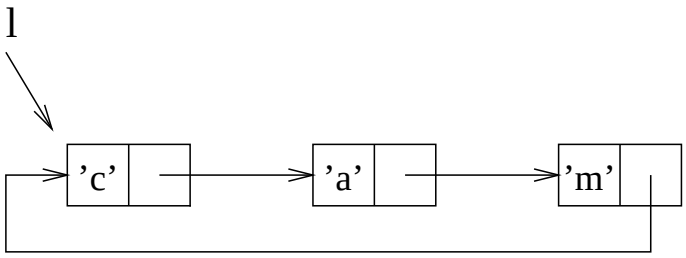


Figure 9.2: Memory representation of a circular list.

equal to one (even the first element, for the tail points to the head). This value is no longer accessible and yet cannot be reclaimed because its reference counter is not zero. In languages equipped with memory reclamation via reference counting—such as Python—and which allow the construction of circular values, it is necessary to add a memory sweep algorithm.

Sweep Algorithms

Sweep algorithms allow us to explore the graph of accessible values on the heap. This exploration uses a set of roots indicating the beginning of the traversal. These roots are exterior to the heap, stored most often in a stack. In the example in figure 9.1, we can suppose that the values of `u` and `v` are roots. The traversal starting from these roots constructs the graph of the values to save: the cells and pointers marked with heavy lines in figure 9.3.

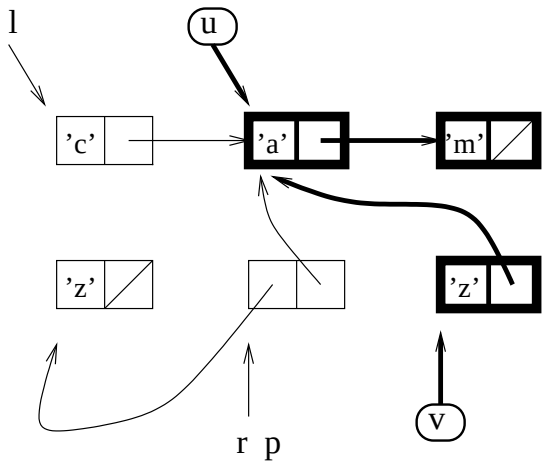


Figure 9.3: Memory reclamation after a garbage collection.

The traversal of this graph necessitates knowing how to distinguish immediate values from pointers in the heap. If a root points to an integer, we must not consider this value to be the address of another cell. In functional languages, this distinction is made by using a few bits of each cell of the heap. We call these bits *tag bits*. This is why Objective Caml integers only use 31 bits. This option is described in Chapter 12, page 325. We describe other solutions to the problem of distinguishing between pointers and immediate values in this chapter, page 260.

The two most commonly used algorithms are *Mark&Sweep*, which constructs the list of the free cells in the heap, and *Stop&Copy*, which copies cells that are still alive to a second memory region.

The heap should be seen as a vector of memory boxes. The representation of the state of the heap for the example of figure 9.1 is illustrated in figure 9.4.

We use the following characteristics to evaluate a sweep algorithm:

- efficiency: does the time-complexity depend on the size of the heap or only on the number of the living cells?
- reclamation factor: is all of the free memory usable?
- compactness: is all of the free memory usable in a single block?

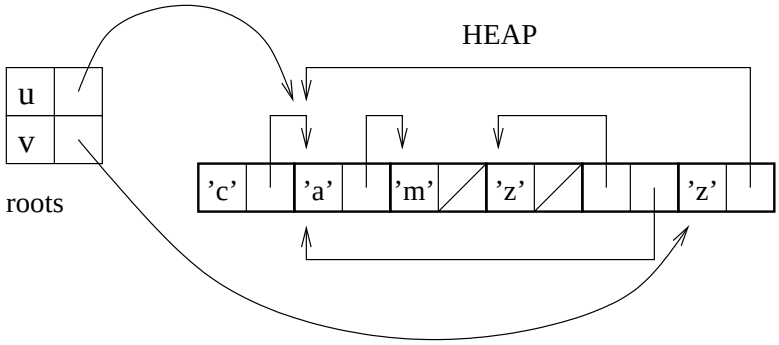


Figure 9.4: State of the heap.

- localization: are all of the different cells of a structured value close to one another?
- memory needs: does the algorithm need to use part of the memory when it runs?
- relocation: do values change location following a garbage collection?

Localization avoids changing memory pages when traversing a structured value. Compactness avoids fragmentation of the heap and allows allocations equal to the amount of available memory. The efficiency, reclamation factor, and supplementary memory needs are intimately linked to the time and space complexity of the algorithm.

Mark&Sweep

The idea of *Mark&Sweep* is to keep an up-to-date list of the free cells in the heap called the free list. If, at the time of an allocation request, the list is empty or no longer contains a free cell of a sufficient size, then a *Mark&Sweep* occurs.

It proceeds in two stages:

1. the marking of the memory regions in use, starting from a set of roots (called the *Mark* phase); then
2. reclamation of the unmarked memory regions by sequentially sweeping through the whole heap (called the *Sweep* phase).

One can illustrate the memory management of *Mark&Sweep* by using four "colorings" of the heap cells: white, gray[1], black, and hached. The mark phase uses the gray; the sweep phase, the hached; and the allocation phase, the white.

The meaning of the gray and black used by marking is as follows:

- gray: marked cells whose descendents are not yet marked;
- black: marked cells whose descendents are also marked.

1. In the online version of the book, the gray is slightly bluish.

It is necessary to keep the collection of grayed cells in order to be sure that everything has been explored. At the end of the marking each cell is either white or black, with black cells being those that were reached from the roots. Figure 9.5 shows an intermediate marking stage for the example of figure 9.4: the root `u` has been swept, and the sweeping of `v` is about to begin.

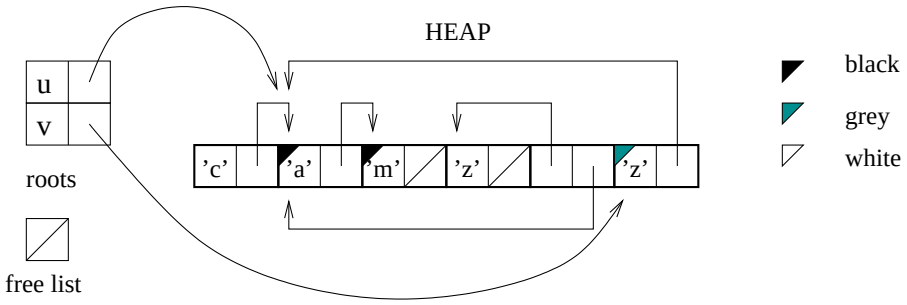


Figure 9.5: Marking phase.

It's during the sweep phase that the free list is constructed. The sweep phase modifies the colorings as follows:

- black becomes white, as the cell is alive;
- white becomes hached, and the cell is added to the free list.

Figure 9.6 shows the evolution of the colors and the construction of the free list.

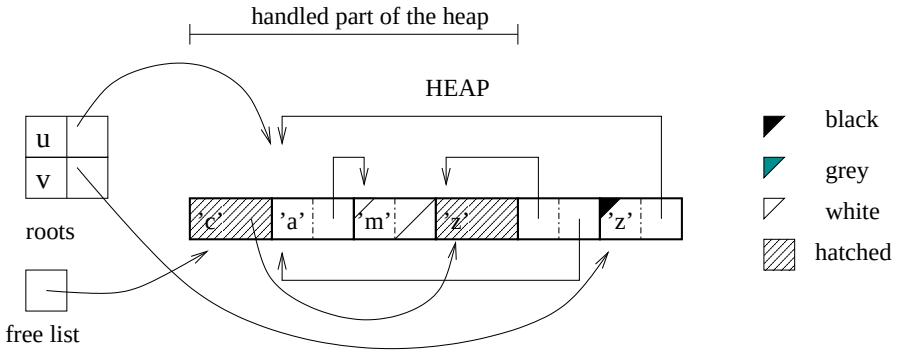


Figure 9.6: Sweep phase.

Characteristics of *Mark&Sweep* are that it:

- depends on the size of the entire heap (*Sweep* phase);
- reclaims all possible memory;
- does not compact memory;

- does not guarantee localization;
- does not relocate data.

The marking phase is generally implemented by a recursive function, and therefore uses space on the execution stack. One can give a completely iterative version of *Mark&Sweep* that does not require a stack of indefinite size, but it turns out to be less efficient than the partially recursive version.

Finally, *Mark&Sweep* needs to know the size of values. The size is either encoded in the values themselves, or deduced from the memory address by splitting the heap into regions that allocate objects of a bounded size. The *Mark&Sweep* algorithm, implemented since the very first versions of Lisp, is still widely used. A part of the Objective Caml garbage collector uses this algorithm.

Stop&Copy

The principal idea of this garbage collector is to use a secondary memory in order to copy and compact the memory regions to be saved. The heap is divided into two parts: the useful part (called *from-space*), and the part being re-written (called *to-space*).

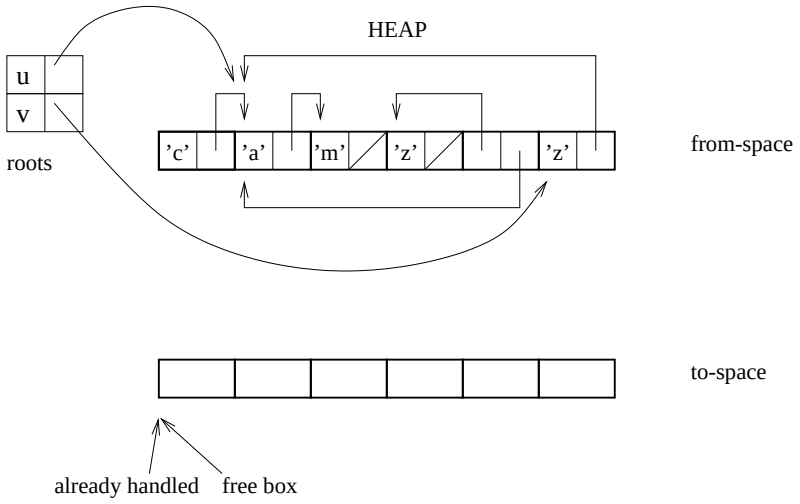


Figure 9.7: Beginning of *Stop&Copy*.

The algorithm is the following. Beginning from a set of roots, each useful part of the *from-space* is copied to the *to-space*; the new address of a relocated value is saved (most often in its old location) in order to update all of the other values that point to this value.

The contents of the rewritten cells gives new roots. As long as there are unprocessed roots the algorithm continues.

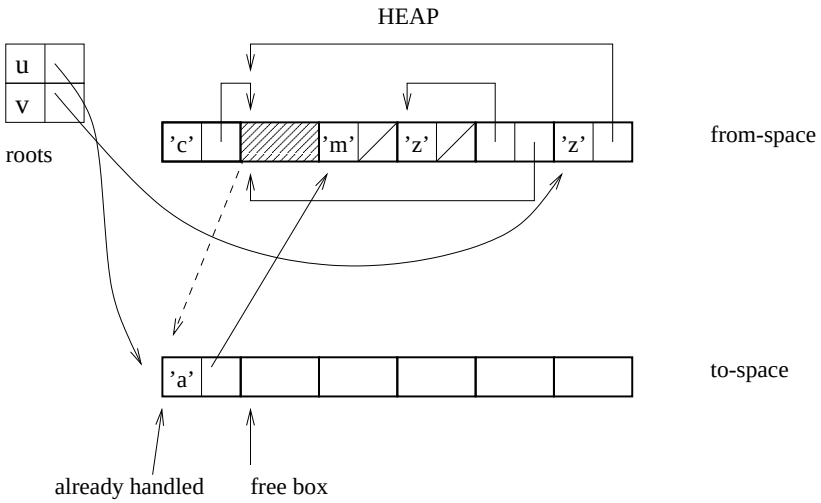


Figure 9.8: Rewriting from *from-space* into *to-space*.

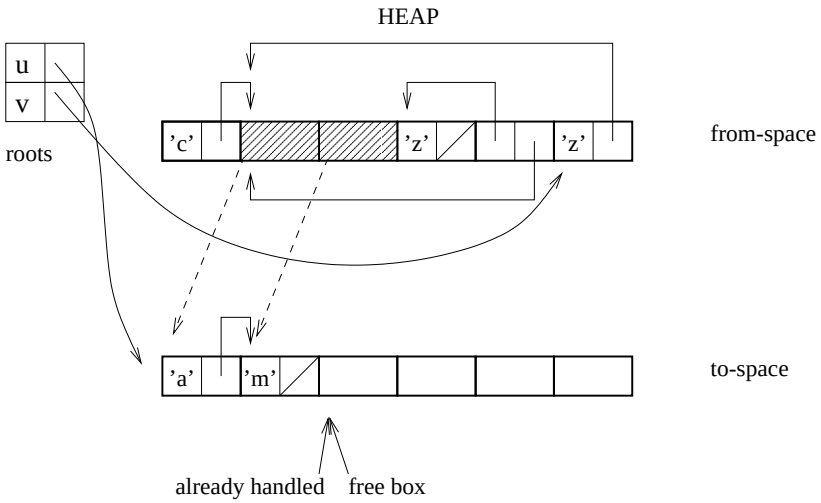


Figure 9.9: New roots.

In the case of sharing, in other words, when attempting to relocate a value that has already been relocated, it suffices to use the new address.

At the end of garbage collection, all of the roots are updated to point to their new addresses. Finally, the roles of the two parts are reversed for the next garbage collection.

The principal characteristics of this garbage collector are the following:

- it depends solely on the size of the objects to be kept;

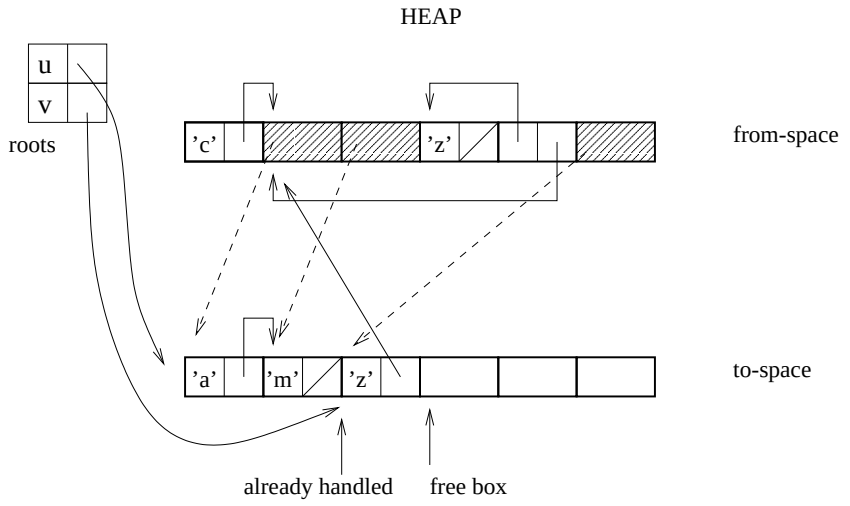


Figure 9.10: Sharing.

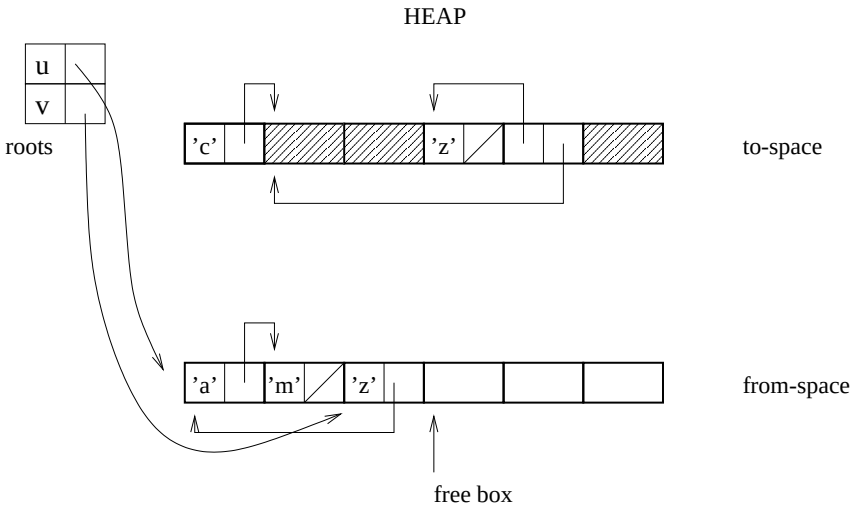


Figure 9.11: Reversing the two parts.

- only half of the memory is available;
- it compacts memory;
- it may localize values (using breadth-first traversal);
- it does not use extra memory (only *from-space*+*to-space*);
- the algorithm is not recursive;
- it relocates values into the new part of memory;

Other Garbage Collectors

Many other techniques, often derived from the two preceding, have been used: either in particular applications, *e.g.,* the manipulation of large matrices in symbolic calculations, or in a general way linked to compilation techniques. Generational garbage collectors allow optimizations based on the age of the values. Conservative garbage collectors are used where there is not an explicit differentiation between immediate values and pointers (for example, when one translates into C). Finally, incremental garbage collectors allow us to avoid a noticeable slow-down at the time of garbage collection activation.

Generational Garbage Collection

Functional programs are, in general, programs that allocate frequently. We notice that a very large number of values have a very short lifetime[2]. On the other hand, when a value has survived several garbage collections, it is quite likely to survive for a while longer. In order to avoid complete traversal of the heap—as in *Mark&Sweep*—during each memory reclamation, we would like to be able to traverse only the values that have survived one or more garbage collections. Most frequently, it is among the young values that we will recover the most space. In order to take advantage of this property, we give objects dates, either a time-stamp or the number of garbage collections survived. To optimize garbage collection, we use different algorithms according to the age of the values:

- The garbage collections for young objects should be fast and traverse only the younger generations.
- The garbage collections for old objects should be rare and do well at collecting free space from the entire memory.

As a value ages it should take part less and less in the most frequent garbage collections. The difficulty, therefore, is taking count of only the region of memory occupied by young objects. In a purely functional language, that is, a language without assignment, younger objects reference older objects, and on the other hand, older objects do not possess pointers to younger objects because they were created before the young objects existed. Therefore, these garbage collection techniques lend themselves well to functional languages, with the exception of those with delayed evaluation which can in fact evaluate the constituents of a structure after evaluating the structure itself. On the other hand, for functional languages with assignment it is always possible to modify part of an older object to refer to a younger object. The problem then is to save young memory regions referenced only by an older value. For this, it is necessary to keep an up-to-date table of references from old objects to young objects in order to have a correct garbage collection. We study the case of Objective Caml in the following section.

2. Most values do not survive a single garbage collection.

Conservative Garbage Collectors

To this point, all of the garbage collection techniques presume knowing how to tell a pointer from an immediate value. Note that in functional languages with parametric polymorphism values are uniformly represented, and in general occupy one word of memory[3]. This is what allows having generic code for polymorphic functions.

However, this restriction on the range for integers may not be acceptable. In this case, conservative garbage collectors make it possible to avoid marking immediate values such as integers. In this case, every value uses an entire memory word without any tag bits. In order to avoid traversing a memory region starting from a root actually containing an integer, we use an algorithm for discriminating between immediate values and pointers that relies on the following observations:

- the addresses of the beginning and end of the heap are known so any value outside of these bounds is an immediate value;
- allocated objects are aligned on a word address. Every value that does not correspond to such an alignment must also be an immediate value.

Thus each heap value that is valid from the point of view of being an address into the heap is considered to be a pointer and the garbage collector tries to keep this region, including those cases where the value is in fact an immediate value. These cases may become very rare by using specific memory pages according to the size of the objects. It is not possible to guarantee that the entire unused heap is collected. This is the principal defect of this technique. However, we remain certain that only unused regions are reclaimed.

In general, conservative garbage collectors are conservative, *i.e.,* they do not relocate objects. Indeed, as the garbage collector considers some immediate values as pointers, it would be harmful to change their value. Nevertheless, some refinements can be introduced for building the sets of roots, which allow to relocate corresponding to clearly known roots.

Garbage collection techniques for ambiguous roots are often used when compiling a functional language into C, seen here as a portable assembler. They allow the use of immediate C values coded in a memory word.

Incremental Garbage Collection

One of the criticisms frequently made of garbage collection is that it stops the execution of a running program for a time that is perceptible to the user and is unbounded. The first is embarrassing in certain applications, for instance, rapid-action games where the halting of the game for a few seconds is too often prejudicial to the player, as the execution restarts without warning. The latter is a source of loss of control for applications which must process a certain number of events in a limited time. This is

3. The only exception in Objective Caml relates to arrays of floating point values (see chapter 12, page 331).

typically the case for embedded programs which control a physical device such as a vehicle or a machine tool. These applications, which are real-time in the sense that they must respond in a bounded time, most often avoid using garbage collectors.

Incremental garbage collectors must be able to be interrupted during any one of their processing phases and be able to restart while assuring the safety of memory reclamation. They give a sufficiently satisfactory method for dealing with the former case, and can be used in the latter case by enforcing a programming discipline that clearly isolates the software components that use garbage collection from those that do not.

Let us reconsider the *Mark&Sweep* example and see what adaptations are necessary in order to make it incremental. There are essentially two:

1. how to be sure of having marked everything during the marking phase?
2. how to allocate during either the marking phase or the reclamation phase?

If *Mark&Sweep* is interrupted in the *Mark* phase, it is necessary to assure that cells allocated between the interruption of marking and its restart are not unduly reclaimed by the *Sweep* that follows. For this, we mark cells allocated during the interruption in black or gray in anticipation.

If the *Mark&Sweep* is interrupted during the *Sweep* phase, it can continue as usual in re-coloring the allocated cells white. Indeed, as the *Sweep* phase sequentially traverses the heap, the cells allocated during the interruption are localized before the point where the sweep restarts, and they will not be re-examined before the next garbage collection cycle.

Figure 9.12 shows an allocation during the reclamation phase. The root `w` is created by:

```
# let w = 'f' :: v;;
val w : char list = ['f'; 'z'; 'a'; 'm']
```

Memory Management by Objective Caml

Objective Caml's garbage collector combines the various techniques described above. It works on two generations, the old and the new. It mainly uses a *Stop&Copy* on the new generation (a minor garbage collection) and an incremental *Mark&Sweep* on the old generation (major garbage collection).

A young object that survives a minor garbage collection is relocated to the old generation. The *Stop&Copy* uses the old generation as the `to-space`. When it is finished, the entire `from-space` is completely freed.

When we presented generational garbage collectors, we noted the difficulty presented by impure functional languages: an old-generation value may reference an object of the new generation. Here is a small example.

```
# let older = ref [1] ;;
```

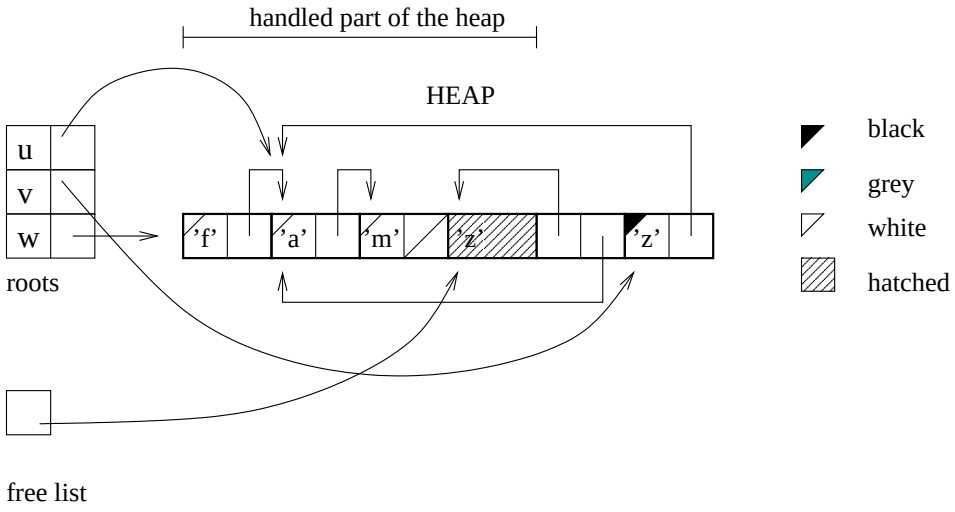


Figure 9.12: Allocation during reclamation.

```
val older : int list ref = {contents=[1]}
(* ... *)
# let newer = [2;5;8] in
    older := newer ;;
- : unit = ()
```

The comment `(* ... *)` replaces a long sequence of code in which `older` passes into the older generation. The minor garbage collection must take account of certain old generation values. Therefore we must keep an up-to-date table of the references from the old generation to the new that becomes part of the set of roots for the minor garbage collection. This table of roots grows very little and becomes empty just after a minor garbage collection.

It is to be noted that the *Mark&Sweep* of the old generation is incremental, which means that a part of the major garbage collection happens during each minor garbage collection. The major garbage collection is a *Mark&Sweep* that follows the algorithm presented on page 259. The relevance of this incremental approach is the reduction of waiting time for a major garbage collection by advancing the marking phase with each minor garbage collection. When a major garbage collection is activated, the marking of the unprocessed regions is finished, and the reclamation phase is begun. Finally, as *Mark&Sweep* may fragment the old generation significantly, a compaction algorithm may be activated after a major garbage collection.

Putting this altogether, we arrive at the following stages:

1. minor garbage collection: perform a *Stop&Copy* on the young generation; age the surviving objects by having them change zone; and then do part of the *Mark&Sweep* of the old generation.
   It fails if the zone change fails, in which case we go to step 2.

2. end of the major garbage collection cycle.
   When this fails go on to step 3.
3. another major garbage collection, to see if the objects counted as used during the incremental phases have become free.
   When this fails, go on to step 4.
4. Compaction of the old generation in order to obtain maximal contiguous free space. If this last step does not succeed, there are no other possibilities, and the program itself fails.

The `GC` module allows activation of the various phases of the garbage collector.

A final detail of the memory management of Objective Caml is that the heap space is not allocated once and for all at the beginning of the program, but evolves with time (increasing or decreasing by a given size).

Module `Gc`

The `Gc` module lets one obtain statistics about the heap and gives control over its evolution as well as allowing the activation of various garbage collector phases. Two concrete record types are defined: *`stat`* and *`control`*. The fields of type *`control`* are modifiable; whereas those of *`stat`* are not. The latter simply reflect the state of the heap at a given moment.

The fields of a *`stat`* mainly contain counters indicating:

- the number of garbage collections: `minor_collections`, `major_collections` and `compactions`;
- the number of words allocated and transfered since the beginning of the program: `minor_words`, `promoted_words`, and `major_words`.

The fields of the record *`control`* are:

- `minor_heap_size`, which defines the size of the zone allotted to the younger generation;
- `major_heap_increment`, which defines the increment applied to the growth of the region for the older generation;
- `space_overhead`, which defines the percentage of the memory used beyond which a major garbage collection is begun (the default value is 42);
- `max_overhead`, which defines the connection between free memory and occupied memory after which compactification is activated. A value of 0 causes a systematic compactification after every major garbage collection. The maximal value of 1000000 inhibits compactification.
- `verbose` is an integer parameter governing the tracing of the activities of the garbage collector.

Functions manipulating the types *`stat`* and *`control`* are given in figure 9.13.

stat	*unit → stat*
print_stat	*out_channel → unit*
get	*unit → control*
set	*control → unit*

Figure 9.13: Control and statistical functions for the heap.

The following functions, of type *unit -> unit*, force the execution of one or more stages of the Objective Caml garbage collector: `minor` (stage 1), `major` (stages 1 and 2), `full_major` (stages 1, 2 and 3) and `compact` (stages 1, 2, 3 and 4).

Examples

Here is what the `Gc.stat` call shows:

```
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words=555677; Gc.promoted_words=61254; Gc.major_words=205249;
 Gc.minor_collections=17; Gc.major_collections=3; Gc.heap_words=190464;
 Gc.heap_chunks=3; Gc.live_words=157754; Gc.live_blocks=35600;
 Gc.free_words=32704; Gc.free_blocks=83; Gc.largest_free=17994;
 Gc.fragments=6; Gc.compactions=0}
```

We see the number of executions of each phase: minor garbage collection, major garbage collection, compaction, as well as the number of words handled by the different memory spaces. Calling `compact` forces the four stages of the garbage collector, causing the heap statistics to be modified (see the call of `Gc.stat`).

```
# Gc.compact();;
- : unit = ()
# Gc.stat();;
- : Gc.stat =
{Gc.minor_words=562155; Gc.promoted_words=62288; Gc.major_words=206283;
 Gc.minor_collections=18; Gc.major_collections=5; Gc.heap_words=190464;
 Gc.heap_chunks=3; Gc.live_words=130637; Gc.live_blocks=30770;
 Gc.free_words=59827; Gc.free_blocks=1; Gc.largest_free=59827;
 Gc.fragments=0; Gc.compactions=1}
```

The fields `GC.minor_collections` and `compactions` are incremented by 1, whereas the field `Gc.major_collections` is incremented by 2. All of the fields of type *GC.control* are modifiable. For them to be taken into account, we must use the function `Gc.set`, which takes a value of type *control* and modifies the behavior of the garbage collector.

For example, the field `verbose` may take a value from 0 to 127, controlling 7 different indicators.

```
# c.Gc.verbose <- 31;;
```

```
Characters 1-2:
This expression has type int * int but is here used with type Gc.control
# Gc.set c;;
Characters 7-8:
This expression has type int * int but is here used with type Gc.control
# Gc.compact();;
- : unit = ()
```

which prints:

```
<>Starting new major GC cycle
allocated_words = 329
extra_heap_memory = 0u
amount of work to do = 3285u
Marking 1274 words
!Starting new major GC cycle
Compacting heap...
done.
```

The different phases of the garbage collector are indicated as well as the number of objects processed.

Module **Weak**

A weak pointer is a pointer to a region which the garbage collector may reclaim at any moment. It may be surprising to speak of a value that might disappear at any moment. In fact, we must see these weak pointers as a reservoir of values that may still be available. This turns out to be particularly useful when memory resources are small compared to the elements to be saved. The classic case is the management of a memory cache: a value may be lost, but it remains directly accessible as long as it exists.

In Objective Caml one cannot directly manipulate weak pointers, only arrays of weak pointers. The `Weak` module defines the abstract type *`'a Weak.t`*, corresponding to the type *`'a option array`*, a vector of weak pointers of type `'a`. The concrete type *`'a option`* is defined as follows:

```
type 'a option = None | Some of 'a;;
```

The main functions of this module are defined in figure 9.14.

The `create` function allocates an array of weak pointers, each initialized to `None`. The `set` function puts a value of type *`'a option`* at a specified index. The `get` function returns the value contained at index n in a table of weak pointers. The returned value is then referenced, and no longer reclaimable as long as this reference exists. To

function	type
`create`	*`int -> 'a t`*
`set`	*`'a t -> int -> 'a option -> unit`*
`get`	*`'a t -> int -> 'a option`*
`check`	*`'a t -> int -> bool`*

Figure 9.14: Main functions of the `Weak` module.

verify the effective existence of a value, one uses either the `check` function or pattern matching on the *`'a option`* type's patterns. The former solution does not depend on the representation choice for weak pointers.

Standard functions for sequential structures also exist: `length`, for the length, and `fill` and `blit` for copies of parts of the array.

Example: an Image Cache

In an image-processing application, it is not rare to work on several images. When the user moves from one image to another, the first is saved to a file, and the other is loaded from another file. In general, only the names of the latest images processed are saved. In order to avoid overly frequent disk access while at the same time not using too much memory space, we use a memory cache which contains the last images loaded. The contents of the cache may be freed if necessary. We implement this with a table of weak pointers, leaving the decision of when to free the images up to the garbage collector. To load an image we first search the cache. If the image is there, it becomes the current image. If not, its file is read.

We define a table of images in the following manner:

```
# type table_of_images = {
    size : int;
    mutable ind : int;
    mutable name : string;
    mutable current : Graphics.color array array;
    cache : ( string * Graphics.color array array) Weak.t } ;;
```

The field `size` gives the size of the table; the field `ind` gives the index of the current image; the field `name`, the name of the current image; the field `current`, the current image, and the field `cache` contains the array of weak pointers to the images. It contains the last images loaded and their names.

The function `init_table` initializes the table with its first image.

```
# let open_image filename =
    let ic = open_in filename
    in let i = ((input_value ic) : Graphics.color array array)
    in ( close_in ic ; i ) ;;
val open_image : string -> Graphics.color array array = <fun>
```

```
# let init_table n filename =
    let i = open_image filename
    in let c =  Weak.create n
    in Weak.set c 0 (Some (filename,i)) ;
       { size=n; ind=0; name = filename; current = i; cache = c } ;;
val init_table : int -> string -> table_of_images = <fun>
```

The loading of a new image saves the current image in the table and loads the new one. To do this, we must first try to find the image in the cache.

```
# exception Found of int * Graphics.color array array ;;
# let search_table filename table =
    try
      for i=0 to table.size-1 do
        if i<>table.ind then match Weak.get table.cache i with
               Some (n,img) when n=filename  → raise (Found (i,img))
             | _ → ()
        done ;
      None
    with Found (i,img) → Some (i,img) ;;



# let load_table filename table =
    if table.name = filename then ()  (* the image is the current image *)
    else
      match search_table filename table with
        Some (i,img) →
        (* the image found becomes the current image *)
          table.current <- img ;
          table.name <- filename ;
          table.ind <- i
      | None →
        (* the image isn't in the cache, need to load it *)
        (* find an empty spot in the cache *)
          let i = ref 0 in
            while (!i<table.size && Weak.check table.cache !i) do incr i done ;
            (* if none are free, take a full slot *)
              ( if !i=table.size then i:=(table.ind+1) mod table.size ) ;
              (* load the image here and make it the current one *)
              table.current <- open_image filename ;
              table.ind <- !i ;
              table.name <- filename ;
              Weak.set table.cache table.ind (Some (filename,table.current)) ;;
val load_table : string -> table_of_images -> unit = <fun>
```

The `load_table` function tests to see if the image requested is current. If not, it checks the cache to see if the image exists; if that fails, the function loads the image from disk. In either of the latter two cases, it makes the image become the current one.

To test this program, we use the following cache-printing function:

```
# let print_table table =
    for i = 0 to table.size-1 do
      match Weak.get table.cache ((i+table.ind) mod table.size) with
        None → print_string "[] "
      | Some (n,_) → print_string n ; print_string " "
    done ;;
val print_table : table_of_images -> unit = <fun>
```

Then we test the following program:

```
# let t = init_table 10 "IMAGES/animfond.caa" ;;
val t : table_of_images =
  {size=10; ind=0; name="IMAGES/animfond.caa";
   current=
    [|[|7372452; 7372452; 7372452; 7372452; 7372452; 7372452; 7372452;
        7372452; 7372452; 7372452; 7372452; 7372452; 7505571; 7505571; ...|];
      ...|];
   cache=...}
# load_table "IMAGES/anim.caa" t ;;
- : unit = ()
# print_table t ;;
IMAGES/anim.caa [] [] [] [] [] [] [] [] [] - : unit = ()
```

This cache technique can be adapted to various applications.

Exercises

Following the evolution of the heap

In order to follow the evolution of the heap, we suggest writing a function that keeps information on the heap in the form of a record with the following format:

```
# type tr_gc = {state : Gc.stat;
                time : float; number : int};;
```

The time corresponds to the number of milliseconds since the program began and the number serves to distinguish between calls. We use the function `Unix.time` (see chapter 18, page 572) which gives the running time in milliseconds.

1. Write a function `trace_gc` that returns such a record.
2. Modify this function so that it can save a value of type *tr_gc* in a file in the form of a persistant value. This new function needs an output channel in order to write. We use the `Marshal` module, described on page 228, to save the record.
3. Write a stand-alone program, taking as input the name of a file containing records of type of *tr_gc*, and displaying the number of major and minor garbage collections.

4. Test this program by creating a trace file at the interactive loop level.

Memory Allocation and Programming Styles

This exercise compares the effect of programming styles on the growth of the heap. To do this, we reconsider the exercise on prime numbers from chapter 8 page 244. We are trying to compare two versions, one tail-recursive and the other not, of the sieve of Eratosthenes.

1. Write a tail-recursive function `erart` (this name needs fixing) that calculates the prime numbers in a given interval. Then write a function that takes an integer and returns the list of smaller prime numbers.
2. By using the preceding functions, write a program (change the name) that takes the name of a file and a list of numbers on the command line and calculates, for each number given, the list of prime numbers smaller than it. This function creates a garbage collection trace in the indicated file. Trace commands from previous exercice are gathered in file `trgc.ml`
3. Compile these files and create a stand-alone executable; test it with the following call, and display the result.

   ```
                                                    %
   erart trace_rt 3000 4000 5000 6000
   ```

4. Do the same work for the non tail recursive function.
5. Compare trace results.

Summary

This chapter has presented the principal families of algorithms for automatic memory reclamation with the goal of detailing those used in Objective Caml. The Objective Caml garbage collector is an incremental garbage collector with two generations. It uses *Mark&Sweep* for the old generation, and *Stop&Copy* for the young generation. Two modules directly linked to the garbage collector allow control of the evolution of the heap. The `Gc` module allows analysis of the behavior of the garbage collector and modification of certain parameters with the goal of optimizing specific applications. With the `Weak` module one can save in arrays values that are potentially reclaimable, but which are still accessible. This module is useful for implementing a memory cache.

To Learn More

Memory reclamation techniques have been studied for forty years—in fact, since the first implementations of the Lisp programming language. For this reason, the literature in this area is enormous.

A comprehensive reference is Jones' book [Jon98]. Paul Wilson's tutorial [Wil92] is an excellent introduction to the field, with many references. The following web pages also provide a good view of the state of the art in memory management.

Link: ftp://ftp.netcom.com/pub/hb/hbaker/home.html

is an introduction to sequential garbage collectors.

Link: http://www.cs.ukc.ac.uk/people/staff/rej/gc.html

contains the presentation of [Jon98] and includes a large searchable bibliography.

Link: http://www.cs.colorado.edu/~zorn/DSA.html

lists different tools for debugging garbage collection.

Link: http://reality.sgi.com/boehm_mti/

offers C source code for a conservative garbage collector for the C language. This garbage collector replaces the classical allocator `malloc` by a specialized version `GC_malloc`. Explicit recovery by `free` is replaced by a new version that no longer does anything.

Link: http://www.harlequin.com/mm/reference/links.html

maintains a list of links on this subject.

In chapter 12 on the interface between C and Objective Caml we come back to memory management.

10

Program Analysis Tools

Program analysis tools provide supplementary information to the programmer in addition to the feedback from the compiler and the linker. Some of these tools perform a static analysis, i.e. they look at the code (either as text or in the form of a syntax tree) and determine certain properties like interdependency of modules or uncaught exceptions. Other tools perform a dynamic analysis, i.e. they look at the flow of execution. Analysis tools are useful for determining the number of calls to certain functions, getting a trace of the flow of arguments, or determining the time spent in certain parts of the program. Some are interactive, like the tools for debugging. In this case program execution is modified to account for user interaction. It is then possible to set breakpoints, in order to look at values or to restart program execution with different arguments.

The Objective Caml distribution includes such tools. Some of them have rather unusual characteristics, mostly dealing with static typing. It is, in fact, this static typing that guarantees the absence of type errors during program execution and enables the compiler to produce efficient code with a small memory footprint. Typing information is partly lost for constructed Objective Caml values. This creates certain difficulties, e.g. the impossibility of showing the arguments of polymorphic functions.

Chapter Overview

This short chapter presents the program analysis tools in the Objective Caml distribution. The first chapter describes the `ocamldep` command, which finds the dependencies in a set of Objective Caml files that make up an application.

The second section deals with debugging tools including tracing the execution of functions and the `ocamldebug` debugger, running under Unix.

The third section takes a look at the profiler, which can be used to analyze the execution of a program with an eye towards its optimization.

Dependency Analysis

Dependency analysis of a set of implementation and interface files that make up an Objective Caml application pursues a double end. The first is to get a global view of the interdependencies between modules. The second is to use this information in order to recompile only the absolutely necessary files after modifications of certain files.

The `ocamldep` command takes a set of `.ml` and `.mli` files and outputs the dependencies between files in `Makefile`[1] format.

These dependencies originate from global declarations in other modules, either by using dot.notation (e.g. `M1.f`) or by opening a module (e.g. **`open`** `M1`).

Suppose the following files exist:

dp.ml :
```
let print_vect v =
    for i = 0 to Array.length v do
      Printf.printf "%f " v.(i)
    done;
    print_newline();;
```

and d1.ml :
```
let init n e =
  let v = Array.create 4 3.14 in
    Dp.print_vect v;
    v;;
```

Given the name of these files, the `ocamldep` command will output the following dependencies:

```
$ ocamldep dp.ml d1.ml array.ml array.mli printf.ml printf.mli
dp.cmo: array.cmi printf.cmi
dp.cmx: array.cmx printf.cmx
d1.cmo: array.cmi dp.cmo
d1.cmx: array.cmx dp.cmx
array.cmo: array.cmi
array.cmx: array.cmi
printf.cmo: printf.cmi
printf.cmx: printf.cmi
```

1. `Makefile` files are used by the `make` command for the maintenance of a set of programs or files to keep everything up to date after modifications to some of them.

The dependencies are determined for both the bytecode and the native compiler. The output is to be read in the following manner: production of the file `dp.cmo` depends on the files `array.cmi` and `printf.cmi`. Files with the extension `.cmi` depend on files with the same name and extension `.mli`. And the same holds by analogy for `.ml` files with `.cmo` and `.cmx` files.

The object files of the distribution do not show up in the dependency lists. In fact, if `ocamldep` does not find the files `array.ml` and `printf.ml` in the current directory, it will find them in the library directory of the installation and produce the following output:

```
$ ocamldep dp.ml d1.ml
d1.cmo: dp.cmo
d1.cmx: dp.cmx
```

To give new file search paths to the `ocamldep` command, the `-I directory` option is used, which adds a directory to the list of include directories.

Debugging Tools

There are two *debugging* tools. The first is a *trace* mechanism that can be used on the global functions in the toplevel loop. The second tool is a *debugger* that is not used in the normal toplevel loop. After a first program run it is possible to go back to breakpoints, and to inspect values or to restart certain functions with different arguments. This second tool only runs under Unix, because it duplicates the running process via a `fork` (see page 582).

Trace

The *trace* of a function is the list of the values of its parameters together with its result in the course of a program run.

The trace commands are directives in the toplevel loop. They allow to trace a function, stop its trace or to stop all active traces. These three directives are shown in the table below.

#trace `name`	trace function `name`
#untrace `name`	stop tracing function `name`
#untrace_all	stop all traces

Here is a first example of the definition of a function `f`:

```
# let f x = x + 1;;
val f : int -> int = <fun>
# f 4;;
```

```
- : int = 5
```

Now we will trace this function, so that its arguments and its return value will be shown.

```
#  #trace f;;
f is now traced.
# f 4;;
f <-- 4
f --> 5
- : int = 5
```

Passing of the argument `4` to `f` is shown, then the function `f` calculates the desired value and the result is returned and also shown. The arguments of a function call are indicated by a left arrow and the return value by an arrow to the right.

Functions of Several Arguments

Functions of several arguments (or functions returning a closure) are also traceable. Each argument passed is shown. To distinguish the different closures, the number of arguments already passed to the closures is marked with a `*`. Let the function `verif_div` take 4 numbers (a, b, q, r) corresponding to the integer division: $a = bq + r$.

```
# let verif_div a b q r =
    a = b*q + r;;
val verif_div : int -> int -> int -> int -> bool = <fun>
# verif_div 11 5 2 1;;
- : bool = true
```

Its trace shows the passing of 4 arguments:

```
#  #trace verif_div;;
verif_div is now traced.
# verif_div 11 5 2 1;;
verif_div <-- 11
verif_div --> <fun>
verif_div* <-- 5
verif_div* --> <fun>
verif_div** <-- 2
verif_div** --> <fun>
verif_div*** <-- 1
verif_div*** --> true
- : bool = true
```

Recursive Functions

The trace gives valuable information about recursive functions, e.g. poor stopping criteria are easily detected.

Let the function `belongs_to` which tests whether an integer belongs to a list of integers be defined in the following manner:

```
# let rec belongs_to (e : int) l = match l with
     [] → false
  | t :: q → (e = t) || belongs_to e q ;;
val belongs_to : int -> int list -> bool = <fun>
# belongs_to 4 [3;5;7] ;;
- : bool = false
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
- : bool = true
```

The trace of the function invocation `belongs_to 4 [3;5;7]` will show the four calls of this function and the results returned.

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 4 [3;5;7] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
- : bool = false
```

At each call of the function `belongs_to` the argument `4` and the list to search in are passed as arguments. When the list becomes empty, the functions return `false` as a return value which is passed along to each waiting recursive invocation.

The following example shows the section of the list when the element searched for appears:

```
# belongs_to 4 [1; 2; 3; 4; 5; 6; 7; 8] ;;
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [1; 2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
```

```
belongs_to* <-- [2; 3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [3; 4; 5; 6; 7; 8]
belongs_to <-- 4
belongs_to --> <fun>
belongs_to* <-- [4; 5; 6; 7; 8]
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

As soon as 4 becomes head of the list, the functions return `true` which gets passed along to each waiting recursive invocation.

If the sequence of statements around `||` were changed, the function `belongs_to` would still return the right result but would always have to go over the complete list.

```
# let rec belongs_to (e : int) = function
      [] → false
    | t :: q →   belongs_to e q || (e = t) ;;
val belongs_to : int -> int list -> bool = <fun>
# #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [3;5;7] ;;
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [3; 5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [5; 7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- [7]
belongs_to <-- 3
belongs_to --> <fun>
belongs_to* <-- []
belongs_to* --> false
belongs_to* --> false
belongs_to* --> false
belongs_to* --> true
- : bool = true
```

Even though 3 is the first element of the list, it is traversed completely. So, trace also provides a mechanism for the efficiency analysis of recursive functions.

Polymorphic Functions

The trace does not show the value corresponding to an argument of a parameterized type. If for example the function `belongs_to` can be written without an explicit type

constraint:

```
# let rec belongs_to e l = match l with
    [] → false
  | t :: q → (e = t) || belongs_to e q ;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

The type of the function `belongs_to` is now polymorphic, and the trace does no longer show the value of its arguments but replaces them with the indication (`poly`).

```
#  #trace belongs_to ;;
belongs_to is now traced.
# belongs_to 3 [2;3;4] ;;
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>; <poly>]
belongs_to <-- <poly>
belongs_to --> <fun>
belongs_to* <-- [<poly>; <poly>]
belongs_to* --> true
belongs_to* --> true
- : bool = true
```

The Objective Caml toplevel loop can only show monomorphic types. Moreover, it only keeps the inferred types of global declarations. Therefore, after compilation of the expression `belongs_to 3 [2;3;4]`, the toplevel loop in fact no longer possesses any further type information about the function `belongs_to` apart form the type *`'a -> 'a list -> bool`*. The (monomorphic) types of `3` and `[2;3;4]` are lost, because the values do not keep any type information: this is static typing. This is the reason why the trace mechanism attributes the polymorphic types *`'a`* and *`'a list`* to the arguments of the function `belongs_to` and does not show their values.

It is this absence of typing information in values that entails the impossibility of constructing a generic `print` function of type *`'a -> unit`*.

Local Functions

Local functions cannot be traced for the same reasons as above, relating again to static typing. Only global type declarations are kept in the environment of the toplevel loop. Still the following programming style is common:

```
# let belongs_to e l =
    let rec bel_aux l = match l with
      [] → false
    | t :: q → (e = t) || (bel_aux q)
    in
      bel_aux l;;
val belongs_to : 'a -> 'a list -> bool = <fun>
```

The global function only calls on the local function, which does the interesting part of the work.

Notes on Tracing

Tracing is actually the only multi-platform debugging tool. Its two weaknesses are the absence of tracing information for local functions and the inability to show the value of polymorphic parameters. This strongly restricts its usage, mainly during the first steps with the language.

Debug

`ocamldebug`, is a *debugger* in the usual sense of the word. It permits step-by-step execution, the insertion of breakpoints and the inspection and modification of values in the environment.

Single-stepping a program presupposes the knowledge of what comprises a *program step*. In imperative programming this is an easy enough notion: a step corresponds (more or less) to a single instruction of the language. But this definition does not make much sense in functional programming; one instead speaks of program *events*. These are applications, entries to functions, pattern matching, a conditional, a loop, an element of a sequence, etc.

Warning This tool only runs under Unix.

Compiling with Debugging Mode

The `-g` compiler option produces a `.cmo` file that allows the generation of the necessary instructions for debugging. Only the bytecode compiler knows about this option. It is necessary to set this option during compilation of the files encompassing an application. Once the executable is produced, execution in *debug* mode can be accomplished with the following `ocamldebug` command:

`ocamldebug [`*options*`] executable [`*arguments*`]`

Take the following example file `fact.ml` which calculates the factorial function:

```
let fact n =
  let rec fact_aux p q n =
    if n = 0 then p
    else fact_aux (p+q) p (n-1)
  in
fact_aux 1 1 n;;
```

The main program in the file `main.ml` goes off on a long recursion after the call of `Fact.fact` on -1.

```
let x = ref 4;;
let go () =
  x := -1;
  Fact.fact !x;;
```

```
go();;
```

The two files are compiled with the `-g` option:

```
$ ocamlc -g -i -o fact.exe fact.ml main.ml
val fact : int -> int
val x : int ref
val go : unit -> int
```

Starting the Debugger

Once an executable is compiled with *debug* mode, it can be run in this mode.

```
$ ocamldebug fact.exe
        Objective Caml Debugger version 3.00

(ocd)
```

Execution Control

Execution control is done via program events. It is possible to go forward and backwards by n program events, or to go forward or backwards to the next breakpoint (or the nth breakpoint). A breakpoint can be set on a function or a program event. The choice of language element is shown by line and column number or the number of characters. This locality may be relative to a module.

In the example below, a breakpoint is set at the fourth line of module `Main`:

```
(ocd) step 0
Loading program... done.
Time : 0
Beginning of program.
(ocd)  break @ Main 4
Breakpoint 1 at 5028 : file Main, line 4 column 3
```

The initialisations of the module are done before the actual program. This is the reason the breakpoint at line 4 occurs only after 5028 instructions.

We go forward or backwards in the execution either by program elements or by breakpoints. `run` and `reverse` run the program just to the next breakpoint. The first in the direction of program execution, the second in the backwards direction. The `step` command advanced by 1 or n program elements, entering into functions, `next` steps over them. `backstep` and `previous` respectively do the same in the backwards direction. `finish` finally completes the current functions invocations, whereas `start` returns to the program element before the function invocation.

To continue our example, we go forward to the breakpoint and then execute three program instructions:

```
(ocd) run
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) step
Time : 7 - pc : 4860 - module Fact
2   <|b|>let rec fact_aux p q n =
(ocd) step
Time : 8 - pc : 4876 - module Fact
6 <|b|>fact_aux 1 1 n;;
(ocd) step
Time : 9 - pc : 4788 - module Fact
3     <|b|>if n = 0 then p
```

Inspection of Values

At a breakpoint, the values of variables in the activation record can be inspected. The `print` and `display` commands output the values associated with a variable according to the different depths.

We will print the value of `n`, then go back three steps to print the contents of `x`:

```
(ocd) print n
n : int = -1
(ocd) backstep 3
Time : 6 - pc : 4964 - module Main
Breakpoint : 1
4   <|b|>Fact.fact !x;;
(ocd) print x
x : int ref = {contents=-1}
```

Access to the fields of a record or via the index of an array is accepted by the printing commands.

```
(ocd) print x.contents
1 : int = -1
```

Execution Stack

The execution stack permits a visualization of the entanglement of function invocations. The `backtrace` or `bt` command shows the stack of calls. The `up` and `down` commands select the next or preceding activation record. Finally, the `frame` command gives a description of the current record.

Profiling

This tool allows measuring a variety of metrics concerning program execution, including how many times a particular function or control structure (including conditionals, pattern matchers and loops) are executed. The results are recorded in a file. By examining this information, you may be able to locate either algorithmic errors or crucial locations for optimization.

In order for the profiler to do its work, it is necessary to compile the code using a special mode that adds profiling instructions. There are two *profiling* modes: one for the bytecode compiler, and the other for the native-code compiler. There are also two commands used to analyze the results. Analysis of native code will retrieve the time spent in each function.

Profiling an application therefore proceeds in three stages:

1. compilation in *profiling* mode;
2. program execution;
3. presentation of measurements.

Compilation Commands

The commands to compile in *profiling* mode are the following:

- `ocamlcp -p` *options* for the bytecode compiler;
- `ocamlopt -p` *options* for the native-code compiler.

These compilers produce the same type of files as the usual commands (see chapter 7). The different options are described in figure 10.1.

`f`	function call
`i`	branch of **`if`**
`l`	**`while`** and **`for`** loops
`m`	branches of **`match`**
`t`	branches of **`try`**
`a`	all options

Figure 10.1: Options of the *profiling* commands

These indicate which control structures must be taken into account. By default, the `fm` options are activated.

Program Execution

Bytecode Compiler

The execution of a program compiled in profiling mode will, if it terminates, produce a file named `ocamlprof.dump` which contains the information wanted.

We resume the example of the product of a list of integers. We write the following file `f1.ml`:

```
let rec interval a b =
  if b < a then []
  else a :: (interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 let rec mult_rec l = match l with
     [] → 1
  | 0::_ → raise Found_zero
  | n :: x → n * (mult_rec x)
 in
  try mult_rec l with Found_zero → 0
;;
```

and the file `f2.ml` which uses the functions of `f1.ml`:

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

The compilation of these files in profiling mode is shown in the following:

```
ocamlcp -i -p a -c f1.ml
val profile_f1_ : int array
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
```

With the `-p` option, the compiler adds a new function (`profile_f1_`) for the initialization of the counters in module `F1`. It is the same for file `f2.ml`:

```
ocamlcp -i -p a -o f2.exe f1.cmo f2.ml
val profile_f2_ : int array
val l1 : int list
val l2 : int list
val l3 : int list
```

Native Compiler

The native code compilation gives the following result:

```
$ ocamlopt -i -p  -c f1.ml
val interval : int -> int -> int list
exception Found_zero
val mult_list : int list -> int
$ ocamlopt -i -p -o f2nat.exe f1.cmx f2.ml
```

Only the `-p` option without argument is used. The execution of `f2nat.exe` produces a file named `gmon.out` which is in a format that can be handled by the usual Unix commands (see page 284).

Presentation of the Results

Since the information gathered by the two *profiling* modes differs, their presentation follows suit. In the first (bytecode) mode comments on the number of passages through the control structures are added to the program text. In the second (native) mode, the time spent in its body and the number of calls is associated with each function.

Bytecode Compiler

The `ocamlprof` command gives the analysis of the measurement results. It uses the information contained in the file `camlprof.dump`. This command takes the source of the program on entry, then reads the measurements file and produces a new program text with the desired counts added as comments.

For our example this gives:

```
ocamlprof f1.ml

let rec interval a b =
  (* 62 *) if b < a then (* 2 *) []
  else (* 60 *) a::(interval (a+1) b);;

exception Found_zero ;;

let mult_list l =
 (* 2 *) let rec mult_rec l = (* 62 *) match l with
```

```
    [] -> (* 1 *) 1
  | 0::_ -> (* 1 *) raise Found_zero
  | n::x -> (* 60 *) n * (mult_rec x)
 in
  try mult_rec l with Found_zero -> (* 1 *) 0
;;
```

These counters reflect the calculations done in `F2` quite well. There are two calls of `mult_list` and 62 of the auxiliary function `mult_rec`. Examination of the different branches of the pattern matching show 60 passages through the common case, one through the pattern [] and the only match where the head is 0, raising an exception, which can be seen in the counter of the **`try`** statement.

The `ocamlprof` command accepts two options. The first *-f file* indicates the name of the file to contain the measurements. The second `-F string` specifies a string to add to the comments associated with the control structures treated.

Native Compilation

To get the time spent in the calls of the functions for multiplying the elements of a list, we write the following file `f3.ml`:

```
let l1 = F1.interval 1 30;;
let l2 = F1.interval 31 60;;
let l3 = l1 @ (0 :: l2);;

for i=0 to 100000 do
  F1.mult_list l1;
  F1.mult_list l3
done;;

print_int (F1.mult_list l1);;
print_newline();;

print_int (F1.mult_list l3);;
print_newline();;
```

This is the same file as `f2.ml` with a loop of 100000 iterations.

Execution of the program creates the file `gmon.out`. This is in a format readable by `gprof`, a command that can be found on Unix systems. The following call to `gprof` prints information about the time spent and the call graph. Since the output is rather long, we show only the first page which contains the name of the functions that are called at least once and the time spent in each.

```
$ gprof  f3nat.exe
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls  us/call  us/call  name
```

```
 92.31      0.36     0.36   200004     1.80     1.80  F1_mult_rec_45
  7.69      0.39     0.03   200004     0.15     1.95  F1_mult_list_43
  0.00      0.39     0.00     2690     0.00     0.00  oldify
  0.00      0.39     0.00      302     0.00     0.00  darken
  0.00      0.39     0.00      188     0.00     0.00  gc_message
  0.00      0.39     0.00      174     0.00     0.00  aligned_malloc
  0.00      0.39     0.00      173     0.00     0.00  alloc_shr
  0.00      0.39     0.00      173     0.00     0.00  fl_allocate
  0.00      0.39     0.00       34     0.00     0.00  caml_alloc3
  0.00      0.39     0.00       30     0.00     0.00  caml_call_gc
  0.00      0.39     0.00       30     0.00     0.00  garbage_collection
...
```

The main lesson is that almost all of the execution time is spent in the function `F1_mult_rec_45`, which corresponds to the function `F1.mult_rec` in file `f1.ml`. On the other hand we recognize a lot of other functions that are called. The first on the list are memory management functions in the runtime library (see chapter 9).

Exercises

Tracing Function Application

This exercise shows the evaluation of arguments at the moment of function application.

1. Activate tracing of the function `List.fold_left` and evaluate the following expression:
   `List.fold_left (-) 1 [2; 3; 4; 5];;`
   What does the trace show you?
2. Define the function `fold_left_int`, identical to `List.fold_left`, but with type: $(int \rightarrow int \rightarrow int) \rightarrow int \rightarrow int\ list \rightarrow int$.
   Trace this function. Why is the output of the trace different?

Performance Analysis

We continue the exercise proposed in chapter 9, page 247, where we compared the evolution of the heap of two programs (one tail recursive and the other not) for calculating primes. This time we will compare the execution times of each function with the *profiling* tools. This exercise shows the importance of inline expansion (see chapter 7).

1. Compile the two programs `erart` and `eranrt` with *profiling* options using the bytecode compiler and the native code compiler respectively.
2. Execute the programs passing them the numbers 3000 4000 5000 6000 on the command line.
3. Visualize the results with the `ocamlprof` and `gprof` commands. What can you say about the results?

Summary

This chapter presented the different programming support tools that come with the Objective Caml distribution.

The first tool performs a static analysis in order to determine the dependencies of a set of compilation units. This information can then be put in a `Makefile`, allowing for separate compilation (if you alter one source file in a program, you only have to compile that file, and the files that have dependencies to it, rather than the entire program).

Other tools give information about the execution of a program. The interactive toplevel offers a trace of the execution; but, as we have seen, polymorphism imposes quite heavy restrictions on the observable values. In fact, only the global declarations of monomorphic values are visible, which nevertheless includes the arguments of monomorphic functions and permits tracing of recursive functions.

The last tools are those in the tradition of development under Unix, namely a *debugger* and a *profiler*. With the first, you can execute a program step by step to examine it's operation, and the second gives information about its performance. Both are usable only under Unix.

To Learn More

The results produced by the `ocamldep` command can be visualized in graphical form by the `ocamldot` utility, which can be found on the following page:

Link: http://www.cis.upenn.edu/~tjim/ocamldot/index.html

`ocamldot` makes use of an independent program (`dot`), also downloadable:

Link: http://www.research.att.com/sw/tools/graphviz/

Several generic `Makefile` templates for Objective Caml have been proposed to ease the burden of project management:

Link: http://caml.inria.fr/FAQ/Makefile_ocaml-eng.html

Link: http://www.ai.univie.ac.at/~markus/ocaml_sources

These integrate the output of `ocamldep`.

In [HF+96] a performance evaluation of about twenty implementations of functional languages, among them several ML implementations, can be found. The benchmark is an example of numerical calculations on large datastructures.

11

Tools for lexical analysis and parsing

The development of lexical analysis and parsing tools has been an important area of research in computer science. This work has produced the lexer and parser generators `lex` and `yacc` whose worthy scions `camllex` and `camlyacc` are presented in this chapter. These two tools are the de-facto standard for implementing lexers and parsers, but there are other tools, like streams or the *regular expression* library `str`, that may be adequate for applications which do not need a powerful analysis.

The need for such tools is especially acute in the area of state-of-the-art programming languages, but other applications can profit from such tools: for example, database systems offering the possibility of issuing queries, or spreadsheets defining the contents of cells as the result of the evaluation of a formula. More modestly, it is common to use plain text files to store data; for example system configuration files or spreadsheet data. Even in such limited cases, processing the data involves some amount of lexical analysis and parsing.

In all of these examples the problem that lexical analysis and parsing must solve is that of transforming a linear character stream into a data item with a richer structure: a string of words, a record structure, the abstract syntax tree for a program, etc.

All languages have a set of vocabulary items (lexicon) and a grammar describing how such items may be combined to form larger items (syntax). For a computer or program to be able to correctly process a language, it must obey precise lexical and syntactic rules. A computer does not have the detailed semantic understanding required to resolve ambiguities in natural language. To work around the limitation, computer languages typically obey clearly stated rules without exceptions. The lexical and syntactic structure of such languages has received formal definitions that we briefly introduce in this chapter before introducing their uses.

Chapter Structure

This chapter introduces the tools of the Objective Caml distribution for lexical analysis and parsing. The latter normally supposes that the former has already taken place. In the first section, we introduce a simple tool for lexical analysis provided by module `Genlex`. Next we give details about the definition of sets of lexical units by introducing the formalism of *regular expressions*. We illustrate their behavior within module `Str` and the `ocamllex` tool. In section two we define grammars and give details about sentence production rules for a language to introduce two types of parsing: bottom-up and top-down. They are further illustrated by using *`Stream`* and the `ocamlyacc` tool. These examples use context-free grammars. We then show how to carry out contextual analysis with *`Streams`*. In the third section we go back to the example of a BASIC interpreter from page 159, using `ocamllex` and `ocamlyacc` to implement the lexical analysis and parsing functions.

Lexicon

Lexical analysis is the first step in character string processing: it segments character strings into a sequence of words also known as *lexical units* or *lexemes*.

Module `Genlex`

This module provides a simple primitive allowing the analysis of a string of characters using several categories of predefined lexical units. These categories are distinguished by type:

```
# type token =
    Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char ;;
```

Hence, we will be able to recognize within a character string an integer (constructor `Int`) and to recover its value (constructor argument of type *`int`*). Recognizable strings and characters respect the usual conventions: a string is delimited by two (`"`) characters and character literals by two (`'`) characters. A float is represented by using either floating-point notation (for example `0.01`) or exponent-mantissa notation (for example `1E-2`).

Constructor `Ident` designates the category of *identifiers*. These are the names of variables or functions in programming languages, for example. They comprise all strings

of letters and digits including underscore (_) or apostrophe (’). Such a string should not start with a digit. We also consider as identifiers (for this module at least) strings containing operator symbols, such as `+`, `*`, `>` or `=`. Finally, constructor `Kwd` defines the category of keywords containing distinguished identifiers or special characters (specified by the programmer when invoking the lexer).

The only variant of the token type controlled by parameters is that of keywords. The following primitive allows us to create a lexical analyser (lexer) taking as keywords the list passed as first argument to it.

```
# Genlex.make_lexer ;;
- : string list -> char Stream.t -> Genlex.token Stream.t = <fun>
```

The result of applying `make_lexer` to a list of keywords is a function taking as input a stream of characters and returning a stream of lexical units (of type *token*.)

Thus we can easily obtain a lexer for our BASIC interpreter. We declare the set of keywords:

```
# let keywords =
    [ "REM"; "GOTO"; "LET"; "PRINT"; "INPUT"; "IF"; "THEN";
      "-"; "!"; "+"; "-"; "*"; "/"; "%";
      "="; "<"; ">"; "<="; ">="; "<>";
      "&"; "|" ] ;;
```

With this definition in place, we define the lexer:

```
# let line_lexer l = Genlex.make_lexer keywords (Stream.of_string l) ;;
val line_lexer : string -> Genlex.token Stream.t = <fun>
# line_lexer "LET x = x + y * 3" ;;
- : Genlex.token Stream.t = <abstr>
```

Function `line_lexer` takes as input a string of characters and returns the corresponding stream of lexemes.

Use of Streams

We can carry out the lexical analysis “by hand” by directly manipulating streams.

The following example is a lexer for arithmetical expressions. Function `lexer` takes a character stream and returns a stream of lexical units of type *lexeme Stream.t*[1].

1. Type *lexeme* is defined on page 163

Spaces, tabs and newline characters are removed. To simplify, we do not consider variables or negative integers.

```
# let rec spaces s =
    match s with parser
      [<'' ' ; rest >] → spaces rest
    | [<''\t' ; rest >] → spaces rest
    | [<''\n' ; rest >] → spaces rest
    | [<>] → ();;
val spaces : char Stream.t -> unit = <fun>
# let rec lexer s =
    spaces s;
    match s with parser
      [< ''(' >] → [< 'Lsymbol "(" ; lexer s >]
    | [< '')' >] → [< 'Lsymbol ")" ; lexer s >]
    | [< ''+' >] → [< 'Lsymbol "+" ; lexer s >]
    | [< ''-' >] → [< 'Lsymbol "-" ; lexer s >]
    | [< ''*' >] → [< 'Lsymbol "*" ; lexer s >]
    | [< ''/' >] → [< 'Lsymbol "/" ; lexer s >]
    | [< ''0'..'9' as c;
         i,v = lexint (Char.code c - Char.code('0')) >]
       → [<'Lint i ; lexer v>]
  and lexint r s =
    match s with parser
      [< ''0'..'9' as c >]
      → let u = (Char.code c) - (Char.code '0') in  lexint (10*r + u) s
    | [<>] → r,s
  ;;
val lexer : char Stream.t -> lexeme Stream.t = <fun>
val lexint : int -> char Stream.t -> int * char Stream.t = <fun>
```

Function `lexint` carries out the lexical analysis for the portion of a stream describing an integer constant. It is called by function `lexer` when `lexer` finds a digit on the input stream. Function `lexint` then consumes all consecutive digits to obtain the corresponding integer value.

Regular Expressions

[2]

Let's abstract a bit and consider the problem of lexical units from a more theoretical point of view.

2. Note of translators: From an academic standpoint, the proper term would have been *"Rational Expressions"*; we chose the term "regular" to follow the programmers' tradition.

From this point of view, a lexical unit is a *word*. A word is formed by concatening items in an *alphabet*. For our purposes, the alphabet we are considering is a subset of the ASCII characters. Theoretically, a word may contain no characters (the *empty word*[3]) or just a single character. The theoretical study of the assembly of lexical items (lexemes) from members of an alphabet has brought about a simple formalism known as *regular expressions*.

Definition A regular expression defines a set of words. For example, a regular expression could specify the set of words that are valid identifiers. Regular expressions are specified by a few set-theoretic operations. Let M and N be two sets of words. Then we can specify:

1. the union of M and N, denoted by M `|` N.
2. the complement of M, denoted by `^`M. This is the set of all words not in M.
3. the concatenation of M and N. This is the set of all the words formed by placing a word from M before a word from N. We denote this set simply by MN.
4. the set of words formed by a finite sequence of words in M, denoted M`+`.
5. for syntactic convenience, we write M`?` to denote the set of words in M, with addition of the empty word.

Individual characters denote the singleton set of words containing just that character. Expression `a | b | c` thus describes the set containing three words: `a`, `b` ant `c`. We will use the more compact syntax `[abc]` to define such a set. As our alphabet is ordered (by the ASCII code order) we can also define intervals. For example, the set of digits can be written: `[0-9]`. We can use parentheses to group expressions.

If we want to use one of the operator characters as a character in a regular expression, it should be preceded by the escape character `\`. For example, `(\*)*` denotes the set of sequences of stars.

Example Let's consider the alphabet comprising digits (`0, 1, 2, 3, 4, 5, 6, 7, 8, 9`) the plus (`+`), minus (`-`) and dot (`.`) signs and letter `E`. We can define the set *num* of words denoting numbers. Let's call *integers* the set defined with `[0-9]+`. We define the set *unum* of unsigned numbers as:

$$integers\texttt{?(.}integers\texttt{)?(E(\+|-)?}integers\texttt{)?}$$

The set of signed numbers is thus defined as:

$$unum \texttt{ | -}unum \text{ or with } -?unum$$

Recognition While regular expressions are a useful formalism in their own right, we usually wish to implement a program that determines whether a string of characters (or

3. By convention, the empty word is denoted by the greek character epsilon: ϵ

one of its substrings) is a member of the set of words described by a regular expression. For that we need to translate the formal definition of the set into a recognition and expression processing program. In the case of regular expressions such a translation can be automated. Such translation techniques are carried out by module `Genlex` in library `Str` (described in the next section) and by the `ocamllex` tools that we introduce in the following two sections.

The `Str` *Library*

This module contains an abstract data type *`regexp`* which represents regular expressions as well as a function `regexp` which takes a string describing a regular expression, more or less following the syntax described above, and returns its abstract representation.

This module contains, as well, a number of functions which exploit regular expressions and manipulate strings. The syntax of regular expressions for library `Str` is given in figure 11.1.

`.`	any character except `\n`
`*`	zero or more occurences of the preceding expression
`+`	one or more occurences of the preceding expression
`?`	zero or one occurences of the preceding expression
`[..]`	set of characters (example `[abc]`)
	intervals, denoted by `-` (example `[0-9]`)
	set complements, denoted by `^` (example `[^A-Z]`)
`^`	start of line (not to be mistaken with the use of `^` as a set complement)
`$`	end of line
`|`	alternative
`(..)`	grouping of a complex expression (we can later refer to such an expression by an integer index – see below)
i	an integer constant, referring to the string matched by the i-th complex expression
`\`	escape character (used when matching a reserved character in regular expressions)

Figure 11.1: Regular expressions.

Example We want to write a function translating dates in anglo-saxon format into French dates within a data file. We suppose that the file is organised into lines of data fields and the components of an anglo-saxon date are separated by dots. Let's define a function which takes as argument a string (*i.e.* a line from the file), isolates the date, decomposes and translates it, then replaces the original with the translation.

```
# let french_date_of d =
    match d with
     [mm; dd; yy] → dd^"/"^mm^"/"^yy
    | _ → failwith "Bad date format" ;;
val french_date_of : string list -> string = <fun>

# let english_date_format = Str.regexp "[0-9]+\.[0-9]+\.[0-9]+" ;;
val english_date_format : Str.regexp = <abstr>

# let trans_date l =
    try
     let i=Str.search_forward english_date_format l 0 in
     let d1 = Str.matched_string l in
     let d2 = french_date_of (Str.split (Str.regexp "\.") d1) in
       Str.global_replace english_date_format d2 l
    with Not_found → l ;;
val trans_date : string -> string = <fun>

# trans_date ".............06.13.99............" ;;
- : string = ".............13/06/99............"
```

The `ocamllex` *Tool*

The `ocamllex` tool is a lexical analyzer generator built for Objective Caml after the model of the `lex` tool for the C language. It generates a source Objective Caml file from a file describing the lexical elements to be recognized in the form of regular expressions. The programmer can augment each lexical element description with a processing action known as a *semantic action*. The generated code manipulates an abstract type ***`lexbuf`*** defined in module `Lexing`. The programmer can use this module to control processing of lexical buffers.

Usually the lexical description files are given the extension `.mll`. Later, to obtain a Objective Caml source from a `lex_file.mll` you type the command

```
ocamllex lex_file.mll
```

A file `lex_file.ml` is generated containing the code for the corresponding analyzer. This file can then be compiled with other modules of an Objective Caml application. For each set of lexical analysis rules there is a corresponding function taking as input a lexical *buffer* (of type ***`Lexing.lexbuf`***) and returning the value defined by the semantic actions. Consequently, all actions in the same rule must produce values of the same type.

The general format for an `ocamllex` file is

```
{
```

```
  header
}

let ident = regexp
  ...
rule ruleset1 = parse
       regexp { action }
     | ...
     | regexp { action }
and ruleset2 = parse
     ...
and ...
{

  trailer-and-end
}
```

Both section "header" and "trailer-and-end" are optional. They contain Objective Caml code defining types, functions, etc. needed for processing. The code in the last section can use the lexical analysis functions that will be generated by the middle section. The declaration list preceding the rule definition allows the user to give names to some regular expressions. They can later be invoked by name in the definition of rules.

Example Let's revisit our BASIC example. We will want to refine the type of lexical units returned. We will once again define function `lexer` (as we did on page 163) with the same type of output (*`lexeme`*), but taking as input a *buffer* of type *`Lexing.lexbuf`*.

```
{
 let string_chars s =
  String.sub s 1 ((String.length s)-2) ;;
}

let op_ar = ['-' '+' '*' '%' '/']
let op_bool = ['!' '&' '|']
let rel = ['=' '<' '>']

rule lexer = parse
   [' ']   { lexer lexbuf }

 | op_ar   { Lsymbol (Lexing.lexeme lexbuf) }
 | op_bool { Lsymbol (Lexing.lexeme lexbuf) }

 | "<="    { Lsymbol (Lexing.lexeme lexbuf) }
 | ">="    { Lsymbol (Lexing.lexeme lexbuf) }
 | "<>"    { Lsymbol (Lexing.lexeme lexbuf) }
 | rel     { Lsymbol (Lexing.lexeme lexbuf) }
```

```
| "REM"   { Lsymbol (Lexing.lexeme lexbuf) }
| "LET"   { Lsymbol (Lexing.lexeme lexbuf) }
| "PRINT" { Lsymbol (Lexing.lexeme lexbuf) }
| "INPUT" { Lsymbol (Lexing.lexeme lexbuf) }
| "IF"    { Lsymbol (Lexing.lexeme lexbuf) }
| "THEN"  { Lsymbol (Lexing.lexeme lexbuf) }

| '-'? ['0'-'9']+   { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+        { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'  { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

The translation of this file by `ocamllex` returns function `lexer` of type *`Lexing.lexbuf -> lexeme`*. We will see later how to use such a function in conjunction with syntactic analysis (see page 305).

Syntax

Thanks to lexical analysis, we can split up input streams into more structured units: lexical units. We still need to know how to assemble these units so that they amount to syntactically correct sentences in a given language. The syntactic assembly rules are defined by *grammar rules*. This formalism was originally developed in the field of linguistics, and has proven immensely useful to language-theoretical mathematicians and computer scientists in that field. We have already seen on page 160 an instance of a grammar for the Basic language. We will resume this example to introduce the basic concepts for grammars.

Grammar

Formally, a grammar is made up of four elements:

1. a set of symbols called *terminals*. Such symbols represent the lexical units of the language. In Basic, the lexical units (terminals) are: the operator- and arithmetical and logical relation-symbols (`+`, `&`, `<`, `<=`, ..), the keywords of the language (`GOTO`, `PRINT`, `IF`, `THEN`, ..), integers (*integer* units) and variables (*variable* units).
2. A set of symbols called *non-terminals*. Such symbols stand for syntactic terms of the language. For example, a Basic program is composed of lines (and thus we have the term Line), a line may contain and Expression, etc.
3. A set of so-called *production* rules. These describe how terminal and non-terminals symbols may be combined to produce a syntactic term. A Basic line is made up of a number followed by an instruction. This is expressed in the following rule:

   Line ::= *integer* Instruction

   For any given term, there may be several alternative ways to form that term. We separate the alternatives with the symbol — as in

INSTRUCTION	::=	`LET` *variable* `=` EXPRESSION
	—	`GOTO` *integer*
	—	`PRINT` EXPRESSION

etc.

4. Finally, we designate a particular non-terminal as the *start symbol*. The start symbol identifies a complete translation unit (program) in our language, and the corresponding production rule is used as the starting point for parsing.

Production and Recognition

Production rules allow *recognition* that a sequence of lexemes belongs to a particular language.

Let's consider, for instance, a simple language for arithmetic expressions:

EXP	::=	*integer*	(R1)
	—	EXP `+` EXP	(R2)
	—	EXP `*` EXP	(R3)
	—	`(` EXP `)`	(R4)

where (R1) (R2) (R3) and (R4) are the names given to our rules. After lexical analysis, the expression `1*(2+3)` becomes the sequence of lexemes:

integer `*` `(` *integer* `+` *integer* `)`

To analyze this sentence and recognize that it really belongs to the language of arithmetic expressions, we are going to use the rules from right to left: if a subexpression matches the right-side member of a rule, we replace it with the corresponding left-side member and we re-run the process until reducing the expression to the non-terminal *start* (here EXP). Here are the stages of such an analysis[4]:

integer `*` `(` *integer* `+` *integer* `)`	$\xleftarrow{(R1)}$	EXP `*` `(` *integer* `+` *integer* `)`
	$\xleftarrow{(R1)}$	EXP `*` `(` EXP `+` *integer* `)`
	$\xleftarrow{(R1)}$	EXP `*` `(` EXP `+` EXP `)`
	$\xleftarrow{(R2)}$	EXP `*` `(` EXP `)`
	$\xleftarrow{(R4)}$	EXP `*` EXP
	$\xleftarrow{(R3)}$	EXP

Starting from the last line containing only EXP and following the arrows upwards we read how our expression could be produced from the start rule EXP: therefore it is a well-formed sentence of the language defined by the grammar.

4. We underline the portion of input processed at each stage and we point out the rule used.

The translation of grammars into programs capable of recognizing that a sequence of lexemes belongs to the language defined by a grammar is a much more complex problem than that of using regular expressions. Indeed, a mathematical result tells us that all sets (of words) defined by means of a regular expression formalism can also be defined by another formalism: *deterministic finite automata*. And these latter are easy to explain as programs taking as input a sequence of characters. We do not have a similar result for the case of generic grammars. However, we have weaker results establishing the equivalence between certain classes of grammars and somewhat richer automata: *pushdown automata*. We do not want to enter into the details of such results, nor give an exact definition of what an automaton is. Still, we need to identify a class of grammars that may be used with parser-generating tools or parsed directly.

Top-down Parsing

The analysis of the expresion `1*(2+3)` introduced in the previous paragraph is not unique: it could also have started by reducing *integers* from right to left, which would have permitted rule (R2) to reduce `2+3` from the beginning instead. These two ways to proceed constitute two types of analysis: top-down parsing (right-to-left) and bottom-up parsing (left-to-right). The latter is easily realizable with lexeme streams using module `Stream`. Bottom-up parsing is that carried-out by the `ocamlyacc` tool. It uses an explicit stack mechanism like the one already described for the parsing of Basic programs. The choice of parsing type is significant, as top-down analysis may or may not be possible given the form of the grammar used to specify the language.

A Simple Case

The canonical example for top-down parsing is the prefix notation of arithmetic expressions defined by:

$$\begin{array}{lcl} \textsc{Expr} & ::= & \textit{integer} \\ & — & \texttt{+}\ \textsc{Expr}\ \textsc{Expr} \\ & — & \texttt{*}\ \textsc{Expr}\ \textsc{Expr} \end{array}$$

In this case, knowing the first lexeme is enough to decide which production rule can be used. This immediate predictability obviates managing the parse stack explicitly by instead using the stack of recursive calls in the parser. Therefore, it is very easy to write a program implementing top-down analysis using the features in modules `Genlex` and `Stream`. Function `infix_of` is an example; it takes a prefix expression and returns its equivalent infix expression.

```
# let lexer s =
    let ll = Genlex.make_lexer ["+";"*"]
    in  ll (Stream.of_string s) ;;
val lexer : string -> Genlex.token Stream.t = <fun>
# let rec stream_parse s =
```

```
    match s with parser
      [<'Genlex.Ident x>] → x
    | [<'Genlex.Int n>] → string_of_int n
    | [<'Genlex.Kwd "+"; e1=stream_parse; e2=stream_parse>] → "("^e1^"+"^e2^")"
    | [<'Genlex.Kwd "*"; e1=stream_parse; e2=stream_parse>] → "("^e1^"*"^e2^")"
    | [<>] → failwith "Parse error"
  ;;
val stream_parse : Genlex.token Stream.t -> string = <fun>
# let infix_of s = stream_parse (lexer s) ;;
val infix_of : string -> string = <fun>
# infix_of "* +3 11 22";;
- : string = "((3+11)*22)"
```

One has to be careful, because this parser is rather unforgiving. It is advisable to introduce a blank between lexical units in the input string systematically.

```
# infix_of "*+3 11 22";;
- : string = "*+"
```

A Less Simple Case

Parsing using streams is predictive. It imposes two conditions on grammars.

1. There must be no *left-recursive* rules in the grammar. A rule is left-recursive when a right-hand expression starts with a non-terminal which is the left-hand member of the expression, as in Exp ::= Exp + Exp;
2. No two rules may start with the same expression.

The usual grammar for arithmetical expressions on page 296 is not directly suitable for top-down analysis: it does not satisfy any of the above-stated criteria. To be able to use top-down parsing, we must reformulate the grammar so as to suppress left-recursion and non-determinism in the rules. For arithmetic expressions, we may use, for instance:

Expr	::=	Atom NextExpr
NextExpr	::=	+ Atom
	—	- Atom
	—	* Atom
	—	/ Atom
	—	ϵ
Atom	::=	*integer*
	—	(Expr)

Note that the use of the empty word ϵ in the definition of NextExpr is compulsory if we want a single integer to be an expression.

Our grammar allows the implementation of the following parser which is a simple translation of the production rules. This parser produces the abstract syntax tree of arithmetic expressions.

```
# let rec rest = parser
      [< 'Lsymbol "+"; e2 = atom >]  → Some (PLUS,e2)
    | [< 'Lsymbol "-"; e2 = atom >]  → Some (MINUS,e2)
    | [< 'Lsymbol "*"; e2 = atom >]  → Some (MULT,e2)
    | [< 'Lsymbol "/"; e2 = atom >]  → Some (DIV,e2)
    | [< >] → None
  and atom = parser
      [< 'Lint i >] → ExpInt i
    | [< 'Lsymbol "("; e = expr ; 'Lsymbol ")" >] → e
  and expr s =
    match s with parser
      [< e1 = atom >] →
        match rest s with
            None → e1
          | Some (op,e2) → ExpBin(e1,op,e2) ;;
val rest : lexeme Stream.t -> (bin_op * expression) option = <fun>
val atom : lexeme Stream.t -> expression = <fun>
val expr : lexeme Stream.t -> expression = <fun>
```

The problem with using top-down parsing is that it forces us to use a grammar which is very restricted in its form. Moreover, when the object language is naturally described with a left-recursive grammar (as in the case of infix expressions) it is not always trivial to find an equivalent grammar (*i.e.* one defining the same language) that satisfies the requirements of top-down parsing. This is the reason why tools such as `yacc` and `ocamlyacc` use a bottom-up parsing mechanism which allows the definition of more natural-looking grammars. We will see, however, that not everything is possible with them, either.

Bottom-up Parsing

On page 165, we introduced intuitively the actions of bottom-up parsing: *shift* and *reduce*. With each of these actions the state of the stack is modified. We can deduce from this sequence of actions the grammar rules, provided the grammar allows it, as in the case of top-down parsing. Here, also, the difficulty lies in the non-determinism of the rules which prevents choosing between shifting and reducing. We are going to illustrate the inner workings of bottom-up parsing and its failures by considering those pervasive arithmetic expressions in postfix and prefix notation.

The Good News The simplified grammar for postfix arithmetic expressions is:

Expr	::=	*integer*	(R1)
	|	Expr Expr `+`	(R2)
	|	Expr Expr `*`	(R3)

This grammar is dual to that of prefix expressions: it is necessary to wait until the end of each analysis to know which rule has been used, but then one knows exactly what to do. In fact, the bottom-up analysis of such expressions resembles quite closely a stack-based evaluation mechanism. Instead of pushing the results of each calculation, we simply push the grammar symbols. The idea is to start with an empty stack, then obtain a stack which contains only the start symbol once the input is used up. The modifications to the stack are the following: when we shift, we push the present non-terminal; if we may reduce, it is because the first elements in the stack match the right-hand member of a rule (in reverse order), in which case we replace these elements by the corresponding left-hand non-terminal.

Figure 11.2 illustrates how bottom-up parsing processes expression: `1 2 + 3 * 4 +`. The input lexical unit is underlined. The end of input is noted with a `$` sign.

Action	Input	Stack
	1`2+3*4+$`	[]
Shift		
	2`+3*4+$`	[1]
Reduce (R1)		
	2`+3*4+$`	[Expr]
Shift		
	+`3*4+$`	[2Expr]
Reduce (R1)		
	+`3*4+$`	[Expr Expr]
Shift, Reduce (R2)		
	3`*4+$`	[Expr]
Shift, Reduce (R1)		
	*`4+$`	[Expr Expr]
Shift, Reduce (R3)		
	4`+$`	[Expr]
Shift, Reduce (R1)		
	+`$`	[Expr Expr]
Shift, Reduce (R2)		
	`$`	[Expr]

Figure 11.2: Bottom-up parsing example.

The Bad News The difficulty of migrating the grammar into the recognition program is determining which type of action to apply. We will illustrate this difficulty with three examples which generate three types of indeterminacy.

The first example is a grammar for expressions using only addition:

$$\begin{array}{lcll} \text{E0} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E0 + E0} & \text{(R2)} \end{array}$$

The indeterminacy in this grammar stems from rule (R2). Let's suppose the following situation:

Action	Input	Stack
⋮		
	$\underline{+}\ldots$	[E0 + E0 ...]
⋮		

In such a case, it is impossible to determine whether we have to shift and push the + or to reduce using (R2) both E0's and the + in the stack. We are in the presence of a shift-reduce conflict. This is because expression *integer* + *integer* + *integer* can be produced in two ways by right-derivation.

First way:
$$\begin{array}{lcl} \text{E0} & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R1)} & \underline{\text{E0}} \text{ + } \textit{integer} \\ & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \text{ + } \textit{integer} \end{array}$$
etc.

Second way:
$$\begin{array}{lcl} \text{E0} & \xrightarrow{(R2)} & \text{E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R2)} & \text{E0 + E0 + } \underline{\text{E0}} \\ & \xrightarrow{(R1)} & \text{E0 + } \underline{\text{E0}} \text{ + } \textit{integer} \end{array}$$
etc.

The expressions obtained by each derivation may look similar from the point of view of expression evaluation:

(*integer* + *integer*) + *integer* and *integer* + (*integer* + *integer*)

but different for building a syntax tree (see figure 6.3 on page 166).

The second instance of a grammar generating a conflict between shifting and reducing has the same type of ambiguity: an implicit parenthesizing. But contrary to the previous case, the choice between shifting and reducing modifies the meaning of the parsed expression. Let's consider the following grammar:

$$\begin{array}{lcll} \text{E1} & ::= & \textit{integer} & \text{(R1)} \\ & | & \text{E1 + E1} & \text{(R2)} \\ & | & \text{E1 * E1} & \text{(R3)} \end{array}$$

We find in this grammar the above-mentioned conflict both for + and for *. But there is an added conflict between + and *. Here again, an expression may be produced in two ways. There are two right-hand derivations of

$$\textit{integer} + \textit{integer} * \textit{integer}$$

$$\begin{array}{lll} \text{First way:} \quad \text{E1} & \xrightarrow{(R3)} & \text{E1} * \underline{\text{E1}} \\ & \xrightarrow{(R1)} & \underline{\text{E1}} * \textit{integer} \\ & \xrightarrow{(R2)} & \text{E1} + \underline{\text{E1}} * \textit{integer} \\ & \text{etc.} & \end{array}$$

$$\begin{array}{lll} \text{Second way:} \quad \text{E1} & \xrightarrow{(R2)} & \text{E1} + \underline{\text{E1}} \\ & \xrightarrow{(R3)} & \text{E1} + \text{E1} * \underline{\text{E1}} \\ & \xrightarrow{(R1)} & \text{E1} + \underline{\text{E1}} * \textit{integer} \\ & \text{etc.} & \end{array}$$

Here both pairs of parenthesis (implicit) are not equivalent:

$$(\textit{integer} + \textit{integer}) * \textit{integer} \neq \textit{integer} + (\textit{integer} * \textit{integer})$$

This problem has already been cited for Basic expressions (see page 165). It was solved by attributing different precedence to each operator: we reduce (R3) before (R2), which is equivalent to parenthesizing products.

We can also solve the problem of choosing between + and * by modifying the grammar. We introduce two new terminals: T (for terms), and F (for factors), which gives:

$$\begin{array}{lcll} \text{E} & ::= & \text{E + T} & \text{(R1)} \\ & | & \text{T} & \text{(R2)} \\ \text{T} & ::= & \text{T * F} & \text{(R3)} \\ & | & \text{F} & \text{(R4)} \\ \text{F} & ::= & \textit{integer} & \text{(R5)} \end{array}$$

There is now but a single way to reach the production sequence *integer* + *integer* * *integer*: using rule (R1).

The third example concerns conditional instructions in programming languages. A language such as Pascal offers two conditionals : `if .. then` and `if .. then .. else`. Let's imagine the following grammar:

$$\begin{array}{lcll} \textsc{Instr} & ::= & \texttt{if } \textsc{Exp} \texttt{ then } \textsc{Instr} & \text{(R1)} \\ & — & \texttt{if } \textsc{Exp} \texttt{ then } \textsc{Instr} \texttt{ else } \textsc{Instr} & \text{(R2)} \\ & — & \text{etc...} & \end{array}$$

In the following situation:

ACTION	INPUT	STACK
⋮		
	`else`...	[INSTR `then` EXP `if`...]
⋮		

We cannot decide whether the first elements in the stack relate to conditional (R1), in which case it must be reduced, or to the first INSTR in rule (R2), in which case it must be shifted.

Besides shift-reduce conflicts, bottom-up parsing may also generate reduce-reduce conflicts.

We now introduce the `ocamlyacc` tool which uses the bottom-up parsing technique and may find these conflicts.

The `ocamlyacc` *Tool*

The `ocamlyacc` tools is built with the same principles as `ocamllex`: it takes as input a file describing a grammar whose rules have semantic actions attached, and returns two Objective Caml files with extensions `.ml` ant `.mli` containing a parsing function and its interface.

General format The syntax description files for `ocamlyacc` use extension `.mly` by convention and they have the following structure:

```
%{
    header
}%
    declarations
%%
    rules
%%
    trailer-and-end
```

The rule format is:

```
non-terminal   :   symbol...symbol   { semantic action }
               |   ...
               |   symbol...symbol   { semantic action }
               ;
```

A symbol is either a terminal or a non-terminal. Sections "header" and "trailer-and-end" play the same role as in `ocamllex` with the only exception that the header is only

visible by the rules and not by declarations. In particular, this implies that module openings (**open**) are not taken into consideration in the declaration part and the types must therefore be fully qualified.

Semantic actions Semantic actions are pieces of Objective Caml code executed when the parser reduces the rule they are associated with. The body of a semantic action may reference the components of the right-hand term of the rule. These are numbered from left to right starting with 1. The first component is referenced by `$1`, the second by `$2`, etc.

Start Symbols We may declare several start symbols in the grammar, by writing in the declaration section:

```
%start non-terminal .. non-terminal
```

For each of them a parsing function will be generated. We must precisely note, always in the declaration section, the output type of these functions.

```
%type <output-type> non-terminal
```

The `output-type` must be qualified.

Warning Non-terminal symbols become the name of parsing functions. Therefore, they must not start with a capital letter which is reserved for constructors.

Lexical units Grammar rules make reference to lexical units, the terminals or terminal symbols in the rules.

One (or several) lexemes are declared in the following fashion:

```
%token PLUS MINUS MULT DIV MOD
```

Certain lexical units, like identifiers, represent a set of (character) strings. When we find an identifier we may be interested in recovering its character string. We specify in the parser that these lexemes have an associated value by enclosing the type of this value between `<` and `>`:

```
%token <string> IDENT
```

Warning After being processed by `ocamlyacc` all these declarations are transformed into constructors of type *`token`*. Therefore, they must start with a capital letter.

We may use character strings as implicit terminals as in:

```
expr : expr "+" expr   { ... }
     | expr "*" expr   { ... }
     | ...
     ;
```

in which case it is pointless to declare a symbol which represents them: they are directly processed by the parser without passing through the lexer. In the interest of uniformity, we do not advise this procedure.

Precedence, associativity We have seen that many bottom-up parsing conflicts arise from implicit operator association rules or precedence conflicts between operators. To handle these conflicts, we may declare default associativity rules (left-to-right or non-associative) for operators as well as precedence rules. The following declaration states that operators `+` (lexeme `PLUS`) and `*` (lexeme `MULT`) associate to the right by default and `*` has a higher precedence than `+` because `MULT` is declared after `PLUS`.

```
%left PLUS
%left MULT
```

Two operators declared on the same line have the same precedence.

Command options `ocamlyacc` has two options:

- `-b name`: the generated Objective Caml files are `name.ml` and `name.mli`;
- `-v`: create a file with extension `.output` contaning rule numeration, the states in the automaton recognizing the grammar and the sources of conflicts.

Joint usage with `ocamllex` We may compose both tools `ocamllex` and `ocamlyacc` so that the transformation of a character stream into a lexeme stream is the input to the parser. To do this, type *`lexeme`* should be known to both. This type is defined in the files with extensions `.mli` and `.ml` generated by `ocamlyacc` from the declaration of the **`token`**s in the matching file with extension `.mly`. The `.mll` file imports this type; `ocamllex` translates this file into an Objective Caml function of type *`Lexing.lexbuf -> lexeme`*. The example on page 307 illustrates this interaction and describes the different phases of compilation.

Contextual Grammars

Types generated by `ocamlyacc` process languages produced by so-called *context-free* grammars. A parser for such a grammar does not depend on previously processed syntactic values to process the next lexeme. This is not the case of the language L described by the following formula:

$$L ::= wCw \mid w \text{ with } w \in (A|B)^*$$

where A, B and C are terminal symbols. We have written wCw (with $w \in (A|B)^*$) and not simply $(A|B)^*C(A|B)^*$ because we want the *same* word to the left and right of the middle C.

To parse the words in L, we must remember what has already been found before letter C to verify that we find exactly the same thing afterwards. Here is a solution for this problem based on "visiting" a stream. The general idea of the algorithm is to build a stream parsing function which will recognize exactly the subword before the possible occurrence of C.

We use the type:

```
# type token = A | B | C ;;
```

Function `parse_w1` builds the memorizing function for the first w under the guise of a list of atomic stream parsers (*i.e.* for a single *token*):

```
# let rec parse_w1 s =
    match s with parser
      [<'A; l = parse_w1  >] → (parser [<'A >] → "a") :: l
    | [<'B; l = parse_w1  >] → (parser [<'B >] → "b") :: l
    | [< >] → [] ;;
val parse_w1 : token Stream.t -> (token Stream.t -> string) list = <fun>
```

The result of the function returned by `parse_w1` is simply the character string containing the parsed lexical unit.

Function `parse_w2` takes as argument a list built by `parse_w1` to compose each of its elements into a single parsing function:

```
# let rec parse_w2 l =
    match l with
      p :: pl → (parser [< x = p; l = (parse_w2 pl)  >] → x^l)
    | [] → parser [<>] → "" ;;
val parse_w2 : ('a Stream.t -> string) list -> 'a Stream.t -> string = <fun>
```

The result of applying `parse_w2` will be the string representing subword w. By construction, function `parse_w2` will not be able to recognize anything but the subword visited by `parse_w1`.

Using the ability to name intermediate results in streams, we write the recognition function for the words in the language L:

```
# let parse_L = parser  [< l = parse_w1 ; 'C; r = (parse_w2 l) >] → r ;;
val parse_L : token Stream.t -> string = <fun>
```

Here are two small examples. The first results in the string surrounding C, the second fails because the words surrounding C are different:

```
# parse_L [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"
# parse_L [< 'A; 'B; 'C; 'B; 'A >];;
Uncaught exception: Stream.Error("")
```

Basic Revisited

We now want to use `ocamllex` and `ocamlyacc` to replace function `parse` on page 169 for Basic by some functions generated from files specifying the lexicon and syntax of the language.

To do this, we may not re-use as-is the type of lexical units that we have defined. We will be forced to define a more precise type which permits us to distinguish between operators, commands and keywords.

We will also need to isolate the type declarations describing abstract syntax within a file `basic_types.mli`. This will contain the declaration of type `sentences` and all types needed by it.

File `basic_parser.mly`

Header The file header imports the types needed for the abstract syntax as well as two auxiliary functions to convert from character strings to their equivalent in the abstract syntax.

```
%{
open Basic_types ;;

let phrase_of_cmd c =
 match c with
   "RUN" → Run
 | "LIST" → List
 | "END" → End
 | _ → failwith "line : unexpected command"
;;

let bin_op_of_rel r =
 match r with
   "=" → EQUAL
 | "<" → INF
 | "<=" → INFEQ
 | ">" → SUP
 | ">=" → SUPEQ
 | "<>" → DIFF
```

```
 | _ → failwith "line : unexpected relation symbol"
;;

%}
```

Declarations contains three sections: lexeme declarations, their rule associativity and precedence declarations, and the declaration of the start symbol `line` which stands for the parsing of a command or program line.

Lexical units are the following:

```
%token <int> Lint
%token <string> Lident
%token <string> Lstring
%token <string> Lcmd
%token Lplus Lminus Lmult Ldiv Lmod
%token <string> Lrel
%token Land Lor Lneg
%token Lpar Rpar
%token <string> Lrem
%token Lrem Llet Lprint Linput Lif Lthen Lgoto
%token Lequal
%token Leol
```

Their names are self-explanatory and they are described in file `basic_lexer.mll` (see page 310).

Precedence rules between operators once again take the values assigned by functions `priority_uop` and `priority_binop` defined when first giving the grammar for our Basic (see page 160).

```
%right Lneg
%left Land Lor
%left Lequal Lrel
%left Lmod
%left Lplus Lminus
%left Lmult Ldiv
%nonassoc Lop
```

Symbol `Lop` will be used to process unary minus. It is not a terminal in the grammar, but a "pseudo non-terminal" which allows overloading of operators when two uses of an operator should not receive the same precedence depending on context. This is the case with the minus symbol (`-`). We will reconsider this point once we have specified the rules in the grammar.

Since the start symbol is `line`, the function generated will return the syntax tree for the parsed line.

```
%start line
%type <Basic_types.phrase> line
```

Grammar rules are decomposed into three non-terminals: `line` for a line; `inst` for an instruction in the language; `exp` for expressions. The action associated with each rule simply builds the corresponding abstract syntax tree.

```
%%
line :
   Lint inst Leol              { Line {num=$1; inst=$2} }
 | Lcmd Leol                   { phrase_of_cmd $1 }
 ;

inst :
   Lrem                        { Rem $1 }
 | Lgoto Lint                  { Goto $2 }
 | Lprint exp                  { Print $2 }
 | Linput Lident               { Input $2 }
 | Lif exp Lthen Lint          { If ($2, $4) }
 | Llet Lident Lequal exp       { Let ($2, $4) }
 ;

exp :
   Lint                        { ExpInt $1 }
 | Lident                      { ExpVar $1 }
 | Lstring                     { ExpStr $1 }
 | Lneg exp                    { ExpUnr (NOT, $2) }
 | exp Lplus exp               { ExpBin ($1, PLUS, $3) }
 | exp Lminus exp              { ExpBin ($1, MINUS, $3) }
 | exp Lmult exp               { ExpBin ($1, MULT, $3) }
 | exp Ldiv exp                { ExpBin ($1, DIV, $3) }
 | exp Lmod exp                { ExpBin ($1, MOD, $3) }
 | exp Lequal exp               { ExpBin ($1, EQUAL, $3) }
 | exp Lrel exp                { ExpBin ($1, (bin_op_of_rel $2), $3) }
 | exp Land exp                { ExpBin ($1, AND, $3) }
 | exp Lor exp                 { ExpBin ($1, OR, $3) }
 | Lminus exp %prec Lop       { ExpUnr(OPPOSITE, $2) }
 | Lpar exp Rpar               { $2 }
 ;
%%
```

These rules do not call for particular remarks except:

```
exp :
 ...
 | Lminus exp %prec Lop { ExpUnr(OPPOSITE, $2) }
```

It concerns the use of unary `-`. Keyword `%prec` that we find in it declares that this rule should receive the precedence of `Lop` (here the highest precedence).

File `basic_lexer.mll`

Lexical analysis only contains one rule, `lexer`, which corresponds closely to the old function `lexer` (see page 165). The semantic action associated with the recognition of each lexical unit is simply the emission of the related constructor. As the type of lexical units is declared in the syntax rule file, we have to include the file here. We add a simple auxiliary function that strips double quotation marks from character strings.

```
{
 open Basic_parser ;;

 let string_chars s = String.sub s 1 ((String.length s)-2) ;;
}

rule lexer = parse
   [' ' '\t']            { lexer lexbuf }

 | '\n'                  { Leol }

 | '!'                   { Lneg }
 | '&'                   { Land }
 | '|'                   { Lor }
 | '='                   { Lequal }
 | '%'                   { Lmod }
 | '+'                   { Lplus }
 | '-'                   { Lminus }
 | '*'                   { Lmult }
 | '/'                   { Ldiv }

 | ['<' '>']             { Lrel (Lexing.lexeme lexbuf) }
 | "<="                  { Lrel (Lexing.lexeme lexbuf) }
 | ">="                  { Lrel (Lexing.lexeme lexbuf) }

 | "REM" [^ '\n']*       { Lrem (Lexing.lexeme lexbuf) }
 | "LET"                 { Llet }
 | "PRINT"               { Lprint }
 | "INPUT"               { Linput }
 | "IF"                  { Lif }
 | "THEN"                { Lthen }
 | "GOTO"                { Lgoto }

 | "RUN"                { Lcmd (Lexing.lexeme lexbuf) }
 | "LIST"               { Lcmd (Lexing.lexeme lexbuf) }
 | "END"                { Lcmd (Lexing.lexeme lexbuf) }
```

```
| ['0'-'9']+          { Lint (int_of_string (Lexing.lexeme lexbuf)) }
| ['A'-'z']+          { Lident (Lexing.lexeme lexbuf) }
| '"' [^ '"']* '"'    { Lstring (string_chars (Lexing.lexeme lexbuf)) }
```

Note that we isolated symbol `=` which is used in both expressions and assignments.

Only two of these regular expressions need further remarks. The first concerns comment lines (`"REM" [^ '\n']*`). This rule recognizes keyword `REM` followed by an arbitrary number of characters other than `'\n'`. The second remark concerns character strings (`'"' [^ '"']* '"'`) considered as sequences of characters different from `"` and contained between two `"`.

Compiling, Linking

The compilation of the lexer and parser must be carried out in a definite order. This is due to the mutual dependency between the declaration of lexemes. To compile our example, we must enter the following sequence of commands:

```
ocamlc -c basic_types.mli
ocamlyacc basic_parser.mly
ocamllex basic_lexer.mll
ocamlc -c basic_parser.mli
ocamlc -c basic_lexer.ml
ocamlc -c basic_parser.ml
```

Which will generate files `basic_lexer.cmo` and `basic_parser.cmo` which may be linked into an application.

We now have at our disposal all the material needed to reimplement the application.

We suppress all types and all functions in paragraphs "lexical analysis" (on page 163) and "parsing" (on page 165) of our Basic application; in function `one_command` (on page 174), we replace expression

```
    match parse (input_line stdin) with
```

with

```
    match line lexer (Lexing.from_string ((input_line stdin)^"\n")) with
```

We need to remark that we must put back at the end of the line the character `'\n'` which function `input_line` had filtered out. This is necessary because the `'\n'` character indicates the end of a command line (`Leol`).

Exercises

Filtering Comments Out

Comments in Objective Caml are hierarchical. We can thus comment away sections of text, including those containing comments. A comment starts with characters `(*` and finishes with `*)`. Here's an example:

```
(* comment spread
      over several
        lines *)

 let succ x = (* successor function *)
   x + 1;;

(* level 1 commented text
  let old_succ y = (* level 2 successor function level 2 *)
    y +1;;
    level 1 *)
  succ 2;;
```

The aim of this exercise is to create a new text without comments. You are free to choose whatever lexical analysis tool you wish.

1. Write a lexer able to recognize Objective Caml comments. These start with a `(*` and end with a `*)`. Your lexer should ensure comments are balanced, that is to say the number of comment openings equals the number of comment closings. We are not interested in other constructions in the language which may contain characters `(*` and `*)`.
2. Write a program which takes a file, reads it, filters comments away and writes a new file with the remaining text.
3. In Objective Caml character strings may contain any character, even the sequences `(*` and `*)`. For example, character string `"what(*ever te*)xt"` should not be considered a comment. Modify your lexer to consider character strings.
4. Use this new lexer to remove comments from an Objective Caml program .

Evaluator

We will use `ocamlyacc` to implement an expression evaluator. The idea is to perform the evaluation of expressions directly in the grammar rules.

We choose a (completely parenthesized) prefix arithmetic expression language with variable arity operators. For example, expression `(ADD e1 e2 .. en)` is equivalent to `e1 + e2 + .. + en`. Plus and times operators are right-associative and subtraction and division are left-associative.

1. Define in file `opn_parser.mly` the parsing and evaluation rules for an expression.

2. Define in file `opn_lexer.mll` the lexical analysis of expressions.
3. Write a simple main program `opn` which reads a line from standard input containing an expression and prints the result of evaluating the expression.

Summary

This chapter has introduced several Objective Caml tools for lexical analysis (lexing) and syntax analysis (parsing). We explored (in order of occurrence):

- module `Str` to filter rational expressions;
- module `Genlex` to easily build simple lexers;
- the `ocamllex` tool, a typed integration of the `lex` tool;
- the `ocamlyacc` tool, a typed integration of the `yacc` tool;
- the use of streams to build top-down parsers, including contextual parsers.

Tools `ocamllex` and `ocamlyacc` were used to define a parser for the language Basic more easily maintained than that introduced in page 159.

To Learn More

The reference book on lexical analysis and parsing is known affectionately as the "dragon book", a reference to the book's cover illustration. Its real name is *Compilers: principles, techniques and tools* ([ASU86]). It covers all aspects of compiler design and implementation. It explains clearly the construction of automata matching a given context-free grammar and the techniques to minimize it. The tools `lex` and `yacc` are described in-depth in several books, a good reference being [LMB92]. The interesting features of `ocamllex` and `ocamlyac` with respect to their original versions are the integration of the Objective Caml language and, above all, the ability to write typed lexers and parsers. With regard to *streams*, the research report by Michel Mauny and Daniel de Rauglaudre [MdR92] gives a good description of the operational semantics of this extension. On the other hand, [CM98] shows how to build such an extension. For a better integration of grammars within the Objective Caml language, or to modify the grammars of the latter, we may also use the `camlp4` tool found at:

Link: http://caml.inria.fr/camlp4/

12

Interoperability with C

Developing programs in a given language very often requires one to integrate libraries written in other languages. The two main reasons for this are:

- to use libraries that cannot be written in the language, thus extending its functionality;
- to use high-performance libraries already implemented in another language.

A program then becomes an assembly of software components written in various languages, where each component has been written in the language most appropriate for the part of the problem it addresses. Those software components interoperate by exchanging values and requesting computations.

The Objective Caml language offers such a mechanism for interoperability with the C language. This mechanism allows Objective Caml code to call C functions with Caml-provided arguments, and to get back the result of the computation in Objective Caml. The converse is also possible: a C program can trigger an Objective Caml computation, then work on its result.

The choice of C as interoperability language is justified by the following reasons:

- it is a standardized language (ISO C);
- C is a popular implementation language for operating systems (Unix, Windows, MacOS, etc.);
- a great many libraries are written in C;
- most programming languages offer a C interface, thus it is possible to interface Objective Caml with these languages by going through C.

The C language can therefore be viewed as the esperanto of programming languages.

Cooperation between C and Objective Caml raises a number of difficulties that we review below.

- **Machine representation of data**
  For instance, values of base types (*int*, *char*, *float*) have different machine representations in the two languages. This requires conversion between the representations, in both directions. The same holds for data structures such as records, sum types[1], or arrays.
- **The Objective Caml garbage collector**
  Standard C does not provide garbage collection. (However, garbage collectors are easily written in C.) Moreover, calling a C function from Objective Caml must not modify the memory in ways incompatible with the Objective Caml GC.
- **Aborted computations**
  Standard C does not support exceptions, and provides different mechanisms for aborting computations. This complicates Objective Caml's exception handling.
- **Sharing common resources**
  For instance, files and other input-output devices are shared between Objective Caml and C, but each language maintains its own input-output buffers. This may violate the proper sequencing of input-output operations in mixed programs.

Programs written in Objective Caml benefit from the safety of static typing and automatic memory management. This safety must not be compromised by improper use of C libraries and interfacing with other languages through C. The programmer must therefore adhere to rather strict rules to ensure that both languages coexist peacefully.

Chapter outline

This chapter introduces the tools that allow interoperability between Objective Caml and C by building executables containing code fragments written in both languages. These tools include functions to convert between the data representations of each language, allocation functions using the Objective Caml heap and garbage collector, and functions to raise Objective Caml exceptions from C.

The first section shows how to call C functions from Objective Caml and how to build executables and interactive toplevel interpreters including the C code implementing those functions. The second section explores the C representation of Objective Caml values. The third section explains how to create and modify Objective Caml values from C. It discusses the interactions between C allocations and the Objective Caml garbage collector, and presents the mechanisms ensuring safe allocation from C. The fourth section describes exception handling: how to raise exceptions and how to handle them. The fifth section reverses the roles: it shows how to include Objective Caml code in an application whose main program is written in C.

1. Objective Caml's sum types are discriminated unions. Refer to chapter 2, page 45 for a full description.

Note

This chapter assumes a working knowledge of the C language. Moreover, reading chapter 9 can be helpful in understanding the issues raised by automatic memory management.

Communication between C and Objective Caml

Communication between parts of a program written in C and in Objective Caml is accomplished by creating an executable (or a new toplevel interpreter) containing both parts. These parts can be separately compiled. It is therefore the responsibility of the linking phase[2] to establish the connection between Objective Caml function names and C function names, and to create the final executable. To this end, the Objective Caml part of the program contains external declarations describing this connection.

Figure 12.1 shows a sample program composed of a C part and an Objective Caml part. Each part comprises code (function definitions and toplevel expressions for Objective

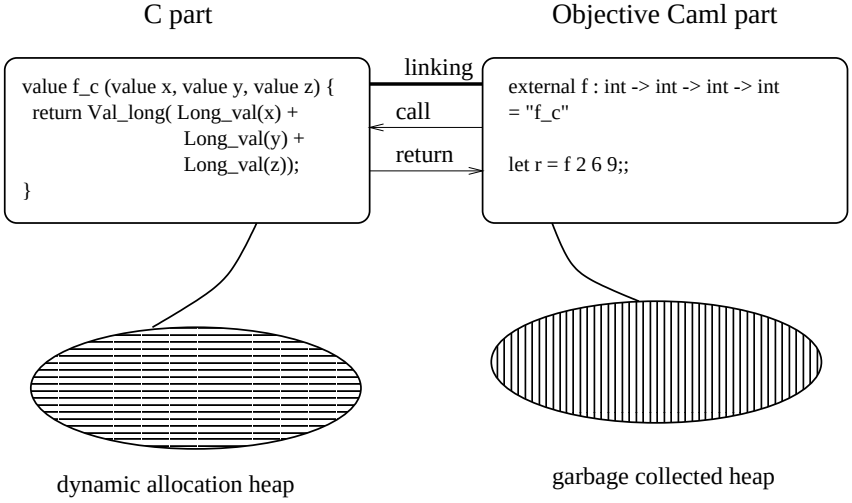


Figure 12.1: Communication between Objective Caml and C.

Caml) and a memory area for dynamic allocation. Calling the function `f` with three Objective Caml integer arguments triggers a call to the C function `f_c`. The body of the C function converts the three Objective Caml integers to C integers, computes their sum, and returns the result converted to an Objective Caml integer.

2. Linking is performed differently for the bytecode compiler and the native-code compiler.

We now introduce the basic mechanisms for interfacing C with Objective Caml: external declarations, calling conventions for C functions invoked from Objective Caml, and linking options. Then, we show an example using input-output.

External declarations

External function declarations in Objective Caml associate a C function definition with an Objective Caml name, while giving the type of the latter.

The syntax is as follows:

Syntax : **external** *caml_name* **:** *type* **=** "*C_name*"

This declaration indicates that calling the function *caml_name* from Objective Caml code performs a call to the C function *C_name* with the given arguments. Thus, the example in figure 12.1 declares the function `f` as the Objective Caml equivalent of the C function `f_c`.

An external function can be declared in an interface (*i.e.*, in an `.mli` file) either as an external or as a regular value:

Syntax :

external *caml_name* **:** *type* **=** "*C_name*"
val *caml_name* **:** *type*

In the latter case, calls to the C function first go through the general function application mechanism of Objective Caml. This is slightly less efficient, but hides the implementation of the function as a C function.

Declaration of the C functions

C functions intended to be called from Objective Caml must have the same number of arguments as described in their external declarations. These arguments have type `value`, which is the C type for Objective Caml values. Since those values have uniform representations (see chapter 9), a single C type suffices to encode all Objective Caml values. On page 323, we will present the facilities for encoding and decoding values, and illustrate them by a function that explores the representations of Objective Caml values.

The example in figure 12.1 respects the constraints mentioned above. The function `f_c`, associated with an Objective Caml function of type `int -> int -> int -> int`, is indeed a function with three parameters of type `value` returning a result of type `value`.

The Objective Caml bytecode interpreter evaluates calls to external functions differently, depending on the number of arguments[3]. If the number of arguments is less than or equal to five, the arguments are passed directly to the C function. If the number of arguments is greater than five, the C function's first parameter will get an array containing all of the arguments, and the C function's second parameter will get the number of arguments. These two cases must therefore be distinguished for external C functions that can be called from the bytecode interpreter. On the other hand, the Objective Caml native-code compiler always calls external functions by passing all the arguments directly, as function parameters.

External functions with more than five arguments

For external functions with more than five arguments, the programmer must provide two C functions: one for bytecode and the other for native-code. The syntax of external declarations allows the declaration of one Objective Caml function associated with two C functions:

Syntax : **external** *caml_name* **:** *type* **=** "*C_name_bytecode*" "*C_name_native*"

The function *C_name_bytecode* takes two parameters: an array of values of type `value` (*i.e.* a C pointer of type `value*`) and an integer giving the number of elements in this array.

Example

The following C program defines two functions for adding together six integers: `plus_native`, callable from native code, and `plus_bytecode`, callable from the bytecode compiler. The C code must include the file `mlvalues.h` containing the definitions of C types, Objective Caml values, and conversion macros.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value plus_native (value x1,value x2,value x3,value x4,value x5,value x6)
{
  printf("<< NATIVE PLUS >>\n") ; fflush(stdout) ;
  return Val_long ( Long_val(x1) + Long_val(x2) + Long_val(x3)
                  + Long_val(x4) + Long_val(x5) + Long_val(x6)) ;
}

value plus_bytecode (value * tab_val, int num_val)
{
  int i;
  long res;
```

3. Recall that a function such as `fst`, of type *'a * 'b -> 'a*, does not have two arguments, but only one that happens to be a pair; on the other hand, a function of type *int -> int -> int* has two arguments.

```
  printf("<< BYTECODED PLUS >> : ") ; fflush(stdout) ;
  for (i=0,res=0;i<num_val;i++) res += Long_val(tab_val[i]) ;
  return Val_long(res) ;
}
```

The following Objective Caml program `exOCAML.ml` calls these two C functions.

```
external plus : int → int → int → int → int → int → int
              = "plus_bytecode" "plus_native" ;;
print_int (plus 1 2 3 4 5 6) ;;
print_newline () ;;
```

We now compile these programs with the two Objective Caml compilers and a C compiler that we call `cc`. We must give it the access path for the `mlvalues.h` include file.

```
$ cc -c -I/usr/local/lib/ocaml  exC.c

$ ocamlc -custom exC.o exOCAML.ml -o ex_byte_code.exe
$ ex_byte_code.exe
<< BYTECODED PLUS >> : 21

$ ocamlopt exC.o exOCAML.ml -o ex_native.exe
$ ex_native.exe
<< NATIVE PLUS >> : 21
```

Note

To avoid writing the C function twice (with the same body but different calling conventions), it suffices to implement the bytecode version as a call to the native-code version, as in the following sketch:

```
value prim_nat (value x1, ..., value xn) { ... }
value prim_bc (value *tbl, int n)
{ return prim_nat(tbl[0],tbl[1],...,tbl[n-1]) ; }
```

Linking with C

The linking phase creates an executable from C and Objective Caml files compiled with their respective compilers. The result of the native-code compiler is shown in figure 12.2.

The compilation of the C and Objective Caml sources generates machine code that is stored in the static allocation area of the program. The dynamic allocation area contains the execution stack (corresponding to the function calls in progress) and the heaps for C and Objective Caml.

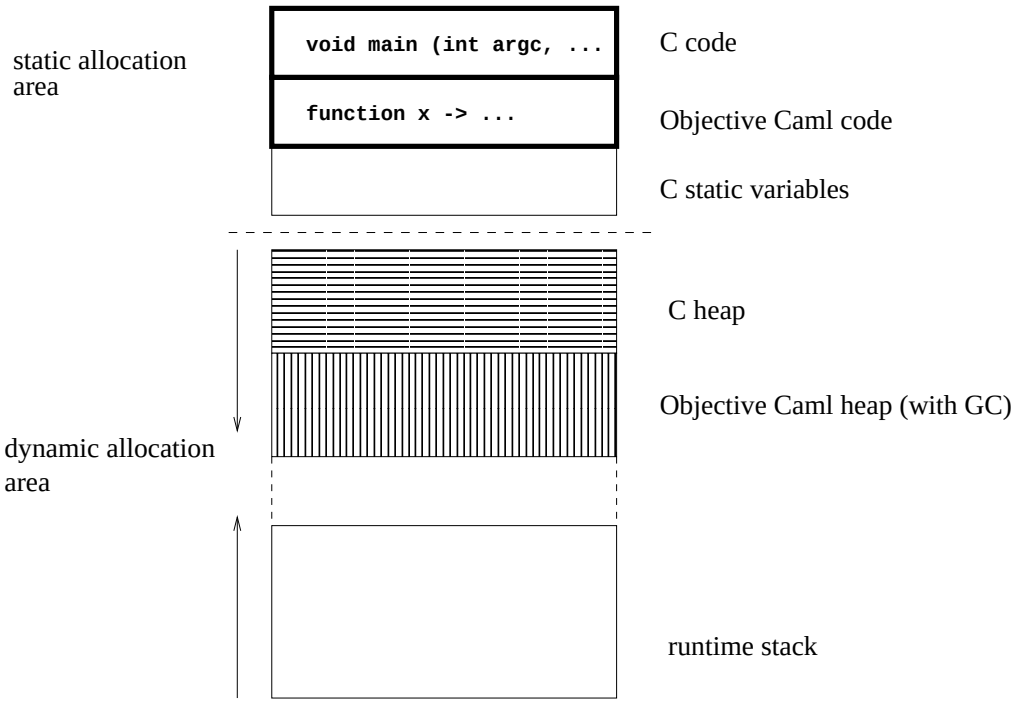


Figure 12.2: Mixed-language executable.

Run-time libraries

The C functions that can be called from a program using only the standard Objective Caml library are contained in the execution library of the abstract machine (see figure 7.3 page 200). For such a program, there is no need to provide additional libraries at link-time. However, when using Objective Caml libraries such as `Graphics`, `Num` or `Str`, the programmer must explicitly provide the corresponding C libraries at link-time. This is the purpose of the `-custom` compiler option (see see chapter 7, page 207). Similarly, when we wish to call our C functions from Objective Caml, we must provide the object file containing those C functions at link-time. The following example illustrates this.

The three linking modes

The linking commands differ slightly between the native-code compiler, the bytecode compiler, and the construction of toplevel interactive loops. The compiler options relevant to these linking modes are described in chapter 7.

To illustrate these linking modes, we consider again the example in figure 12.1. Assume the Objective Caml source file is named `progocaml.ml`. It uses the external function `f_c` defined in the C file `progC.c`. In turn, the function `f_c` refers to a C library

`a_C_library.a`. Once all these files are compiled separately, we link them together using the following commands:

- bytecode:
  `ocamlc -custom -o vbc.exe progC.o a_C_library.a progocaml.cmo`
- native code:
  `ocamlopt progC.o -o vn.exe a_C_library.a progocaml.cmx`

We obtain two executable files: `vbc.exe` for the bytecode version, and `vn.exe` for the native-code version.

Building an enriched abstract machine

Another possibility is to augment the run-time library of the abstract machine with new C functions callable from Objective Caml. This is achieved by the following commands:

```
ocamlc -make-runtime -o new_ocamlrun progC.o a_C_library.a
```

We can then build a bytecode executable `vbcnam.exe` targeted to the new abstract machine:

```
ocamlc -o vbcnam.exe -use-runtime new_ocamlrun progocaml.cmo
```

To run this bytecode executable, either give it as the first argument to the new abstract machine, as in `new_ocaml vbcnam.exe` , or run it directly as `vbcnam.exe`

Note

Linking in `-custom` mode scans the object files (`.cmo`) to build a table of all external functions mentioned. The bytecode required to use them is generated and added to the bytecode corresponding to the Objective Caml code.

Building a toplevel interactive loop

To be able to use an external function in the toplevel interactive loop, we must first build a new toplevel interpreter containing the C code for the function, as well as an Objective Caml file containing its declaration.

We assume that we have compiled the file `progC.c` containing the function `f_c`. We then build the toplevel loop `ftop` as follows:

```
ocamlmktop -custom -o ftop progC.o a_C_library.a ex.ml
```

The file `ex.ml` contains the external declaration for the function `f`. The new toplevel interpreter `ftop` then knows this function and contains the corresponding C code, as found in `progC.o`.

Mixing input-output in C and in Objective Caml

The input-output functions in C and in Objective Caml do not share their file buffers. Consider the following C program:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value hello_world (value v)
  { printf("Hello World !!");  fflush(stdout);  return v; }
```

Writes to standard output must be flushed explicitly (`fflush`) to guarantee that they will be printed in the intended order.

```
# external caml_hello_world : unit → unit = "hello_world"  ;;
external caml_hello_world : unit -> unit = "hello_world"
# print_string "<< " ;
   caml_hello_world () ;
   print_string " >>\n" ;
   flush stdout ;;
Hello World !!<<  >>
- : unit = ()
```

The outputs from C and from Objective Caml are not intermingled as expected, because each language buffers its outputs independently. To get the correct behavior, the Objective Caml part must be rewritten as follows:

```
# print_string "<< " ; flush stdout ;
   caml_hello_world () ;
   print_string " >>\n" ; flush stdout ;;
<< Hello World !! >>
- : unit = ()
```

By flushing the Objective Caml output buffer after each write, we ensure that the outputs from each language appear in the expected order.

Exploring Objective Caml values from C

The machine representation of Objective Caml values differs from that of C values, even for fundamental types such as integers. This is because the Objective Caml garbage collector needs to record additional information in values. Since Objective Caml values are represented uniformly, their representations all belong to the same C type, named (unsurprisingly) `value`.

When Objective Caml calls a C function, passing it one or several arguments, those arguments must be decoded before using them in the C function. Similarly, the result of this C function must be encoded before being returned to Objective Caml.

These conversions (decoding and encoding) are performed by a number of macros and C functions provided by the Objective Caml runtime system. These macros and functions are declared in the include files listed in figure 12.3. These include files are part of the Objective Caml installation, and can be found in the directory where Objective Caml libraries are installed[4]

`caml/mlvalues.h`	definition of the `value` type and basic value conversion macros.
`caml/alloc.h`	functions for allocating Objective Caml values.
`caml/memory.h`	macros for interfacing with the Objective Caml garbage collector.

Figure 12.3: Include files for the C interface.

Classification of Objective Caml representations

An Objective Caml representation, that is, a C datum of type `value`, is one of:

- an immediate value (represented as an integer);
- a pointer into the Objective Caml heap;
- a pointer pointing outside the Objective Caml heap.

The Objective Caml heap is the memory area that is managed by the Objective Caml garbage collector. C code can also allocate and manipulate data structures in its own memory space, and communicate pointers to these data structures to Objective Caml.

Figure 12.4 shows the macros for classifying representations and converting between C integers and their Objective Caml representation. Note that C offers several integer

`Is_long(v)`	is `v` an Objective Caml integer?
`Is_block(v)`	is `v` an Objective Caml pointer?
`Long_val(v)`	extract the integer contained in `v`, as a C "long"
`Int_val(v)`	extract the integer contained in `v`, as a C "int"
`Bool_val(v)`	extract the boolean contained in `v` (0 if `false`, non-zero if `true`)

Figure 12.4: Classification of representations and conversion of immediate values.

types of varying sizes (`short`, `int`, `long`, etc), while Objective Caml has only one integer type, `int`.

4. Under Unix, this directory is `/usr/local/lib/ocaml` by default, or sometimes `/usr/lib/ocaml`. Under Windows, the default location is `C: \OCAML\LIB`, or the value of the environment variable `CAMLLIB`, if set.

Accessing immediate values

All Objective Caml immediate values are represented as integers:

- integers are represented by their value;
- characters are represented by their ASCII code[5];
- constant constructors are represented by an integer corresponding to their position in the datatype declaration: the n$^{\text{th}}$ constant constructor of a datatype is represented by the integer $n - 1$.

The following program defines a C function `inspect` that inspects the representation of its argument:

```
#include <stdio.h>
#include <caml/mlvalues.h>
value inspect (value v)
{
  if (Is_long(v))
    printf ("v is an integer (%ld) : %ld", (long) v, Long_val(v));
  else if (Is_block(v))
    printf ("v is a pointer");
  else
    printf ("v is neither an integer nor a pointer (???)");
  printf("   ");
  fflush(stdout) ;
  return v ;
}
```

The function `inspect` tests whether its argument is an Objective Caml integer. If so, it prints the integer twice, first viewed as a C long integer (without conversion), then converted by the `Long_val` macro, which extracts the actual integer represented in the argument.

On the following example, we see that the machine representation of integers in Objective Caml differs from that of C:

```
# external inspect : 'a → 'a = "inspect" ;;
external inspect : 'a -> 'a = "inspect"
# inspect 123 ;;
v is an integer (247) : 123   - : int = 123
# inspect max_int;;
v is an integer (2147483647) : 1073741823   - : int = 1073741823
```

We can also inspect values of other predefined types, such as *char* and *bool*:

```
# inspect 'A' ;;
v is an integer (131) : 65   - : char = 'A'
# inspect true ;;
v is an integer (3) : 1   - : bool = true
```

5. More precisely, by their ISO Latin-1 code, which is an 8-bit character encoding extending ASCII with accented letters and signs for Western languages. Objective Caml does not yet handle wider internationalized character sets such as Unicode.

```
# inspect false ;;
v is an integer (1) : 0   - : bool = false
# inspect [] ;;
v is an integer (1) : 0   - : 'a list = []
```

Consider the Objective Caml type `foo` defined thus:

```
# type foo = C1 | C2 of int | C3 | C4 ;;
```

The `inspect` function shows that constant constructors and non-constant constructors of this type are represented differently:

```
# inspect C1 ;;
v is an integer (1) : 0   - : foo = C1
# inspect C4 ;;
v is an integer (5) : 2   - : foo = C4
# inspect (C2 1) ;;
v is a pointer   - : foo = C2 1
```

When the function `inspect` detects an immediate value, it prints first the "physical" representation of this value (*i.e.* the representation viewed as a word-sized C integer of C type `long`); then it prints the "logical" contents of this value (*i.e.* the Objective Caml integer it represents, as returned by the decoding macro `Long_val`). The examples above show that the "physical" and the "logical" contents differ. This difference is due to the tag bit[6] used by the garbage collector to distinguish immediate values from pointers (see chapter 9, page 253).

Representation of structured values

Non-immediate Objective Caml values are said to be structured values. Those values are allocated in the Objective Caml heap and represented as a pointer to the corresponding memory block. All memory blocks contain a header word indicating the kind of the block as well as its size expressed in machine words. Figure 12.5 shows the structure of a block for a 32-bit machine. The two "color" bits are used by the garbage collector for walking the memory graph (see chapter 9, page 254). The "tag" field, or "tag" for short, contains the kind of the block. The "size" field contains the size of the block, in words, excluding the header. The macros listed in figure 12.6 return the tag and size of a block. The tag of a memory block can take the values listed in figure 12.7. Depending on the block tag, different macros are used to access the contents of the blocks. These macros are described in figure 12.8. When the tag is less than `No_scan_tag`, the heap block is structured as an array of Objective Caml value representations. Each element of the array is called a "field" of the memory block. In accordance with C and Objective Caml conventions, the first field is at index 0, and the last field is at index `Wosize_val(v) - 1`.

6. Here, the tag bit is the least significant bit.

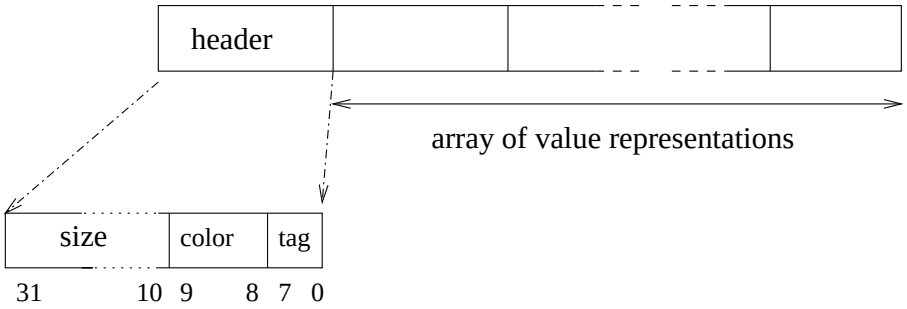


Figure 12.5: Structure of an Objective Caml heap block.

`Wosize_val(v)`	return the size of the block `v` (header excluded)
`Tag_val(v)`	return the tag of the block `v`

Figure 12.6: Accessing header information in memory blocks.

As we did earlier for immediate values, we now define a function to inspect memory blocks. The C function `print_block` takes an Objective Caml value representation, tests whether it is an immediate value or a memory block, and in the latter case prints the kind and contents of the block. It is called from the wrapper function `inspect_block`, which can be called from Objective Caml.

```
#include <stdio.h>
#include <caml/mlvalues.h>

void margin (int n)
  { while (n-- > 0) printf(".");  return; }

void print_block (value v,int m)
{
  int size, i;
```

from `0` to `No_scan_tag-1`	an array of Objective Caml value representations
`Closure_tag`	a function closure
`String_tag`	a character string
`Double_tag`	a double-precision float
`Double_array_tag`	an array of float
`Abstract_tag`	an abstract data type
`Final_tag`	an abstract data type equipped with a finalization function

Figure 12.7: Tags of memory blocks.

`Field(v,n)`	return the `n`$^{\text{th}}$ field of `v`.
`Code_val(v)`	return the code pointer for a closure.
`string_length(v)`	return the length of a string.
`Byte(v,n)`	return the `n` $^{\text{th}}$ character of a string, with C type `char`.
`Byte_u(v,n)`	same, but result has C type `unsigned char`.
`String_val(v)`	return the contents of a string with C type `(char *)`.
`Double_val(v)`	return the float contained in `v`.
`Double_field(v,n)`	return the `n` $^{\text{th}}$ float contained in the float array `v`.

Figure 12.8: Accessing the content of a memory block.

```
  margin(m);
  if (Is_long(v))
    { printf("immediate value (%d)\n", Long_val(v));  return; };
  printf ("memory block: size=%d  -  ", size=Wosize_val(v));
  switch (Tag_val(v))
   {
    case Closure_tag :
        printf("closure with %d free variables\n", size-1);
        margin(m+4); printf("code pointer: %p\n",Code_val(v)) ;
        for (i=1;i<size;i++)  print_block(Field(v,i), m+4);
        break;
    case String_tag :
        printf("string: %s (%s)\n", String_val(v),(char *) v);
        break;
    case Double_tag:
        printf("float: %g\n", Double_val(v));
        break;
    case Double_array_tag :
        printf ("float array: ");
        for (i=0;i<size/Double_wosize;i++)  printf("  %g", Double_field(v,i));
        printf("\n");
        break;
    case Abstract_tag : printf("abstract type\n"); break;
    case Final_tag : printf("abstract finalized type\n"); break;
    default:
        if (Tag_val(v)>=No_scan_tag) { printf("unknown tag"); break; };
        printf("structured block (tag=%d):\n",Tag_val(v));
        for (i=0;i<size;i++)  print_block(Field(v,i),m+4);
   }
  return ;
}

value inspect_block (value v)
```

```
  { print_block(v,4); fflush(stdout); return v; }
```

Each possible tag for a block corresponds to a case of the `switch` construct. In the case of a block containing an array of Objective Caml values, we recursively call `print_block` on each field of the array. We then redefine the `inspect` function:

```
# external inspect : 'a → 'a = "inspect_block" ;;
external inspect : 'a -> 'a = "inspect_block"
```

We can now explore the representations of Objective Caml structured values. We must be careful not to apply `inspect_block` to a cyclic value, since the recursive traversal of the value would then loop indefinitely.

Arrays, tuples, and records

Arrays and tuples are represented by structured blocks. The n^{th} field of the block contains the representation of the n^{th} element of the array or tuple.

```
# inspect [| 1; 2; 3 |] ;;
....memory block: size=3  -  structured block (tag=0):
........immediate value (1)
........immediate value (2)
........immediate value (3)
- : int array = [|1; 2; 3|]
# inspect ( 10 , true , () ) ;;
....memory block: size=3  -  structured block (tag=0):
........immediate value (10)
........immediate value (1)
........immediate value (0)
- : int * bool * unit = 10, true, ()
```

Records are also represented as structured blocks. The values of the record fields appear in the order given at record declaration time. Mutable fields and immutable fields are represented identically.

```
# type foo = { fld1: int ; mutable fld2: int } ;;
type foo = { fld1: int; mutable fld2: int }
# inspect { fld1=10 ; fld2=20 } ;;
....memory block: size=2  -  structured block (tag=0):
........immediate value (10)
........immediate value (20)
- : foo = {fld1=10; fld2=20}
```

Warning

> Nothing prevents a C function from physically modifying an immutable record field. It is the programmers' responsibility to make sure that their C functions do not introduce inconsistencies in Objective Caml data structures.

Sum types

We previously saw that constant constructors are represented like integers. A non-constant constructor is represented by a block containing the constructor's arguments, with a tag identifying the constructor. The tag associated with a non-constant constructor represents its position in the type declaration: the first non-constant constructor has tag 0, the second one has tag 1, and so on.

```
# type foo = C1 of int * int * int | C2 of int | C3 | C4 of int * int ;;
type foo = | C1 of int * int * int | C2 of int | C3 | C4 of int * int
# inspect (C1 (1,2,3)) ;;
....memory block: size=3  -  structured block (tag=0):
........immediate value (1)
........immediate value (2)
........immediate value (3)
- : foo = C1 (1, 2, 3)
# inspect (C4 (1,2)) ;;
....memory block: size=2  -  structured block (tag=2):
........immediate value (1)
........immediate value (2)
- : foo = C4 (1, 2)
```

Note

The type *list* is a sum type whose declaration is:
type *'a list* = [] | :: **of** *'a * 'a list*. This type has only one non-constant constructor (::). Thus, a non-empty list is represented by a memory block with tag 0.

Character strings

Characters inside strings occupy one byte each. Thus, the memory block representing a string uses one word per group of four characters (on a 32-bit machine) or eight characters (on a 64-bit machine).

Warning

Objective Caml strings can contain the null character whose ASCII code is 0. In C, the null character represents the end of a string, and cannot appear inside a string.

```
#include <stdio.h>
#include <caml/mlvalues.h>

value explore_string (value v)
{
  char *s;
  int i,size;
  s = (char *) v;
  size = Wosize_val(v) * sizeof(value);
```

```
  for (i=0;i<size;i++)
    {
      int p = (unsigned int) s[i] ;
      if ((p>31) && (p<128)) printf("%c",s[i]); else printf("(#%u)",p);
    }
  printf("\n");
  fflush(stdout);
  return v;
}
```

The length and position of last character of an Objective Caml string are determined not by looking for a terminating null character, as in C, but by combining the size of the memory block that contains the string with the last byte of the last word of this block, which indicates the number of *unused* bytes in the last word. The following examples clarify the role played by this last byte.

```
# external explore : string → string = "explore_string" ;;
external explore : string -> string = "explore_string"
# ignore(explore "");
  ignore(explore "a");
  ignore(explore "ab");
  ignore(explore "abc");
  ignore(explore "abcd");
  ignore(explore "abcd\000") ;;
(#0)(#0)(#0)(#3)
a(#0)(#0)(#2)
ab(#0)(#1)
abc(#0)
abcd(#0)(#0)(#0)(#3)
abcd(#0)(#0)(#0)(#2)
- : unit = ()
```

In the last two examples (`"abcd"` and `"abcd\000"`), the strings are of length 4 and 5 respectively. This explains why the last byte takes two different values, although the other bytes of the string representations are identical.

Floats and float arrays

Objective Caml offers only one type (*float*) of floating-point numbers. This type corresponds to 64-bit, double-precision floating point numbers in C (type `double`). Values of type *float* are heap-allocated and represented by a memory block of size 2 words (on a 32-bit machine) or 1 word (on a 64-bit machine).

```
# inspect 1.5 ;;
....memory block: size=2  -  float: 1.5
- : float = 1.5
# inspect 0.0;;
....memory block: size=2  -  float: 0
- : float = 0
```

Arrays of floats are represented specially to reduce their memory occupancy: the floats contained in the array are stored consecutively in the memory block, rather than having each float heap-allocated separately. Therefore, float arrays possess a specific tag and specific access macros.

```
# inspect [| 1.5 ; 2.5 ; 3.5 |] ;;
....memory block: size=6  -  float array:   1.5  2.5  3.5
- : float array = [|1.5; 2.5; 3.5|]
```

This optimized representation encourages the use of Objective Caml for numerical computations that manipulate many float arrays: operations on array elements are much more efficient than if each float was heap-allocated separately.

Warning

When allocating an Objective Caml float array from C, the size of the block should be the number of array elements multiplied by `Double_wosize`. The `Double_wosize` macro represents the number of words occupied by a double-precision float (2 words on a 32-bit machine, but only 1 word on a 64-bit machine).

With the exception of float arrays, floating-point numbers contained in other data structures are always treated as a structured, heap-allocated value. The following example shows the representation of a list of floats.

```
# inspect [ 3.14; 1.2; 7.6];;
....memory block: size=2  -  structured block (tag=0):
........memory block: size=2  -  float: 3.14
........memory block: size=2  -  structured block (tag=0):
............memory block: size=2  -  float: 1.2
............memory block: size=2  -  structured block (tag=0):
................memory block: size=2  -  float: 7.6
................immediate value (0)
- : float list = [3.14; 1.2; 7.6]
```

The list is viewed as a block with size 2, containing its head and its tail. The head of the list is a float, which is also a block of size 2.

Closures

A function value is represented by the code to be executed when the function is applied, and by its environment (see chapter 2, page 23). There are two ways to build a function value: either by explicit abstraction (as in `fun x -> x+1`) or by partial application of a curried function (as in `(fun x -> fun y -> x+y) 1`).

The environment of a closure can contain three kinds of variables: those declared globally, those declared locally, and the function parameters already instantiated by a partial application. The implementation treats those three kinds differently. Global variables are stored in a global environment that is not explicitly part of any closure. Local variables and instantiated parameters can appear in closures, as we now illustrate.

A closure with an empty environment is simply a memory block containing a pointer to the code of the function:

```
# let f = fun x y z → x+y+z ;;
val f : int -> int -> int -> int = <fun>
# inspect f ;;
....memory block: size=1  -  closure with 0 free variables
........code pointer: 0x808c9d4
- : int -> int -> int -> int = <fun>
```

Functions with free local variables are represented by closures with non-empty environments. Here, the closure contains both a pointer to the code of the function, and the values of its free local variables.

```
# let g = let x = 1 and y = 2 in fun z → x+y+z ;;
val g : int -> int = <fun>
# inspect g ;;
....memory block: size=3  -  closure with 2 free variables
........code pointer: 0x808ca38
........immediate value (1)
........immediate value (2)
- : int -> int = <fun>
```

The Objective Caml virtual machine treats partial applications of functions specially for better performance. A partial application of an abstraction is represented by a closure containing a value for each of the instantiated parameters, plus a pointer to the closure for the initial abstraction.

```
# let a1 = f 1 ;;
val a1 : int -> int -> int = <fun>
# inspect (a1) ;;
....memory block: size=3  -  closure with 2 free variables
........code pointer: 0x808c9d0
........memory block: size=1  -  closure with 0 free variables
............code pointer: 0x808c9d4
........immediate value (1)
- : int -> int -> int = <fun>
# let a2 = a1 2 ;;
val a2 : int -> int = <fun>
# inspect (a2) ;;
....memory block: size=4  -  closure with 3 free variables
........code pointer: 0x808c9d0
........memory block: size=1  -  closure with 0 free variables
............code pointer: 0x808c9d4
........immediate value (1)
........immediate value (2)
- : int -> int = <fun>
```

Figure 12.9 depicts the result of the inspection above.

The function `f` has no free variables, hence the environment part of its closure is empty. The code pointer for a function with several arguments points to the code that should be

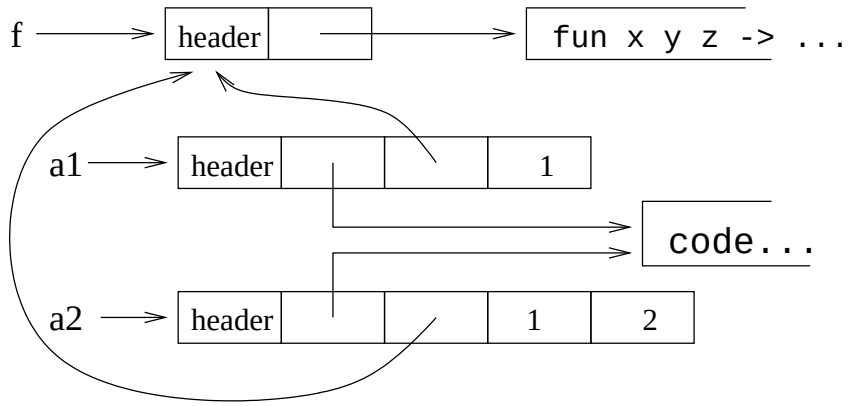


Figure 12.9: Closure representation.

called when all arguments are provided. In the case of `f`, this is the code corresponding to `x+y+z`. Partial applications of this function result in intermediate closures that point to a shared code (it is the same code pointer for `a1` and `a2`). The role of this code is to accumulate the arguments and detect when all arguments have been provided. If so, it pushes all arguments and calls the actual code for the function body; if not, it creates a new closure. For instance, the application of `a1` to 2 fails to provide all arguments to the function `f` (the last argument is still missing), hence a closure is created containing the first two arguments, 1 and 2. Notice that the closures resulting from partial applications always contain, in the first environment slot, a pointer to the original closure. The original closure will be called when all arguments have been gathered.

Mixing local declarations and partial applications results in the following representation:

```
# let g x = let y=2 in fun z → x+y+z ;;
val g : int -> int -> int = <fun>
# let a1 = g 1 ;;
val a1 : int -> int = <fun>
# inspect a1 ;;
....memory block: size=3  -  closure with 2 free variables
........code pointer: 0x808ca78
........immediate value (1)
........immediate value (2)
- : int -> int = <fun>
```

Abstract types

Values of an abstract type are represented like those of its implementation type. Actually, type information is used only during type-checking and compilation. During

execution, the types are not needed – only the memory representation (tag bits on values, size and tag fields on memory blocks) needs to be communicated to the garbage collector.

For instance, a value of the abstract type *'a Stack.t* is represented as a reference to a list, since the type *'a Stack.t* is implemented as *'a list ref*.

```
# let p = Stack.create();;
val p : '_a Stack.t = <abstr>
# Stack.push 3 p;;
- : unit = ()
# inspect p;;
....memory block: size=1  -  structured block (tag=0):
........memory block: size=2  -  structured block (tag=0):
............immediate value (3)
............immediate value (0)
- : int Stack.t = <abstr>
```

On the other hand, some abstract types are implemented by representations that cannot be expressed in Objective Caml. Typical examples include arrays of weak pointers and input-output channels. Often, values of those abstract types are represented as memory blocks with tag `Abstract_tag`.

```
# let w = Weak.create 10;;
val w : '_a Weak.t = <abstr>
# Weak.set w 0 (Some p);;
- : unit = ()
# inspect w;;
....memory block: size=11  -  abstract type
- : int Stack.t Weak.t = <abstr>
```

Sometimes, a finalization function is attached to those values. Finalization functions are C functions which are called by the garbage collector just before the value is collected. They are very useful to free external resources, such as an input-output buffer, just before the memory block referring to those resources disappears. For instance, inspection of the "standard output" channel reveals that the type *out_channel* is represented by abstract memory blocks with a finalization function:

```
# inspect (stdout) ;;
....memory block: size=2  -  abstract finalized type
- : out_channel = <abstr>
```

Creating and modifying Objective Caml values from C

A C function called from Objective Caml can modify its arguments in place, or return a newly-created value. This value must match the Objective Caml type for the function result. For base types, several C macros are provided to convert a C datum to an Objective Caml value. For structured types, the new value must be allocated in the

Objective Caml heap, with the correct size, and its fields initialized with values of the correct types. Considerable care is required here: it is easy to construct bad values from C, and these bad values may crash the Objective Caml program.

Any allocation in the Objective Caml heap can trigger a garbage collection, which will deallocate unused memory blocks and may move live blocks. Therefore, any Objective Caml value manipulated from C must be registered with the Objective Caml garbage collector, if they are to survive the allocation of a new block. These values must be treated as extra memory roots by the garbage collector. To this end, several macros are provided for registering extra roots with the garbage collector.

Finally, C code can allocate Objective Caml heap blocks that contain C data instead of Objective Caml values. This C data will then benefit from Objective Caml's automatic memory management. If the C data requires explicit deallocation, a finalization function can be attached to the heap block.

Modifying Objective Caml values

The following macros allow the creation of immediate Objective Caml values from the corresponding C data, and the modification of structured values in place.

`Val_long(l)`	return the value representing the long integer `l`
`Val_int(i)`	return the value representing the integer `l`
`Val_bool(x)`	return `false` if `x=0`, `true` otherwise
`Val_true`	the representation of `true`
`Val_false`	the representation of `false`
`Val_unit`	the representation of `()`
`Store_field(b,n,v)`	store the value `v` in the `n`-th field of block `b`
`Store_double_field(b,n,d)`	store the float `d` in the `n`-th field of the float array `b`

Figure 12.10: Creation of immediate values and modification of structured blocks.

Moreover, the macros `Byte` and `Byte_u` can be used on the left-hand side of an assignment to modify the characters of a string. The `Field` macro can also be used for assignment on blocks with tag `Abstract_tag` or `Final_tag`; use `Store_field` for blocks with tag between `0` and `No_scan_tag-1`. The following function reverses a character string in place:

```
#include <caml/mlvalues.h>
value swap_char(value v, int i, int j)
  { char c=Byte(v,i); Byte(v,i)=Byte(v,j); Byte(v,j)=c; }
value swap_string (value v)
{
  int i,j,t = string_length(v) ;
```

```
  for (i=0,j=t-1; i<t/2; i++,j--)  swap_char(v,i,j) ;
  return v ;
}

# external mirror : string → string = "swap_string" ;;
external mirror : string -> string = "swap_string"
# mirror "abcdefg" ;;
- : string = "gfedcba"
```

Allocating new blocks

The functions listed in figure 12.11 allocate new blocks in the Objective Caml heap. The

`alloc(n, t)`	return a new block of size `n` words and tag `t`
`alloc_tuple(n)`	same, with tag 0
`alloc_string(n)`	return an uninitialized string of length `n` characters
`copy_string(s)`	return a string initialized with the C string `s`
`copy_double(d)`	return a block containing the double float `d`
`alloc_array(f, a)`	return a block representing an array, initialized by applying the conversion function `f` to each element of the C array of pointers `a`, null-terminated.
`copy_string_array(p)`	return a block representing an array of strings, obtained from the C string array `p` (of type `char **`), null-terminated.

Figure 12.11: Functions for allocating blocks.

function `alloc_array` takes an array of pointers `a`, terminated by a null pointer, and a conversion function `f` taking a pointer and returning a value. The result of `alloc_array` is an Objective Caml array containing the results of applying `f` in turn to each pointer in `a`. In the following example, the function `make_str_array` uses `alloc_array` to convert a C array of strings.

```
#include <caml/mlvalues.h>
value make_str (char *s) { return copy_string(s); }
value make_str_array (char **p) { return alloc_array(make_str,p) ; }
```

It is sometimes necessary to allocate blocks of size 0, for instance to represent an empty Objective Caml array. Such a block is called an *atom*.

```
# inspect [| |] ;;
....memory block: size=0  -  structured block (tag=0):
- : '_a array = [||]
```

Because atoms are allocated statically and do not reside in the dynamic part of the Objective Caml heap, the allocation functions in figure 12.11 must not be used to allocate atoms. Instead, atoms are created in C by the macro `Atom(t)`, where `t` is the desired tag for the block of size 0.

Storing C data in the Objective Caml heap

It is sometimes convenient to use the Objective Caml heap to store arbitrary C data that does not respect the constraints imposed by the garbage collector. In this case, blocks with tag `Abstract_tag` must be used.

A natural example is the manipulation of native C integers (of size 32 or 64 bits) in Objective Caml. Since these integers are not tagged as the Objective Caml garbage collector expects, they must be kept in one-word heap blocks with tag `Abstract_tag`.

```
#include <caml/mlvalues.h>
#include <stdio.h>

value Cint_of_OCAMLint (value v)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Long_val(v) ;
  return res ;
}

value OCAMLint_of_Cint (value v)  { return Val_long(Field(v,0)) ; }

value Cplus (value v1,value v2)
{
  value res = alloc(1,Abstract_tag) ;
  Field(res,0) = Field(v1,0) + Field(v2,0) ;
  return res ;
}

value printCint (value v)
{
  printf ("%d",(long) Field(v,0)) ; fflush(stdout) ;
  return Val_unit ;
}
```

```
# type cint
  external cint_of_int : int → cint = "Cint_of_OCAMLint"
  external int_of_cint : cint → int = "OCAMLint_of_Cint"
  external plus_cint : cint → cint → cint = "Cplus"
  external print_cint : cint → unit = "printCint" ;;
```

We can now work on native C integers, without losing the use of the tag bit, while remaining compatible with Objective Caml's garbage collector. However, such integers are heap-allocated, instead of being immediate values, which renders arithmetic operations less efficient.

```
# let a = 1000000000 ;;
val a : int = 1000000000
# a+a ;;
- : int = -147483648
# let c = let b = cint_of_int a in plus_cint b b ;;
val c : cint = <abstr>
# print_cint c ; print_newline () ;;
2000000000
- : unit = ()
# int_of_cint c ;;
- : int = -147483648
```

Finalization functions

Abstract blocks can also contain pointers to memory blocks allocated outside the Objective Caml heap. We know that Objective Caml blocks that are no longer used by the program are deallocated by the garbage collector. But what happens to a block allocated in the C heap and referenced by an abstract block that was reclaimed by the GC? To avoid memory leaks, we can associate a *finalization function* to the abstract block; this function is called by the GC before reclaiming the abstract block.

An abstract block with an attached finalization function is allocated via the function `alloc_final (n, f, used, max)` .

- `n` is the size of the block, in words. The first word of the block is used to store the finalization function; hence the size occupied by the user data must be increased by one word.
- `f` is the finalization function itself, with type `void f (value)`. It receives the abstract block as argument, just before this block is reclaimed by the GC.
- `used` represents the memory space (outside the Objective Caml heap) occupied by the C data. `used` must be ¡= `max`.
- `max` is the maximum memory space outside the Objective Caml heap that we tolerate not being reclaimed immediately.

For efficiency reasons, the Objective Caml garbage collector does not reclaim heap blocks as soon as they become unused, but some time later. The ratio `used/max` controls the proportion of finalized abstract blocks that the garbage collector may leave allocated while they are no longer used. A ratio of 0 (that is, `used = 0`) lets the garbage collector work at its usual pace; higher ratios (no greater than 1) cause it to work harder and spend more CPU time finding unused finalized blocks and reclaiming them.

The following program manipulates arrays of C integers allocated in the C heap via `malloc`. To allow the Objective Caml garbage collector to reclaim these arrays auto-

matically, the `create` function wraps them in a finalized abstract block, containing both a pointer to the array and the finalization function `finalize_it`.

```
#include <malloc.h>
#include <stdio.h>
#include <caml/mlvalues.h>

typedef struct {
        int size ;
        long * tab ; } IntTab ;

IntTab *alloc_it (int s)
{
  IntTab *res = malloc(sizeof(IntTab)) ;
  res->size = s ;
  res->tab = (long *) malloc(sizeof(long)*s) ;
  return res ;
}
void free_it (IntTab *p)  { free(p->tab) ; free(p) ; }
void put_it (int n,long q,IntTab *p)  { p->tab[n] = q ; }
long get_it (int n,IntTab *p)  { return p->tab[n]; }

void finalize_it (value v)
{
  IntTab *p = (IntTab *) Field(v,1) ;
  int i;
  printf("reclamation of an IntTab by finalization [") ;
  for (i=0;i<p->size;i++) printf("%d ",p->tab[i]) ;
  printf("]\n"); fflush(stdout) ;
  free_it ((IntTab *) Field(v,1)) ;
}
value create (value s)
{
  value block ;
  block = alloc_final (2, finalize_it,Int_val(s)*sizeof(IntTab),100000) ;
  Field(block,1) = (value) alloc_it(Int_val(s)) ;
  return block ;
}
value put (value n,value q,value t)
{
  put_it (Int_val(n), Long_val(q), (IntTab *) Field(t,1)) ;
  return Val_unit ;
}
value get (value n,value t)
{
  long res = get_it (Int_val(n), (IntTab *) Field(t,1)) ;
  return Val_long(res) ;
}
```

The C functions visible from Objective Caml are: `create`, `put` and `get`.

```
# type c_int_array
  external cia_create : int → c_int_array = "create"
  external cia_get : int → c_int_array → int = "get"
  external cia_put : int→ int → c_int_array → unit = "put" ;;
```

We can now manipulate our new data structure from Objective Caml:

```
# let tbl = cia_create 10 and tbl2 = cia_create 10
  in for i=0 to 9 do cia_put i (i*2) tbl done ;
     for i=0 to 9 do print_int (cia_get i tbl) ; print_string " " done ;
     print_newline () ;
     for i=0 to 9 do cia_put (9-i) (cia_get i tbl) tbl2 done ;
     for i=0 to 9 do print_int (cia_get i tbl2) ; print_string " " done ;;
0 2 4 6 8 10 12 14 16 18
18 16 14 12 10 8 6 4 2 0 - : unit = ()
```

We now force a garbage collection to check that the finalization function is called:

```
# Gc.full_major () ;;
reclaimation of an IntTab by finalization [18 16 14 12 10 8 6 4 2 0 ]
reclaimation of an IntTab by finalization [0 2 4 6 8 10 12 14 16 18 ]
- : unit = ()
```

In addition to freeing C heap blocks, finalization functions can also be used to close files, terminate processes, etc.

Garbage collection and C parameters and local variables

A C function can trigger a garbage collection, either during an allocation (if the heap is full), or voluntarily by calling `void Garbage_collection_function ()`.

Consider the following example. Can you spot the error?

```
#include <caml/mlvalues.h>
#include <caml/memory.h>

value identity (value x)
{
  Garbage_collection_function() ;
  return x;
}
```

```
# external id : 'a → 'a = "identity" ;;
external id : 'a -> 'a = "identity"
# id [1;2;3;4;5] ;;
- : int list = [538918066; 538918060; 538918054; 538918048; 538918042]
```

The list passed as parameter to `id`, hence to the C function `identity`, can be moved or reclaimed by the garbage collector. In the example, we forced a garbage collection, but any allocation in the Objective Caml heap could have triggered a garbage collection as well. The anonymous list passed to `id` was reclaimed by the garbage collector, because it is not reachable from the set of known roots. To avoid this, any C function that allocates anything in the Objective Caml heap must tell the garbage collector about the C function's parameters and local variables of type `value`. This is achieved by using the macros described next.

For parameters, these macros are used within the body of the C function as if they were additional declarations:

`CAMLparam1(v)`	:	for one parameter `v` of type `value`
`CAMLparam2(v1,v2)`	:	for two parameters
...		...
`CAMLparam5(v1,...,v5)`	:	for five parameters
`CAMLparam0 ;`	:	required when there are no `value` parameters.

If the C function has more than five `value` parameters, the first five are declared with the `CAMLparam5` macro, and the remaining parameters with the macros `CAMLxparam1`, ..., `CAMLxparam5`, used as many times as necessary to list all `value` parameters.

`CAMLparam5(v1,...,v5);`		
`CAMLxparam5(v6,...,v10);`		
`CAMLxparam2(v11,v12);`	:	for 12 parameters of type `value`

For local variables, these macros are used instead of normal C declarations of the variables. Local variables of type `value` must also be registered with the garbage collector, using the macros `CAMLlocal1`, ..., `CAMLlocal5`. An array of values is declared with `CAMLlocalN(tbl,n)` where `n` is the number of elements of the array `tbl`. Finally, to return from the C function, we must use the macro `CAMLreturn` instead of C's `return` construct.

Here is the corrected version of the previous example:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
value identity2 (value x)
{
  CAMLparam1(x) ;
  Garbage_collection_function() ;
  CAMLreturn x;
}
```

```
# external id : 'a → 'a = "identity2" ;;
external id : 'a -> 'a = "identity2"
# let a = id [1;2;3;4;5] ;;
val a : int list = [1; 2; 3; 4; 5]
```

We now obtain the expected result.

Calling an Objective Caml closure from C

To apply a closure (*i.e.* an Objective Caml function value) to one or several arguments from C, we can use the functions declared in the header file `callback.h`.

`callback(f,v)`	:	apply the closure `f` to the argument `v`,
`callback2(f,v1,v2)`	:	same, to two arguments,
`callback3(f,v1,v2,v3)`	:	same, to three arguments,
`callbackN(f,n,tbl)`	:	same, to `n` arguments stored in the array `tbl`.

All these functions return a `value`, which is the result of the application.

Registering Objective Caml functions with C

The `callback` functions require the Objective Caml function to be applied as a closure, that is, as a value that was passed as an argument to the C function. We can also register a closure from Objective Caml, giving it a name, then later refer to the closure by its name in a C function.

The function `register` from module `Callback` associates a name (of type `string`) with a closure or with any other Objective Caml value (of any type, that is, `'a`). This closure or value can be recovered from C using the C function `caml_named_value`, which takes a character string as argument and returns a pointer to the closure or value associated with that name, if it exists, or the null pointer otherwise.

An example is in order:

```
# let plus x y = x + y ;;
val plus : int -> int -> int = <fun>
# Callback.register "plus3_ocaml" (plus 3);;
- : unit = ()
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>

value plus3_C (value v)
{
  CAMLparam1(v);
  CAMLlocal1(f);
  f = *(caml_named_value("plus3_ocaml")) ;
  CAMLreturn callback(f,v) ;
}

# external plusC : int → int = "plus3_C" ;;
```

```
external plusC : int -> int = "plus3_C"
# plusC 1 ;;
- : int = 4
# Callback.register "plus3_ocaml" (plus 5);;
- : unit = ()
# plusC 1 ;;
- : int = 6
```

Do not confuse the declaration of a C function with **external** and the registration of an Objective Caml closure with the function `register`. In the former case, the declaration is static, the correspondence between the two names is established at link time. In the latter case, the binding is dynamic: the correspondence between the name and the closure is performed at run time. In particular, the name–closure binding can be modified dynamically by registering a different closure with the same name, thus modifying the behavior of C functions using that name.

Exception handling in C and in Objective Caml

Different languages have different mechanisms for raising and handling exceptions: C relies on `setjmp` and `longjmp`, while Objective Caml has built-in constructs for exceptions (**try ... with**, **raise**). Of course, these mechanisms are not compatible: they do not keep the same information when setting up a handler. It is extremely hard to safely implement the nesting of exception handlers of different kinds, while ensuring that an exception correctly "jumps over" handlers. For this reason, only Objective Caml exceptions can be raised and handled from C; `setjmp` and `longjmp` in C cannot be caught from Objective Caml, and must not be used to skip over Objective Caml code.

All functions and macros introduced in this section are defined in the header file `fail.h`.

Raising a predefined exception

From a C function, it is easy to raise one of the exceptions `Failure`, `Invalid_argument` or `Not_found` from the `Pervasives` module: just use the following functions.

`failwith(s)`	:	raise the exception `Failure(s)`
`invalid_argument(s)`	:	raise the exception `Invalid_argument(s)`
`raise_not_found()`	:	raise the exception `Not_found`

In the first two cases, `s` is a C string (`char *`) that ends up as the argument to the exception raised.

Raising a user-defined exception

A registration mechanism similar to that for closures enables user-defined exceptions to be raised from C. We must first register the exception using the `Callback` module's `register_exception` function. Then, from C, we retrieve the exception identifier using the `caml_named_value` function (see page 343). Finally, we raise the exception, using one of the following functions:

`raise_constant(e)`	raise the exception `e` with no argument,
`raise_with_arg(e,v)`	raise the exception `e` with the value `v` as argument,
`raise_with_string(e,s)`	same, but the argument is taken from the C string `s`.

Here is an example C function that raises an Objective Caml exception:

```
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/fail.h>

value divide (value v1,value v2)
{
  CAMLparam2(v1,v2);
  if (Long_val(v2) == 0)
    raise_with_arg(*caml_named_value("divzero"),v1) ;
  CAMLreturn Val_long(Long_val(v1)/Long_val(v2)) ;
}
```

And here is an Objective Caml transcript showing the use of that C function:

```
# external divide : int → int → int = "divide" ;;
external divide : int -> int -> int = "divide"
# exception Division_zero of int ;;
exception Division_zero of int
# Callback.register_exception "divzero" (Division_zero 0) ;;
- : unit = ()
# divide 20 4 ;;
- : int = 5
# divide 22 0 ;;
Uncaught exception: Division_zero(22)
```

Catching an exception

In a C function, we cannot catch an exception raised from another C function. However, we can catch Objective Caml exceptions arising from the application of an Objective Caml function (callback). This is achieved via the functions `callback_exn`,

`callback2_exn`, `callback3_exn` and `callbackN_exn`, which are similar to the standard `callback` functions, except that if the callback raises an exception, this exception is caught and returned as the result of the callback. The result value of the `callback_exn` functions must be tested with `Is_exception_result(v)`; this predicate returns "true" if the result value represents an uncaught exception, and "false" otherwise. The macro `Extract_exception(v)` returns the exception value contained in an exceptional result value.

The C function `divide_print` below calls the Objective Caml function `divide` using `callback2_exn`, and checks whether the result is an exception. If so, it prints a message and raises the exception again; otherwise it prints the result.

```
#include <stdio.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/callback.h>
#include <caml/fail.h>

value divide_print (value v1,value v2)
{
  CAMLparam2(v1,v2) ;
  CAMLlocal3(div,dbz,res) ;
  div = * caml_named_value("divide") ;
  dbz = * caml_named_value("div_by_0") ;
  res = callback2_exn (div,v1,v2) ;
  if (Is_exception_result(res))
    {
      value exn=Extract_exception(res);
      if (Field(exn,0)==dbz)  printf("division by 0\n") ;
      else printf("other exception\n");
      fflush(stdout);
      if (Wosize_val(exn)==1)  raise_constant(Field(exn,0)) ;
      else raise_with_arg(Field(exn,0),Field(exn,1)) ;
    }
  printf("result = %d\n",Long_val(res)) ;
  fflush(stdout) ;
  CAMLreturn Val_unit ;
}
```

```
# Callback.register "divide" (/) ;;
- : unit = ()
# Callback.register_exception "div_by_0" Division_by_zero ;;
- : unit = ()
# external divide_print : int → int → unit = "divide_print" ;;
external divide_print : int -> int -> unit = "divide_print"
# divide_print 42 3 ;;
result = 14
- : unit = ()
```

```
# divide_print 21 0 ;;
division by 0
Uncaught exception: Division_by_zero
```

As the examples above show, it is possible to raise an exception from C and catch it in Objective Caml, and also to raise an exception from Objective Caml and catch it in C. However, a C program cannot by itself raise and catch an Objective Caml exception.

Main program in C

Until now, the entry point of our programs was in Objective Caml; the program could then call C functions. Nothing prevents us from writing the entry point in C, and having the C code call Objective Caml functions when desired. To do this, the program must define the usual C `main` function. This function will then initialize the Objective Caml runtime system by calling the function `caml_main(char **)`, which takes as an argument the array of command-line arguments that corresponds to the `Sys.argv` array in Objective Caml. Control is then passed to the Objective Caml code using callbacks (see page 343).

Linking Objective Caml code with C

The Objective Caml compiler can output C object files (with extension `.o`) instead of Objective Caml object files (with extension `.cmo` or `.cmx`). All we need to do is set the `-output-obj` compiler flag.

```
ocamlc -output-obj files.ml
ocamlopt -output-obj.cmxa files.ml
```

From the Objective Caml source files, an object file with default name `camlprog.o` is produced.

The final executable is obtained by linking, using the C compiler, and adding the library `-lcamlrun` if the Objective Caml code was compiled to bytecode, or the library `-lasmrun` if it was compiled to native code.

```
cc camlprog.o filesC.o -lcamlrun
cc camlprog.o filesC.o -lasmrun
```

Calling Objective Caml functions from the C program is performed as described previously, via the `callback` functions. The only difference is that the initialization of the Objective Caml runtime system is performed via the function `caml_startup` instead of `caml_main`.

Exercises

Polymorphic Printing Function

We wish to define a printing function `print` with type *`'a -> unit`* able to print any Objective Caml value. To this end, we extend and improve the `inspect` function.

1. In C, write the function `print_ws` which prints Objective Caml as follows:
   - immediate values: as C integers;
   - strings: between quotes;
   - floats: as usual;
   - arrays of floats: between `[| |]`
   - closures: as `< code, env >`
   - everything else: as a tuple, between `( )`

   The function should handle structured types recursively.
2. To avoid looping on circular values, and to display sharing properly, modify this function to keep track of the addresses of heap blocks it has already seen. If an address appears several times, name it when it is first printed (`v = name`), and just print the name when this address is encountered again.
   (a) Define a data structure to record the addresses, determine when they occur several times, and associate a name with each address.
   (b) Traverse the value once first to determine all the addresses it contains and record them in the data structure.
   (c) The second traversal prints the value while naming addresses at their first occurrences.
   (d) Define the function `print` combining both traversals.

Matrix Product

1. Define an abstract type *`float_matrix`* for matrices of floating-point numbers.
2. Define a C type for these matrices.
3. Write a C function to convert values of type *`float array array`* to values of type *`float_matrix`*.
4. Write a C function performing the reverse conversion.
5. Add the C functions computing the sum and the product of these matrices.
6. Interface them with Objective Caml and use them.

Counting Words: Main Program in C

The Unix command `wc` counts the number of characters, words and lines in a file. The goal of this exercise is to implement this command, while counting repeated words only once.

1. Write the program `wc` in C. This program will simply count words, lines and characters in the file whose name is passed on the command line.
2. Write in Objective Caml a function `add_word` that uses a hash table to record how many times the function was invoked with the same character string as argument.
3. Write two functions `num_repeated_words` and `num_unique_words` counting respectively the number of word repetitions and the number of unique words, as determined from the hash table built by `add_word`.
4. Register the three previous functions so that they can be called from a C program.
5. Rewrite the main function of the `wc` program so that it prints the number of unique words instead of the number of words.
6. Write the `main` function and the commands required to compile this program as an Objective Caml program.
7. Write the `main` function and the commands required to compile this program as a C program.

Summary

This chapter introduced the interface between the Objective Caml language and the C language. This interface allows C functions to operate on Objective Caml values. Using abstract Objective Caml types, the converse is also possible. An important feature of this interface is the ability to use the Objective Caml garbage collector to perform automatic reclamation of values created in C. This interface supports the combination, in the same program, of components developed in the two languages. Finally, Objective Caml exceptions can be raised and (with some limitations) handled from C.

To Learn More

For a better understanding of the C language, especially argument passing and data representations, the book *C: a reference manual* [HS94] is highly recommended.

Concerning exceptions and garbage collection, several works add these missing features to C. The technical report [Rob89] describes an implementation of exceptions in C, based on open macros and on the `setjmp` and `longjmp` functions from the C library. Hans Boehm distributes a conservative collector with ambiguous roots that can be added (as a library) to any C program:

Link: http://www.hpl.hp.com/personal/Hans_Boehm/gc/

Concerning interoperability between Objective Caml and C, the tools described in this chapter are rather low-level and difficult to use. However, they give the programmer full

control on copying or sharing of data structures between the two languages. A higher-level tool called `CamlIDL` is available; it automatically generates the Objective Caml "stubs" (encapsulation functions) for calling C functions and converting data types. The C types and functions are described in a language called IDL (Interface Definition Language), similar to a subset of C++ and C. This description is then passed through the `CamlIDL` compiler, which generates the corresponding `.mli`, `.ml` and `.c` files. This tool is distributed from the following page:

Link: http://caml.inria.fr/camlidl/

Other interfaces exist between Objective Caml and languages other than C. They are available on the "Caml hump" page:

Link: http://caml.inria.fr/hump.html

They include several versions of interfaces with Fortran, and also an Objective Caml bytecode interpreter written in Java.

Finally, interoperability between Objective Caml and other languages can also be achieved via data exchanges between separate programs, possibly over the network. This approach is described in the chapter on distributed programming (see chapter 20).

13

Applications

This chapter presents two applications which seek to illustrate the use of the many different programming concepts presented previously in Part III.

The first application builds a library of graphic components, `Awi` (Application Window Interface). Next the library will be applied in a simple Francs to Euros converter. The components library reacts to user input by calling event handlers. Although this is a simple application algorithmically, it shows the benefits of using closures to structure the communication between components. Indeed the various event handlers share certain values via their environment. To appreciate the construction of `Awi` it is necessary to know the base library `Graphics` (see chapter 5, page 117).

The second application is a search for a least cost path in a directed graph. It uses Dijkstra's algorithm which calculates all the least cost paths from a source node to all the other nodes connected to this source. A cache mechanism implemented using a table of weak pointers (see page 265) is used to speed the search. The GC can free the elements of this table at any time but they can be recalculated as necessary. The graph visualization uses the simple button component of the `Awi` library for selecting the origin and destination nodes of the path sought. We then compare the efficiency of running the algorithm both with and without the cache. To facilitate timing measurements between the two versions a file with the description of the graph and the origin and destination nodes is passed as an argument to the search algorithm. Finally, a small graphical interface will be added to the search program.

Constructing a Graphical Interface

The implementation of a graphical interface for a program is a tedious job if the tools at your disposal are not powerful enough, as this is the case with the `Graphics` library. The user-friendliness of a program derives in part from its interface. To ease the task of creating a graphical interface we will start by creating a new library called `Awi` which

sits on top of `Graphics` and then we will use it as a simple module to help us construct the interface for an application.

This graphical interface manipulates *components*. A component is a region of the main window which can be displayed in a certain graphical context and can handle events that are sent to it. There are basically two kinds of components: simple components, such as a confirmation button or a text entry field, and *containers* which allow other components to be placed within them. A component can only be attached to a single container. Thus the interface of an application is built as a tree whose root corresponds to the main container (the graphics window), the nodes are also containers and the leaves are simple components or empty containers. This treelike structure helps us to propagate events arising from user interaction. If a container receives an event it checks whether one of its children can handle it, if so then it sends the event to that child, otherwise it deals with the event using its own handler.

The component is the essential element in this library. We define it as a record which contains details of size, a graphic context, the parent and child components along with functions for display and for handling events. Containers include a function for displaying their components. To define the *`component`* type, we build the types for the graphics context, for events and for initialization options. A graphical context is used to contain the details of "graphical styles" such as the colors of the background and foreground, the size of the characters, the current location of the component and the fonts that have been chosen. Then must we define the kinds of events which can be sent to the component. These are more varied than those in the `Graphics` library on which they are based. We include a simple option mechanism which helps us to configure graphics contexts or components. One implementation difficulty arises in positioning components within a container.

The general event handling loop receives physical events from the input function of the `Graphics` library, decides whether other events should be generated as a result of these physical events, and then sends them to the root container. We shall consider the following components: text display, buttons, list boxes, input regions and enriched components. Next we will show how the components are assembled to construct graphical interfaces, illustrating this with a program to convert between Francs and Euros. The various components of this application communicate with each other over a shared piece of state.

Graphics Context, Events and Options

Let's start by defining the base types along with the functions to initialize and modify graphics contexts, events and options. There is also an option type to help us parametrize the functions which create graphical objects.

Graphics Context

The graphics context allows us to keep track of the foreground and background colors, the font, its size, the current cursor position, and line width. This results in the following type.

```
type  g_context = {
      mutable bcol : Graphics.color;
      mutable fcol : Graphics.color;
      mutable font :  string;
      mutable font_size : int;
      mutable lw : int;
      mutable x : int;
      mutable y : int };;
```

The `make_default_context` function creates a new graphics context containing default values [1].

```
# let default_font = "fixed"
  let default_font_size = 12
  let make_default_context () =
    { bcol = Graphics.white; fcol = Graphics.black;
      font = default_font;
      font_size = default_font_size;
      lw = 1;
      x = 0; y = 0;};;
val default_font : string = "fixed"
val default_font_size : int = 12
val make_default_context : unit -> g_context = <fun>
```

Access functions for the individual fields allow us to retrieve their values without knowing the implementation of the type itself.

```
# let get_gc_bcol gc  = gc.bcol
  let get_gc_fcol gc  = gc.fcol
  let get_gc_font  gc  = gc.font
  let get_gc_font_size  gc  = gc.font_size
  let get_gc_lw gc  = gc.lw
  let get_gc_cur gc = (gc.x,gc.y);;
val get_gc_bcol : g_context -> Graphics.color = <fun>
val get_gc_fcol : g_context -> Graphics.color = <fun>
val get_gc_font : g_context -> string = <fun>
```

1. The name of the character font may vary according to the system being used.

```
val get_gc_font_size : g_context -> int = <fun>
val get_gc_lw : g_context -> int = <fun>
val get_gc_cur : g_context -> int * int = <fun>
```

The functions to modify those fields work on the same principle.

```
# let set_gc_bcol gc c = gc.bcol <- c
  let set_gc_fcol gc c = gc.fcol <- c
  let set_gc_font  gc f = gc.font <- f
  let set_gc_font_size  gc s = gc.font_size <- s
  let set_gc_lw gc i = gc.lw <- i
  let set_gc_cur gc (a,b) = gc.x<- a; gc.y<-b;;
val set_gc_bcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_fcol : g_context -> Graphics.color -> unit = <fun>
val set_gc_font : g_context -> string -> unit = <fun>
val set_gc_font_size : g_context -> int -> unit = <fun>
val set_gc_lw : g_context -> int -> unit = <fun>
val set_gc_cur : g_context -> int * int -> unit = <fun>
```

We can thus create new contexts, and read and write various fields of a value of the *g_context* type.

The use_gc function applies the data of a graphic context to the graphical window.

```
# let use_gc gc =
      Graphics.set_color (get_gc_fcol gc);
      Graphics.set_font (get_gc_font gc);
      Graphics.set_text_size (get_gc_font_size gc);
      Graphics.set_line_width (get_gc_lw gc);
      let (a,b) = get_gc_cur gc in Graphics.moveto a b;;
val use_gc : g_context -> unit = <fun>
```

Some data, such as the background color, are not directly used by the Graphics library and do not appear in the use_gc function.

Events

The Graphics library only contains a limited number of interaction events: mouse click, mouse movement and key press. We want to enrich the kind of event that arises from interaction by integrating events arising at the component level. To this end we define the type *rich_event*:

```
# type rich_event =
```

```
    MouseDown | MouseUp | MouseDrag | MouseMove
  | MouseEnter | MouseExit | Exposure
  | GotFocus | LostFocus | KeyPress | KeyRelease;;
```

To create such events it is necessary to keep a history of previous events. The `MouseDown` and `MouseMove` events correspond to mouse events (clicking and moving) which are created by `Graphics`. Other mouse events are created by virtue of either the previous event `MouseUp`, or the last component which handled a physical event `MouseExit`. The `Exposure` event corresponds to a request to redisplay a component. The concept of *focus* expresses that a given component is interested in a certain kind of event. Typically the input of text to a component which has grabbed the focus means that this component alone will handle `KeyPress` and `KeyRelease` events. A `MouseDown` event on a text input component hands over the input focus to it and takes it away from the component which had it before.

These new events are created by the event handling loop described on page 360.

Options

A graphical interface needs rules for describing the creation options for graphical objects (components, graphics contexts). If we wish to create a graphics context with a certain color it is currently necessary to construct it with the default values and then to call the two functions to modify the color fields in that context. In the case of more complex graphic objects this soon becomes tedious. Since we want to extend these options as we build up the components of the library, we need an "extensible" sum type. The only one provided by Objective Caml is the *exn* type used for exceptions. Because using *exn* for handling options would affect the clarity of our programs we will only use this type for real exceptions. Instead, we will simulate an extensible sum type using pseudo constructors represented by character strings. We define the type *opt_val* for the values of these options. An option is a tuple whose first element is the name of the option and the second its value. The *lopt* type encompasses a list of such options.

```
# type opt_val = Copt of Graphics.color | Sopt of string
                 | Iopt of int | Bopt of bool;;
# type lopt = (string * opt_val) list ;;
```

The decoding functions take as input a list of options, an option name and a default value. If the name belongs to the list then the associated value is returned, if not then we get the default value. We show here only the decoding functions for integers and booleans, the others work on the same principle.

```
# exception OptErr;;
exception OptErr
# let theInt lo name default =
    try
```

```
        match List.assoc name lo with
          Iopt i → i
        | _  → raise OptErr
      with Not_found → default;;
val theInt : ('a * opt_val) list -> 'a -> int -> int = <fun>
# let theBool lo name default =
      try
        match List.assoc name lo with
          Bopt b → b
        | _  → raise OptErr
      with Not_found → default;;
val theBool : ('a * opt_val) list -> 'a -> bool -> bool = <fun>
```

We can now write a function to create a graphics context using a list of options in the following manner:

```
# let set_gc gc lopt =
      set_gc_bcol gc (theColor lopt "Background" (get_gc_bcol gc));
      set_gc_fcol gc (theColor lopt "Foreground" (get_gc_fcol gc));
      set_gc_font gc (theString lopt "Font" (get_gc_font gc));
      set_gc_font_size gc (theInt lopt "FontSize" (get_gc_font_size gc));
      set_gc_lw gc (theInt lopt "LineWidth" (get_gc_lw gc));;
val set_gc : g_context -> (string * opt_val) list -> unit = <fun>
```

This allows us to ignore the order in which the options are passed in.

```
# let dc = make_default_context () in
      set_gc dc [ "Foreground", Copt Graphics.blue;
                  "Background", Copt Graphics.yellow];
      dc;;
- : g_context =
{bcol=16776960; fcol=255; font="fixed"; font_size=12; lw=1; x=0; y=0}
```

This results in a fairly flexible system which unfortunately partially evades the type system. The name of an option is of the type *string* and nothing prevents the construction of a nonexistant name. The result is simply that the value is ignored.

Components and Containers

The component is the essential building block of this library. We want to be able to create components and then easily assemble them to construct interfaces. They must be able to display themselves, to recognize an event destined for them, and to handle

that event. Containers must be able to receive events from other components or to hand them on. We assume that a component can only be added to one container.

Construction of Components

A value of type *component* has a size (`w` and `h`), an absolute position in the main window (`x` and `y`), a graphics context used when it is displayed (`gc`), a flag to show whether it is a container (`container`), a parent - if it is itself attached to a container (`parent`), a list of child components (`children`) and four functions to handle positioning of components. These control how children are positioned within a component (`layout`), how the component is displayed (`display`), whether any given point is considered to be within the area of the component (`mem`) and finally a function for event handling (`listener`) which returns `true` if the event was handled and `false` otherwise. The parameter of the `listener` is of type (type *rich_status*) and contains the name of the event the lowlevel event information coming from the `Graphics` module, information on the keyboard focus and the general focus, as well as the last component to have handled an event. So we arrive at the following mutually recursive declarations:

```
# type  component =
    { mutable info : string;
      mutable x : int;  mutable y : int;
      mutable w :int ; mutable h : int;
      mutable gc : g_context;
      mutable container : bool;
      mutable parent :   component list;
      mutable children :  component list;
      mutable layout_options : lopt;
      mutable layout : component →  lopt → unit;
      mutable display : unit  → unit;
      mutable mem : int * int → bool;
      mutable listener : rich_status  → bool }
  and rich_status =
    { re : rich_event;
      stat : Graphics.status;
      mutable key_focus : component;
      mutable gen_focus : component;
      mutable last : component};;
```

We access the data fields of a component with the following functions.

```
# let get_gc c = c.gc;;
val get_gc : component -> g_context = <fun>
# let is_container c = c.container;;
val is_container : component -> bool = <fun>
```

The following three functions define the default behavior of a component. The function to test whether a given mouse position applies to a given component (`in_rect`) checks that the coordinate is within the rectangle defined by the coordinates of the component.

The default display function (`display_rect`) fills the rectangle of the component with the background color found in the graphic context of that component. The default layout function (`direct_layout`) places components relatively within their containers. Valid options are `"PosX"` and `"PosY"`, corresponding to the coordinates relative to the container.

```
# let in_rect  c  (xp,yp)   =
    (xp >= c.x) && (xp < c.x + c.w) && (yp >= c.y) && (yp < c.y + c.h) ;;
val in_rect : component -> int * int -> bool = <fun>
# let display_rect c ()   =
    let gc = get_gc c in
      Graphics.set_color (get_gc_bcol gc);
      Graphics.fill_rect c.x c.y c.w c.h ;;
val display_rect : component -> unit -> unit = <fun>
# let direct_layout c c1 lopt =
    let px = theInt lopt "PosX" 0
    and py = theInt lopt "PosY" 0 in
      c1.x <- c.x + px; c1.y <- c.y + py ;;
val direct_layout :
  component -> component -> (string * opt_val) list -> unit = <fun>
```

It is now possible to define a component using the function `create_component` which takes width and height as parameters and uses the three preceding functions.

```
# let create_component iw ih  =
    let dc =
       {info="Anonymous";
        x=0; y=0; w=iw; h=ih;
        gc = make_default_context() ;
        container = false;
        parent = []; children = [];
        layout_options = [];
        layout = (fun a b → ());
        display = (fun () → ());
        mem = (fun s → false);
        listener = (fun s  → false);}
    in
      dc.layout <- direct_layout dc;
      dc.mem <- in_rect dc;
      dc.display <- display_rect dc;
      dc ;;
val create_component : int -> int -> component = <fun>
```

We then define the following empty component:

```
# let empty_component = create_component 0 0 ;;
```

This is used as a default value when we construct values which need to contain at least one component (for example a value of type *rich_status*).

Adding Child Components

The difficult part of adding a component to a container is how to position the component within the container. The `layout` field contains this positioning function. It takes a component (a child) and a list of options and calculates the new coordinates of the child within the container. Different options can be used according to the positioning function. We describe several layout functions when we talk about about the *panel* component (see *below*, page 366). Here we simply describe the mechanism for propagating the display function through the tree of components, coordinate changes, and propagating events. The propagation of actions makes intensive use of the `List.iter` function, which applies a function to all the elements of a list.

The function `change_coord` applies a relative change to the coordinates of a component and those of all its children.

```
# let rec change_coord  c (dx,dy)   =
    c.x <- c.x + dx; c.y <- c.y + dy;
    List.iter (fun s → change_coord s (dx,dy) ) c.children;;
val change_coord : component -> int * int -> unit = <fun>
```

The `add_component` function checks that the conditions for adding a component have been met and then joins the parent (`c`) and the child (`c1`). The list of positioning options is retained in the child component, which allows us to reuse them when the positioning function of the parent changes. The list of options passed to this function are those used by the positioning function. There are three conditions which need to be prohibited: the child component is already a parent, the parent is not a container or the child is too large for parent

```
# let add_component  c c1 lopt   =
    if c1.parent <> [] then failwith "add_component: already a parent"
    else
      if not (is_container c ) then
        failwith "add_component: not a container"
      else
        if (c1.x + c1.w > c.w) || (c1.y + c1.h > c.h)
        then failwith "add_component: bad position"
        else
          c.layout  c1 lopt;
          c1.layout_options <- lopt;
          List.iter (fun s → change_coord s (c1.x,c1.y)) c1.children;
          c.children <- c1 :: c.children;
          c1.parent <-  [c] ;;
val add_component : component -> component -> lopt -> unit = <fun>
```

The removal of a component from some level in the tree, implemented by the following function, entails both a change to the link between the parent and the child and also a change to the coordinates of the child and all its own children:

```
# let remove_component c c1 =
      c.children <- List.filter ((!=) c1)  c.children;
      c1.parent <- List.filter ((!=) c) c1.parent;
      List.iter (fun s → change_coord s (- c1.x, - c1.y)) c1.children;
      c1.x <- 0; c1.y <- 0;;
val remove_component : component -> component -> unit = <fun>
```

A change to the positioning function of a container depends on whether it has any children. If it does not the change is immediate. Otherwise we must first remove the children of the container, modify the container's positioning function and then add the components back in with the same options used when they were originally added.

```
# let set_layout f c =
    if c.children = [] then c.layout <- f
    else
      let ls = c.children in
        List.iter (remove_component c) ls;
        c.layout <- f;
        List.iter (fun s → add_component c s s.layout_options) ls;;
val set_layout : (component -> lopt -> unit) -> component -> unit = <fun>
```

This is why we kept the list of positioning options. If the list of options is not recognized by the new function it uses the defaults.

When a component is displayed, the display event must be propagated to its children. The container is displayed behind its children. The order of display of the children is unimportant because they never overlap.

```
# let rec display  c  =
    c.display ();
    List.iter (fun cx → display cx ) c.children;;
val display : component -> unit = <fun>
```

Event Handling

The handling of physical events (mouse click, key press, mouse movement) uses the `Graphics.wait_next_event` function (see page 132) which returns a physical status (of type `Graphics.status`) following any user interaction. This physical status is used to calculate a rich status (of type *rich_status*) containing the event type (of type *rich_event*), the physical status, the components possessing the keyboard focus and the general focus along with the last component which successfully handled such an event. The general focus is a component which accepts all events.

Next we describe the functions for the manipulating of rich events, the propagation of this status information to components for them to be handled, the creation of the information and the main event-handling loop.

Functions used on Status

The following functions read the values of the mouse position and the focus. Functions on focus need a further parameter: the component which is capturing or losing that focus.

```
# let get_event e = e.re;;
# let get_mouse_x e = e.stat.Graphics.mouse_x;;
# let get_mouse_y e = e.stat.Graphics.mouse_y;;
# let get_key e = e.stat.Graphics.key;;

# let has_key_focus e c = e.key_focus == c;;
# let take_key_focus e c = e.key_focus <- c;;
# let lose_key_focus e c = e.key_focus <- empty_component;;
# let has_gen_focus e c = e.gen_focus == c;;
# let take_gen_focus e c = e.gen_focus <- c;;
# let lose_gen_focus e c = e.gen_focus <- empty_component;;
```

Propagation of Events

A rich event is sent to a component to be handled. Analogous to the display mechanism discussed earlier, child components have priority over their parents for handling simple mouse movement. If a component receives status information associated with an event, it looks to see if it has a child which can handle it. If so, the child returns `true` otherwise `false`. If no child can handle the event, the parent component tries to use the function in its own `listener` field.

Status information coming from keyboard activity is propagated differently. The parent component looks to see if it possesses the keyboard focus, and if so it handles the event, otherwise it propagates to its children.

Some events are produced as a result of handling an initial event. For example, if one component captures the focus, then this means another has lost it. Such events are handled immediately by the target component. This is the same with the entry and exit events caused when the mouse is moved between different components.

The `send_event` function takes a value of type *`rich_status`* and a component. It returns a boolean indicating whether the event was handled or not.

```
# let rec send_event rs c =
    match get_event rs with
      MouseDown | MouseUp | MouseDrag | MouseMove →
       if c.mem(get_mouse_x rs, get_mouse_y rs) then
         if  List.exists (fun sun → send_event rs sun) c.children then true
         else ( if c.listener rs then (rs.last <-c; true) else false )
       else false
    | KeyPress | KeyRelease →
       if has_key_focus rs c then
```

```
          ( if c.listener rs then (rs.last<-c; true)
            else false )
        else List.exists (fun sun → send_event rs sun) c.children
    | _  → c.listener rs;;
val send_event : rich_status -> component -> bool = <fun>
```

Note that the hierarchical structure of the components is really a tree and not a cyclic graph. This guarantees that the recursion in the `send_event` function cannot cause an infinite loop.

Event Creation

We differentiate between two kinds of events: those produced by a physical action (such as a mouse click) and those which arise from some action linked with the previous history of the system (such as the movement of the mouse cursor out of the screen area occupied by a component). As a result we define two functions for creating rich events.

The function which deals with the former kind constructs a rich event out of two sets of physical status information:

```
# let compute_rich_event s0 s1  =
    if s0.Graphics.button <> s1.Graphics.button then
    begin
      if s0.Graphics.button then MouseDown else MouseUp
    end
    else if s1.Graphics.keypressed then KeyPress
    else if (s0.Graphics.mouse_x <> s1.Graphics.mouse_x  ) ||
            (s0.Graphics.mouse_y <> s1.Graphics.mouse_y  ) then
    begin
      if s1.Graphics.button then MouseDrag else MouseMove
    end
    else raise Not_found;;
val compute_rich_event : Graphics.status -> Graphics.status -> rich_event =
  <fun>
```

The function creating the latter kind of event uses the last two rich events:

```
# let send_new_events res0 res1 =
      if res0.key_focus <> res1.key_focus then
      begin
        ignore(send_event  {res1 with re = LostFocus} res0.key_focus);
        ignore(send_event  {res1 with re = GotFocus} res1.key_focus)
      end;
      if (res0.last <> res1.last) &&
         (( res1.re = MouseMove) || (res1.re = MouseDrag)) then
      begin
        ignore(send_event  {res1 with re = MouseExit} res0.last);
        ignore(send_event  {res1 with re = MouseEnter} res1.last )
```

```
      end;;
val send_new_events : rich_status -> rich_status -> unit = <fun>
```

We define an initial value for the *rich_event* type. This is used to initialize the history of the event loop.

```
# let initial_re =
    { re = Exposure;
      stat = { Graphics.mouse_x=0; Graphics.mouse_y=0;
               Graphics.key = ' ';
               Graphics.button = false;
               Graphics.keypressed = false };
      key_focus = empty_component;
      gen_focus = empty_component;
      last = empty_component } ;;
```

Event Loop

The event loop manages the sequence of interactions with a component, usually the ancestor component for all the components of the interface. It is supplied with two booleans indicating whether the interface should be redisplayed after every physical event has been handled (`b_disp`) and whether to handle mouse movement (`b_motion`). The final argument (`c`), is the root of the component tree.

```
# let loop b_disp b_motion  c =
    let res0 = ref initial_re  in
    try
      display c;
      while true do
        let lev = [Graphics.Button_down; Graphics.Button_up;
                   Graphics.Key_pressed] in
        let flev = if b_motion then (Graphics.Mouse_motion) :: lev
                   else lev in
        let s = Graphics.wait_next_event flev
        in
          let res1 = {!res0 with stat = s} in
           try
              let res2 = {res1 with
                          re = compute_rich_event !res0.stat res1.stat} in
                ignore(send_event res2 c);
                send_new_events !res0 res2;
                res0 := res2;
               if b_disp then display c
           with Not_found → ()
      done
    with e → raise e;;
val loop : bool -> bool -> component -> unit = <fun>
```

The only way out of this loop is when one of the handling routines raises an exception.

Test Functions

We define the following two functions to create by hand status information corresponding to mouse and keyboard events.

```
# let make_click e x y =
    {re = e;
     stat = {Graphics.mouse_x=x; Graphics.mouse_y=y;
             Graphics.key = ' '; Graphics.button = false;
             Graphics.keypressed = false};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component}

  let make_key e ch c =
    {re = e;
     stat = {Graphics.mouse_x=0; Graphics.mouse_y=0;
             Graphics.key = c; Graphics.button = false;
             Graphics.keypressed = true};
     key_focus = empty_component;
     gen_focus = empty_component;
     last = empty_component};;
val make_click : rich_event -> int -> int -> rich_status = <fun>
val make_key : rich_event -> 'a -> char -> rich_status = <fun>
```

We can now simulate the sending of a mouse event to a component for test purposes.

Defining Components

The various mechanisms for display, coordinate change and, propagating event are now in place. It remains for us to define some components which are both useful and easy to use. We can classify components into the following three categories:

- simple components which do not handle events, such as text to be displayed;
- simple components which handle events, such as text entry fields;
- containers and their various layout strategies.

Values are passed between components, or between a component and the application by modification of shared data. The sharing is implemented by closures which contain in their environment the data to be modified. Moreover, as the behavior of the component can change as a result of event handling, components also contain an internal state in the closures of their handling functions. For example the handling function for an input field has access to text while it is being written. To this end we implement components in the following manner:

- define a type to represent the internal state of the component;
- declare functions for the manipulation of this state;
- implement the functions for display, testing whether a coordinate is within the component and handling events;
- implement the function to create the component, thereby associating those closures with fields in the component;
- test the component by simulating the arrival of events.

Creation functions take a list of options to configure the graphics context. The calculation of the size of a component when it is created needs to make use of graphics context of the graphical window in order to determine the width of the text to be displayed.

We describe the implementation of the following components:

- simple text (`label`);
- simple container (`panel`);
- simple button (`button`);
- choice among a sequence of strings (`choice`);
- text entry field (`textfield`);
- rich component (`border`).

The Label Component

The simplest component, called a *label*, displays a string of characters on the screen. It does not handle events. We will start by describing the display function and then the creation function.

Display must take account of the foreground and background colors and the character font. It is the job of the `display_init` function to erase the graphical region of the component, select the foreground color and position the cursor. The function `display_label` displays the string passed as a parameter immediately after the call to `display_init`.

```
# let display_init c =
    Graphics.set_color (get_gc_bcol (get_gc c)); display_rect  c ();
    let gc= get_gc c in
      use_gc gc;
      let (a,b) = get_gc_cur gc in
        Graphics.moveto (c.x+a) (c.y+b)
  let display_label s c () =
      display_init c; Graphics.draw_string s;;
val display_init : component -> unit = <fun>
val display_label : string -> component -> unit -> unit = <fun>
```

As this component is very simple it is not necessary to create any internal state. Only the function `display_label` knows the string to be displayed, which is passed by the

creation function.

```
# let create_label s lopt =
    let gc = make_default_context () in   set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size s in
        let u = create_component w h  in
          u.mem <- (fun x → false);  u.display <- display_label s  u;
          u.info <- "Label"; u.gc <- gc;
          u;;
val create_label : string -> (string * opt_val) list -> component = <fun>
```

If we wish to change the colors of this component, we need to manipulate its graphic context directly.

The display of *label* `l1` below is depicted in figure 13.1.

```
# let courier_bold_24 = Sopt "*courier-bold-r-normal-*24*"
  and courier_bold_18  = Sopt "*courier-bold-r-normal-*18*";;
# let l1 = create_label "Login: " ["Font", courier_bold_24;
                                   "Background", Copt gray1];;
```

Figure 13.1: Displaying a *label*.

The panel Component, Containers and Layout

A *panel* is a graphical area which can be a container. The function which creates a panel is very simple. It augments the general function for creating components with a boolean indicating whether it is a container. The functions for testing location within the *panel* and for display are those assigned by default in the `create_component` function.

```
# let create_panel b  w h  lopt =
    let u = create_component w h   in
      u.container <- b;
      u.info <- if b then "Panel container" else "Panel";
      let gc = make_default_context () in set_gc gc lopt; u.gc <- gc;
      u;;
val create_panel :
  bool -> int -> int -> (string * opt_val) list -> component = <fun>
```

The tricky part with containers lies in the positioning of their child components. We define two new layout functions: `center_layout` and `grid_layout`. The first, `center_layout` places a component at the center of a container:

```
# let center_layout c c1 lopt =
      c1.x <- c.x + ((c.w -c1.w) /2); c1.y <- c.y + ((c.h -c1.h) /2);;
```

```
val center_layout : component -> component -> 'a -> unit = <fun>
```

The second, `grid_layout` divides a container into a grid where each box has the same size. The layout options in this case are `"Col"` for the column number and `"Row"` for the row number.

```
# let grid_layout (a, b)  c c1 lopt =
    let px = theInt lopt "Col" 0
    and py = theInt lopt "Row" 0 in
     if (px >= 0) && (px < a) && ( py >=0) && (py < b) then
       let lw = c.w /a
       and lh = c.h /b in
            if (c1.w > lw) || (c1.h > lh) then
              failwith "grid_placement: too big component"
            else
              c1.x <- c.x + px * lw + (lw - c1.w)/2;
              c1.y <- c.y + py * lh + (lh - c1.h)/2;
       else  failwith "grid_placement: bad position";;
val grid_layout :
  int * int -> component -> component -> (string * opt_val) list -> unit =
  <fun>
```

It is clearly possible to define more. One can also customize a container by changing its layout function (`set_layout`). Figure 13.2 shows a *panel*, declared as a container, in which two *labels* have been added and which corresponds to the following program:

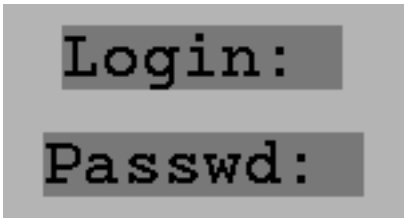


Figure 13.2: A *panel* component.

```
# let l2 = create_label "Passwd: " ["Font", courier_bold_24;
                                    "Background", Copt gray1] ;;
# let p1 = create_panel true 150 80 ["Background", Copt gray2] ;;
# set_layout  (grid_layout (1,2) p1) p1;;
# add_component p1 l1 ["Row", Iopt 1];;
# add_component p1 l2 ["Row", Iopt 0];;
```

Since the components need at least one parent so that they can be integrated into the interface, and since the `Graphics` library only supports one window, we must define a

principle window which is a container.

```
# let open_main_window w h =
    Graphics.close_graph();
    Graphics.open_graph (" "^(string_of_int w)^"x"^(string_of_int h));
    let u = create_component  w h in
       u.container <- true;
       u.info <- "Main Window";
       u;;
val open_main_window : int -> int -> component = <fun>
```

The Button Component

A *button* is a component which displays a text in its graphical region and reacts to mouse clicks which occur there. To support this behavior it retains a piece of state, a value of type *button_state*, which contains the text and the mouse handling function.

```
# type button_state =
   { txt : string; mutable action :  button_state → unit } ;;
```

The function `create_bs` creates this state. The `set_bs_action` function changes the handling function and the function `get_bs_text` retrieves the text of a button.

```
# let create_bs s = {txt = s; action = fun x → ()}
  let set_bs_action bs f = bs.action <- f
  let get_bs_text bs = bs.txt;;
val create_bs : string -> button_state = <fun>
val set_bs_action : button_state -> (button_state -> unit) -> unit = <fun>
val get_bs_text : button_state -> string = <fun>
```

The display function is similar to that used by *labels* with the exception that the text derives this time from the state information belonging to the button. By default the listening function activates the action function when a mouse button is pressed.

```
# let display_button  c bs  () =
     display_init c;  Graphics.draw_string (get_bs_text bs)
  let listener_button c bs  e = match get_event e with
    MouseDown → bs.action bs; c.display (); true
  | _ → false;;
val display_button : component -> button_state -> unit -> unit = <fun>
val listener_button : component -> button_state -> rich_status -> bool =
  <fun>
```

We now have all we need to define the creation function for simple buttons:

```
# let create_button s lopt     =
```

```
    let bs = create_bs s in
      let gc = make_default_context () in
        set_gc gc lopt; use_gc gc;
        let w,h = Graphics.text_size (get_bs_text bs) in
          let u = create_component w h   in
            u.display <- display_button u bs;
            u.listener <- listener_button u bs;
            u.info <- "Button";
            u.gc <- gc;
            u,bs;;
val create_button :
  string -> (string * opt_val) list -> component * button_state = <fun>
```

This returns a tuple of which the first element is the button which has been created and the second is the internal state of the button. The latter value is particularly useful if we want to change the action function of the button since the button state is not accessible via the button function.

Figure 13.3 shows a *panel* to which a button has been added. We have associated an action function which displays the string contained by the button on the standard output.

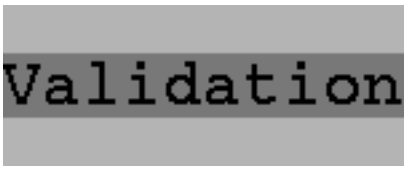


Figure 13.3: Button display and reaction to a mouseclick.

```
# let b,bs = create_button "Validation" ["Font", courier_bold_24;
                                         "Background", Copt gray1];;
# let p2 = create_panel true 150 60 ["Background", Copt gray2];;
# set_bs_action bs (fun bs → print_string ( (get_bs_text bs)^ "...");
                             print_newline());;
# set_layout  (center_layout p2) p2;;
# add_component p2 b  [];;
```

In contrast to *labels*, a button component knows how to react to a mouse click. To test this feature we send the event "mouse click" to a central position on the *panel* `p2`, which is occupied by the button. This causes the action associated with the button to be carried out:

```
# send_event (make_click MouseDown 75 30) p2;;
Validation...
- : bool = true
```

and returns the value `true` showing that the event has indeed been handled.

The choice Component

The *choice* component allows us to select one of the choices offered using a mouse click. There is always a current choice. A mouse click on another choice causes the current choice to change and causes an action to be carried out. We use the same technique we used previously for simple buttons. We start by defining the state needed by a choice list:

```
# type choice_state =
    { mutable ind : int; values : string array; mutable sep : int;
      mutable height : int; mutable action : choice_state → unit } ;;
```

The index `ind` shows which string is to be highlighted in the list of `values`. The `sep` and `height` fields describe in pixels the distance between two choices and the height of a choice. The action function takes an argument of type *choice_state* as an input and does its job using the index.

We now define the function to create a set of status information and the function to change to way it is handled.

```
# let create_cs sa = {ind = 0; values = sa; sep = 2;
                      height = 1; action = fun x → ()}
  let set_cs_action cs f = cs.action <- f
  let get_cs_text cs = cs.values.(cs.ind);;
val create_cs : string array -> choice_state = <fun>
val set_cs_action : choice_state -> (choice_state -> unit) -> unit = <fun>
val get_cs_text : choice_state -> string = <fun>
```

The display function shows the list of all the possible choices and accentuates the current choice in inverse video. The event handling function reacts to a release of the mouse button.

```
# let display_choice c cs  () =
     display_init c;
     let (x,y) = Graphics.current_point()
     and nb = Array.length cs.values in
       for i = 0 to nb-1 do
         Graphics.moveto x (y + i*(cs.height+ cs.sep));
         Graphics.draw_string cs.values.(i)
       done;
       Graphics.set_color (get_gc_fcol (get_gc c));
       Graphics.fill_rect x (y+ cs.ind*(cs.height+ cs.sep)) c.w cs.height;
       Graphics.set_color (get_gc_bcol (get_gc c));
       Graphics.moveto x  (y + cs.ind*(cs.height + cs.sep));
       Graphics.draw_string cs.values.(cs.ind) ;;
val display_choice : component -> choice_state -> unit -> unit = <fun>

# let listener_choice c cs e = match e.re with
    MouseUp →
```

```
      let x = e.stat.Graphics.mouse_x
      and y = e.stat.Graphics.mouse_y in
        let cy = c.y in
          let i = (y - cy) / ( cs.height + cs.sep) in
            cs.ind <- i; c.display ();
            cs.action cs; true
  |   _  → false ;;
val listener_choice : component -> choice_state -> rich_status -> bool =
  <fun>
```

To create a list of possible choices we take a list of strings and a list of options, and we return the component itself along with its internal state.

```
# let create_choice lc lopt  =
    let sa =  (Array.of_list (List.rev lc)) in
    let cs = create_cs sa in
    let gc = make_default_context () in
      set_gc gc lopt;  use_gc gc;
        let awh = Array.map (Graphics.text_size) cs.values in
        let w = Array.fold_right (fun (x,y) → max x)  awh 0
        and h = Array.fold_right (fun (x,y) → max y)  awh 0 in
        let h1 = (h+cs.sep) * (Array.length sa) + cs.sep  in
        cs.height <- h;
        let u = create_component w h1   in
          u.display <- display_choice u cs;
          u.listener <- listener_choice u cs ;
          u.info <- "Choice "^ (string_of_int (Array.length cs.values));
          u.gc <- gc;
          u,cs;;
val create_choice :
  string list -> (string * opt_val) list -> component * choice_state = <fun>
```

The sequence of three pictures in figure 13.4 shows a *panel* to which a list of choices has been added. To it we have bound an action function which displays the chosen string to the standard output. The pictures arise from mouse clicks simulated by the following program.

```
# let c,cs = create_choice ["Helium"; "Gallium"; "Pentium"]
                           ["Font", courier_bold_24;
                            "Background", Copt gray1];;
# let p3 = create_panel true 110 110  ["Background", Copt gray2];;
# set_cs_action cs (fun cs → print_string ( (get_cs_text cs)^"...");
                             print_newline());;
# set_layout  (center_layout p3) p3;;
# add_component p3 c [];;
```

Figure 13.4: Displaying and selecting from a choice list.

Here also we can test the component straight away by sending several events. The following changes the selection, as is shown in the central picture in figure 13.4.

```
# send_event (make_click MouseUp 60 55 ) p3;;
Gallium...
- : bool = true
```

The sending of the following event selects the first element in the choice list

```
# send_event (make_click MouseUp 60 90 ) p3;;
Helium...
- : bool = true
```

The textfield Component

The text input field, or *textfield*, is an area which enables us to input a text string. The text can be aligned to the left or (typically for a calculator) the right. Furthermore a cursor shows where the next character will be entered. Here we need a more complex internal state. This includes the text which is being entered, the direction of the justification, a description of the cursor, a description of how the characters are displayed and the action function.

```
# type textfield_state =
    { txt : string;
      dir : bool; mutable ind1 : int; mutable ind2 : int; len : int;
      mutable visible_cursor : bool; mutable cursor : char;
      mutable visible_echo : bool; mutable echo : char;
      mutable action : textfield_state → unit } ;;
```

To create this internal state we need the initial text, the number of characters available for the text input field and the justification of the text.

```
# let create_tfs txt size dir  =
    let l = String.length txt in
      (if size < l then failwith "create_tfs");
      let ind1 = if dir then 0 else size-1-l
```

```
      and ind2 = if dir then l else size-1 in
      let n_txt = (if dir then (txt^(String.make (size-l) ' '))
                          else ((String.make (size-l) ' ')^txt )) in
    {txt = n_txt; dir=dir; ind1 = ind1; ind2 = ind2; len=size;
     visible_cursor  = false;  cursor = ' '; visible_echo =  true; echo = ' ';
     action= fun x → ()};;
val create_tfs : string -> int -> bool -> textfield_state = <fun>
```

The following functions allow us to access various fields, including the displayed text.

```
# let set_tfs_action tfs f = tfs.action <- f
  let set_tfs_cursor b c tfs =  tfs.visible_cursor <- b; tfs.cursor <- c
  let set_tfs_echo b c tfs =  tfs.visible_echo <- b; tfs.echo <- c
  let get_tfs_text tfs =
    if tfs.dir then String.sub tfs.txt tfs.ind1 (tfs.ind2 - tfs.ind1)
    else String.sub tfs.txt (tfs.ind1+1) (tfs.ind2 - tfs.ind1);;
```

The `set_tfs_text` function changes the text within the internal state `tfs` of the component `tf` with the string `txt`.

```
# let set_tfs_text tf tfs txt =
     let l = String.length txt in
       if l > tfs.len then failwith "set_tfs_text";
       String.blit  (String.make tfs.len ' ') 0 tfs.txt 0 tfs.len;
       if tfs.dir then (String.blit txt 0 tfs.txt 0 l;
                    tfs.ind2 <- l )
       else   ( String.blit txt 0 tfs.txt (tfs.len -l) l;
                tfs.ind1 <- tfs.len-l-1 );
       tf.display ();;
val set_tfs_text : component -> textfield_state -> string -> unit = <fun>
```

Display operations must take account of how the character is echoed and the visibility of the cursor. The `display_textfield` function calls the `display_cursor` function which shows where the cursor is.

```
# let display_cursor c tfs =
    if tfs.visible_cursor then
      ( use_gc (get_gc c);
        let (x,y) = Graphics.current_point() in
        let (a,b) = Graphics.text_size " " in
        let shift =  a *  (if tfs.dir then max (min (tfs.len-1) tfs.ind2)  0
                           else tfs.ind2) in
          Graphics.moveto (c.x+x + shift) (c.y+y);
          Graphics.draw_char tfs.cursor);;
val display_cursor : component -> textfield_state -> unit = <fun>
# let display_textfield c tfs  () =
     display_init c;
     let s = String.make tfs.len ' '
```

```
      and txt = get_tfs_text tfs in
        let nl = String.length txt in
        if (tfs.ind1 >= 0) && (not tfs.dir) then
           Graphics.draw_string (String.sub s 0 (tfs.ind1+1) );
        if tfs.visible_echo  then (Graphics.draw_string (get_tfs_text tfs))
        else Graphics.draw_string (String.make (String.length txt) tfs.echo);
        if (nl > tfs.ind2) && (tfs.dir)
          then Graphics.draw_string (String.sub s tfs.ind2 (nl-tfs.ind2));
        display_cursor c tfs;;
val display_textfield : component -> textfield_state -> unit -> unit = <fun>
```

The event-listener function for this kind of component is more complex. According to the input direction (left or right justified) we may need to move the string which has already been input. Capture of focus is achieved by a mouse click in the input zone.

```
# let listener_text_field u tfs e =
       match e.re with
         MouseDown → take_key_focus e u ; true
       | KeyPress →
           ( if Char.code (get_key e)  >= 32 then
             begin
               ( if tfs.dir then
                 ( ( if tfs.ind2 >= tfs.len then (
                       String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2-1);
                       tfs.ind2 <- tfs.ind2-1) );
                     tfs.txt.[tfs.ind2] <- get_key e;
                     tfs.ind2 <- tfs.ind2 +1 )
                 else
                 ( String.blit tfs.txt 1 tfs.txt 0 (tfs.ind2);
                   tfs.txt.[tfs.ind2] <- get_key e;
                   if tfs.ind1 >= 0 then tfs.ind1 <- tfs.ind1 -1
                 );
               )
             end
             else (
             ( match Char.code (get_key e) with
                 13 → tfs.action tfs
               |  9 → lose_key_focus e u
               |  8 → if (tfs.dir && (tfs.ind2 > 0))
                         then tfs.ind2 <- tfs.ind2 -1
                         else if (not tfs.dir) && (tfs.ind1 < tfs.len -1)
                              then tfs.ind1 <- tfs.ind1+1
               | _ → ()
             ))); u.display(); true
       | _ → false;;
val listener_text_field :
  component -> textfield_state -> rich_status -> bool = <fun>
```

The function which creates text entry fields repeats the same pattern we have seen in the previous components.

```
# let create_text_field  txt size dir lopt  =
    let tfs = create_tfs txt size dir
    and l = String.length txt in
     let gc = make_default_context () in
      set_gc gc lopt; use_gc gc;
      let (w,h) = Graphics.text_size (tfs.txt) in
        let u = create_component w h   in
          u.display <- display_textfield u tfs;
          u.listener <-  listener_text_field u tfs ;
          u.info <- "TextField"; u.gc <- gc;
          u,tfs;;
val create_text_field :
  string ->
  int -> bool -> (string * opt_val) list -> component * textfield_state =
  <fun>
```

This function returns a tuple consisting of the component itself, and the internal state of that component. We can test the creation of the component in figure 13.5 as follows:

```
# let tf1,tfs1 = create_text_field "jack"  8 true ["Font", courier_bold_24];;
# let tf2,tfs2 = create_text_field "koala" 8  false ["Font", courier_bold_24];;
# set_tfs_cursor true '_' tfs1;;
# set_tfs_cursor true '_' tfs2;;
# set_tfs_echo false '*' tfs2;;
# let p4 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p4) p4;;
# add_component p4 tf1 ["Row", Iopt 1];;
# add_component p4 tf2 ["Row", Iopt 0];;
```

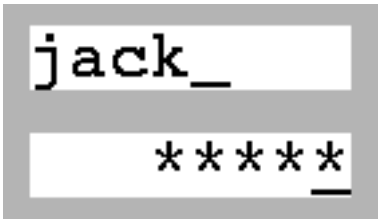


Figure 13.5: Text input component.

Enriched Components

Beyond the components described so far, it is also possible to construct new ones, for example components with bevelled edges such as those in the calculator on page 136. To create this effect we construct a *panel* larger than the component, fill it out in a certain way and add the required component to the center.

```
# type border_state =
    {mutable relief : string; mutable line : bool;
     mutable bg2 : Graphics.color; mutable size : int};;
```

The creation function takes a list of options and constructs an internal state.

```
# let create_border_state lopt =
    {relief = theString lopt "Relief" "Flat";
     line = theBool lopt "Outlined" false;
     bg2 = theColor lopt "Background2" Graphics.black;
     size = theInt lopt "Border_size" 2};;
val create_border_state : (string * opt_val) list -> border_state = <fun>
```

We define the profile of the border used in the boxes of figure 5.6 (page 130) by defining the options `"Top"`, `"Bot"` and `"Flat"`.

```
# let display_border bs c1 c () =
    let x1 = c.x  and y1 = c.y in
    let x2 = x1+c.w-1 and y2 = y1+c.h-1 in
    let ix1 = c1.x and iy1 =  c1.y in
    let ix2 = ix1+c1.w-1 and iy2 = iy1+c1.h-1 in
    let border1 g = Graphics.set_color g;
      Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix2,iy1);(x2,y1) |] ;
      Graphics.fill_poly [| (x2,y1);(ix2,iy1);(ix2,iy2);(x2,y2) |]
    in
      let border2 g =  Graphics.set_color g;
        Graphics.fill_poly [| (x1,y2);(ix1,iy2);(ix2,iy2);(x2,y2) |] ;
        Graphics.fill_poly [| (x1,y1);(ix1,iy1);(ix1,iy2);(x1,y2) |]
      in
        display_rect c ();
        if bs.line then (Graphics.set_color (get_gc_fcol (get_gc c));
                         draw_rect x1 y1 c.w c.h);
       let b1_col = get_gc_bcol ( get_gc c)
       and b2_col = bs.bg2 in
        match bs.relief with
          "Top" →  (border1 b1_col; border2 b2_col)
        |  "Bot" → (border1 b2_col; border2 b1_col)
        |  "Flat" →  (border1 b1_col; border2 b1_col)
        |  s → failwith ("display_border: unknown relief: "^s)
  ;;
val display_border : border_state -> component -> component -> unit -> unit =
  <fun>
```

The function which creates a border takes a component and a list of options, it constructs a *panel* containing that component.

```
# let create_border c lopt =
      let bs = create_border_state lopt in
      let p = create_panel true (c.w + 2 * bs.size)
                                (c.h + 2 * bs.size) lopt in
        set_layout (center_layout p) p;
        p.display <- display_border bs c p;
        add_component p c [] ; p;;
val create_border : component -> (string * opt_val) list -> component = <fun>
```

Now we can test creating a component with a border on the *label* component and the text entry field `tf1` defined by in our previous tests. The result is show in figure 13.6.

```
# remove_component   p1 l1;;
# remove_component   p4 tf1;;
# let b1 = create_border l1 [] ;;
# let b2 = create_border tf1 ["Relief", Sopt "Top";
                              "Background", Copt Graphics.red;
                              "Border_size", Iopt 4];;
# let p5 = create_panel true 140 80 ["Background", Copt gray2];;
# set_layout  (grid_layout (1,2) p5) p5;;
# add_component p5 b1 ["Row", Iopt 1];;
# add_component p5 b2 ["Row", Iopt 0];;
```

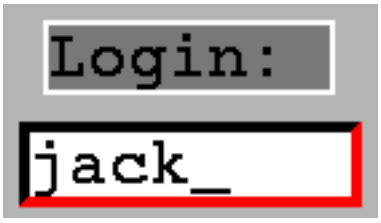


Figure 13.6: An enriched component.

Setting up the `Awi` *Library*

The essential parts of our library have now been written. All declarations [2] of types and values which we have seen so far in this section can be grouped together in one file. This library consists of one single module. If the file is called `awi.ml` then we get

2. except for those used in our test examples

a module called `Awi`. The link between the name of the file and that of the module is described in chapter 14.

Compiling this file will produce a compiled interface file `awi.cmi` and, depending on the compiler being used, the bytecode itself `awi.cmo` or else the native machine code `awi.cmx`. To use the bytecode compiler we enter the following command

```
ocamlc -c awi.ml
```

To use it at the interactive toplevel, we need to load the bytecode of our new library with the command `#load "awi.cmo";;` having also previously ensured that we have loaded the `Graphics` library. We can then start calling functions from the module to create and work with components.

```
# open Awi;;
# create_component;;
- : int -> int -> Awi.component = <fun>
```

The result type of this function is `Awi.component`, chapter 14 explains more about this.

Example: A Franc-Euro Converter

We will now build a currency converter between Francs and Euros using this new library. The actual job of conversion is trivial, but the construction of the interface will show how the components communicate with each other. While we are getting used to the new currency we need to convert in both directions. Here are the components we have chosen:

- a list of two choices to describe the direction of the conversion;
- two text entry fields for inputting values and displaying converted results;
- a simple button to request that the calculation be performed;
- two *labels* to show the meaning of each text entry field.

These different components are shown in figure 13.7.

Communication between the components is implemented by sharing state. For this purpose we define the type *state_conv* which hold the fields for francs (`a`), euros (`b`), the direction in which the conversion is to be performed (`dir`) and the conversion factors (`fa` and `fb`).

```
# type state_conv =
    { mutable a:float; mutable b:float; mutable dir : bool;
      fa : float; fb : float } ;;
```

We define the initial state as follows:

```
# let e =  6.55957074
```

```
  let fe = { a =0.0; b=0.0; dir = true; fa = e; fb = 1./. e};;
```

The conversion function returns a floating result following the direction of the conversion.

```
# let calculate fe =
    if fe.dir then fe.b <- fe.a /. fe.fa else fe.a <- fe.b /. fe.fb;;
val calculate : state_conv -> unit = <fun>
```

A mouse click on the list of two choices changes the direction of the conversion. The text of the choice strings is `"->"` and `"<-"`.

```
# let action_dir fe cs = match get_cs_text cs with
    "->" → fe.dir <- true
  | "<-" → fe.dir <- false
  | _ → failwith "action_dir";;
val action_dir : state_conv -> choice_state -> unit = <fun>
```

The action associated with the simple button causes the calculation to be performed and displays the result in one of the two text entry fields. For this to be possible we pass the two text entry fields as parameters to the action.

```
# let action_go fe tf_fr tf_eu tfs_fr tfs_eu  x =
    if fe.dir then
      let r = float_of_string (get_tfs_text tfs_fr) in
        fe.a <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.b in
            set_tfs_text tf_eu tfs_eu  sr
    else
      let r = float_of_string (get_tfs_text tfs_eu) in
        fe.b <- r; calculate fe;
        let sr = Printf.sprintf "%.2f" fe.a in
            set_tfs_text tf_fr tfs_fr sr;;
val action_go :
  state_conv ->
  component -> component -> textfield_state -> textfield_state -> 'a -> unit =
  <fun>
```

It now remains to build the interface. The following function takes a width, a height and a conversion state and returns the main container with the three active components.

```
# let create_conv w h fe =
  let gray1 = (Graphics.rgb 120 120 120) in
  let m = open_main_window w h
  and  p = create_panel true (w-4) (h-4)  []
  and  l1 = create_label "Francs" ["Font", courier_bold_24;
                                   "Background", Copt gray1]
```

```
  and l2 = create_label "Euros" ["Font", courier_bold_24;
                                 "Background", Copt gray1]
  and c,cs = create_choice ["->"; "<-"] ["Font", courier_bold_18]
  and tf1,tfs1 = create_text_field  "0" 10 false ["Font", courier_bold_18]
  and tf2,tfs2 = create_text_field "0" 10 false ["Font", courier_bold_18]
  and b,bs = create_button " Go " ["Font", courier_bold_24]
  in
    let gc = get_gc m in
      set_gc_bcol gc gray1;
      set_layout (grid_layout (3,2) m ) m;
      let tb1 = create_border tf1  []
      and tb2 = create_border tf2  []
      and bc = create_border c  []
      and bb =
        create_border  b
          ["Border_size", Iopt 4; "Relief", Sopt "Bot";
           "Background", Copt gray2; "Background2", Copt Graphics.black]
      in
        set_cs_action cs (action_dir fe);
        set_bs_action bs (action_go fe tf1 tf2 tfs1 tfs2);
        add_component m l1 ["Col",Iopt 0;"Row",Iopt 1];
        add_component m l2 ["Col",Iopt 2;"Row",Iopt 1];
        add_component m bc ["Col",Iopt 1;"Row",Iopt 1];
        add_component m tb1 ["Col",Iopt 0;"Row",Iopt 0];
        add_component m tb2 ["Col",Iopt 2;"Row",Iopt 0];
        add_component m bb  ["Col",Iopt 1;"Row",Iopt 0];
        m,bs,tf1,tf2;;
val create_conv :
  int ->
  int -> state_conv -> component * button_state * component * component =
  <fun>
```

The event handling loop is started on the container `m` constructed below. The resulting display is shown in figure 13.7.

```
# let (m,c,t1,t2) = create_conv 420 150 fe ;;
# display m ;;
```

One click on the choice list changes both the displayed text and the direction of the conversion because all the event handling closures share the same state.

Where to go from here

Closures allow us to register handling methods with graphical components. It is however impossible to "reopen" these closures to extend an existing handler with additional behavior. We need to define a completely new handler. We discuss the possibilities for

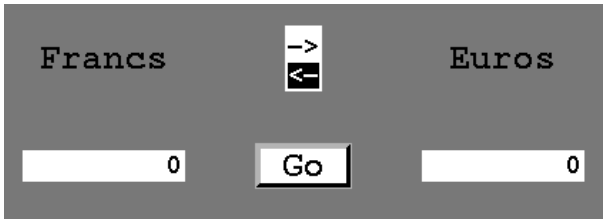


Figure 13.7: Calculator window.

extending handlers in chapter 16 where we compare the functional and object-oriented paradigms.

In our application many of the structures declared have fields with identical names (for example `txt`). The last declaration masks all previous occurences. This means that it becomes difficult to use the field names directly and this is why we have declared a set of access functions for every type we have defined. Another possibility would be to cut our library up into several modules. From then on field names could be disambiguated by using the module names. Nonetheless, with the help of the access functions, we can already make full use of the library. Chapter 14 returns to the topic of type overlaying and introduces abstract data types. The use of overlaying can, among other things, increase robustness by preventing the modification of sensitive data fields, such as the parent child relationships between the components which should not allow the construction of a circular graph.

There are many possible ways to improve this library.

One criterion in our design for components was that it should be possible to write new ones. It is fairly easy to create components of an arbitrary shape by using new definitions of the `mem` and `display` functions. In this way one could create buttons which have an oval or tear-shaped form.

The few layout algorithms presented are not as helpful as they could be. One could add a grid layout whose squares are of variable size and width. Or maybe we want to place components alongside each other so long as there is enough room. Finally we should anticipate the possibility that a change to the size of a component may be propagated to its children.

Finding Least Cost Paths

Many applications need to find least cost paths through weighted directed graphs. The problem is to find a path through a graph in which non-negative weights are associated with the arcs. We will use Dijkstra's algorithm to determine the path.

This will be an opportunity to use several previously introduced libraries. In the order of appearance, the following modules will be used: `Genlex` and `Printf` for input and

output, the module `Weak` to implement a cache, the module `Sys` to track the time saved by a cache, and the `Awi` library to construct a graphical user interface. The module `Sys` is also used to construct a standalone application that takes the name of a file describing the graph as a command line argument.

*Graph Represention*s

A weighted directed graph is defined by a set of nodes, a set of edges, and a mapping from the set of edges to a set of values. There are numerous implementations of the data type *weighted directed graph*.

- adjacency matrices:
  each element $(m(i,j))$ of the matrix represents an edge from node i to j and the value of the element is the weight of the edge;
- adjacency lists:
  each node i is associated with a list $[(j_1,w_1);..;(j_n,w_n)]$ of nodes and each triple (i,j_k,w_k) is an edge of the graph, with weight w_k;
- a triple:
  a list of nodes, a list of edges and a function that computes the weights of the edges.

The behavior of the representations depends on the size and the number of edges in the graph. Since one goal of this application is to show how to cache certain previously executed computations without using all of memory, an adjacency matrix is used to represent weighted directed graphs. In this way, memory usage will not be increased by list manipulations.

```
# type  cost  = Nan | Cost of float;;
# type  adj_mat =  cost array array;;
# type 'a graph = { mutable ind : int;
                    size : int;
                    nodes : 'a array;
                    m : adj_mat};;
```

The field `size` indicates the maximal number of nodes, the field `ind` the actual number of nodes.

We define functions to let us create graphs edge by edge.

The function to create a graph takes as arguments a node and the maximal number of nodes.

```
# let create_graph n s =
    { ind = 0; size = s; nodes = Array.create s n;
      m = Array.create_matrix s s Nan } ;;
val create_graph : 'a -> int -> 'a graph = <fun>
```

The function `belongs_to` checks if the node `n` is contained in the graph `g`.

```
# let belongs_to n g =
    let rec aux i =
      (i < g.size) & ((g.nodes.(i) = n) or (aux (i+1)))
    in aux 0;;
val belongs_to : 'a -> 'a graph -> bool = <fun>
```

The function `index` returns the index of the node `n` in the graph `g`. If the node does not exist, a `Not_found` exception is thrown.

```
# let index n g =
    let rec aux i =
      if i >= g.size then raise Not_found
      else if g.nodes.(i) = n then i
           else aux (i+1)
    in aux 0 ;;
val index : 'a -> 'a graph -> int = <fun>
```

The next two functions are for adding nodes and edges of cost `c` to graphs.

```
# let add_node n g =
    if g.ind = g.size then failwith "the graph is full"
    else if belongs_to n g then failwith "the node already exists"
    else (g.nodes.(g.ind) <- n; g.ind <- g.ind + 1) ;;
val add_node : 'a -> 'a graph -> unit = <fun>
# let add_edge e1 e2 c g =
    try
      let x = index e1 g and y = index e2 g in
        g.m.(x).(y) <- Cost c
    with Not_found → failwith "node does not exist" ;;
val add_edge : 'a -> 'a -> float -> 'a graph -> unit = <fun>
```

Now it is easy to create a complete weighted directed graph starting with a list of nodes and edges. The function `test_aho` constructs the graph of figure 13.8:

```
# let test_aho () =
    let g = create_graph "nothing" 5 in
      List.iter (fun x → add_node x g) ["A"; "B"; "C"; "D"; "E"];
      List.iter (fun (a,b,c) → add_edge a b c g)
        ["A","B",10.;
         "A","D",30.;
         "A","E",100.0;
         "B","C",50.;
         "C","E",10.;
         "D","C",20.;
         "D","E",60.];
      for i=0 to g.ind -1 do g.m.(i).(i) <- Cost 0.0 done;
       g;;
```

```
val test_aho : unit -> string graph = <fun>
# let a = test_aho();;
val a : string graph =
  {ind=5; size=5; nodes=[|"A"; "B"; "C"; "D"; "E"|];
   m=[|[|Cost 0; Cost 10; Nan; Cost 30; Cost ...|]; ...|]}
```

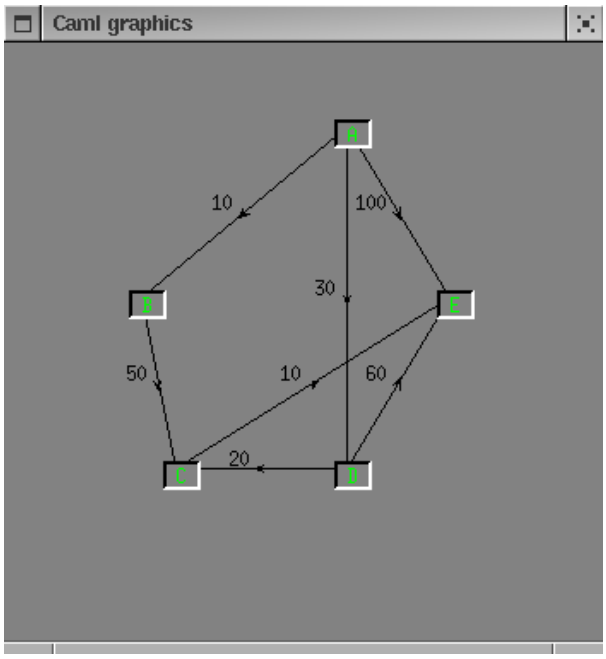


Figure 13.8: The test graph

Constructing Graphs

It is tedious to directly construct graphs in a program. To avoid this, we define a concise textual representation for graphs. We can define the graphs in text files and construct them in applications by reading the text files.

The textual representation for a graph consists of lines of the following forms:

- the number of nodes: `SIZE number`;
- the name of a node: `NODE name`;
- the cost of an edge: `EDGE name1 name2 cost`;
- a comment: `# comment`.

For example, the following file, `aho.dat`, describes the graph of figure 13.8 :

```
SIZE 5
```

```
NODE A
NODE B
NODE C
NODE D
NODE E
EDGE A B 10.0
EDGE A D 30.0
EDGE A E 100.0
EDGE B C 50.
EDGE C E 10.
EDGE D C 20.
EDGE D E 60.
```

To read graph files, we use the lexical analysis module `Genlex`. The lexical analyser is constructed from a list of keywords `keywords`.

The function `parse_line` executes the actions associated to the key words by modifying the reference to a graph.

```
# let keywords = [ "SIZE"; "NODE"; "EDGE"; "#"];;
val keywords : string list = ["SIZE"; "NODE"; "EDGE"; "#"]
# let lex_line l = Genlex.make_lexer keywords (Stream.of_string l);;
val lex_line : string -> Genlex.token Stream.t = <fun>
# let parse_line g s = match  s with parser
    [< '(Genlex.Kwd  "SIZE"); '(Genlex.Int n) >] →
        g := create_graph "" n
  | [< '(Genlex.Kwd  "NODE"); '(Genlex.Ident name) >] →
        add_node name !g
  | [< '(Genlex.Kwd  "EDGE"); '(Genlex.Ident e1);
       '(Genlex.Ident  e2); '(Genlex.Float c) >] →
        add_edge e1 e2 c !g
  | [< '(Genlex.Kwd  "#") >] → ()
  | [<>] → () ;;
val parse_line : string graph ref -> Genlex.token Stream.t -> unit = <fun>
```

The analyzer is used to define the function creating a graph from the description in the file:

```
# let create_graph name =
    let g = ref {ind=0; size=0; nodes =[||]; m = [||]} in
    let ic = open_in name in
     try
       print_string ("Loading "^name^": ");
       while true do
        print_string ".";
         let l = input_line ic in parse_line g (lex_line l)
       done;
       !g
     with End_of_file → print_newline(); close_in ic; !g ;;
```

```
val create_graph : string -> string graph = <fun>
```
The following command constructs a graph from the file `aho.dat`.
```
# let b = create_graph "PROGRAMMES/aho.dat" ;;
Loading PROGRAMMES/aho.dat: ..............
val b : string graph =
  {ind=5; size=5; nodes=[|"A"; "B"; "C"; "D"; "E"|];
   m=[|[|Nan; Cost 10; Nan; Cost 30; Cost 100|]; ...|]}
```

Dijkstra's Algorithm

Dijkstra's algorithm finds a least cost path between two nodes. The cost of a path between node $n1$ and node $n2$ is the sum of the costs of the edges on that path. The algorithm requires that costs always be positive, so there is no benefit in passing through a node more than once.

Dijkstra's algorithm effectively computes the minimal cost paths of all nodes of the graph which can be reached from a source node $n1$. The idea is to consider a set containing only nodes of which the least cost path to $n1$ is already known. This set is enlarged successively, considering nodes which can be accessed directly by an edge from one of the nodes already contained in the set. From these candidates, the one with the best cost path to the source node is added to the set.

To keep track of the state of the computation, the type *comp_state* is defined, as well as a function for creating an initial state:
```
# type comp_state = { paths : int array;
                      already_treated : bool array;
                      distances : cost array;
                      source : int;
                      nn : int};;
# let create_state () =  { paths = [||]; already_treated = [||]; distances = [||];
                           nn = 0; source = 0};;
```
The field `source` contains the start node. The field `already_treated` indicates the nodes whose optimal path from the source is already known. The field `nn` indicates the total number of the graph's nodes. The vector `distances` holds the minimal distances between the source and the other nodes. For each node, the vector `path` contains the preceding node on the least cost path. The path to the source can be reconstructed from each node by using `path`.

Cost Functions

Four functions on costs are defined: `a_cost` to test for the existence of an edge, `float_of_cost` to return the floating point value, `add_cost` to add two costs and `less_cost` to check if one cost is smaller than another.

```
# let a_cost c = match c with Nan → false | _-> true;;
```

```
val a_cost : cost -> bool = <fun>
# let float_of_cost c = match c with
    Nan → failwith "float_of_cost"
  | Cost x → x;;
val float_of_cost : cost -> float = <fun>
# let add_cost  c1 c2 = match (c1,c2) with
    Cost x, Cost y → Cost (x+.y)
  | Nan, Cost y → c2
  | Cost x, Nan → c1
  | Nan, Nan →  c1;;
val add_cost : cost -> cost -> cost = <fun>
# let less_cost  c1 c2 = match (c1,c2) with
    Cost x, Cost y → x < y
  | Cost x, Nan → true
  | _, _ →  false;;
val less_cost : cost -> cost -> bool = <fun>
```

The value `Nan` plays a special role in the computations and in the comparison. We will come back to this when we have presented the main function (page 388).

Implementing the Algorithm

The search for the next node with known least cost path is divided into two functions. The first, `first_not_treated`, selects the first node not already contained in the set of nodes with known least cost paths. This node serves as the initial value for the second function, `least_not_treated`, which returns a node not already in the set with a best cost path to the source. This path will be added to the set.

```
# exception Found of int;;
exception Found of int
# let first_not_treated cs =
    try
      for i=0 to cs.nn-1 do
        if not cs.already_treated.(i) then raise (Found i)
      done;
      raise Not_found;
      0
    with Found i → i ;;
val first_not_treated : comp_state -> int = <fun>
# let least_not_treated p cs =
    let ni = ref p
    and nd = ref cs.distances.(p) in
      for i=p+1 to cs.nn-1 do
        if not cs.already_treated.(i) then
         if less_cost cs.distances.(i)  !nd then
           ( nd := cs.distances.(i);
             ni := i )
      done;
      !ni,!nd;;
```

```
val least_not_treated : int -> comp_state -> int * cost = <fun>
```

The function one_round selects a new node, adds it to the set of treated nodes and computes the distances for any next candidates.

```
# exception No_way;;
exception No_way
# let one_round cs  g =
    let p = first_not_treated cs  in
    let np,nc = least_not_treated p cs in
      if not(a_cost nc  ) then raise No_way
      else
      begin
        cs.already_treated.(np) <- true;
        for i = 0 to cs.nn -1 do
          if not cs.already_treated.(i) then
          if a_cost g.m.(np).(i) then
            let ic = add_cost cs.distances.(np)  g.m.(np).(i) in
              if less_cost ic cs.distances.(i)   then (
                cs.paths.(i) <- np;
                cs.distances.(i) <- ic
               )
        done;
        cs
      end;;
val one_round : comp_state -> 'a graph -> comp_state = <fun>
```

The only thing left in the implementation of Dijkstra's algorithm is to iterate the preceding function. The function dij takes a node and a graph as arguments and returns a value of type comp_state, with the information from which the least cost paths from the source to all the reachable nodes of the graph can be deduced.

```
# let dij s g =
    if belongs_to s g then
    begin
      let i = index s g in
      let cs = { paths = Array.create g.ind (-1) ;
                 already_treated = Array.create g.ind false;
                 distances = Array.create g.ind Nan;
                 nn = g.ind;
                 source = i}  in
        cs.already_treated.(i) <- true;
        for j=0 to g.ind-1 do
          let c = g.m.(i).(j) in
            cs.distances.(j) <- c;
            if a_cost c then cs.paths.(j) <- i
        done;
        try
```

```
          for k = 0 to cs.nn-2 do
            ignore(one_round cs g)
          done;
          cs
        with No_way → cs
    end
    else failwith "dij: node unknown";;
val dij : 'a -> 'a graph -> comp_state = <fun>
```

`Nan` is the initial value of the distances. It represents an infinite distance, which conforms to the comparison function `less_cost`. In contrast, for the addition of costs (function `add_cost`), this value is treated as a zero. This allows a simple implementation of the table of distances.

Now the search with Dijkstra's algorithm can be tested.

```
# let g = test_aho ();;
# let r = dij "A" g;;
```

The return values are:

```
# r.paths;;
- : int array = [|0; 0; 3; 0; 2|]
# r.distances;;
- : cost array = [|Cost 0; Cost 10; Cost 50; Cost 30; Cost 60|]
```

Displaying the Results

To make the results more readable, we now define a display function.

The table `paths` of the state returned by `dij` only contains the last edges of the computed paths. In order to get the entire paths, it is necessary to recursively go back to the source.

```
# let display_state f (g,st)  dest =
    if belongs_to dest g then
       let d = index dest g in
         let rec aux is =
            if is = st.source then Printf.printf "%a"  f g.nodes.(is)
            else (
              let old = st.paths.(is) in
              aux old;
              Printf.printf " -> (%4.1f) %a" (float_of_cost g.m.(old).(is))
                                             f g.nodes.(is)
            )
         in
           if not(a_cost st.distances.(d)) then Printf.printf "no way\n"
           else (
```

```
            aux d;
            Printf.printf " = %4.1f\n" (float_of_cost st.distances.(d));;
val display_state :
  (out_channel -> 'a -> unit) -> 'a graph * comp_state -> 'a -> unit = <fun>
```

This recursive function uses the command stack to display the nodes in the right order. Note that the use of the format `"a"` requires the function parameter `f` to preserve the polymorphism of the graphs for the display.

The optimal path between the nodes "A" (index 0) and "E" (index 4) is displayed in the following way:

```
# display_state (fun x y → Printf.printf "%s!" y) (a,r) "E";;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

The different nodes of the path and the costs of each route are shown.

Introducing a Cache

Dijkstra's algorithm computes all least cost paths starting from a source. The idea of preserving these least cost paths for the next inquiry with the same source suggests itself. However, this storage could occupy a considerable amount of memory. This suggests the use of "weak pointers." If the results of a computation starting from a source are stored in a table of weak pointers, it will be possible for the next computation to check if the computation has already been done. Because the pointers are weak, the memory occupied by the states can be freed by the garbage collector if needed. This avoids interrupting the rest of the program through the allocation of too much memory. In the worst case, the computation has to be repeated for a future inquiry.

Implementing a Cache

A new type *'a comp_graph* is defined:

```
# type 'a comp_graph =
    { g : 'a graph; w : comp_state Weak.t } ;;
```

The fields `g` and `w` correspond to the graph and to the table of weak pointers, pointing to the computation states for each possible source.

Such values are constructed by the function `create_comp_graph`.

```
# let create_comp_graph g =
    { g = g;
      w = Weak.create g.ind } ;;
val create_comp_graph : 'a graph -> 'a comp_graph = <fun>
```

The function `dij_quick` checks to see if the computation has already been done. If it has, the stored result is returned. Otherwise, the computation is executed and the result is registered in the table of weak pointers.

```
# let dij_quick s cg =
```

```
    let i = index s cg.g in
      match Weak.get cg.w i with
        None → let cs = dij s cg.g in
                  Weak.set cg.w i (Some cs);
                  cs
      | Some cs → cs;;
val dij_quick : 'a -> 'a comp_graph -> comp_state = <fun>
```

The display function still can be used:

```
# let cg_a = create_comp_graph a in
  let r = dij_quick "A" cg_a in
   display_state (fun x y → Printf.printf "%s!" y) (a,r) "E" ;;
A! -> (30.0) D! -> (20.0) C! -> (10.0) E! = 60.0
- : unit = ()
```

Performance Evaluation

We will test the performance of the functions `dij` and `dij_quick` by iterating each one on a random list of sources. In this way an application which frequently computes least cost paths is simulated (for example a railway route planning system).

We define the following function to time the calculations:

```
# let exe_time f g ss =
    let t0 = Sys.time() in
     Printf.printf "Start (%5.2f)\n" t0;
     List.iter (fun s → ignore(f s g)) ss;
     let t1 = Sys.time() in
      Printf.printf "End (%5.2f)\n" t1;
      Printf.printf "Duration = (%5.2f)\n" (t1 -. t0) ;;
val exe_time : ('a -> 'b -> 'c) -> 'b -> 'a list -> unit = <fun>
```

We create a random list of 20000 nodes and measure the performance on the graph `a`:

```
# let ss =
   let ss0 = ref [] in
   let i0 = int_of_char 'A' in
   let new_s i = Char.escaped (char_of_int (i0+i)) in
   for i=0 to 20000 do ss0 := (new_s (Random.int a.size))::!ss0 done;
   !ss0 ;;
val ss : string list =
  ["A"; "B"; "D"; "A"; "E"; "C"; "B"; "B"; "D"; "E"; "B"; "E"; "C"; "E"; "E";
   "D"; "D"; "A"; "E"; ...]
# Printf.printf"Function dij :\n";
   exe_time dij a ss ;;
Function dij :
Start ( 1.09)
```

```
End ( 1.41)
Duration = ( 0.32)
- : unit = ()
# Printf.printf"Function dij_quick :\n";
   exe_time dij_quick (create_comp_graph a) ss ;;
Function dij_quick :
Start ( 1.41)
End ( 1.44)
Duration = ( 0.03)
- : unit = ()
```

The results confirm our assumption. The direct access to a result held in the cache is considerably faster than a second computation of the result.

A Graphical Interface

We use the `Awi` library to construct a graphical interface to display graphs. The interface allows selection of the source and destination nodes of the path. When the path is found, it is displayed graphically. We define the type *'a gg*, containing fields describing the graph and the computation, as well as fields of the graphical interface.

```
#  #load "PROGRAMMES/awi.cmo";;

# type 'a gg = { mutable src : 'a * Awi.component;
                 mutable dest : 'a * Awi.component;
                 pos : (int * int) array;
                 cg : 'a comp_graph;
                 mutable state : comp_state;
                 mutable main : Awi.component;
                 to_string : 'a → string;
                 from_string : string → 'a } ;;
```

The fields `src` and `dest` are tuples (node, component), associating a node and a component. The field `pos` contains the position of each component. The field `main` is the main container of the set of components. The two functions `to_string` and `from_string` are conversion functions between type *'a* and strings. The elements necessary to construct these values are the graph information, the position table and the conversion functions.

```
# let create_gg cg vpos ts fs =
    {src = cg.g.nodes.(0),Awi.empty_component;
     dest = cg.g.nodes.(0),Awi.empty_component;
     pos = vpos;
     cg = cg;
     state = create_state () ;
```

```
      main = Awi.empty_component;
      to_string = ts;
      from_string = fs};;
val create_gg :
  'a comp_graph ->
  (int * int) array -> ('a -> string) -> (string -> 'a) -> 'a gg = <fun>
```

Visualisation

In order to display the graph, the nodes have to be drawn, and the edges have to be traced. The nodes are represented by button components of the `Awi` library. The edges are traced directly in the main window. The function `display_edge` displays the edges. The function `display_shortest_path` displays the found path in a different color.

Drawing Edges An edge connects two nodes and has an associated weight. The connection between two nodes can be represented by a line. The main difficulty is indicating the orientation of the line. We choose to represent it by an arrow. The arrow is rotated by the angle the line has with the abscissa (the x-axis) to give it the proper orientation. Finally, the costs are displayed beside the edge.

To draw the arrow of an edge we define the functions `rotate` and `translate` which care respectively for rotation and shifting. The function `display_arrow` draws the arrow.

```
# let rotate l a =
    let ca = cos a  and sa = sin a in
      List.map (function (x,y) → ( x*.ca +. -.y*.sa, x*.sa +. y*.ca)) l;;
val rotate : (float * float) list -> float -> (float * float) list = <fun>
# let translate l (tx,ty) =
    List.map (function (x,y) → (x +. tx, y +. ty)) l;;
val translate :
  (float * float) list -> float * float -> (float * float) list = <fun>
# let display_arrow (mx,my) a =
    let triangle = [(5.,0.); (-3.,3.); (1.,0.); (-3.,-3.); (5.,0.)] in
    let tr =  rotate triangle a in
    let ttr = translate tr (mx,my) in
    let tt = List.map (function (x,y) → (int_of_float x, int_of_float y)) ttr
    in
      Graphics.fill_poly (Array.of_list tt);;
val display_arrow : float * float -> float -> unit = <fun>
```

The position of the text indicating the weight of an edge depends on the angle of the edge.

```
# let display_label (mx,my) a lab =
    let (sx,sy) = Graphics.text_size lab in
    let pos = [ float(-sx/2),float(-sy) ] in
    let pr = rotate pos a in
```

```
    let pt = translate pr (mx,my) in
      let px,py = List.hd pt in
        let ox,oy = Graphics.current_point () in
          Graphics.moveto ((int_of_float mx)-sx-6)
                          ((int_of_float my) );
          Graphics.draw_string lab;
          Graphics.moveto ox oy;;
val display_label : float * float -> float -> string -> unit = <fun>
```

The preceding functions are now used by the function `display_edge`. Parameters are the graphical interface `gg`, the nodes `i` and `j`, and the color (`col`) to use.

```
# let display_edge gg col i j =
    let g = gg.cg.g in
    let x,y = gg.main.Awi.x,gg.main.Awi.y in
      if a_cost g.m.(i).(j) then (
        let (a1,b1) = gg.pos.(i)
        and (a2,b2) = gg.pos.(j) in
          let x0,y0 = x+a1,y+b1   and x1,y1 = x+a2,y+b2 in
          let rxm =  (float(x1-x0)) /. 2.  and rym =  (float(y1-y0)) /. 2. in
          let xm = (float x0) +. rxm and ym = (float y0) +. rym in
            Graphics.set_color col;
            Graphics.moveto x0 y0;
            Graphics.lineto x1 y1;
            let a =  atan2 rym rxm in
            display_arrow (xm,ym) a;
            display_label (xm,ym) a
              (string_of_float(float_of_cost g.m.(i).(j))));;
val display_edge : 'a gg -> Graphics.color -> int -> int -> unit = <fun>
```

Displaying a Path To display a path, all edges along the path are displayed. The graphical display of a path towards a destination uses the same technique as the textual display.

```
# let rec display_shortest_path gg col dest =
    let g = gg.cg.g in
      if belongs_to dest g then
        let d = index dest g in
         let rec aux is =
            if is = gg.state.source then ()
            else (
              let old = gg.state.paths.(is) in
                display_edge gg col old is;
                aux old )
         in
           if not(a_cost gg.state.distances.(d)) then Printf.printf "no way\n"
           else aux d;;
```

```
val display_shortest_path : 'a gg -> Graphics.color -> 'a -> unit = <fun>
```

Displaying a Graph The function `display_gg` displays a complete graph. If the destination node is not empty, the path between the source and the destination is traced.

```
#  let display_gg gg () =
     Awi.display_rect gg.main ();
     for i=0 to gg.cg.g.ind -1 do
       for j=0 to gg.cg.g.ind -1 do
         if i<> j then display_edge gg (Graphics.black) i j
       done
     done;
     if snd gg.dest != Awi.empty_component then
        display_shortest_path gg Graphics.red (fst gg.dest);;
val display_gg : 'a gg -> unit -> unit = <fun>
```

The Node Component

The nodes still need to be drawn. Since the user is allowed to choose the source and destination nodes, we define a component for nodes.

The user's main action is choosing the end nodes of the path to be found. Thus a node must be a component that reacts to mouse clicks, using its state to indicate if it has been chosen as a source or destination. We choose the button component, which reacts to mouse clicks.

Node Actions It is necessary to indicate node selection. To show this, the background color of a node is changed by the function `inverse`.

```
# let inverse b =
     let gc = Awi.get_gc b in
       let fcol = Awi.get_gc_fcol gc
       and bcol = Awi.get_gc_bcol gc in
         Awi.set_gc_bcol gc fcol;
         Awi.set_gc_fcol gc bcol;;
val inverse : Awi.component -> unit = <fun>
```

The function `action_click` effects this selection. It is called when a node is clicked on by the mouse. As parameters it takes the node associated with the button and the graph to modify the source or the destination of the search. When both nodes are selected, the function `dij_quick` finds a least cost path.

```
# let action_click node gg b bs =
     let (s1,s) = gg.src
     and (s2,d) = gg.dest in
```

```
      if  s == Awi.empty_component then (
        gg.src  <- (node,b); inverse b )
      else
       if d == Awi.empty_component then (
         inverse b;
         gg.dest <- (node,b);
         gg.state <- dij_quick s1 gg.cg;
        display_shortest_path gg (Graphics.red) node
       )
       else (inverse s; inverse d;
             gg.dest <- (s2,Awi.empty_component);
             gg.src <- node,b; inverse b);;
val action_click : 'a -> 'a gg -> Awi.component -> 'b -> unit = <fun>
```

Creating an Interface The main function to create an interface takes an interface graph and a list of options, creates the different components and associates them with the graph. The parameters are the graph (`gg`), its dimensions (`gw` and `gh`), a list of graph and node options (`lopt`) and a list of node border options (`lopt2`).

```
# let main_gg gg gw gh lopt lopt2 =
    let gc = Awi.make_default_context () in
      Awi.set_gc gc lopt;
  (* compute the maximal button size *)
      let vs = Array.map gg.to_string gg.cg.g.nodes in
      let vsize = Array.map Graphics.text_size  vs in
        let w = Array.fold_right (fun (x,y) → max x)  vsize 0
        and h = Array.fold_right (fun (x,y) → max y)  vsize 0 in
  (* create the main panel *)
        gg.main <- Awi.create_panel true gw gh lopt;
        gg.main.Awi.display <- display_gg gg;
  (* create the buttons *)
          let vb_bs =
            Array.map (fun x → x,Awi.create_button (" "^(gg.to_string x)^" ")
                        lopt)
                      gg.cg.g.nodes in
            let f_act_b = Array.map (fun (x,(b,bs)) →
                                       let ac = action_click x  gg b
                                       in Awi.set_bs_action bs ac)  vb_bs in
            let bb =
              Array.map (function (_,(b,_)) → Awi.create_border b lopt2) vb_bs
            in
              Array.iteri
                (fun i (b) → let x,y = gg.pos.(i) in
                             Awi.add_component gg.main  b
                               ["PosX",Awi.Iopt (x-w/2);
                                "PosY", Awi.Iopt (y-h/2)]) bb;
```

```
            ();;
val main_gg :
  'a gg ->
  int ->
  int -> (string * Awi.opt_val) list -> (string * Awi.opt_val) list -> unit =
  <fun>
```

The buttons are created automatically. They are positioned on the main window.

Testing the Interface Everything is ready to create an interface now. We use a graph whose nodes are character strings to simplify the conversion functions. We construct the graph gg as follows:

```
# let id x = x;;
# let pos =  [| 200, 300; 80, 200 ; 100, 100; 200, 100;  260, 200 |];;
# let gg = create_gg  (create_comp_graph (test_aho())) pos id id;;
# main_gg  gg 400 400 ["Background", Awi.Copt  (Graphics.rgb 130 130 130);
               "Foreground",Awi.Copt  Graphics.green]
   [ "Relief", Awi.Sopt "Top";"Border_size", Awi.Iopt 2];;
```

Calling `Awi.loop true false gg.main;;` starts the interaction loop of the `Awi` library.

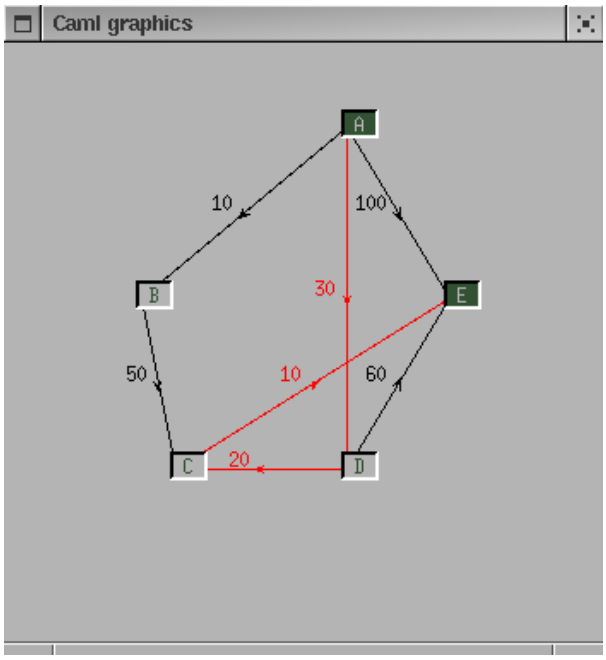


Figure 13.9: Selecting the nodes for a search

Figure 13.9 shows the computed path between the nodes "A" and "E". The edges on the path have changed their color.

Creating a Standalone Application

We will now show the steps needed to construct a standalone application. The application takes the name of a file describing the graph as an argument. For standalone applications, it is not necesary to have an Objective Caml distribution on the execution machine.

A Graph Description File

The file containes information about the graph as well as information used for the graphical interface. For the latter information, we define a second format. From this graphical description, we construct a value of the type *g_info*.

```
# type g_info = {npos : (int * int) array;
                 mutable opt : Awi.lopt;
                 mutable g_w : int;
                 mutable g_h : int};;
```

The format for the graphical information is described by the four key words of list key2.

```
# let key2 = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"];;
val key2 : string list = ["HEIGHT"; "LENGTH"; "POSITION"; "COLOR"]
# let lex2 l = Genlex.make_lexer key2 (Stream.of_string l);;
val lex2 : string -> Genlex.token Stream.t = <fun>

# let pars2 g gi s = match s with parser
      [< '(Genlex.Kwd "HEIGHT"); '(Genlex.Int i) >] → gi.g_h <- i
  |   [< '(Genlex.Kwd "LENGTH"); '(Genlex.Int i) >] → gi.g_w <- i
  |   [< '(Genlex.Kwd "POSITION"); '(Genlex.Ident s);
         '(Genlex.Int i); '(Genlex.Int j) >] → gi.npos.(index s g) <- (i,j)
  |   [< '(Genlex.Kwd "COLOR"); '(Genlex.Ident s);
         '(Genlex.Int r); '(Genlex.Int g); '(Genlex.Int b) >] →
              gi.opt <- (s, Awi.Copt (Graphics.rgb r g b)) :: gi.opt
  | [<>] → ();;
val pars2 : string graph -> g_info -> Genlex.token Stream.t -> unit = <fun>
```

Creating the Application

The function create_graph takes the name of a file as input and returns a couple composed of a graph and associated graphical information.

```
# let create_gg_graph name =
    let g = create_graph name in
```

```
      let gi = {npos = Array.create g.size (0,0); opt=[]; g_w =0; g_h = 0;} in
    let ic = open_in name in
    try
      print_string ("Loading (pass 2) " ^name ^" : ");
      while true do
        print_string ".";
        let l = input_line ic in pars2 g gi (lex2 l)
      done ;
      g,gi
    with End_of_file → print_newline(); close_in ic; g,gi;;
val create_gg_graph : string -> string graph * g_info = <fun>
```

The function `create_app` constructs the interface of a graph.

```
# let create_app name =
    let g,gi = create_gg_graph name in
    let size = (string_of_int gi.g_w) ^ "x" ^ (string_of_int gi.g_h) in
    Graphics.open_graph (" "^size);
    let gg = create_gg (create_comp_graph g) gi.npos id id in
    main_gg gg gi.g_w gi.g_h
      [ "Background", Awi.Copt (Graphics.rgb 130 130 130) ;
        "Foreground", Awi.Copt Graphics.green ]
      [ "Relief", Awi.Sopt "Top" ; "Border_size", Awi.Iopt 2 ] ;
    gg;;
val create_app : string -> string gg = <fun>
```

Finally, the function `main` takes the name of the file from the command line, constructs a graph with an interface and starts the interaction loop on the main component of the graph interface.

```
# let main () =
    if (Array.length Sys.argv ) <> 2
    then Printf.printf "Usage: dij.exe filename\n"
    else
      let gg = create_app Sys.argv.(1) in
        Awi.loop true false gg.main;;
val main : unit -> unit = <fun>
```

The last expression of that program starts the function `main`.

The Executable

The motivation for making a standalone application is to support its distribution. We collect the types and functions described in this section in the file `dij.ml`. Then we compile the file, adding the different libraries which are used. Here is the command to compile it under Linux.

```
ocamlc -custom -o dij.exe graphics.cma awi.cmo graphs.ml \
    -cclib -lgraphics -cclib -L/usr/X11/lib -cclib -lX11
```

Compiling standalone applications using the `Graphics` library is described in chapters 5 and 7.

Final Notes

The skeleton of this application is sufficiently general to be used in contexts other than the search for traveling paths. Different types of problems can be represented by a weighted graph. For example the search for a path in a labyrinth can be coded in a graph where each intersection is a node. Finding a solution corresponds to computing the shortest path between the start and the goal.

To compare the performance betwen C and Objective Caml, we wrote Dijkstra's algorithm in C. The C program uses the Objective Caml data structures to perform the calculations.

To improve the graphical interface, we add a *`textfield`* for the name of the file and two buttons to load and to store a graph. The user may then modify the positions of the nodes by mouse to improve the appearance.

A second improvement of the graphical interface is the ability to choose the form of the nodes. To display a button, a function tracing a rectangle is called. The display functions can be specialized to use polygons for nodes.

Part III

Application Structure

The third part of this work is dedicated to application development and describes two ways of organizing applications: modules and objects. The goal is to easily structure an application for incremental and rapid development, maintenance facilitated by the ability to change gracefully, and the possibility of reusing large parts for future development.

We have already presented the language's predefined modules (see chapter 8) viewed as compilation units. Objective Caml's module language supports on the one hand the definition of new simple modules in order to build one's own libraries, perhaps including abstract types, and on the other hand the definition of modules parameterized by other modules, called *functors*. The advantage of this parameterization lies in being able to "apply" a module to different argument modules in order to create specialized modules. Communication between modules is thus explicit, via the parameter module signature, which contains the types of its global declarations. However, nothing stops you from applying a functor to a module with a more extended signature, as long as it remains compatible with the specified parameter signature.

Besides, the Objective Caml language has an object-oriented extension. First of all object-oriented programming permits structured communication between objects. Rather than applying a function to some arguments, one sends a message (a request) to an object which knows how to deal with it. The object, an instance of a class (a structure gathering together data and methods), then executes the corresponding code. The main relation between classes is inheritance, which lets one describe subclasses which retain all the declarations of the ancestor class. Late binding between the name of a message and the corresponding code within the object takes place during program execution. Nevertheless Objective Caml typing guarantees that the receiving object will always have a method of this name, otherwise type inference would have raised a compile-time error. The second important relation is subtyping, where an object of a certain class can always be used in place of an object of another class. In this way a new type of polymorphism is introduced: inclusion polymorphism.

Finally the construction of a graphical interface, begun in chapter 5, uses different event management models. One puts together in an interface several components with respect to which the user or the system can produce events. The association of a component with a handler for one or more events taking place on it allows one to easily add to and modify such interfaces. The component-event-handler association can be cloaked in several forms: definition of a function (called a callback), inheritance with redefinition of handler methods, or finally registration of a handling object (delegation model).

Chapter 14 is a presentation of modular programming. The different prevailing terminologies of abstract data types and module languages are explained and illustrated by simple modules. Then the module language is detailed. The correspondence between modules (simple or not) and compilation units is made clear.

Chapter 15 contains an introduction to object-oriented programming. It brings a new way of structuring Objective Caml programs, an alternative to modules. This chapters shows how the notions of object-oriented programming (simple and multiple inheritance, abstract classes, parameterized classes, late binding) are articulated with

respect to the language's type system, and extend it by the subtyping relation to inclusion polymorphism.

Chapter 16 compares the two preceding software models and explains what factors to consider in deciding between the two, while also demonstrating how to simulate one by the other. It treats various cases of mixed models. Mixing leads to the enrichment of each of these two models, in particular with parameterized classes using the abstract type of a module.

Chapter 17 presents two classes of applications: two-player games, and the construction of a world of virtual robots. The first example is organized via various parameterized modules. In particular, a parameterized module is used to represent games for application of the minimax $\alpha\beta$ algorithm. It is then applied to two specific games: Connect 4 and Stone Henge. The second example uses an object model of a world and of abstract robots, from which, by inheritence, various simulations are derived. This example is presented in chapter 21.

14

Programming with Modules

Modular design and modular programming support the decomposition of a program into several *software units*, also called *modules*, which can be developed largely independently. A module can be compiled separately from the other modules comprising the program. Consequently, the developer of a program that uses a module does not need access to the source code of the module: the compiled code of the module is enough for building an executable program. However, the programmer must know the *interface* of the modules used, that is, which values, functions, types, exceptions, or even sub-modules are provided by the module, under which names, and with which types.

Explicitly writing down the interface of a module hides the details of its implementation from the programs that use this module. All these programs know about the module are the names and types of exported definitions; their exact implementations are not known. Thus, the maintainer of the module has considerable flexibility in evolving the module implementation: as long as the interface is unchanged and the semantics are preserved, users of the module will not notice the change in implementation. This can greatly facilitate the maintenance and evolution of large programs. Like local declarations, a module interface also supports hiding parts of the implementation that the module designer does not wish to publicize. An important application of this hiding mechanism is the implementation of *abstract data types*.

Finally, advanced module systems such as that of Objective Caml support the definition of *parameterized modules*, also called *generics*. These are modules that take other modules as parameters, thus increasing opportunities for code reuse.

Chapter Outline

Section 1 illustrates Objective Caml modules on the example of the `Stack` module from the standard library, and develops an alternate implementation of this module

with the same interface. Section 2 introduces the *module language* of Objective Caml in the case of simple modules, and shows some of its uses. In particular, we discuss type sharing between modules. Section 3 covers parameterized modules, which are called *functors* in Objective Caml. Finally, section 4 develops an extended example of modular programming: managing bank accounts with multiple views (the bank, the customer) and several parameters.

Modules as Compilation Units

The Objective Caml distribution includes a number of predefined modules. We saw in chapter 8 how to use these modules in a program. Here, we will show how users can define similar modules.

Interface and Implementation

The module `Stack` from the distribution provides the main functions on stacks, that is, queues with "last in, first out" discipline.

```
# let queue = Stack.create () ;;
val queue : '_a Stack.t = <abstr>
# Stack.push 1 queue ; Stack.push 2 queue ; Stack.push 3 queue ;;
- : unit = ()
# Stack.iter (fun n → Printf.printf "%d " n)  queue ;;
3 2 1 - : unit = ()
```

Since Objective Caml is distributed with full source code, we can look at the actual implementation of stacks.

ocaml-2.04/stdlib/stack.ml

```
type 'a t = { mutable c : 'a list }
exception Empty
let create () = { c = [] }
let clear s = s.c <- []
let push x s = s.c <- x :: s.c
let pop s = match s.c with  hd::tl → s.c <- tl; hd  |  []  → raise Empty
let length s = List.length s.c
let iter f s = List.iter f s.c
```

We see that the type of stacks (written `Stack.t` outside the `Stack` module and just `t` inside) is a record with one mutable field containing a list. The list holds the contents of the stack, with the list head corresponding to the stack top. Stack operations are implemented as the basic list operations applied to the field of the record.

Armed with this insider's knowledge, we could try to access directly the list representing a stack. However, Objective Caml will not let us do this.

```
# let list = queue.c ;;
Characters 12-19:
Unbound label c
```

The compiler complains as if it did not know that `Stack.t` is a record type with a field `c`. It is actually the case, as we can see by looking at the interface of the `Stack` module.

ocaml-2.04/stdlib/stack.mli

```
(* Module [Stack]: last-in first-out stacks *)
(* This module implements stacks (LIFOs), with in-place modification. *)

type 'a t          (* The type of stacks containing elements of type ['a]. *)

exception Empty    (* Raised when [pop] is applied to an empty stack. *)

val create: unit → 'a t
        (* Return a new stack, initially empty. *)
val push: 'a → 'a t → unit
        (* [push x s] adds the element [x] at the top of stack [s]. *)
val pop: 'a t → 'a
        (* [pop s] removes and returns the topmost element in stack [s],
           or raises [Empty] if the stack is empty. *)
val clear : 'a t → unit
        (* Discard all elements from a stack. *)
val length: 'a t → int
        (* Return the number of elements in a stack. *)
val iter: ('a → unit) → 'a t → unit
        (* [iter f s] applies [f] in turn to all elements of [s],
           from the element at the top of the stack to the element at the
           bottom of the stack. The stack itself is unchanged. *)
```

In addition to comments documenting the functions of the module, this file lists explicitly the value, type and exception identifiers defined in the file `stack.ml` that should be visible to clients of the `Stack` module. More precisely, the interface declares the names and type specifications for these exported definitions. In particular, the type name `t` is exported, but the representation of this type (that is, as a record with one `c` field) is not given in this interface. Thus, clients of the `Stack` module do not know how the type `Stack.t` is represented, and cannot access directly values of this type. We say that the type `Stack.t` is *abstract*, or *opaque*.

The interface also declares the functions operating on stacks, giving their names and types. (The types must be provided explicitly so that the type checker can check that

these functions are correctly used.) Declaration of values and functions in an interface is achieved via the following construct:

Syntax : **val** *nom* **:** *type*

Relating Interfaces and Implementations

As shown above, the `Stack` is composed of two parts: an implementation providing definitions, and an interface providing declarations for those definitions that are exported. All module components declared in the interface must have a matching definition in the implementation. Also, the types of values and functions as defined in the implementation must match the types declared in the interface.

The relationship between interface and implementation is not symmetrical. The implementation can contain more definitions than requested by the interface. Typically, the definition of an exported function can use auxiliary functions whose names will not appear in the interface. Such auxiliary functions cannot be called directly by a client of the module. Similarly, the interface can restrict the type of a definition. Consider a module defining the function `id` as the identity function (**let** *id x* = *x*). Its interface can declare `id` with the type *int -> int* (instead of the more general *'a -> 'a*). Then, clients of this module can only apply `id` to integers.

Since the interface of a module is clearly separated from its implementation, it becomes possible to have several implementations for the same interface, for instance to test different algorithms or data structures for the same operations. As an example, here is an alternate implementation for the `Stack` module, based on arrays instead of lists.

```
type 'a t = { mutable sp : int; mutable c : 'a array }
exception Empty
let create () = { sp=0 ; c = [||] }
let clear s = s.sp <- 0; s.c <- [||]
let size = 5
let increase s = s.c <- Array.append s.c (Array.create size s.c.(0))

let push x s =
  if s.sp >= Array.length s.c then increase s ;
  s.c.(s.sp) <- x ;
  s.sp <- succ s.sp

let pop s =
  if s.sp = 0 then raise Empty
  else let x = s.c.(s.sp) in s.sp <- pred s.sp ; x

let length s = s.sp
let iter f s = for i = pred s.sp downto 0 do f s.sc.(i) done
```

This new implementation satisfies the requisites of the interface file `stack.mli`. Thus, it can be used instead of the predefined implementation of `Stack` in any program.

Separate Compilation

Like most modern programming languages, Objective Caml supports the decomposition of programs into multiple compilation units, separately compiled. A compilation unit is composed of two files, an implementation file (with extension `.ml`) and an interface file (with extension `.mli`). Each compilation unit is viewed as a module. Compiling the implementation file `name.ml` defines the module named `Name`[1].

Values, types and exceptions defined in a module can be referenced either via the *dot notation* (`Module.identifier`), also known as *qualified identifiers*, or via the **open** construct.

a.ml	b.ml
type *t* = { *x*:*int* ; *y*:*int* } ;; **let** *f c* = *c.x* + *c.y* ;;	**let val** = { *A.x* = 1 ; *A.y* = 2 } ;; *A.f* **val** ;; **open** *A* ;; *f* **val** ;;

An interface file (`.mli` file) must be compiled using the `ocamlc -c` command before any module that depends on this interface is compiled; this includes both clients of the module and the implementation file for this module as well.

If no interface file is provided for an implementation file, Objective Caml considers that the module exports everything; that is, all identifiers defined in the implementation file are present in the implicit interface with their most general types.

The linking phase to produce an executable file is performed as described in chapter 7: the `ocamlc` command (without the `-c` option), followed by the object files for all compilation units comprising the program. Warning: object files must be provided on the command line in dependency order. That is, if a module `B` references another module `A`, the object file `a.cmo` must precede `b.cmo` on the linker command line. Consequently, cross dependencies between two modules are forbidden.

For instance, to generate an executable file from the source files `a.ml` and `b.ml`, with matching interface files `a.mli` and `b.mli`, we issue the following commands:

```
> ocamlc -c a.mli
> ocamlc -c a.ml
> ocamlc -c b.mli
> ocamlc -c b.ml
> ocamlc a.cmo b.cmo
```

Compilation units, composed of one interface file and one implementation file, support separate compilation and information hiding. However, their abilities as a general program structuring tool are low. In particular, there is a one-to-one connection

1. Both files `name.ml` and `Name.ml` result in the same module name.

between modules and files, preventing a program to use simultaneously several implementations of a given interface, or also several interfaces for the same implementation. Nested modules and module parameterization are not supported either. To palliate those weaknesses, Objective Caml offers a module language, with special syntax and linguistic constructs, to manipulate modules inside the language itself. The remainder of this chapter introduces this module language.

The Module Language

The Objective Caml language features a sub-language for modules, which comes in addition to the core language that we have seen so far. In this module language, the interface of a module is called a *signature* and its implementation is called a *structure*. When there is no ambiguity, we will often use the word "module" to refer to a structure.

The syntax for declaring signatures and structures is as follows:

Syntax :

```
module type NAME =
  sig
    interface declarations
  end
```

Syntax :

```
module Name =
  struct
    implementation definitions
  end
```

Warning The name of a module *must* start with an uppercase letter. There are no such case restrictions on names of signatures, but by convention we will use names in uppercase for signatures.

Signatures and structures do not need to be bound to names: we can also use anonymous signature and structure expressions, writing simply

Syntax : `sig` *declarations* `end`

Syntax : `struct` *definitions* `end`

We write *signature* and *structure* to refer to either names of signatures and structures, or anonymous signature and structure expressions.

Every structure possesses a default signature, computed by the type inference system, which reveals all the definitions contained in the structure, with their most general types. When defining a structure, we can also indicate the desired signature by adding

a signature constraint (similar to the type constraints from the core language), using one of the following two syntactic forms:

Syntax : **module** *Name* **:** *signature* **=** *structure*

Syntax : **module** *Name* **= (***structure* **:** *signature***)**

When an explicit signature is provided, the system checks that all the components declared in the signature are defined in the structure *structure*, and that the types are consistent. In other terms, the system checks that the explicit signature provided is "included in", or implied by, the default signature. If so, *Name* is viewed in the remainder of the code with the signature "*signature*", and only the components declared in the signature are accessible to the clients of the module. (This is the same behavior we saw previously with interface files.)

Access to the components of a module is via the dot notation:

Syntax : $Name_1$**.**$name_2$

We say that the name $name_2$ is *qualified* by the name $Name_1$ of its defining module.

The module name and the dot can be omitted using a directive to *open* the module:

Syntax : **open** *Name*

In the scope of this directive, we can use short names $name_2$ to refer to the components of the module *Name*. In case of name conflicts, opening a module hides previously defined entities with the same names, as in the case of identifier redefinitions.

Two Stack Modules

We continue the example of stacks by recasting it in the module language. The signature for a stack module is obtained by wrapping the declarations from the `stack.mli` file in a signature declaration:

```
# module type STACK =
    sig
      type 'a t
      exception Empty
      val create: unit → 'a t
      val push: 'a → 'a t → unit
      val pop: 'a t → 'a
      val clear : 'a t → unit
      val length: 'a t → int
      val iter: ('a → unit) → 'a t → unit
    end ;;
module type STACK =
  sig
    type 'a t
    exception Empty
    val create : unit -> 'a t
```

```
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

A first implementation of stacks is obtained by reusing the `Stack` module from the standard library:

```
# module StandardStack = Stack ;;
module StandardStack :
  sig
    type 'a t = 'a Stack.t
    exception Empty
    val create : unit -> 'a t
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val clear : 'a t -> unit
    val length : 'a t -> int
    val iter : ('a -> unit) -> 'a t -> unit
  end
```

We then define an alternate implementation based on arrays:

```
# module MyStack =
    struct
      type 'a t = { mutable sp : int; mutable c : 'a array }
      exception Empty
      let create () = { sp=0 ; c = [||] }
      let clear s = s.sp <- 0; s.c <- [||]
      let increase s x =  s.c <- Array.append s.c (Array.create 5 x)
      let push x s =
        if s.sp >= Array.length s.c then increase s x;
        s.c.(s.sp) <- x;
        s.sp <- succ s.sp
      let pop s =
        if s.sp =0 then raise Empty
        else (s.sp <- pred s.sp ; s.c.(s.sp))
      let length s = s.sp
      let iter f s = for i = pred s.sp downto 0 do f s.c.(i) done
    end ;;
module MyStack :
  sig
    type 'a t = { mutable sp: int; mutable c: 'a array }
    exception Empty
    val create : unit -> 'a t
    val clear : 'a t -> unit
    val increase : 'a t -> 'a -> unit
    val push : 'a -> 'a t -> unit
    val pop : 'a t -> 'a
    val length : 'a t -> int
    val iter : ('a -> 'b) -> 'a t -> unit
```

```
  end
```

These two modules implement the type `t` of stacks by different data types.

```
# StandardStack.create () ;;
- : '_a StandardStack.t = <abstr>
# MyStack.create () ;;
- : '_a MyStack.t = {MyStack.sp=0; MyStack.c=[||]}
```

To abstract over the type representation in `Mystack`, we add a signature constraint by the `STACK` signature.

```
# module MyStack = (MyStack : STACK) ;;
module MyStack : STACK
# MyStack.create() ;;
- : '_a MyStack.t = <abstr>
```

The two modules `StandardStack` and `MyStack` implement the same interface, that is, provide the same set of operations over stacks, but their `t` types are different. It is therefore impossible to apply operations from one module to values from the other module:

```
# let s = StandardStack.create() ;;
val s : '_a StandardStack.t = <abstr>
# MyStack.push 0 s ;;
Characters 15-16:
This expression has type 'a StandardStack.t = 'a Stack.t
but is here used with type int MyStack.t
```

Even if both modules implemented the *t* type by the same concrete type, constraining `MyStack` by the signature `STACK` suffices to abstract over the *t* type, rendering it incompatible with any other type in the system and preventing sharing of values and operations between the various stack modules.

```
# module S1 = ( MyStack : STACK ) ;;
module S1 : STACK
# module S2 = ( MyStack : STACK ) ;;
module S2 : STACK
# let s = S1.create () ;;
val s : '_a S1.t = <abstr>
# S2.push 0 s ;;
Characters 10-11:
This expression has type 'a S1.t but is here used with type int S2.t
```

The Objective Caml system compares abstract types by names. Here, the two types `S1.t` and `S2.t` are both abstract, and have different names, hence they are considered as incompatible. It is precisely this restriction that makes type abstraction effective, by preventing any access to the definition of the type being abstracted.

Modules and Information Hiding

This section shows additional examples of signature constraints hiding or abstracting definitions of structure components.

Hiding Type Implementations

Abstracting over a type ensures that the only way to construct values of this type is via the functions exported from its definition module. This can be used to restrict the values that can belong to this type. In the following example, we implement an abstract type of integers which, by construction, can never take the value 0.

```
# module Int_Star =
    ( struct
        type t = int
        exception Isnul
        let of_int = function 0 → raise Isnul | n → n
        let mult = ( * )
      end
    :
      sig
        type t
        exception Isnul
        val of_int : int → t
        val mult : t → t → t
      end
    ) ;;
module Int_Star :
  sig type t exception Isnul val of_int : int -> t val mult : t -> t -> t end
```

Hiding Values

We now define a symbol generator, similar to that of page 103, using a signature constraint to hide the state of the generator.

We first define the signature `GENSYM` exporting only two functions for generating symbols.

```
# module type GENSYM =
    sig
      val reset : unit → unit
      val next : string → string
    end ;;
```

We then implement this signature as follows:

```
# module Gensym : GENSYM =
    struct
      let c = ref 0
      let reset () = c:=0
      let next s = incr c ; s ^ (string_of_int !c)
```

```
    end;;
module Gensym : GENSYM
```

The reference `c` holding the state of the generator `Gensym` is not accessible outside the two exported functions.

```
# Gensym.reset();;
- : unit = ()
# Gensym.next "T";;
- : string = "T1"
# Gensym.next "X";;
- : string = "X2"
# Gensym.reset();;
- : unit = ()
# Gensym.next "U";;
- : string = "U1"
# Gensym.c;;
Characters 0-8:
Unbound value Gensym.c
```

The definition of `c` is essentially local to the structure `Gensym`, since it is hidden by the associated signature. The signature constraint achieves more simply the same goal as the local definition of a reference in the definition of the two functions `reset_s` and `new_s` on page 103.

Multiple Views of a Module

The module language and its signature constraints support taking several views of a given structure. For instance, we can have a "super-user interface" for the module `Gensym`, allowing the symbol counter to be reset, and a "normal user interface" that permits only the generation of new symbols, but no other intervention on the counter. To implement the latter interface, it suffices to declare the signature:

```
# module type USER_GENSYM =
    sig
      val next : string → string
    end;;
module type USER_GENSYM = sig val next : string -> string end
```

We then implement it by a mere signature constraint.

```
# module UserGensym = (Gensym : USER_GENSYM) ;;
module UserGensym : USER_GENSYM
# UserGensym.next "U" ;;
- : string = "U2"
# UserGensym.reset() ;;
Characters 0-16:
Unbound value UserGensym.reset
```

The `UserGensym` module fully reuses the code of the `Gensym` module. In addition, both modules share the same counter:

```
# Gensym.next "U" ;;
- : string = "U3"
# Gensym.reset() ;;
- : unit = ()
# UserGensym.next "V" ;;
- : string = "V1"
```

Type Sharing between Modules

As we saw on page 411, abstract types with different names are incompatible. This can be problematic when we wish to share an abstract type between several modules. There are two ways to achieve this sharing: one is via a special sharing construct in the module language; the other one uses the lexical scoping of modules.

Sharing via Constraints

The following example illustrates the sharing issue. We define a module `M` providing an abstract type *M.t*. We then restrict `M` on two different signatures exporting different subsets of operations.

```
# module M =
  (
    struct
      type t = int ref
      let create() = ref 0
      let add x = incr x
      let get x = if !x>0 then (decr x; 1) else failwith "Empty"
    end
    :
    sig
      type t
      val create : unit → t
      val add : t → unit
      val get : t → int
    end
  ) ;;

# module type S1 =
    sig
      type t
      val create : unit → t
      val add : t → unit
    end ;;

# module type S2 =
    sig
      type t
```

```
      val get : t → int
    end ;;
# module M1 = (M:S1) ;;
module M1 : S1
# module M2 = (M:S2) ;;
module M2 : S2
```

As written above, the types `M1.t` and `M2.t` are incompatible. However, we would like to say that both types are abstract but identical. To do this, Objective Caml offers special syntax to declare a type equality over an abstract type in a signature.

Syntax : *NAME* **with type** t_1 **=** t_2 **and** ...

This type constraint forces the type t_1 declared in the signature *NAME* to be equal to the type t_2.

Type constraints over all types exported by a sub-module can be declared in one operation with the syntax

Syntax : *NAME* **with module** $Name_1$ **=** $Name_2$

Using these type sharing constraints, we can declare that the two modules `M1` and `M2` define identical abstract types.

```
# module M1 = (M:S1 with type t = M.t) ;;
module M1 : sig type t = M.t val create : unit -> t val add : t -> unit end
# module M2 = (M:S2 with type t = M.t) ;;
module M2 : sig type t = M.t val get : t -> int end
# let x = M1.create() in M1.add x ;  M2.get x ;;
- : int = 1
```

Sharing and Nested Modules

Another possibility for ensuring type sharing is to use nested modules. We define two sub-modules (`M1` et `M2`) sharing an abstract type defined in the enclosing module `M`.

```
# module M =
    ( struct
        type t = int ref
        module M_hide =
          struct
            let create() = ref 0
            let add x = incr x
            let get x = if !x>0 then (decr x; 1) else failwith "Empty"
          end
        module M1 = M_hide
        module M2 = M_hide
      end
     :
      sig
        type t
```

```
        module M1 : sig  val create : unit → t  val add : t → unit end
        module M2 : sig  val get : t → int  end
      end ) ;;
module M :
  sig
    type t
    module M1 : sig val create : unit -> t val add : t -> unit end
    module M2 : sig val get : t -> int end
  end
```

As desired, values created by `M1` can be operated upon by `M2`, while hiding the representation of these values.

```
# let x = M.M1.create() ;;
val x : M.t = <abstr>
# M.M1.add x ; M.M2.get x ;;
- : int = 1
```

This solution is heavier than the previous solution based on type sharing constraints: the functions from `M1` and `M2` can only be accessed via the enclosing module `M`.

Extending Simple Modules

Modules are closed entities, defined once and for all. In particular, once an abstract type is defined using the module language, it is impossible to add further operations on the abstract type that depend on the type representation without modifying the module definition itself. (Operations derived from existing operations can of course be added later, outside the module.) As an extreme example, if the module exports no creation function, clients of the module will never be able to create values of the abstract type!

Therefore, adding new operations that depend on the type representation requires editing the sources of the module and adding the desired operations in its signature and structure. Of course, we then get a different module, and clients need to be recompiled. However, if the modifications performed on the module signature did not affect the components of the original signature, the remainder of the program remains correct and does not need to be modified, just recompiled.

Parameterized Modules

Parameterized modules are to modules what functions are to base values. Just like a function returns a new value from the values of its parameters, a parameterized module builds a new module from the modules given as parameters. Parameterized modules are also called *functors*.

The addition of functors to the module language increases the opportunities for code reuse in structures.

Functors are defined using a function-like syntax:

Syntax : **functor (** *Name* **:** *signature* **)** –> *structure*

```
# module Couple = functor ( Q : sig type t end ) →
    struct type couple = Q.t * Q.t end ;;
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

As for functions, syntactic sugar is provided for defining and naming a functor:

Syntax : **module** $Name_1$ **(** $Name_2$ **:** *signature* **)** **=** *structure*

```
# module Couple ( Q : sig type t end ) =  struct type couple = Q.t * Q.t end ;;
module Couple :
  functor(Q : sig type t end) -> sig type couple = Q.t * Q.t end
```

A functor can take several parameters:

Syntax :

functor ($Name_1$ **:** $signature_1$ **)** –>
⋮
functor ($Name_n$ **:** $signature_n$ **)** –>
structure

The syntactic sugar for defining and naming a functor extends to multiple-argument functors:

Syntax : **module** *Name* **(**$Name_1$ **:** $signature_1$ **)** ... **(** $Name_n$ **:** $signature_n$ **)** **=**
structure

The application of a functor to its arguments is written thus:

Syntax : **module** *Name* **=** *functor* **(** $structure_1$ **)** ... **(** $structure_n$ **)**

Note that each parameter is written between parentheses. The result of the application can be either a simple module or a partially applied functor, depending on the number of parameters of the functor.

Warning There is no equivalent to functors at the level of signature: it is not possible to build a signature by application of a "functorial signature" to other signatures.

A closed functor is a functor that does not reference any module except its parameters. Such a closed functor makes its communications with other modules entirely explicit. This provides maximal reusability, since the modules it references are determined at application time only. There is a strong parallel between a closed function (without free variables) and a closed functor.

Functors and Code Reuse

The Objective Caml standard library provides three modules defining functors. Two of them take as argument a module implementing a totally ordered data type, that is, a module with the following signature:

```
# module type OrderedType =
    sig
      type t
      val compare: t → t → int
    end ;;
module type OrderedType = sig type t val compare : t -> t -> int end
```

Function `compare` takes two arguments of type `t` and returns a negative integer if the first is less than the second, zero if both are equal, and a positive integer if the first is greater than the second. Here is an example of totally ordered type: pairs of integers equipped with lexicographic ordering.

```
# module OrderedIntPair =
    struct
      type t = int * int
      let compare (x1,x2) (y1,y2) =
        if x1 < y1 then -1
        else if x1 > y1 then 1
        else if x2 < y2 then -1
        else if x2 > y2 then 1
        else 0
    end ;;
module OrderedIntPair :
  sig type t = int * int val compare : 'a * 'b -> 'a * 'b -> int end
```

The functor `Make` from module `Map` returns a module that implements association tables whose keys are values of the ordered type passed as argument. This module provides operations similar to the operations on association lists from module `List`, but using a more efficient and more complex data structure (balanced binary trees).

```
# module AssocIntPair = Map.Make (OrderedIntPair) ;;
module AssocIntPair :
  sig
    type key = OrderedIntPair.t
    and 'a t = 'a Map.Make(OrderedIntPair).t
    val empty : 'a t
    val add : key -> 'a -> 'a t -> 'a t
    val find : key -> 'a t -> 'a
    val remove : key -> 'a t -> 'a t
    val mem : key -> 'a t -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val map : ('a -> 'b) -> 'a t -> 'b t
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

```
  end
```

The `Make` functor allows to construct association tables over any key type for which we can write a `compare` function.

The standard library module `Set` also provides a functor named `Make` taking an ordered type as argument and returning a module implementing sets of sets of values of this type.

```
# module SetIntPair = Set.Make (OrderedIntPair) ;;
module SetIntPair :
  sig
    type elt = OrderedIntPair.t
    and t = Set.Make(OrderedIntPair).t
    val empty : t
    val is_empty : t -> bool
    val mem : elt -> t -> bool
    val add : elt -> t -> t
    val singleton : elt -> t
    val remove : elt -> t -> t
    val union : t -> t -> t
    val inter : t -> t -> t
    val diff : t -> t -> t
    val compare : t -> t -> int
    val equal : t -> t -> bool
    val subset : t -> t -> bool
    val iter : (elt -> unit) -> t -> unit
    val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a
    val cardinal : t -> int
    val elements : t -> elt list
    val min_elt : t -> elt
    val max_elt : t -> elt
    val choose : t -> elt
  end
```

The type `SetIntPair.t` is the type of sets of integer pairs, with all the usual set operations provided in `SetIntPair`, including a set comparison function `SetIntPair.compare`. To illustrate the code reuse made possible by functors, we now build sets of sets of integer pairs.

```
# module SetofSet = Set.Make (SetIntPair) ;;

# let x = SetIntPair.singleton (1,2) ;;          (* x = { (1,2) }        *)
val x : SetIntPair.t = <abstr>
# let y = SetofSet.singleton SetIntPair.empty ;; (* y = { {} }           *)
val y : SetofSet.t = <abstr>
# let z = SetofSet.add x y ;;                    (* z = { {(1,2)} ; {} } *)
val z : SetofSet.t = <abstr>
```

The `Make` functor from module `Hashtbl` is similar to that from the `Map` module, but implements (imperative) hash tables instead of (purely functional) balanced trees. The

argument to `Hashtbl.Make` is slightly different: in addition to the type of the keys for the hash table, it must provide an equality function testing the equality of two keys (instead of a full-fledged comparison function), plus a hash function, that is, a function associating integers to keys.

```
# module type HashedType =
    sig
      type t
      val equal: t → t → bool
      val hash: t → int
    end ;;
module type HashedType =
  sig type t val equal : t -> t -> bool val hash : t -> int end
# module IntMod13 =
    struct
      type t = int
      let equal = (=)
      let hash x = x mod 13
    end ;;
module IntMod13 :
  sig type t = int val equal : 'a -> 'a -> bool val hash : int -> int end
# module TblInt = Hashtbl.Make (IntMod13) ;;
module TblInt :
  sig
    type key = IntMod13.t
    and 'a t = 'a Hashtbl.Make(IntMod13).t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val add : 'a t -> key -> 'a -> unit
    val remove : 'a t -> key -> unit
    val find : 'a t -> key -> 'a
    val find_all : 'a t -> key -> 'a list
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
  end
```

Local Module Definitions

The Objective Caml core language allows a module to be defined locally to an expression.

Syntax :

```
let module Name = structure
  in expr
```

For instance, we can use the `Set` module locally to write a sort function over integer lists, by inserting each list element into a set and finally converting the set to the sorted list of its elements.

```
# let sort l =
```

```
    let module M =
       struct
         type t = int
         let compare x y =
           if x < y then -1 else if x > y then 1 else 0
       end
   in
     let module MSet = Set.Make(M)
     in  MSet.elements (List.fold_right MSet.add l MSet.empty) ;;
val sort : int list -> int list = <fun>

# sort [ 5 ; 3 ; 8 ; 7 ; 2 ; 6 ; 1 ; 4 ] ;;
- : int list = [1; 2; 3; 4; 5; 6; 7; 8]
```

Objective Caml does not allow a value to escape a `let module` expression if the type of the value is not known outside the scope of the expression.

```
# let test =
    let module Foo =
      struct
        type t
        let id x = (x:t)
      end
    in Foo.id ;;
Characters 15-101:
This 'let module' expression has type Foo.t -> Foo.t
In this type, the locally bound module name Foo escapes its scope
```

Extended Example: Managing Bank Accounts

We conclude this chapter by an example illustrating the main aspects of modular programming: type abstraction, multiple views of a module, and functor-based code reuse.

The goal of this example is to provide two modules for managing a bank account. One is intended to be used by the bank, and the other by the customer. The approach is to implement a general-purpose parameterized functor providing all the needed operations, then apply it twice to the correct parameters, constraining it by the signature corresponding to its final user: the bank or the customer.

Organization of the Program

The two end modules `BManager` and `CManager` are obtained by constraining the module `Manager`. The latter is obtained by applying the functor `FManager` to the modules

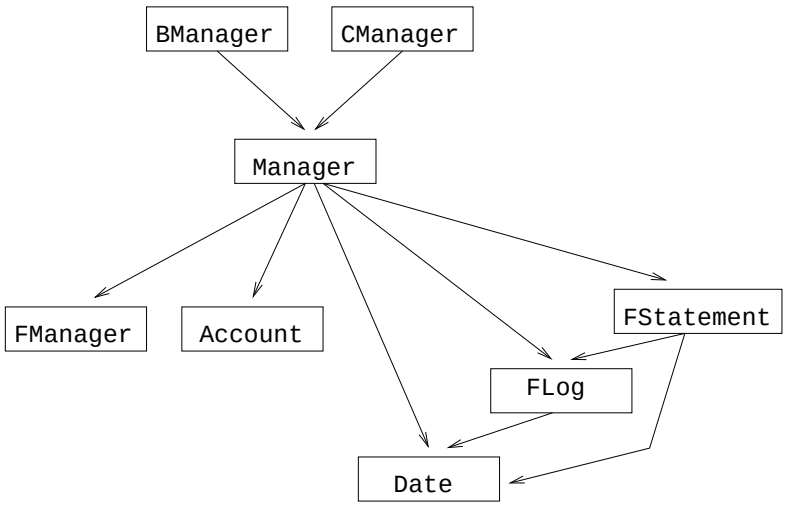


Figure 14.1: Modules dependency graph.

`Account`, `Date` and two additional modules built by application of the functors `FLog` and `FStatement`. Figure 14.1 illustrates these dependencies.

Signatures for the Module Parameters

The module for account management is parameterized by four other modules, whose signatures we now detail.

The bank account. This module provides the basic operations on the contents of the account.

```
# module type ACCOUNT = sig
    type t
    exception BadOperation
    val create : float → float → t
    val deposit : float → t → unit
    val withdraw : float → t → unit
    val balance : t → float
  end ;;
```

This set of functions provide the minimal operations on an account. The creation operation takes as arguments the initial balance and the maximal overdraft allowed. Excessive withdrawals may raise the `BadOperation` exception.

Ordered keys. Operations are recorded in an operation log described in the next paragraph. Each log entry is identified by a key. Key management functions are described by the following signature:

```
# module type OKEY =
```

```
    sig
      type t
      val create : unit → t
      val of_string : string → t
      val to_string : t → string
      val eq : t → t → bool
      val lt : t → t → bool
      val gt : t → t → bool
    end ;;
```

The `create` function returns a new, unique key. The functions `of_string` and `to_string` convert between keys and character strings. The three remaining functions are key comparison functions.

History. Logs of operations performed on an account are represented by the following abstract types and functions:

```
# module type LOG =
    sig
      type tkey
      type tinfo
      type t
      val create : unit → t
      val add : tkey → tinfo → t → unit
      val nth : int → t → tkey*tinfo
      val get : (tkey → bool) → t → (tkey*tinfo) list
    end ;;
```

We keep unspecified for now the types of the log keys (type *tkey*) and of the associated data (type *tinfo*), as well as the data structure for storing logs (type *t*). We assume that new informations added with the `add` function are kept in sequence. Two access functions are provided: access by position in the log (function `nth`) and access following a search predicate on keys (function `get`).

Account statements. The last parameter of the manager module provides two functions for editing a statement for an account:

```
# module type STATEMENT =
     sig
       type tdata
       type tinfo
       val editB : tdata → tinfo
       val editC : tdata → tinfo
     end ;;
```

We leave abstract the type of data to process (*tdata*) as well as the type of informations extracted from the data (*tinfo*).

The Parameterized Module for Managing Accounts

Using only the information provided by the signatures above, we now define the general-purpose functor for managing accounts.

```
# module FManager =
    functor (C:ACCOUNT) →
    functor (K:OKEY) →
    functor (L:LOG with type tkey=K.t and type tinfo=float) →
    functor (S:STATEMENT with type tdata=L.t and type tinfo
            = (L.tkey*L.tinfo) list) →
      struct
        type t = { accnt : C.t; log : L.t }
        let create s d = { accnt = C.create s d; log = L.create() }
        let deposit s g =
          C.deposit s g.accnt ; L.add (K.create()) s g.log
        let withdraw s g =
          C.withdraw s g.accnt ; L.add (K.create()) (-.s) g.log
        let balance g = C.balance g.accnt
        let statement edit g =
          let f (d,i) = (K.to_string d) ^ ":" ^ (string_of_float i)
          in List.map f (edit g.log)
        let statementB = statement S.editB
        let statementC = statement S.editC
      end ;;
module FManager :
  functor(C : ACCOUNT) ->
    functor(K : OKEY) ->
      functor
        (L : sig
               type tkey = K.t
               and tinfo = float
               and t
               val create : unit -> t
               val add : tkey -> tinfo -> t -> unit
               val nth : int -> t -> tkey * tinfo
               val get : (tkey -> bool) -> t -> (tkey * tinfo) list
             end) ->
        functor
          (S : sig
                 type tdata = L.t
                 and tinfo = (L.tkey * L.tinfo) list
                 val editB : tdata -> tinfo
                 val editC : tdata -> tinfo
               end) ->
          sig
            type t = { accnt: C.t; log: L.t }
            val create : float -> float -> t
            val deposit : L.tinfo -> t -> unit
            val withdraw : float -> t -> unit
            val balance : t -> float
            val statement : (L.t -> (K.t * float) list) -> t -> string list
            val statementB : t -> string list
```

```
  val statementC : t -> string list
end
```

Sharing between types. The type constraint over the parameter `L` of the `FManager` functor indicates that the keys of the log are those provided by the `K` parameter, and that the informations stored in the log are floating-point numbers (the transaction amounts). The type constraint over the `S` parameter indicates that the informations contained in the statement come from the log (the `L` parameter). The signature inferred for the `FManager` functor reflects the type sharing constraints in the inferred signatures for the functor parameters.

The type `t` in the result of `FManager` is a pair of an account (`C.t`) and its transaction log.

Operations. All operations defined in this functor are defined in terms of lower-level functions provided by the module parameters. The creation, deposit and withdrawal operations affect the contents of the account and add an entry in its transaction log. The other functions return the account balance and edit statements.

Implementing the Parameters

Before building the end modules, we must first implement the parameters to the `FManager` module.

Accounts. The data structure for an account is composed of a float representing the current balance, plus the maximum overdraft allowed. The latter is used to check withdrawals.

```
# module Account:ACCOUNT =
    struct
     type t = { mutable balance:float; overdraft:float }
     exception BadOperation
     let create b o = { balance=b; overdraft=(-. o) }
     let deposit s c =  c.balance <- c.balance +. s
     let balance c = c.balance
     let withdraw s c =
      let ss = c.balance -. s in
       if ss < c.overdraft then raise BadOperation
       else c.balance <- ss
    end ;;
module Account : ACCOUNT
```

Choosing log keys. We decide that keys for transaction logs should be the date of the transaction, expressed as a floating-point number as returned by the `time` function from module `Unix`.

```
# module Date:OKEY =
    struct
     type t = float
     let create() = Unix.time()
     let of_string = float_of_string
     let to_string = string_of_float
     let eq = (=)
     let lt = (<)
     let gt = (>)
    end ;;
module Date : OKEY
```

The log. The transaction log depends on a particular choice of log keys. Hence we define logs as a functor parameterized by a key structure.

```
# module FLog (K:OKEY) =
    struct
     type tkey = K.t
     type tinfo = float
     type t = { mutable contents : (tkey*tinfo) list }
     let create() = { contents = [] }
     let add c i l = l.contents <- (c,i) :: l.contents
     let nth i l = List.nth l.contents i
     let get f l = List.filter (fun (c,_) → (f c)) l.contents
    end ;;
module FLog :
  functor(K : OKEY) ->
    sig
      type tkey = K.t
      and tinfo = float
      and t = { mutable contents: (tkey * tinfo) list }
      val create : unit -> t
      val add : tkey -> tinfo -> t -> unit
      val nth : int -> t -> tkey * tinfo
      val get : (tkey -> bool) -> t -> (tkey * tinfo) list
    end
```

Notice that the type of informations stored in log entries must be consistent with the type used in the account manager functor.

Statements. We define two functions for editing statements. The first (`editB`) lists the five most recent transactions, and is intended for the bank; the second (`editC`) lists all transactions performed during the last 10 days, and is intended for the customer.

```
# module FStatement (K:OKEY) (L:LOG with type tkey=K.t) =
```

```
    struct
     type tdata = L.t
     type tinfo = (L.tkey*L.tinfo) list
     let editB h =
      List.map (fun i → L.nth i h) [0;1;2;3;4]
     let editC h =
      let c0 = K.of_string (string_of_float ((Unix.time()) -. 864000.)) in
      let f = K.lt c0 in
       L.get f h
    end ;;
module FStatement :
  functor(K : OKEY) ->
    functor
      (L : sig
             type tkey = K.t
             and tinfo
             and t
             val create : unit -> t
             val add : tkey -> tinfo -> t -> unit
             val nth : int -> t -> tkey * tinfo
             val get : (tkey -> bool) -> t -> (tkey * tinfo) list
           end) ->
       sig
         type tdata = L.t
         and tinfo = (L.tkey * L.tinfo) list
         val editB : L.t -> (L.tkey * L.tinfo) list
         val editC : L.t -> (L.tkey * L.tinfo) list
       end
```

In order to define the 10-day statement, we need to know exactly the implementation of keys as floats. This arguably goes against the principles of type abstraction. However, the key corresponding to ten days ago is obtained from its string representation by calling the `K.of_string` function, instead of directly computing the internal representation of this date. (Our example is probably too simple to make this subtle distinction obvious.)

End modules. To build the modules `MBank` and `MCustomer`, for use by the bank and the customer respectively, we proceed as follows:

1. define a common "account manager" structure by application of the `FManager` functor;
2. declare two signatures listing only the functions accessible to the bank or to the customer;
3. constrain the structure obtained in 1 with the signatures declared in 2.

```
# module Manager =
    FManager (Account)
             (Date)
```

```
            (FLog(Date))
            (FStatement (Date) (FLog(Date))) ;;
module Manager :
  sig
    type t =
      FManager(Account)(Date)(FLog(Date))(FStatement(Date)(FLog(Date))).t =
      { accnt: Account.t;
        log: FLog(Date).t }
    val create : float -> float -> t
    val deposit : FLog(Date).tinfo -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statement :
      (FLog(Date).t -> (Date.t * float) list) -> t -> string list
    val statementB : t -> string list
    val statementC : t -> string list
  end

# module type MANAGER_BANK =
    sig
     type t
     val create : float → float → t
     val deposit : float → t → unit
     val withdraw : float → t → unit
     val balance : t → float
     val statementB : t →  string list
    end ;;

# module MBank = (Manager:MANAGER_BANK with type t=Manager.t) ;;
module MBank :
  sig
    type t = Manager.t
    val create : float -> float -> t
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementB : t -> string list
  end

# module type MANAGER_CUSTOMER =
    sig
     type t
     val deposit : float → t → unit
     val withdraw : float → t → unit
     val balance : t → float
     val statementC : t →  string list
    end ;;

# module MCustomer = (Manager:MANAGER_CUSTOMER with type t=Manager.t) ;;
module MCustomer :
  sig
    type t = Manager.t
```

```
    val deposit : float -> t -> unit
    val withdraw : float -> t -> unit
    val balance : t -> float
    val statementC : t -> string list
  end
```

In order for accounts created by the bank to be usable by clients, we added the type constraint on *`Manager.t`* in the definition of the `MBank` and `MCustomer` structures, to ensure that their *`t`* type components are compatible.

Exercises

Association Lists

In this first simple exercise, we will implement a polymorphic abstract type for association lists, and present two different views of the implementation.

1. Define a signature `ALIST` declaring an abstract type with two type parameters (one for the keys, the other for the associated values), a creation function, an add function, a lookup function, a membership test, and a deletion function. The interface should be functional, *i.e.* without in-place modifications of the abstract type.
2. Define a module `Alist` implementing the signature `ALIST`
3. Define a signature `ADM_ALIST` for "administrators" of association lists. Administrators can only create association lists, and add or remove entries from a list.
4. Define a signature `USER_ALIST` for "users" of association lists. Users can only perform lookups and membership tests.
5. Define two modules `AdmAlist` and `UserAlist` for administrators and for users. Keep in mind that users must be able to access lists created by administrators.

Parameterized Vectors

This exercise illustrates the genericity and code reuse abilities of parameterized modules. We will define a functor for manipulating two-dimensional vectors (pairs of (x, y) coordinates) that can be instantiated with different types for the coordinates.

Numbers have the following signature:

```
# module type NUMBER =
  sig
   type a
   type t
   val create : a → t
   val add : t → t → t
   val string_of : t → string
```

```
end ;;
```

1. Define the functor `FVector`, parameterized by a module of signature `NUMBER`, and defining a type *`t`* of two-dimensional vectors over these numbers, a creation function, an addition function, and a conversion to strings.
2. Define a signature `VECTOR`, without parameters, where the types of numbers and vectors are abstract.
3. Define three structures `Rational`, `Float` et `Complex` implementing the signature `NUMBER`.
4. Use these structures to define (by functor application) three modules for vectors of rationals, reals and complex.

Lexical Trees

This exercise follows up on the lexical trees introduced in chapter 2, page 63. The goal is to define a generic module for handling lexical trees, parameterized by an abstract type of words.

1. Define the signature `WORD` defining an abstract type *`alpha`* for letters of the alphabet, and another abstract type *`t`* for words on this alphabet. Declare also the empty word, the conversion from an alphabet letter to a one-letter word, the accessor to a letter of a word, the sub-word operation, the length of a word, and word concatenation.
2. Define the functor `LexTree`, parameterized by a module implementing `WORD`, that defines (as a function of the types and operations over words) the type of lexical trees and functions `exists`, `insert` et `select` similar to those from chapter 2, page 63.
3. Define the module `Chars` implementing the `WORD` signature for the types *`alpha = char`* and *`t = string`*. Use it to obtain a module `CharDict` implementing dictionaries whose keys are character strings.

Summary

In this chapter, we introduced all the facilities that the Objective Caml module language offers, in particular parameterized modules.

As all module systems, it reflects the duality between interfaces and implementations, here presented as a duality between signatures and structures. Signatures allow hiding information about type, value or exception definitions.

By hiding type representation, we can make certain types abstract, ensuring that values of these types can only be manipulated through the operations provided in the module signature. We saw how to exploit this mechanism to facilitate sharing of values hidden in closures, and to offer multiple views of a given implementation. In the latter

case, explicit type sharing annotations are sometimes necessary to achieve the desired behavior.

Parameterized modules, also called functors, go one step beyond and support code reuse through simple mechanisms similar to function abstraction and function application.

To Learn More

Other examples of modules and functors can be found in chapter 4 of the Objective Caml manual.

The underlying theory and the type checking for modules can be found in a number of research articles and course notes by Xavier Leroy, at

Link: http://cristal.inria.fr/~xleroy

The Objective Caml module system follows the same principles as that of its cousin the SML language. Chapter 22 compares these two languages in more details and provides bibliographical references for the interested reader.

Other languages feature advanced module systems, in particular Modula-3 (2 and 3), and ADA. They support the definition of modules parameterized by types and values.

15

Object-Oriented Programming

As you may have guessed from the name, Objective Caml supports object-oriented programming. Unlike imperative programming, in which execution is driven by explicit sequencing of operations, or functional programming, where it is driven by the required computations, object-oriented programming can be thought of as data driven. Using objects introduces a new organization of programs into classes of related objects. A class groups together data and operations. The latter, also known as *methods*, define the possible behaviors of an object. A method is invoked by *sending a message* to an object. When an object receives a message, it performs the action or the computation corresponding to the method specified by the message. This is different from applying a function to arguments because a message (which contains the method name) is sent to an object. It is up to the object itself to determine the code that will actually be executed; such a *delayed* binding between name and code makes behavior more adaptable and code easier to reuse.

With object-oriented programming, relations are defined between classes. Classes also define how objects communicate through message parameters. *Aggregation* and *inheritance* relations between classes allow new kinds of application modeling. A class that inherits from another class includes all definitions from the parent's class. However, it may extend the set of data and methods and redefine inherited behaviors, provided typing constraints are respected. We will use a graphical notation[1] to represent relations between classes.

Objective Caml's object extensions are integrated with the type system of the language: a class declaration defines a type with the same name as the class. Two kinds of polymorphism coexist. One of them is parametric polymorphism, which we have already seen with parameterized types: parameterized classes. The other one, known as *inclusion polymorphism*, uses the subtyping relation between objects and delayed binding. If the type of the class `sc` is a subtype of the class `c` then any object from `sc`

1. A number of notations exist for describing relations, e.g. UML (*Unified Modeling Language*).

may be used in place of an object from `c`. The subtype constraint must be stated explicitly. Inclusion polymorphism makes it possible to construct non-homogeneous lists where the type of each element is a subtype of a type common to all list elements. Since binding is delayed, sending the same message to all elements of such a list can activate different methods according to the sub-classes of the actual elements.

On the other hand, Objective Caml does not include the notion of method overloading, which would allow several definitions for one method name. Without this restriction, type inference might encounter ambiguous situations requiring additional information from the programmer.

It should be emphasized that Objective Caml is the only language with an object extension that provides both parameterized and inclusion polymorphism, while still being fully statically typed through type inference.

Chapter Plan

This chapter describes Objective Caml's object extension. This extension does not change any of the features of the language that we already studied in the previous chapters. A few new reserved keywords are added for the object-oriented syntax.

The first section describes class declaration syntax, object instantiation, and message passing. The second section explains the various relations that may exist between classes. The third section clarifies the notion of *object type* and demonstrates the richness of the object extension, thanks to abstract classes, multiple inheritance, and generic parameterized classes. The fourth section explains the subtyping relation and shows its power through inclusion polymorphism. The fifth section deals with a functional style of object-oriented programming, where the internal state of the object is not modified, but a modified copy of the receiving object is returned. The sixth section clarifies other parts of the object-oriented extension, such as interfaces and local declarations in classes, which allow class variables to be created.

Classes, Objects, and Methods

The object-oriented extension of Objective Caml is integrated with the functional and imperative kernels of the language, as well as with its type system. Indeed, this last point is unique to the language. Thus we have an object-oriented, statically typed language, with type inference. This extension allows definition of classes and instances, class inheritance (including multiple inheritance), parameterized classes, and abstract classes. Class interfaces are generated from their definition, but may be made more precise through a signature, similarly to what is done for modules.

Object-Oriented Terminology

We summarize below the main object-oriented programming terms.

class: a *class* describes the contents of the objects that belong to it: it describes an aggregate of data fields (called *instance variables*), and defines the operations (called *methods*).

object: an object is an element (or *instance*) of a class; objects have the behaviors of their class. The object is the actual component of programs, while the class specifies how instances are created and how they behave.

method: a method is an action which an object is able to perform.

sending a message *sending a message* to an object means asking the object to execute or *invoke* one of its methods.

Class Declaration

The simplest syntax for defining a class is as follows. We shall develop this definition throughout this chapter.

Syntax :

```
class name p1 ... pn =
  object
      ⋮
    instance variables
      ⋮
    methods
      ⋮
  end
```

$p_1, \ldots, p_n$ are the parameters for the constructor of the class; they are omitted if the class has no parameters.

An instance variable is declared as follows:

Syntax :

```
val name = expr
or
val mutable name = expr
```

When a data field is declared **mutable**, its value may be modified. Otherwise, the value is always the one that was computed when *expr* was evaluated during object creation.

Methods are declared as follows:

Syntax :

```
method name p1 ... pn = expr
```

Other clauses than **val** and **method** can be used in a class declaration: we shall introduce them as needed.

Our first class example. We start with the unavoidable class `point`:

- the data fields `x` and `y` contain the coordinates of the point,
- two methods provide access to the data fields (`get_x` and `get_y`),
- two displacement methods (`moveto`: absolute displacement) and (`rmoveto`: relative displacement),
- one method presents the data as a *string* (`to_string`),
- one method computes the distance to the point from the origin (`distance`).

```
# class point (x_init,y_init) =
    object
      val mutable x = x_init
      val mutable y = y_init
      method get_x = x
      method get_y = y
      method moveto (a,b) =  x <- a ; y <- b
      method rmoveto (dx,dy) =  x <- x + dx ; y <- y + dy
      method to_string () =
          "( " ^ (string_of_int x) ^ ", " ^ (string_of_int y) ^")"
      method distance () = sqrt (float(x*x + y*y))
    end ;;
```

Note that some methods do not need parameters; this is the case for `get_x` and `get_y`. We usually access instance variables with parameterless methods.

After we declare the class `point`, the system prints the following text:

```
class point :
  int * int ->
  object
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

This text contains two pieces of information. First, the type for objects of the class; this type will be abbreviated as *point*. The type of an object is the list of names and types of methods in its class. In our example, *point* is an abbreviation for:

```
  < distance : unit → unit; get_x : int; get_y : int;
    moveto : int * int → unit; rmoveto : int * int → unit;
    to_string : unit → unit >
```

Next, we have a constructor for instances of class `point`, whose type is *int*int* `-->` *point*. The constructor allows us to construct `point` objects (we´ll just say "`point`s" to be brief) from the initial values provided as arguments. In this case, we construct a

`point` from a pair of integers (meaning the initial position). The constructor `point` is used with the keyword **`new`**.

It is possible to define class types:

```
# type simple_point = < get_x : int; get_y : int; to_string : unit → unit > ;;
type simple_point = < get_x : int; get_y : int; to_string : unit -> unit >
```

Note

Type *`point`* does not repeat all the informations shown after a class declaration. Instance variables are not shown in the type. Only methods have access to these instance variables.

Warning A class declaration is a type declaration. As a consequence, it cannot contain a free type variable.

We will come back to this point later when we deal with type constraints (page 454) and parameterized classes (page 460).

A Graphical Notation for Classes

We adapt the UML notation for the syntax of Objective Caml types. Classes are denoted by a rectangle with three parts:

- the top part shows the name of the class,
- the middle part lists the attributes (data fields) of a class instance,
- the bottom part shows the methods of an instance of the class.

Figure 15.1 gives an example of the graphical representation for the class `caml`.

Figure 15.1: Graphical representation of a class.

Type information for the fields and methods of a class may be added.

Instance Creation

An object is a value of a class, called an *instance* of the class. Instances are created with the generic construction primitive `new`, which takes the class and initialization values as arguments.

Syntax : **new** *name* $expr_1 \ldots expr_n$

The following example creates several instances of class ***point***, from various initial values.

```
# let p1 = new point (0,0);;
val p1 : point = <obj>
# let p2 = new point (3,4);;
val p2 : point = <obj>
# let coord = (3,0);;
val coord : int * int = 3, 0
# let p3 = new point coord;;
val p3 : point = <obj>
```

In Objective Caml, the constructor of a class is unique, but you may define your own specific function `make_point` for point creation:

```
# let make_point x = new point (x,x) ;;
val make_point : int -> point = <fun>
# make_point 1 ;;
- : point = <obj>
```

Sending a Message

The notation `#` is used to send a message to an object. [2]

Syntax : obj_1**#***name* $p_1 \ldots p_n$

The message with method name "*name*" is sent to the object *obj*. The arguments p_1, $\ldots$, p_n are as expected by the method *name*. The method must be defined by the class of the object, i.e. visible in the type. The types of arguments must conform to the types of the formal parameters. The following example shows several queries performed on objects from the class `point`.

```
# p1#get_x;;
- : int = 0
# p2#get_y;;
- : int = 4
# p1#to_string();;
- : string = "( 0, 0)"
# p2#to_string();;
```

2. In most object-oriented languages, a dot notation is used. However, the dot notation was already used for records and modules, so a new symbol was needed.

```
- : string = "( 3, 4)"
# if (p1#distance()) = (p2#distance())
   then print_string ("That's just chance\n")
   else print_string ("We could bet on it\n");;
We could bet on it
- : unit = ()
```

From the type point of view, objects of type ***point*** can be used by polymorphic functions of Objective Caml, just as any other value in the language:

```
# p1 = p1 ;;
- : bool = true
# p1 = p2;;
- : bool = false
# let l = p1::[];;
val l : point list = [<obj>]
# List.hd l;;
- : point = <obj>
```

Warning Object equality is defined as physical equality.

We shall clarify this point when we study the subtyping relation (page 469).

Relations between Classes

Classes can be related in two ways:

1. An aggregation relation, named *Has-a*:
   class C_2 is related by *Has-a* with class C_1 when C_2 has a field whose type is that of class C_1. This relation can be generalized as: C_2 has at least one field whose type is that of class C_1.
2. An inheritance relation, named *Is-a*:
   class C_2 is a subclass of class C_1 when C_2 extends the behavior of C_1. One big advantage of object-oriented programming is the ability to extend the behavior of an existing class while reusing the code written for the original class. When a class is extended, the new class inherits all the fields (data and methods) of the class being extended.

Aggregation

Class C_1 aggregates class C_2 when at least one of its instance variables has type C_2. One gives the arity of the aggregation relation when it is known.

An Example of Aggregation

Let us define a figure as a set of points. Therefore we declare class *picture* (see figure 15.2), in which one of the fields is an array of points. Then the class *picture* aggregates *point*, using the generalized relation *Has-a*.

```
# class picture n =
    object
      val mutable ind = 0
      val tab = Array.create n (new point(0,0))
      method add p =
        try  tab.(ind)<-p ; ind <- ind + 1
        with Invalid_argument("Array.set")
                → failwith ("picture.add:ind =" ^ (string_of_int ind))
      method remove () = if (ind > 0) then ind <-ind-1
      method to_string () =
        let s = ref "["
        in for i=0 to ind-1 do s:= !s ^ " " ^ tab.(i)#to_string() done ;
           (!s) ^ "]"
    end ;;
class picture :
  int ->
  object
    val mutable ind : int
    val tab : point array
    method add : point -> unit
    method remove : unit -> unit
    method to_string : unit -> string
  end
```

To build a figure, we create an instance of class *picture*, and insert the points as required.

```
# let pic = new picture 8;;
val pic : picture = <obj>
# pic#add p1; pic#add p2; pic#add p3;;
- : unit = ()
# pic#to_string ();;
- : string = "[ ( 0, 0) ( 3, 4) ( 3, 0)]"
```

A Graphical Notation for Aggregation

The relation between class *picture* and class *point* is represented graphically in figure 15.2. An arrow with a diamond at the tail represents aggregation. In this example, class *picture* has 0 or more points. Furthermore, we show above the arrow the arity of the relation.

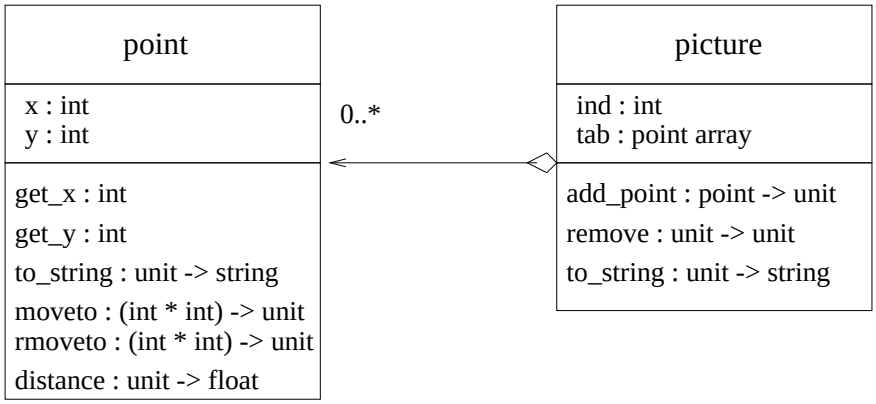


Figure 15.2: Aggregation relation.

Inheritance Relation

This is the main relation in object-oriented programming. When class `c2` inherits from class `c1`, it inherits all fields from the parent class. It can also define new fields, or redefine inherited methods to specialize them. Since the parent class has not been modified, the applications using it do not need to be adapted to the changes introduced in the new class.

The syntax of inheritance is as follows:

Syntax : **`inherit`** $name_1$ p_1 $\ldots$ p_n [**`as`** $name_2$]

Parameters p_1, $\ldots$, p_n are what is expected from the constructor of class $name_1$. The optional keyword **`as`** associates a name with the parent class to provide access to its methods. This feature is particularly useful when the child class redefines a method of the parent class (see page 445).

An Example of Simple Inheritance

Using the classic example, we can extend class ***`point`*** by adding a color attribute to the points. We define the class ***`colored_point`*** inheriting from class ***`point`***. The color is represented by the field `c` of type ***`string`***. We add a method `get_color` that returns the value of the field. Finally, the string conversion method is overridden to recognize the new attribute.

Note

The `x` and `y` variables seen in `to_string` are the fields, not the class initialization arguments.

```
# class colored_point (x,y) c =
    object
```

```
      inherit point (x,y)
      val mutable c = c
      method get_color = c
      method set_color nc = c <- nc
      method to_string () = "( " ^ (string_of_int x) ^
                            ", " ^ (string_of_int y) ^ ")" ^
                            " [" ^ c ^ "] "
    end ;;
class colored_point :
  int * int ->
  string ->
  object
    val mutable c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_color : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method set_color : string -> unit
    method to_string : unit -> string
  end
```

The constructor arguments for *colored_point* are the pair of coordinates required for the construction of a *point* and the color for the colored point.

The methods inherited, newly defined or redefined correspond to the behaviors of instances of the class.

```
# let pc = new colored_point (2,3) "white";;
val pc : colored_point = <obj>
# pc#get_color;;
- : string = "white"
# pc#get_x;;
- : int = 2
# pc#to_string();;
- : string = "( 2, 3) [white] "
# pc#distance;;
- : unit -> float = <fun>
```

We say that the class *point* is a *parent* class of class *colored_point* and that the latter is the *child* of the former.

Warning When redefining a method in a child class, you must respect the method type defined in the parent class.

A Graphical Notation for Inheritance

The inheritance relation between classes is denoted by an arrow from the child class to the parent class. The head of the arrow is a closed triangle. In the graphical represen-

tation of inheritance, we only show the new fields and methods, and redefined methods in the child class. Figure 15.3 displays the relation between class *colored_point* and its parent *point*.

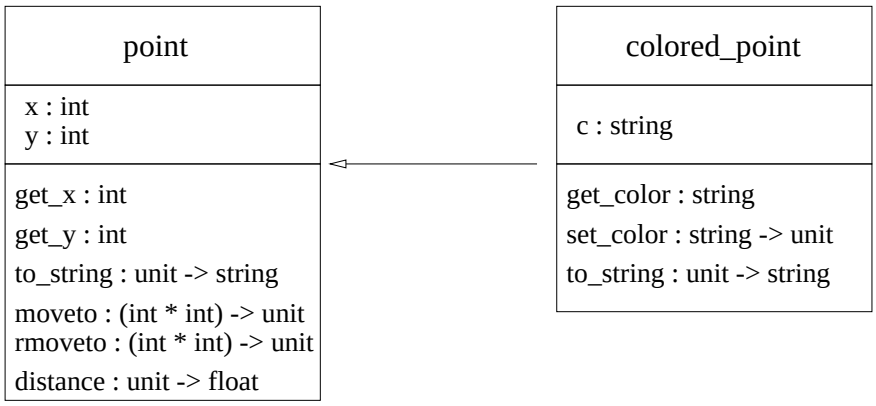


Figure 15.3: Inheritance Relation.

Since it contains additional methods, type *colored_point* differs from type *point*. Testing for equality between instances of these classes produces a long error message containing the whole type of each class, in order to display the differences.

```
# p1 = pc;;
Characters 6-8:
This expression has type
  colored_point =
    < distance : unit -> float; get_color : string; get_x : int; get_y :
      int; moveto : int * int -> unit; rmoveto : int * int -> unit;
      set_color : string -> unit; to_string : unit -> string >
but is here used with type
  point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method get_color
```

Other Object-oriented Features

References: `self` *and* `super`

When defining a method in a class, it may be convenient to be able to invoke a method from a parent class. For this purpose, Objective Caml allows the object itself, as well as (the objects of) the parent class to be named. In the former case, the chosen name is given after the keyword **`object`**, and in the latter, after the inheritance declaration.

For example, in order to define the method `to_string` of colored points, it is better to invoke the method `to_string` from the parent class and to extend its behavior with a new method, `get_color`.

```
# class colored_point (x,y) c  =
    object (self)
      inherit point (x,y) as super
      val c = c
      method get_color = c
      method to_string () = super#to_string() ^  " [" ^ self#get_color ^ "] "
    end ;;
```

Arbitrary names may be given to the parent and child class objects, but the names `self` and `this` for the current class and `super` for the parent are conventional. Choosing other names may be useful with multiple inheritance since it makes it easy to differentiate the parents (see page 457).

Warning You may not reference a variable of an instance's parent if you declare a new variable with the same name since it masks the former.

Delayed Binding

With *delayed binding* the method used when a message is sent is decided at runtime; this is opposed to static binding where the decision is made at compile time. In Objective Caml, delayed binding of methods is used; therefore, the exact piece of code to be executed is determined by the recipient of the message.

The above declaration of class *colored_point* redefines the method `to_string`. This new definition uses method `get_color` from this class. Now let us define another class *colored_point_1*, inheriting from *colored_point*; this new class redefines method `get_color` (testing that the character string is appropriate), but does not redefine `to_string`.

```
# class colored_point_1 coord c =
    object
      inherit colored_point coord  c
      val true_colors = ["white"; "black"; "red"; "green"; "blue"; "yellow"]
      method get_color = if List.mem c true_colors then c else "UNKNOWN"
    end ;;
```

Method `to_string` is the same in both classes of colored points; but two objects from these classes will have a different behavior.

```
# let p1 = new colored_point (1,1) "blue as an orange" ;;
val p1 : colored_point = <obj>
# p1#to_string();;
- : string = "( 1, 1) [blue as an orange] "
# let p2 = new colored_point_1 (1,1) "blue as an orange" ;;
```

```
val p2 : colored_point_1 = <obj>
# p2#to_string();;
- : string = "( 1, 1) [UNKNOWN] "
```

The binding of `get_color` within `to_string` is not fixed when the class *`colored_point`* is compiled. The code to be executed when invoking the method `get_color` is determined from the methods associated with instances of classes *`colored_point`* and *`colored_point_1`*. For an instance of *`colored_point`*, sending the message `to_string` causes the execution of `get_color`, defined in class *`colored_point`*. On the other hand, sending the same message to an instance of *`colored_point_1`* invokes the method from the parent class, and the latter triggers method `get_color` from the child class, controlling the relevance of the string representing the color.

Object Representation and Message Dispatch

An object is split in two parts: one may vary, the other is fixed. The varying part contains the instance variables, just as for a record. The fixed part corresponds to a methods table, shared by all instances of the class.

The methods table is a sparse array of functions. Every method name in an application is given a unique id that serves as an index into the methods table. We assume the existence of a machine instruction `GETMETHOD(o,n)`, that takes two parameters: an object `o` and an index `n`. It returns the function associated with this index in the methods table. We write `f_n` for the result of the call `GETMETHOD(o,n)`. Compiling the message send `o#m` computes the index `n` of the method name `m` and produces the code for applying `GETMETHOD(o,n)` to object `o`. This corresponds to applying function `f_n` to the receiving object `o`. Delayed binding is implemented through a call to `GETMETHOD` at run time.

Sending a message to `self` within a method is also compiled as a search for the index of the message, followed by a call to the function found in the methods table.

In the case of inheritance, since the method name always has the same index, regardless of redefinition, only the entry in new class' methods table is changed for redefinitions. So sending message `to_string` to an instance of class `point` will apply the conversion function of a point, while sending the same message to an instance of `colored_point` will find at the same index the function corresponding to the method which has been redefined to recognize the color field.

Thanks to this index invariance, subtyping (see page 465) is insured to be coherent with respect to the execution. Indeed if a colored point is explicitly constrained to be a point, then upon sending the message `to_string` the method index from class `point` is computed, which coincides with that from class `colored_point`. Searching for the method will be done within the table associated with the receiving instance, i.e. the `colored_point` table.

Although the actual implementation in Objective Caml is different, the principle of dynamic search for the method to be used is still the same.

Initialization

The class definition keyword **initializer** is used to specify code to be executed during object construction. An initializer can perform any computation and field access that is legal in a method.

Syntax : **initializer** *expr*

Let us again extend the class `point`, this time by defining a verbose point that will announce its creation.

```
# class verbose_point p =
    object(self)
      inherit point p
      initializer
        let xm = string_of_int x and ym = string_of_int y
        in Printf.printf ">> Creation of a point at (%s %s)\n"
                         xm ym ;
           Printf.printf "   , at distance %f from the origin\n"
                         (self#distance()) ;
    end ;;

# new verbose_point (1,1);;
>> Creation of a point at (1 1)
   , at distance 1.414214 from the origin
- : verbose_point = <obj>
```

An amusing but instructive use of initializers is tracing class inheritance on instance creation. Here is an example:

```
# class c1 =
    object
      initializer print_string "Creating an instance of c1\n"
    end ;;

# class c2 =
    object
      inherit c1
      initializer print_string "Creating an instance of c2\n"
    end ;;

# new c1 ;;
Creating an instance of c1
- : c1 = <obj>
# new c2 ;;
Creating an instance of c1
Creating an instance of c2
- : c2 = <obj>
```

Constructing an instance of *c2* requires first constructing an instance of the parent class.

Private Methods

A method may be declared *private* with the keyword **private**. It will appear in the interface to the class but not in instances of the class. A private method can only be invoked from other methods; it cannot be sent to an instance of the class. However, private methods are inherited, and therefore can be used in definitions of the hierarchy[3].

Syntax : **method private** *name* = *expr*

Let us extend the class *point*: we add a method `undo` that revokes the last move. To do this, we must remember the position held before performing a move, so we introduce two new fields, `old_x` and `old_y`, together with their update method. Since we do not want the user to have direct access to this method, we declare it as private. We redefine the methods `moveto` and `rmoveto`, keeping note of the current position before calling the previous methods for performing a move.

```
# class point_m1 (x0,y0) =
    object(self)
      inherit point (x0,y0) as super
      val mutable old_x = x0
      val mutable old_y = y0
      method private mem_pos () = old_x <- x ; old_y <- y
      method undo () = x <- old_x; y <- old_y
      method moveto (x1, y1) = self#mem_pos () ; super#moveto (x1, y1)
      method rmoveto (dx, dy) = self#mem_pos () ; super#rmoveto (dx, dy)
  end ;;
class point_m1 :
  int * int ->
  object
    val mutable old_x : int
    val mutable old_y : int
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method get_x : int
    method get_y : int
    method private mem_pos : unit -> unit
    method moveto : int * int -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
    method undo : unit -> unit
  end
```

We note that method `mem_pos` is preceded by the keyword **private** in type *point_m1*. It can be invoked from within method `undo`, but not on another instance. The situation is the same as for instance variables. Even though fields `old_x` and `old_y` appear in the results shown by compilation, that does not imply that they may be handled directly (see page 438).

```
# let p = new point_m1 (0, 0) ;;
```

3. The **private** of Objective Caml corresponds to **protected** of Objective C, C++ and Java

```
val p : point_m1 = <obj>
# p#mem_pos() ;;
Characters 0-1:
This expression has type point_m1
It has no method mem_pos
# p#moveto(1, 1) ; p#to_string() ;;
- : string = "( 1, 1)"
# p#undo() ; p#to_string() ;;
- : string = "( 0, 0)"
```

Warning A type constraint may make public a method declared with attribute **private**.

Types and Genericity

Besides the ability to model a problem using aggregation and inheritance relations, object-oriented programming is interesting because it provides the ability to reuse or modify the behavior of classes. However, extending an Objective Caml class must preserve the static typing properties of the language.

With abstract classes, you can factorize code and group their subclasses into one "communication protocol". An abstract class fixes the names and types of messages that may be received by instances of child classes. This technique will be better appreciated in connection with multiple inheritance.

The notion of an *open object type* (or simply an *open type*) that specifies the required methods allows code to work with instances using generic functions. But you may need to make the type constraints precise; this will be necessary for parameterized classes, which provide the genericity of parameterized polymorphism in the context of classes. With this latter object layer feature, Objective Caml can really be generic.

Abstract Classes and Methods

In abstract classes, some methods are declared without a body. Such methods are called *abstract*. It is illegal to instantiate an abstract class; **new** cannot be used. The keyword **virtual** is used to indicate that a class or method is abstract.

Syntax : **class virtual** *name* **= object** ... **end**

A class *must* be declared abstract as soon as one of its methods is abstract. A method is declared abstract by providing only the method type.

Syntax : **method virtual** *name* : *type*

When a subclass of an abstract class redefines *all* of the abstract methods of its parent, then it may become concrete; otherwise it also has to be declared abstract.

As an example, suppose we want to construct a set of displayable objects, all with a method `print` that will display the object's contents translated into a character string. All such objects need a method `to_string`. We define class `printable`. The string may vary according to the nature of the objects that we consider; therefore method `to_string` is abstract in the declaration of `printable` and consequently the class is also abstract.

```
# class virtual printable () =
    object(self)
      method virtual to_string : unit → string
      method print () = print_string (self#to_string())
    end ;;
class virtual printable :
  unit ->
  object
    method print : unit -> unit
    method virtual to_string : unit -> string
  end
```

We note that the abstractness of the class and of its method `to_string` is made clear in the type we obtain.

From this class, let us try to define the class hierarchy of figure 15.4.

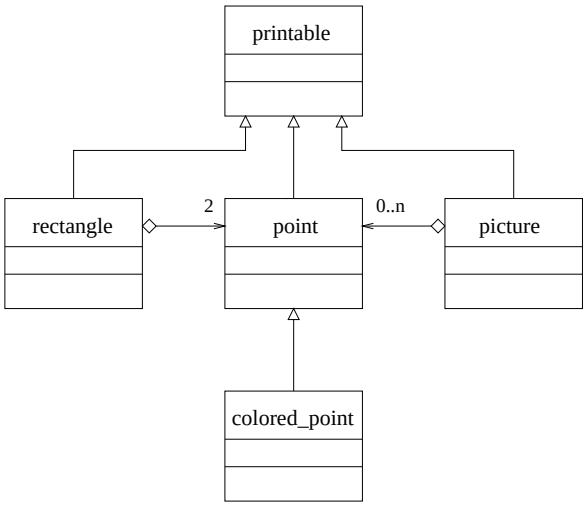


Figure 15.4: Relations between classes of displayable objects.

It is easy to redefine the classes *point*, *colored_point* and *picture* by adding to their declarations a line `inherit printable ()` that provides them with a method *print* through inheritance.

```
# let p = new point (1,1) in p#print() ;;
( 1, 1)- : unit = ()
# let pc = new colored_point (2,2) "blue" in pc#print() ;;
( 2, 2) with color blue- : unit = ()
# let t = new picture 3 in t#add (new point (1,1)) ;
```

```
                          t#add (new point (3,2)) ;
                          t#add (new point (1,4)) ;
                          t#print() ;;
[ ( 1, 1) ( 3, 2) ( 1, 4)]- : unit = ()
```

Subclass *rectangle* below inherits from *printable*, and defines method `to_string`. Instance variables `llc` (resp. `urc`) mean the lower left corner point (resp. upper right corner point) in the rectangle.

```
# class rectangle (p1,p2) =
    object
      inherit printable ()
      val llc = (p1 : point)
      val urc = (p2 : point)
      method to_string () = "[" ^ llc#to_string() ^ "," ^ urc#to_string() ^ "]"
    end ;;
class rectangle :
  point * point ->
  object
    val llc : point
    val urc : point
    method print : unit -> unit
    method to_string : unit -> string
  end
```

Class *rectangle* inherits from the abstract class *printable*, and thus receives method `print`. It has two instance variables of type *point*: the lower left corner (`llc`) and upper right corner. Method `to_string` sends the message `to_string` to its `point` instance variables `llc` and `urc`.

```
# let r = new rectangle (new point (2,3), new point (4,5));;
val r : rectangle = <obj>
# r#print();;
[( 2, 3),( 4, 5)]- : unit = ()
```

Classes, Types, and Objects

You may remember that the type of an object is determined by the type of its methods. For instance, the type *point*, inferred during the declaration of class *point*, is an abbreviation for type:

```
point =
  < distance : unit -> float; get_x : int; get_y : int;
    moveto : int * int -> unit; rmoveto : int * int -> unit;
    to_string : unit -> string  >
```

This is a closed type; that is, all methods and associated types are fixed. No additional methods and types are allowed. Upon a class declaration, the mechanism of type inference computes the closed type associated with class.

Open Types

Since sending a message to an object is part of the language, you may define a function that sends a message to an object whose type is still undefined.

```
# let f x = x#get_x ;;
val f : < get_x : 'a; .. > -> 'a = <fun>
```

The type inferred for the argument of `f` is an object type, since a message is sent to `x`, but this object type is *open*. In function `f`, parameter `x` must have at least a method `get_x`. Since the result of sending this message is not used within function `f`, its type has to be as general as possible (i.e. a variable of type *'a*). So type inference allows the function `f` to be used with any object having a method `get_x`. The double points (..) at the end of the type *< get_x : 'a; .. >* indicate that the type of `x` is open.

```
# f (new point(2,3)) ;;
- : int = 2
# f (new colored_point(2,3) "emerald") ;;
- : int = 2
# class c () =
    object
      method get_x = "I have a method get_x"
    end ;;
class c : unit -> object method get_x : string end
# f (new c ()) ;;
- : string = "I have a method get_x"
```

Type inference for classes may generate open types, particularly for initial values in instance construction. The following example constructs a class *couple*, whose initial values `a` and `b` have a method `to_string`.

```
# class couple (a,b) =
    object
      val p0 = a
      val p1 = b
      method to_string() = p0#to_string() ^ p1#to_string()
      method copy () = new couple (p0,p1)
    end ;;
class couple :
  (< to_string : unit -> string; .. > as 'a) *
  (< to_string : unit -> string; .. > as 'b) ->
  object
    val p0 : 'a
    val p1 : 'b
    method copy : unit -> couple
    method to_string : unit -> string
```

```
  end
```

The types of both `a` and `b` are open types, with method `to_string`. We note that these two types are considered to be different. They are marked "*as 'a*" and "*as 'b*", respectively. Variables of types *'a* and *'b* are constrained by the generated type.

We use the sharp symbol to indicate the open type built from a closed type *obj_type*:

Syntax : *#obj_type*

The type obtained contains all of the methods of type *obj_type* and terminates with a double point.

Type Constraints.

In the chapter on functional programming (see page 28), we showed how an expression can be constrained to have a type more precise than what is produced by inference. Object types (open or closed) can be used to enhance such constraints. One may want to open *a priori* the type of a defined object, in order to apply it to a forthcoming method. We can use an open object constraint:

Syntax : **(***name***:#***type***)**

Which allows us to write:

```
# let g (x : #point) = x#message;;
val g :
  < distance : unit -> float; get_x : int; get_y : int; message : 'a;
    moveto : int * int -> unit; print : unit -> unit;
    rmoveto : int * int -> unit; to_string : unit -> string; .. > ->
  'a = <fun>
```

The type constraint with *#point* forces `x` to have at least all of the methods of *point*, and sending message "`message`" adds a method to the type of parameter `x`.

Just as in the rest of the language, the object extension of Objective Caml provides static typing through inference. When this mechanism does not have enough information to determine the type of an expression, a type variable is assigned. We have just seen that this process is also valid for typing objects; however, it sometimes leads to ambiguous situations which the user must resolve by explicitly giving type information.

```
# class a_point p0 =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
Characters 6-89:
Some type variables are unbound in this type:
  class a_point :
    (< to_string : unit -> 'b; .. > as 'a) ->
    object val p : 'a method to_string : unit -> 'b end
The method to_string has type unit -> 'a where 'a is unbound
```

We resolve this ambiguity by saying that parameter p0 has type ***#point***.

```
# class a_point (p0 : #point) =
    object
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

In order to set type constraints in several places in a class declaration, the following syntax is used:

Syntax : **constraint** $type_1 = type_2$

The above example can be written specifying that parameter p0 has type ***'a***, then putting a type constraint upon variable ***'a***.

```
# class a_point (p0 : 'a) =
    object
      constraint 'a = #point
      val p = p0
      method to_string() = p#to_string()
    end ;;
class a_point :
  (#point as 'a) -> object val p : 'a method to_string : unit -> string end
```

Several type constraints can be given in a class declaration.

Warning An open type cannot appear as the type of a method.

This strong restriction exists because an open type contains an uninstantiated type variable coming from the rest of the type. Since one cannot have a free variable type in a type declaration, a method containing such a type is rejected by type inference.

```
# class b_point p0  =
    object
      inherit a_point p0
      method get = p
    end ;;
Characters 6-77:
Some type variables are unbound in this type:
  class b_point :
    (#point as 'a) ->
    object val p : 'a method get : 'a method to_string : unit -> string end
The method get has type #point where .. is unbound
```

In fact, due to the constraint "**constraint** *'a = #point*", the type of get is the open type ***#point***. The latter contains a free variable type noted by a double point (..), which is not allowed.

Inheritance and the Type of `self`

There exists an exception to the prohibition of a type variable in the type of methods: a variable may stand for the type of the object itself (`self`). Consider a method testing the equality between two points.

```
# class point_eq (x,y) =
    object (self : 'a)
      inherit point (x,y)
      method eq (p:'a) = (self#get_x = p#get_x) && (self#get_y = p#get_y)
    end ;;
class point_eq :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

The type of method `eq` is *`'a -> bool`*, but the type variable stands for the type of the instance at construction time.

You can inherit from the class *`point_eq`* and redefine the method `eq`, whose type is still parameterized by the instance type.

```
# class colored_point_eq (xc,yc) c =
    object (self : 'a)
      inherit point_eq (xc,yc) as super
      val c = (c:string)
      method get_c = c
      method eq (pc : 'a) =  (self#get_x = pc#get_x) && (self#get_y = pc#get_y)
                          && (self#get_c = pc#get_c)
    end ;;
class colored_point_eq :
  int * int ->
  string ->
  object ('a)
    val c : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method eq : 'a -> bool
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
```

```
    method to_string : unit -> string
  end
```

The method `eq` from class *colored_point_eq* still has type *'a -> bool*; but now the variable *'a* stands for the type of an instance of class *colored_point_eq*. The definition of `eq` in class *colored_point_eq* masks the inherited one. Methods containing the type of the instance in their type are called binary methods. They will cause some limitations in the subtyping relation described in page 465.

Multiple Inheritance

With multiple inheritance, you can inherit data fields and methods from several classes. When there are identical names for fields or methods, only the last declaration is kept, according to the order of inheritance declarations. Nevertheless, it is possible to reference a method of one of the parent classes by associating different names with the inherited classes. This is not true for instance variables: if an inherited class masks an instance variable of a previously inherited class, the latter is no longer directly accessible. The various inherited classes do not need to have an inheritance relation. Multiple inheritance is of interest because it increases class reuse.

Let us define the abstract class *geometric_object*, which declares two virtual methods `compute_area` and `compute_peri` for computing the area and perimeter.

```
# class virtual geometric_object () =
  object
    method virtual compute_area : unit → float
    method virtual compute_peri : unit → float
  end;;
```

Then we redefine class `rectangle` as follows:

```
# class rectangle_1 ((p1,p2) :'a) =
  object
    constraint 'a = point * point
    inherit printable ()
    inherit geometric_object ()
    val llc = p1
    val urc = p2
    method to_string () =
     "["^llc#to_string()^","^urc#to_string()^"]"
    method compute_area() =
     float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
    method compute_peri() =
     float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
  end;;
class rectangle_1 :
  point * point ->
  object
    val llc : point
    val urc : point
```

```
    method compute_area : unit -> float
    method compute_peri : unit -> float
    method print : unit -> unit
    method to_string : unit -> string
  end
```

This implementation of classes respects the inheritance graph of figure 15.5.

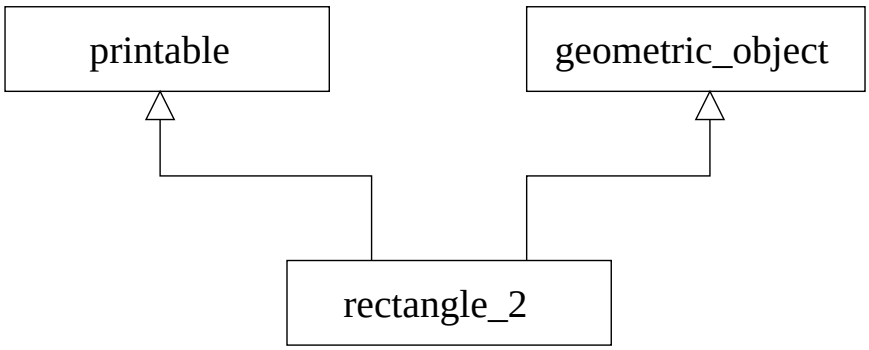


Figure 15.5: Multiple inheritance.

In order to avoid rewriting the methods of class `rectangle`, we may directly inherit from `rectangle`, as shown in figure 15.6.

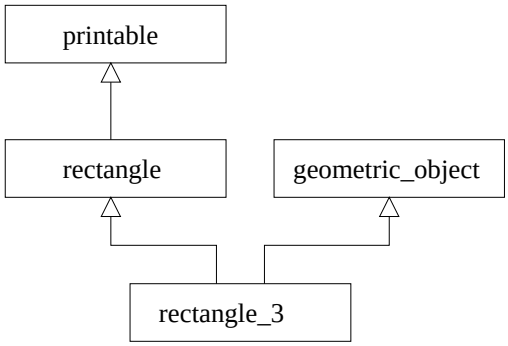


Figure 15.6: Multiple inheritance (continued).

In such a case, only the abstract methods of the abstract class `geometric_object` must be defined in `rectangle_2`.

```
# class rectangle_2 (p2 :'a) =
    object
    constraint 'a = point * point
    inherit rectangle p2
    inherit geometric_object ()
    method compute_area() =
```

```
    float ( abs(urc#get_x - llc#get_x) * abs(urc#get_y - llc#get_y))
  method compute_peri() =
    float ( (abs(urc#get_x - llc#get_x) + abs(urc#get_y - llc#get_y)) * 2)
end;;
```

Continuing in the same vein, the hierarchies `printable` and `geometric_object` could have been defined separately, until it became useful to have a class with both behaviors. Figure 15.7 displays the relations defined in this way.

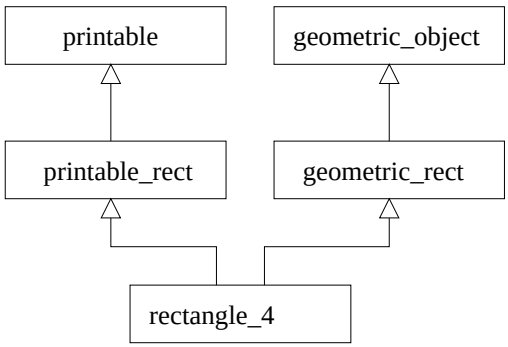


Figure 15.7: Multiple inheritance (end).

If we assume that classes `printable_rect` and `geometric_rect` define instance variables for the corners of a rectangle, we get class `rectangle_3` with four points (two per corner).

```
class rectangle_3 (p1,p2) =
 inherit printable_rect (p1,p2) as super_print
 inherit geometric_rect (p1,p2) as super_geo
end;;
```

In the case where methods of the same type exist in both classes `..._rect`, then only the last one is visible. However, by naming parent classes (`super_...`), it is always possible to invoke a method from either parent.

Multiple inheritance allows factoring of the code by integrating methods already written from various parent classes to build new entities. The price paid is the size of constructed objects, which are bigger than necessary due to duplicated fields, or inherited fields useless for a given application. Furthermore, when there is duplication (as in our last example), communication between these fields must be established manually (update, etc.). In the last example for class `rectangle_3`, we obtain instance variables of classes `printable_rect` and `geometric_rect`. If one of these classes has a method which modifies these variables (such as a scaling factor), then it is necessary to propagate these modifications to variables inherited from the other class. Such a heavy communication between inherited instance variables often signals a poor modeling of the given problem.

Parameterized Classes

Parameterized classes let Objective Caml's parameterized polymorphism be used in classes. As with the type declarations of Objective Caml, class declarations can be parameterized with type variables. This provides new opportunities for genericity and code reuse. Parameterized classes are integrated with ML-like typing when type inference produces parameterized types.

The syntax differs slightly from the declaration of parameterized types; type parameters are between brackets.

Syntax : **class [***'a, 'b, ...***]** *name* **= object** ... **end**

The Objective Caml type is noted as usual: *('a,'b,...) name*.

For instance, if a class `pair` is required, a naive solution would be to set:

```
# class pair x0 y0 =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
Characters 6-106:
Some type variables are unbound in this type:
  class pair :
    'a ->
    'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
The method fst has type 'a where 'a is unbound
```

One again gets the typing error mentioned when class *a_point* was defined (page 452). The error message says that type variable *'a*, assigned to parameter `x0` (and therefore to `x` and `fst`), is not bound.

As in the case of parameterized types, it is necessary to parameterize class *pair* with two type variables, and force the type of construction parameters `x0` and `y0` to obtain a correct typing:

```
# class ['a,'b] pair (x0:'a) (y0:'b) =
    object
      val x = x0
      val y = y0
      method fst = x
      method snd = y
    end ;;
class ['a, 'b] pair :
  'a ->
  'b -> object val x : 'a val y : 'b method fst : 'a method snd : 'b end
```

Type inference displays a class interface parameterized by variables of type `'a` and `'b`.

When a value of a parameterized class is constructed, type parameters are instantiated with the types of the construction parameters:

```
# let p = new pair 2 'X';;
val p : (int, char) pair = <obj>
# p#fst;;
- : int = 2
# let q = new pair 3.12 true;;
val q : (float, bool) pair = <obj>
# q#snd;;
- : bool = true
```

Note

In class declarations, type parameters are shown between brackets, but in types, they are shown between parentheses.

Inheritance of Parameterized Classes

When inheriting from a parameterized class, one has to indicate the parameters of the class. Let us define a class *acc_pair* that inherits from *('a,'b) pair*; we add two methods for accessing the fields, `get1` and `get2`,

```
# class ['a,'b] acc_pair (x0 : 'a) (y0 : 'b) =
    object
      inherit ['a,'b] pair x0 y0
      method get1 z = if x = z then y else raise Not_found
      method get2 z = if y = z then x else raise Not_found
    end;;
class ['a, 'b] acc_pair :
  'a ->
  'b ->
  object
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method snd : 'b
  end
# let p = new acc_pair 3 true;;
val p : (int, bool) acc_pair = <obj>
# p#get1 3;;
- : bool = true
```

We can make the type parameters of the inherited parameterized class more precise, e.g. for a pair of points.

```
# class point_pair (p1,p2) =
    object
      inherit [point,point] pair p1 p2
    end;;
class point_pair :
  point * point ->
  object
```

```
    val x : point
    val y : point
    method fst : point
    method snd : point
  end
```

Class *point_pair* no longer needs type parameters, since parameters *'a* and *'b* are completely determined.

To build pairs of displayable objects (i.e. having a method `print`), we reuse the abstract class `printable` (see page 451), then we define the class `printable_pair` which inherits from `pair`.

```
# class printable_pair x0 y0 =
    object
      inherit [printable, printable] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
```

This implementation allows us to construct pairs of instances of `printable`, but it cannot be used for objects of another class with a method `print`.

We could try to open type `printable` used as a type parameter for `acc_pair`:

```
# class printable_pair (x0 ) (y0 ) =
    object
      inherit [ #printable, #printable ] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
Characters 6-149:
Some type variables are unbound in this type:
  class printable_pair :
    (#printable as 'a) ->
    (#printable as 'b) ->
    object
      val x : 'a
      val y : 'b
      method fst : 'a
      method get1 : 'a -> 'b
      method get2 : 'b -> 'a
      method print : unit -> unit
      method snd : 'b
    end
The method fst has type #printable where .. is unbound
```

This first attempt fails because methods `fst` and `snd` contain an open type.

So we shall keep the type parameters of the class, while constraining them to the open type *#printable*.

```
# class ['a,'b] printable_pair (x0 ) (y0 ) =
    object
      constraint 'a = #printable
      constraint 'b = #printable
```

```
      inherit ['a,'b] acc_pair x0 y0
      method print () = x#print(); y#print ()
    end;;
class ['a, 'b] printable_pair :
  'a ->
  'b ->
  object
    constraint 'a = #printable
    constraint 'b = #printable
    val x : 'a
    val y : 'b
    method fst : 'a
    method get1 : 'a -> 'b
    method get2 : 'b -> 'a
    method print : unit -> unit
    method snd : 'b
  end
```

Then we construct a displayable pair containing a point and a colored point.

```
# let pp = new printable_pair
             (new point (1,2)) (new colored_point (3,4) "green");;
val pp : (point, colored_point) printable_pair = <obj>
# pp#print();;
( 1, 2)( 3, 4) with color green- : unit = ()
```

Parameterized Classes and Typing

From the point of view of types, a parameterized class is a parameterized type. A value of such a type can contain weak type variables.

```
# let r = new pair [] [];;
val r : ('_a list, '_b list) pair = <obj>
# r#fst;;
- : '_a list = []
# r#fst = [1;2];;
- : bool = false
# r;;
- : (int list, '_a list) pair = <obj>
```

A parameterized class can also be viewed as a closed object type; therefore nothing prevents us from also using it as an open type with the sharp notation.

```
# let compare_nothing ( x : ('a, 'a) #pair) =
    if x#fst = x#fst then x#mess else x#mess2;;
val compare_nothing :
  < fst : 'a; mess : 'b; mess2 : 'b; snd : 'a; .. > -> 'b = <fun>
```

This lets us construct parameterized types that contain weak type variables that are also open object types.

```
# let prettytype x ( y : ('a, 'a) #pair) = if x = y#fst then y else y;;
val prettytype : 'a -> (('a, 'a) #pair as 'b) -> 'b = <fun>
```

If this function is applied to one parameter, we get a closure, whose type variables are weak. An open type, such as ***#pair***, still contains uninstantiated parts, represented by the double point (..). In this respect, an open type is a partially known type parameter. Upon weakening such a type after a partial application, the displayer specifies that the type variable representing this open type has been weakened. Then the notation is *_#pair*.

```
# let g = prettytype 3;;
val g : ((int, int) _#pair as 'a) -> 'a = <fun>
```

Now, if function g is applied to a pair, its weak type is modified.

```
# g (new acc_pair 2 3);;
- : (int, int) acc_pair = <obj>
# g;;
- : (int, int) acc_pair -> (int, int) acc_pair = <fun>
```

Then we can no longer use g on simple pairs.

```
# g (new pair 1 1);;
Characters 4-16:
This expression has type (int, int) pair = < fst : int; snd : int >
but is here used with type
  (int, int) acc_pair =
    < fst : int; get1 : int -> int; get2 : int -> int; snd : int >
Only the second object type has a method get1
```

Finally, since parameters of the parameterized class can also get weakened, we obtain the following example.

```
# let h = prettytype [];;
val h : (('_b list, '_b list) _#pair as 'a) -> 'a = <fun>
# let h2 = h (new pair [] [1;2]);;
val h2 : (int list, int list) pair = <obj>
# h;;
- : (int list, int list) pair -> (int list, int list) pair = <fun>
```

The type of the parameter of h is no longer open. The following application cannot be typed because the argument is not of type ***pair***.

```
# h (new acc_pair [] [4;5]);;
Characters 4-25:
This expression has type
  ('a list, int list) acc_pair =
    < fst : 'a list; get1 : 'a list -> int list; get2 : int list -> 'a list;
      snd : int list >
but is here used with type
```

```
  (int list, int list) pair = < fst : int list; snd : int list >
Only the first object type has a method get1
```

Note

Parameterized classes of Objective Caml are absolutely necessary as soon as one has methods whose type includes a type variable different from the type of `self`.

Subtyping and Inclusion Polymorphism

Subtyping makes it possible for an object of some type to be considered and used as an object of another type. An object type *ot2* could be a subtype of *ot1* if:

1. it includes all of the methods of *ot1*,
2. each method of *ot2* that is a method of *ot1* is a subtype of the *ot1* method.

The subtype relation is only meaningful between objects: it can only be expressed between objects. Furthermore, the subtype relation must always be explicit. It is possible to indicate either that a type is a subtype of another, or that an object has to be considered as an object of a super type.

Syntax :

```
(name : sub_type :> super_type)
(name :> super_type)
```

Example

Thus we can indicate that an instance of `colored_point` can be used as an instance of `point`:

```
# let pc = new colored_point (4,5) "white";;
val pc : colored_point = <obj>
# let p1 = (pc : colored_point :> point);;
val p1 : point = <obj>
# let p2 = (pc :> point);;
val p2 : point = <obj>
```

Although known as a point, `p1` is nevertheless a colored point, and sending the method `to_string` will trigger the method relevant for colored points:

```
# p1#to_string();;
- : string = "( 4, 5) with color white"
```

This way, it is possible to build lists containing both points and colored points:

```
# let l = [new point (1,2) ; p1] ;;
val l : point list = [<obj>; <obj>]
# List.iter (fun x → x#print(); print_newline()) l;;
( 1, 2)
```

```
( 4, 5) with color white
- : unit = ()
```

Of course, the actions that can be performed on the objects of such a list are restricted to those allowed for points.

```
# p1#get_color () ;;
Characters 1-3:
This expression has type point
It has no method get_color
```

The combination of delayed binding and subtyping provides a new form of polymorphism: *inclusion polymorphism*. This is the ability to handle values of any type having a subtype relation with the expected type. Although static typing information guarantees that sending a message will always find the corresponding method, the behavior of the method depends on the actual receiving object.

Subtyping is not Inheritance

Unlike mainstream object-oriented languages such as C++, Java, and SmallTalk, subtyping and inheritance are different concepts in Objective Caml. There are two main reasons for this.

1. Instances of the class `c2` may have a type that is a subtype of the object type `c1` even if the class `c2` does not inherit from the class `c1`. Indeed, the class `colored_point` can be defined independently from the class `point`, provided the type of its instances are constrained to the object type `point`.
2. Class `c2` may inherit from the class `c1` but have instances whose type is not a subtype of the object type `c1`. This is illustrated in the following example, which uses the ability to define an abstract method that takes an as yet undetermined instance as an argument of the class being defined. In our example, this is method `eq` of class `equal`.

```
# class virtual equal () =
    object(self:'a)
     method virtual eq : 'a → bool
    end;;
class virtual equal : unit -> object ('a) method virtual eq : 'a -> bool end
# class c1 (x0:int) =
    object(self)
     inherit equal ()
     val x = x0
     method get_x = x
     method eq o = (self#get_x = o#get_x)
    end;;
class c1 :
  int ->
  object ('a) val x : int method eq : 'a -> bool method get_x : int end
```

```
# class c2 (x0:int) (y0:int) =
    object(self)
     inherit equal ()
     inherit c1 x0
     val y = y0
     method get_y = y
     method eq o = (self#get_x = o#get_x) && (self#get_y = o#get_y)
    end;;
class c2 :
  int ->
  int ->
  object ('a)
    val x : int
    val y : int
    method eq : 'a -> bool
    method get_x : int
    method get_y : int
  end
```

We cannot force the type of an instance of `c2` to be the type of instances of `c1`:

```
# let a = ((new c2 0 0) :> c1) ;;
Characters 11-21:
This expression cannot be coerced to type
  c1 = < eq : c1 -> bool; get_x : int >;
it has type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
but is here used with type < eq : c1 -> bool; get_x : int; get_y : int >
Type c2 = < eq : c2 -> bool; get_x : int; get_y : int >
is not compatible with type c1 = < eq : c1 -> bool; get_x : int >
Only the first object type has a method get_y
```

Types *c1* and *c2* are incompatible because the type of `eq` in `c2` is not a subtype of the type of `eq` in `c1`. To see why this is true, let `o1` be an instance of *c1*. If `o21` were an instance of *c2* subtyped to *c1*, then since `o21` and `o1` would both be of type *c1* the type of `eq` in *c2* would be a subtype of the type of `eq` in *c1* and the expression `o21#eq(o1)` would be correctly typed. But at run-time, since `o21` is an instance of class *c2*, the method `eq` of *c2* would be triggered. But this method would try to send the message `get_y` to `o1`, which does not have such a method; our type system would have failed!

For our type system to fulfill its role, the subtyping relation must be defined less naïvely. We do this in the next paragraph.

Formalization

Subtyping between objects. Let $t =< m_1 : \tau_1; \ldots\ m_n : \tau_n >$ and $t' =< m_1 : \sigma_1\ ; \ldots\ ; m_n : \sigma_n; m_{n+1} : \sigma_{n+1}; etc \ldots >$ we shall say that t' is a subtype of t, denoted by $t' \leq t$, if and only if $\sigma_i \leq \tau_i$ for $i \in \{1, \ldots, n\}$.

Function call. If $f : t \to s$, and if $a : t'$ and $t' \leq t$ then (fa) is well typed, and has type s.

Intuitively, a function f expecting an argument of type t may safely receive 'an argument of a subtype t' of t.

Subtyping of functional types. Type $t' \to s'$ is a subtype of $t \to s$, denoted by $t' \to s' \leq t \to s$, if and only if

$$s' \leq s \text{ and } t \leq t'$$

The relation $s' \leq s$ is called *covariance*, and the relation $t \leq t'$ is called *contravariance*. Although surprising at first, this relation between functional types can easily be justified in the context of object-oriented programs with dynamic binding.

Let us assume that two classes $c1$ and $c2$ both have a method m. Method m has type $t_1 \to s_1$ in $c1$, and type $t_2 \to s_2$ in $c2$. For the sake of readability, let us denote by $m_{(1)}$ the method m of $c1$ and $m_{(2)}$ that of $c2$. Finally, let us assume $c2 \leq c1$, i.e. $t_2 \to s_2 \leq t_1 \to s_1$, and let us look at a simple example of the covariance and contravariance relations.

Let $g : s_1 \to \alpha$, and $h\ (o : c1)\ (x : t_1)\ =\ g(o\#m(x))$

Covariance: function h expects an object of type $c1$ as its first argument. Since $c2 \leq c1$, it is legal to pass it an object of type $c2$. Then the method invoked by $o\#m(x)$ is $m_{(2)}$, which returns a value of type s_2. Since this value is passed to g which expects an argument of type s_1, clearly we must have $s_2 \leq s_1$.

Contravariance: for its second argument, h requires a value of type t_1. If, as above, we give h a first argument of type $c2$, then method $m_{(2)}$ is invoked. Since it expects an argument of type t_2, $t_1 \leq t_2$.

Inclusion Polymorphism

By "polymorphism" we mean the ability to apply a function to arguments of any "shape" (type), or to send a message to objects of various shapes.

In the context of the functional/imperative kernel of the language, we have already seen parameterized polymorphism, which enables you to apply a function to arguments of arbitrary type. The polymorphic parameters of the function have types containing type variables. A polymorphic function will execute the same code for various types of parameters. To this end, it will not depend on the structure of these arguments.

The subtyping relation, used in conjunction with delayed binding, introduces a new kind of polymorphism for methods: inclusion polymorphism. It lets the same message be sent to instances of different types, provided they have been constrained to the same subtype. Let us construct a list of points where some of them are in fact colored points treated as points. Sending the same message to all of them triggers the execution

of different methods, depending on the class of the receiving instance. This is called inclusion polymorphism because it allows messages from class `c`, to be sent to any instance of class `sc` that is a subtype of `c` (*`sc :> c`*) that has been constrained to `c`. Thus we obtain a polymorphic message passing for all classes of the tree of subtypes of `c`. Contrary to parameterized polymorphism, the code which is executed may be different for these instances.

Thanks to parameterized classes, both forms of polymorphism can be used together.

Equality between Objects

Now we can explain the somewhat surprising behavior of structural equality between objects which was presented on page 441. Two objects are structurally equal when they are physically the same.

```
# let p1 =  new point (1,2);;
val p1 : point = <obj>
# p1  = new point (1,2);;
- : bool = false
# p1 = p1;;
- : bool = true
```

This comes from the subtyping relation. Indeed we can try to compare an instance `o2` of a class *`sc`* that is a subtype of `c`, constrained to `c`, with an instance of `o1` from class `c`. If the fields which are common to these two instances are equal, then these objects might be considered as equal. This is wrong from a structural point of view because `o2` could have additional fields. Therefore Objective Caml considers that two objects are structurally different when they are physically different.

```
# let pc1 = new colored_point (1,2) "red";;
val pc1 : colored_point = <obj>
# let q = (pc1 :> point);;
val q : point = <obj>
# p1 = q;;
- : bool = false
```

This restrictive view of equality guarantees that an answer `true` is not wrong, but an answer `false` guarantees nothing.

Functional Style

Object-oriented programming usually has an imperative style. A message is sent to an object that physically modifies its internal state (i.e. its data fields). It is also possible to use a functional approach to object-oriented programming: sending a message returns a new object.

Object Copy

Objective Caml provides a special syntactic construct for returning a copy of an object `self` with some of the fields modified.

Syntax : {**<** *name*$_1$**=***expr*$_1$**;**...; *name*$_n$**=***expr*$_n$ **>**}

This way we can define functional points where methods for relative moves have no side effect, but instead return a new point.

```
# class f_point p =
    object
     inherit point p
     method f_rmoveto_x (dx) = {< x = x + dx >}
     method f_rmoveto_y (dy) = {< y = y + dy >}
    end ;;
class f_point :
  int * int ->
  object ('a)
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

With the new methods, movement no longer modifies the receiving object; instead a new object is returned that reflects the movement.

```
# let p = new f_point (1,1) ;;
val p : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# let q = p#f_rmoveto_x 2 ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 3, 1)- : unit = ()
```

Since these methods construct an object, it is possible to send a message directly to the result of the method `f_rmoveto_x`.

```
# print_string ((p#f_rmoveto_x 3)#to_string()) ;;
( 4, 1)- : unit = ()
```

The result type of the methods `f_rmoveto_x` and `f_rmoveto_y` is the type of the instance of the defined class, as shown by the *'a* in the type of `f_rmoveto_x`.

```
# class f_colored_point (xc, yc) (c:string) =
    object
      inherit f_point(xc, yc)
      val color = c
      method get_c = color
    end ;;
class f_colored_point :
  int * int ->
  string ->
  object ('a)
    val color : string
    val mutable x : int
    val mutable y : int
    method distance : unit -> float
    method f_rmoveto_x : int -> 'a
    method f_rmoveto_y : int -> 'a
    method get_c : string
    method get_x : int
    method get_y : int
    method moveto : int * int -> unit
    method print : unit -> unit
    method rmoveto : int * int -> unit
    method to_string : unit -> string
  end
```

Sending `f_rmoveto_x` to an instance of `f_colored_point` returns a new instance of `f_colored_point`.

```
# let fpc = new f_colored_point (2,3) "blue" ;;
val fpc : f_colored_point = <obj>
# let fpc2 = fpc#f_rmoveto_x 4 ;;
val fpc2 : f_colored_point = <obj>
# fpc2#get_c;;
- : string = "blue"
```

One can also obtain a copy of an arbitrary object, using the the primitive `copy` from module `Oo`:

```
# Oo.copy ;;
- : (< .. > as 'a) -> 'a = <fun>
# let q = Oo.copy p ;;
val q : f_point = <obj>
# print_string (p#to_string()) ;;
( 1, 1)- : unit = ()
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
# p#moveto(4,5) ;;
- : unit = ()
# print_string (p#to_string()) ;;
( 4, 5)- : unit = ()
```

```
# print_string (q#to_string()) ;;
( 1, 1)- : unit = ()
```

Example: a Class for Lists

A functional method may use the object itself, `self`, to compute the value to be returned. Let us illustrate this point by defining a simple hierarchy of classes for representing lists of integers.

First we define the abstract class, parameterized by the type of list elements.

```
# class virtual ['a] o_list () =
  object
    method virtual empty : unit → bool
    method virtual cons  : 'a → 'a o_list
    method virtual head :  'a
    method virtual tail : 'a o_list
  end;;
```

We define the class of non empty lists.

```
# class  ['a] o_cons (n ,l)  =
  object (self)
    inherit ['a] o_list ()
    val car = n
    val cdr = l
    method empty () = false
    method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
    method head = car
    method tail = cdr
  end;;
class ['a] o_cons :
  'a * 'a o_list ->
  object
    val car : 'a
    val cdr : 'a o_list
    method cons : 'a -> 'a o_list
    method empty : unit -> bool
    method head : 'a
    method tail : 'a o_list
  end
```

We should note that method `cons` returns a new instance of *`'a o_cons`*. To this effect, the type of `self` is constrained to *`'a #o_list`*, then subtyped to *`'a o_list`*. Without subtyping, we would obtain an open type (*`'a #o_list`*), which appears in the type of the methods, and is strictly forbidden (see page 456). Without the additional constraint, the type of `self` could not be a subtype of *`'a o_list`*.

This way we obtain the expected type for method `cons`. So now we know the trick and we define the class of empty lists.

```
# exception EmptyList ;;
# class ['a] o_nil () =
    object(self)
     inherit ['a] o_list ()
     method empty () = true
     method cons x = new o_cons (x, (self : 'a #o_list :> 'a o_list))
     method head = raise EmptyList
     method tail = raise EmptyList
    end ;;
```

First of all we build an instance of the empty list, and then a list of integers.

```
# let i = new o_nil ();;
val i : '_a o_nil = <obj>
# let l = new o_cons (3,i);;
val l : int o_list = <obj>
# l#head;;
- : int = 3
# l#tail#empty();;
- : bool = true
```

The last expression sends the message `tail` to the list containing the integer 3, which triggers the method `tail` from the class *'a o_cons*. The message `empty()`, which returns `true`, is sent to this result. You can see that the method which has been executed is `empty` from the class *'a o_nil*.

Other Aspects of the Object Extension

In this section we describe the declaration of "object" types and local declarations in classes. The latter can be used for class variables by making constructors that reference the local environment.

Interfaces

Class interfaces are generally infered by the type system, but they can also be defined by a type declaration. Only public methods appear in this type.

Syntax :

```
class type name =
  object
    ⋮
    val name_i : type_i
    ⋮
    method name_j : type_j
    ⋮
  end
```

Thus we can define the class `point` interface:

```
# class type interf_point  =
  object
    method get_x : int
    method get_y : int
    method moveto : (int * int ) → unit
    method rmoveto : (int * int )  → unit
    method to_string : unit → string
    method distance : unit → float
  end ;;
```

This declaration is useful because the defined type can be used as a type constraint.

```
# let seg_length (p1:interf_point) (p2:interf_point) =
   let x = float_of_int (p2#get_x - p1#get_x)
   and y = float_of_int (p2#get_y - p1#get_y) in
    sqrt ((x*.x) +. (y*.y)) ;;
val seg_length : interf_point -> interf_point -> float = <fun>
```

Interfaces can only mask fields of instance variables and private methods. They cannot mask abstract or public methods.

This is a restriction in their use, as shown by the following example:

```
# let p = ( new point_m1 (2,3) : interf_point);;
Characters 11-29:
This expression has type
  point_m1 =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string; undo : unit -> unit >
but is here used with type
  interf_point =
    < distance : unit -> float; get_x : int; get_y : int;
      moveto : int * int -> unit; rmoveto : int * int -> unit;
      to_string : unit -> string >
Only the first object type has a method undo
```

Nevertheless, interfaces may use inheritance. Interfaces are especially useful in combination with modules: it is possible to build the signature of a module using object types, while only making available the description of class interfaces.

Local Declarations in Classes

A class declaration produces a type and a constructor. In order to make this chapter easier to read, we have been presenting constructors as functions without an environment. In fact, it is possible to define constructors which do not need initial values to create an instance: that means that they are no longer functional. Furthermore one

can use local declarations in the class. Local variables captured by the constructor are shared and can be treated as class variables.

Constant Constructors

A class declaration does not need to use initial values passed to the constructor. For example, in the following class:

```
# class example1 =
    object
      method print () = ()
    end ;;
class example1 : object method print : unit -> unit end
# let p = new example1 ;;
val p : example1 = <obj>
```

The instance constructor is constant. The allocation does not require an initial value for the instance variables. As a rule, it is better to use an initial value such as `()`, in order to preserve the functional nature of the constructor.

Local Declarations for Constructors

A local declaration can be written directly with abstraction.

```
# class example2 =
    fun a →
      object
        val mutable r = a
        method get_r = r
        method plus x = r <- r + x
     end;;
class example2 :
  int ->
  object val mutable r : int method get_r : int method plus : int -> unit end
```

Here it is easier to see the functional nature of the constructor. The constructor is a closure which may have an environment that binds free variables to an environment of declarations. The syntax for class declarations allows local declarations in this functional expression.

Class Variables

Class variables are declarations which are known at class level and therefore shared by all instances of the class. Usually these class variables can be used outside of any instance creation. In Objective Caml, thanks to the functional nature of a constructor with a non-empty environment, we can make these values (particularly the modifiable ones) shared by all instances of a class.

We illustrate this facility with the following example, which allows us to keep a register of the number of instances of a class. To do this we define a parameterized abstract class *'a om*.

```
# class virtual  ['a] om =
    object
      method  finalize ()  = ()
      method virtual destroy : unit → unit
      method virtual to_string : unit → string
      method virtual all : 'a list
    end;;
```

Then we declare class `'a lo`, whose constructor contains local declarations for `n`, which associates a unique number with each instance, and for `l`, which contains the list of pairs (number, instance) of still active instances.

```
# class ['a] lo =
    let l = ref []
    and n = ref 0 in
      fun s →
        object(self: 'b )
          inherit ['a] om
          val mutable num = 0
          val name = s
          method to_string () = s
          method print () = print_string s
          method print_all () =
            List.iter (function (a,b) →
                         Printf.printf "(%d,%s) " a (b#to_string())) !l
          method destroy () = self#finalize();
                              l:= List.filter (function (a,b) → a <> num) !l; ()
          method all = List.map snd !l
          initializer incr n; num <- !n;  l:= (num, (self :> 'a  om) ) :: !l ; ()
        end;;
class ['a] lo :
  string ->
  object
    constraint 'a = 'a om
    val name : string
    val mutable num : int
    method all : 'a list
    method destroy : unit -> unit
    method finalize : unit -> unit
    method print : unit -> unit
    method print_all : unit -> unit
    method to_string : unit -> string
  end
```

At each creation of an instance of class *lo*, the initializer increments the reference `n` and adds the pair (number, `self`) to the list `l`. Methods `print` and `print_all` display respectively the receiving instance and all the instances containing in `l`.

```
# let m1 = new lo "start";;
val m1 : ('a om as 'a) lo = <obj>
```

```
# let m2 = new lo "between";;
val m2 : ('a om as 'a) lo = <obj>
# let m3 = new lo "end";;
val m3 : ('a om as 'a) lo = <obj>
# m2#print_all();;
(3,end) (2,between) (1,start) - : unit = ()
# m2#all;;
- : ('a om as 'a) list = [<obj>; <obj>; <obj>]
```

Method `destroy` removes an instance from the list of instances, and calls method `finalize` to perform a last action on this instance before it disappears from the list. Method `all` returns all the instances of a class created with **`new`**.

```
# m2#destroy();;
- : unit = ()
# m1#print_all();;
(3,end) (1,start) - : unit = ()
# m3#all;;
- : ('a om as 'a) list = [<obj>; <obj>]
```

We should note that instances of subclasses are also kept in this list. Nothing prevents you from using the same technique by specializing some of these subclasses. On the other hand, the instances obtained by a copy (`Oo.copy` or {**`< >`**}) are not tracked.

Exercises

Stacks as Objects

Let us reconsider the stacks example, this time in object oriented style.

1. Define a class `intstack` using Objective Caml's lists, implementing methods `push`, `pop`, `top` and `size`.
2. Create an instance containing 3 and 4 as stack elements.
3. Define a new class `stack` containing elements answering the method `print : unit -> unit`.
4. Define a parameterized class `['a] stack`, using the same methods.
5. Compare the different classes of stacks.

Delayed Binding

This exercise illustrates how delayed binding can be used in a setting other than subtyping.

Given the program below:

1. Draw the relations between classes.
2. Draw the different messages.
3. Assuming you are in character mode without echo, what does the program display?

```
exception CrLf;;
class chain_read (m) =
  object (self)
  val msg = m
  val mutable res = ""

  method char_read =
    let c = input_char stdin in
      if (c != '\n') then begin
        output_char stdout c; flush stdout
      end;
      String.make 1 c

  method  private chain_read_aux =
      while true do
        let s = self#char_read in
          if s = "\n" then raise CrLf
          else res <- res ^ s;
      done

  method private chain_read_aux2 =
    let s = self#lire_char in
          if s = "\n" then raise CrLf
          else begin res <- res ^ s; self#chain_read_aux2 end

  method chain_read =
    try
      self#chain_read_aux
    with End_of_file  -> ()
      | CrLf -> ()

  method input = res <- ""; print_string msg; flush stdout;
                 self#chain_read

  method get = res
end;;

class mdp_read (m) =
  object (self)
  inherit chain_read m
  method char_read = let c  = input_char stdin in
                     if (c != '\n') then begin
                       output_char stdout '*'; flush stdout
                     end;
```

```
                    let s = " " in s.[0] <- c; s
end;;

let login = new chain_read("Login : ");;
let passwd = new mdp_read("Passwd : ");;
login#input;;
passwd#input;;
print_string (login#get);;print_newline();;
print_string (passwd#get);;print_newline();;
```

Abstract Classes and an Expression Evaluator

This exercise illustrates code factorization with abstract classes.

All constructed arithmetic expressions are instances of a subclass of the abstract class *expr_ar*.

1. Define an abstract class *expr_ar* for arithmetic expressions with two abstract methods: `eval` of type *float*, and `print` of type *unit*, which respectively evaluates and displays an arithmetic expression.
2. Define a concrete class *constant*, a subclass of *expr_ar*.
3. Define an abstract subclass *bin_op* of *expr_ar* implementing methods `eval` and `print` using two new abstract methods `oper`, of type *(float * float) -> float* (used by `eval`) and `symbol` of type *string* (used by `print`).
4. Define concrete classes *add* and *mul* as subclasses of *bin_op* that implement the methods `oper` and `symbol`.
5. Draw the inheritance tree.
6. Write a function that takes a sequence of `Genlex.token`, and constructs an object of type *expr_ar*.
7. Test this program by reading the standard input using the generic lexical analyzer `Genlex`. You can enter the expressions in post-fix form.

The Game of Life and Objects.

Define the following two classes:

- `cell` : for the cells of the world, with the method `isAlive : unit -> bool`
- `world` : with an array of `cell`, and the messages:
  ```
  display : unit -> unit
  nextGen : unit -> unit
  setCell : int * int -> cell -> unit
  getCell : int * int -> cell
  ```

1. Write the class `cell`.

2. Write an abstract class `absWorld` that implements the abstract methods `display`, `getCell` and `setCell`. Leave the method `nextGen` abstract.
3. Write the class `world`, a subclass of `absWorld`, that implements the method `nextGen` according to the growth rules.
4. Write the main program which creates an empty world, adds some cells, and then enters an interactive loop that iterates displaying the world, waiting for an interaction and computing the next generation.

Summary

This chapter described the object extension of the language Objective Caml. The class organization is an alternative to modules that, thanks to inheritance and delayed binding, allows object modeling of an application, as well as reusability and adaptability of programs. This extension is integrated with the type system of Objective Caml and adds the notion of subtype, which allows instances to be used as a subtype in any place where a value of this type is expected. By combining subtyping and delayed binding, we obtain inclusion polymorphism, which, for instance, allows us to build homogeneous lists from the point of view of types, albeit non-homogeneous with regard to behavior.

To Learn More

There are a huge number of publications on object-oriented programming. Each language implements a different model.

A general introduction (still topical for the first part) is "Langages à Objets " ([MNC+91]) which explains the object-oriented approach. A more specialized book, "Langages et modèles à objets" [DEMN98], gives the examples in this domain.

For modeling, the book "Design patterns" ([GHJV95]) gives a catalogue of design patterns that show how reusability is possible.

The reference site for the UML notation is Rational:

Link: http://www.rational.com/uml/resources

For functional languages with an object extension, we mention the "Lisp" objects, coming from the SMALLTALK world, and CLOS (meaning *Common Lisp Object System*), as well as a number of Scheme's implementing generic functions similar to those in CLOS.

Other proposals for object-oriented languages have been made for statically typed functional languages, such as Haskell, a pure functional language which has parameterized and *ad hoc* polymorphism for overloading.

The paper [RV98] presents the theoretical aspects of the object extension of Objective Caml.

To learn more on the static object typing in Objective Caml, you can look at several lectures available online.

Lectures by María-Virginia Aponte:

Link: http://tulipe.cnam.fr/personne/aponte/ocaml.html

A short presentation of objects by Didier Rémy:

Link: http://pauillac.inria.fr/~remy/objectdemo.html

Lectures by Didier Rémy at Magistère MMFAI:

Link: http://pauillac.inria.fr/~remy/classes/magistere/

Lectures by Roberto Di Cosmo at Magistère MMFAI:

Link: http://www.dmi.ens.fr/users/dicosmo/CourseNotes/OO/

16

Comparison of the Models of Organisation

Chapters 14 and 15 respectively presented two models of application organisation: The functional/modular model and the object model. These two models address, each in its own way, the needs of application development:

- logical organisation of a program: module or class;
- separate compilation: simple module;
- abstract data types: module (abstract type) or object;
- reuse of components: functors/sharing of types with parametric polymorphism or inheritance/subtyping with parameterized classes;
- modifiability of components: late binding (object).

The development of a modular application begins by dividing it into logical units: modules. This is followed by the actualization of their specification by writing their signature, and finally by implementation. During the implementation of a module, it may be necessary to modify its signature or that of its parameters; it is then necessary to modify their sources. This is unsatisfactory if the same module is already used by another application. Nevertheless, this process offers a strict and reassuring framework for the programmer.

In the object model, the analysis of a problem results in the description of the relations between classes. If, later on, a class does not provide the required functionality, it is always possible to extend it by subclassing. This process permits the reuse of large hierarchies of classes without modifying their sources, and thus not modifying the behavior of an application that uses them, either. Unfortunately, this technique leads to code bloat, and poses difficulties of duplication with multiple inheritance.

Many problems necessitate recursive data types and operations which manipulate values of these types. It often happens that the problem evolves, sometimes in the course of implementation, sometimes during maintenance, requiring an extension of the types and operations. Neither of these two models permits extension in both ways. In the

functional/modular model, types are not extensible, but one can create new functions (operations) on the types. In the object model, one can extend the objects, but not the methods (by creating a new subclass on an abstract class which implements its methods.) In this respect, the two models are duals.

The advantage of uniting these two models in the same language is to be able to choose the most appropriate model for the resolution of the problem in question, and to mix them in order to overcome the limitations of each model.

Plan of the Chapter

The first section compares the functional/modular model and the object model. This comparison brings out the particular features of each model, in order to show how many of them may be translated by hand into the other model. One can thus simulate inheritance with modules and use classes to implement simple modules. The limitations of each model are then reviewed. The second section is concerned with the problem of extensibility for data structures and methods, and proposes a solution which mixes the two models. The third section describes some other combinations of the two models by the use of abstract module types for objects.

Comparison of Modules and Objects

The main difference between modular programming and object programming in Objective Caml comes from the type system. In effect, programming with modules remains within the ML type system (*i.e.* parametric polymorphism code is executed for different types of parameter), while programming with objects entails an *ad hoc* polymorphism (in which the sending of a message to an object triggers the application of different pieces of code). This is particularly clear with subtyping. This extension of the ML type system can not be simulated in pure ML. It will always be impossible to construct heterogeneous lists without breaking the type system.

Modular programming and object programming are two safe (thanks to typing) approaches to the logical organisation of a program, permitting the reusability and the modifiability of software components. Programming with objects in Objective Caml allows parametric polymorphism (parameterized classes) and *inclusion/subtype polymorphism* (sending of messages) thanks to late binding and subtyping, with restrictions due to equality, facilitating incremental programming. Modular programming allows one to restrict parametric polymorphism and use immediate binding, which can be useful for conserving efficiency of execution.

The modular programming model permits the easy extension of functions on non-extensible recursive data types. If one wishes to add a case in a variant type, it will be necessary to modify a large part of the sources.
The object model of programming defines a set of recursive data types using classes. One interprets a class as a case of the data type.

Efficiency of Execution

Late binding corresponds to an indirection in the method table (see page 447). Just as the access to an instance variable from outside the class goes through a message dispatch, this accumulation of indirections can prove to be costly.

To show this loss of efficiency, we construct the following class hierarchy:

```
# class virtual test () =
    object
     method virtual sum : unit → int
     method virtual sum2 : unit → int
    end;;
# class a x  =
    object(self)
     inherit test ()
     val a = x
     method a = a
     method sum () = a
     method sum2 () = self#a
    end;;
# class b x y  =
    object(self)
    inherit a x as super
     val b = y
     method b = b
     method  sum () = b + a
     method  sum2 () = self#b + super#sum2()
    end;;
```

Now, we compare the execution time, on one hand of the dispatch of messages `sum` and `sum2` to an instance of class `b`, and on the other hand of a call to the following function `f`.

```
# let f a b = a + b ;;
# let iter g a n = for i = 1 to n do ignore(g a) done ; g a ;;
# let go i j = match i with
                  1 → iter (fun x → x#sum()) (new b 1 2) j
                | 2 → iter (fun x → x#sum2()) (new b 1 2) j
                | 3 → iter (fun x → f 1 x) 2 j ;;

# go (int_of_string (Sys.argv.(1))) (int_of_string (Sys.argv.(2))) ;;
```

For 10 million iterations, we get the following results:

	bytecode	native
case 1	07,5 s	0,6 s
case 2	15,0 s	2,3 s
case 3	06,0 s	0,3 s

This example has been constructed in order to show that late binding has a cost relative to the standard static binding. This cost depends on the quantity of calculation relative to the number of message dispatches in a function. The use of the native compiler reduces the calculation component without changing the indirection component of the test. We can see in case 2 that the multiple indirections at the dispatch of message `sum2` have an "incompressible" cost.

Example: Graphical Interface

The `AWI` graphical library (see page 377) was designed using the functional/imperative core of the language. It is very easy to adapt it into module form. Each component becomes an independent module, thus permitting a harmonization of function names. To add a component, it is necessary to know the concrete type of its components. It is up to the new module to modify the fields necessary to describe its appearance and its behaviors.

The library can also be rewritten as an object. For this we construct the hierarchy of classes shown in figure 16.1.

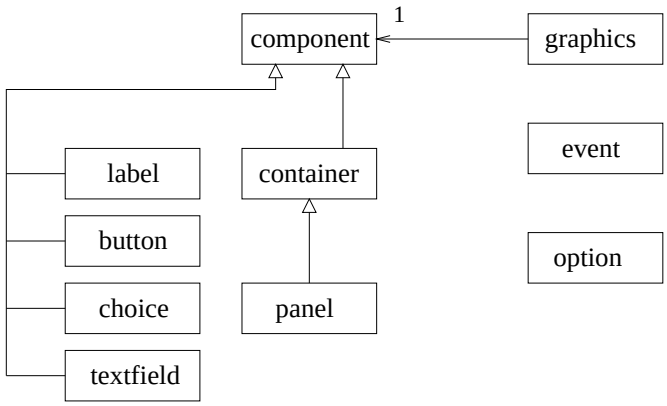


Figure 16.1: Class hierarchy for `AWI`.

It is easier to add new components, thanks to inheritance, than when using modules; however, the absence of overloading still requires options to be encoded as method parameters. The use of the subtyping relation makes it easy to construct a list of the constituents of a container. Deferred linking selects the methods appropriate to the component. The interest of the object model also comes from the possibility of extending or modifying the graphics context, and the other types that are used, again thanks to inheritance. This is why the principal graphics libraries are organised according to the object model.

Translation of Modules into Classes

A simple module which only declares one type and does not have any type-independent polymorphic functions can be translated into a class. According to the nature of the type used (record type or variant type) one translates the module into a class in a different way.

Type Declarations

Record type. A record type can be written directly in the form of a class in which every field of the record type becomes an instance variable.

Variant type. A variant type translates into many classes, using the conceptual model of a "composite". An abstract class describes the operations (functions) on this type. Every branch of the variant type thus becomes a subclass of the abstract class, and implements the abstract methods for its branch. We no longer have pattern matching but instead choose the method specific to the branch.

Parameterized types. Parameterized types are implemented by parameterized classes.

Abstract types. We can consider a class as an abstract type. At no time is the internal state of the class visible outside its hierarchy. Nevertheless, nothing prevents us from defining a subclass in order to access the variables of the instances of a class.

Mutually recursive types. The declarations of mutually recursive types are translated into declarations of mutually recursive classes.

Function Declarations

Those functions with parameters dependent on the module type, `t`, are translatable into methods. Functions in which `t` does not appear may be declared **`private`** inasmuch as their membership of the module is not directly linked to the type `t`. This has the added advantage that there is no problem if type variables appear in the type of the parameters. We are left with the problem of functions in which one parameter is of type `t` and another is of type `'a`. These functions are very rare in the modules of the standard library. We can identify "peculiar" modules like `Marshal` or `Printf` which have non-standard typing, and modules (that operate) on linear structures like `List`. For this last, the function `fold_left`, of type `('a -> 'b -> 'a) -> 'a -> 'b list -> 'a` is difficult to translate, especially in a method of the class `['b] list` because the type variable `'a` is free and may not appear in the type of the method. Rather than adding a type parameter to the `list` class, it is preferable to break these functions out into new classes, parameterized by two type variables and having a list field.

Binary methods. Binary methods do not pose any problem, outside subtyping.

Other declarations. Declarations of non-functional values. We can accept the declaration of non-functional values outside classes. This is also true for exceptions.

Example: Lists with Iterator. We are trying to translate a module with the following signature `LIST` into an object.

```
# module type LIST =  sig
    type 'a list = C0 | C1 of 'a * 'a list
    val add : 'a list → 'a → 'a list
    val length : 'a list → int
    val hd : 'a list → 'a
    val tl : 'a list → 'a list
    val append : 'a list → 'a list → 'a list
    val fold_left : ('a → 'b → 'a) → 'a → 'b list → 'a
  end ;;
```

First of all, we declare the abstract class *'a list* corresponding to the definition of the type.

```
# class virtual ['a] list () =
    object (self : 'b)
      method virtual add : 'a → 'a list
      method virtual empty : unit → bool
      method virtual hd : 'a
      method virtual tl : 'a list
      method virtual length : unit → int
      method virtual append : 'a list  → 'a list
    end ;;
```

Then we define the two subclasses `c1_list` and `c0_list` for each constituent of the variant type. Each of these classes should define the methods of the ancestor abstract class

```
# class ['a] c1_list (t, q)  =
    object (self )
      inherit ['a] list () as super
      val t = t
      val q = q
      method add x = new c1_list (x, (self : 'a #list :> 'a list))
      method empty () = false
      method length () = 1 + q#length()
      method hd = t
      method tl = q
      method append l  = new c1_list (t,q#append l)
    end ;;
# class ['a] c0_list () =
    object (self)
```

```
      inherit ['a] list () as super
      method add x = new c1_list (x, (self : 'a #list :> 'a list))
      method empty () = true
      method length () = 0
      method hd = failwith "c0_list : hd"
      method tl = failwith "c0_list : tl"
      method append l  = l
    end ;;
# let l = new c1_list (4, new c1_list (7, new c0_list ())) ;;
val l : int list = <obj>
```

The function `LIST.fold_left` has not been incorporated into the `list` class to avoid introducing a new type parameter. We prefer to define the class `fold_left` to implement this method. For this, we use a functional instance variable (`f`).

```
# class virtual ['a,'b] fold_left () =
    object(self)
      method virtual f : 'a → 'b → 'a
      method iter r (l : 'b list) =
        if l#empty() then r else self#iter (self#f r (l#hd)) (l#tl)
    end ;;
# class ['a,'b] gen_fl f =
    object
      inherit ['a,'b] fold_left ()
      method f = f
    end ;;
```

Thus we construct an instance of the class `gen_fl` for addition:

```
# let afl = new gen_fl (+) ;;
val afl : (int, int) gen_fl = <obj>
# afl#iter 0 l ;;
- : int = 11
```

Simulation of Inheritance with Modules

Thanks to the relation of inheritance between classes, we can retrieve in a subclass the collection of variable declarations and methods of the ancestor class. We can simulate this relation by using modules. The subclass which inherits is transformed into a parameterized module, of which the parameter is the ancestor class. Multiple inheritance increases the number of parameters of the module. We revisit the classic example of points and colored points, described in chapter 15, to translate it into modules.

The class `point` becomes the module `Point` with the following signature `POINT`.

```
# module type POINT =
    sig
      type point
      val new_point : (int * int) → point
      val get_x : point → int
```

```
    val get_y : point → int
    val moveto : point → (int * int) → unit
    val rmoveto : point → (int * int) → unit
    val display : point → unit
    val distance : point → float
  end ;;
```

The class `colored_point` is transformed into a parameterized module `ColoredPoint` which has the signature `POINT` as its parameter.

```
# module ColoredPoint = functor (P : POINT) →
  struct
    type colored_point = {p:P.point;c:string}
    let new_colored_point p c = {p=P.new_point p;c=c}
    let get_c self = self.c
  (* begin *)
    let get_x self = let super = self.p in P.get_x super
    let get_y self = let super = self.p in P.get_y super
    let moveto self = let super = self.p in P.moveto super
    let rmoveto self = let super = self.p in P.rmoveto super
    let display self = let super = self.p in P.display super
    let distance self = let super = self.p in P.distance super
  (* end *)
    let display self =
      let super = self.p in P.display super; print_string ("has color "^ self.c)
  end ;;
```

The burden of "inherited" declarations can be lightened by an automatic translation procedure, or an extension of the language. Recursive method declarations can be written with a single `let rec ... and`. Multiple inheritance leads to functors with many parameters. The cost of redefinition is not greater than that of late binding.

Late binding is not implemented in this simulation. To achieve it, it is necessary to define a record in which each field corresponds to the type of its functions/methods.

Limitations of each Model

The functional/modular module offers a reassuring but rigid framework for the modifiability of code. Objective Caml's object model suffers from "double vision" of classes: structuring and type, implying the absence of overloading and the impossibility of imposing type constraints from an ancestor type on a descendant type.

Modules

The principal limitations of the functional/modular model arise from the difficulty of extending types. Although abstract types allow us to get away from the concrete representation of a type, their use in parameterized modules requires that type equalities between modules be indicated by hand, complicating the writing of signatures.

Recursive dependencies. The dependence graph of the modules in an application is a directed acyclic graph (*DAG*). This implies on the one hand that there are no types that are mutually recursive between two modules, and on the other prevents the declaration of mutually recursive values.

Difficulties in writing signatures. One of the attractions of type inference is that it is not necessary to specify the types of function parameters. The specification of signatures sacrifices this convenience. It becomes necessary to specify the types of the declarations of the signature "by hand." One can use the `-i` option of the compiler `ocamlc` to display the type of all the global declarations in a `.ml` file and use this information to construct the signature of a module. In this case, we lose the "software engineering" discipline which consists of specifying the module before implementing it. In addition, if the signature and module undergo large changes, we will have to go back to editing the signature. Parameterized modules need signatures for their parameters and those should also be written by hand. Finally if we associate a functional signature with a parameterized module, it is impossible to recove the signature resulting from the application of the functor. This obliges us to mostly write non-functional signatures, leaving it until later to assemble them to construct a functional signature.

Import and export of modules. The importation of the declarations of a simple module is achieved either by dot notation (`Module.name`) or directory by the name of a declaration (`name`) if the model has been opened (**`open`** `Module`). The declaration of the interface of the imported module is not directly exportable at the level of the module in process of being defined. It has access to these declarations, but they are not considered as declarations of the module. In order to do this it is necessary to declare, in the same way as the simulation of inheritance, imported values. The same is true for parameterized modules. The declarations of the module parameters are not considered as declarations of the current module.

Objects

The principle limitations of the Objective Caml object model arise from typing.

- no methods containing parameters of free type;
- difficulty of escaping from the type of a class in one of its methods;
- absence of type constraint from the ancestor type on its descendant;
- no overloading;

The most disconcerting point when you start with the object extension of Objective Caml is the impossibility of constructing methods containing a parameterized type in which the type parameter is free. The declaration of a class can be seen as the definition of a new type, and hence arises the general rule forbidding the presence of variables with free type in the declaration of a type. For this reason, parameterized classes are indispensable in the Objective Caml object model because they permit the linking of their type variables.

Absence of overloading. The Objective Caml object model does not allow method overloading. As the type of an object corresponds to types of its methods, the fact of possessing many methods with the same name but different types would result in numerous ambiguities, due to parametric polymorphism, which the system could only resolve dynamically. This would be contradictory to the vision of totally static typing. We take a class `example` which has two `message` methods, the first having an integer parameter, and the second a float parameter. Let `e` be an instance of this class and `f` be the following function:

```
# let f x a = x#message a ;;
```

The calls `f e 1` et `f e 1.1` cannot be statically resolved because there is no information about the class `example` in the code of the function `f`.

An immediate consequence of this absence is the uniqueness of instance constructors. The declaration of a class indicates the parameters to supply to the creation function. This constructor is unique.

Initialization. The initialization of instance variables declared in a class can be problematic when it should be calculated based on the values passed to the constructor.

Equality between instances. The only equality which applies to objects is physical equality. Structural equality always returns `false` when it is applied to two physically different objects. This can be surprising inasmuch as two instances of the same class share the same method table. One can imagine a physical test on the method table and a structural test on the values (`val`) of objects. These are the implementation choices of the linear pattern-matching style.

Class hierarchy. There is no class hierarchy in the language distribution. In fact the collection of libraries are supplied in the form of simple or parameterized modules. This demonstrates that the object extension of the language is still stabilizing, and makes little case for its extensive use.

Extending Components

We call a collection of data and methods on the data a component. In the functional/modular model, a component consists of the definition of a type and some functions which manipulate the type. Similarly a component in the object model consists of a hierarchy of classes, inheriting from one (single) class and therefore having all of its behaviors. The problem of the extensibility of components consists of wanting on the one hand to extend the behaviors and on the other to extend the data operated on, and all this without modifying the initial program sources. For example a component `image` can be either a rectangle or a circle which one can draw or move.

	rectangle	circle	group
draw	X	X	
move	X	X	
grow			

We might wish to extend the `image` component with the method `grow` and create groups of images. The behavior of the two models differs depending on the direction of the extension: data or methods. First we define, in each model, the common part of the `image` component, and then we try to extend it.

In the Functional Model

We define the type *image* as a variant type which contains two cases. The methods take a parameter of type *image* and carry out the required action.

```
# type image = Rect of float | Circle of float ;;
# let draw = function Rect r → ... | Circle c → ... ;;
# let move = ... ;;
```

Afterwards, we could encapsulate these global declarations in a simple module.

Extension of Methods

The extension of the methods depends on the representation of the type *image* in the module. If this type is abstract, it is no longer possible to extend the methods. In the case where the type remains concrete, it is easy to add a `grow` function which changes the scale of an image by choosing a rectangle or a circle by pattern matching.

Extension of Data Types

The extension of data types cannot be achieved with the type *image*. In fact Objective Caml types are not extensible, except in the case of the type *exn* which represents exceptions. It is not possible to extend data while keeping the same type, therefore it is necessary to define a new type *n_image* in the following way:

```
type n_image = I of image | G of n_image * n_image;;
```

Thus we should redefine the methods for this new type, simulating a kind of inheritance. This becomes complex when there are many extensions.

In the Object Model

We define the classes `rectangle` and `circle`, subclasses of the abstract class `image` which has two abstract methods, `draw` and `move`.

```
# class virtual image () =
    object(self:'a)
      method virtual draw : unit → unit
      method virtual move : float * float → unit
    end;;
# class rectangle x y w h =
    object
      inherit image ()
      val mutable x = x
      val mutable y = y
      val mutable w = w
      val mutable h = h
      method draw () = Printf.printf "R: (%f,%f) [%f,%f]" x y w h
      method move (dx,dy) = x <- x +. dx; y <- y +. dy
    end;;
# class circle x y r =
    object
      val mutable x = x
      val mutable y = y
      val mutable r = r
      method draw () = Printf.printf "C: (%f,%f) [%f]" x y r
      method move (dx, dy)  = x <- x +. dx; y <- y +. dy
    end;;
```

The following program constructs a list of images and displays it.

```
# let r = new rectangle 1. 1. 3. 4.;;
val r : rectangle = <obj>
# let c = new circle 1. 1. 4.;;
val c : circle = <obj>
# let l =  [ (r :> image); (c :> image)];;
val l : image list = [<obj>; <obj>]
# List.iter (fun x → x#draw(); print_newline()) l;;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]
- : unit = ()
```

Extension of Data Types

The data are easily extended by adding a subclass of the class image in the following way.

```
# class group i1 i2 =
    object
      val i1 = (i1:#image)
      val i2 = (i2:#image)
      method draw () = i1#draw(); print_newline (); i2#draw()
      method move p  = i1#move p; i2#move p
    end;;
```

We notice now that the "type" *image* becomes recursive because the class *group* depends outside inheritance on the class *image*.

```
# let g = new group (r:>image) (c:>image);;
val g : group = <obj>
# g#draw();;
R: (1.000000,1.000000) [3.000000,4.000000]
C: (1.000000,1.000000) [4.000000]- : unit = ()
```

Extension of Methods

We define an abstract subclass of *image* which contains a new method.

```
# class virtual e_image () =
    object
      inherit image ()
      method virtual surface : unit → float
    end;;
```

We can define classes *e_rectangle* and *e_circle* which inherit from *e_image* and from *rectangle* and *circle* respectively. We can then work on extended image to use this new method. There is a remaining difficulty with the class *group*. This contains two fields of type *image*, so even when inheriting from the class *e_image* it will not be possible to send the *grow* message to the image fields. It is thus possible to extend the methods, except in the case of subclasses corresponding to recursive types.

Extension of Data and Methods

To implement extension in both ways, it is necessary to define recursive types in the for of a parameterized class. We redefine the class *group*.

```
# class ['a] group i1 i2 =
    object
      val i1 = (i1:'a)
      val i2 = (i2:'a)
      method draw () = i1#draw(); i2#draw()
      method move p = i1#move p; i2#move p
    end;;
```

We then carry on the same principle for the class *e_image*.

```
# class virtual  ext_image () =
    object
      inherit image ()
      method virtual surface : unit → float
    end;;
# class ext_rectangle x y  w h =
    object
      inherit ext_image ()
      inherit rectangle x y w h
```

```
      method surface () = w *. h
    end;;
# class ext_circle x y  r=
    object
      inherit ext_image ()
      inherit circle x y r
      method surface () = 3.14 *. r *.r
    end;;
```

The extension of the class *group* thus becomes

```
# class ['a] ext_group ei1 ei2 =
    object
      inherit image()
      inherit ['a] group ei1 ei2
      method surface () = ei1#surface() +. ei2#surface ()
    end;;
```

We get the following program which constructs a list `le` of the type *ext_image*.

```
# let er = new ext_rectangle 1. 1. 2. 4. ;;
val er : ext_rectangle = <obj>
# let ec = new ext_circle 1. 1. 8.;;
val ec : ext_circle = <obj>
# let eg = new ext_group er ec;;
val eg : ext_rectangle ext_group = <obj>
# let le = [ (er:>ext_image); (ec :> ext_image); (eg :> ext_image)];;
val le : ext_image list = [<obj>; <obj>; <obj>]
# List.map (fun x → x#surface()) le;;
- : float list = [8; 200.96; 208.96]
```

Generalization

To generalize the extension of the methods it is preferable to integrate some functions in a method `handler` and to construct a parameterized class with the return type of the method. For this we define the following class:

```
# class virtual ['a]  get_image (f: 'b → unit → 'a) =
    object(self:'b)
      inherit image ()
      method handler () = f(self) ()
    end;;
```

The following classes then possess an additional functional parameter for the construction of their instances.

```
# class ['a] get_rectangle f x y  w h =
    object(self:'b)
      inherit ['a] get_image f
      inherit rectangle x y w h
      method get = (x,y,w,h)
```

```
    end;;
# class ['a] get_circle f x y  r=
    object(self:'b)
      inherit ['a] get_image f
      inherit circle x y r
      method get = (x,y,r)
    end;;
```

The extension of the class *group* thus takes two type parameters:

```
# class ['a,'c] get_group f eti1 eti2 =
    object
      inherit ['a] get_image f
      inherit ['c] group eti1 eti2
      method get = (i1,i2)
    end;;
```

We get the program which extends the method of the instance of *get_image*.

```
# let etr = new get_rectangle
    (fun r () → let (x,y,w,h) = r#get in w *. h) 1. 1. 2. 4. ;;
val etr : float get_rectangle = <obj>
# let etc = new get_circle
    (fun c () → let (x,y,r) = c#get in 3.14 *. r *. r) 1. 1. 8.;;
val etc : float get_circle = <obj>
# let etg = new get_group
    (fun g () → let (i1,i2) = g#get in i1#handler() +. i2#handler())
    (etr :> float get_image) (etc :> float get_image);;
val etg : (float, float get_image) get_group = <obj>
# let gel = [ (etr :> float get_image) ; (etc :> float get_image) ;
              (etg :> float get_image) ];;
val gel : float get_image list = [<obj>; <obj>; <obj>]
# List.map (fun x → x#handler()) gel;;
- : float list = [8; 200.96; 208.96]
```

The extension of data and methods is easier in the object model when it is combined with the functional model.

Mixed Organisations

The last example of the preceding section showed the advantages that there are in mixing the two models for the problem of the extensibility of components. We now propose to mix parameterized modules and late binding to benefit from the power of these two features. The application of the functor will produce new modules containing classes which use the type and functions of the parameterized module. If, moreover, the signature obtained is compatible with the signature of the parameterized module, it is then possible to re-apply the parameterized module to the resulting module, thus making it possible to construct new classes automatically.

A concrete example is given in the last part of this book which is dedicated to concurrent and/or distributed programs (page 651). We use a functor to generate a communication protocol starting from a data type; a second functor permits us to then deduce from this protocol a class which implements a generic server which handles requests expressed in the protocol. Inheritance can then be used to specialize the server into the service that is actually required.

Exercises

Classes and Modules for Data Structures

We wish to construct class hierarchies based on the application of functors for classical data structures.

We define the following structures

```
# module type ELEMENT =
    sig
      class element :
         string →
         object
           method to_string : unit → string
           method of_string : string → unit
         end
    end ;;

# module type STRUCTURE =
    sig
      class ['a] structure :
        object
          method add : 'a → unit
          method del : 'a → unit
          method mem : 'a → bool
          method get : unit → 'a
          method all : unit → 'a list
          method iter : ('a → unit) → unit
        end
    end ;;
```

1. Write a module with 2 parameters `M1` and `M2` of types `ELEMENT` and `STRUCTURE`, constructing a sub-class of *`['a] structure`* in which *`'a`* is constrained to *`M1.element`*.
2. Write a simple module `Integer` which respects the signature `ELEMENT`.
3. Write a simple module`Stack` which respects the signature `STRUCTURE`.
4. Apply the functor to its two parameters.

5. Modify the functor by adding the methods `to_string` and `of_string`.
6. Apply the functor again , and then apply it to the result .

Abstract Types

Continuing from the previous exercise, we wish to implement a module with signature `ELEMENT` of which the class `element` uses one instance variable of abstract type.

We define the following parameterized type:
```
# type 'a t = {mutable x : 'a t; f : 'a t → unit};;
```

1. Write the functions `apply`, `from_string` and `to_string`. These last two functions will use the `Marshal` module.
2. Write a signature `S` which corresponds to the signature previously inferred by abstracting the type *t*.
3. Write a functor which takes a parameter with signature `S` and returns a module of which the signature is compatible with `ELEMENT`.
4. Use the resulting module as the parameter of the module from the previous exercise.

Summary

This chapter has compared the respective merits of the functional/modular and object models of organisation. Each tries to address in its own way the problems of reusability and modifiability of software. The main differences come from their type systems, equality of types between parameters of functors and sub-typing in the object model, and the evaluation of objects with late binding. The two models do not succeed on their own in resolving the problem of the extensibility of components, from whence we get the idea of a mixed organization. This organization mix also permits new ways of structuring.

To Learn More

The modular model suffers from weak code reuse and difficulties for incremental development. The article "Modular Programming with overloading and delayed linking" ([AC96]) describes a simple extension of the module language, allowing the extension of a module as well as overloading. The choice of code for an overloaded function derives from the techniques used for generic functions in CLOS. The correction of the type system to accommodate these extended modules has not been established.

The issues of mixing the models are well discussed in the article "Modular Object-Oriented Programming with Units and Mixing"([FF98]), in terms of the ease with

which code can be reused. The problem of extensibility of components is described in detail.

This article is available in HTML at the following address:

Link: http://www.cs.rice.edu/CS/PLT/Publications/icfp98-ff/paper.shtml

We can see in these concepts that there is still some dynamic typing involved in type constraints and/or the resolution of type conflicts. It is probably not unreasonable to relax static typing to obtain languages that are "primarily" statically typed in the pursuit of increasing the reusability of the code by facilitating its incremental development.

17

Applications

This chapter illustrates program structure via two examples: the first uses a modular model; the second, an object model.

The first application provides a set of parametric modules for two player games. A functor implements the minimax-$\alpha\beta$ algorithm for the evaluation of a search tree. A second functor allows modifying the human/machine interface for the game. These parametric modules are then applied to two games: a vertical tic-tac-toe game, and another involving the construction of mystic ley-lines.

The second application constructs a world where robots evolve. The world and robots are structured as classes. The different behaviors of robots are obtained by inheritance from a common abstract class. It is then easy to define new behaviors. There, too, the human/machine interface may be modified.

Each of the applications, in its structure, contains reusable components. It is easy to construct a new two player game with different rules that uses the same base classes. Similarly, the general mechanism for the motion of robots in a world may be applied to new types of robots.

Two Player Games

The application presented in this section pursues two objectives. On the one hand, it seeks to resolve problems related to the complexity in searching state spaces, as well as showing that Objective Caml provides useful tools for dealing with symbolic applications. On the other hand, it also explores the benefits of using parametric modules to define a generic scheme for constructing two player games, providing the ability to factor out one part of the search, and making it easy to customize components such as functions for evaluating or displaying a game position.

We first present the problem of games involving two players, then describe the *minimax-$\alpha\beta$* algorithm which provides an efficient search of the tree of possible moves. We present a parametric model for two player games. Then, we apply these functors to implement two games: "Connect Four" (a vertical tic-tac-toe), and Stonehenge (a game that involves constructing ley-lines).

The Problem of Two Player Games

Games involving two players represent one of the classic applications of symbolic programming and provide a good example of problem solving for at least two reasons:

- The large number of solutions to be analyzed to obtain the best possible move necessitates using methods other than brute force.
  For instance, in the game of chess, the number of possible moves typically is around 30, and a game often involves around 40 moves per player. This would require a search tree of around 30^{80} positions just to explore the complete tree for one player.
- The quality of a solution is easily verifiable. In particular, it is possible to test the quality of a proposed solution from one program by comparing it to that of another.

First, assume that we are able to explore the total list of all possible moves, given, as a starting point, a specific legal game position. Such a program will require a function to generate legal moves based on a starting position, as well as a function to evaluate some "score" for each resulting position. The evaluation function must give a maximum score to a winning position, and a minimal score to a losing position. After picking an initial position, one may then construct a tree of all possible variations, where each node corresponds to a position, the adjacent siblings are obtained by having played a move and with leaves having positions indicating winning, losing, or null results. Once the tree is constructed, its exploration permits determining if there exists a route leading to victory, or a null position, failing that. The shortest path may then be chosen to attain the desired goal.

As the overall size of such a tree is generally too large for it to be fully represented, it is typically necessary to limit what portions of the tree are constructed. A first strategy is to limit the "depth" of the search, that is, the number of moves and responses that are to be evaluated. One thus reduces the breadth of the tree as well as its height. In such cases, leaf nodes will seldom be found until nearly the end of the game.

On the other hand, we may try to limit the number of moves selected for additional evaluation. For this, we try to avoid evaluating any but the most favorable moves, and start by examining the moves that appear to be the very best. This immediately eliminates entire branches of the tree. This leads to the *minimax* $\alpha\beta$ algorithm presented in the next subsection.

Minimax $\alpha\beta$

We present the *minimax* search and describe a variant optimized using $\alpha\beta$ cuts. The implementation of this algorithm uses a parametric module, `FAlphabeta` along with a representation of the game and its evaluation function. We distinguish between the two players by naming them A and B.

Minimax

The *minimax* algorithm is a depth-first search algorithm with a limit on the depth to which search is done. It requires:

- a function to generate legal moves based on a position, and
- a function to evaluate a game position.

Starting with some initial game position, the algorithm explores the tree of all legal moves down to the requested depth. Scores associated with leaves of the tree are calculated using an evaluation function. A positive score indicates a good position for player A, while a negative score indicates a poor position for player A, and thus a good position for player B. For each player, the transition from one position to another is either maximized (for player A) or minimized (for player B). Each player tries to select his moves in a manner that will be most profitable for him. In searching for the best play for player A, a search of depth 1 tries to determine the immediate move that maximizes the score of the new position.

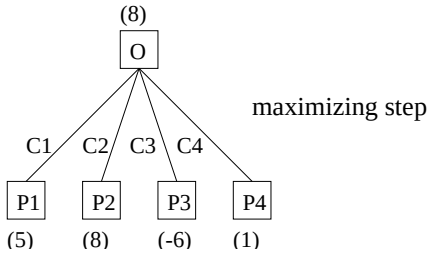


Figure 17.1: Maximizing search at a given location.

In figure 17.1, player A starts at position O, finds four legal moves, constructs these new configurations, and evaluates them. Based on these scores, the best position is P2, with a score of 8. This value is propagated to position O, indicating that this position provides a move to a new position, giving a score of 8 when the player moves to C2. The search of depth 1 is, as a general rule, insufficient, as it does not consider the possible response of an adversary. Such a shallow search results in programs that search greedily for immediate material gains (such as the prize of a queen, in chess) without perceiving that the pieces are protected or that the position is otherwise a losing one (such as a gambit of trading one's queen for a mate). A deeper exploration to depth 2 permits perceiving at least the simplest such countermoves.

Figure 17.2 displays a supplementary analysis of the tree that takes into consideration the possible responses of player B. This search considers B's best moves. For this, the *minimax* algorithm minimizes scores of depth 2.

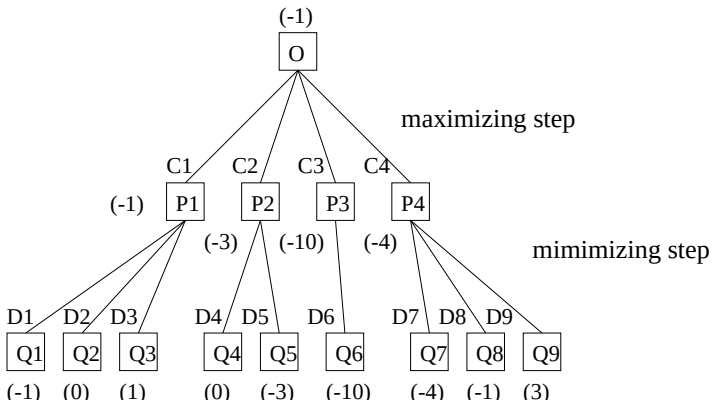


Figure 17.2: Maximizing and minimizing in depth-2 search.

Move P2, which provided an immediate position score of 8, leads to a position with a score of -3. In effect, if B plays D5, then the score of Q5 will be -3. Based on this deeper examination, the move C1 limits the losses with a score of -1, and is thus the preferred move.

In most games, it is possible to try to confuse the adversary, making him play forced moves, trying to muddle the situation in the hope that he will make a mistake. A shallow search of depth 2 would be completely inadequate for this sort of tactic. These sorts of strategies are rarely able to be well exploited by a program because it has no particular vision as to the likely evolution of the positions towards the end of the game.

The difficulty of increased depth of search comes in the form of a combinatorial "explosion." For example, with chess, the exploration of two additional levels adds a factor of around a thousand times more combinations (30×30). Thus, if one searches to a depth of 10, one obtains around 5^{14} positions, which represents too much to search. For this reason, you must try to somehow trim the search tree.

One may note in figure 17.2 that it may be useless to search the branch P3 insofar as the score of this position at depth 1 is poorer than that found in branch P1. In addition the branch P4 does not need to be completely explored. Based on the calculation of Q7, one obtains a score inferior to that of P1, which has already been completely explored. The calculations for Q8 and Q9 cannot improve this situation even if their scores are better than Q7. In a minimizing mode, the poorest score is dropped. The player knows then that these branches provide no useful new options. The minimax variant $\alpha\beta$ uses this approach to decrease the number of branches that must be explored.

Minimax-$\alpha\beta$

We call the α cut the *lower limit* of a maximizing node, and cut β the *upper limit* of a minimizing node. Figure 17.3 shows the cuts carried out in branches P3 and P4 based on knowing the lower limit -1 of P1.

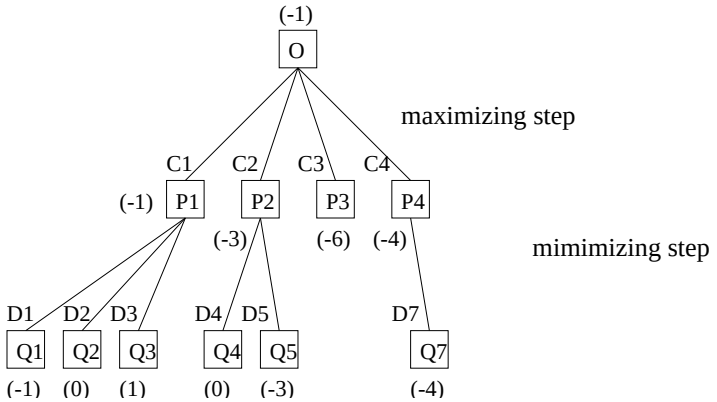


Figure 17.3: Limit α to one level max-min.

As soon as the tree gets broader or deeper the number of cuts increases, thus indicating large subtrees.

A Parametric Module for $\alpha\beta$ *Minimax*

We want to produce a parametric module, `FAlphabeta`, implementing this algorithm, which will be generically reusable for all sorts of two player games. The parameters correspond, on the one hand, to all the information about the proceedings of moves in the game, and on the other hand, to the evaluation function.

Interfaces. We declare two signatures: `REPRESENTATION` to represent plays; and `EVAL` to evaluate a position.

```
# module type REPRESENTATION =
  sig
    type game
    type move
    val game_start : unit → game
    val legal_moves: bool → game → move list
    val play: bool → move → game → game
  end ;;
module type REPRESENTATION =
  sig
    type game
    and move
    val game_start : unit -> game
```

```
    val legal_moves : bool -> game -> move list
    val play : bool -> move -> game -> game
  end

# module type EVAL =
  sig
    type game
    val evaluate: bool → game → int
    val moreI : int
    val lessI: int
    val is_leaf: bool → game → bool
    val is_stable: bool → game → bool
    type state = G | P | N | C
    val state_of : bool → game → state
  end ;;
module type EVAL =
  sig
    type game
    val evaluate : bool -> game -> int
    val moreI : int
    val lessI : int
    val is_leaf : bool -> game -> bool
    val is_stable : bool -> game -> bool
    type state = | G | P | N | C
    val state_of : bool -> game -> state
  end
```

Types `game` and `move` represent abstract types. A player is represented by a boolean value. The function `legal_moves` takes a player and position, and returns the list of possible moves. The function `play` takes a player, a move, and a position, and returns a new position. The values `moreI` and `lessI` are the limits of the values returned by function `evaluate`. The predicate `is_leaf` verifies if a player in a given position can play. The predicate `is_stable` indicates whether the position for the player represents a stable position. The results of these functions influence the pursuit of the exploration of moves when one attains the specified depth.

The signature `ALPHABETA` corresponds to the signature resulting from the complete application of the parametric module that one wishes to use. These hide the different auxiliary functions that we use to implement the algorithm.

```
# module type ALPHABETA = sig
    type game
    type move
    val alphabeta : int → bool → game → move
  end ;;
module type ALPHABETA =
  sig type game and move val alphabeta : int -> bool -> game -> move end
```

The function `alphabeta` takes as parameters the depth of the search, the player, and the game position, returning the next move.

We then define the functional signature `FALPHABETA` which must correspond to that of the implementation of the functor.

```
# module type FALPHABETA = functor (Rep : REPRESENTATION)
    → functor (Eval : EVAL with type game = Rep.game)
      → ALPHABETA with type game = Rep.game
                   and type move = Rep.move ;;
module type FALPHABETA =
  functor(Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
                val evaluate : bool -> game -> int
                val moreI : int
                val lessI : int
                val is_leaf : bool -> game -> bool
                val is_stable : bool -> game -> bool
                type state = | G | P | N | C
                val state_of : bool -> game -> state
              end) ->
      sig
        type game = Rep.game
        and move = Rep.move
        val alphabeta : int -> bool -> game -> move
      end
```

Implementation. The parametric module `FAlphabeta0` makes explicit the partition of the type `game` between the two parameters `Rep` and `Eval`. This module has six functions and two exceptions. The player `true` searches to maximize the score while the player `false` seeks to minimize the score. The function `maxmin_iter` calculates the maximum of the best score for the branches based on a move of player `true` and the pruning parameter α.

The function `maxmin` takes four parameters: `depth`, which indicates the actual calculation depth, `node`, a game position, and α and β, the pruning parameters. If the node is a leaf of the tree or if the maximum depth is reached, the function will return its evaluation of the position. If this is not the case, the function applies `maxmin_iter` to all of the legal moves of player `true`, passing it the search function, diminishing the depth remaining (`minmax`). The latter searches to minimize the score resulting from the response of player `false`.

The movements are implemented using exceptions. If the move β is found in the iteration across the legal moves from the function `maxmin`, then it is returned immediately, the value being propagated using an exception. The functions `minmax_iter` and `minmax` provide the equivalents for the other player. The function `search` determines the move to play based on the best score found in the lists of scores and moves.

The principal function `alphabeta` of this module calculates the legal moves from a given position for the requested player, searches down to the requested depth, and returns the best move.

```
# module FAlphabetaO
      (Rep : REPRESENTATION)  (Eval : EVAL with type game = Rep.game)  =
    struct
      type game = Rep.game
      type move = Rep.move
      exception AlphaMovement of int
      exception BetaMovement of int

      let maxmin_iter node minmax_cur beta alpha cp =
        let alpha_resu =
          max alpha (minmax_cur (Rep.play true cp node) beta alpha)
        in if alpha_resu >= beta then  raise (BetaMovement alpha_resu)
        else alpha_resu

      let minmax_iter node maxmin_cur alpha beta cp =
        let beta_resu =
          min beta (maxmin_cur (Rep.play false cp node) alpha beta)
        in if beta_resu <= alpha then raise (AlphaMovement beta_resu)
        else beta_resu

      let rec maxmin depth node alpha beta =
        if (depth < 1 & Eval.is_stable true node)
            or Eval.is_leaf true node
        then Eval.evaluate true node
        else
          try  let prev = maxmin_iter node (minmax (depth - 1)) beta
          in List.fold_left prev alpha (Rep.legal_moves true node)
          with BetaMovement a → a

      and minmax depth node beta alpha =
        if (depth < 1 & Eval.is_stable false node)
            or Eval.is_leaf false node
        then Eval.evaluate false node
        else
          try let prev = minmax_iter node (maxmin (depth - 1)) alpha
          in List.fold_left prev beta (Rep.legal_moves false node)
          with AlphaMovement b → b

      let rec search a l1 l2 = match (l1,l2) with
        (h1 :: q1, h2 :: q2) → if a = h1 then h2  else search a q1 q2
      | ([], [] ) → failwith  ("AB: "^(string_of_int a)^" not found")
      | (_ ,  _) → failwith  "AB: length differs"

      (* val alphabeta : int -> bool -> Rep.game -> Rep.move *)
      let alphabeta depth player level =
        let alpha = ref Eval.lessI and beta = ref Eval.moreI in
        let l = ref [] in
        let cpl = Rep.legal_moves player level in
```

```
        let eval =
          try
            for i = 0 to (List.length cpl) - 1 do
              if player then
                let b = Rep.play player (List.nth cpl i) level in
                let a = minmax (depth-1) b !beta !alpha
                in l := a :: !l ;
                alpha := max !alpha a ;
                (if !alpha >= !beta then raise (BetaMovement !alpha))
              else
                let a = Rep.play player (List.nth cpl i) level in
                let b = maxmin (depth-1) a !alpha !beta
                in l := b :: !l ;
                beta := min !beta b ;
                (if !beta <= !alpha then raise (AlphaMovement !beta))
            done ;
            if player then !alpha else !beta
          with
            BetaMovement a → a
          | AlphaMovement b → b
        in
        l := List.rev !l ;
        search eval !l cpl
    end ;;
module FAlphabeta0 :
  functor(Rep : REPRESENTATION) ->
    functor
      (Eval : sig
                type game = Rep.game
                val evaluate : bool -> game -> int
                val moreI : int
                val lessI : int
                val is_leaf : bool -> game -> bool
                val is_stable : bool -> game -> bool
                type state = | G | P | N | C
                val state_of : bool -> game -> state
              end) ->
      sig
        type game = Rep.game
        and move = Rep.move
        exception AlphaMovement of int
        exception BetaMovement of int
        val maxmin_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val minmax_iter :
          Rep.game ->
          (Rep.game -> int -> int -> int) -> int -> int -> Rep.move -> int
        val maxmin : int -> Eval.game -> int -> int -> int
        val minmax : int -> Eval.game -> int -> int -> int
        val search : int -> int list -> 'a list -> 'a
        val alphabeta : int -> bool -> Rep.game -> Rep.move
```

```
    end
```

We may close module `FAlphabetaO` by associating with it the following signature:

```
# module FAlphabeta = (FAlphabetaO : FALPHABETA) ;;
module FAlphabeta : FALPHABETA
```

This latter module may be used with many different game representations and functions to play different games.

Organization of a Game Program

The organization of a program for a two player game may be separated into a portion specific to the game in question as well as a portion applicable to all sorts of games. For this, we propose using several parametric modules parameterized by specific modules, permitting us to avoid the need to rewrite the common portions each time. Figure 17.4 shows the chosen organization.

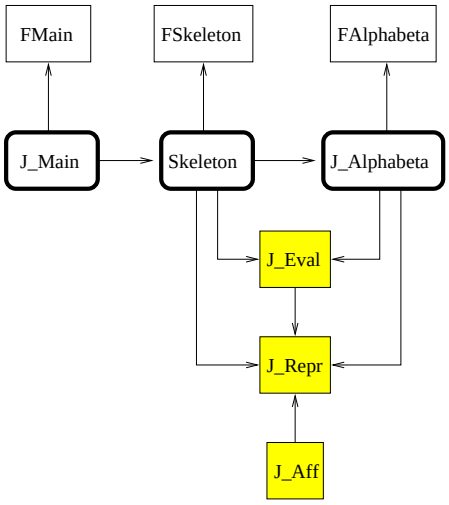


Figure 17.4: Organization of a game application.

The modules with no highlighting correspond to the common parts of the application. These are the parametric modules. We see again the functor `FAlphabeta`. The modules with gray highlighting are the modules designed specifically for a given game. The three principal modules are the representation of the game (`J_Repr`), display of the game (`J_Disp`), and the evaluation function (`J_Eval`). The modules with rounded gray borders are obtained by applying the parametric modules to the simple modules specific to the game.

The module `FAlphabeta` has already been described. The two other common modules are `FMain`, containing the main loop, and `FSkeleton`, that manages the players.

Module `FMain`

Module `FMain` contains the main loop for execution of a game program. It is parameterized using the signature module `SKELETON`, describing the interaction with a player using the following definition:

```
# module type SKELETON = sig
    val home: unit → unit
    val init: unit → ((unit → unit) * (unit → unit))
    val again: unit → bool
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    exception Won
    exception Lost
    exception Nil
  end ;;
module type SKELETON =
  sig
    val home : unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
    val again : unit -> bool
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    exception Won
    exception Lost
    exception Nil
  end
```

The function `init` constructs a pair of action functions for each player. The other functions control the interactions. Module `FMain` contains two functions: `play_game` which alternates between the players, and `main` which controls the main loop.

```
# module FMain (P : SKELETON) =
    struct
      let play_game movements = while true do (fst movements) () ;
        (snd movements) () done

      let main () = let finished = ref false
      in P.home ();
      while not !finished do
        ( try  play_game (P.init ())
        with P.Won → P.won ()
```

```
        | P.Lost  → P.lost ()
        | P.Nil   → P.nil () );
        finished := not (P.again ())
      done ;
      P.exit ()
  end ;;
module FMain :
  functor(P : SKELETON) ->
    sig
      val play_game : (unit -> 'a) * (unit -> 'b) -> unit
      val main : unit -> unit
    end
```

Module **FSkeleton**

Parametric module `FSkeleton` controls the moves of each player according to the rules provided at the start of the section based on the nature of the players (automated or not) and the order of the players. It needs various parameters to represent the game, game states, the evaluation function, and the $\alpha\beta$ search as described in figure 17.4.

We start with the signature needed for game display.

```
# module type DISPLAY = sig
    type game
    type move
    val home: unit → unit
    val exit: unit → unit
    val won: unit → unit
    val lost: unit → unit
    val nil: unit → unit
    val init: unit → unit
    val position : bool → move → game → game → unit
    val choice : bool →  game → move
    val q_player : unit → bool
    val q_begin : unit → bool
    val q_continue : unit → bool
  end ;;
module type DISPLAY =
  sig
    type game
    and move
    val home : unit -> unit
    val exit : unit -> unit
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> move -> game -> game -> unit
    val choice : bool -> game -> move
    val q_player : unit -> bool
```

```
    val q_begin : unit -> bool
    val q_continue : unit -> bool
  end
```

It is worth noting that the representation of the game and of the moves must be shared by all the parametric modules, which constrain the types. The two principal functions are `playH` and `playM`, respectively controlling the move of a human player (using the function `Disp.choice`) and that of an automated player. The function `init` determines the nature of the players and the sorts of responses for `Disp.q_player`.

```
# module FSkeleton
      (Rep   : REPRESENTATION)
      (Disp   : DISPLAY with type game = Rep.game and  type move = Rep.move)
      (Eval  : EVAL with type game = Rep.game)
      (Alpha : ALPHABETA with type game = Rep.game and  type move = Rep.move) =
    struct
      let depth = ref 4
      exception Won
      exception Lost
      exception Nil
      let won = Disp.won
      let lost = Disp.lost
      let nil = Disp.nil
      let again = Disp.q_continue
      let play_game = ref (Rep.game_start())
      let exit = Disp.exit
      let home = Disp.home

      let playH player () =
        let choice = Disp.choice player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.P → raise Lost
        | Eval.G → raise Won
        | Eval.N → raise Nil
        | _      → ()

      let playM player () =
        let choice = Alpha.alphabeta !depth player !play_game in
        let old_game = !play_game
        in play_game := Rep.play player choice !play_game ;
        Disp.position player choice old_game !play_game ;
        match Eval.state_of player !play_game with
          Eval.G → raise Won
        | Eval.P → raise Lost
        | Eval.N → raise Nil
        | _      → ()
```

```
      let init () =
        let a = Disp.q_player () in
        let b = Disp.q_player()
        in play_game := Rep.game_start () ;
        Disp.init () ;
        match (a,b) with
          true,true   → playM true, playM false
        | true,false  → playM true, playH false
        | false,true  → playH true, playM false
        | false,false → playH true, playH false
  end ;;
module FSkeleton :
  functor(Rep : REPRESENTATION) ->
    functor
      (Disp : sig
                type game = Rep.game
                and move = Rep.move
                val home : unit -> unit
                val exit : unit -> unit
                val won : unit -> unit
                val lost : unit -> unit
                val nil : unit -> unit
                val init : unit -> unit
                val position : bool -> move -> game -> game -> unit
                val choice : bool -> game -> move
                val q_player : unit -> bool
                val q_begin : unit -> bool
                val q_continue : unit -> bool
              end) ->
      functor
        (Eval : sig
                  type game = Rep.game
                  val evaluate : bool -> game -> int
                  val moreI : int
                  val lessI : int
                  val is_leaf : bool -> game -> bool
                  val is_stable : bool -> game -> bool
                  type state = | G | P | N | C
                  val state_of : bool -> game -> state
                end) ->
          functor
            (Alpha : sig
                       type game = Rep.game
                       and move = Rep.move
                       val alphabeta : int -> bool -> game -> move
                     end) ->
            sig
              val depth : int ref
              exception Won
              exception Lost
              exception Nil
              val won : unit -> unit
```

```
  val lost : unit -> unit
  val nil : unit -> unit
  val again : unit -> bool
  val play_game : Disp.game ref
  val exit : unit -> unit
  val home : unit -> unit
  val playH : bool -> unit -> unit
  val playM : bool -> unit -> unit
  val init : unit -> (unit -> unit) * (unit -> unit)
end
```

The independent parts of the game are thus implemented. One may then begin programming different sorts of games. This modular organization facilitates making modifications to the movement scheme or to the evaluation function for a game as we shall soon see.

Connect Four

We will next examine a simple game, a vertical tic-tac-toe, known as Connect Four. The game is represented by seven columns each consisting of six lines. In turn, a player places on a column a piece of his color, where it then falls down to the lowest free location in this column. If a column is completely filled, neither player is permitted to play there. The game ends when one of the players has built a line of four pieces in a row (horizontal, vertical, or diagonal), at which point this player has won, or when all the columns are filled with pieces, in which the outcome is a draw. Figure 17.5 shows a completed game.

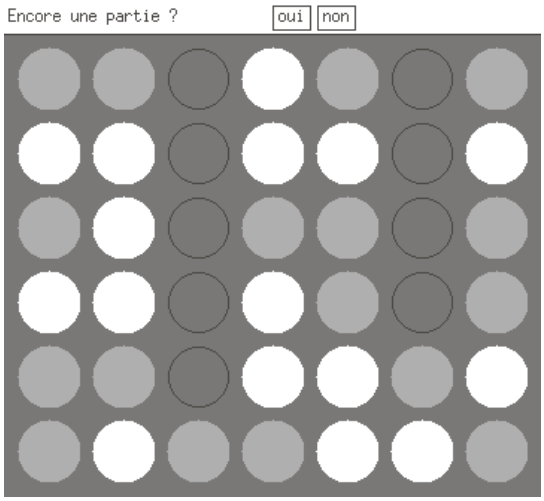


Figure 17.5: An example of Connect Four.

Note the "winning" line of four gray pieces in a diagonal, going down and to the right.

Game Representation: module `C4_rep`. We choose for this game a matrix-based representation. Each element of the matrix is either empty, or contains a player's piece. A move is numbered by the column. The legal moves are the columns in which the final (top) row is not filled.

```
# module C4_rep = struct
    type cell = A | B | Empty
    type game = cell array array
    type move = int
    let col = 7 and row = 6
    let game_start () = Array.create_matrix row col Empty

    let legal_moves b m =
      let l = ref [] in
      for c = 0 to col-1 do if m.(row-1).(c) = Empty then l := (c+1) :: !l done;
      !l

    let augment mat c =
      let l = ref row
      in while !l > 0 & mat.(!l-1).(c-1) = Empty do  decr l done ; !l + 1

    let player_gen cp m e =
      let mj = Array.map Array.copy  m
      in  mj.((augment mj cp)-1).(cp-1) <- e ; mj

    let play b cp m = if b then player_gen cp m A else player_gen cp m B
  end ;;
module C4_rep :
  sig
    type cell = | A | B | Empty
    and game = cell array array
    and move = int
    val col : int
    val row : int
    val game_start : unit -> cell array array
    val legal_moves : 'a -> cell array array -> int list
    val augment : cell array array -> int -> int
    val player_gen : int -> cell array array -> cell -> cell array array
    val play : bool -> int -> cell array array -> cell array array
  end
```

We may easily verify if this module accepts the constraints of the signature `REPRESENTATION`.

```
# module C4_rep_T = (C4_rep : REPRESENTATION) ;;
module C4_rep_T : REPRESENTATION
```

Game Display: Module `C4_text`. Module `C4_text` describes a text-based interface for the game Connect Four that is compatible with the signature `DISPLAY`. It it is not particularly sophisticated, but, nonetheless, demonstrates how modules are assembled together.

```
# module C4_text = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move

    let print_game mat =
      for l = row - 1 downto 0 do
        for c = 0 to col - 1 do
          match mat.(l).(c) with
            A     → print_string "X "
          | B     → print_string "O "
          | Empty → print_string ". "
        done;
        print_newline ()
      done ;
      print_newline ()

    let home () = print_string "C4 ...\n"
    let exit () = print_string "Bye for now ... \n"
    let question s =
      print_string s;
      print_string " y/n ? " ;
      read_line() = "y"
    let q_begin () = question "Would you like to begin?"
    let q_continue () = question "Play again?"
    let q_player () = question "Is there to be a machine player ?"

    let won  ()= print_string "The first player won" ; print_newline ()
    let lost () = print_string "The first player lost" ; print_newline ()
    let nil () = print_string "Stalemate" ; print_newline ()

    let init () =
      print_string "X: 1st player   O: 2nd player";
      print_newline () ; print_newline () ;
      print_game (game_start ()) ; print_newline()

    let position b c aj j = print_game j

    let is_move = function '1'..'7' → true | _ → false

    exception Move of int
    let rec choice player game =
      print_string ("Choose player" ^ (if player then "1" else "2") ^ " : ") ;
      let l = legal_moves player game
      in try  while true do
        let i = read_line()
```

```
        in ( if (String.length i > 0)  && (is_move  i.[0])
        then let c = (int_of_char i.[0]) - (int_of_char '0')
        in if List.mem c l then raise (Move c) );
        print_string "Invalid move - try again"
      done ;
        List.hd l
      with  Move i → i
      | _ → List.hd l
  end ;;
module C4_text :
  sig
    type game = C4_rep.game
    and move = C4_rep.move
    val print_game : C4_rep.cell array array -> unit
    val home : unit -> unit
    val exit : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : 'a -> 'b -> 'c -> C4_rep.cell array array -> unit
    val is_move : char -> bool
    exception Move of int
    val choice : bool -> C4_rep.cell array array -> int
  end
```

We may immediately verify that this conforms to the constraints of the signature DISPLAY

```
# module C4_text_T = (C4_text : DISPLAY) ;;
module C4_text_T : DISPLAY
```

Evaluation Function: module C4_eval. The quality of a game player depends primarily on the position evaluation function. Module C4_eval defines evaluate, which evaluates the value of a position for the specified player. This function calls eval_bloc for the four compass directions as well as the diagonals. eval_bloc then calls eval_four to calculate the number of pieces in the requested line. Table value provides the value of a block containing 0, 1, 2, or 3 pieces of the same color. The exception Four is raised when 4 pieces are aligned.

```
# module C4_eval = struct open C4_rep type game = C4_rep.game
    let value =
      Array.of_list [0; 2; 10; 50]
    exception Four of int
```

```
exception Nil_Value
exception Arg_invalid
let lessI = -10000
let moreI = 10000
let eval_four m l_dep c_dep delta_l delta_c =
  let n = ref 0 and e = ref Empty
  and x = ref c_dep  and  y = ref l_dep
  in try
    for i = 1 to 4 do
      if !y<0 or !y>=row or !x<0 or !x>=col then raise Arg_invalid ;
      ( match m.(!y).(!x) with
        A    → if !e = B then raise Nil_Value ;
          incr n ;
          if !n = 4 then raise (Four moreI) ;
          e := A
      | B    → if !e = A then raise Nil_Value ;
          incr n ;
          if !n = 4 then raise (Four lessI);
          e := B;
      | Empty → () ) ;
          x := !x + delta_c ;
      y := !y + delta_l
    done ;
    value.(!n) * (if !e=A then 1 else -1)
  with
    Nil_Value | Arg_invalid  → 0

let eval_bloc m e cmin cmax lmin lmax dx dy =
  for c=cmin to cmax do for l=lmin to lmax do
    e := !e + eval_four m l c dx dy
  done done

let evaluate b m =
  try let evaluation = ref 0
  in (* evaluation of rows *)
  eval_bloc m evaluation 0 (row-1) 0 (col-4) 0 1 ;
  (* evaluation of columns *)
  eval_bloc m evaluation 0 (col-1) 0 (row-4) 1 0 ;
  (* diagonals coming from the first line (to the right) *)
  eval_bloc m evaluation 0 (col-4) 0 (row-4) 1 1 ;
  (* diagonals coming from the first line (to the left) *)
  eval_bloc m evaluation 1 (row-4) 0 (col-4) 1 1 ;
  (* diagonals coming from the last line (to the right) *)
  eval_bloc m evaluation 3 (col-1) 0 (row-4) 1 (-1) ;
  (* diagonals coming from the last line (to the left) *)
  eval_bloc m evaluation 1 (row-4) 3 (col-1) 1 (-1) ;
  !evaluation
  with Four v → v

let is_leaf b m = let v = evaluate b m
in v=moreI or v=lessI or legal_moves b m = []
```

```
    let is_stable b j = true

    type state = G | P | N | C

    let state_of player m =
      let v = evaluate player m
      in if v = moreI then if player then G else P
      else if v = lessI then if player then P else G
      else if legal_moves player m = [] then N else C
  end ;;
module C4_eval :
  sig
    type game = C4_rep.game
    val value : int array
    exception Four of int
    exception Nil_Value
    exception Arg_invalid
    val lessI : int
    val moreI : int
    val eval_four :
      C4_rep.cell array array -> int -> int -> int -> int -> int
    val eval_bloc :
      C4_rep.cell array array ->
      int ref -> int -> int -> int -> int -> int -> int -> unit
    val evaluate : 'a -> C4_rep.cell array array -> int
    val is_leaf : 'a -> C4_rep.cell array array -> bool
    val is_stable : 'a -> 'b -> bool
    type state = | G | P | N | C
    val state_of : bool -> C4_rep.cell array array -> state
  end
```

Module `C4_eval` is compatible with the constraints of signature `EVAL`.

```
# module C4_eval_T = (C4_eval : EVAL) ;;
module C4_eval_T : EVAL
```

To play two evaluation functions against one another, it is necessary to modify `evaluate` to apply the proper evaluation function for each player.

Assembly of the modules All the components needed to realize the game of Connect Four are now implemented. We only need assemble them together based on the schema of diagram 17.4. First, we construct `C4_skeleton`, which is the application of parameter module `FSkeleton` to modules `C4_rep`, `C4_text`, `C4_eval` and the result of the application of parametric module `FAlphaBeta` to `C4_rep` and `C4_eval`.

```
# module C4_skeleton =
```

```
    FSkeleton (C4_rep) (C4_text) (C4_eval)  (FAlphabeta (C4_rep) (C4_eval)) ;;
module C4_skeleton :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : C4_text.game ref
    val exit : unit -> unit
    val home : unit -> unit
    val playH : bool -> unit -> unit
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

We then obtain the principal module `C4_main` by applying parametric module `FMain` on the result of the preceding application `C4_skeleton`

```
# module C4_main = FMain(C4_skeleton) ;;
module C4_main :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

The game is initiated by the application of function `C4_main.main` on `()`.

Testing the Game. Once the general game skeleton has been written, games may be played in various ways. Two human players may play against each other, with the program merely verifying the validity of the moves; a person may play against a programmed player; or programs may play against each other. While this last mode might not be interesting for the human, it does make it easy to run tests without having to wait for a person's responses. The following game demonstrates this scenario.

```
# C4_main.main () ;;
C4 ...
Is there to be a machine player ? y/n ? y
Is there to be a machine player ? y/n ? y
X: 1st player   O: 2nd player

. . . . . .
. . . . . .
. . . . . .
```

```
. . . . . .
. . . . . .
. . . . . .
```

Once the initial position is played, player 1 (controlled by the program) calculates its move which is then applied.

```
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . .
. . . . . X .
```

Player 2 (always controlled by the program) calculates its response and the game proceeds, until a game-ending move is found. In this example, player 1 wins the game based on the following final position:

```
. O O O . O .
. X X X . X .
X O O X . O .
X X X O . X .
X O O X X O .
X O O O X X O
Player 1 wins
Play again(y/n) ? n
Good-bye ...
- : unit = ()
```

Graphical Interface. To improve the enjoyment of the game, we define a graphical interface for the program, by defining a new module, `C4_graph`, compatible with the signature `DISPLAY`, which opens a graphical window, controlled by mouse clicks. The text of this module may be found in the subdirectory `Applications` on the CD-ROM (see page 1).

```
# module C4_graph = struct
    open C4_rep
    type game = C4_rep.game
    type move = C4_rep.move
    let r = 20          (* color of piece *)
    let ec = 10         (* distance between pieces *)
    let dec = 30        (* center of first piece *)
    let cote = 2*r + ec (* height of a piece looked at like a checker *)
    let htexte = 25     (* where to place text *)
    let width = col * cote + ec          (* width of the window *)
    let height = row * cote + ec + htexte (* height of the window *)
```

```
  let height_of_game = row * cote + ec   (* height of game space *)
  let hec = height_of_game + 7  (* line for messages *)
  let lec = 3                    (* columns for messages *)
  let margin = 4                  (* margin for buttons *)
  let xb1 = width / 2   (* position x of button1 *)
  let xb2 = xb1 + 30      (* position x of button2 *)
  let yb = hec - margin    (* position y of the buttons *)
  let wb = 25             (* width of the buttons *)
  let hb = 16             (* height of the buttons *)

(* val t2e : int -> int *)
(* Convert a matrix coordinate into a graphical coordinate *)
  let t2e i = dec + (i-1)*cote

(* The Colors *)
  let cN = Graphics.black   (* trace *)
  let cA = Graphics.red     (* Human player *)
  let cB = Graphics.yellow  (* Machine player *)
  let cF = Graphics.blue    (* Game Background color *)
(* val draw_table : unit -> unit :  Trace an empty table *)
  let draw_table () =
    Graphics.clear_graph();
    Graphics.set_color cF;
    Graphics.fill_rect 0 0 width height_of_game;
    Graphics.set_color cN;
    Graphics.moveto 0 height_of_game;
    Graphics.lineto width height_of_game;
    for l = 1 to row do
      for c = 1 to col do
        Graphics.draw_circle (t2e c) (t2e l) r
      done
    done

(* val draw_piece : int -> int -> Graphics.color -> unit *)
(* 'draw_piece l c co' draws a piece of color co at coordinates l c *)
  let draw_piece l c col =
    Graphics.set_color col;
    Graphics.fill_circle (t2e c) (t2e l) (r+1)

(* val augment : Rep.item array array -> int -> Rep.move *)
(* 'augment m c' redoes the line or drops the piece for c in m *)
  let augment mat c =
    let l = ref row in
    while !l > 0 & mat.(!l-1).(c-1) = Empty do
      decr l
    done;
    !l

(* val conv : Graphics.status -> int *)
(* convert the region where player has clicked in controlling the game *)
  let conv st =
```

```
    (st.Graphics.mouse_x - 5) / 50 + 1

(* val wait_click : unit -> Graphics.status *)
(* wait for a mouse click *)
  let wait_click () = Graphics.wait_next_event [Graphics.Button_down]

(* val choiceH : Rep.game -> Rep.move *)
(* give opportunity to the human player to choose a move *)
(* the function offers possible moves *)
  let rec choice player  game  =
    let c = ref 0 in
    while not ( List.mem !c (legal_moves player game) ) do
      c := conv ( wait_click() )
    done;
    !c
(* val home : unit -> unit : home screen *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int width) ^ "x" ^ (string_of_int height) ^ "+50+50");
    Graphics.moveto (height/2) (width/2);
    Graphics.set_color cF;
    Graphics.draw_string "C4";
    Graphics.set_color cN;
    Graphics.moveto 2 2;
    Graphics.draw_string "by Romuald COEFFIER & Mathieu DESPIERRE";
    ignore (wait_click ());
    Graphics.clear_graph()

(* val end : unit -> unit ,  the end of the game *)
  let exit () = Graphics.close_graph()

(* val draw_button : int -> int -> int -> int -> string -> unit *)
(* 'draw_button x y w h s' draws a rectangular button at coordinates *)
(* x,y with width w and height h and appearance s *)
  let draw_button x y w h s =
    Graphics.set_color cN;
    Graphics.moveto x y;
    Graphics.lineto x (y+h);
    Graphics.lineto (x+w) (y+h);
    Graphics.lineto (x+w) y;
    Graphics.lineto x y;
    Graphics.moveto (x+margin) (hec);
    Graphics.draw_string s

(* val draw_message : string -> unit  * position message s *)
  let draw_message s =
    Graphics.set_color cN;
    Graphics.moveto lec hec;
    Graphics.draw_string s

(* val erase_message : unit -> unit  erase the starting position *)
```

```
  let erase_message () =
    Graphics.set_color Graphics.white;
    Graphics.fill_rect 0 (height_of_game+1) width htexte

(* val question : string -> bool *)
(* 'question s' poses the question s, the response being obtained by *)
(*    selecting one of two buttons, 'yes' (=true) and 'no' (=false) *)
  let question s =
    let rec attente () =
      let e = wait_click () in
      if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
        if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
          true
        else
          if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
            false
          else
            attente()
      else
      attente () in
    draw_message s;
    draw_button xb1 yb wb hb "yes";
    draw_button xb2 yb wb hb "no";
    attente()

(* val q_begin : unit -> bool *)
(* Ask, using function 'question', if the player wishes to start *)
(* (yes=true) *)
  let q_begin () =
    let b = question "Would you like to begin ?" in
    erase_message();
    b

(* val q_continue : unit -> bool *)
(* Ask, using function 'question', if the player wishes to play again *)
(* (yes=true) *)
  let q_continue () =
    let b = question "Play again ?" in
    erase_message();
    b

  let q_player () =
    let b = question "Is there to be a machine player?" in
    erase_message ();
    b
(* val won : unit -> unit *)
(* val lost : unit -> unit *)
(* val nil : unit -> unit *)
(* Three functions for these three cases *)
  let won () =
    draw_message "I won :-)" ; ignore (wait_click ()) ; erase_message()
```

```
    let lost () =
      draw_message "You won :-("; ignore (wait_click ()) ; erase_message()
    let nil () =
      draw_message "Stalemate" ; ignore (wait_click ()) ; erase_message()

  (* val init : unit -> unit *)
  (* This is called at every start of the game for the position *)
    let init  = draw_table

    let position b c aj nj  =
      if b then
        draw_piece (augment nj c) c cA
      else
        draw_piece (augment nj c) c cB

  (* val drawH : int -> Rep.item array array -> unit *)
  (* Position when the human player chooses move cp in situation j *)
    let drawH cp j =  draw_piece (augment j cp) cp cA

  (* val drawM : int -> cell array array -> unit*)
  (* Position when the machine player chooses move cp in situation j *)
    let drawM cp j =  draw_piece (augment j cp) cp cB
  end ;;
module C4_graph :
  sig
    type game = C4_rep.game
    and move = C4_rep.move
    val r : int
    val ec : int
    val dec : int
    val cote : int
    val htexte : int
    val width : int
    val height : int
    val height_of_game : int
    val hec : int
    val lec : int
    val margin : int
    val xb1 : int
    val xb2 : int
    val yb : int
    val wb : int
    val hb : int
    val t2e : int -> int
    val cN : Graphics.color
    val cA : Graphics.color
    val cB : Graphics.color
    val cF : Graphics.color
    val draw_table : unit -> unit
    val draw_piece : int -> int -> Graphics.color -> unit
    val augment : C4_rep.cell array array -> int -> int
    val conv : Graphics.status -> int
```

```
    val wait_click : unit -> Graphics.status
    val choice : 'a -> C4_rep.cell array array -> int
    val home : unit -> unit
    val exit : unit -> unit
    val draw_button : int -> int -> int -> int -> string -> unit
    val draw_message : string -> unit
    val erase_message : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val q_player : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val position : bool -> int -> 'a -> C4_rep.cell array array -> unit
    val drawH : int -> C4_rep.cell array array -> unit
    val drawM : int -> C4_rep.cell array array -> unit
  end
```

We may also create a new skeleton (`C4_skeletonG`) which results from the application of parametric module `FSkeleton`.

```
# module C4_skeletonG =
    FSkeleton (C4_rep) (C4_graph) (C4_eval) (FAlphabeta (C4_rep) (C4_eval)) ;;
```

Only the display parameter differs from the text version application of `FSkeleton`. We may thereby create a principal module for Connect Four with a graphical user interface.

```
# module C4_mainG = FMain(C4_skeletonG) ;;
module C4_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

The evaluation of the expression `C4_mainG.main()` opens a graphical window as in figure 17.5 and controls the interaction with the user.

Stonehenge

Stonehenge, created by Reiner Knizia, is a game involving construction of "ley-lines." The rules are simple to understand but our interest in the game lies in its high number of possible moves. The rules may be found at:

Link: http://www.cix.co.uk/~convivium/files/stonehen.htm

The initial game position is represented in figure 17.6.

Game Presentation

The purpose of the game is to win at least 8 "ley-lines" (clear lines) out of the 15 available. One gains a line by positioning pieces (or megaliths) on gray positions along a ley-line.

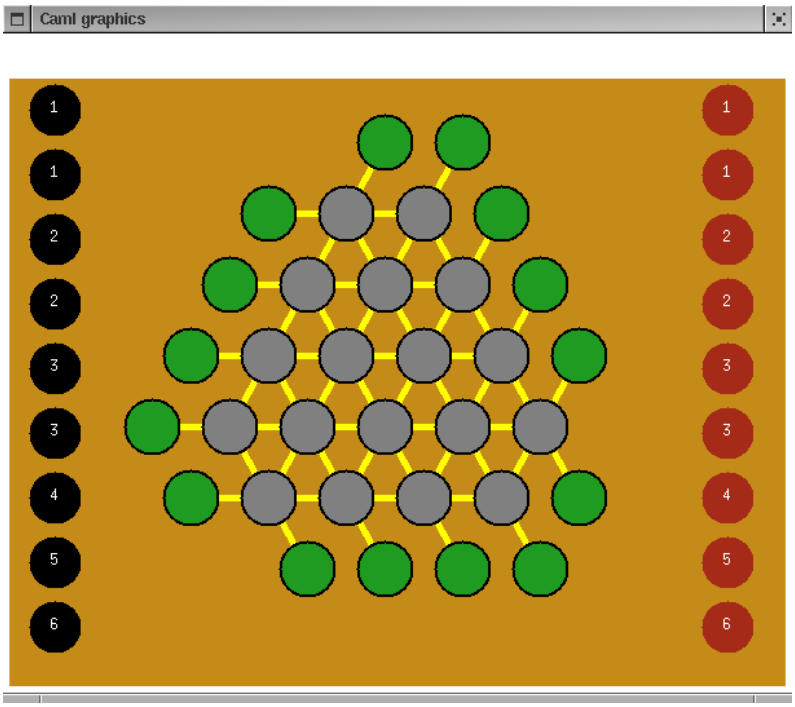


Figure 17.6: Initial position of Stonehenge.

In turn, each player places one of his 9 pieces, numbered from 1 to 6, on one of the 18 gray internal positions. They may not place a piece on a position that is already occupied. Each time a piece is placed, one or several ley-lines may be won or lost.

A ley-line is won by a player if the total of the values of his pieces on the line is greater than the total of the pieces for the other player. There may be empty spaces left if the opponent has no pieces left that would allow winning the line.

For example in figure 17.7, the black player starts by placing the piece of value 3, the red player his "2" piece, then the black player plays the "6" piece, winning a line.

Red then plays the "4" piece, also winning a ley-line. This line has not been completely filled, but red has won because there is no way for black to overcome red's score.

Note that the red player might just as well have played "3" rather than "4," and still won the line. In effect, there is only one free case for this ley-line where the strongest black piece has a value of 5, and so black cannot beat red for this particular line.

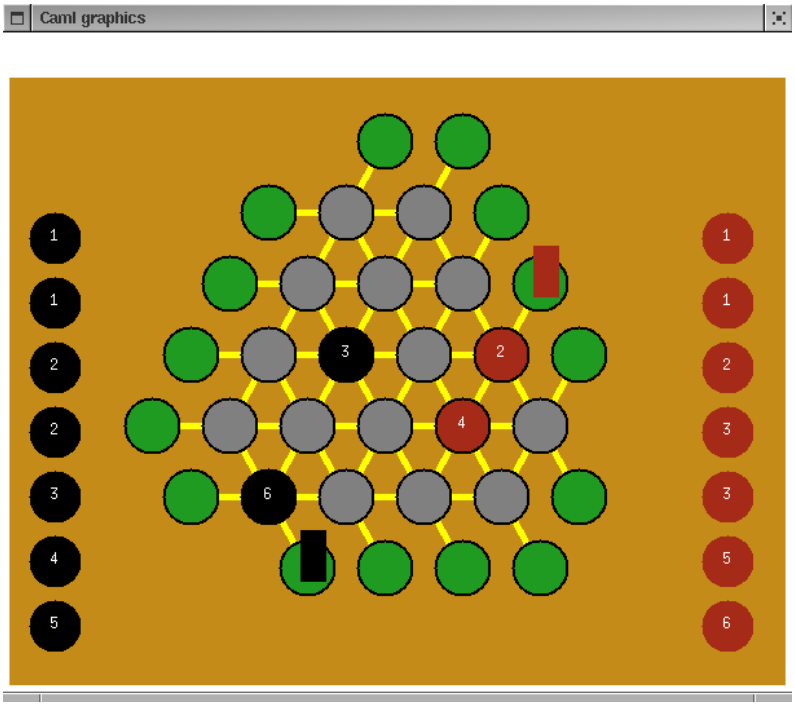


Figure 17.7: Position after 4 moves.

In the case where the scores are equal across a full line, whoever placed the last piece *without* having beaten his adversary's score, loses the line. Figure 17.8 demonstrates such a situation.

The last red move is piece "4". On the full line where the "4" is placed, the scores are equal. Since red was the last player to have placed a piece, but did not beat his adversary, red loses the line, as indicated by a black block.

We may observe that the function `play` fills the role of arbitrating and accounting for these subtleties in the placement of lines.

There can never be a tie in this game. There are 15 lines, each of which will be accounted for at some point in the game, at which point one of the players will have won at least 8 lines.

Search Complexity

Before completely implementing a new game, it is important to estimate the number of legal moves between two moves in a game, as well as the number of possible moves

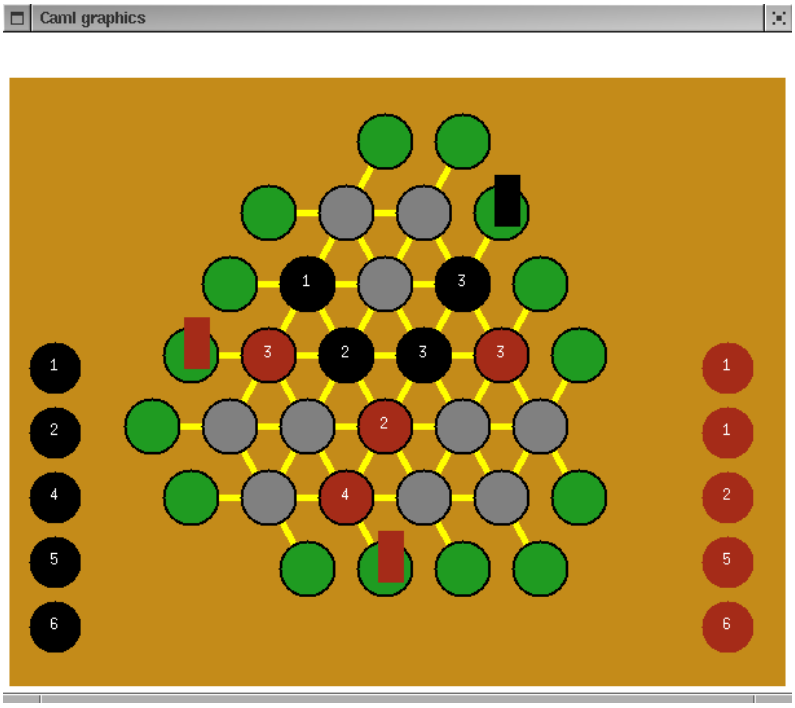


Figure 17.8: Position after 6 moves.

for each side. These values may be used to estimate a reasonable maximum depth for the minimax-$\alpha\beta$ algorithm.

In the game Stonehenge, the number of moves for each side is initially based on the number of pieces for the two players, that is, 18. The number of possible moves diminishes as the game progresses. At the first move, the player has 6 different pieces and 18 positions free. At the second move, the second player has 6 different pieces, and 17 positions in which they may be placed (102 legal moves). Moving from a depth of 2 to 4 for the initial moves of the game results in the number of choices going from about 10^4 to about 10^8.

On the other hand, near the end of the game, in the final 8 moves, the complexity is greatly reduced. If we take a pessimistic calculation (where all pieces are different), we obtain about 23 million possibilities:

$$4 * 8 * 4 * 7 * 3 * 6 * 3 * 5 * 2 * 4 * 2 * 3 * 1 * 2 * 1 * 1 = 23224320$$

It might seem appropriate to calculate with a depth of around 2 for the initial set of moves. This may depend on the evaluation function, and on its ability to evaluate the positions at the start of the game, when there are few pieces in place. On the other

hand, near the end of the game, the depth may readily be increased to around 4 or 6, but this would probably be too late a point to recover from a weak position.

Implementation

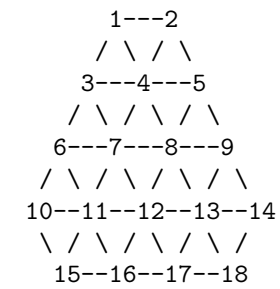
We jump straight into describing the game representation and arbitration so that we may concentrate on the evaluation function.

The implementation of this game follows the architecture used for Connect Four, described in figure 17.4. The two principal difficulties will be to follow the game rules for the placement of pieces, and the evaluation function, which must be able to evaluate positions as quickly as possible while remaining useful.

Game Representation. There are four notable data structures in this game:

- the pieces of the players (type *piece*),
- the positions (type *placement*),
- the 15 ley-lines,
- the 18 locations where pieces may be placed.

We provide a unique number for each location:



Each location participates in 3 ley-lines. We also number each ley-line. This description may be found in the declaration of the list *lines*, which is converted to a vector (`vector_l`). A location is either empty, or contains a piece that has been placed, and the piece's possessor. We also store, for each location, the number of the lines that pass through it. This table is calculated by `lines_per_case` and is named `num_line_per_case`.

The game is represented by the vector of 18 cases, the vector of 15 ley-lines either won or not, and the lists of pieces left for the two players. The function `game_start` creates these four elements.

The calculation of a player's legal moves resolves into a Cartesian product of the pieces available against the free positions. Various utility functions allow counting the score of a player on a line, calculating the number of empty locations on a line, and verifying

if a line has already been won. We only need to implement `play` which plays a move and decides which pieces to place. We write this function at the end of the listing of module `Stone_rep`.

```
# module Stone_rep = struct
    type player = bool
    type piece =  P of int
    let int_of_piece = function P x → x
    type placement = None | M of player
    type case = Empty | Occup of player*piece
    let value_on_case = function
        Empty → 0
      | Occup (j, x) → int_of_piece x

    type game = J of case array * placement array * piece list * piece list
    type move =  int * piece

    let lines = [
      [0;1]; [2;3;4]; [5; 6; 7; 8;]; [9; 10; 11; 12; 13]; [14; 15; 16; 17];
      [0; 2; 5; 9]; [1; 3; 6; 10; 14]; [4; 7; 11; 15]; [8; 12; 16]; [13; 17];
      [9; 14]; [5; 10; 15]; [2; 6; 11; 16]; [0; 3; 7; 12; 17]; [1; 4; 8; 13]  ]

    let vector_l = Array.of_list lines

    let lines_per_case v =
      let t =  Array.length v  in
      let r = Array.create 18 [||] in
      for i = 0 to 17  do
        let w = Array.create 3 0
        and p = ref 0 in
        for j=0 to t-1 do if List.mem i v.(j) then (w.(!p) <- j; incr p)
        done;
        r.(i) <- w
      done;
      r

    let num_line_per_case = lines_per_case vector_l
    let rec lines_of_i i l = List.filter (fun t → List.mem i t) l

    let lines_of_cases l =
      let a = Array.create 18 l in
      for i=0 to 17 do
        a.(i) <-  (lines_of_i i l)
      done; a

    let ldc = lines_of_cases lines

    let game_start ()= let lp = [6; 5; 4; 3; 3; 2; 2; 1; 1] in
    J ( Array.create 18 Empty, Array.create 15 None,
        List.map (fun x → P x) lp, List.map (fun x → P x) lp )

    let rec unicity l = match l with
```

```
   [] → []
| h :: t → if List.mem h t then unicity t else h :: (unicity t)

let legal_moves player (J (ca, m, r1, r2)) =
  let r = if player then r1 else r2 in
  if r = [] then []
  else
    let l = ref [] in
    for i = 0 to 17 do
      if value_on_case ca.(i) = 0 then l:= i :: !l
    done;
    let l2 = List.map (fun x→
      List.map (fun y→ x,y) (List.rev(unicity r)) ) !l in
    List.flatten l2
let copy_board p = Array.copy p

let  carn_copy m = Array.copy m
let rec play_piece stone l = match l with
   [] → []
| x :: q → if x=stone  then q
else x :: (play_piece stone q)

let count_case player case = match case with
  Empty → 0
| Occup (j,p) → if j = player then (int_of_piece p) else  0

let count_line player line pos =
  List.fold_left (fun x y → x +  count_case player pos.(y)) 0 line

let rec count_max n = function
    [] → 0
  | t :: q →
      if (n>0) then
        (int_of_piece t) + count_max (n-1) q
      else
        0

let rec nbr_cases_free ca  l  = match l with
   []   → 0
| t :: q → let c = ca.(t) in
  match c with
    Empty → 1 + nbr_cases_free ca q
  | _     → nbr_cases_free ca q

let a_placement i ma =
  match ma.(i) with
    None → false
  | _ → true

let which_placement i ma =
  match ma.(i) with
```

```
        None → failwith "which_placement"
      | M j  → j

    let is_filled l ca =  nbr_cases_free ca l = 0

    (* function play : arbitrates the game *)
    let play player move game =
      let (c, i) = move in
      let J (p, m, r1, r2) = game in
      let nr1,nr2 = if player then play_piece i r1,r2
      else r1,  play_piece i r2  in
      let np = copy_board p in
      let nm = carn_copy m in
      np.(c)<-Occup(player,i);  (* on play le move *)
      let lines_of_the_case = num_line_per_case.(c) in

      (* calculation of the placements of the three lines *)
      for k=0 to 2 do
        let l =  lines_of_the_case.(k) in
        if not (a_placement l nm) then (
          if is_filled vector_l.(l) np then (
            let c1 = count_line player vector_l.(l) np
            and c2 = count_line (not player) vector_l.(l) np in
            if (c1 > c2) then nm.(l) <- M player
            else  ( if c2 > c1 then nm.(l) <- M (not player)
            else nm.(l) <- M (not player ))))
      done;

      (* calculation of other placements *)
      for k=0 to 14 do
        if not (a_placement k nm ) then
          if is_filled vector_l.(k) np then failwith "player"
          else
            let c1 = count_line player  vector_l.(k) np
            and c2 = count_line (not player) vector_l.(k) np in
            let cases_free = nbr_cases_free np vector_l.(k) in
            let max1  = count_max cases_free
                (if player then nr1 else nr2)
            and max2 = count_max cases_free
                (if player then nr2 else nr1) in
            if c1 >= c2 + max2 then nm.(k) <- M player
            else if c2 >= c1 + max1 then nm.(k) <- M (not player)
      done;
      J(np,nm,nr1,nr2)
   end ;;
module Stone_rep :
  sig
    type player = bool
    and piece = | P of int
    val int_of_piece : piece -> int
    type placement = | None | M of player
    and case = | Empty | Occup of player * piece
```

```
  val value_on_case : case -> int
  type game = | J of case array * placement array * piece list * piece list
  and move = int * piece
  val lines : int list list
  val vector_l : int list array
  val lines_per_case : int list array -> int array array
  val num_line_per_case : int array array
  val lines_of_i : 'a -> 'a list list -> 'a list list
  val lines_of_cases : int list list -> int list list array
  val ldc : int list list array
  val game_start : unit -> game
  val unicity : 'a list -> 'a list
  val legal_moves : bool -> game -> (int * piece) list
  val copy_board : 'a array -> 'a array
  val carn_copy : 'a array -> 'a array
  val play_piece : 'a -> 'a list -> 'a list
  val count_case : player -> case -> int
  val count_line : player -> int list -> case array -> int
  val count_max : int -> piece list -> int
  val nbr_cases_free : case array -> int list -> int
  val a_placement : int -> placement array -> bool
  val which_placement : int -> placement array -> player
  val is_filled : int list -> case array -> bool
  val play : player -> int * piece -> game -> game
end
```

The function `play` decomposes into three stages:

1. Copying the game position and placing a move onto this position;
2. Determination of the placement of a piece on one of the three lines of the case played;
3. Treatment of the other ley-lines.

The second stage verifies that, of the three lines passing through the position of the move, none has already been won, and then checks if they are able to be won. In the latter case, it counts scores for each player and determines which strictly has the greatest score, and attributes the line to the appropriate player. In case of equality, the line goes to the most recent player's adversary. In effect, there are no lines with just one case. A filled line has at least two pieces. Thus if the player which just played has just matched the score of his adversary, he cannot expect to win the line which then goes to his adversary. If the line is not filled, it will be analyzed by "stage 3."

The third stage verifies for each line not yet attributed that it is not filled, and then checks if a player cannot be beaten by his opponent. In this case, the line is immediately given to the opponent. To perform this test, it is necessary to calculate the maximum total potential score of a player on the line (that is, by using his best pieces). If the line is still under dispute, nothing more is done.

Evaluation. The evaluation function must remain simple due to the large number of cases to deal with near the beginning of the game. The idea is not to excessively simplify the game by immediately playing the strongest pieces which would then leave the remainder of the game open for the adversary to play his strong pieces.

We will use two criteria: the number of lines won and an estimate of the potential of future moves by calculating the value of the remaining pieces. We may use the following formula for player 1:

$$score \;=\; 50 * (c_1 - c_2) + 10 * (pr_1 - pr_2)$$

where c_i is the number of lines won, and pr_i is the sum of the pieces remaining for player i.

The formula returns a positive result if the differences between won lines $(c_1 - c_2)$ and the potentials $(pr_1 - pr_2)$ turn to the advantage of player 1. We may see thus that a placement of piece 6 is not appropriate unless it provides a win of at least 2 lines. The gain of one line provides 50, while using the "6" piece costs 10×6 points, so we would thus prefer to play "1" which results in the same score, namely a loss of 10 points.

```
# module Stone_eval = struct
    open Stone_rep
    type game = Stone_rep.game

    exception Done of bool
    let moreI = 1000 and  lessI = -1000

    let nbr_lines_won  (J(ca, m,r1,r2))  =
      let c1,c2 = ref 0,ref 0 in
      for i=0 to 14 do
        if a_placement i m then if which_placement i m then incr c1 else incr c2
      done;
      !c1,!c2

    let rec nbr_points_remaining lig = match lig with
      []  → 0
    | t :: q → (int_of_piece t) + nbr_points_remaining q

    let evaluate player game  =
      let (J (ca,ma,r1,r2)) = game in
      let c1,c2  = nbr_lines_won  game in
      let pr1,pr2 = nbr_points_remaining r1, nbr_points_remaining r2 in
      match player with
        true → if c1 > 7 then moreI else 50 * (c1 - c2) + 10 * (pr1 - pr2)
      | false → if c2 > 7 then lessI else 50 * (c1 - c2) + 10 * (pr1 - pr2)

    let is_leaf player game =
      let v = evaluate player game  in
      v = moreI or v = lessI or legal_moves player game  = []
```

```
    let is_stable player game = true

    type state = G | P | N | C
    let state_of player m =
      let v = evaluate player m in
      if v = moreI then if player then G else P
      else
        if v = lessI
        then if player then P else G
        else
          if legal_moves player  m = [] then  N else C
  end;;
module Stone_eval :
  sig
    type game = Stone_rep.game
    exception Done of bool
    val moreI : int
    val lessI : int
    val nbr_lines_won : Stone_rep.game -> int * int
    val nbr_points_remaining : Stone_rep.piece list -> int
    val evaluate : bool -> Stone_rep.game -> int
    val is_leaf : bool -> Stone_rep.game -> bool
    val is_stable : 'a -> 'b -> bool
    type state = | G | P | N | C
    val state_of : bool -> Stone_rep.game -> state
  end

# module Stone_graph = struct
    open Stone_rep
    type piece = Stone_rep.piece
    type placement = Stone_rep.placement
    type case = Stone_rep.case
    type game = Stone_rep.game
    type move = Stone_rep.move

  (* brightness for a piece *)
    let brightness = 20

  (* the colors  *)
    let cBlack  = Graphics.black
    let cRed    = Graphics.rgb 165 43 24
    let cYellow = Graphics.yellow
    let cGreen  = Graphics.rgb 31 155 33 (*Graphics.green*)
    let cWhite  = Graphics.white
    let cGray   = Graphics.rgb 128 128 128
    let cBlue   = Graphics.rgb 196 139 25 (*Graphics.blue*)

  (* width and height *)
    let width = 600
    let height = 500
  (* the border at the top of the screen from which drawing begins *)
    let top_offset = 30
```

```
(* height of foundaries *)
  let bounds = 5

(* the size of the border on the left side of the virtual table *)
  let virtual_table_xoffset = 145

(* left shift for the black pieces *)
  let choice_black_offset = 40

(* left shift for the red pieces *)
  let choice_red_offset = 560

(* height of a case for the virtual table *)
  let virtual_case_size = 60

(* corresp : int*int -> int*int  *)
(* establishes a correspondence between a location in the matrix *)
(* and a position on the virtual table servant for drawing *)
  let corresp cp =
    match cp with
      0  → (4,1)
    | 1 → (6,1)
    | 2 → (3,2)
    | 3 → (5,2)
    | 4 → (7,2)
    | 5 → (2,3)
    | 6 → (4,3)
    | 7 → (6,3)
    | 8 → (8,3)
    | 9 → (1,4)
    | 10 → (3,4)
    | 11 → (5,4)
    | 12 → (7,4)
    | 13 → (9,4)
    | 14 → (2,5)
    | 15 → (4,5)
    | 16 → (6,5)
    | 17 → (8,5)
    | _ → (0,0)

  let corresp2 ((x,y) as cp) =
    match cp with
      (0,0) → 0
    |  (0,1) → 1
    |  (1,0) → 2
    |  (1,1) → 3
    |  (1,2) → 4
    |  (2,0) → 5
    |  (2,1) → 6
    |  (2,2) → 7
```

```
    |  (2,3) → 8
    |  (3,0) → 9
    |  (3,1) → 10
    |  (3,2) → 11
    |  (3,3) → 12
    |  (3,4) → 13
    |  (4,0) → 14
    |  (4,1) → 15
    |  (4,2) → 16
    |  (4,3) → 17
    |  (x,y) → print_string "Err ";
        print_int x;print_string " ";
        print_int y; print_newline() ; 0

  let col = 5
  let lig = 5

(* draw_background : unit -> unit  *)
(* draw the screen background  *)
  let draw_background () =
    Graphics.clear_graph() ;
    Graphics.set_color cBlue ;
    Graphics.fill_rect bounds bounds width (height-top_offset)

(* draw_places : unit -> unit  *)
(* draw the pieces at the start of the game *)
  let draw_places () =
    for l = 0 to 17 do
      let cp = corresp l in
      if cp <> (0,0) then
        begin
          Graphics.set_color cBlack ;
          Graphics.draw_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) (brightness+1) ;
          Graphics.set_color cGray ;
          Graphics.fill_circle
            ((fst cp)*30 + virtual_table_xoffset)
            (height - ((snd cp)*55 + 25)-50) brightness
        end

    done

(* draw_force_lines : unit -> unit  *)
(* draws ley-lines  *)
  let draw_force_lines () =
    Graphics.set_color cYellow ;
    let lst = [((2,1),(6,1)); ((1,2),(7,2)); ((0,3),(8,3));
               ((-1,4),(9,4)); ((0,5),(8,5)); ((5,0),(1,4));
               ((7,0),(2,5)); ((8,1),(4,5)); ((9,2),(6,5));
               ((10,3),(8, 5)); ((3,6),(1,4)); ((5,6),(2,3));
```

```
              ((7,6),(3,2)); ((9,6),(4,1)); ((10,5),(6,1))] in
    let rec lines l =
      match l with
        [] → ()
      | h :: t → let deb = fst h and complete = snd h in
        Graphics.moveto
          ((fst deb) * 30 + virtual_table_xoffset)
          (height -     ((snd deb) * 55 + 25) -50) ;
        Graphics.lineto
          ((fst complete) * 30 + virtual_table_xoffset)
          (height -     ((snd complete) * 55 + 25) -50) ;
        lines t
    in lines lst

(* draw_final_places : unit -> unit  *)
(* draws final cases for each ley-line *)
(* coordinates represent in the virtual array
used for positioning *)

let draw_final_places () =
  let lst = [(2,1); (1,2); (0,3); (-1,4); (0,5); (3,6); (5,6);
             (7,6); (9,6); (10,5); (10,3); (9,2); (8,1); (7,0);
             (5,0)] in
  let rec final l =
    match l with
      [] → ()
    | h :: t →  Graphics.set_color cBlack ;
        Graphics.draw_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) (brightness+1) ;
        Graphics.set_color cGreen ;
        Graphics.fill_circle
          ((fst h)*30 + virtual_table_xoffset)
          (height - ((snd h)*55 + 25)-50) brightness ;
        final t
  in final lst


(* draw_table : unit -> unit  *)
(* draws the whole game  *)
let draw_table () =
  Graphics.set_color cYellow ;
  draw_background () ;
  Graphics.set_line_width 5 ;
  draw_force_lines () ;
  Graphics.set_line_width 2 ;
  draw_places () ;
  draw_final_places ();
  Graphics.set_line_width 1

(* move -> couleur -> unit  *)
```

```
let draw_piece  player  (n_case,P cp)  = (*   (n_caOccup(c,v),cp) col =*)
  Graphics.set_color (if player then cBlack else cRed); (*col;*)
  let co = corresp n_case in
  let x = ((fst co)*30 + 145) and y = (height - ((snd co)*55 + 25)-50) in
  Graphics.fill_circle x y brightness ;
  Graphics.set_color cWhite ;
  Graphics.moveto (x - 3) (y - 3) ;
  let dummy = 5 in
  Graphics.draw_string (string_of_int cp) (*;*)
(*    print_string "---";print_int n_case; print_string " "; print_int cp ;print_newline() *)

(* conv : Graphics.status -> int  *)
(* convert a mouse click into a position on a virtual table permitting *)
(* its drawing *)
let conv st =
  let xx = st.Graphics.mouse_x and yy = st.Graphics.mouse_y in
  let y = (yy+10)/virtual_case_size - 6 in
  let dec =
    if y = ((y/2)*2) then 60 else 40 in
  let offset = match (-1*y) with
    0 → -2
  | 1 → -1
  | 2 → -1
  | 3 → 0
  | 4 → -1
  | _ → 12 in
  let x = (xx+dec)/virtual_case_size - 3 + offset in
  (-1*y, x)

(* line_number_to_aff : int -> int*int *)
(* convert a line number into a polition on the virtual table serving *)
(* for drawing *)
(* the coordinate returned corresponds to the final case for the line *)
let line_number_to_aff n =
  match n with
    0 → (2,1)
  | 1 → (1,2)
  | 2 → (0,3)
  | 3 → (-1,4)
  | 4 → (0,5)
  | 5 → (5,0)
  | 6 → (7,0)
  | 7 → (8,1)
  | 8 → (9,2)
  | 9 → (10,3)
  | 10 → (3,6)
  | 11 → (5,6)
  | 12 → (7,6)
  | 13 → (9,6)
  | 14 → (10,5)
  | _ → failwith "line" (*raise Rep.Out_of_bounds*)
```

```
  (* draw_lines_won : game -> unit  *)
  (* position a marker indicating the player which has taken the line *)
  (* this is done for all lines *)
  let  drawb  l  i =
    match l with
      None → failwith "draw"
    | M j  → let pos = line_number_to_aff i in
  (*               print_string "'''''";
  print_int i;
  print_string "---";
  Printf.printf "%d,%d\n" (fst pos) (snd pos);
*)                 Graphics.set_color (if j then cBlack else cRed);
      Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
        (height - ((snd pos)*55 + 25)-60) 20 40

  let draw_lines_won om nm  =
    for i=0 to 14 do
      if om.(i) <> nm.(i) then drawb nm.(i) i
    done
  (*********************
  let black_lines = Rep.lines_won_by_player mat Rep.Noir and
  red_lines = Rep.lines_won_by_player mat Rep.Rouge
  in
  print_string "black : "; print_int (Rep.list_size black_lines);
  print_newline () ;
  print_string "red : "; print_int (Rep.list_size red_lines);
  print_newline() ;

  let rec draw l col =
  match l with
  [] -> ()
  | h::t -> let pos = line_number_to_aff h in
  Graphics.set_color col ;
  Graphics.fill_rect ((fst pos)*30 + virtual_table_xoffset-bounds)
  (height - ((snd pos)*55 + 25)-60) 20 40 ;
  draw t col
  in draw black_lines cBlack ;
  draw red_lines cRed
  ************************************************)

  (* draw_poss : item list -> int -> unit  *)
  (* draw the pieces available for a player based on a list *)
  (* the parameter "off" indicates the position at which to place the list *)
  let draw_poss player lst off =
    let c = ref (1) in
    let rec draw l =
      match l with
        [] → ()
      | v :: t → if player  then Graphics.set_color cBlack
      else Graphics.set_color cRed;
          let x = off   and
```

```
            y = 0+(!c)*50   in
        Graphics.fill_circle x y brightness ;
        Graphics.set_color cWhite ;
        Graphics.moveto (x - 3) (y - 3) ;
        Graphics.draw_string (string_of_int v) ;
        c := !c + 1 ;
        draw t

  in draw (List.map (function P x → x) lst)

(* draw_choice : game -> unit *)
(* draw the list of pieces still available for each player *)
let draw_choice (J (ca,ma,r1,r2)) =
  Graphics.set_color cBlue ;
  Graphics.fill_rect (choice_black_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  Graphics.fill_rect (choice_red_offset-30) 10 60
    (height - (top_offset + bounds)) ;
  draw_poss true r1 choice_black_offset ;
  draw_poss false r2 choice_red_offset

(* wait_click : unit -> unit *)
(* wait for a mouse click *)
let wait_click () = Graphics.wait_next_event [Graphics.Button_down]


(* item list -> item  *)
(* return, for play, the piece chosen by the user *)
let select_pion player  lst =
  let ok = ref false and
      choice = ref 99 and
      pion = ref (P(-1))
  in
  while not !ok do
    let st = wait_click () in
    let size = List.length lst in
    let x = st.Graphics.mouse_x and y = st.Graphics.mouse_y in
    choice := (y+25)/50 - 1 ;
    if !choice <= size && ( (player && x < 65 )
                        || ( (not player) && (x > 535))) then ok := true
    else ok := false ;
    if !ok then
      try
        pion := (List.nth  lst !choice) ;
        Graphics.set_color cGreen ;
        Graphics.set_line_width 2 ;
        Graphics.draw_circle
          (if player then choice_black_offset else choice_red_offset)
          ((!choice+1)*50) (brightness + 1)
      with _  → ok := false ;
  done ;
```

```
    !pion

  (* choiceH : game -> move  *)
  (* return a move for the human player.
  return the choice of the number, the case, and the piece *)
  let rec choice player game = match game with (J(ca,ma,r1,r2)) →
    let choice = ref (P(-1))
    and c = ref (-1, P(-1)) in
    let lcl = legal_moves player game in
    while not (List.mem !c lcl) do
      print_newline();print_string "CHOICE";
      List.iter (fun (c,P p) → print_string "["; print_int c;print_string " ";
        print_int p;print_string "]")
        (legal_moves player game);
      draw_choice game;
      choice :=  select_pion player  (if player then r1 else r2) ;
  (*       print_string "choice "; print_piece !choice;*)
      c :=  (corresp2  (conv (wait_click())), !choice)
  (*       let (x,y) = !c in
   (print_string "...";print_int x; print_string " "; print_piece y;
   print_string " -> ";
   print_string "END_CHOICE";print_newline())
*)     done ;
    !c (* case, piece *)


  (* home : unit -> unit  *)
  (* place a message about the game *)
  let home () =
    Graphics.open_graph
      (" " ^ (string_of_int (width + 10)) ^ "x" ^ (string_of_int (height + 10))
       ^ "+50+50") ;
    Graphics.moveto (height / 2) (width / 2) ;
    Graphics.set_color cBlue ;
    Graphics.draw_string "Stonehenge" ;
    Graphics.set_color cBlack ;
    Graphics.moveto 2 2 ;
    Graphics.draw_string "Mixte Projets Maîtrise & DESS GLA" ;
    wait_click () ;
    Graphics.clear_graph ()

  (* exit : unit -> unit *)
  (* close everything ! *)
  let exit () =
    Graphics.close_graph ()

  (* draw_button : int -> int -> int -> int -> string -> unit  *)
  (* draw a button with a message *)
  let draw_button x y w h s =
    Graphics.set_line_width 1 ;
    Graphics.set_color cBlack ;
    Graphics.moveto x y ;
```

```
  Graphics.lineto x (y+h) ;
  Graphics.lineto (x+w) (y+h) ;
  Graphics.lineto (x+w) y ;
  Graphics.lineto x y ;
  Graphics.moveto (x+bounds) (height - (top_offset/2)) ;
  Graphics.draw_string s

(* draw_message : string -> unit  *)
(* position a message *)
let draw_message s =
  Graphics.set_color cBlack;
  Graphics.moveto 3 (height - (top_offset/2)) ;
  Graphics.draw_string s

(* erase_message : unit -> unit  *)
(* as the name indicates *)
let erase_message () =
  Graphics.set_color Graphics.white;
  Graphics.fill_rect 0 (height-top_offset+bounds) width top_offset

(* question : string -> bool  *)
(* pose the user a question, and wait for a yes/no response *)
let question s =
  let xb1 = (width/2) and xb2 = (width/2 + 30) and wb = 25 and hb = 16
  and yb = height - 20  in
  let rec attente () =
    let e = wait_click () in
    if (e.Graphics.mouse_y < (yb+hb)) & (e.Graphics.mouse_y > yb) then
      if (e.Graphics.mouse_x > xb1) & (e.Graphics.mouse_x < (xb1+wb)) then
        true
      else
        if (e.Graphics.mouse_x > xb2) & (e.Graphics.mouse_x < (xb2+wb)) then
          false
        else
          attente()
    else
      attente () in
  draw_message s;
  draw_button xb1 yb wb hb "yes";
  draw_button xb2 yb wb hb "no";
  attente()

(* q_begin : unit -> bool *)
(* Ask if the player wishes to be the first player or not *)
let q_begin () =
  let b = question "Would you like to play first ?" in
  erase_message();
  b

(* q_continue : unit -> bool *)
(* Ask if the user wishes to play the game again *)
```

```
  let q_continue () =
    let b = question "Play again ?" in
    erase_message();
    b
  (* won : unit -> unit  *)
  (* a message indicating the machine has won *)
  let won () = draw_message "I won :-)"; wait_click(); erase_message()

  (* lost : unit -> unit  *)
  (* a message indicating the machine has lost *)
  let lost () = draw_message "You won :-("; wait_click(); erase_message()

  (* nil : unit -> unit  *)
  (* a message indicating stalemate *)
  let nil () = draw_message "Stalemate"; wait_click(); erase_message()

  (* init : unit -> unit *)
  (* draw the initial game board *)
  let init () = let game = game_start () in
  draw_table () ;
  draw_choice game

  (* drawH : move -> game -> unit *)
  (* draw a piece for the human player *)
  (* let drawH cp j =  draw_piece cp cBlack ;
   draw_lines_won j
*)
  (* drawM : move -> game -> unit *)
  (* draw a piece for the machine player *)
  (* let drawM cp j =  draw_piece cp cRed ;
   draw_lines_won j
*)
  let print_placement m = match m with
    None → print_string "None "
  | M j  → print_string ("Pl "^(if j then "1 " else "2 "))

  let position player move
      (J(ca1,m1,r11,r12))
      (J(ca2,m2,r21,r22) as new_game) =
    draw_piece player move;
    draw_choice new_game;
  (*  print_string "_______OLD___________________\n";
   Array.iter print_placement m1; print_newline();
   List.iter print_piece r11; print_newline();
   List.iter print_piece r12; print_newline();
   print_string "_______NEW___________________\n";
   Array.iter print_placement m2; print_newline();
   List.iter print_piece r21; print_newline();
   List.iter print_piece r22; print_newline();
*)  draw_lines_won m1 m2

  (*
```

```
   if player then draw_piece move cBlack
   else  draw_piece move cRed
*)
  let q_player () =
    let b = question "Is there a machine playing?" in
    erase_message ();
    b
  end;;
Characters 11114-11127:
Warning: this expression should have type unit.
Characters 13197-13209:
Warning: this expression should have type unit.
Characters 13345-13357:
Warning: this expression should have type unit.
Characters 13478-13490:
Warning: this expression should have type unit.
module Stone_graph :
  sig
    type piece = Stone_rep.piece
    and placement = Stone_rep.placement
    and case = Stone_rep.case
    and game = Stone_rep.game
    and move = Stone_rep.move
    val brightness : int
    val cBlack : Graphics.color
    val cRed : Graphics.color
    val cYellow : Graphics.color
    val cGreen : Graphics.color
    val cWhite : Graphics.color
    val cGray : Graphics.color
    val cBlue : Graphics.color
    val width : int
    val height : int
    val top_offset : int
    val bounds : int
    val virtual_table_xoffset : int
    val choice_black_offset : int
    val choice_red_offset : int
    val virtual_case_size : int
    val corresp : int -> int * int
    val corresp2 : int * int -> int
    val col : int
    val lig : int
    val draw_background : unit -> unit
    val draw_places : unit -> unit
    val draw_force_lines : unit -> unit
    val draw_final_places : unit -> unit
    val draw_table : unit -> unit
    val draw_piece : bool -> int * Stone_rep.piece -> unit
    val conv : Graphics.status -> int * int
    val line_number_to_aff : int -> int * int
    val drawb : Stone_rep.placement -> int -> unit
```

```
    val draw_lines_won :
      Stone_rep.placement array -> Stone_rep.placement array -> unit
    val draw_poss : bool -> Stone_rep.piece list -> int -> unit
    val draw_choice : Stone_rep.game -> unit
    val wait_click : unit -> Graphics.status
    val select_pion : bool -> Stone_rep.piece list -> Stone_rep.piece
    val choice : bool -> Stone_rep.game -> int * Stone_rep.piece
    val home : unit -> unit
    val exit : unit -> unit
    val draw_button : int -> int -> int -> int -> string -> unit
    val draw_message : string -> unit
    val erase_message : unit -> unit
    val question : string -> bool
    val q_begin : unit -> bool
    val q_continue : unit -> bool
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val init : unit -> unit
    val print_placement : Stone_rep.placement -> unit
    val position :
      bool ->
      int * Stone_rep.piece -> Stone_rep.game -> Stone_rep.game -> unit
    val q_player : unit -> bool
  end
```

Assembly. We thus write module `Stone_graph` which describes a graphical interface compatible with signature `DISPLAY`. We construct `Stone_skeletonG` similar to `C4_skeletonG`, passing in the arguments appropriate for the Stonehenge game, applying the parametric module `FSkeleton`.

```
# module Stone_skeletonG = FSkeleton (Stone_rep)
                                     (Stone_graph)
                                     (Stone_eval)
                                     (FAlphabeta (Stone_rep) (Stone_eval)) ;;
module Stone_skeletonG :
  sig
    val depth : int ref
    exception Won
    exception Lost
    exception Nil
    val won : unit -> unit
    val lost : unit -> unit
    val nil : unit -> unit
    val again : unit -> bool
    val play_game : Stone_graph.game ref
    val exit : unit -> unit
    val home : unit -> unit
    val playH : bool -> unit -> unit
```

```
    val playM : bool -> unit -> unit
    val init : unit -> (unit -> unit) * (unit -> unit)
  end
```

We may thus construct the principal module `Stone_mainG`.

```
# module Stone_mainG = FMain(Stone_skeletonG) ;;
module Stone_mainG :
  sig
    val play_game : (unit -> 'a) * (unit -> 'b) -> unit
    val main : unit -> unit
  end
```

Launching `Stone_mainG.main ()` opens the window shown in figure 17.6. After displaying a dialogue to show who is playing, the game begins. A human player will select a piece and place it.

To Learn More

This organization of these applications involves using several parametric modules that permit direct reuse of `FAlphabeta` and `FSkeleton` for the two games we have written. With Stonehenge, some of the functions from `Stone_rep`, needed for `play`, which do not appear in `REPRESENTATION`, are used by the evaluation function. That is why the module `Stone_rep` was not closed immediately by `REPRESENTATION`. This partitioning of modules for the specific aspects of games allows incremental development without making the game schema dependencies (presented in figure 17.4) fragile.

A first enhancement involves games where given a position and a move, it is easy to determine the preceding position. In such cases, it may be more efficient to not bother making a copy of the game for function `play`, but rather to conserve a history of moves played to allow *backtracking*. This is the case for Connect 4, but not for Stonehenge.

A second improvement is to capitalize on a player's response time by evaluating future positions while the other player is selecting his next move. For this, one may use threads (see chapter 19), which allow concurrent calculation. If the player's response is one that has already been explored, the gain in time will be immediate, if not we must start again from the new position.

A third enhancement is to build and exploit dictionaries of opening moves. We have been able to do so with Stonehenge, but it is also useful for many other games where the set of legal moves to explore is particularly large and complex at the start of the game. There is much to be gained from estimating and precalculating some "best" moves from the starting positions and retaining them in some sort of database. One may add a bit of "spice" (and perhaps unpredictability) to the games by introducing an element of chance, by picking randomly from a set of moves with similar or identical values.

A fourth view is to not limit the search depth to a fixed depth value, but rather to limit the search by a calculation time period that is not to be exceeded. In this manner, the program will be able to efficiently search to deeper depths when the number of remaining moves becomes limited. This modification requires slight modification to `minmax` in order to be able to re-examine a tree to increase its depth.

A game-dependent heuristic, parameterized by `minmax`, may be to choose which branches in the search should be pursued and which may be quickly abandoned.

There are also many other games that require little more than to be implemented or reimplemented. We might cite many classic games: Checkers, Othello, Abalone, ..., but also many lesser-known games that are, nevertheless, readily playable by computer. You may find on the web various student projects including Checkers or the game Nuba.

Link: http://www.gamecabinet.com/rules/Nuba.html

Games with stochastic qualities, such as card games and dice games, necessitate a modification of the minimax-$\alpha\beta$ algorithm in order to take account of the probabilities of the selections.

We will return to the interfaces of games in chapter 21 in constructing web-based interfaces, providing without further cost the ability to return to the last move. This also allows further benefits from the modular organization that allows modifying no more than just an element, here the game state and interactions, to extend the functionality to support two player games.

Fancy Robots

The example in this section illustrates the use of objects from the graphics library. We will revisit the concepts of simple inheritance, overriding methods and dynamic dispatch. We also see how parametric classes may be profitably used.

The application recognizes two principal categories of objects: a world and robots. The world represents a state space within which the robots evolve. We will have various classes of robots, each possessing its own strategy to move around in the world.

The principle of interaction between robots and the world here is extremely simple. The world is completely in control of the game: it asks, turn by turn, each of the robots if they know their next position. Each robot determines its next position fairly blindly. They do not know the geometry of the world, nor what other robots may be present. If the position requested by a robot is legal and open, the world will place the robot at that position.

The world displays the evolution of the robots via an interface. The (relative) complexity of the conception and development of this example is in the always-necessary separation between a behavior (here the evolution of the robots) and its interface (here the tracking of this evolution).

General Description The application is developed in two stages.

1. A group of definitions providing pure calculation classes for the world and for the diverse set of envisaged robots.
2. A group of definitions using the preceding set, adding whatever is necessary to add in an interface.
   We provide two examples of such interfaces: a rudimentary text-based interface, and a more elaborate one using a graphical library.

In the first section, we provide the *abstract* definitions for the robots. Then (page 553), we provide the pure abstract definition for the world. In the next section (page 554), we introduce the text interface for the robots, and in the fourth section (page 556), the interface for the world. On page 559 we introduce a graphical interface for the robots and finally (page 562) we define a world for the graphical interface.

"Abstract" Robots

The first thing to do is to examine robots abstractly, independent of any consideration of the environment in which they will move, that is to say, the interface that displays them.

```
# class virtual robot (i0:int) (j0:int) =
    object
      val mutable i = i0
      val mutable j = j0
      method get_pos = (i,j)
      method set_pos (i', j') = i <- i'; j <- j'
      method virtual next_pos : unit → (int * int)
    end ;;
```

A robot is an entity which knows, or believes it knows, its position (`i` and `j`), is capable of communicating that position to a requester (`get_pos`), is able to modify this knowledge if it knows precisely where it should be (`set_pos`) and may decide to move towards a new position (`next_pos`).

To improve the readability of the program, we define relative movements based on absolute directions:

```
# type dir = North | East | South | West | Nothing  ;;

# let walk (x,y) = function
      North → (x,y+1) | South → (x,y-1)
    | West  → (x-1,y) | East  → (x+1,y)
    | Nothing → (x,y) ;;
val walk : int * int -> dir -> int * int = <fun>

# let turn_right = function
```

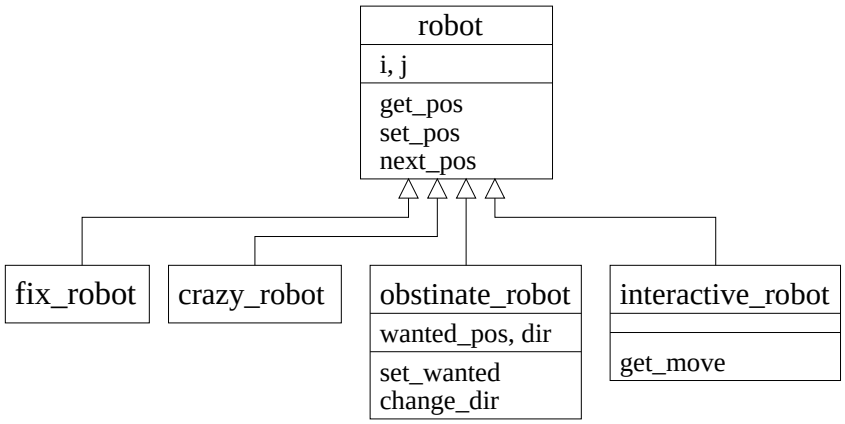


Figure 17.9: Hierarchy of pure robot classes

```
    North → East | East → South | South → West | West → North | x → x ;;
val turn_right : dir -> dir = <fun>
```

The schema is shown by the virtual class `robots` from which we define four distinct species of robots (see figure 17.9) to more precisely see their manner of motion:

- Fixed robots which never move:

```
# class fix_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos() = (i,j)
    end ;;
```

- Crazy robots which move at random:

```
# class crazy_robot i0 j0 =
    object
      inherit robot i0 j0
      method next_pos () = ( i+(Random.int 3)-1 , j+(Random.int 3)-1 )
    end ;;
```

- Obstinate robots which keep trying to advance in one direction whenever they are able to do so,

```
# class obstinate_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      val mutable wanted_pos = (i0,j0)
      val mutable dir = West

      method private set_wanted_pos d = wanted_pos <- walk (i,j) d
      method private change_dir = dir <- turn_right dir
```

```
      method next_pos () = if (i,j) = wanted_pos
        then let np = walk (i,j) dir in ( wanted_pos <- np ; np )
        else ( self#change_dir ; wanted_pos <- (i,j) ; (i,j) )
    end ;;
```

- Interactive robots which obey the commands of an exterior operator:

```
# class virtual interactive_robot i0 j0 =
    object(self)
      inherit robot i0 j0
      method virtual private get_move : unit → dir
      method next_pos () = walk (i,j) (self#get_move ())
     end ;;
```

The case of the interactive robot is different from the others in that its behavior is controlled by an interface that permits communicating orders to it. To deal with this, we provide a virtual method to communicate this order. As a consequence, the class `interactive_robot` remains abstract.

Note that not only do the four specialized robot classes inherit from class `robot`, but also any others that have the same type. In effect, the only methods that we have added are the private methods that therefore do not appear in the type signatures of the instances of these classes (see page 449). This property is indispensable if we wish to consider all the robots to be objects of the same type.

Pure World

A *pure* world is a world that is independent of an interface. It is understood as the state space of positions which a robot may occupy. It takes the form of a grid of size `l` × `h`, with a method `is_legal` to assure that a coordinate is a valid position in the world, and a method `is_free` indicates whether or not a robot occupies a given position.

In practice, a world manages the list of `robots` present on its surface while a method, `add`, allows new robots to enter the world.

Finally, a world is made visible by the method `run`, allowing the world to come to life.

```
# class virtual ['robot_type] world (l0:int) (h0:int) =
    object(self)
      val l = l0
      val h = h0
      val mutable robots =  ( [] : 'robot_type list )
      method add r = robots <- r :: robots
      method is_free p = List.for_all (fun r → r#get_pos <> p)  robots
      method virtual is_legal : (int * int) → bool

      method private run_robot r =
        let p = r#next_pos ()
```

```
        in if (self#is_legal p) & (self#is_free p) then r#set_pos p

      method run () =
        while true do List.iter (function r → self#run_robot r) robots done
    end ;;
class virtual ['a] world :
  int ->
  int ->
  object
    constraint 'a =
      < get_pos : int * int; next_pos : unit -> int * int;
        set_pos : int * int -> unit; .. >
    val h : int
    val l : int
    val mutable robots : 'a list
    method add : 'a -> unit
    method is_free : int * int -> bool
    method virtual is_legal : int * int -> bool
    method run : unit -> unit
    method private run_robot : 'a -> unit
  end
```

The Objective Caml type system does not permit leaving the types of robots undetermined (see page 460). To resolve this problem, we might consider restraining the type to those of the class `robot`. But that would forbid populating a world with objects other than those having exactly the same type as `robot`. As a result, we have instead decided to parameterize `world` with the type of the robots that populate it. We may thereby instantiate this type parameter with textual robots or graphical robots.

Textual Robots

Text Objects To obtain robots controllable via a textual interface, we define a class of text objects (`txt_object`).

```
# class txt_object (s0:string) =
    object
      val name = s0
       method get_name = name
    end ;;
```

An Interface Class: Abstract Textual Robots By double inheritance from `robots` and `txt_object`, we obtain the abstract class `txt_robot` of textual robots.

```
# class virtual txt_robot i0 j0 =
    object
```

```
      inherit robot i0 j0
      inherit txt_object "Anonymous"
   end ;;
class virtual txt_robot :
  int ->
  int ->
  object
    val mutable i : int
    val mutable j : int
    val name : string
    method get_name : string
    method get_pos : int * int
    method virtual next_pos : unit -> int * int
    method set_pos : int * int -> unit
  end
```

This class defines a world with a textual interface (see page 556). The inhabitants of this world will not be objects of `txt_robot` (since this class is abstract) nor inheritors of this class. The class `txt_robot` is, in a way, an *interface classe* permitting the compiler to identify the method types (calculations and interfaces) of the inhabitants of the text interface world. The use of such a specification class provides the separation we wish to maintain between calculations and interface.

Concrete Text Robots These are simply obtained via double inheritance; figure 17.10 shows the hierarchy of classes.

```
# class fix_txt_robot i0 j0 =
    object
      inherit fix_robot i0 j0
      inherit txt_object "Fix robot"
    end ;;

# class crazy_txt_robot i0 j0 =
    object
      inherit crazy_robot i0 j0
      inherit txt_object "Crazy robot"
    end ;;

# class obstinate_txt_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit txt_object "Obstinate robot"
    end ;;
```

The interactive robots require, for a workable implementation, defining their method of interacting with the user.

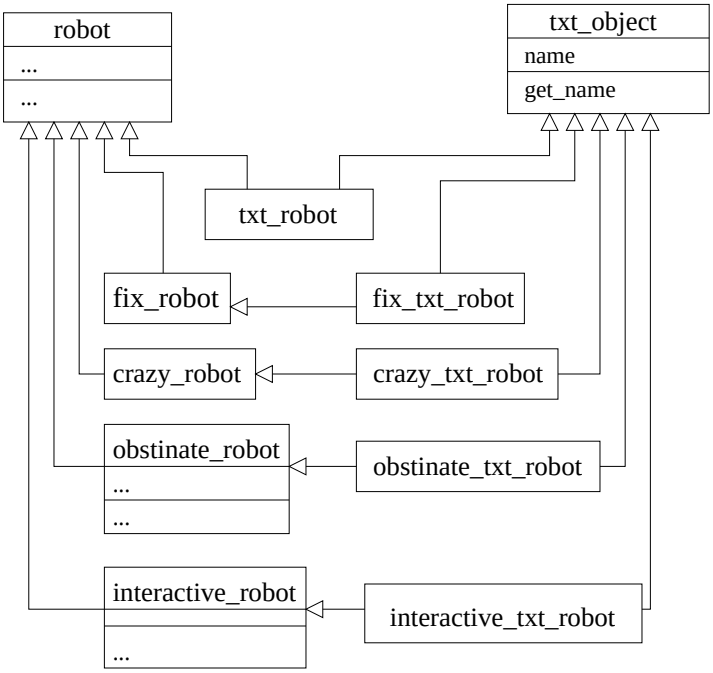


Figure 17.10: Hierarchy of classes for text mode robots

```
# class interactive_txt_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit txt_object "Interactive robot"
      method private get_move () =
        print_string "Which dir : (n)orth (e)ast (s)outh (w)est ? ";
        match read_line() with
            "n" → North  | "s" → South
          | "e" → East   | "w" → West
          |  _  → Nothing
    end ;;
```

Textual World

The text interface world is derived from the pure world by:

1. Inheritance from the generic class `world` by instantiating its type parameter with the class specified by `txt_robot`, and
2. Redefinition of the method `run` to include the different textual methods.

```
# class virtual txt_world (l0:int) (h0:int) =
    object(self)
      inherit [txt_robot] world l0 h0 as super

      method private display_robot_pos r =
        let (i,j) = r#get_pos in Printf.printf "(%d,%d)" i j

      method private run_robot r =
        let p = r#next_pos ()
        in if (self#is_legal p) & (self#is_free p)
          then
            begin
              Printf.printf "%s is moving from " r#get_name ;
              self#display_robot_pos r ;
              print_string " to " ;
              r#set_pos p;
              self#display_robot_pos r ;
            end
          else
            begin
              Printf.printf "%s is staying at " r#get_name ;
              self#display_robot_pos r
            end ;
          print_newline () ;
          print_string"next - ";
          ignore (read_line())

      method run () =
        let print_robot r =
          Printf.printf "%s is at " r#get_name ;
          self#display_robot_pos r ;
          print_newline ()
        in
          print_string "Initial state :\n";
          List.iter print_robot robots;
          print_string "Running :\n";
          super#run()  (* 1 *)
    end ;;
```

We direct the reader's attention to the call to `run` of the ancestor class (this method call is marked `(* 1 *)` in the code) in the redefinition of the same method. There we have an illustration of the two possible types of method dispatch: static or dynamic (see page 446). The call to *super#run* is static. This is why we name the superclass: to be able to call the methods when they are redefined. On the other hand, in *super#run* we find a call to *self#run_robot*. This is a dynamic dispatch; the method defined in class `txt_world` is executed, not that of `world`. Were the method from `world` executed, nothing would be displayed, and the method in `txt_world` would remain useless.

The planar rectangular text world is obtained by implementing the final method that still remains abstract: `is_legal`.

```
# class closed_txt_world l0 h0 =
    object(self)
      inherit txt_world l0 h0
      method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
    end ;;
```

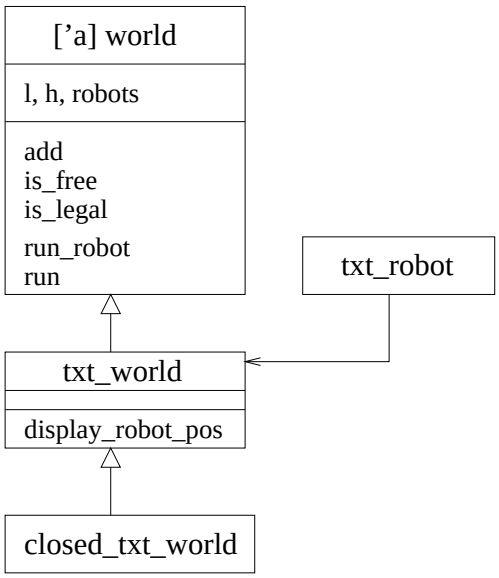


Figure 17.11: Hierarchy of classes in the textual planar rectangular world

We may proceed with a small essay in typing:

```
let w = new closed_txt_world 5 5
and r1 = new fix_txt_robot 3 3
and r2 = new crazy_txt_robot 2 2
and r3 = new obstinate_txt_robot 1 1
and r4 = new interactive_txt_robot 0 0
in w#add r1; w#add r2; w#add r3; w#add r4; w#run () ;;
```

We may skip, for the moment, the implementation of a graphical interface for our world of robots. In due course, we will obtain an application having an appearance like figure 17.12.

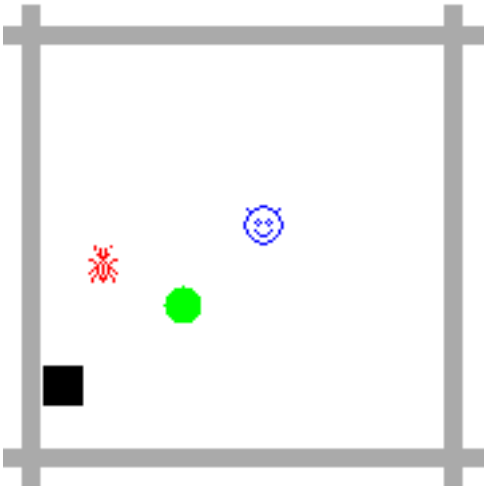


Figure 17.12: The graphical world of robots

Graphical Robots

We may implement robots in a graphical mode by following the same approach as with the text mode:

1. define a generic graphical object,
2. define an abstract class of graphical robots by double inheritance from robots and graphical objects (analogous to the interface class of page 554),
3. define, through double inheritance, the particular behavior of robots.

Generic Graphical Objects

A simple graphical object is an object possessing a `display` method which takes, as arguments, the coordinates of a pixel and displays it.

```
# class virtual graph_object =
    object
      method virtual display : int → int → unit
    end ;;
```

From this specification, it would be possible to implement graphical objects with extremely complex behavior. We will content ourselves for now with a class `graph_item`, displaying a bitmap that serves to represent the object.

```
# class graph_item x y im =
    object (self)
```

```
      val size_box_x = x
      val size_box_y = y
      val bitmap = im
      val mutable last = None

      method private erase = match last with
      Some (x,y,img) → Graphics.draw_image img x y
        | None → ()

      method private draw i j = Graphics.draw_image bitmap i j
      method private keep i j =
        last <- Some (i,j,Graphics.get_image i j size_box_x size_box_y) ;

      method display i j = match last with
      Some (x,y,img) → if x<>i || y<>j
        then ( self#erase ; self#keep i j ; self#draw i j )
        | None → ( self#keep i j ; self#draw i j )
    end ;;
```

An object of graph_item stores the portion of the image upon which it is drawn in order to restore it in subsequent redraws. In addition, if the image has not been moved, it will not be redrawn.

```
# let foo_bitmap =  [|[| Graphics.black |]|] ;;
# class square_item x col =
    object
      inherit graph_item x x (Graphics.make_image foo_bitmap)
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_rect (i+1) (j+1) (x-2) (x-2)
    end ;;

# class disk_item r col =
    object
      inherit graph_item (2*r) (2*r) (Graphics.make_image foo_bitmap)
      method private draw i j =
        Graphics.set_color col ;
        Graphics.fill_circle (i+r) (j+r) (r-2)
    end ;;

# class file_bitmap_item name =
    let ch = open_in name
    in let x = Marshal.from_channel ch
    in let y = Marshal.from_channel ch
    in let im = Marshal.from_channel ch
    in let () = close_in ch
    in object
    inherit graph_item x y (Graphics.make_image im)
       end ;;
```

We specialize the `graph_item` with instances of crosses, disks, and other bitmaps, read from a file.

The abstract graphical robot is both a robot and a graphical object.

```
# class virtual graph_robot i0 j0 =
    object
      inherit robot i0 j0
      inherit graph_object
    end ;;
```

Graphical robots that are fixed, crazy, and obstinate are specialized graphical objects.

```
# class fix_graph_robot i0 j0 =
    object
      inherit fix_robot i0 j0
      inherit disk_item 7 Graphics.green
    end ;;

# class crazy_graph_robot i0 j0 =
    object
      inherit crazy_robot i0 j0
      inherit file_bitmap_item "crazy_bitmap"
     end ;;

# class obstinate_graph_robot i0 j0 =
    object
      inherit obstinate_robot i0 j0
      inherit square_item 15 Graphics.black
    end ;;
```

The interactive graphical robot uses the primitives `key_pressed` and `read_key` of module `Graphics` to determine its next move. We again see the key presses `8`, `6`, `2` and `4` on the numeric keypad (`NumLock` button active). In this manner, the user is not obliged to provide direction at each step in the simulation.

```
# class interactive_graph_robot i0 j0 =
    object
      inherit interactive_robot i0 j0
      inherit file_bitmap_item "interactive_bitmap"
      method private get_move () =
        if not (Graphics.key_pressed ()) then Nothing
        else match Graphics.read_key() with
```

```
        '8' → North | '2' → South | '4' → West  | '6' → East | _ → Nothing
    end ;;
```

Graphical World

We obtain a world with a graphical interface by inheriting from the pure world, instantiating the parameter `'a_robot` with the graphical robot abstract class `graph_robot`. As with the text mode world, the graphical world provides its own method, `run_robot`, to implement the robot's behavior as well as the general activation method `run`.

```
# let delay x = let t = Sys.time () in while (Sys.time ()) -. t < x do () done ;;

# class virtual graph_world l0 h0 =
    object(self)
      inherit [graph_robot] world l0 h0 as super
      initializer
        let gl = (l+2)*15 and  gh = (h+2)*15 and lw=7 and cw=7
        in Graphics.open_graph (" "^(string_of_int gl)^"x"^(string_of_int gh)) ;
           Graphics.set_color (Graphics.rgb 170 170 170) ;
           Graphics.fill_rect 0 lw gl lw ;
           Graphics.fill_rect (gl-2*lw) 0 lw gh ;
           Graphics.fill_rect 0 (gh-2*cw) gl cw ;
           Graphics.fill_rect lw 0 lw gh

      method run_robot r = let p = r#next_pos ()
                           in delay 0.001 ;
                              if (self#is_legal p) & (self#is_free p)
                              then ( r#set_pos p ; self#display_robot r)

      method display_robot r = let (i,j) = r#get_pos
                               in r#display (i*15+15) (j*15+15)

      method run() = List.iter self#display_robot robots ;
                     super#run()
    end ;;
```

Note that the graphical window is created at the time that an object of this class is initialized.

The rectangular planar graphical world is obtained in much the same manner as with the rectangular planar textual world.

```
# class closed_graph_world l0 h0 =
    object(self)
      inherit graph_world l0 h0
```

```
      method is_legal (i,j) = (0<=i) & (i<l) & (0<=j) & (j<h)
    end ;;
class closed_graph_world :
  int ->
  int ->
  object
    val h : int
    val l : int
    val mutable robots : graph_robot list
    method add : graph_robot -> unit
    method display_robot : graph_robot -> unit
    method is_free : int * int -> bool
    method is_legal : int * int -> bool
    method run : unit -> unit
    method run_robot : graph_robot -> unit
  end
```

We may then test the graphical application by typing in:

```
let w = new closed_graph_world 10 10 ;;
w#add (new fix_graph_robot 3 3) ;;
w#add (new crazy_graph_robot 2 2) ;;
w#add (new obstinate_graph_robot 1 1) ;;
w#add (new interactive_graph_robot 5 5) ;;
w#run () ;;
```

To Learn More

The implementation of the method `run_robot` in different worlds suggests that the robots are potentially able to move to any point on the world the moment it is empty and legal. Unfortunately, nothing prevents a robot from modifying its position arbitrarily; the world cannot prevent it. One remedy would consist of having robot positions being controlled by the world; when a robot attempts to move, the world verifies not only that the new position is legal, but also that it constitutes an authorized move. In that case, the robot must be capable of asking the world its actual position, with the result that the robot class must become dependent on the world's class. The robot class would take, as a type parameter, the world class.

This modification permits defining robots capable of querying the world in which they run, thus behaving as dependents of the world. We may then implement robots which follow or avoid other robots, try to block them, and so forth.

Another extension would be to permit robots to communicate with one another, exchanging information, perhaps constituting themselves into teams of robots.

The chapters of the next section allow making execution of robots independent from one another: by making use of `Threads` (see page 599), each may execute as a distinct

process. They may profit from the possibilities of distributed computing (see 623) where the robots become *clients* executing on remote machines that announce their movements or request other information from a world that behaves as a *server*. This problem is dealt with on page 656.

Part IV

Concurrency and distribution

The fourth part introduces the concepts of parallel programming and presents models for shared and distributed memory. It is not necessary to have access to a parallel supercomputer to express concurrent algorithms or to implement distributed applications. In this preamble we define the different terms used in the following chapters.

In sequential programming, one instruction is executed after another. This is called causal dependency. Sequential programs have the property of being deterministic. For the same input, one program will always terminate or never terminate. In the case of termination, always the same result will be produced. Determinism implies that the same circumstances will always lead to the same effects. The only exceptions, which we have already met in Objective Caml, are provided by functions taking external information as input, such as the function `Sys.time`.

In parallel programming, a program is split into several active processes. Each process is sequential, but several instructions, belonging to different processes, are executed in parallel, "at the same time." The sequence is transformed into concurrency. This is called causal independence. The same circumstances may lead to different, mutually exclusive effects (only one effect is produced). An immediate consequence is the loss of determinism: the same program with the same input may or may not terminate. In the case of termination different results may be produced.

In order to control the execution of a parallel program, it is necessary to introduce two new notions:

- synchronization, which introduces a conditional wait to several processes;
- communication through the passing of messages between processes.

From the point of view of causality, synchronization assures that several independent circumstances have to be reproduced before an effect may take place. Communications have a temporal constraint: a message can not be received before it is sent. Communication can occur in different variants: communication directed from one process to one other (point-to-point) or as distribution (one-to-all, or all-to-all).

The two models of parallel programming described by figure 17.13 differ in execution control by the forms of synchronization and communication.

Each process P_i corresponds to a sequential process. The set of these processes, interacting via shared memory (**M**) or via a *medium*, constitutes a parallel application.

Shared Memory Model Communication is implicit in the shared memory model. An information is given through writing into a zone of the shared memory. It is received when another process reads this zone. The synchronization, in contrast, has to be explicit. Constructs of mutual exclusion and waiting conditions are used.

This model is used when shared resources are used in a concurrent way. The construction of operating systems can be cited as an example.

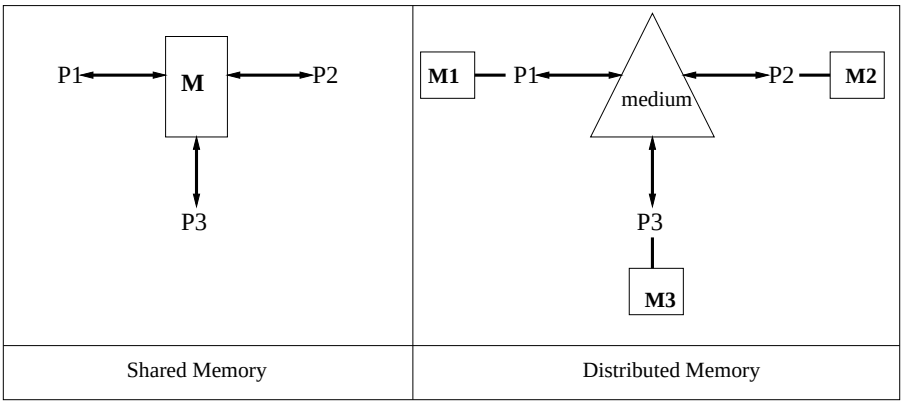


Figure 17.13: Models of parallelism

Distributed Memory Model In this model each sequential process P_i has a private memory M_i, to which no other process has access. The processes have to communicate in order to transmit information through a medium. The difficulties in this model arise from the implementation of the medium. The programs which care for this are called *protocols*.

Protocols are organized in layers. The higher-level protocols implement more elaborate services, using the lower-level services.

There exist several types of communication. They depend on the capability of the medium to store information and of the blocking, respectively non-blocking character of sender and receiver. We talk about synchronous communication when the transfer of information is not possible before a global synchronization between sender and receiver takes place. In his case both sender and receiver may be blocked.

If the medium has the storage capabilities, it can store messages for a later transmission. Therefore the communication can be asynchronous and non-blocking. It may be necessary to indicate the storage capacity of the medium, the order of transmission, the delay and the reliability of transmissions.

Finally, if the transmission is non-blocking with a medium not able to store messages, a volatile communication results: only the receiving processes which are ready will receive the sent message, which is lost for the other processes.

In the model of distributed memory the communication is explicit, but the synchronization is implicit (Synchronization is produced by communication). It is dual to the model of shared memory.

Physical and Logical Parallelism The model of distributed memory is valid in the case of physical and logical parallelism. Physical parallelism refers for example to a computer network. Examples for logical parallelism are Unix processes communicating

via *pipes*, or lightweight processes communicating via channels. There are no common global values known by all proceses like, for example, a global clock.

The model of distributed memory is closer to physical parallelism, where there is no effectively shared memory. Nevertheless, shared memory can be simulated across a computer nerwork.

The fourth part will show how to construct parallel applications with Objective Caml using the two presented models. It relies on the `Unix` library, which interfaces Unix system calls to Objective Caml, and on the `Thread` library, which implements lightweight processes. A major part of the `Unix` library is ported to Windows, especially the functions on file descriptors. These are used to read and to write on files, but also for communication *pipes* and for *sockets* of computer networks.

Chapter 18 describes essential concepts of the `Unix` library. It concentrates on the communication of a process with it's exterior and with other processes. The notion of process in this chapter is that of a "heavyweight process" as in Unix. They are created by the `fork` system call which duplicates the execution context and the memory for the data, producing a chain of processes. The interaction between processes is implemented by signals or by communication pipes.

Chapter 19 concentrates on the notion of lightweight processes of the `Thread` library. In contrast to the heavy processes mentioned before, they duplicate nothing but the execution context of an existing process. The memory is shared between the creator and the *thread*. Depending on the programming style, the light Objective Caml processes permit to use the parallelism model of shared memory (imperative style) or the model of separated memory (purely functional style). The `Thread` library contains several modules allowing to start and stop *threads*, to manage locks for mutual exclusion, to wait for a condition and to communicate between *threads* via channels. In this model, there is no gain in execution time, not even for multi-processor machines. But the formulation of parallel algorithms is made easier.

Chapter 20 is devoted to the construction of distributed Internet applications. The Internet is presented from the point of view of low-level protocols. With the help of communication sockets several processes running on different machines are able to communicate with each other. The communication through sockets is an asynchronous point-to-point communication. The role of the different processes taking part in the communication of a distributed application is in general asymmetrical. This is the case for client-server architectures. The server is a process accepting requests and trying to respond. The other process, the client, sends a request to the server and waits for a response. Many services accessible in the Internet follow this architecture.

Chapter 21 presents a library and two complete applications. The library allows to define the communication between clients and servers starting from a given protocol. The first application revisits the robots of chapter 17 to give a distributed version. The second application constructs an HTTP server to manage a request form taking up again the management of associations presented in chapter 6.

18

Communication and Processes

This chapter approaches two important aspects of the interface between a programming language and the operating system: communication and processes. The `Sys` module presented in chapter 8 has already shown how to pass values to a program and how to start a program from another one. The goal of this chapter is to discuss the notions of processes and communication between processes.

The term "process" is used for an executing program. Processes are the main components of a parallel application. We introduce processes in the classical way originating from the Unix system. In this context a process is created by another process, establishing a parent-child relationship between them. This relationship allows the parent to wait for the child to terminate, as well as to set up privileged communications between the two. The underlying model of parallelism is that of distributed memory.

The term "communication" covers three aspects:

- input and output via *file descriptors*. The notion of file descriptors under Unix has a much broader meaning than the simple reading or writing of data from or to a storage medium. We will see this in chapter 20, where programs running on different machines are communicating via such descriptors;
- the use of *pipes* between processes which allow the exchange of data using the principle of waiting queues;
- the generation and handling of *signals*, which allow a simple interaction between processes.

The functions presented in this chapter are similar to those in the `Unix` module which accompanies the Objective Caml distribution. The terminology and the notions come from the Unix world. But many of the functions of this module can also be used under Windows. Later we will indicate the applicability of the presented functions.

Chapter Overview

The first section indicates how to use the `Unix` module. We will talk about the handling of errors specific to that module and about the portability of system calls to Windows.

The second section presents file descriptors in the Unix sense, and their use for input and output operations of a lower level than those provided by the preloaded module `Pervasives`.

Processes are introduced in the third section. We talk about their creation, their disappearance and about the way in which all processes support their descendence relation in the Unix model.

The fourth section describes the basic means of communications between processes: pipes and signals.

The two last sections will be continued in chapters 19 and 20 by the presentation of lightweight processes and sockets.

The `Unix` Module

This module contains the interfaces to the most important Unix library functions. Much of this module has been ported to Windows and can be used there. Whenever necessary we will indicate the restrictions in the use of the presented functions. A table resuming the restrictions is given by figure 18.1.

The `Unix` library belongs to the Objective Caml non-standard libraries which have to be bound by the `-custom` compiler command (see chapter 7, page 197). Depending on the desired form of the program, one of the following commands is used under Unix to produce bytecode, native code or an interaction loop:

```
$ ocamlc -custom unix.cma  fichiers.ml -cclib -lunix
$ ocamlopt unix.cma fichiers.ml -cclib -lunix
$ ocamlmktop -custom -o unixtop unix.cma -cclib -lunix
```

The purpose of constructing an interaction loop (of which the name will be `unixtop`) is to support an incremental development style. Each function can be compiled quickly from its type declaration. It is also possible to execute functional tests.

Depending on the version of Unix in use, the system library may not be located at the default place. If necessary the access path of the libraries may be indicated with the option `-ccopt` (see chapter 7).

Under Windows the commands to compile become:

```
$ ocamlc -custom unix.cma  fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlopt unix.cma fichiers.ml %CAMLLIB%\libunix.lib wsock32.lib
$ ocamlmktop -custom -o unixtop.exe unix.cma %CAMLLIB%\libunix.lib wsock32.lib
```

The name of the obtained interaction loop is `unixtop.exe`.

Error Handling

Errors produced by system calls throw `Unix_error` exceptions, which can be handled by the Objective Caml program. Such errors contain three arguments: a value of type `Unix.error` which can be transformed into a character string by the function `error_message`, a string containing the name of the function producing the error and optionally, a string containing the argument of the function when the argument is of type *string*.

It is possible to define a generic calling function with error treatment:

```
# let wrap_unix funct arg =
    try (funct arg) with
     Unix.Unix_error (e,fm,argm) →
       Printf.printf "%s %s %s" (Unix.error_message e) fm argm ;;
val wrap_unix : ('a -> unit) -> 'a -> unit = <fun>
```

The function `wrap_unix` takes a function and its argument, and applies one to the other. If a Unix error occurs, an explaining message is printed. An equivalent function is defined in the `Unix` module:

```
# Unix.handle_unix_error ;;
- : ('a -> 'b) -> 'a -> 'b = <fun>
```

Portability of System Calls

Figure 18.1 indicates which of the communication and process handling functions presented in this chapter are accessible under Windows. The main shortcoming is the lack of the two functions `fork` and `kill` to create new processes and to send signals.

Furthermore, the function `wait` waiting for the end of a child process is not implemented, because `fork` is not.

File Descriptors

In chapter 3 we have seen functions from the standard module `Pervasives`. These functions allow us to access files via input / output channels. There is also a lower-level way to access files, using their descriptors.

A file descriptor is an abstract value of type `Unix.file_descr`, containing information necessary to use a file: a pointer to the file, the access rights, the access modes (read or write), the current position in the file, etc.

Three descriptors are predefined. They correspond to standard input, standard output, and standard error.

Fonction	Unix	Windows	Comment
`openfile`	×	×	
`close`	×	×	
`dup`	×	×	
`dup2`	×	×	
`read`	×	×	
`write`	×	×	
`lseek`	×	×	
`execv`	×	×	
`execve`	×	×	
`execvp`	×	×	
`execvpe`	×	×	
`fork`	×		use `create_process`
`getpid`	×	×	
`sleep`	×	×	
`wait`	×		
`waitpid`	×	×	only for a given number of processes
`create_process`	×	×	
`create_process_env`	×	×	
`kill`	×		
`pipe`	×	×	
`mkfifo`	×		
`open_process`	×		use the interpretation of `/bin/sh` commands
`close_process`	×		

Figure 18.1: Portability of the module `Unix` functions used in this chapter.

```
# ( Unix.stdin , Unix.stdout , Unix.stderr ) ;;
- : Unix.file_descr * Unix.file_descr * Unix.file_descr =
<abstr>, <abstr>, <abstr>
```

Be careful not to confuse them with the corresponding input / output channels:

```
# ( Pervasives.stdin , Pervasives.stdout , Pervasives.stderr ) ;;
- : in_channel * out_channel * out_channel = <abstr>, <abstr>, <abstr>
```

The conversion functions between channels and file descriptors are described at page 577.

File Access Rights. Under Unix each file has an associated owner and group. The rights to read, write and execute are attached to each file according to three categories of users: the owner of a file, the members of the file's group[1] and all other users.

The access rights of a file are represented by 9 bits divided into three groups of three bits each. The first group represents the rights of the owner, the second the rights of the members of the owner's group, and the last the rights of all other users. In each group of three bits, the first bit represents the right to read, the second bit the right to write and the third bit the right to execute. It is common to abbreviate these three rights by the letters `r`, `w` and `x`. The absence of the rights is represented in each case by a dash (`-`). For exampple, the right to read for all and the right to write only for the owner is written as `rw-r--r--`. This corresponds to the integer `420` (which is the binary number `0b110100100`). Frequently the more comfortable octal notation `0o644` is used. These file access rights are not used under Windows.

REVIEWER'S QUESTION: IS THIS STILL TRUE UNDER WIN2K?

File Manipulation

Opening a file. Opening a file associates the file to a file descriptor. Depending on the intended use of the file there are several modes to open a file. Each mode corresponds to a value of type *open_flag* described by figure 18.2.

`O_RDONLY`	read only
`O_WRONLY`	write only
`O_RDWR`	reading and writing
`O_NONBLOCK`	non-blocking opening
`O_APPEND`	appending at the end of the file
`O_CREAT`	create a new file if it does not exist
`O_TRUNC`	set the file to 0 if it exists
`O_EXCL`	chancel, if the file already exists

Figure 18.2: Values of type *open_flag*.

These modes can be combined. In consequence, the function `openfile` takes as argument a list of values of type *open_flag*.

```
# Unix.openfile ;;
- : string -> Unix.open_flag list -> Unix.file_perm -> Unix.file_descr =
```

1. Under Unix each user belongs to one or more user groups, which allows the organization of their rights.

```
<fun>
```
The first argument is the name of the file. The last is an integer[2] coding the rights to attach to the file in the case of creation.

Here is an example of how to open a file for reading, or to create it with the rights `rw-r--r--` if it does not exist:
```
# let file = Unix.openfile "test.dat" [Unix.O_RDWR; Unix.O_CREAT] 0o644 ;;
val file : Unix.file_descr = <abstr>
```

Closing a file. The function `Unix.close` closes a file. It is applied to the descriptor of the file to close.
```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
# Unix.close file ;;
- : unit = ()
```

Redirecting file descriptors. It is possible to attach several file descriptors to one input / output. If there is only one file descriptor available and another one is desired we can use:
```
# Unix.dup ;;
- : Unix.file_descr -> Unix.file_descr = <fun>
```

If we have two file descriptors and we want to assign to the second the input / output of the first, we can use the function:
```
# Unix.dup2 ;;
- : Unix.file_descr -> Unix.file_descr -> unit = <fun>
```

For example, the error output can be directed to a file in the following way:
```
# let error_output = Unix.openfile "err.log" [Unix.O_WRONLY;Unix.O_CREAT] 0o644 ;;
val error_output : Unix.file_descr = <abstr>
# Unix.dup2 Unix.stderr error_output ;;
- : unit = ()
```
Data written to the standard error output will now be directed to the file `err.log`.

Input / Output on Files

The functions to read and to write to a file `Unix.read` and `Unix.write` use a character string as medium between the file and the Objective Caml program.
```
# Unix.read ;;
- : Unix.file_descr -> string -> int -> int -> int = <fun>
# Unix.write ;;
```

2. The type *file_perm* is an alias for the type *int*.

```
- : Unix.file_descr -> string -> int -> int -> int = <fun>
```

In addition to the file descriptor and the string the functions take two integers as arguments. One is the index of the first character and the other the number of characters to read or to write. The returned integer is the number of characters effectively read or written.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let fl = Unix.openfile "file" mode 0o644 in
  let str = "012345678901234565789" in
  let n = Unix.write fl str 4 5
  in Printf.printf "We wrote %s to the file\n" (String.sub str 4 n) ;
     Unix.close fl ;;
We wrote 45678 to the file
- : unit = ()
```

Reading a file works the same way:

```
# let fl = Unix.openfile "file" [Unix.O_RDONLY] 0o644 in
  let str = String.make 20 '.' in
  let n = Unix.read fl str 2 10 in
   Printf.printf "We read  %d characters" n;
   Printf.printf " and got the string %s\n" str;
   Unix.close fl ;;
We read  5 characters and got the string ..45678.............
- : unit = ()
```

Access to a file always takes place at the current position of its descriptor. The current position can be modified by the function:

```
# Unix.lseek ;;
- : Unix.file_descr -> int -> Unix.seek_command -> int = <fun>
```

The first argument is the file descriptor. The second specifies the displacement as number of characters. The third argument is of type *Unix.seek_command* and indicates the origin of the displacement. The third argument may take one of three posssible values:

- SEEK_SET: relative to the beginning of the file,
- SEEK_CUR: relative to the current position,
- SEEK_END: relative to the end of the file.

A function call with an erronous position will either raise an exception or return a value equal to 0.

Input / output channels. The Unix module provides conversion functions between file descriptors and the input / output channels of module Pervasives:

```
# Unix.in_channel_of_descr  ;;
- : Unix.file_descr -> in_channel = <fun>
```

```
# Unix.out_channel_of_descr ;;
- : Unix.file_descr -> out_channel = <fun>
# Unix.descr_of_in_channel  ;;
- : in_channel -> Unix.file_descr = <fun>
# Unix.descr_of_out_channel ;;
- : out_channel -> Unix.file_descr = <fun>
```

It is necessary to indicate whether the input / output channels obtained by the conversion transfer binary data or character data.

```
# set_binary_mode_in ;;
- : in_channel -> bool -> unit = <fun>
# set_binary_mode_out ;;
- : out_channel -> bool -> unit = <fun>
```

In the following example we create a file by using the functions of module `Unix`. We read using the opening function of module `Unix` and the higher-level input function `input_line`.

```
# let mode = [Unix.O_WRONLY;Unix.O_CREAT;Unix.O_TRUNC] in
  let f = Unix.openfile "file" mode 0o666 in
  let s = "0123456789\n0123456789\n" in
  let n = Unix.write f s 0 (String.length s)
  in Unix.close f ;;
- : unit = ()
# let f = Unix.openfile "file" [Unix.O_RDONLY;Unix.O_NONBLOCK] 0 in
  let c = Unix.in_channel_of_descr f in
  let s = input_line c
  in print_string s ;
     close_in c ;;
0123456789- : unit = ()
```

Availability. A program may have to work with multiple inputs and outputs. Data may not always be available on a given channel, and the program cannot afford to wait for one channel to be available while ignoring the others. The following function lets you determine which of a given list of inputs/outputs is available for use at a given time:

```
# Unix.select ;;
- : Unix.file_descr list ->
    Unix.file_descr list ->
    Unix.file_descr list ->
    float ->
    Unix.file_descr list * Unix.file_descr list * Unix.file_descr list
= <fun>
```

The first three arguments represent lists of respectively inputs, of outputs and error--outputs. The last argument indicates a delay in seconds. A negative value means the null delay. The results are the lists of available input, output and error-output.

Warning `select` is not implemented under Windows

Processes

Unix associates a *process* with each execution of a program. In [CDM98] Card, Dumas and Mével describe the difference between a program and a process: "a program itself is not a process: a program is a passive entity (an executable file on a disc), while a process is an active entity with a counter specifying the next instruction to execute and a set of associated resources."

Unix is a *multi-task* operating system: many processes may be executed at the same time. It is *preemptive*, which means that the execution of processes is entrusted to a particular process. A process is therefore not totally master of its resources. Especially a process can not determine the time of its execution. A process has to be created.

Each process has his own private memory space. Processes can communicate via files or communication channels. Thus the distributed memory model of parallelism is simulated on a single machine.

The system gives each process a unique identifier: the PID (Process IDentifier). Under Unix each process, except the initial process, is created by another process, which is called its *parent*.

The set of all active processes can be listed by the Unix command `ps`[3]:

```
$ ps -f
PID    PPID    CMD
1767   1763   csh
2797   1767   ps -f
```

The use of the option `-f` adds for each active process its identifier (`PID`), that of its parent (`PPID`) and the name of the started program (`CMD`). Here we have two processes, the command line interpreter `csh` and the command `ps` itself. It can be seen that `ps` has been started from the command line interpreter `csh`. The parent of its process is the process associated with the execution of `csh`.

Executing a Program

Execution Context

Three values are associated with an executing program, which is started from the command line:

1. The command line used to start it. It is contained in the value `Sys.argv`.
2. The environment variables of the command line interpreter. These can be accessed by the command `Sys.getenv`.

3. The options and the behavior of this command are not standardized. The given example may not be reproducible.

3. An execution status until the program is terminated.

Command line. The command line allows you to read arguments or options of a program call. The behavior of the program may depend from these values. Here is a small example. We write the following program into the file `argv_ex.ml`:

```
if Array.length Sys.argv = 1 then
 Printf.printf "Hello world\n"
else if Array.length Sys.argv = 2 then
 Printf.printf "Hello %s\n" Sys.argv.(1)
else Printf.printf "%s : too many arguments\n" Sys.argv.(0)
```

We compile it:

```
$ ocamlc -o argv_ex argv_ex.ml
```

And we execute it:

```
$ argv_ex
Hello world
$ argv_ex reader
Hello reader
$ argv_ex dear reader
./argv_ex : too many arguments
```

Environment variables. Environment variables may contain values necessary for execution. The number and the names of these variables depend on the operating system and on the user configuration. The values of these variables can be accessed by the function `getenv`, which takes as argument the name of a variable in form of a character string:

```
# Sys.getenv "HOSTNAME";;
- : string = "zinc.pps.jussieu.fr"
```

Execution Status

The return value of a program is generally a fixed integer, indicating if the program did terminate with an error or not. The exact values may differ from one operating system to another. The programer can always explicitly stop his program and return the execution status value with the function call:

```
# Pervasives.exit ;;
- : int -> 'a = <fun>
```

Process Creation

A program is started by another process, which is called the current process. The executed program becomes a new process. There are three different relations between the two processes:

- The two processes are independent from each other and can be executed concurrently.
- The parent process is waiting for the child process to terminate.
- The created process replaces the parent process, which terminates.

It is also possible to duplicate the current process to obtain two instances. The two instances of the process do not differ but in their PID. This is the famous `fork` which we will describe later.

Independent Processes

The `Unix` module offers a portable function to create a process.

```
# Unix.create_process ;;
- : string ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

The first argument is the name of the program (it may be a path). The second is the array of arguments for the program. The last three arguments are the descriptors indicating the standard input, standard output and standard error output of the process. The return value is the PID of the created process.

There also exists a variant of this function which allows you to indicate the values of environment variables:

```
# Unix.create_process_env ;;
- : string ->
    string array ->
    string array ->
    Unix.file_descr -> Unix.file_descr -> Unix.file_descr -> int
= <fun>
```

These two functions can be used under Unix and Windows.

GGH

Process Stacks

It is not always useful for a created process to be of concurrent nature. The parent process may have to wait for the created process to terminate. The two following functions take as argument the name of a command and execute it.

```
# Sys.command;;
- : string -> int = <fun>
# Unix.system;;
- : string -> Unix.process_status = <fun>
```

They differ in the type of the return code. The type *process_status* is explained in more detail on page 586. During the execution of the command the parent process is blocked.

Replacement of Current Processes

The replacement of current processes by freshly created processes allows you to limit the number of concurrently executed processes. The four following functions allow this:

```
# Unix.execv ;;
- : string -> string array -> unit = <fun>
# Unix.execve ;;
- : string -> string array -> string array -> unit = <fun>
# Unix.execvp ;;
- : string -> string array -> unit = <fun>
# Unix.execvpe ;;
- : string -> string array -> string array -> unit = <fun>
```

Their first argument is the name of the program. Using `execvp` or `execvpe`, this name may indicate a path in the file system. The second argument contains the program arguments. The last argument of the functions `execve` and `execvpe` additionally allows you to indicate the values of system variables.

Creation of Processes by Duplication

The original system call to create processes under Unix is:

```
# Unix.fork ;;
- : unit -> int = <fun>
```

The function `fork` starts a new process, not a new program. Its effect is to *duplicate* the calling process. The code of the new process is the same as that of its parent. Under Unix the same code can be shared by several processes, each process possessing its own execution context. Therefore we speak about *reentrant code*.

Let's look at the following small program (we use the function `getpid` which returns the PID of the process associated with the execution):

```
Printf.printf "before fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
Unix.fork () ;;
Printf.printf "after fork : %d\n" (Unix.getpid ()) ;;
flush stdout ;;
```

We obtain the following output:

```
before fork : 10529
after fork : 10529
after fork : 10530
```

After the execution of `fork`, two processes execute the code. This leads to the output of two PID's "`after`" the `fork`. We note that one process has kept the PID of the

beginning (the parent). The other one has a new PID (the child), which corresponds to the return value of the `fork` call. For the parent process the return value of `fork` is the PID of the child, while for the child, it is `0`.

It is this difference in the return value of `fork` which allows *in one program source* to decide which code shall be executed by the child and which by the parent:

```
Printf.printf "before fork : %d\n" (Unix.getpid ())  ;;
flush stdout ;;
let pid = Unix.fork () ;;
if pid=0 then  (* -- Code of the child *)
  Printf.printf "I am  the child: %d\n" (Unix.getpid ())
else           (* -- Code of the father *)
  Printf.printf "I am the father: %d of child: %d\n"  (Unix.getpid ()) pid ;;
flush stdout ;;
```

Here is the trace of the execution of this program:

```
before fork : 10539
I am the father: 10539 of child: 10540
I am  the child: 10540
```

It is also possible to use the return value for matching:

```
match Unix.fork () with
   0  → Printf.printf "I am the child: %d\n" (Unix.getpid ())
| pid → Printf.printf "I am the father: %d of child: %d\n"
                      (Unix.getpid ()) pid ;;
```

The fertility of a process may be very big. Therefore the number of descendents of a process is limited by the configuration of the operating system. The following example creates two generations of processes with grandparent, parents, uncles and cousins.

```
let pid0 = Unix.getpid ();;
let print_generation1 pid ppid =
  Printf.printf "I am %d, son of %d\n" pid ppid;
  flush stdout ;;

let print_generation2 pid ppid pppid  =
  Printf.printf "I am %d, son of %d, grandson of %d\n"
                pid ppid pppid;
  flush stdout ;;

match Unix.fork() with
    0 → let pid01 = Unix.getpid ()
        in ( match Unix.fork() with
                 0 → print_generation2 (Unix.getpid ()) pid01 pid0
               | _ → print_generation1 pid01 pid0)
  | _ → match Unix.fork () with
            0 → ( let pid02 = Unix.getpid ()
                  in match Unix.fork() with
                         0 → print_generation2 (Unix.getpid ()) pid02 pid0
                       | _ → print_generation1 pid02 pid0 )
          | _ → Printf.printf "I am %d, father and grandfather\n" pid0 ;;
```

We obtain:
```
I am 10644, father and grandfather
I am 10645, son of 10644
I am 10648, son of 10645, grandson of 10644
I am 10646, son of 10644
I am 10651, son of 10646, grandson of 10644
```

Order and Moment of Execution

A sequence of process creations without synchronization may lead to surprising effects. This is illustrated by the following poem writing program à la M. Jourdain[4]:
```
match Unix.fork () with
  0 → Printf.printf "fair Marquise " ; flush stdout
| _ → match Unix.fork () with
          0 → Printf.printf "your beautiful eyes " ; flush stdout
        | _ → match Unix.fork () with
                  0 → Printf.printf "make me die " ; flush stdout
                | _ → Printf.printf "of love\n" ;  flush stdout ;;
```

It may produce the following result:
```
of love
fair Marquise your beautiful eyes make me die
```

We usually want our program to be able to assure the order of execution of its processes. More generally speaking, an application which makes use of several processes may have to synchronize them. Depending on the model of parallelism in use, the synchronization is realized by communication between the processes or by waiting conditions. This subject is presented more profoundly by the two following chapters. For the moment, we can improve our poem writing program in two ways:

- Give the child the time to write its phrase before writing the own.
- Wait for the termination of the child, which will then have written its phrase, before writing our own phrase.

Delays. A process can suspend its activity by calling the function:
```
# Unix.sleep ;;
- : int -> unit = <fun>
```
The argument provides the number of seconds during which the process wants to suspend its activities.

Using this function, we write:

4. Molière, *Le Bourgeois Gentilhome*, Acte II, scène 4.
Link: http://www.site-moliere.com/pieces/bourgeoi.htm

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _ → Unix.sleep 1 ;
         match Unix.fork () with
             0 → Printf.printf"your beautiful eyes "; flush stdout
           | _ → Unix.sleep 1 ;
             match Unix.fork () with
                 0 → Printf.printf"make me die "; flush stdout
               | _ → Unix.sleep 1 ; Printf.printf "of love\n" ; flush stdout ;;
```

And we can obtain:

```
fair Marquise your beautiful eyes make me die of love
```

Nevertheless, this method is not sure. In theory, it would be possible that the system gives enough time to one of the processes to sleep and to write its output at the same turn. Therefore we prefer the following method for assuring the execution order of our processes.

GGH

Waiting for the termination of the child. A parent process may wait for his child to terminate through a call to the function:

```
# Unix.wait ;;
- : unit -> int * Unix.process_status = <fun>
```

The execution of the parent is suspended until one of its children terminates. If `wait` is called by a process not having any children, a `Unix_error` is thrown. We will discuss later the return value of `wait`. For the moment, we will just use the command to pronounce our poem:

```
match Unix.fork () with
    0 → Printf.printf "fair Marquise " ; flush stdout
  | _  → ignore (Unix.wait ()) ;
          match Unix.fork () with
              0 → Printf.printf "your beautiful eyes " ; flush stdout
            | _ → ignore (Unix.wait ()) ;
                  match Unix.fork () with
                      0 → Printf.printf "make me die " ; flush stdout
                    | _ → ignore (Unix.wait ()) ;
                   Printf.printf "of love\n" ;
                   flush stdout
```

Indeed, we obtain:

```
fair Marquise your beautiful eyes make me die of love
```

Warning `fork` is proprietary to the Unix system

Descendence, Death and Funerals of Processes

The function `wait` is useful not only to wait for the termination of a child. It also has the responsibility to complete the death of the child process.

Whenever a process is created, the system adds an entry in a table. The table serves to keep track of all processes. When a process terminates, the entry does not disappear automatically in the table. It is the responsibility of the parent to assure the deletion by the call of `wait`. If this is not done, the child process keeps an entry in the table. This is called a *zombie* process.

When the system is started, a first process called `init` is started. After the initialization of some parameters, the essential role of this "forefather" is to take care of orphan processes and to call the `wait` which deletes them from the process table after their termination.

Waiting for the Termination of a Given Process

There is a variation of the function `wait`, named `waitpid`. This command is supported on Unix and Windows:

```
# Unix.waitpid ;;
- : Unix.wait_flag list -> int -> int * Unix.process_status = <fun>
```

The first argument specifies the waiting modalities. The second indicates which process or which group of processes are treated.

After the termination of a process, two pieces of information can be accessed by its parent as a result of the function calls `wait` or `waitpid`: the number of the terminated process and its exit status. The status is represented by a value of type *Unix.process_status*. This type has three constructors. Each of them takes an integer as argument.

- `WEXITED n`: the process has terminated normally with the return code `n`.
- `WSIGNALED n`: the process has been killed by the signal `n`.
- `WSTOPPED n`: the process has been stopped by the signal `n`.

The last value only makes sense for the function `waitpid` which can listen for such signals as indicated by its first argument. We will discuss signals and their treatment at page 590.

Managing of Waiting by Ancestors

In order to avoid having to care for the termination of child processes oneself, it is possible to delegate this responsibility to an ancestor process. "Double fork" allows a process not to take care of the funerals of all its child processes, but to delegate this responsibility to the `init` process. Here is the principle: a process P_0 creates a process P_1, which in turn creates a third process P_2. Then P_1 terminates. So P_2 is orphan and will be adopted by `init`, which waits for its termination. The initial process P_0 can

execute a `wait` for P_1 which will be of short duration. The idea is to delegate to the grandchild the work which otherwise would have been for the child.

The schema is the following:

```
# match Unix.fork() with                          (* P0 creates P1 *)
    0 → if Unix.fork() = 0 then exit 0 ;          (* P1 creates P2 and terminates *)
         Printf.printf "P2 did its work\n" ;
         exit 0
  | pid → ignore (Unix.waitpid [] pid) ;          (* P0 waits for P1 to terminate *)
          Printf.printf "P0 can do other things without waiting\n" ;;
P2 did its work
P0 can do other things without waiting
- : unit = ()
```

We will apply this principle to handle requests sent to a server in chapter 20.

Communication Between Processes

The use of processes in application development allows you to delegate work. Nevertheless, these jobs may not be independent and it may be necessary for the processes to communicate with each other.

We introduce two methods of communication between processes: communication pipes and signals. This chapter does not discuss all possibilities of process communication. It is only a first approach to the applications developed in chapters 19 and 20.

Communication Pipes

It is possible for processes to communicate directly between each other in a file oriented style.

Pipes are something like virtual files from which it is possible to read and to write with the input / output functions `read` and `write`. They are of limited size, the exact limit depending from the system. They behave like queues: the first input is also the first output. Whenever data is read from a pipe, it is also removed from it.

This queue behavior is realized by the association of two descriptors with a pipe: one corresponding to the end of the pipe where new entries are written and one for the end where they are read. A pipe is created by the function:

```
# Unix.pipe ;;
- : unit -> Unix.file_descr * Unix.file_descr = <fun>
```

The first component of the resulting pair is the exit of the pipe used for reading. The second is the entry of the pipe used for writing. All processes knowing them can close the descriptors.

Reading from a pipe is blocking, unless all processes knowing its input descriptor (and therefore able to write to it) have closed it; in the latter case, the function `read` returns 0. If a process tries to write to a full pipe, it is suspended until another process has done a read operation. If a process tries to write to a pipe while no other process is available

to read from it (all having closed their output descriptors), the process trying to write receives the signal `sigpipe`, which, if not indicated otherwise, leads to its termination.

The following example shows a use of pipes in which grandchildren tell their process number to their grandparents.

```
let output, input = Unix.pipe();;

let write_pid input =
  try
    let m = "(" ^ (string_of_int (Unix.getpid ())) ^ ")"
    in ignore (Unix.write input m 0 (String.length m)) ;
       Unix.close input
   with
     Unix.Unix_error(n,f,arg) →
       Printf.printf "%s(%s) : %s\n" f arg (Unix.error_message n) ;;

match Unix.fork () with
    0 → for i=0 to 5 do
          match Unix.fork() with
              0 → write_pid input ; exit 0
            | _ → ()
        done ;
        Unix.close input
  | _  → Unix.close input;
         let s = ref ""  and  buff = String.create 5
         in while true do
              match Unix.read output buff 0 5 with
                  0 → Printf.printf "My grandchildren are %s\n" !s ; exit 0
                | n → s := !s ^ (String.sub buff 0 n) ^ "."
          done ;;
```

We obtain the trace:

```
My grandchildren are (1067.3).(1067.4).(1067.8).(1067.7).(1067.6).(1067.5).
```

We have introduced points between each part of the sequence read. This way it is possible to read from the trace the succession of contents of the pipe. Note how the reading is desynchronized: whenever an entry is made, even a partial one, it is consumed.

Named pipes. Some Unix systems support named pipes, which look as if they were normal files. It is possible then to communicate between two processes without a descendence relation using the name of the pipe. The following function allows you to create such a pipe.

```
# Unix.mkfifo ;;
- : string -> Unix.file_perm -> unit = <fun>
```

The file descriptors necessary to use the pipe are obtained by `openfile`, as for usual files, but their behavior is that of pipes. In particular, the command `lseek` can not be used, since we have waiting lines.

Warning `mkfifo` is not implemented for Windows.

Communication Channels

The `Unix` module provides a high level function allowing you to start a program associating with it input or output channels of the calling program:

```
# Unix.open_process ;;
- : string -> in_channel * out_channel = <fun>
```

The argument is the name of the program, or more precisely the calling path of the program, as we would write it to a command line interpreter. The string may contain arguments for the program to execute. The two output values are file descriptors associated with the standard input / output of the started program. It will be executed in parallel with the calling program.

Warning The program started by `open_process` is executed via a call to the Unix command line interpreter `/bin/sh`. The use of that function is therefore only possible for systems that have this interpreter.

We can end the execution of a program started by `open_process` by using:

```
# Unix.close_process ;;
- : in_channel * out_channel -> Unix.process_status = <fun>
```

The argument is the pair of channels associated with a process we want to close. The return value is the execution status of the process whose termination we wait.

There are variants of that functions, opening and closing only one input or output channel:

```
# Unix.open_process_in ;;
- : string -> in_channel = <fun>
# Unix.close_process_in ;;
- : in_channel -> Unix.process_status = <fun>
# Unix.open_process_out ;;
- : string -> out_channel = <fun>
# Unix.close_process_out ;;
- : out_channel -> Unix.process_status = <fun>
```

Here is a nice small example for the use of `open_process`: we start `ocaml` from `ocaml`!

```
# let n_print_string s = print_string s ; print_string "(* <-- *)" ;;
val n_print_string : string -> unit = <fun>
# let p () =
    let oc_in, oc_out = Unix.open_process "/usr/local/bin/ocaml"
    in n_print_string (input_line oc_in) ; print_newline() ;
       n_print_string (input_line oc_in) ; print_newline() ;
       print_char (input_char oc_in) ;
       print_char (input_char oc_in) ;
       flush stdout ;
       let s = input_line stdin
       in output_string oc_out s ;
```

```
          output_string oc_out "#quit\n" ;
          flush oc_out ;
          let r = String.create 250 in
          let n = input oc_in r 0 250
          in n_print_string (String.sub r 0 n) ;
             print_string "Thank you for your visit\n" ;
             flush stdout ;
             Unix.close_process (oc_in, oc_out) ;;
val p : unit -> Unix.process_status = <fun>
```

The call of the function `p` starts a *toplevel* of Objective Caml. We note that it is version 2.03 which is in directory `/usr/local/bin`. The first four read operations allow us to get the header, which is shown by *toplevel*. The line `let x = 1.2 +. 5.6;;` is read from the keybard, then sent to `oc_out` (the output channel bound to the standard input of the new process). This one evaluates the passed Objective Caml expression and writes the result to the standard output which is bound to the input channel `oc_in`. This result is read and written to the output by the function `input`. Also the string `"Thank you for your visit"` is written to the output. We send the command #*quit*;; to exit the new process.

```
# p();;
        Objective Caml version 2.03

# let x = 1.2 +. 5.6;;
val x : float = 6.8
Thank you for your visit
- : Unix.process_status = Unix.WSIGNALED 13
#
```

Signals under Unix

One possibility to communicate with a process is to send it a *signal*. A signal may be received at any moment during the execution of a program. Reception of a signal causes a logical interruption. The execution of a program is interrupted to treat the received signal. Then the execution continues at the point of interruption. The number of signals is quite restricted (32 under Linux). The information carried by a signal is quite rudimentary: it is only the identity (the number) of the signal. The processes have a predefined reaction to each signal. However, the reactions can be redefined for most of the signals.

The data and functions to handle signals are distributed between the modules `Sys` and `Unix`. The module `Sys` contains signals conforming to the POSIX norm (described in [Ste92]) as well as some functions to handle signals. The module `Unix` defines the function `kill` to send a signal. The use of signals under Windows is restricted to `sigint`.

A signal may have several sources: the keyboard, an illegal attempt to access memory, etc. A process may send a signal to another by calling the function
Unix.kill ;;

```
- : int -> int -> unit = <fun>
```
Its first parameter is the PID of the receiver. The second is the signal which we want to send.

Handling Signals

There are three categories of reactions associated with a signal. For each category there is a constructor of type `signal_behavior`:

- `Signal_default`: the default behavior defined by the system. In most of the cases this is the termination of the process, with or without the creation of a file describing the process state (`core` file).
- `Signal_ignore`: the signal is ignored.
- `Signal_handle`: the behavior is redefined by an Objective Caml function of type *`int -> unit`* which is passed as an argument to the constructor. For the modified handling of the signal, the number of the signal is passed to the handling function.

On reception of a signal, the execution of the receiving process is diverted to the function handling the signal. The function allowing you to redefine the behavior associated with a signal is provided by the module `Sys`:
```
# Sys.set_signal;;
- : int -> Sys.signal_behavior -> unit = <fun>
```
The first argument is the signal to redefine. The second is the associated behavior.

The module `Sys` provides another modification function to handle signals:
```
# Sys.signal ;;
- : int -> Sys.signal_behavior -> Sys.signal_behavior = <fun>
```

It behaves like `set_signal`, except that it returns in addition the value associated with the signal before the modification. So we can write a function returning the behavioral value associated with a signal. This can be done even without changing this value:
```
# let signal_behavior s =
    let b = Sys.signal s Sys.Signal_default
    in Sys.set_signal s b ; b ;;
val signal_behavior : int -> Sys.signal_behavior = <fun>
# signal_behavior Sys.sigint;;
- : Sys.signal_behavior = Sys.Signal_handle <fun>
```

However, the behavior associated with some signals can not be changed. Therefore our function can not be used for all signals:
```
# signal_behavior Sys.sigkill ;;
Uncaught exception: Sys_error("Invalid argument")
```

Some Signals

We illustrate the use of some essential signals.

sigint. This signal is generally associated with the key combination `CTRL-C`. In the following small example we modify the reaction to this signal so that the receiving process is not interrupted until the third occurence of the signal.

We create the following file `ctrlc.ml`:

```
let sigint_handle =
 let n = ref 0
 in function _ → incr n ;
                 match !n with
                     1 → print_string "You just pushed CTRL-C\n"
                   | 2 → print_string "You pushed CTRL-C a second time\n"
                   | 3 → print_string "If you insist ...\n" ; exit 1
                   | _ → () ;;
Sys.set_signal Sys.sigint (Sys.Signal_handle sigint_handle) ;;
match Unix.fork () with
    0 → while true do () done
  | pid → Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;
          Unix.sleep 1 ; Unix.kill pid Sys.sigint  ;;
```

This program simulates the push of the key combination `CTRL-C` by sending the signal `sigint`. We obtain the following execution trace:

```
$ ocamlc -i -o ctrlc ctrlc.ml
val sigint_handle : int -> unit
$ ctrlc
You just pushed CTRL-C
You pushed CTRL-C a second time
If you insist ...
```

sigalrm. Another frequently used signal is `sigalrm`, which is associated with the system clock. It can be sent by the function

```
# Unix.alarm ;;
- : int -> int = <fun>
```

The argument specifies the number of seconds to wait before the sending of the signal `sigalrm`. The return value indicates the number of remaining seconds before the sending of a second signal, or if there is no alarm set.

We use this function and the associated signal to define the function `timeout`, which starts the execution of another function and interrupts it if neccessary, when the indicated time is elapsed. More precisely, the function `timeout` takes as arguments a function `f`, the argument `arg` expected by `f`, the duration (`time`) of the "*timeout*" and the value (`default_value`) to be returned when the duration time has elapsed.

A `timeout` is handled as follows:

1. We modify the behavior associated with the signal `sigalrm` so that a `Timeout` exception is thrown.

2. We take care to remember the behavior associated originally with `sigalrm`, so that it can be restored.
3. We start the clock.
4. We distinguish two cases:
   (a) If everything goes well, we restore the original state of `sigalrm` and return the value of the calculation.
   (b) If not, we restore `sigalrm`, and if the duration has elapsed, we return the default value.

Here are the corresponding definitions and a small example:

```
# exception Timeout ;;
exception Timeout
# let sigalrm_handler = Sys.Signal_handle (fun _ → raise Timeout) ;;
val sigalrm_handler : Sys.signal_behavior = Sys.Signal_handle <fun>
# let timeout f arg time default_value =
    let old_behavior = Sys.signal Sys.sigalrm sigalrm_handler in
    let reset_sigalrm () = Sys.set_signal Sys.sigalrm old_behavior
    in ignore (Unix.alarm time) ;
       try  let res = f arg in reset_sigalrm () ; res
       with exc → reset_sigalrm () ;
                  if exc=Timeout then default_value else raise exc ;;
val timeout : ('a -> 'b) -> 'a -> int -> 'b -> 'b = <fun>
# let iterate n = for i = 1 to n do () done ; n ;;
val iterate : int -> int = <fun>

Printf.printf "1st execution : %d\n" (timeout iterate 10 1 (-1));
Printf.printf "2nd execution : %d\n" (timeout iterate 100000000 1 (-1)) ;;


1st execution : 10
2nd execution : -1
- : unit = ()
```

sigusr1 and sigusr2. These two signals are provided only for the programer. They are not used by the operating system.

In this example, reception of the signal `sigusr1` by the child triggers the output of the content of variable `i`.

```
let i = ref 0  ;;
let write_i s = Printf.printf "signal received (%d) -- i=%d\n" s !i ;
                flush stdout ;;
Sys.set_signal Sys.sigusr1 (Sys.Signal_handle write_i) ;;

match Unix.fork () with
    0 → while true do incr i done
  | pid → Unix.sleep 0  ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 3 ; Unix.kill pid Sys.sigusr1  ;
          Unix.sleep 1  ; Unix.kill pid Sys.sigkill
```

Here is the trace of a program execution:

```
signal received (10) -- i=0
signal received (10) -- i=167722808
```

When we examine the trace, we can see that after having executed the code associated with signal `sigusr1` the first time, the child process continues to execute the loop and to increment `i`.

sigchld. This signal is sent to a parent on termination of a process. We will use it to make a parent more attentive to the evolution of its children. Here's how:

1. We define a function handling the signal `sigchld`. It handles all terminated children on reception of this signal[5] and terminates the parent when he does not have any more children (exception `Unix_error`). In order not to block the parent if not all his children are dead, we use `waitpid` instead of `wait`.
2. The main program, after having redefined the reaction associated with `sigchld`, loops to create five children. After this, the parent does something else (loop **`while`** `true`) until his children have terminated.

```
let rec sigchld_handle s =
  try  let pid, _ = Unix.waitpid [Unix.WNOHANG] 0
       in if pid <> 0
          then ( Printf.printf "%d is dead and buried at signal %d\n" pid s ;
                 flush stdout ;
                 sigchld_handle s )
  with Unix.Unix_error(_, "waitpid", _) → exit 0 ;;

let i = ref 0
in Sys.set_signal Sys.sigchld (Sys.Signal_handle sigchld_handle) ;
   while true do
     match Unix.fork() with
         0 → let pid = Unix.getpid ()
             in Printf.printf "Creation of %d\n" pid ; flush stdout ;
                Unix.sleep (Random.int (5+ !i)) ;
                Printf.printf "Termination of %d\n" pid ; flush stdout ;
                exit 0
       | _ → incr i ; if !i = 5 then while true do () done
   done ;;
```

We obtain the trace:

```
Creation of 10658
Creation of 10659
Creation of 10662
Creation of 10661
```

5. We recall that the signals are handled in an asynchronous way. So, if two children die one after the other, it is possible that the signal of the first has not been handled.

```
Creation of 10660
Termination of 10662
10662 is dead and buried at signal 17
Termination of 10658
10658 is dead and buried at signal 17
Termination of 10660
Termination of 10659
10660 is dead and buried at signal 17
10659 is dead and buried at signal 17
Termination of 10661
10661 is dead and buried at signal 17
```

Exercises

The three proposed exercises manipulate file descriptors, processes, respectively pipes and signals. The first two exercises stem from Unix system programming. The Objective Caml code can be compared with the C code in the Unix or Linux distributions.

Counting Words: the `wc` *Command*

We want to (re)program the Unix `wc` command, which counts the number of lines, words or characters contained in a text file. Words are separated by a space character, a tab, or a carriage return. We do not count the separators.

1. Write a first version (`wc1`) of the command, which only handles a single file. The name of the file is passed as an argument on the command line.
2. Write a more elaborated version (`wc2`), which can handle the three options `-l`, `-c`, `-w` as well as several file names. The options indicate if we want to count the number of lines, characters or words. The output of each result shall be preceded by the name of the file.

Pipes for Spell Checking

This exercise uses pipes to concatenate a suite of actions. Each action takes the result of the preceding action as argument. The communication is realized by pipes, connecting the output of one process to the input of the following, in the style of the Unix command line symbol `|` .

1. Write a function `pipe_two_progs` of type *`string * string list -> string * string list -> unit`* such that `pipe_two_progs (p1,[a1; ...; an]) (p2,[b1; ...; bp])` starts the programs `p1 a1 ... an` and `p2 b1 ... bp`, redirecting the standard output of `p1` to the standard input of `p2`. *`ai`* and *`bi`* are the command line arguments of each program.
2. We revisit the spell checker function from the exercise on page 115 to write a first program. Modify it so that the list of faulty words is sent without treatment in the form of one line per word to the standard output.

3. The second program takes a sequence of character strings from its standard input and sorts it in lexicographical order. The function `Sort.list` can be used, which sorts a list in an order defined by a given predicate. The sorted list is written to the standard output.
4. Test the function `pipe_two_progs` with the two programs.
5. Write a function `pipe_n_progs` to connect a list of programs.
6. Write a program to suppress multiple occurrences of elements in a list.
7. Test the function `pipe_n_progs` with these three programs.

Interactive Trace

In a complex calculation it may be useful to interact with the program to verify the progression. For this purpose we revisit the exercise on page 244 on the computation of prime numbers contained in an interval.

1. Modify the program so that a global variable `result` always contains the last prime number found.
2. Write a function `sigint_handle` which handles the signal `sigint` and writes the content of `result` to the output.
3. Modify the default signal handling of `sigint` by associating with it the preceding function `sigint_handle`.
4. Compile the program, then start the executable with an upper bound for the computation time. During the computation, send the signal `sigint` to the process, by the Unix `kill` command as well as by the key combination `CTRL-C`.

Summary

This chapter presented the main system interface functions provided by the `Unix` module. Despite of its name, the module offers a large number of functions which can be used under Windows as well (see figure 18.1).
In the area of process creation, we did concentrate on the possibilities of communication between several Objective Caml programs running at the same time on the same machine. Operations handling lower level file access, signals and communication pipes have been discussed in detail.

To Learn More

The `Unix` module provides functions of the Unix system library. Most of the underlying programming paradigms are not described in Objective Caml. The reader may refer to standard books about system programming. We cite [Ste92], or [CDM98], more specific to Linux.

Further, the excellent lecture notes from Xavier Leroy [Ler92] have for subject system programming in Caml-Light. They can be accessed under the following address:

Link: http://pauillac.inria.fr/~xleroy/publi/unix-in-caml.ps.gz

The implementation of the `Unix` module is a good example for the cooperation between C and Objective Caml. A large number of functions are just calls to C system functions, with the additional type transcription of the data. The implementation sources are good examples of how to interface an Objective Caml program with a C library. The programs can be found in the directories `otherlibs/unix` and `otherlibs/win32unix` of the Objective Caml distribution.

The chapter did present several functionalities of the `Unix` module. Some more points will be approached in chapter 20 about communication sockets and Internet addresses. Other notions, like that of terminals, of file systems, etc. are not discussed in this book. They can be explored in one of the books mentioned above.

19

Concurrent Programming

Concurrency is the word used to describe causal independence between a number of actions, such as the execution of a number of instructions "at the same time". This is also the definition which we give of the term "parallel" in the introduction of this fourth part. The processes of the `Unix` library presented in the preceding chapter could be considered as concurrent to the extent that the `Unix` system provides the appearance of their simultaneous execution on a uniprocessor machine. But the notion of process and concurrency does not apply only to those obtained by the `fork` system call.

The Objective Caml language possesses a library for lightweight processes (threads.) The principle difference between a thread and a process is in the sharing or non-sharing of memory between the different child processes of the same program. Only the context of execution differs between two threads: the code and memory sections of the data are shared. Threads do not improve the execution time of an application. Their principal attraction is to make it possible to express the programming of concurrent algorithms within a language.

The nature of the chosen language, imperative or functional, affects the model of concurrency. For an imperative program, as every thread can modify the communal/shared memory, we are in a shared memory model. Communication between processes can be achieved by values written and read in this memory. For a purely functional program, that is to say, without side effects, even though the memory is shared, the calculations which each process executes do not act on this shared memory. In this case, the model used is that of separate memory and interaction between processes must be achieved by communication of values though channels.

The Objective Caml language implements both models in its thread library. The `Thread` module makes it possible to start new processes corresponding to a function call with its argument. Modules `Mutex` and `Condition` provide the synchronization tools for mutual exclusion and waiting on a condition. The `Event` model implements a means of communication of language values by events. These values can themselves be

functional, thus making it possible to exchange calculations to be carried out between threads. As always in Objective Caml it is possible to mix the two models.

This library is portable to the different systems where OCAML runs. Unlike the `Unix` module, the `Thread` library facilitates the use of processes on machines that are not running Unix.

Plan of the Chapter

The first section details the possible interactions between threads, and proceeds with describing module `Thread`, and showing how to execute many processes in the same application.

The second part deals with the synchronization between threads by mutual exclusion (`Mutex` module), and with waiting for conditions (`Condition` module). Two complete examples show the difficulties inherent to this module.

The third section explains the mode of communication by events provided by the `Event` module and the new possibilities which is provides.

The fourth section concludes the chapter with the implementation of a shared queue for the different counters at a post office.

Concurrent Processes

With an application composed of many concurrent processes, we lose the convenience offered by the determinism of sequential programs. For processes sharing the same zone of memory, the result of the following program cannot be deduced from reading it.

main program	
`let x = ref 1;;`	
process P	process Q
`x := !x + 1;;`	`x := !x * 2;;`

At the end of the execution of P and Q, the reference `x` can point to 2, 3 or 4, depending on the order of execution of each process.

This indeterminism applies also to terminations. When the memory state depends on the execution of each parallel process, an application can fail to terminate on a particular execution, and terminate on another. To provide some control over the execution, the processes must be synchronized.

For processes using distinct memory areas, but communicating between each other, their interaction depends on the type of communication. We introduce for the following example two communication primitives: `send` which sends a value, showing the

destination, and `receive` which receives a value from a process. Let P and Q be two communicating processes:

process P	process Q
`let x = ref 1;;`	`let y = ref 1;;`
`send(Q,!x);`	`y := !y + 3;`
`x := !x * 2;`	`y := !y + receive(P);`
`send(Q,!x);`	`send(P,!y);`
`x := !x + receive(Q);`	`y := !y + receive(P);`

In the case of a transient communication, process Q can miss the messages of P. We fall back into the non-determinism of the preceding model.

For an asynchronous communication, the medium of the communication channel stores the different values that have been transmitted. Only reception is blocking. Process P can be waiting for Q, even if the latter has not yet read the two messages from P. However, this does not prevent it from transmitting.

We can classify concurrent applications into five categories according to the program units that compose them:

1. unrelated;
2. related, but without synchronization;
3. related, with mutual exclusion;
4. related, with mutual exclusion and communication;
5. related, without mutual exclusion, and with synchronous communication.

The difficulty of implementation comes principally from these last categories. Now we will see how to resolve these difficulties by using the Objective Caml libraries.

Compilation with Threads

The Objective Caml thread library is divided into five modules, of which the first four each define an abstract type:

- module `Thread`: creation and execution of threads. (type *`Thread.t`*);
- module `Mutex`: creation, locking and release of mutexes. (type *`Mutex.t`*);
- module `Condition`: creation of conditions (signals), waiting and waking up on a condition (type *`Condition.t`*);
- module `Event`: creation of communication channels (type *`'a Event.channel`*), the values which they carry (type *`'a Event.event`*), and communication functions.
- module `ThreadUnix`: redefinitions of I/O functions of module `Unix` so that they are not blocking.

This library is not part of the execution library of Objective Caml. Its use requires the option `-custom` both for compiling programs and for constructing a new toplevel by using the commands:

```
$ ocamlc -thread -custom threads.cma  files.ml -cclib -lthreads
$ ocamlmktop -tread -custom -o threadtop thread.cma -cclib -lthreads
```

The `Threads` library is not usable with the native compiler unless the platform implements threads conforming to the POSIX 1003[1]. Thus we compile executables by adding the libraries `unix.a` and `pthread.a`:

```
$ ocamlc -thread -custom threads.cma files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamltop -thread -custom threads.cma files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
$ ocamlcopt -thread threads.cmxa files.ml -cclib -lthreads \
  -cclib -lunix -cclib -lpthread
```

Module `Thread`

The Objective Caml `Thread` module contains the primitives for creation and management of threads. We will not make an exhaustive presentation, for instance the operations of file I/O have been described in the preceding chapter.

A thread is created through a call to:

```
# Thread.create ;;
- : ('a -> 'b) -> 'a -> Thread.t = <fun>
```

The first argument, of type *'a -> 'b*, corresponds to the function executed by the created process; the second argument, of type *'a*, is the argument required by the executed function; the result of the call is the descriptor associated with the process. The process thus created is automatically destroyed when the associated function terminates.

Knowing its descriptor, we can ask for the execution of a process and wait for it to finish by using the function `join`. Here is a usage example:

```
# let f_proc1 () = for i=0 to 10 do Printf.printf "(%d)" i; flush stdout done;
                   print_newline() ;;
val f_proc1 : unit -> unit = <fun>
# let t1 = Thread.create f_proc1 () ;;
val t1 : Thread.t = <abstr>
# Thread.join t1 ;;
(0)(1)(2)(3)(4)(5)(6)(7)(8)(9)(10)
- : unit = <unknown constructor>
```

1. In this case, the Objective Caml compilers should have been constructed to indicate that they used the library furnished by the platform, and not the one provided by the distribution.

Warning

> The result of the execution of a process is not recovered by the parent process, but lost when the child process terminates.

We can also brutally interrupt the execution of a process of which we know the descriptor with the function `kill`. For instance, we create a process which is immediately interrupted:

```
# let n = ref 0 ;;
val n : int ref = {contents=0}
# let f_proc1 () = while true do incr n done ;;
val f_proc1 : unit -> unit = <fun>
# let go () = n := 0 ;
              let t1 = Thread.create f_proc1 ()
              in Thread.kill t1 ;
                 Printf.printf "n = %d\n" !n ;;
val go : unit -> unit = <fun>
# go () ;;
n = 0
- : unit = ()
```

A process can put an end to its own activity by the function:

```
# Thread.exit ;;
- : unit -> unit = <fun>
```

It can suspend its activity for a given time by a call to:

```
# Thread.delay ;;
- : float -> unit = <fun>
```

The argument stands for the number of seconds to wait.

Let us consider the previous example, and add timing. We create a first process `t1` of which the associated function `f_proc2` creates in its turn a process `t2` which executes `f_proc1`, then `f_proc2` delays for `d` seconds, and then terminates `t2`. On termination of `t1`, we print the contents of `n`.

```
# let f_proc2 d =
    n := 0 ;
    let t2 = Thread.create f_proc1 ()
    in Thread.delay d ;
       Thread.kill t2 ;;
val f_proc2 : float -> unit = <fun>
# let t1 = Thread.create f_proc2 0.25
  in Thread.join t1 ; Printf.printf "n = %d\n" !n ;;
n = 132862
- : unit = ()
```

Synchronization of Processes

In the setting of processes sharing a common zone of memory, the word "concurrency" carries its full meaning: the various processes involved are compete for access to the unique resource of the memory[2]. To the problem of division of resources, is added that of the lack of control of the alternation and of the execution times of the concurrent processes.

The system which manages the collection of processes can at any moment interrupt a calculation in progress. Thus when two processes cooperate, they must be able to guarantee the integrity of the manipulations of certain shared data. For this, a process should be able to remain owner of these data as long as it has not completed a calculation or any other operation (for example, an acquisition of data from a peripheral). To guarantee the exclusivity of access to the data to a single process, we set up a mechanism called *mutual exclusion*.

Critical Section and Mutual Exclusion

The mechanisms of mutual exclusion are implemented with the help of particular data structures called *mutexes*. The operations on mutexes are limited to their creation, their setting, and their disposal. A mutex is the smallest item of data shared by a collection of concurrent processes. Its manipulation is always exclusive. To the notion of exclusivity of manipulation of a mutex is added that of exclusivity of *possession*: only the process which has taken a mutex can free it; if other processes wish to use the mutex, then they must wait for it to be released by the process that is holding it.

`Mutex` Module

Module `Mutex` is used to create mutexes between processes related by mutual exclusion on an area of memory. We will illustrate their use with two small classic examples of concurrency.

The functions of creation, locking, and unlocking of mutexes are:

```
# Mutex.create ;;
- : unit -> Mutex.t = <fun>
# Mutex.lock ;;
- : Mutex.t -> unit = <fun>
# Mutex.unlock ;;
- : Mutex.t -> unit = <fun>
```

There exists a variant of mutex locking that is non-blocking:

```
# Mutex.try_lock;;
- : Mutex.t -> bool = <fun>
```

2. In a more general sense, we can be in contention for other resources such as I/O peripherals

If the mutex is already locked, the function returns **`false`**. Otherwise, the function locks the mutex and returns **`true`**.

The Dining Philosophers

This little story, due to Dijkstra, illustrates a pure problem of resource allocation. It goes as follows:

"Five oriental philosophers divide their time between study and coming to the refectory to eat a bowl of rice. The room devoted to feeding the philosophers contains nothing but a single round table on which there is a large dish of rice (always full), five bowls, and five chopsticks."

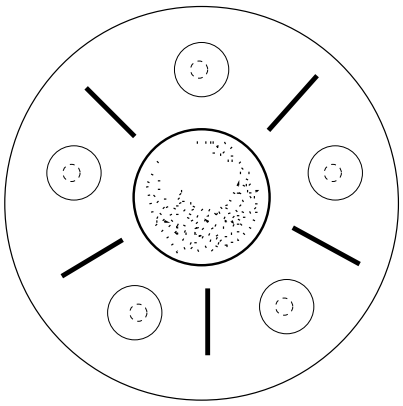


Figure 19.1: The Table of the Dining Philosophers

As we can see in the figure 19.1, a philosopher who takes his two chopsticks beside his bowl stops his neighbours from doing the same. When he puts down one of his chopsticks, his neighbour, famished, can grab it. If needs be, this latter should wait until the other chopstick is available. Here the chopsticks are the resources to be allocated.

To simplify things, we suppose that each philosopher habitually comes to the same place at the table. We model the five chopsticks as five mutexes stored in a vector `b`.

```
# let b =
    let b0 = Array.create 5 (Mutex.create()) in
     for i=1 to 4 do b0.(i) <- Mutex.create() done;
     b0 ;;
val b : Mutex.t array = [|<abstr>; <abstr>; <abstr>; <abstr>; <abstr>|]
```

Eating and meditation are simulated by a suspension of processes.

```
# let meditation = Thread.delay
  and eating = Thread.delay ;;
val meditation : float -> unit = <fun>
val eating : float -> unit = <fun>
```

We model a philosopher by a function which executes an infinite sequence of actions from Dijsktra's story. Taking a chopstick is simulated by the acquisition of a mutex, thus a single philosopher can hold a given chopstick at a time. We introduce a little time of reflection between taking and dropping of each of the two chopsticks while a number of output commands track the activity of the philosopher.

```
# let philosopher i =
    let ii = (i+1) mod 5
    in while true do
         meditation 3. ;
         Mutex.lock b.(i);
         Printf.printf "Philosopher (%d) takes his left-hand chopstick" i ;
         Printf.printf " and meditates a little while more\n";
         meditation 0.2;
         Mutex.lock b.(ii);
         Printf.printf "Philosopher (%d) takes his right-hand chopstick\n" i;
         eating 0.5;
         Mutex.unlock b.(i);
         Printf.printf "Philosopher (%d) puts down his left-hand chopstick" i;
         Printf.printf " and goes back to meditating\n";
         meditation 0.15;
         Mutex.unlock b.(ii);
         Printf.printf "Philosopher (%d) puts down his right-hand chopstick\n" i
       done ;;
val philosopher : int -> unit = <fun>
```

We can test this little program by executing:

```
for i=0 to 4 do ignore (Thread.create philosopher i) done ;
while true do Thread.delay 5. done ;;
```

We suspend, in the infinite loop **while**, the main process in order to increase the chances of the philosopher processes to run. We use randomly chosen delays in the activity loop with the aim of creating some disparity in the parallel execution of the processes.

Problems of the naïve solution. A terrible thing can happen to our philosophers: they all arrive at the same time and seize the chopstick on their left. In this case we are in a situation of *dead-lock*. None of the philosophers can eat! We are in a situation of *starvation*.

To avoid this, the philosophers can put down a chopstick if they do not manage to take the second one. This is highly courteous, but still allows two philosophers to gang up against a third to stop him from eating, by not letting go of their chopsticks, except the ones that their other neighbour has given them. There exist numerous solutions to this problem. One of them is the object of the exercise on page 619.

Producers and Consumers I

The pair of *producers-consumers* is a classic example of concurrent programming. A group of processes, designated the producers, are in charge of storing data in a queue: a second group, the consumers, is in charge of removing it. Each intervening party excludes the others.

We implement this scheme using a queue shared between the producers and the consumers. To guarantee the proper operation of the system, the queue is manipulated in mutual exclusion in order to guarantee the integrity of the operations of addition and removal.

`f` is the shared queue, and `m` is the mutex.

```
# let f = Queue.create () and m = Mutex.create () ;;
val f : '_a Queue.t = <abstr>
val m : Mutex.t = <abstr>
```

We divide the activity of a producer into two parts: creating a product (function `produce`) and storing a product (fonction `store`). Only the operation of storage needs the mutex.

```
# let produce i p d =
    incr p ;
    Thread.delay d ;
    Printf.printf "Producer (%d) has produced %d\n" i !p ;
    flush stdout ;;
val produce : int -> int ref -> float -> unit = <fun>

# let store i p =
    Mutex.lock m ;
    Queue.add (i,!p) f ;
    Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
    flush stdout ;
    Mutex.unlock m ;;
val store : int -> int ref -> unit = <fun>
```

The code of the producer is an endless loop of creation and storage. We introduce a random delay at the end of each iteration in order to desynchronize the execution.

```
# let producer i =
    let p = ref 0 and d = Random.float 2.
    in while true do
         produce i p d ;
         store i p ;
         Thread.delay (Random.float 2.5)
       done ;;
val producer : int -> unit = <fun>
```

The only operation of the consumer is the retrieval of an element of the queue, taking care that the product is actually there.

```
# let consumer i =
```

```
    while true do
      Mutex.lock m ;
      ( try
          let ip, p = Queue.take f
          in Printf.printf "The consumer(%d) " i ;
             Printf.printf "has taken product (%d,%d)\n" ip p ;
             flush stdout ;
        with
           Queue.Empty →
             Printf.printf "The consumer(%d) " i ;
             print_string "has returned empty-handed\n" ) ;
      Mutex.unlock m ;
      Thread.delay (Random.float 2.5)
    done ;;
val consumer : int -> unit = <fun>
```

The following test program creates four producers and four consumers.

```
for i = 0 to 3 do
 ignore (Thread.create producer i);
 ignore (Thread.create consumer i)
done ;
while true do Thread.delay 5. done ;;
```

Waiting and Synchronization

The relation of mutual exclusion is not "fine" enough to describe synchronization between processes. It is not rare that the work of a process depends on the completion of an action by another process, thus modifying a certain condition. It is therefore desirable that the processes should be able to communicate the fact that this condition might have changed, indicating to the waiting processes to test it again. The different processes are thus in a relation of mutual exclusion with communication.

In the preceding example, a consumer, rather than returning empty-handed, could wait until a producer came to resupply the stock. This last could signal to the waiting consumer that there is something to take. The model of waiting on a condition to take a mutex is known as *semaphore*.

Semaphores. A semaphore is an integral variable s which can only take non negative values. Once s is initialised, the only operations allowed are: $wait(s)$ and $signal(s)$, written $P(s)$ and $V(s)$, respectively. They are defined thus, s corresponding to the number of resources of a given type.

- $wait(s)$: if $s > 0$ then $s := s - 1$, otherwise the process, having called $wait(s)$, is suspended.
- $signal(s)$: if a process has been suspended after a prior invocation of $wait(s)$, then wake it up, otherwise $s := s + 1$.

A semaphore which only takes the values 0 or 1 is called a *binary semaphore*.

`Condition` *Module*

The functions of the module `Condition` implement the primitives of putting to sleep and waking up processes on a signal. A signal, in this case, is a variable shared by a collection of processes. Its type is abstract and the manipulation functions are:

`create` : *`unit -> Condition.t`* which creates a new signal.

`signal` : *`Condition.t -> unit`* which wakes up one of the processes waiting on a signal.

`broadcast` : *`Condition.t -> unit`* which wakes up all of the processes waiting on a signal.

`wait` : *`Condition.t -> Mutex.t -> unit`* which suspends the calling process on the signal passed as the first argument. The second argument is a mutex used to protect the manipulation of the signal. It is released, and then reset at each execution of the function.

Producers and Consumers (2)

We revisit the example of producers and consumeres by using the mechanism of condition variables to put to sleep a consumer arriving when the storehouse is empty.

To implement synchronization between waiting consumers and production, we declare:

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
```

We modify the storage function of the producer by adding to it the sending of a signal:

```
# let store2 i p =
    Mutex.lock m ;
    Queue.add (i,!p) f ;
    Printf.printf "Producer (%d) has added its %dth product\n" i !p ;
    flush stdout ;
    Condition.signal c ;
    Mutex.unlock m ;;
val store2 : int -> int ref -> unit = <fun>
# let producer2 i =
    let p = ref 0 in
    let d = Random.float 2.
    in while true do
         produce i p d;
         store2 i p;
```

```
        Thread.delay (Random.float 2.5)
      done ;;
val producer2 : int -> unit = <fun>
```

The activity of the consumer takes place in two phases: waiting until a product is available, then taking the product. The mutex is taken when the wait is finished and it is released when the consumer has taken its product. The wait takes place on the variable `c`.

```
# let wait2 i =
    Mutex.lock m ;
    while Queue.length f = 0 do
      Printf.printf "Consumer (%d) is waiting\n" i ;
      Condition.wait c m
    done ;;
val wait2 : int -> unit = <fun>
# let take2 i =
    let ip, p = Queue.take f in
      Printf.printf "Consumer (%d) " i ;
      Printf.printf "takes product (%d, %d)\n" ip p ;
      flush stdout ;
      Mutex.unlock m ;;
val take2 : int -> unit = <fun>
# let consumer2 i =
    while true do
      wait2 i;
      take2 i;
      Thread.delay (Random.float 2.5)
   done ;;
val consumer2 : int -> unit = <fun>
```

We note that it is no longer necessary, once a consumer has begun to wait in the queue, to check for the existence of a product. Since the end of its wait corresponds to the locking of the mutex, it does not run the risk of having the new product stolen before it takes it.

Readers and Writers

Here is another classic example of concurrent processes in which the agents do not have the same behaviour with respect to the shared data.

A writer and some readers operate on some shared data. The action of the first may cause the data to be momentarily inconsistent, while the second group only have a passive action. The difficulty arises from the fact that we do not wish to prohibit multiple readers from examining the data simultaneously. One solution to this problem is to keep a counter of the number of readers in the processes of accessing the data. Writing is not allowed except if the number of readers is 0.

The data is symbolized by the integer `data` which takes the value `0` or `1`. The value `0` indicates that the data is ready for reading:

```
# let data = ref 0 ;;
val data : int ref = {contents=0}
```

Operations on the counter `n` are protected by the mutex `m`:

```
# let n = ref 0 ;;
val n : int ref = {contents=0}
# let m = Mutex.create () ;;
val m : Mutex.t = <abstr>
# let cpt_incr () = Mutex.lock m ; incr n ; Mutex.unlock m ;;
val cpt_incr : unit -> unit = <fun>
# let cpt_decr () = Mutex.lock m ; decr n ; Mutex.unlock m ;;
val cpt_decr : unit -> unit = <fun>
# let cpt_signal () = Mutex.lock m ;
                      if !n=0 then Condition.signal c ;
                      Mutex.unlock m ;;
val cpt_signal : unit -> unit = <fun>
```

The readers update the counter and emit the signal `c` when no more readers are present. This is how they indicate to the writer that it may come into action.

```
# let c = Condition.create () ;;
val c : Condition.t = <abstr>
# let read i =
    cpt_incr () ;
    Printf.printf "Reader (%d) read (data=%d)\n" i !data ;
    Thread.delay (Random.float 1.5) ;
    Printf.printf "Reader (%d) has finished reading\n" i ;
    cpt_decr () ;
    cpt_signal () ;;
val read : int -> unit = <fun>

# let reader i = while true do read i ; Thread.delay (Random.float 1.5) done ;;
val reader : int -> unit = <fun>
```

The writer needs to block the counter to prevent the readers from accessing the shared data. But it can only do so if the counter is 0, otherwise it waits for the signal indicating that this is the case.

```
# let write () =
    Mutex.lock m ;
    while !n<>0 do Condition.wait c m done ;
    print_string  "The writer is writing\n" ; flush stdout ;
    data := 1 ; Thread.delay (Random.float 1.) ; data := 0 ;
    Mutex.unlock m ;;
val write : unit -> unit = <fun>

# let writer () =
    while true do write () ; Thread.delay (Random.float 1.5) done ;;
val writer : unit -> unit = <fun>
```

We create a reader and six writers to test these functions.

```
ignore (Thread.create writer ());
for i=0 to 5 do ignore(Thread.create reader i) done;
while true do Thread.delay 5. done ;;
```

This solution guarantees that the writer and the readers cannot have access to the data at the same time. On the contrary, nothing guarantees that the writer could ever "fufill his officé', there we are confronted again with a case of starvation.

Synchronous Communication

Module `Event` from the thread library implements the communication of assorted values between two processes through particular "communication channels". The effective communication of the value is synchronized through send and receive events.

This model of communication synchronized by events allows the transfer through typed channels of the values of the language, including closures, objects, and events.

It is described in [Rep99].

Synchronization using Communication Events

The primitive communication events are:

- `send c v` sends a value `v` on the channel `c`;
- `receive c` receives a value on the channel `c`

So as to implement the physical action with which they are associated, two events should be synchronized. For this purpose, we introduce an operation of synchronization (`sync`) on events. The sending and receiving of a value are not effective unless the two communicating processes are in phase. If a single process wishes to synchronize itself, the operation gets blocked, waiting for the second process to perform its synchronization. This implies that a sender wishing to synchronize the sending of a value (`sync (send c v)`) can find itself blocked waiting for a synchronization from a receiver (`sync (receive c)`).

Transmitted Values

The communication channels through which the exchanged values travel are typed: Nothing prevents us from creating multiple channels for communicating each type of value. As this communication takes place between Objective Caml threads, any value of the language can be sent on a channel of the same type. This is useful for closures, objects, and also events, for a "relayed" synchronization request.

Module **Event**

The values encapsulated in communication events travel through communication channels of the abstract data type *'a channel*. The creation function for channels is:

```
# Event.new_channel ;;
- : unit -> 'a Event.channel = <fun>
```

Send and receive events are created by a function call:

```
# Event.send ;;
- : 'a Event.channel -> 'a -> unit Event.event = <fun>
# Event.receive ;;
- : 'a Event.channel -> 'a Event.event = <fun>
```

We can consider the functions `send` and `receive` as constructors of the abstract type *'a event*. The event constructed by send does not preserve the information about the type of the value to transmit (type *unit Event.event*). On the other hand, the receive event takes account of it to recover the value during a synchronization. These functions are non-blocking in the sense that the transmission of a value does not take place until the time of the synchronization of two processes by the function:

```
# Event.sync ;;
- : 'a Event.event -> 'a = <fun>
```

This function may be blocking for the sender and the receiver.

There is a non-blocking version:

```
# Event.poll ;;
- : 'a Event.event -> 'a option = <fun>
```

This function verifies that another process is waiting for synchronization.

If this is the case, it performs the transmissions, and returns the value `Some v`, if `v` is the value associated with the event, and `None` otherwise. The received message, extracted by the function `sync`, can be the result of a more or less complicated process, triggering other exchanges of messages.

Example of synchronization. We define three threads. The first, `t1`, sends a chain of characters on channel `c` (function `g`) shared by all the processes. The two others `t2` and `t3` wait for a value on the same channel. Here are the functions executed by the different processes:

```
# let c = Event.new_channel ();;
val c : '_a Event.channel = <abstr>
# let f () =
    let ids = string_of_int (Thread.id (Thread.self ()))
    in print_string ("-------- before  -------" ^ ids) ; print_newline() ;
       let e = Event.receive c
       in print_string ("-------- during  -------" ^ ids) ; print_newline() ;
          let v = Event.sync e
```

```
          in print_string (v ^ " " ^ ids ^ " ") ;
             print_string ("-------- after  -------" ^ ids) ; print_newline() ;;
val f : unit -> unit = <fun>
# let g () =
    let ids = string_of_int (Thread.id (Thread.self ()))
    in print_string ("Start of " ^ ids ^ "\n");
       let e2 = Event.send c "hello"
       in Event.sync e2 ;
          print_string ("End of " ^ ids) ;
          print_newline () ;;
val g : unit -> unit = <fun>
```

The three processes are created and executed:

```
# let t1,t2,t3 = Thread.create f (), Thread.create f (), Thread.create g ();;
val t1 : Thread.t = <abstr>
val t2 : Thread.t = <abstr>
val t3 : Thread.t = <abstr>
# Thread.delay 1.0;;
Start of 5
-------- before  -------6
-------- during  -------6
hello 6 -------- after  -------6
-------- before  -------7
-------- during  -------7
End of 5
- : unit = <unknown constructor>
```

The transmission may block. The trace of `t1` is displayed after the synchronization traces of `t2` and `t3`. Only one of the two processes `t1` or `t2` is really terminated, as the following calls show:

```
# Thread.kill t1;;
- : unit = ()
# Thread.kill t2;;
Uncaught exception: Failure("Thread.kill: killed thread")
```

Example: Post Office

We present, to end this chapter, a slightly more complete example of a concurrent program: modelling a common queue at a number of counters at a post office.

As always in concurrent programming the problems are posed metaphorically, but replace the counters of the post office by a collection of printers and you have the solution to a genuine problem in computing.

Here the policy of service that we propose; it is well tried and tested, rather than original: each client takes a number when he arrives; when a clerk has finished serving

a client, he calls for a number. When his number is called, the client goes to the corresponding counter.

Organization of development. We distinguish in our development *resources*, and *agents*. The former are: the number dispenser, the number announcer, and the windows. The latter are: the clerks and the clients. The resources are modeled by objects which manage their own mutual exclusion mechanisms. The agents are modelled by functions executed by a thread. When an agent wishes to modify or examine the state of an object, it does not itself have to know about or manipulate mutexes, which allows a simplified organization for access to sensitive data, and avoids oversights in the coding of the agents.

The Components

The Dispenser. The number dispenser contains two fields: a counter and a mutex. The only method provided by the distributor is the taking of a new number.

```
# class dispenser () =
    object
      val mutable n = 0
      val m = Mutex.create()
      method take () = let r = Mutex.lock m ; n <- n+1 ; n
                       in Mutex.unlock m ; r
    end ;;
class dispenser :
  unit ->
  object val m : Mutex.t val mutable n : int method take : unit -> int end
```

The mutex prevents two clients from taking a number at the same time. Note the way in which we use an intermediate variable (`r`) to guarantee that the number calculated in the critical section is the same as the one return by the method call.

The Announcer. The announcer contains three fields: an integer (the client number being called); a mutex and a condition variable. The two methods are: (`wait`) which reads the number, and (`call`), which modifies it.

```
# class announcer () =
    object
      val mutable nclient = 0
      val m = Mutex.create()
      val c = Condition.create()

      method wait n =
        Mutex.lock m;
        while n > nclient do Condition.wait c m done;
        Mutex.unlock m;
```

```
    method call () =
      let r = Mutex.lock m ;
              nclient <- nclient+1 ;
              nclient
      in Condition.broadcast c ;
         Mutex.unlock m ;
         r
  end ;;
```

The condition variable is used to put the clients to sleep, waiting for their number. They are all woken up when the method `call` is invoked. Reading or writing access to the called number is protected by the mutex.

The window. The window consists of five fields: a fixed window number (variable `ncounter`); the number of the client being waited for (variable `nclient`); a boolean (variable `available`); a mutex, and a condition variable.

It offers eight methods, of which two are private: two simple access methods (methods `get_ncounter` and `get_nclient`): a group of three methods simulating the waiting period of the clerk between two clients (private method `wait` and public methods `await_arrival`, `await_departure`); a group of three methods simulate the occupation of the window (private method `set_available` and methods `arrive`, `depart`).

```
# class counter (i:int) =
    object(self)
      val ncounter = i
      val mutable nclient = 0
      val mutable available = true
      val m = Mutex.create()
      val c = Condition.create()

      method get_ncounter = ncounter
      method get_nclient  = nclient

      method private wait f =
        Mutex.lock m ;
        while f () do Condition.wait c m done ;
        Mutex.unlock m

      method wait_arrival n = nclient <- n ; self#wait (fun () → available)
      method wait_departure () =  self#wait (fun () → not available)

      method private set_available b =
        Mutex.lock m ;
        available <- b ;
        Condition.signal c ;
        Mutex.unlock m
      method arrive () = self#set_available false
      method leave () = self#set_available true
```

```
    end ;;
```

A post office. We collect these three resources in a record type:

```
# type office = { d : dispenser ; a : announcer ;  cs : counter array } ;;
```

Clients and Clerks

The behaviour of the system as a whole will depend on the three following parameters:

```
# let service_delay = 1.7 ;;
# let arrival_delay = 1.7 ;;
# let counter_delay = 0.5 ;;
```

Each represents the maximum value of the range from which each effective value will be randomly chosen. The first parameter models the time taken to serve a client; the second, the delay between the arrival of clients in the post office; the last, the time it takes a clerk to call a new client after the last one has left.

The Clerk. The work of a clerk consists of looping indefinitely over the following sequence:

1. Call for a number.
2. Wait for the arrival of a client holding the called number.
3. Wait for the departure of the client occupying his counter.

Adding some output, we get the function:

```
# let clerk ((a:announcer), (c:counter)) =
    while true do
      let n = a#call ()
      in Printf.printf "Counter %d calls %d\n" c#get_ncounter n ;
         c#wait_arrival n ;
         c#wait_departure () ;
         Thread.delay (Random.float counter_delay)
    done ;;
val clerk : announcer * counter -> unit = <fun>
```

The Client. A client executes the following sequence:

1. Take a waiting number.

2. Wait until his number is called.
3. Go to the window having called for the number to obtain service.

The only slightly complex activity of the client is to find the counter where they are expected.

We give, for this, the auxiliary function:

```
# let find_counter n cs =
    let i = ref 0 in while cs.(!i)#get_ncounter <> n do incr i done ; !i ;;
val find_counter : 'a -> < get_ncounter : 'a; .. > array -> int = <fun>
```

Adding some output, the principal function of the client is:

```
# let client o =
    let n = o.d#take()
    in Printf.printf "Arrival of client %d\n" n ; flush stdout ;
       o.a#wait n ;
       let ic = find_counter n o.cs
       in o.cs.(ic)#arrive () ;
          Printf.printf "Client %d occupies window %d\n" n ic ;
          flush stdout ;
          Thread.delay (Random.float service_delay) ;
          o.cs.(ic)#leave () ;
          Printf.printf "Client %d leaves\n" n ; flush stdout ;;
val client : office -> unit = <fun>
```

The System

The main programme of the application creates a post office and its clerks (each clerk is a process) then launches a process which creates an infinite stream of clients (each client is also a process).

```
# let main () =
   let o =
    { d = new dispenser();
      a = new announcer();
      cs = (let cs0 = Array.create 5 (new counter 0) in
            for i=0 to 4 do cs0.(i) <- new counter i done;
            cs0)
    }
   in for i=0 to 4 do ignore (Thread.create clerk (o.a, o.cs.(i))) done ;
      let create_clients o = while true do
                               ignore (Thread.create client o) ;
                               Thread.delay (Random.float arrival_delay)
                             done
      in ignore (Thread.create create_clients o) ;
         Thread.sleep () ;;
val main : unit -> unit = <fun>
```

The last instruction puts the process associated with the program to sleep in order to pass control immediately to the other active processes of the application.

Exercises

The Philosophers Disentangled

To solve the possible deadlock of the dining philosophers, it suffices to limit access to the table to four at once. Implement this solution.

More of the Post Office

We suggest the following modification to the post office described on page 614: some impatient clients may leave before there number has been called.

1. Add a method `wait` (with type *`int -> unit`*) to the class `dispenser` which causes the caller to wait while the last number distributed is less than or equal to the parameter of the method (it is necessary to modify `take` so that it emits a signal).
2. Modify the method `await_arrival` of class `counter`, so that it returns the boolean value `true` if the expected client arrives, and `false` if the client has not arrived at the end of a certain time.
3. Modify the class `announcer` by passing it a number dispenser as a parameter and:
   (a) adding a method `wait_until` which returns `true` if the expected number has been called during a given waiting period, and `false` otherwise;
   (b) modifying the method `call` to take a counter as parameter and update the field `nclient` of this counter (it is necessary to add an update method in the `counter` class).
4. Modify the function `clerk` to take fruitless waits into account.
5. Write a function `impatient_client` which simulates the behaviour of an impatient client.

Object Producers and Consumers

This exercise revisits the producer-consumer algorithm with the following variation: the storage warehouse is of finite size (*i.e.* a table rather than a list managed as a FIFO). Also, we propose to make an implementation that uses objects to model resources, like the post office.

1. Define a class `product` with signature:
```
class product : string →
  object
```

```
    val name : string
    method name : string
  end
```

2. Define a class `shop` such that:

```
class show : int →
  object
    val mutable buffer : product array
    val c : Condition.t
    val mutable ic : int
    val mutable ip : int
    val m : Mutex.t
    val mutable np : int
    val size : int
    method dispose : product → unit
    method acquire : unit → product
  end
```

   The indexes `ic` and `ip` are manipulated by the producers and the consumers, respectively. The index `ic` holds the index of the last product taken and `ip` that of the last product stored. The counter `np` gives the number of products in stock. Mutual exclusion and control of the waiting of producers and consumers will be managed by the methods of this class.

3. Define a function `consumer`: *shop → string → unit*.

4. Define a function `create_product` of type *string -> product*. The name given to a product will be composed of the string passed as an argument concatenated with a product number incremented at every invocation of the function.
   Use this function to define `producer`: *shop → string → unit*.

Summary

This chapter tackled the topic of concurrent programming in which a number of processes interact, either through shared memory, or by synchronous communication. The first case represents concurrency for imperative programming. In particular, we have detailed the mechanisms of mutual exclusion whose use permits the synchronization of processes for access to shared memory. Synchronous communication offers a model for concurrency in functional programming. In particular, the possibility of sending closures and synchronization events on communication channels facilitates the composition of calculations carried out in different processes.

The processes used in this chapter are the threads of the Objective Caml `Thread` module.

To Learn More

The first requirements for concurrent algorithms arose from systems programming. For this application, the imperative model of shared memory is the most widely used. For example, the relation of mutual exclusion and semaphores are used to manage shared resources. The different low-level mechanisms of managing processes accessing shared memory are described in [Ari90].

Nonetheless, the possibility of expressing concurrent algorithms in one's favorite languages makes it possible to investigate this kind of algorithm, as presented in [And91]. It may be noted that while the concepts of such algorithms can simplify the solution of certain problems, the production of the corresponding programs is quite hard work.

The model of synchronous communication presented by CML, and followed by the `Event` module, is fully described in [Rep99]. The online version is at the following address:

Link: http://cm.bell-labs.com/cm/cs/who/jhr/index.html

An interesting example is the threaded graphical library `EXene`, implemented in CML under X-Windows. The preceding link contains a pointer to this library.

20

Distributed Programming

With distributed programming, you can build applications running on several machines that work together through a network to accomplish a task. The computation model described here is parallel programming with distributed memory. Local and remote programs communicate using a *network protocol*. The best-known and most widely-used of these is IP (*Internet protocol*) and its TCP and UDP layers. Beginning with these low-level layers, many services are built on the *client-server* model, where a server waits for requests from different clients, processes those requests, and sends responses. As an example, the HTTP protocol allows communication between Web browsers and Web servers. The distribution of tasks between clients and servers is suitable for many different software architectures.

The Objective Caml language offers, through its `Unix` library, various means of communication between programs. Sockets allow communication through the TCP/IP and UDP/IP protocols. This part of the `Unix` library has been ported to Windows. Because you can create "heavyweight" processes with `Unix.fork` as well as lightweight processes with `Thread.create`, you can create servers that accept many requests at once. Finally, an important point when creating a new service is the definition of a protocol appropriate to the application.

Outline of the Chapter

This chapter presents the basic elements of the Internet, sockets, for the purpose of building distributed applications (particularly client-server applications) while detailing the problems in designing communications protocols.

The first section briefly explains the Internet, its addressing system and its main services.

The second section illustrates communications through sockets between different Objective Caml processes, both local and remote.

The third section describes the client-server model, while presenting server programs and universal clients.

The fourth section shows the importance of communications protocols for building network services.

This chapter is best read after the chapters on systems programming (Chapter 18) and on concurrent programming (Chapter 19).

The Internet

The Internet is a network of networks. Their interconnection is organized as a hierarchy of domains, subdomains, and so on, through *interfaces*. An interface is the hardware in a computer that allows it to be connected (typically, an Ethernet card). Some computers may have several interfaces. Each interface has a unique IP address that respects, in general, the interconnection hierarchy. Message routing is also organized hierarchically: from domain to domain; then from domain to subdomains, and so on, until a message reaches its destination interface. Besides their interface addresses, computers usually also have a name, as do domains and subdomains. Some machines have a particular role in the network:

bridges connect one network to another;

routers use their knowledge of the topology of the Internet to route data;

name servers track the correspondence between machine names and network addresses.

The purpose of the Internet protocol (*i.e.*, of the IP) is to make the network of networks into a single entity. This is why one can speak of **the** Internet. Any two machines connected via the Internet can communicate. Many kinds of machines and systems coexist on the Internet. All of them use IP protocols and most of them, the UDP and TCP layers.

The different protocols and services used by the Internet are described in RFC's (Requests For Comments), which can be found on the Jussieu mirror site:

Link: ftp://ftp.lip6.fr/pub/rfc

Internet Protocols and Services

The unit of transfer used by the IP protocol is the *datagram* or *packet*. This protocol in unreliable: it does not assure proper order, safe arrival, or non-duplication of transmitted packets. It only deals with correct routing of packets and signaling of errors

when a packet is unable to reach its destination. Addresses are coded into 32 bits in the current version of the protocol: IPv4. These 32 bits are divided into four fields, each containing values between 0 and 255. IP addresses are written with the four fields separated by periods, for example: `132.227.60.30`.

The IP protocol is in the midst of an important change made necessary by the exhaustion of address space and the growing complexity of routing problems due to the expansion of the Internet. The new version of the IP protocol is IPv6, which is described in [Hui97].

Above IP, two protocols allow higher-level transmissions: UDP (User Datagram Protocol, and TCP (Transfer Control Protocol). These two protocols use IP for communication between machines, also allowing communication between applications (or programs) running on those machines. They deal with correct transmission of information, independent of contents. The identification of applications on a machine is done via a *port number*.

UDP is a connectionless, unreliable protocol: it is to applications as IP is to interfaces. TCP is a connection-oriented, reliable protocol: it manages acknowledgement, retransmission, and ordering of packets. Further, it is capable of optimizing transmission by a windowing technique.

The standard services (applications) of the Internet most often use the client-server model. The server manages requests by clients, offering them a specific service. There is an asymmetry between client and server. The services establish high-level protocols for keeping track of transmitted contents. Among the standard services, we note:

- FTP (File Transfer Protocol);
- TELNET (Terminal Protocol);
- SMTP (Simple Mail Transfer Protocol);
- HTTP (Hypertext Transfer Protocol).

Other services use the client-server model:

- NFS (Network File System);
- X-Windows
- Unix services: rlogin, rwho ...

Communication between applications takes place via sockets. Sockets allow communication between processes residing on possibly different machines. Different processes can read and write to sockets.

The `Unix` *Module and* IP *Addressing*

The `Unix` library defines the abstract type *`inet_addr`* representing Internet addresses, as well as two conversion functions between an internal representation of addresses and strings:

```
# Unix.inet_addr_of_string ;;
```

```
- : string -> Unix.inet_addr = <fun>
# Unix.string_of_inet_addr ;;
- : Unix.inet_addr -> string = <fun>
```

In applications, Internet addresses and port numbers for services (or service numbers) are often replaced by names. The correspondence between names and address or number is managed using databases. The `Unix` library provides functions to request data from these databases and provides datatypes to allow storage of the obtained information. We briefly describe these functions below.

Address tables. The table of addresses (*hosts database*) contains the assocation between machine name(s) and interface address(es). The structure of entries in the address table is represented by:

```
# type host_entry =
    { h_name : string;
      h_aliases : string array;
      h_addrtype : socket_domain;
      h_addr_list : inet_addr array } ;;
```

The first two fields contain the machine name and its aliases; the third contains the address type (see page 627); the last contains a list of machine addresses.

A machine name is obtained by using the function:

```
# Unix.gethostname ;;
- : unit -> string = <fun>
# let my_name = Unix.gethostname() ;;
val my_name : string = "estephe.inria.fr"
```

The functions that query the address table require an entry, either the name or the machine address.

```
# Unix.gethostbyname ;;
- : string -> Unix.host_entry = <fun>
# Unix.gethostbyaddr ;;
- : Unix.inet_addr -> Unix.host_entry = <fun>
# let my_entry_byname = Unix.gethostbyname my_name ;;
val my_entry_byname : Unix.host_entry =
  {Unix.h_name="estephe.inria.fr"; Unix.h_aliases=[|"estephe"|];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[|<abstr>|]}
# let my_addr = my_entry_byname.Unix.h_addr_list.(0) ;;
val my_addr : Unix.inet_addr = <abstr>

# let my_entry_byaddr = Unix.gethostbyaddr my_addr ;;
val my_entry_byaddr : Unix.host_entry =
  {Unix.h_name="estephe.inria.fr"; Unix.h_aliases=[|"estephe"|];
   Unix.h_addrtype=Unix.PF_INET; Unix.h_addr_list=[|<abstr>|]}

# let my_full_name = my_entry_byaddr.Unix.h_name ;;
val my_full_name : string = "estephe.inria.fr"
```

These functions raise the `Not_found` exception in case the request fails.

Table of services. The table of services contains the correspondence between service names and port numbers. The majority of Internet services are standardized. The structure of entries in the table of services is:

```
# type service_entry =
    { s_name : string;
      s_aliases : string array;
      s_port : int;
      s_proto : string } ;;
```

The first two fields are the service name and its eventual aliases; the third field contains the port number; the last field contains the name of the protocol used.

A service is in fact characterized by its port number and the underlying protocol. The query functions are:

```
# Unix.getservbyname ;;
- : string -> string -> Unix.service_entry = <fun>
# Unix.getservbyport ;;
- : int -> string -> Unix.service_entry = <fun>
# Unix.getservbyport 80 "tcp" ;;
- : Unix.service_entry =
{Unix.s_name="www"; Unix.s_aliases=[|"http"|]; Unix.s_port=80;
 Unix.s_proto="tcp"}
# Unix.getservbyname "ftp" "tcp" ;;
- : Unix.service_entry =
{Unix.s_name="ftp"; Unix.s_aliases=[||]; Unix.s_port=21; Unix.s_proto="tcp"}
```

These functions raise the `Not_found` exception if they cannot find the service requested.

Sockets

We saw in chapters 18 and 19 two ways to perform interprocess communication, namely, pipes and channels. These first two methods use a logical model of concurrency. In general, they do not give better performance to the degree that the communicating processes share resources, in particular, the same processor. The third possibility, which we present in this section, uses sockets for communication. This method originated in the Unix world. Sockets allow communication between processes executing on the same machine or on different machines.

Description and Creation

A socket is responsible for establishing communication with another socket, with the goal of transferring information. We enumerate the different situations that may be encountered as well as the commands and datatypes that are used by TCP/IP sockets. The classic metaphor is to compare sockets to telephone sets.

- In order to work, the machine must be connected to the network (`socket`).
- To receive a call, it is necessary to possess a number of the type *`sock_addr`* (`bind`).
- During a call, it is possible to receive another call if the configuration allows it (`listen`).

- It is not necessary to have one's own number to call another set, once the connection is established in both directions (`connect`).

Domains. Sockets belong to different *domains*, according to whether they are meant to communicate internally or externally. The `Unix` library defines two possible domains corresponding to the type constructors:

```
# type socket_domain = PF_UNIX | PF_INET;;
```

The first domain corresponds to local communication, and the second, to communication over the Internet. These are the principal domains for *sockets*.

In the following, we use sockets belonging only to the Internet domain.

Types and protocols. Regardless of their domain, sockets define certain communications properties (reliability, ordering, etc.) represented by the type constructors:

```
# type socket_type = SOCK_STREAM | SOCK_DGRAM | SOCK_SEQPACKET | SOCK_RAW ;;
```

According to the type of socket used, the underlying communications protocol obeys definite characteristics. Each type of communication is associated with a default protocol.

In fact, we will only use the first kind of communication — `SOCK_STREAM` — with the default protocol TCP. This guarantees reliability, order, prevents duplication of exchanged messages, and works in connected mode.

For more information, we refer the reader to the Unix literature, for example [Ste92].

Creation. The function to create sockets is:

```
# Unix.socket ;;
- : Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr = <fun>
```

The third argument allows specification of the protocol associated with communication. The value 0 is interpreted as "the default protocol" associated with the pair (domain, type) argument used for the creation of the socket. The value returned by this function is a file descriptor. Thus such a value can be used with the standard input-output functions in the `Unix` library.

We can create a TCP/IP socket with:

```
# let s_descr = Unix.socket Unix.PF_INET Unix.SOCK_STREAM 0 ;;
val s_descr : Unix.file_descr = <abstr>
```

Warning

> Even though the `socket` function returns a value of type *file_descr*, the system distinguishes descriptors for a files and those associated with sockets. You can use the file functions in the `Unix` library with descriptors for sockets; but an exception is raised when a classical file descriptor is passed to a function expecting a descriptor for a socket.

Closing. Like all file descriptors, a socket is closed by the function:

```
# Unix.close ;;
- : Unix.file_descr -> unit = <fun>
```

When a process finishes via a call to `exit`, all open file descriptors are closed automatically.

Addresses and Connections

A socket does not have an address when it is created. In order to setup a connection between two sockets, the caller must know the address of the receiver.

The address of a socket (TCP/IP) consists of an IP address and a port number. A socket in the Unix domain consists simply of a file name.

```
# type sockaddr =
    ADDR_UNIX of string | ADDR_INET of inet_addr * int ;;
```

Binding a socket to an address. The first thing to do in order to receive calls after the creation of a socket is to *bind* the socket to an address. This is the job of the function:

```
# Unix.bind ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

In effect, we already have a socket descriptor, but the address that is associated with it at creation is hardly useful, as shown by the following example:

```
# let (addr_in, p_num) =
    match Unix.getsockname s_descr with
      Unix.ADDR_INET (a,n) → (a,n)
    | _ → failwith "not INET" ;;
val addr_in : Unix.inet_addr = <abstr>
val p_num : int = 0
# Unix.string_of_inet_addr addr_in ;;
- : string = "0.0.0.0"
```

We need to create a useful address and to associate it with our socket. We reuse our local address `my_addr` as described on page 626 and choose port 12345 which, in general, is unused.

```
# Unix.bind s_descr (Unix.ADDR_INET(my_addr, 12345)) ;;
- : unit = ()
```

Listening and accepting connections. It is necessary to use two operations before our socket is completely operational to receive calls: define its listening capacity and allow it to accept connections. Those are the respective roles of the two functions:

```
# Unix.listen ;;
- : Unix.file_descr -> int -> unit = <fun>
# Unix.accept ;;
- : Unix.file_descr -> Unix.file_descr * Unix.sockaddr = <fun>
```

The second argument to the `listen` function gives the maximum number of connections. The call to the `accept` function waits for a connection request. When `accept` finishes, it returns the descriptor for a socket, the so-called *service socket*. This service socket is automatically linked to an address. The `accept` function may only be applied to sockets that have called `listen`, that is, to sockets that have setup a queue of connection requests.

Connection requests. The function reciprocal to `accept` is;

```
# Unix.connect ;;
- : Unix.file_descr -> Unix.sockaddr -> unit = <fun>
```

A call to *Unix.connect s_descr s_addr* establishes a connection between the local socket *s_descr* (which is automatically bound) and the socket with address *s_addr* (which must exist).

Communication. From the moment that a connection is established between two sockets, the processes owning them can communicate in both directions. The input-output functions are those in the `Unix` module, described in Chapter 18.

Client-server

Interprocess communication between processes on the same machine or on different machines through TCP/IP sockets is a mode of point-to-point asynchronous communication. The reliability of such transmissions is assured by the TCP protocol. It is nonetheless possible to simulate the broadcast to a group of processes through point-to-point communication to all receivers.

The roles of different processes communicating in an application are asymmetric, as a general rule. That description holds for client-server architectures. A server is a process (or several processes) accepting requests and trying to respond to them. The client, itself a process, sends a request to the server, hoping for a response.

Client-server Action Model

A server provides a *service* on a given port by waiting for connections from future clients. Figure 20.1 shows the sequence of principal tasks for a server and a client.

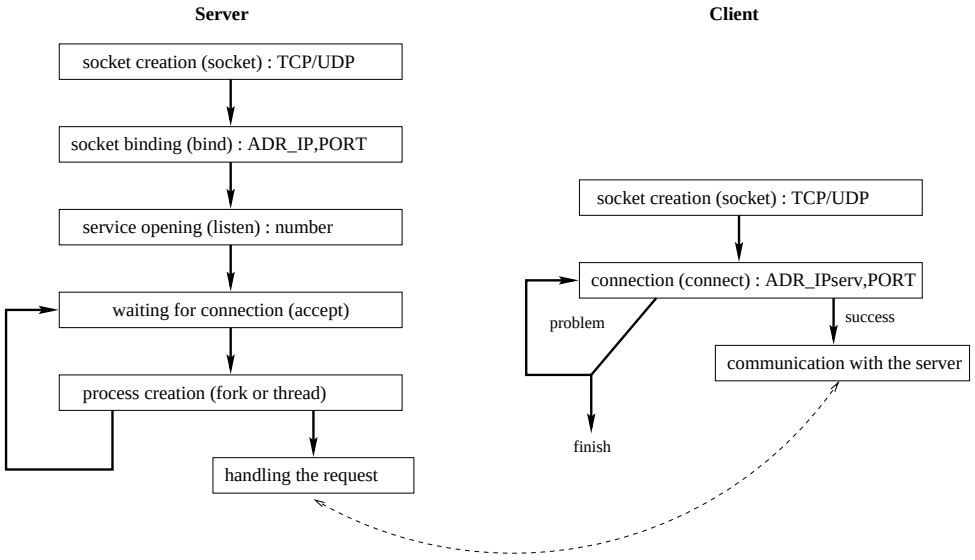


Figure 20.1: Model of a server and client

A client can connect to a service once the server is ready to accept connections (`accept`). In order to make a connection, the client must know the IP number of the server machine and the port number of the service. If the client does not know the IP number, it needs to request name/number resolution using the function `gethostbyname`. Once the connection is accepted by the server, each program can communicate via input-output channels over the sockets created at both ends.

Client-server Programming

The mechanics of client-server programming follows the model described in Figure 20.1. These tasks are always performed. For these tasks, we write generic functions parameterized by particular functions for a given server. As an example of such a program, we describe a server that accepts a connection from a client, waits on a socket until a line of text has been received, converting the line to CAPITALS, and sending back the converted text to the client.

Figure 20.2 shows the communication between the service and different clients[1].

Certain tasks run on the same machine as the server, while others are found on remote machines.

We will see

1. Note of translator: "boulmich" is a colloquial abbreviation for "Boulevard Saint-Michel", one the principal avenues of Quartier Latin in Paris...

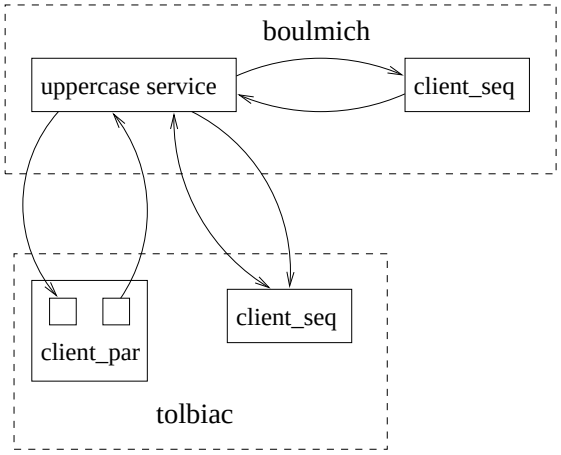


Figure 20.2: CAPITAL service and its clients

1. How to write the code for a "generic server" and instantiate it for our particular capitalization service.
2. How to test the server, without writing the client, by using the `telnet` program.
3. How to create two types of clients:
   - a sequential client, which waits for a response after sending a request;
   - a parallel client, which separates the send and receive tasks.

   Therefore, there are two processes for this client.

Code for the Server

A server may be divided into two parts: waiting for a connection and the following code to handle the connection.

A Generic Server

The generic server function `establish_server` described below takes as its first argument a function for the service (`server_fun`) that handles requests, and as its second argument, the address of the socket in the Internet domain that listens for requests. This function uses the auxiliary function `domain_of`, which extracts the domain of a socket from its address.

In fact, the function `establish_server` is made up of high-level functions from the `Unix` library. This function sets up a connection to a server.

```
# let establish_server server_fun sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
```

```
    in Unix.bind sock sockaddr ;
       Unix.listen sock 3;
       while true do
         let (s, caller) = Unix.accept sock
         in match Unix.fork() with
                0 → if Unix.fork() <> 0 then exit 0 ;
                     let inchan = Unix.in_channel_of_descr s
                     and outchan = Unix.out_channel_of_descr s
                     in server_fun inchan outchan ;
                        close_in inchan ;
                        close_out outchan ;
                        exit 0
              | id → Unix.close s; ignore(Unix.waitpid [] id)
       done ;;
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit = <fun>
```

To finish building a server with a standalone executable that takes a port number parameter, we write a function `main_server` which takes a parameter indicating a service. The function uses the command-line parameter as the port number of a service. The auxiliary function `get_my_addr`, returns the address of the local machine.

```
# let get_my_addr () =
    (Unix.gethostbyname(Unix.gethostname())).Unix.h_addr_list.(0) ;;
val get_my_addr : unit -> Unix.inet_addr = <fun>

# let main_server  serv_fun =
    if Array.length Sys.argv < 2 then Printf.eprintf "usage : serv_up port\n"
    else try
           let port =  int_of_string Sys.argv.(1) in
           let my_address = get_my_addr()
           in establish_server serv_fun  (Unix.ADDR_INET(my_address, port))
         with
           Failure("int_of_string") →
             Printf.eprintf "serv_up : bad port number\n" ;;
val main_server : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

Code for the Service

The general mechanism is now in place. To illustrate how it works, we need to define the service we're interested in. The service here converts strings to upper-case. It waits for a line of text over an input channel, converts it, then writes it on the output channel, flushing the output buffer.

```
# let uppercase_service ic oc =
    try while true do
          let s = input_line ic in
          let r = String.uppercase s
          in output_string oc (r^"\n") ; flush oc
        done
    with _ → Printf.printf "End of text\n" ; flush stdout ; exit 0 ;;
```

```
val uppercase_service : in_channel -> out_channel -> unit = <fun>
```

In order to correctly recover from exceptions raised in the `Unix` library, we wrap the initial call to the service in an *ad hoc* function from the `Unix` library:

```
# let go_uppercase_service () =
    Unix.handle_unix_error main_server uppercase_service ;;
val go_uppercase_service : unit -> unit = <fun>
```

Compilation and Testing of the Service

We group the functions in the file `serv_up.ml`, adding an actual call to the function `go_uppercase_service`. We compile this file, indicating that the `Unix` library is linked in

```
ocamlc -i -custom -o serv_up.exe unix.cma serv_up.ml -cclib -lunix
```

The transcript from this compilation (using the option `-i`) gives:

```
val establish_server :
  (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit
val main_server : (in_channel -> out_channel -> 'a) -> unit
val uppercase_service : in_channel -> out_channel -> unit
val go_uppercase_service : unit -> unit
```

We launch the server by writing:

```
serv_up.exe 1400
```

The port chosen here is 1400. Now the machine where the server was launched will accept connections on this port.

Testing with `telnet`

We can now begin to test the server by using an existing client to send and receive lines of text. The `telnet` utility, which normally is a client of the `telnetd` service on port 23, and used to control a remote connection, can be diverted from this role by passing a machine name and a different port number. This utility exists on several operating systems. To test our server under Unix, we type:

```
$ telnet boulmich 1400
Trying 132.227.89.6...
Connected to boulmich.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
```

The IP address for `boulmich` is 132.227.89.6 and its complete name, which contains its domain name, is `boulmich.ufr-info-p6.jussieu.fr`. The text displayed by `telnet` indicates a successful connection to the server. The client waits for us to type on the keyboard, sending the characters to the server that we have launched on `boulmich` on port 1400. It waits for a response from the server and displays:

```
The little cat is dead.
THE LITTLE CAT IS DEAD.
We obtained the expected result.
WE OBTAINED THE EXPECTED result.
```

The phrases entered by the user are in lower-case and those sent by the server are in upper-case. This is exactly the role of this service, to perform this conversion.

To exit from the client, we need to close the window where it was run, by executing the `kill` command. This command will close the client's socket, causing the server's socket to close as well. When the server displays the message "`End of text`," the process associated with the service terminates.

The Client Code

While the server is naturally parallel (we would like to handle a particular request while accepting others, up to some limit), the client may or may not be so, according to the nature of the application. Below we give two versions of the client. Beforehand, we present two functions that will be useful for writing these clients.

The function `open_connection` from the `Unix` library allows us to obtain a couple of input-output channels for a socket.

The following code is contained in the language distribution.

```
# let open_connection sockaddr =
    let domain = domain_of sockaddr in
    let sock = Unix.socket domain Unix.SOCK_STREAM 0
    in try Unix.connect sock sockaddr ;
           (Unix.in_channel_of_descr sock , Unix.out_channel_of_descr sock)
      with exn → Unix.close sock ; raise exn ;;
val open_connection : Unix.sockaddr -> in_channel * out_channel = <fun>
```

Similarly, the function `shutdown_connection` closes down a socket.

```
# let shutdown_connection inchan =
    Unix.shutdown (Unix.descr_of_in_channel inchan) Unix.SHUTDOWN_SEND ;;
val shutdown_connection : in_channel -> unit = <fun>
```

A Sequential Client

From these functions, we can write the main function of a sequential client. This client takes as its argument a function for sending requests and receiving responses.

This function analyzes the command line arguments to obtain connection parameters before actual processing.

```
# let main_client client_fun  =
    if Array.length Sys.argv < 3
    then Printf.printf "usage :  client server port\n"
    else let server = Sys.argv.(1) in
         let server_addr =
           try  Unix.inet_addr_of_string server
           with Failure("inet_addr_of_string") →
                  try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                  with Not_found →
                         Printf.eprintf "%s : Unknown server\n" server ;
                         exit 2
         in try
              let port = int_of_string (Sys.argv.(2)) in
              let sockaddr = Unix.ADDR_INET(server_addr,port) in
              let ic,oc = open_connection sockaddr
              in client_fun ic oc ;
                 shutdown_connection ic
            with Failure("int_of_string") → Printf.eprintf "bad port number";
                                            exit 2 ;;
val main_client : (in_channel -> out_channel -> 'a) -> unit = <fun>
```

All that is left is to write the function for client processing.

```
# let client_fun ic oc =
    try
      while true do
        print_string  "Request : " ;
        flush stdout ;
        output_string oc ((input_line stdin)^"\n") ;
        flush oc ;
        let r = input_line ic
        in Printf.printf "Response : %s\n\n" r;
           if r = "END" then ( shutdown_connection ic ; raise Exit) ;
      done
    with
        Exit → exit 0
      | exn → shutdown_connection ic ; raise exn  ;;
val client_fun : in_channel -> out_channel -> unit = <fun>
```

The function `client_fun` enters an infinite loop which reads from the keyboard, sends a string to the server, gets back the transformed upper-case string, and displays it. If the string is `"END"`, then the exception `Exit` is raised in order to exit the loop. If another exception is raised, typically if the server has shut down, the function ceases its calculations.

The client program thus becomes:

```
# let go_client () = main_client client_fun ;;
val go_client : unit -> unit = <fun>
```

We place all these functions in a file named `client_seq.ml`, adding a call to the function `go_client`. We compile the file with the following command line:

```
ocamlc -i -custom -o client_seq.exe unix.cma client_seq.ml -cclib -lunix
```

We run the client as follows:

```
$ client_seq.exe boulmich 1400
Request : The little cat is dead.
Response: THE LITTLE CAT IS DEAD.

Request : We obtained the expected result.
Response: WE OBTAINED THE EXPECTED RESULT.

Request : End
Response: END
```

The Parallel Client with `fork`

The parallel client mentioned divides its tasks between two processes: one for sending, and the other for receiving. The processes share the same socket. The functions associated with each of the processes are passed to them as parameters.

Here is the modified program:

```
# let main_client client_parent_fun client_child_fun  =
    if Array.length Sys.argv < 3
    then Printf.printf "usage :  client server port\n"
    else
      let server = Sys.argv.(1) in
      let server_addr =
        try  Unix.inet_addr_of_string server
        with Failure("inet_addr_of_string")
               → try  (Unix.gethostbyname server).Unix.h_addr_list.(0)
                  with Not_found →
                         Printf.eprintf "%s : unknown server\n" server ;
                         exit 2
      in try
           let port = int_of_string (Sys.argv.(2)) in
           let sockaddr = Unix.ADDR_INET(server_addr,port) in
           let ic,oc = open_connection sockaddr
           in match Unix.fork () with
                  0 → if Unix.fork() = 0 then client_child_fun oc ;
                      exit 0
                | id → client_parent_fun ic ;
                       shutdown_connection ic ;
                       ignore (Unix.waitpid [] id)
         with
            Failure("int_of_string") → Printf.eprintf "bad port number" ;
                                       exit 2 ;;
```

```
val main_client : (in_channel -> 'a) -> (out_channel -> unit) -> unit = <fun>
```

The expected behavior of the parameters is: the (grand)child sends the request and the parent receives the response.

This architecture has the effect that if the child needs to send several requests, then the parent receives the responses to requests as each is processed. Consider again the preceding example for capitalizing strings, modifying the client side program. The client reads the text from one file, while writing the response to another file. For this we need a function that copies from one channel, `ic`, to another, `oc`, respecting our little protocol (that is, it recognizes the string `"END"`).

```
# let copy_channels ic oc =
    try while true do
          let s = input_line ic
          in if s = "END" then raise End_of_file
             else (output_string oc (s^"\n"); flush oc)
        done
    with End_of_file → () ;;
val copy_channels : in_channel -> out_channel -> unit = <fun>
```

We write the two functions for the child and parent using the parallel client model:

```
# let child_fun in_file out_sock =
    copy_channels in_file out_sock ;
    output_string out_sock ("FIN\n") ;
    flush out_sock ;;
val child_fun : in_channel -> out_channel -> unit = <fun>
# let parent_fun out_file in_sock = copy_channels in_sock out_file ;;
val parent_fun : out_channel -> in_channel -> unit = <fun>
```

Now we can write the main client function. It must collect two extra command line parameters: the names of the input and output files.

```
# let go_client () =
   if Array.length Sys.argv < 5
     then Printf.eprintf "usage : client_par server port filein fileout\n"
   else let in_file = open_in Sys.argv.(3)
        and out_file = open_out Sys.argv.(4)
        in main_client (parent_fun out_file) (child_fun in_file) ;
           close_in in_file ;
           close_out out_file ;;
val go_client : unit -> unit = <fun>
```

We gather all of our material into the file `client_par.ml` (making sure to include a call to `go_client`), and compile it. We create a file `toto.txt` containing the text to be converted:

```
The little cat is dead.
We obtained the expected result.
```

We can test the client by typing:

```
client_par.exe boulmich 1400 toto.txt result.txt
```

The file `result.txt` contains the text:

```
$ more result.txt
THE LITTLE CAT IS DEAD.
WE OBTAINED THE EXPECTED RESULT.
```

When the client finishes, the server always displays the message `"End of text"`.

Client-server Programming with Lightweight Processes

The preceding presentation of code for a generic server and a parallel client created processes via the `fork` primitive in the `Unix` library. This works well under Unix; many Unix services are implemented by this technique. Unfortunately, the same cannot be said for Windows. For portability, it is preferable to write client-server code with lightweight processes, which were presented in Chapter 19. In this case, it becomes necessary to examine the interactions among different server processes.

Threads and the `Unix` Library

The simultaneous use of lightweight processes and the `Unix` library causes all active threads to block if a system call does not return immediately. In particular, reads on file descriptors, including those created by `socket`, are blocking.

To avoid this problem, the `ThreadUnix` library reimplements most of the input-output functions from the `Unix` library. The functions defined in that library will only block the thread which is actually making the system call. As a consequence, input and output is handled with the low-level functions `read` and `write` found in the `ThreadUnix` library.

For example, the standard function for reading a string of characters, `input_line`, is redefined in such a way that it does not block other threads while reading a line.

```
# let my_input_line  fd =
    let s = " "  and  r = ref ""
    in while (ThreadUnix.read fd s 0 1 > 0) && s.[0] <> '\n' do r := !r ^s done ;
       !r ;;
val my_input_line : Unix.file_descr -> string = <fun>
```

Classes for a Server with Threads

Now let us recycle the example of the CAPITALIZATION service, this time giving a version using lightweight processes. Shifting to threads poses no problem for our little application on either the server side or the client side, which start processes independently.

Earlier, we built a generic server parameterized over a service function. We were able to achieve this kind of abstraction by relying on the functional aspect of the Objective Caml language. Now we are about to use the object-oriented extensions to the language to show how objects allow us to achieve a comparable abstraction.

The server is organized into two classes: `serv_socket` and `connection`. The first of these handles the service startup, and the second, the service itself. We have introduced some print statements to trace the main stages of the service.

The `serv_socket` class. has two instance variables: `port`, the port number for the service, and `socket`, the socket for listening. When constructing such an object, the initializer opens the service and creates this socket. The `run` method accepts connections and creates a new `connection` object for handling requests. The *`serv_socket`* uses the *`connection`* class described in the following paragraph. Usually, this class must be defined before the *`serv_socket`* class.

```
# class serv_socket p =
    object (self)
      val port = p
      val mutable sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

      initializer
        let my_address = get_my_addr ()
        in Unix.bind sock (Unix.ADDR_INET(my_address,port)) ;
           Unix.listen sock 3

      method private client_addr = function
          Unix.ADDR_INET(host,_) → Unix.string_of_inet_addr host
        | _ → "Unexpected client"

      method run () =
        while(true) do
          let (sd,sa) = ThreadUnix.accept sock in
          let connection = new connection(sd,sa)
          in Printf.printf "TRACE.serv: new connection from %s\n\n"
                           (self#client_addr sa) ;
          ignore (connection#start ())
        done
    end ;;
class serv_socket :
  int ->
  object
    val port : int
    val mutable sock : Unix.file_descr
    method private client_addr : Unix.sockaddr -> string
    method run : unit -> unit
  end
```

It is possible to refine the server by inheriting from this class and redefining the `run` method.

The `connection` class. The instance variables in this class, `s_descr` and `s_addr`, are initialized to the descriptor and the address of the socket created by `accept`. The methods are `start`, `run`, and `stop`. The `start` creates a thread calling the two other methods, and returns its thread identifier, which can be used by the calling instance of `serv_socket`. The `run` method contains the core functionality of the service. We have slightly modified the termination condition for the service: we exit on receipt of an empty string. The `stop` service just closes the socket descriptor for the service.

Each new connection has an associated number obtained by calling the auxiliary function `gen_num` when the instance is created.

```
# let gen_num = let c = ref 0 in (fun () → incr c; !c) ;;
val gen_num : unit -> int = <fun>
# exception Done ;;
exception Done
# class connection (sd,sa) =
    object (self)
      val s_descr = sd
      val s_addr = sa
      val mutable number = 0
      initializer
        number <- gen_num();
        Printf.printf "TRACE.connection : object %d created\n" number ;
        print_newline()

      method start () =  Thread.create (fun x → self#run x ; self#stop x) ()

      method stop() =
        Printf.printf "TRACE.connection : object finished %d\n" number ;
        print_newline () ;
        Unix.close s_descr

     method run () =
       try
         while true do
           let line =  my_input_line s_descr
           in if (line = "") or (line = "\013") then raise Done ;
              let result = (String.uppercase line)^"\n"
              in ignore (ThreadUnix.write s_descr result 0 (String.length result))
         done
       with
          Done → ()
        | exn  → print_string (Printexc.to_string exn) ; print_newline()
    end ;;
class connection :
  Unix.file_descr * 'a ->
  object
    val mutable number : int
    val s_addr : 'a
    val s_descr : Unix.file_descr
```

```
    method run : unit -> unit
    method start : unit -> Thread.t
    method stop : unit -> unit
  end
```

Here again, by inheritance and redefinition of the `run` method, we can define a new service.

We can test this new version of the server by running the `protect_serv` function.

```
# let go_serv () = let s = new serv_socket 1400 in s#run () ;;
# let protect_serv () = Unix.handle_unix_error go_serv () ;;
```

Multi-tier Client-server Programming

Even though the client-server relation is asymmetric, nothing prevents a server from being the client of another service. In this way, we have a communication hierarchy. A typical client-server application might be the following:

- a mail client presents a friendly user interface;
- a word-processing program is run, followed by an interaction with the user;
- the word-processing program accesses a database.

One of the goals of client-server applications is to alleviate the processing of centralized machines. Figure 20.3 shows two client-server architectures with three tiers.

Each tier may run on a different machine. The user interface runs on the machine running the user mail application. The processing part is handled by a machine shared by a collection of users, which itself sends requests to a remote database server. With this application, a particular piece of data may be sent to the user mail application or to the database server.

Some Remarks on the Client-server Programs

In the preceding sections, we constructed servers for a simple CAPITALIZATION service. Each server used a different approach for its implementation. The first such server used the Unix fork mechanism. Once we built that server, it became possible to test it with the telnet client supplied with the Unix, Windows, and MacOS operating systems. Next, we built a simple first client. We were then able to test the client and server together. Clients may have tasks to manage between communications. For this purpose, we built the `client_par.exe` client, which separates reading from writing by using forks. A new kind of server was built using threads to clearly show the relative independence of the server and the client, and to bring input-output into this setting. This server was organized into two easily-reused classes. We note that both functional programming and object-oriented programming support the separation of "mechanical," reusable code from code for specialized processing.

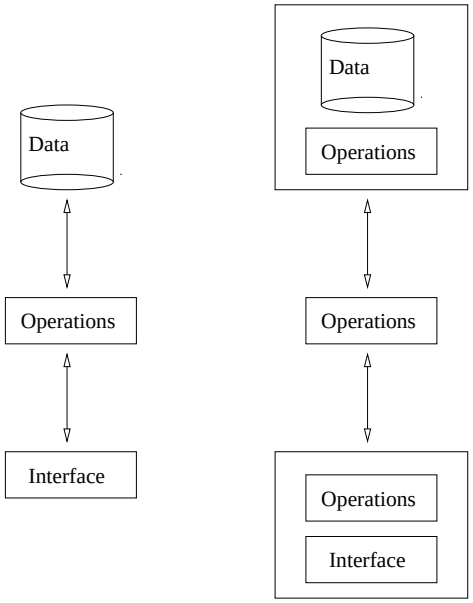


Figure 20.3: Different client-server architectures

Communication Protocols

The various client-server communications described in the previous section consisted of sending a string of characters ending in a carriage-return and receiving another. However simple, this communication pattern defines a protocol. If we wish to communicate more complex values, such as floats, matrices of floats, a tree of arithmetic expressions, a closure, or an object, we introduce the problem of encoding these values. Many solutions exist according to the nature of the communicating programs, which can be characterized by the implementation language, the machine architecture, and in certain cases, the operating system. Depending on the machine architecture, integers can be represented in many different ways (most significant bits on the left, on the right, use of tag bits, and size of a machine word). To communicate a value between different programs, it is necessary to have a common representation of values, referred to as the *external* representation[2]. More structured values, such as records, just as integers, must have an external representation. Nonetheless, there are problems when certain languages allow constructs, such as bit-fields in C, which do not exist in other languages. Passing functional objects or objects, which contain pieces of code, poses a new difficulty. Is the code byte-compatible between the sender and receiver, and does there exist a mechanism for dynamically loading the code? As a general rule, the problem is simplified by supposing that the code exists on both sides. It is not the

2. Such as the XDR representation (eXternal Data Representation), which was designed for C programs.

code itself that is transmitted, but information that allows it to be retrieved. For an object, the instance variables are communicated along with the object's type, which allows retrieval of the object's methods. For a closure, the environment is sent along with the address of its code. This implies that the two communicating programs are actually the same executable.

A second difficulty arises from the complexity of linked exchanges and the necessity of synchronizing communications involving many programs.

We first present text protocols, later discussing acknowledgements and time limits between requests and responses. We also mention the difficulty of communicating internal values, in particular as it relates to interoperability between programs written in different languages.

Text Protocol

Text protocols, that is, communication in ASCII format, are the most common because they are the simplest to implement and the most portable. When a protocol becomes complicated, it may become difficult to implement. In this setting, we define a grammar to describe the communication format. This grammar may be rich, but it will be up to the communicating programs to handle the work of coding and interpreting the text strings sent and received.

As a general rule, a network application does not allow viewing the different layers of protocols in use. This is typified by the case of the HTTP protocol, which allows a browser to communicate with a Web site.

The HTTP *Protocol*

The term "HTTP" is seen frequently in advertising. It corresponds to the communication protocol used by Web applications. The protocol is completely described on the page of the W3 Consortium:

Link: http://www.w3.org

This protocol is used to send requests from browsers (Communicator, Internet Explorer, Opera, etc.) and to return the contents of requested pages. A request made by a browser contains the name of the protocol (HTTP), the name of the machine (www.ufr-info-p6.jussieu.fr), and the path of the requested page (/Public/Localisation/index.html). Together these components define a URL (Uniform Resource Locator):

```
http://www.ufr-info-p6.jussieu.fr/Public/Localisation/index.html
```

When such a URL is requested by a browser, a connection over a socket is established between the browser and the server running on the indicated server, by default on port 80. Then the browser sends a request in the HTTP format, like the following:

```
GET /index.html HTTP/1.0
```

The server responds in the protocol HTTP, with a header:

```
HTTP/1.1 200 OK
Date: Wed, 14 Jul 1999 22:07:48 GMT
Server: Apache/1.3.4 (Unix) PHP/3.0.6 AuthMySQL/2.20
Last-Modified: Thu, 10 Jun 1999 12:53:46 GMT

Accept-Ranges: bytes
Content-Length: 3663
Connection: close
Content-Type: text/html
```

This header indicates that the request has been accepted (code 200 OK), the kind of server, the modification date for the page, the length of the send page and the type of content which follows. Using the GET commmand in the protocol (HTTP/1.0), only the HTML page is transferred. The following connection with telnet allows us to see what is actually transmitted:

```
$ telnet www.ufr-info-p6.jussieu.fr 80
Trying 132.227.68.44...
Connected to triton.ufr-info-p6.jussieu.fr.
Escape character is '^]'.
GET


<!-- index.html -->
<HTML>
<HEAD>
<TITLE>Serveur de l'UFR d'Informatique de Pierre et Marie Curie</TITLE>
</HEAD>
<BODY>

<IMG SRC="/Icons/upmc.gif" ALT="logo-P6" ALIGN=LEFT HSPACE=30>
Unit&eacute; de Formation et de Recherche 922 - Informatique<BR>
Universit&eacute; Pierre et Marie Curie<BR>
4, place Jussieu<BR>
75252 PARIS Cedex 05, France<BR><P>
....
</BODY>
</HTML>
<!-- index.html -->

Connection closed by foreign host.
```

The connection closes once the page has been copied. The base protocol is in text mode so that the language may be interpreted. Note that images are not transmitted with the page. It is up to the browser, when analyzing the syntax of the HTML page, to

observe anchors and images (see the `IMG` tags in the transmitted page). At this time, the browser sends a new request for each image encountered in the HTML source; there is a new connection for each image. The images are displayed when they are received. For this reason, images are often displayed in parallel.

The HTTP protocol is simple enough, but it transports information in the HTML language, which is more complex.

Protocols with Acknowledgement and Time Limits

When a protocol is complex, it is useful that the receiver of a message indicate to the sender that it has received the message and that it is grammatically correct. The client blocks while waiting for a response before working on its tasks. If the part of the server handling this request has a difficulty interpreting the message, the server must indicate this fact to the client rather than ignoring the request. The HTTP protocol has a system of error codes. A correct request results in the code 200. A badly-formed request or a request for an unauthorized page results in an error code 4xx or 5xx according to the nature of the error. These error codes allow the client to know what to do and allow the server to record the details of such incidents in its log files.

When the server is in an inconsistent state, it can always accept a connection from a client, but risks never sending it a response over the socket. For avoiding these blocking waits, it is useful to fix a limit to the time for transmission of the response. After this time has elapsed, the client supposes that the server is no longer responding. Then the client can close this connection in order to go on to its other work. This is how WWW browsers work. When a request has no response after a certain time, the browser decides to indicate that to the user. Objective Caml has input-output with time limits. In the `Thread` library, the functions `wait_time_read` and `wait_time_write` suspend execution until a character can be read or written, within a certain time limit. As input, these function take a file descriptor and a time limit in seconds: *`Unix.file_descr -> float -> bool`*. If the time limit has passed, the function returns `false`, otherwise the I/O is processed.

Transmitting Values in their Internal Representation

The interest in transmission of internal values comes from simplifying the protocol. There is no longer any need to encode and decode data in a textual format. The inherent difficulty in sending and receiving values in their internal representation are the same as those encountered for persistent values (see the `Marshal` library, page 228). In effect, reading or writing a value in a file is equivalent to receiving the same value over a socket.

Functional Values

In the case of transmitting a closure between two Objective Caml programs, the code in the closure is not sent, only its environment and its code pointer (see figure 12.9 page 334). For this strategy to work, it is necessary that the server possess the same code in the same memory location. This implies that the same program is running on the server as on the client. Nothing, however, prevents the two programs from running different parts of the code at the same time. We adapt the matrix calculation service by sending a closure with an environment containing the data for calculation. When it is received, the server applies this closure to () and the calculation begins.

Interoperating with Different Languages

The interest in text protocols is that they are independent of implementation languages for clients and servers. In effect, the ASCII code is always known by programming languages. Therefore, it is up to the client and to the server to analyze syntactically the strings of characters transmitted. An example of such an open protocol is the simulation of soccer players called RoboCup.

Soccer Robots

A soccer team plays against another team. Each member of the team is a client of a referee server. The players on the same team cannot communicate directly with each other. They must send information through the server, which retransmits the dialog. The server shows a part of the field, according to the player's position. All these communications follow a text protocol. A Web page that describes the protocol, the server, and certain clients:

Link: http://www.robocup.org/

The server is written in C. The clients are written in different languages: C, C++, SmallTalk, Objective Caml, etc. Nothing prevents a team from fielding players written in different languages.

This protocol responds to the interoperability needs between programs in different implementation languages. It is relatively simple, but it requires a particular syntax analyzer for each family of languages.

Exercises

The suggested exercises allow you to try different types of distributed applications. The first offers a new network service for setting the time on client machines. The second exercise shows how to use resources on different machines to distribute a calculation.

Service: Clock

This exercise consists of implementing a "clock" service that gives the time to any client. The idea is to have a reference machine to set the time for different machines on a network.

1. Define a protocol for transmitting a date containing the day, month, hour, minute, and second.
2. Write the function or the class for the service reusing one of the generic servers presented in the Chapter. The service sends date information over each accepted connection, then closes the socket.
3. Write the client , which sets the clock every hour.
4. Keep track of time differences when requests are sent.

A Network Coffee Machine

We can build a little service that simulates a beverage vending machine. A summary description of the protocol between the client and service is as follows:

- when it makes a connection, the client receives a list of available drinks;
- it then sends to the server its beverage choice;
- the server returns the price of the beverage;
- the client sends the requested price, or some other sum;
- the server responds with the name of the chosen beverage and shows the change tendered.

The server may also respond with an error message if it has not understood a request, does not have enough change, etc. A client request always contains just one piece of information.

The exchanges between client and server are in the form of strings of characters. The different components of a message are separated by two periods and all strings end in `:$\n`.

The service function communicates with the coffee machine by using a file to pass commands and a hash table for recovering drinks and change.

This exercise will make use of sockets, lightweight processes with a little concurrency, and objects.

1. Rewrite the function `establish_server` using the primitives in `ThreadUnix`.
2. Write two functions, `get_request` and `send_answer`. The first function reads and encodes a request and the second formats and sends a response beginning with a list of strings of characters.
3. Write a class `cmd_fifo` to manage pending commands. Each new command is assigned a unique number. For this purpose, implement a class `num_cmd_gen`.

4. Write a class `ready_table` for stocking the machine with drinks.
5. Write the class `machine` that models the coffee machine. The class contains a method `run` that loops through the sequence: wait for a command, then execute it, as long as there remain drinks available. Define a type ***`drink_descr`*** indicating, for each drink: its name, the quantity in stock, the quantity that will remain after satisying pending commands, and its price. We can use an auxiliary function `array_index` which returns the index of the first element in a table satisfying a criterion passed as a parameter.
6. Write the service function `waiter`.
7. Write the principal function `main` that obtains a port number for the service from the command line and performs a number of initialization tasks. In particular, the coffee machine executes in a process.

Summary

This chapter presented the new possibilities offered by distributed programming. Communication between programs is accomplished with the fundamental mechanism of sockets, used by low-level Internet protocols. The action models used by clients and servers are asymmetric. Communication between clients and servers use some notion of protocol, most often using plain text. Functional programming and object-oriented programming allow us to easily build distributed applications. The client-server model lends itself to different software architectures, with two or three tiers, according to the distribution of tasks between them.

To Learn More

Communication between distant Objective Caml programs can be rich. Use of text protocols is greatly facilitated by utilities for syntactic analysis (see Chapter 11). The persistence mechanism offered by the `Marshal` library (see Chapter 8) allows sending complex data in its internal format including functional values if the two communicating programs are the same. The main deficiency of that mechanism is the inability to send instances of classes. One solution to that problem is to use an ORB (Object Request Broker) to transmit objects or invoke remote methods. This architecture already exists in many object-oriented languages in the form of the CORBA (Common ORB Architecture) standard. This standard from the OMG (Object Management Group), which debuted in 1990, allows the use of remote objects, and is independent of the implementation language used to create classes.

Link: http://www.omg.org

The two principal functions of CORBA are the ability to send objects to a remote program and, especially, the ability to use the same object at many locations in a network, in order to call methods which can modify its instance variables. Further, this standard is independent of the language used to implement these remote objects. To

that end, an ORB furnishes a description language for interfaces called IDL (Interface Definition Language), in the manner of `CAMLIDL` for the interface between Objective Caml and C. For the moment, there is no ORB that works with Objective Caml, but it is possible to build one, since the IDL language is an abstraction of object-oriented languages with classes. To simplify, CORBA furnishes a software bus (IIOP) that allows transferring and addressing remote data.

The ability to reference the same object at many points in a network simulates distributed shared memory, which is not without problems for automatic garbage collection.

The ability to reference a remote object does not cause code to be transferred. One can only receive a copy of an instance of a class if the class exists on the server. For certain client-server applications, it may be necessary to use dynamic loading of code (such as in Java applets) and even to migrate processes along with their code. An interesting example of dynamic loading of remote code is the MMM browser built in Objective Caml by François Rouaix:

Link: http://caml.inria.fr/~rouaix/mmm/

This browser can be used conventionally to view Web pages, but can also load Objective Caml applets from a server and run them in a graphical window.

21

Applications

The first application is really a toolbox to facilitate the construction of client-server applications which transmit Objective Caml values. To build an application using the toolbox, one need only implement serialization functions for the values to be transmitted, then apply a functor to obtain an abstract class for the server, then add the application's processing function by means of inheritance.

The second application revisits the robot simulation, presented on page 550, and adapts it to the client-server model. The server represents the world in which the robot clients move around. We thus simulate distributed memory shared by a group of clients possibly located on various machines on the network.

The third application is an implementation of some small HTTP servers (called *servlets*). A server knows how to respond to an HTTP request such as a request to retrieve an HTML page. Moreover, it is possible to pass values in these requests using the CGI format of HTTP servers. We will use this functionality right away to construct a server for requests on the association database, described on page 148. As a client, we will use a Web browser to which we will send an initial page containing the query form.

Client-server Toolbox

We present a collection of modules to enable client-server interactions among Objective Caml programs. This toolbox will be used in the two applications that follow.

A client-server application differs from others in the protocol that it uses and in the processing that it associates with the protocol. Otherwise, all such applications use very similar mechanisms: waiting for a connection, starting a separate process to handle the connection, and reading and writing sockets.

Taking advantage of Objective Caml's ability to combine modular genericity and extension of objects, we will create a collection of functors which take as argument a

communications protocol and produce generic classes implementing the mechanisms of clients and of servers. We can then subclass these to obtain the particular processing we need.

Protocols

A communications protocol is a type of data that can be translated into a sequence of characters and transmitted from one machine to another via a socket. This can be described using a signature.

```
# module type PROTOCOL =
    sig
      type t
      val to_string : t → string
      val of_string : string → t
    end ;;
```

The signature requires that the data type be monomorphic; yet we can choose a data type as complex as we wish, as long as we can translate it to a sequence of characters and back. In particular, nothing prevents us from using objects as our data.

```
# module Integer =
    struct
      class integer x =
        object
          val v = x
          method x = v
          method str = string_of_int v
        end
      type t = integer
      let to_string o = o#str
      let of_string s = new integer (int_of_string s)
    end ;;
```

By making some restrictions on the types of data to be manipulated, we can use the module `Marshal`, described on page 229, to define the translation functions.

```
# module Make_Protocol = functor ( T : sig type t end ) →
    struct
      type t = T.t
      let to_string (x:t) = Marshal.to_string x [Marshal.Closures]
      let of_string s = (Marshal.from_string s 0 : t)
    end ;;
```

Communication

Since a protocol is a type of value that can be translated into a sequence of characters, we can make these values persistent and store them in a file.

The only difficulty in reading such a value from a file when we do not know its type is that *a priori* we do not know the size of the data in question. And since the file in question is in fact a socket, we cannot simply check an end of file marker. To solve this problem, we will write the size of the data, as a number of characters, before the data itself. The first twelve characters will contain the size, padded with spaces.

The functor `Com` takes as its parameter a module with signature `PROTOCOL` and defines the functions for transmitting and receiving values encoded using the protocol.

```
# module Com = functor (P : PROTOCOL) →
    struct
      let send fd m =
        let mes = P.to_string m in
        let l = (string_of_int (String.length mes)) in
        let buffer = String.make 12 ' ' in
          for i=0 to (String.length l)-1 do buffer.[i] <- l.[i] done ;
          ignore (ThreadUnix.write fd buffer 0 12) ;
          ignore (ThreadUnix.write fd mes 0 (String.length mes))

      let receive fd =
        let buffer = String.make 12 ' '
        in
          ignore (ThreadUnix.read fd buffer 0 12) ;
          let l = let i = ref 0
          in while (buffer.[!i]<>' ') do incr i done ;
             int_of_string (String.sub buffer 0 !i)
        in
          let buffer = String.create l
          in ignore (ThreadUnix.read fd buffer 0 l) ;
             P.of_string buffer
    end ;;
module Com :
  functor(P : PROTOCOL) ->
    sig
      val send : Unix.file_descr -> P.t -> unit
      val receive : Unix.file_descr -> P.t
    end
```

Note that we use the functions `read` and `write` from module `ThreadUnix` and not those from module `Unix`; this will permit us to use our functions in a thread without blocking the execution of other processes.

Server

A server is built as an abstract class parameterized by the type of data in the protocol. Its constructor takes as arguments a port number and the maximum number of simultaneous connections allowed. The method for processing a request is abstract; it must be implemented in a subclass of `server` to obtain a concrete class.

```
# module Server = functor (P : PROTOCOL) →
    struct
      module Com = Com (P)

      class virtual ['a] server p np =
        object (s)
          constraint 'a = P.t
          val port_num = p
          val nb_pending = np
          val sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0

          method start =
            let host = Unix.gethostbyname (Unix.gethostname()) in
            let h_addr = host.Unix.h_addr_list.(0) in
            let sock_addr = Unix.ADDR_INET(h_addr, port_num) in
              Unix.bind sock sock_addr ;
              Unix.listen sock nb_pending ;
              while true do
                let (service_sock, client_sock_addr) = ThreadUnix.accept sock
                in ignore (Thread.create s#process service_sock)
              done
          method send = Com.send
          method receive = Com.receive
          method virtual process : Unix.file_descr → unit
        end
    end ;;
```

In order to show these ideas in use, let us revisit the CAPITAL service, adding the capability of sending lists of strings.

```
# type message = Str of string | LStr of string list ;;
# module Cap_Protocol = Make_Protocol (struct type t=message end) ;;
# module Cap_Server = Server (Cap_Protocol) ;;

# class cap_server p np =
    object (self)
      inherit [message] Cap_Server.server p np
      method process fd =
        match self#receive fd with
            Str s  → self#send fd (Str (String.uppercase s)) ;
                     Unix.close fd
          | LStr l → self#send fd (LStr (List.map String.uppercase l)) ;
                     Unix.close fd
    end ;;
class cap_server :
  int ->
  int ->
  object
    val nb_pending : int
    val port_num : int
    val sock : Unix.file_descr
    method process : Unix.file_descr -> unit
```

```
    method receive : Unix.file_descr -> Cap_Protocol.t
    method send : Unix.file_descr -> Cap_Protocol.t -> unit
    method start : unit
  end
```

The processing consists of receiving a request, examining it, processing it and sending the result. The functor allows us to concentrate on this processing while constructing the server; the rest is generic. However, if we wanted a different mechanism, such as for example using acknowledgements, nothing would prevent us from redefining the inherited methods for communication.

Client

To construct clients using a given protocol, we define three general-purpose functions:

- `connect`: establishes a connection with a server; it takes the address (IP address and port number) and returns a file descriptor corresponding to a socket connected to the server.
- `emit_simple`: opens a connection, sends a message and closes the connection.
- `emit_answer`: same as `emit_simple`, but waits for the server's response before closing the connection.

```
# module Client = functor (P : PROTOCOL) →
    struct
      module Com = Com (P)

      let connect addr port =
        let sock = ThreadUnix.socket Unix.PF_INET Unix.SOCK_STREAM 0
        and in_addr = (Unix.gethostbyname addr).Unix.h_addr_list.(0)
        in ThreadUnix.connect sock (Unix.ADDR_INET(in_addr, port)) ;
           sock

      let emit_simple addr port mes =
        let sock = connect addr port
        in Com.send sock mes ;  Unix.close sock

      let emit_answer addr port mes =
        let sock = connect addr port
        in Com.send sock mes ;
           let res = Com.receive sock
           in Unix.close sock ; res
    end ;;
module Client :
  functor(P : PROTOCOL) ->
    sig
      module Com :
        sig
          val send : Unix.file_descr -> P.t -> unit
```

```
        val receive : Unix.file_descr -> P.t
      end
    val connect : string -> int -> Unix.file_descr
    val emit_simple : string -> int -> P.t -> unit
    val emit_answer : string -> int -> P.t -> P.t
  end
```

The last two functions are of a higher level than the first: the mechanism linking the client and the server does not appear. The caller of `emit_answer` does not even need to know that the computation it is requesting is carried out by a remote machine. As far as the caller is concerned, it invokes a function that is represented by an address and port, with an argument which is the message to be sent, and a value is returned to it. The distributed aspect can seem entirely hypothetical.

A client of the CAPITAL service is extremely easy to construct. Assume that the `boulmich` machine provides the service on port number 12345; then the function `list_uppercase` can be defined by means of a call to the service.

```
# let list_uppercase l =
    let module Cap_client = Client (Cap_Protocol)
    in match Cap_client.emit_answer "boulmich" 12345 (LStr l)
       with Str x → [x]
         | LStr x → x ;;
val list_uppercase : string list -> string list = <fun>
```

To Learn More

The first improvement to be made to our toolbox is some error handling, which has been totally absent so far. Recovery from exceptions which arise from a broken connection, and a mechanism for retrying, would be most welcome.

In the same vein, the client and the server would benefit from a timeout mechanism which would make it possible to limit the time to wait for a response.

Because we have constructed the generic server as a class, which moreover is parameterized by the type of data to be transmitted over the network, it is easy to extend it to augment or modify its behavior in order to implement any desired improvements.

Another approach is to enrich the communication protocols. One can for example add requests for acknowledgement to the protocol, or accompany each request by a checksum allowing verification that the network has not corrupted the data.

The Robots of Dawn

As we promised in the last application of the third part (page 550), we will now revisit the problem of robots in order to treat it in a distributed framework where the world is a server and where each robot is an independent process capable of being executed on a remote machine.

This application is a good summary of the possibilities of the Objective Caml language because we will utilize and combine the majority of its features. In addition to the distributed model which is imposed on us by the exercise, we will make use of concurrency to construct a server in which multiple connections will be handled independently while all sharing a single memory representation of the "world". All access to and modification of the state of affairs of the world will therefore have to be protected by critical sections.

In order to reuse as much as possible the code that we have already built for robots in one section, and the client-server architecture of another section, we will use functors and inheritance of classes at the same time.

This application is quite minimal, but we will see that its architecture lends itself particularly well to extensions in multiple directions.

World-Server

We take a representation of the world similar to that which we developed in Part III. The world is a grid of finite size, and each cell of the grid can be occupied by only one robot. A robot is identified by its name and by its position; the world is determined by its size and by the robots that live in it. This information is represented by the following types:

```
# type position = { x:int ; y:int } ;;

# type robot_info = { name : string ; mutable pos : position }
  type world_info = { length : int ; width  : int ;
                      mutable robots : robot_info list } ;;
```

The world will have to serve two sorts of clients:

- passive clients which simply observe the positions of various robots. They will allow us to build the clients in charge of displays. We will call them *spies*.
- active clients, able to ask the server to move robots and thus modify its state.

These two categories of clients and their behavior will determine the collection of messages exchanged by the server and clients.

When a client connects, it declares itself passive (`Spy`) or active (`Enter`). A spy receives as response to its connection the global state of the world. Then, it is kept informed of all changes. However, it cannot submit any requests. A robot which connects must supply its characteristics (its name and its initial position); the world then confirms its arrival. Then, it can request information: its own position (`GetPos`) or the list of robots that surround it (`Look`). It can also instruct the world to move it. The protocol of requests to the world from distributed robots is represented by the following type:

```
# type query =
    | Spy                   (* initial declaration requests *)
    | Enter of robot_info
```

```
    | Move of position       (* robot requests *)
    | GetPos
    | Look of int

    | World of world_info    (* messages delivered by the world *)
    | Pos of robot_info
    | Exit of robot_info ;;
```

From this protocol, using the functors from the "distributed toolbox" of the previous chapter, we immediately derive the generic server.

```
# module Pquery = Make_Protocol (struct type t = query end ) ;;
# module Squery = Server (Pquery) ;;
```

Now we need only specify the behavior of the server by implementing the method `process` to handle both the data that represent the world and the data for managing connections.

More precisely, the server contains a variable `world` (of type *world_info*) which is protected by the lock `sem` (of type *Mutex.t*). It also contains a variable `spies` which is a list of queues of messages to send to observers, with one queue per spy. To activate the processes in charge of sending these messages, the server also maintains a signal (of type *Condition.t*).

We provide an auxiliary function `dist` to calculate the distance between two positions:

```
# let dist p q = max (abs (p.x-q.x)) (abs (p.y-q.y)) ;;
val dist : position -> position -> int = <fun>
```

The function `critical` encapsulates the calculation of a value within a critical section:

```
# let critical m f a =
    Mutex.lock m ; let r = f a in Mutex.unlock m ; r ;;
val critical : Mutex.t -> ('a -> 'b) -> 'a -> 'b = <fun>
```

Here is the definition of the class `server` implementing the world-server. It is long, but we will follow it up with a step-by-step explanation.

```
# class  server l w n np =
    object (self)
      inherit [query] Squery.server n np
      val world = { length=l ; width=w ; robots=[] }
      val sem = Mutex.create ()
      val mutable spies = []
      val signal = Condition.create ()

      method lock = Mutex.lock sem
      method unlock = Mutex.unlock sem
```

```
method legal_pos p = p.x>=0 && p.x<l && p.y>=0 && p.y<w

method free_pos p =
  let is_not_here r = r.pos.x<>p.x || r.pos.y<>p.y
  in critical sem (List.for_all is_not_here) world.robots

method legal_move r p =
  let dist1 p = (dist r.pos p) <= 1
  in (critical sem dist1 p) && self#legal_pos p && self#free_pos p


method queue_message mes =
  List.iter (Queue.add mes) spies ;
  Condition.broadcast signal

method trace_loop s q =
  let foo = Mutex.create () in
  let f () =
    try
      spies <- q :: spies ;
      self#send s (World world) ;
      while true do
        while Queue.length q = 0 do Condition.wait signal foo done ;
        self#send s (Queue.take q)
      done
    with _ → spies <- List.filter ((!=) q) spies ;
             Unix.close s
  in ignore (Thread.create f ())

method remove_robot r =
  self#lock ;
  world.robots <- List.filter ((<>) r) world.robots ;
  self#queue_message (Exit {r with name=r.name}) ;
  self#unlock

method try_move_robot r p =
  if self#legal_move r p
  then begin
         self#lock ;
         r.pos <- p ;
         self#queue_message (Pos {r with name=r.name}) ;
         self#unlock
       end

method process_robot s r =
  let f () =
    try
      world.robots <- r :: world.robots ;
      self#send s (Pos r) ;
      self#queue_message (Pos r) ;
      while true do
```

```
              Thread.delay 0.5 ;
              match self#receive s with
                  Move p → self#try_move_robot r p
                | GetPos → self#send s (Pos r)
                | Look d →
                    self#lock ;
                    let dist p = max (abs (p.x-r.pos.x)) (abs (p.y-r.pos.y)) in
                    let l = List.filter (fun x → (dist x.pos)<=d) world.robots
                    in self#send s (World { world with robots = l }) ;
                       self#unlock
                | _ → ()
            done
          with  _ → self#unlock ;
                    self#remove_robot r ;
                    Unix.close s
        in ignore (Thread.create f ())

      method process s =
        match self#receive s with
            Spy → self#trace_loop s (Queue.create ())
          | Enter r →
            ( if not (self#legal_pos r.pos && self#free_pos r.pos) then
              let i = ref 0 and j = ref 0 in
              ( try
                  for x=0 to l do
                    for y=0 to w do
                      let p = { x=x ; y=y }
                      in if self#legal_pos p && self#free_pos p
                         then ( i:=x ; j:=y; failwith "process" )
                    done done ;
                  Unix.close s
                with Failure "process" → r.pos <- { x= !i ; y= !j }  )) ;
              self#process_robot s r
          | _ → Unix.close s

    end ;;
class server :
  int ->
  int ->
  int ->
  int ->
  object
    val nb_pending : int
    val port_num : int
    val sem : Mutex.t
    val signal : Condition.t
    val sock : Unix.file_descr
    val mutable spies : Pquery.t Queue.t list
    val world : world_info
    method free_pos : position -> bool
    method legal_move : robot_info -> position -> bool
    method legal_pos : position -> bool
```

```
    method lock : unit
    method process : Unix.file_descr -> unit
    method process_robot : Unix.file_descr -> robot_info -> unit
    method queue_message : Pquery.t -> unit
    method receive : Unix.file_descr -> Pquery.t
    method remove_robot : robot_info -> unit
    method send : Unix.file_descr -> Pquery.t -> unit
    method start : unit
    method trace_loop : Unix.file_descr -> Pquery.t Queue.t -> unit
    method try_move_robot : robot_info -> position -> unit
    method unlock : unit
  end
```

The method `process` starts out by distinguishing between the two types of client. Depending on whether the client is active or passive, it invokes a processing method called: `trace_loop` for an observer, `process_robot` for a robot. In the second case, it checks that the initial position proposed by the client is compatible with the state of the world; if not, it finds a valid initial position. The remainder of the code can be divided into four categories:

1. **General methods**: these are methods which we developed in Part III for general worlds. Mainly, it is a matter of verifying that a displacement is legal for a given robot.
2. **Management of observers**: each observer is associated with a socket through which it is sent data, with a queue containing all the messages which have not yet been sent to it, and with a process. The method `trace_loop` is an infinite loop that empties the queue of messages by sending them; it goes to sleep when the queue is empty. The queues are filled, all at the same time, by the method `queue_message`. Note that after appending a message, the activation signal is sent to all processes.
3. **Management of robots**: here again, each robot is associated with a dedicated process. The method `process_robot` is an infinite loop: it waits for a request, processes it, and responds if necessary. Then it resumes waiting for the next request. Note that it is these robot-management methods which issue calls to the method `queue_message` when the state of the world has been modified. If the connection with a robot is lost—that is, if an exception is raised while waiting for a request—the robot is considered to have terminated and its departure is signaled to the observers.
4. **Inherited methods**: these are the methods of the generic server obtained by application of the functor `Server` to the protocol of our application.

Observer-client

The functor `Client` gives us generic functions for connecting with a server according to the particular protocol that concerns us here.

```
# module Cquery = Client (Pquery) ;;
module Cquery :
```

```
  sig
    module Com :
      sig
        val send : Unix.file_descr -> Pquery.t -> unit
        val receive : Unix.file_descr -> Pquery.t
      end
    val connect : string -> int -> Unix.file_descr
    val emit_simple : string -> int -> Pquery.t -> unit
    val emit_answer : string -> int -> Pquery.t -> Pquery.t
  end
```

The behavior of a spy is simple: it connects to the server and displays the information that the server sends it. The spy includes three display functions which we provide below:

```
# let display_robot r =
    Printf.printf "The robot %s is located at (%d,%d)\n" r.name r.pos.x r.pos.y ;
    flush stdout ;;
val display_robot : robot_info -> unit = <fun>

# let display_exit r =   Printf.printf "The robot %s has terminated\n" r.name ;
    flush stdout ;;
val display_exit : robot_info -> unit = <fun>

# let display_world w =
    Printf.printf "The world is a grid of size %d by %d \n" w.length w.width ;
    List.iter display_robot w.robots ;
    flush stdout ;;
val display_world : world_info -> unit = <fun>
```

The primary function of the spy-client is:

```
# let trace_client name port =
    let sock = Cquery.connect name port
    in Cquery.Com.send sock Spy ;
       ( match Cquery.Com.receive sock with
            World w → display_world w
          | _ → failwith "the server did not follow the protocol" ) ;
       while true do
         match Cquery.Com.receive sock with
            Pos r → display_robot r
          | Exit r → display_exit r
          |_ → failwith "the server did not follow the protocol"
       done ;;
val trace_client : string -> int -> unit = <fun>
```

There are two ways of constructing a graphical display. The first is simple but not very efficient: since the server sends the complete set of information when a connection is established, one can simply open a new connection at regular intervals, display the world in its entirety, and close the connection. The other approach involves using the information sent by the server to maintain a copy of the state of the world. It is then

easy to display only the modifications to the state upon reception of messages. It is this second solution which we have implemented.

Robot-Client

As we defined them in the previous chapter (cf. page 550), the robots conform to the following signature.

```
# module type ROBOT =
    sig
      class robot : int → int →
        object
          val mutable i : int
          val mutable j : int
          method get_pos : int * int
          method next_pos : unit → int * int
          method set_pos : int * int → unit
        end
      end ;;
```

The part that we wish to save from the various classes is that which necessarily varies from one type of robot to another and which defines its behavior: the method `next_pos`.

In addition, we need a method for connecting the robot to the world (`start`) and a loop that alternately calculates a new position and communicates with the server to submit the chosen position.

We define a functor which, when given a class implementing a virtual robot (that is, conforming to the signature `ROBOT`), creates, by inheritance, a new class containing the proper methods to make an autonomous client out of the robot.

```
# module RobotClient (R : ROBOT) =
    struct
      class robot robname x y hostname port =
        object (self)
          inherit R.robot x y as super
          val mutable socket = Unix.stderr
          val mutable rob = { name=robname ; pos={x=x;y=y} }

          method private adjust_pos r =
            rob.pos <- r.pos ; i <- r.pos.x ; j <- r.pos.y

          method get_pos =
            Cquery.Com.send socket GetPos ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; super#get_pos
              | _ → failwith "the server did not follow the protocol"
```

```
          method set_pos =
            failwith "the method set_pos cannot be used"

          method start =
            socket <- Cquery.connect hostname port ;
            Cquery.Com.send socket (Enter rob) ;
            match Cquery.Com.receive socket with
                Pos r → self#adjust_pos r ; self#run
              | _ → failwith "the server did not follow the protocol"

          method run =
            while true do
              let (x,y) = self#next_pos ()
              in Cquery.Com.send socket (Move {x=x;y=y}) ;
                 ignore (self#get_pos)
            done
        end
    end ;;
module RobotClient :
  functor(R : ROBOT) ->
    sig
      class robot :
        string ->
        int ->
        int ->
        string ->
        int ->
        object
          val mutable i : int
          val mutable j : int
          val mutable rob : robot_info
          val mutable socket : Unix.file_descr
          method private adjust_pos : robot_info -> unit
          method get_pos : int * int
          method next_pos : unit -> int * int
          method run : unit
          method set_pos : int * int -> unit
          method start : unit
        end
    end
```

Notice that the method `get_pos` has been redefined as a query to the server: the instance variables `i` and `j` are not reliable, because they can be modified without the consent of the world. For the same reason, the use of `set_pos` has been made invalid: calling it will always raise an exception. This policy may seem severe, but it's a good bet that if this method were used by `next_pos` then a discrepancy would appear between the real position (as known by the server) and the supposed position (as known by the client).

We use the functor `RobotClient` to create various classes corresponding to the various robots.

```
# module Fix = RobotClient (struct class robot = fix_robot end) ;;
# module Crazy = RobotClient (struct class robot = crazy_robot end) ;;
# module Obstinate = RobotClient (struct class robot = obstinate_robot end) ;;
```

The following small program provides a way to launch the server and the various clients from the command line. The argument passed to the program specifies which one to launch.

```
# let port = 1200 in
    if Array.length Sys.argv >=2 then
      match Sys.argv.(1) with
        "1" → let s = new server 25 30 port 10 in s#start
      | "2" → trace_client "localhost" port
      | "3" → let o = new Fix.robot "fix" 10 10 "localhost" port in o#start
      | "4" → let o = new Crazy.robot "crazy" 10 10 "localhost" port in o#start
      | "5" → let o = new Obstinate.robot "obstinate" 10 10 "localhost" port
              in o#start
      | _ → () ;;
```

To Learn More

The world of robots stimulates the imagination. With the elements already given here, one can easily create an "intelligent robot" which is both a robot and a spy. This allows the various inhabitants of the world to cooperate. One can then extend the application to obtain a small action game like "chickens-foxes-snakes" in which the foxes chase the chickens, the snakes chase the foxes and the chickens eat the snakes.

HTTP Servlets

A *servlet* is a "module" that can be integrated into a server application to respond to client requests. Although a servlet need not use a specific protocol, we will use the HTTP protocol for communication (see figure 21.1). In practice, the term servlet refers to an HTTP servlet.

The classic method of constructing dynamic HTML pages on a server is to use CGI (Common Gateway Interface) commands. These take as argument a URL which can contain data coming from an HTML form. The execution then produces a new HTML page which is sent to the client. The following links describe the HTTP and CGI protocols.

Link: http://www.cis.ohio-state.edu/cgi-bin/rfc/rfc1945.html

Link: http://hoohoo.ncsa.uiuc.edu/docs/cgi/overview.html

It is a slightly heavyweight mechanism because it launches a new program for each request.

HTTP servlets are launched just once, and can can decode arguments in CGI format to execute a request. Servlets can take advantage of the Web browser's capabilities to construct a graphical interface for an application.

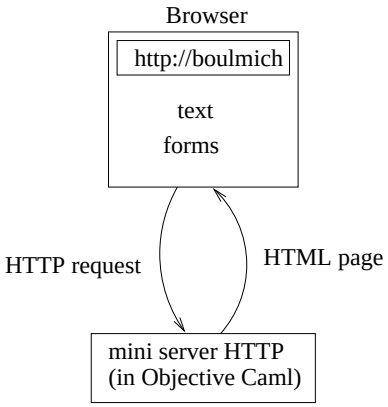


Figure 21.1: communication between a browser and an Objective Camlserver

In this section we will define a server for the HTTP protocol. We will not handle the entire specification of the protocol, but instead will limit ourselves to those functions necessary for the implementation of a server that mimics the behavior of a CGI application.

At an earlier time, we defined a generic server module `Gsd`. Now we will give the code to create an application of this generic server for processing part of the HTTP protocol.

HTTP and CGI Formats

We want to obtain a server that imitates the behavior of a CGI application. One of the first tasks is to decode the format of HTTP requests with CGI extensions for argument passing.

The clients of this server can be browsers such as Netscape or Internet Explorer.

Receiving Requests

Requests in the HTTP protocol have essentially three components: a method, a URL and some data. The data must follow a particular format.

In this section we will construct a collection of functions for reading, decomposing and decoding the components of a request. These functions can raise the exception:

```
# exception Http_error of string ;;
exception Http_error of string
```

Decoding The function `decode`, which uses the helper function `rep_xcode`, attempts to restore the characters which have been encoded by the HTTP client: spaces (which have been replaced by `+`), and certain reserved characters which have been replaced by their hexadecimal code.

```
# let rec rep_xcode s i =
    let xs = "0x00" in
      String.blit s (i+1) xs 2 2;
      String.set s i (char_of_int (int_of_string xs));
      String.blit s (i+3) s (i+1) ((String.length s)-(i+3));
      String.set s ((String.length s)-2) '\000';
      Printf.printf"rep_xcode1(%s)\n" s ;;
val rep_xcode : string -> int -> unit = <fun>

# exception End_of_decode of string ;;
exception End_of_decode of string

# let decode s =
    try
      for i=0 to pred(String.length s) do
        match s.[i] with
          '+' → s.[i] <- ' '
        | '%' → rep_xcode s i
        | '\000' → raise (End_of_decode (String.sub s 0 i))
        | _ → ()
      done;
      s
    with
     End_of_decode s → s ;;
val decode : string -> string = <fun>
```

String manipulation functions The module `String_plus` contains some functions for taking apart character strings:

- `prefix` and `suffix`, which extract the substrings to either side of an index;
- `split`, which returns the list of substrings determined by a separator character;
- `unsplit`, which concatenates a list of strings, inserting separator characters between them.

```
# module String_plus =
   struct
    let prefix s n =
     try String.sub s 0 n
     with Invalid_argument("String.sub") → s
```

```
    let suffix s i =
     try String.sub s i ((String.length s)-i)
     with Invalid_argument("String.sub") → ""

    let rec split c s =
     try
      let i = String.index s c in
      let s1, s2 = prefix s i, suffix s (i+1) in
       s1 :: (split c s2)
     with
      Not_found → [s]

    let unsplit c ss =
     let f s1 s2 = match s2 with "" → s1 | _ → s1^(Char.escaped c)^s2 in
      List.fold_right f ss ""
   end ;;
```

Decomposing data from a form Requests typically arise from an HTML page containing a form. The contents of the form are transmitted as a character string containing the names and values associated with the fields of the form. The function `get_field_pair` transforms such a string into an association list.

```
# let get_field_pair s =
    match String_plus.split '=' s with
      [n;v] → n,v
    | _ → raise (Http_error ("Bad field format : "^s)) ;;
val get_field_pair : string -> string * string = <fun>

# let get_form_content s =
    let ss = String_plus.split '&' s in
      List.map get_field_pair ss ;;
val get_form_content : string -> (string * string) list = <fun>
```

Reading and decomposing The function `get_query` extracts the method and the URL from a request and stores them in an array of character strings. One can thus use a standard CGI application which retrieves its arguments from the array of command-line arguments. The function `get_query` uses the auxiliary function `get`. We arbitrarily limit requests to a maximum size of 2555 characters.

```
# let get =
    let buff_size = 2555 in
      let buff = String.create buff_size in
        (fun ic → String.sub buff 0 (input ic buff 0 buff_size)) ;;
val get : in_channel -> string = <fun>

# let query_string http_frame =
    try
      let i0 = String.index http_frame ' ' in
```

```
      let q0 = String_plus.prefix http_frame i0 in
        match q0 with
          "GET"
          → begin
               let i1 = succ i0 in
               let i2 = String.index_from http_frame i1 ' ' in
               let q = String.sub http_frame i1 (i2-i1) in
               try
                 let i = String.index q '?' in
                 let q1 = String_plus.prefix q i in
                 let q = String_plus.suffix q (succ i) in
                   Array.of_list (q0 :: q1 :: (String_plus.split ' ' (decode q)))
               with
                 Not_found → [|q0;q|]
             end
        | _ → raise (Http_error ("Unsupported method: "^q0))
    with e → raise (Http_error ("Unknown request: "^http_frame)) ;;
val query_string : string -> string array = <fun>

# let get_query_string ic =
   let http_frame = get ic in
     query_string http_frame;;
val get_query_string : in_channel -> string array = <fun>
```

The Server

To obtain a CGI pseudo-server, able to process only the `GET` method, we write the class `http_servlet`, whose argument `fun_serv` is a function for processing HTTP requests such as might have been written for a CGI application.

```
# module Text_Server = Server (struct type t = string
                                      let to_string x = x
                                      let of_string x = x
                                end);;

# module P_Text_Server (P : PROTOCOL) =
  struct
    module Internal_Server = Server (P)

    class http_servlet  n np fun_serv =
      object(self)
        inherit [P.t] Internal_Server.server n np

        method receive_h fd =
          let ic = Unix.in_channel_of_descr fd in
            input_line ic

        method process fd =
          let oc = Unix.out_channel_of_descr fd in (
            try
              let request = self#receive_h fd in
```

```
              let args = query_string request in
                fun_serv oc args;
            with
              Http_error s → Printf.fprintf oc "HTTP error : %s <BR>" s
            | _ → Printf.fprintf oc "Unknown error <BR>" );
            flush oc;
            Unix.shutdown fd Unix.SHUTDOWN_ALL
      end
  end;;
```

As we do not expect the servlet to communicate using Objective Caml's special internal values, we choose the type *string* as the protocol type. The functions `of_string` and `to_string` do nothing.

```
# module Simple_http_server =
    P_Text_Server (struct type t = string
                          let of_string x = x
                          let to_string x = x
                   end);;
```

Finally, we write the primary function to launch the service and construct an instance of the class *http_servlet*.

```
# let cgi_like_server port_num fun_serv =
   let sv =   new Simple_http_server.http_servlet port_num 3  fun_serv
   in sv#start;;
val cgi_like_server : int -> (out_channel -> string array -> unit) -> unit =
  <fun>
```

Testing the Servlet

It is always useful during development to be able to test the parts that are already built. For this purpose, we build a small HTTP server which sends the file specified in the HTTP request as is. The function `simple_serv` sends the file whose name follows the GET request (the second element of the argument array). The function also displays all of the arguments passed in the request.

```
# let send_file oc f =
      let ic = open_in_bin f in
      try
        while true do
          output_byte oc (input_byte ic)
        done
      with End_of_file → close_in ic;;
val send_file : out_channel -> string -> unit = <fun>

# let simple_serv oc args =
    try
      Array.iter (fun x → print_string (x^" ")) args;
      print_newline();
      send_file oc args.(1)
    with _ → Printf.printf "error\n";;
val simple_serv : out_channel -> string array -> unit = <fun>
```

```
# let run n = cgi_like_server n simple_serv;;
val run : int -> unit = <fun>
```

The command *run* `4003` launches this servlet on port 4003. In addition, we launch a browser to issue a request to load the page `baro.html` on port 4003. The figure 21.2 shows the display of the contents of this page in the browser.

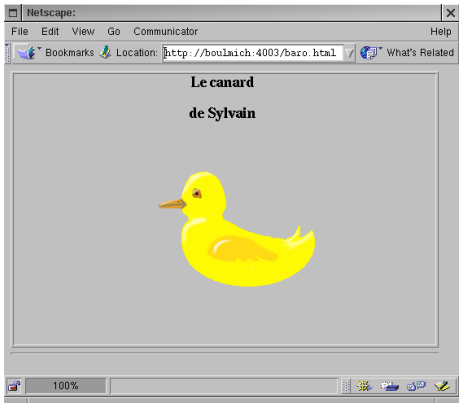


Figure 21.2: HTTP request to an Objective Caml servlet

The browser has sent the request `GET /baro.html` to load the page, and then the request `GET /canard.gif` to load the image.

HTML Servlet Interface

We will use a CGI-style server to build an HTML-based interface to the database of chapter 6 (see page 148).

The menu of the function `main` will now be displayed in a form on an HTML page, providing the same selections. The responses to requests are also HTML pages, generated dynamically by the servlet. The dynamic page construction makes use of the utilities defined below.

Application Protocol

Our application will use several elements from several protocols:

1. Requests are transmitted from a Web browser to our application server in the HTTP request format.
2. The data items within a request are encoded in the format used by CGI applications.
3. The response to the request is presented as an HTML page.

4. Finally, the nature of the request is specified in a format specific to the application.

We wish to respond to three kinds of request: queries for the list of mail addresses, queries for the list of email addresses, and queries for the state of received fees between two given dates. We give these query types respectively the names:
`mail_addr`, `email_addr` and `fees_state`. In the last case, we will also transmit two character strings containing the desired dates. These two dates correspond to the values of the fields `start` and `end` on an HTML form.

When a client first connects, the following page is sent. The names of the requests are encoded within it in the form of HTML anchors.

```
<HTML>
<TITLE> association </TITLE>
<BODY>
<HR>
<H1 ALIGN=CENTER>Association</H1>
<P>
<HR>
<UL>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/mail_addr">
mail addresses
</A>
<LI>List of
<A HREF="http://freres-gras.ufr-info-p6.jussieu.fr:12345/email_addr">
email addresses
</A>
<LI>State of received fees<BR>
<FORM
 method="GET"
 action="http://freres-gras.ufr-info-p6.jussieu.fr:12345/fees_state">
Start date : <INPUT type="text" name="start" value="">
End date : <INPUT type="text" name="end" value="">
<INPUT name="action" type="submit" value="Send">
</FORM>
</UL>
<HR>
</BODY>
</HTML>
```

We assume that this page is contained in the file `assoc.html`.

HTML Primitives

The HTML utility functions are grouped together into a single class called `print`. It has a field specifying the output channel. Thus, it can be used just as well in a CGI

application (where the output channel is the standard output) as in an application using the HTTP server defined in the previous section (where the output channel is a network socket).

The proposed methods essentially allow us to encapsulate text within HTML tags. This text is either passed directly as an argument to the method in the form of a character string, or produced by a function. For example, the principal method `page` takes as its first argument a string corresponding to the header of the page[1], and as its second argument a function that prints out the contents of the page. The method `page` produces the tags corresponding to the HTML protocol.

The names of the methods match the names of the corresponding HTML tags, with additional options added in some cases.

```
# class print (oc0:out_channel) =
    object(self)
      val oc = oc0
      method flush () = flush oc
      method str =
        Printf.fprintf oc "%s"
      method page header (body:unit → unit) =
        Printf.fprintf oc "<HTML><HEAD><TITLE>%s</TITLE></HEAD>\n<BODY>" header;
        body();
        Printf.fprintf oc "</BODY>\n</HTML>\n"
      method p () =
        Printf.fprintf oc "\n<P>\n"
      method br () =
        Printf.fprintf oc "<BR>\n"
      method hr () =
        Printf.fprintf oc "<HR>\n"
      method hr () =
        Printf.fprintf oc "\n<HR>\n"
      method h i s =
        Printf.fprintf oc "<H%d>%s</H%d>" i s i
      method h_center i s =
        Printf.fprintf oc "<H%d ALIGN=\"CENTER\">%s</H%d>" i s i
      method form url (form_content:unit → unit) =
        Printf.fprintf oc "<FORM method=\"post\" action=\"%s\">\n" url;
        form_content ();
        Printf.fprintf oc "</FORM>"
      method input_text =
        Printf.fprintf oc
          "<INPUT type=\"text\" name=\"%s\" size=\"%d\" value=\"%s\">\n"
      method input_hidden_text =
        Printf.fprintf oc "<INPUT type=\"hidden\" name=\"%s\" value=\"%s\">\n"
      method input_submit =
        Printf.fprintf oc "<INPUT name=\"%s\" type=\"submit\" value=\"%s\">"
      method input_radio =
        Printf.fprintf oc "<INPUT type=\"radio\" name=\"%s\" value=\"%s\">\n"
      method input_radio_checked =
```

1. This header is generally displayed in the title bar of the browser window.

```
      Printf.fprintf oc
        "<INPUT type=\"radio\" name=\"%s\" value=\"%s\" CHECKED>\n"
    method option =
      Printf.fprintf oc "<OPTION> %s\n"
    method option_selected opt =
      Printf.fprintf oc "<OPTION SELECTED> %s" opt
    method select name options selected =
      Printf.fprintf oc "<SELECT name=\"%s\">\n" name;
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
        options;
      Printf.fprintf oc "</SELECT>\n"
    method options selected =
      List.iter
        (fun s → if s=selected then self#option_selected s else self#option s)
  end ;;
```

We will assume that these utilities are provided by the module `Html_frame`.

Dynamic Pages for Managing the Association Database

For each of the three kinds of request, the application must construct a page in response. For this purpose we use the utility module `Html_frame` given above. This means that the pages are not really constructed, but that their various components are emitted sequentially on the output channel.
We provide an additional (virtual) page to be returned in response to a request that is invalid or not understood.

Error page The function `print_error` takes as arguments a function for emitting an HTML page (*i.e.*, an instance of the class `print`) and a character string containing the error message.

```
# let print_error (print:Html_frame.print) s =
    let print_body() =
     print#str s; print#br()
    in
     print#page "Error" print_body ;;
val print_error : Html_frame.print -> string -> unit = <fun>
```

All of our functions for emitting responses to requests will take as their first argument a function for emitting an HTML page.

List of mail addresses To obtain the page giving the response to a query for the list of mail addresses, we will format the list of character strings obtained by the function `mail_addresses`, which was defined as part of the database (see page 157). We will

assume that this function, and all others directly involving requests to the database, have been defined in a module named `Assoc`.

To emit this list, we use a function for outputting simple lines:

```
# let print_lines (print:Html_frame.print) ls =
    let print_line l =  print#str l; print#br() in
     List.iter print_line ls ;;
val print_lines : Html_frame.print -> string list -> unit = <fun>
```

The function for responding to a query for the list of mail addresses is:

```
# let print_mail_addresses print db =
     print#page "Mail addresses"
                (fun () → print_lines print (Assoc.mail_addresses db))
   ;;
val print_mail_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

In addition to the parameter for emitting a page, the function `print_mail_addresses` takes the database as its second parameter.

List of email addresses This function is built on the same principles as that giving the list of mail addresses, except that it calls the function `email_addresses` from the module `Assoc`:

```
# let print_email_addresses print db =
     print#page "Email addresses"
                (fun () → print_lines print (Assoc.email_addresses db)) ;;
val print_email_addresses : Html_frame.print -> Assoc.data_base -> unit =
  <fun>
```

State of received fees The same principle also governs the definition of this function: retrieving the data corresponding to the request (which here is a pair), then emitting the corresponding character strings.

```
# let print_fees_state print db d1 d2 =
    let ls, t = Assoc.fees_state db d1 d2 in
    let page_body() =
     print_lines print ls;
     print#str ("Total : "^(string_of_float t));
     print#br()
    in
     print#page "State of received fees" page_body ;;
val print_fees_state :
  Html_frame.print -> Assoc.data_base -> string -> string -> unit = <fun>
```

Analysis of Requests and Response

We define two functions for producing responses based on an HTTP request. The first (`print_get_answer`) responds to a request presumed to be formulated using the GET method of the HTTP protocol. The second alters the production of the answer according to the actual method that the request used.

These two functions take as their second argument an array of character strings containing the elements of the HTTP request as analyzed by the function `get_query_string` (see page 668). The first element of the array contains the method, the second the name of the database request.
In the case of a query for the state of received fees, the start and end dates for the request are contained in the two fields of the form associated with the query. The data from the form are contained in the third field of the array, which must be decomposed by the function `get_form_content` (see page 668).

```
# let print_get_answer print q db =
    match q.(1) with
    | "/mail_addr" → print_mail_addresses print db
    | "/email_addr" → print_email_addresses print db
    | "/fees_state"
      → let nvs = get_form_content q.(2) in
        let d1 = List.assoc "start" nvs
        and d2 = List.assoc "end" nvs in
          print_fees_state print db d1 d2
    | _ → print_error print ("Unknown request: "^q.(1)) ;;
val print_get_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>

# let print_answer print q db =
    try
      match q.(0) with
       "GET" → print_get_answer print q db
      | _ → print_error print ("Unsupported method: "^q.(0))
    with
      e
      → let s = Array.fold_right (^) q "" in
        print_error print ("Something wrong with request: "^s) ;;
val print_answer :
  Html_frame.print -> string array -> Assoc.data_base -> unit = <fun>
```

Main Entry Point and Application

The application is a standalone executable that takes the port number as a parameter. It reads in the database before launching the server. The main function is obtained from the function `print_answer` defined above and from the generic HTTP server function `cgi_like_server` defined in the previous section (see page 670). The latter function is located in the module `Servlet`.

```
# let get_port_num() =
    if (Array.length Sys.argv) < 2 then 12345
    else
      try int_of_string Sys.argv.(1)
      with _ → 12345 ;;
val get_port_num : unit -> int = <fun>

# let main() =
    let db = Assoc.read_base "assoc.dat" in
    let assoc_answer oc q = print_answer (new Html_frame.print oc) q db in
      Servlet.cgi_like_server (get_port_num()) assoc_answer ;;
val main : unit -> unit = <fun>
```

To obtain a complete application, we combine the definitions of the display functions into a file `httpassoc.ml`. The file ends with a call to the function `main`:

```
main() ;;
```

We can then produce an executable named `assocd` using the compilation command:

```
ocamlc -thread -custom -o assocd unix.cma threads.cma \
       gsd.cmo servlet.cmo html_frame.cmo string_plus.cmo assoc.cmo \
       httpassoc.ml -cclib -lunix -cclib -lthreads
```

All that's left is to launch the server, load the HTML page[2] contained in the file `assoc.html` given at the beginning of this section (page 672), and click.

The figure 21.3 shows an example of the application in use. The browser establishes an initial connection with the servlet, which sends it the menu page. Once the entry fields are filled in, the user sends a new request which contains the data entered. The server decodes the request and calls on the association database to retrieve the desired information. The result is translated into HTML and sent to the client, which then displays this new page.

To Learn More

This application has numerous possible enhancements. First of all, the HTTP protocol used here is overly simple compared to the new versions, which add a header supplying the type and length of the page being sent. Likewise, the method `POST`, which allows modification of the server, is not supported.[3]

To be able to describe the type of a page to be returned, the servlet would have to support the MIME convention, which is used for describing documents such as those attached to email messages.

2. ... taking care to update the URL according to your machine
3. Nothing prevents one from using `GET` for this, but that does not correspond to the standard.

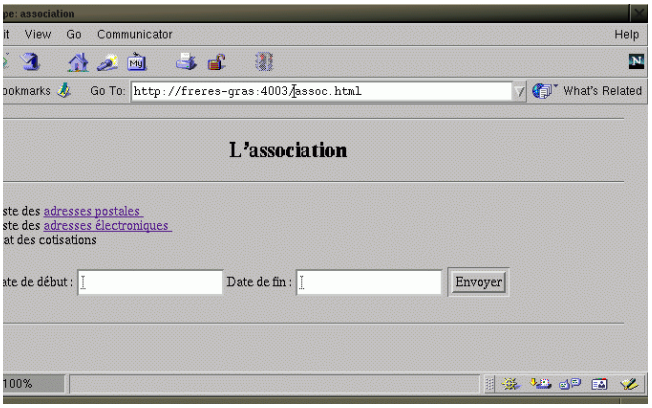


Figure 21.3: HTTP request to an Objective Caml servlet

The transmission of images, such as in figure 21.2, makes it possible to construct interfaces for 2-player games (see chapter 17), where one associates links with drawings of positions to be played. Since the server knows which moves are legal, only the valid positions are associated with links.

The MIME extension also allows defining new types of data. One can thus support a private protocol for Objective Caml values by defining a new MIME type. These values will be understandable only by an Objective Caml program using the same private protocol. In this way, a request by a client for a remote Objective Caml value can be issued via HTTP. One can even pass a serialized closure as an argument within an HTTP request. This, once reconstructed on the server side, can be executed to provide the desired result.

22

Developing applications with Objective Caml

Having reached this point, the reader should no longer doubt the richness of Objective Caml. This language rests on a functional and imperative core, and it integrates the two major application organization models: modules and objects. While presented as libraries, threads are an attractive part of the language. The system primitives, portable for the most part, complete the language with all the possibilities offered by distributed programming. These different programming paradigms are shaped within the general framework of static typing with inference. For all that, these elements do not, in themselves, settle the question of Objective Caml's relevance for developing applications, or more prosaically, "is it a good language?"

None of the following classic arguments can be used in its favor:

- (marketing development) "it's a good language because clients buy it";
- (historical development) "it's a good language because thousands of lines of code have already been written in it";
- (systems development) "it's a good language because the Unix or Windows systems are written in it";
- (beacon application development) "it's a good language because such-and-such application is written in it";
- (standardization development) "it's a good language because it has an ISO specification".

We'll review one last time the various features of the language, but this time from the angle of its relevance for answering a development team's needs. The criteria selected to make up the elements of our evaluation take into account the intrinsic qualities of the language, its development environment, the contributions of its community and the significant applications which have been achieved. Finally we'll compare Objective Caml with several similar functional languages as well as the object-oriented language Java in order to underscore the main differences.

Elements of the evaluation

The Objective Caml distribution supplies two online compilers, one generating bytecodes and the other producing instructions for the most modern processors. The toplevel uses the bytecode compiler. Beyond that, the distribution offers numerous libraries in the form of modules and some tools for calculating dependencies between modules, for profiling, and for debugging. Finally, thanks to its interface with the C language, it is possible to link Objective Caml programs to C programs. Languages which similarly offer an interface with C, such as the JNI (Java Native Interface) of Java, become accessible in this way.

The reference manual gives the language syntax, and describes the development tools and the library signatures. This set (language, tools, libraries, documentation) makes up a development environment.

Language

Specification and implementation

There are two ways to approach a new language. A first way is to read the language specification to have a global vision. A second is to plunge into the language's user manual, following the concepts illustrated by the examples. Objective Caml has neither one, which makes a self-taught approach to it relatively difficult. The absence of a formal specification (such as SML has) or a descriptive one (such as ADA's) is a handicap for understanding how a program works. Another consequence of the lack of a specification is the impossibility of getting a language standard issued by ISO, ANSI, or IEEE. This strongly limits the construction of new implementations tailored for other environments. Fortunately INRIA's implementation is of high quality and best of all, the sources of the distribution can be downloaded.

Syntax

The particulars of syntax are a non-negligible difficulty in approaching Objective Caml. They can be explained by the functional origin of the language, but also by historical, that is to say anecdotal, factors.

The syntax of application of a function to its arguments is defined by simple juxtaposition, as in `f 1 2`. The lack of parentheses bothers the neophyte as much as the C programmer or the confirmed Lisp programmer. Nevertheless, this difficulty only arises when reading the code of a programmer who's stingy with parentheses. Nothing stops the neophyte Objective Caml programmer from using more explicit parenthesization and writing `(f 1 2)`.

Beyond the functional core, Objective Caml adopts a syntax sometimes at odds with customary usage: access to array elements uses the notation `t.(i)` and not the usual brackets; method invocation is noted by a pound (# character) and not a dot, etc. These idiosyncrasies don't make it easier to get a grip on Objective Caml.

Finally, the syntax of Objective Caml and its ancestor Caml has undergone numerous modifications since their first implementation. Which hasn't pleaded in favor of the enduring nature of the developed applications.

To end on a more positive note, the pattern-matching syntax, inherited from the ML family, which Objective Caml incorporates, structures function definitions by case pleasingly and with simplicity.

Static typing

The fundamental character of the Objective Caml language resides in the language's static typing of expressions and declarations. This guarantees that no type error surfaces during program execution. Static type inference was conceived for the functional languages of the ML family, and Objective Caml has been able to maintain the greater part of this type discipline for the imperative and object-oriented extensions. However, in the object-oriented case, the programmer must sometimes give type inference a hand through explicit type constraints. Still, Objective Caml preserves static typing of expressions, and definitions, which provides an unsurpassed measure of execution safety: an Objective Caml program, will not return the "method not found" exception, which is not the case for dynamically typed object-oriented languages.

Objective Caml's parametric polymorphism of types allows the implementation of general algorithms. It is channeled into the object-oriented layer where parametric classes produce generic code, and not an expansion of code as generated by the templates of other languages. In itself, parametric polymorphism is an important component of code reusability.

The object-oriented extension adds a notion of inclusion polymorphism which is obtained by specifying subtyping relationships between objects. It reconciles code reusability, which constitutes the strength of the inheritance relationship between classes, with the security of static typing.

Libraries and tools

The libraries supplied with the distribution cover great needs. The programmer finds there, as a standard, the implementation of the most usual data structures with their basic functions. For example: stacks, queues, hash tables, AVL trees. More advanced tools can be found there as well, such as the treatment of data streams. These libraries are enriched in the course of successive language versions.

The `Unix` library allows lower-level programming as much for I/O as for process management. It is not identical for all platforms, which limits its use to some extent.

The exact arithmetic library and the regular expression library facilitate the development of specific applications.

Unfortunately the portable graphics library offers little functionality to support the construction of graphical user interfaces.

C libraries can easily be interfaced with the Objective Caml language. Here, the free availability of well-structured and duly commented sources definitely unlocks the potential for contact with the outside world and the various and sundry libraries to be found there.

Among the supplied tools, those for lexical and syntactic analysis, indispensable whenever dealing with complex textual data, are especially noteworthy. Based on the classic `lex` and `yacc`, they integrate perfectly with sum types and the functionality of Objective Caml, thus making them simpler to use than their predecessors.

Finally, no matter what soundness its features may bring, use of a language "in actual practice" never avoids the debugging phase of a program. The Objective Caml 2.04 distribution does not supply an IDE. Certainly, using the toplevel allows one to proceed rapidly to compilation and unit tests of functions (which is an undeniable advantage), but this usage will vary by platform and by programmer: cut-and-paste under X-Windows, calling the shell under `emacs`, requesting evaluation of a buffer under Windows. The next version (see appendix B) provides for the first time an environment for Unix containing a browser for interfaces and modules and a structured editor linked to the toplevel. Finally the distribution's debugger remains hard to use (in particular, because of the functional aspect and the language's parametric polymorphism) and limited to the Unix system.

Documentation

The documentation of the distribution consists of the reference manual in printable (PostScript) and online (HTML) format. This manual is not in any case an introduction to the language. On the contrary it is indispensable for finding out what's in the libraries or what parameters the commands take. Some pedagogical material can be found on Inria's Caml page, but it is mostly regarding the Caml-Light language. Thus the language is missing a complete tutorial manual, we hope that this book fills this gap.

Other development tools

We have restricted ourselves up to now to the Objective Caml distribution. Nevertheless the community of developers using this language is active, as demonstrated by the number of messages on the `caml-list@inria.fr` mailing list. Numerous tools, libraries, and extensions are used to facilitate development. In the following we detail the use of tools for editing, syntax extension, interfacing with other languages and parallel programming. We mention as well the numerous graphical interface libraries. Most of these contributions can be found on the "Caml Hump" site:

Link: http://caml.inria.fr/hump.html

Editing tools

There are several modes recognizing Objective Caml syntax for the `emacs` editor. These modes are used to automatically indent text in the course of entering it, making it more readable. This is an alternative to the interaction window under Windows. Since `emacs` runs under Windows, the Objective Caml toplevel can be launched within one of its windows.

Syntax extension

The lexical and syntactic analysis tools provided by the distribution are already quite complete, but they don't support extending the syntax of the language itself. The `camlp4` tool (see the link on page 313) is used in place and instead of Objective Caml's syntactic analyzer. The latter's compilers have only to proceed to typing and code generation. This tool allows the user to extend the syntax of the language, or to change to the original syntax. Moreover it offers pretty-printing facilities for the generated syntax trees. In this way it becomes easy to write a new toplevel for any Objective Caml syntax extension, or even another language implemented in Objective Caml.

Interoperability with other languages

Chapter 12 detailed how to interface the Objective Caml language with C. A multi-language application takes advantage of the features of each one, all while making different codes sharing a single memory space live in harmony. Nevertheless encapsulating C functions to make them callable from Objective Caml requires some tedious work. To simplify it, the `camlIDL` tool (see the link on page 350) supplies an interface generator and tools for importing COM (Windows) components into Objective Caml. The interfaces are generated from an `IDL` interface description file.

Graphical interfaces

The `Graphics` library allows the development of drawings and simple interactions, but it can't be considered a graphical interface worthy of the name. Chapter 13 has shown how this library could be extended to construct graphical components responding to some interactions. Using `Graphics` as a base allows us to preserve the portability of the interface between different platforms (X-Windows, Windows, MacOS), but limits its use to the low level of events and graphics contexts.

Several projects attempt to fill this gap, unfortunately none succeeds in being complete, portable, documented, and simple to use. Here is a list (extracted from the "Caml Hump") of the main projects:

- OlibRt: under this sweet name, it is a veritable toolbox, constructed under X-Windows, but not documented. Its distribution contains complete examples and in particular numerous games.
  Link: http://cristal.inria.fr/~ddr/

- camlTk is a complete and well documented interface to the Tk toolkit. Its weak point is its dependency on particular versions of Tcl/Tk, which makes it difficult to install. It was used to build the web browser `mmm` [Rou96] written in Objective Caml.
  Link: http://caml.inria.fr/~rouaix/camltk-readme.html
- The `Xlib` library has been rewritten in Objective Caml. `Efuns`, a mini-clone of `emacs`, was developed using it. `Xlib` is not really a toolbox, and is not portable to graphical systems other than X-Windows.
- mlGtk is an interface built on Gtk. It is in development and has no documentation. Its interest lies in being portable under Unix and Windows (because Gtk is) in a simpler fashion than Tk. Besides it uses Objective Caml's object-oriented layer—which doesn't happen without sometimes posing some problems.
- LabTk is an interface to Tcl/Tk, for Unix, using extensions to Objective Caml which will be integrated into the next version (see appendix B). It includes its own Tcl/Tk distribution which installs easily.

Despite the efforts of the community, there is a real lack of tools for constructing portable interfaces. It may be hoped that LabTk becomes portable to different systems.

Parallel programming and distribution

Threads and sockets already offer basic mechanisms for concurrent and distributed programming. Interfacing with the C language allows the use of classic parallel programming libraries. The only thing missing is an interface with CORBA for invoking methods of remote objects. On the other hand, there are numerous libraries and language extensions which use different models of parallelism.

Libraries

The two main parallel programming libraries, `MPI` (Message Passing Interface) and `PVM` (Parallel Virtual Machine), are interfaced with Objective Caml. Documentation, links, and sources for these libraries can be found on the site

Link: http://www.netlib.org

The "Caml Hump" contains the various HTTP addresses from which the versions interfaced with Objective Caml can be downloaded.

Extensions

Numerous parallel extensions of Caml-Light or Objective Caml have been developed:

- Caml-Flight ([FC95]) is a SPMD (Simple Program Multiple Data) extension of the Caml-Light language. A program executes a copy of itself on a fixed number of processes. Communications are explicit, there is only one communication operation `get` which can only be executed from within the synchronization operation `sync`.

Link:
http://www.univ-orleans.fr/SCIENCES/LIFO/Members/ghains/caml-flight.html

- BSML [BLH00] is an extension by BSP operations. The language preserves compositionality and allows precise predictions of performance if the number of processors is fixed.
  Link:
  http://www.univ-orleans.fr/SCIENCES/LIFO/Members/loulergu/bsml.html
- OCAMLP3 [DDLP98] is a parallel programming environment based on the skeleton model of the P3L language. The various predefined skeletons can overlap. Programs may be tested either in sequential mode or parallel mode, thus supporting reuse of Objective Caml's own tools.
  Link: http://www.di.unipi.it/~susanna/projects.html
- JoCAML [CL99] is based on the join-calculus model which supports high-level operations for concurrency, communication, and synchronization in the presence of distributed objects and mobile code, all while preserving automatic memory management.
  Link: http://pauillac.inria.fr/jocaml/
- Lucid Synchrone ([CP95]) is a language dedicated to the implemenation of reactive systems. It combines the functionality of Objective Caml and the features of *data-flow synchronous* languages.
  Link: http://www-spi.lip6.fr/~pouzet/lucid-synchrone/

Applications developed in Objective Caml

A certain number of applications have been developed in Objective Caml. We will only speak of "public" applications, that is, those which anyone can use either freely or by buying them.

Like other functional languages, Objective Caml is a good compiler implementation language. The bootstrap[1] of the `ocaml` compiler is a convincing example. As well, numerous language extensions have been contributed, as seen previously for parallel programming, but also for typing such as O'Labl (part of which is in the process of being integrated into Objective Caml, see appendix B) or for physical units. Links to these applications can be found on the "Caml Hump".

Objective Caml's second specialty concerns proof assistants. The major development in this area is the program Coq which accompanies the evolution of Caml almost since its origin. Historically, ML was conceived as the command language of the LCF (Logic for Computable Functions) system, before becoming independent of this application. It is thus natural to find it as the implementation language of an important theorem-proving program.

1. Bootstrapping is the compilation of a compiler by the compiler itself. Arrival at a fixed point, that is to say the compiler and the generated executable are identical, is a good test of compiler correctness.

A third application domain concerns parallelism (see page 684) and communication of which a good example is the `Ensemble` system.

Link: http://www.cs.cornell.edu/Info/Projects/Ensemble/

A list, not exhaustive, of significant applications developed in Objective Caml is maintained on Inria's Caml site:

Link: http://caml.inria.fr/users_programs-eng.html

Let us mention in particular `hevea` which is a LaTeX to HTML translator which we have used to create the HTML version of this book found on the accompanying CD-ROM.

Link: http://pauillac.inria.fr/~maranget/hevea/

While of importance, the applications we've just mentioned don't represent what, at the beginning of this chapter, we christened a "beacon application". Moreover, they don't explore a new specialized domain demonstrating the relevance of using Objective Caml. It is not clear that this example can be issued from academia. It is more likely that it will come from industry, whether in conjunction with language standardization (and so its formal specification), or for the needs of applications having to integrate different programming and program organization styles.

Similar functional languages

There are several languages similar to Objective Caml, whether through the functional aspect, or through typing. Objective Caml is descended from the ML family, and thus it has cousins of which the closest are across the Atlantic and across the channel in the lineage of SML (Standard ML). The Lisp family, and in particular the Scheme language, differs from ML mainly by its dynamic typing. Two lazy languages, Miranda and Haskell, take up or extend ML's typing in the framework of delayed evaluation. Two functional languages, Erlang and SCOL, developed by the Ericsson and Cryo-Networks corporations respectively, are directed towards communication.

ML family

The ML family comprises two main branches: Caml (Categorical Abstract Machine Language) and its derivatives Caml-Light and Objective Caml, SML (Standard ML) and its descendants SML/NJ and mosml. Caml, the ancestor, was developed between 1986 and 1990 by INRIA's FORMEL project in collaboration with University Paris 7 and the École Normale Supérieure. Its implementation was based on Le_Lisp's runtime. It integrated within the language the definition of grammars and pretty-printers, which allowed communication of values between the language described and Caml. Its type system was more restrictive for mutable values, insofar as it did not allow such values to be polymorphic. Its first descendant, Caml-Light, no longer used the CAM machine, but instead used Zinc for its implementation. The name was nevertheless retained to

show its ancestry. It contributed a more lightweight implementation while optimizing the allocation of closures and using an optimizing GC as a precursor to the actual GC. This streamlining allowed it to be used on the PC's of the time. The various Caml-Light versions evolved towards actual typing of imperative features and were enriched with numerous libraries. The following offshoot, Caml Special Light or CSL, introduced parameterized modules and a native-code compiler. Finally the baby is actually Objective Caml which mainly adds the object-oriented extension to CSL. Since there has never been a complete specification of the Caml languages, these various changes have been able to take place in complete freedom.

The SML approach has been the opposite. The formal specification [MTH97] was given before the first implementation. It is difficult to read, and a second book gives a commentary ([MT91]) on it. This method, specification then implementation, has allowed the development of several implementations, of which the best-known is SML/NJ (Standard ML of New Jersey) from Lucent (ex-AT&T). Since its origin, SML has integrated parameterized modules. Its initial type system was different from that of Caml for imperative features, introducing a level of weakness for type variables. The differences between the two languages are detailed in [CKL96]. These differences are being effaced with time. The two families have the same type system for the functional and imperative core, Objective Caml now has parameterized modules. SML has also undergone changes, bringing it closer to Objective Caml, such as for record types. If the two languages don't merge, this mainly derives from their separate development. It is to be noted that there is a commercial development environment for SML, MLWorks, from Harlequin:

Link: http://www.harlequin.com/products/

An SML implementation, `mosml`, based on Caml-Light's runtime, has also been implemented.

Scheme

The Scheme language (1975) is a dialect of the Lisp language (1960). It has been standardized (IEEE Std 1178-1990). It is a functional language with strict evaluation, equipped with imperative features, dynamically typed. Its syntax is regular and particular about the use of parentheses. The principal data structure is the dotted pair (equivalent to an ML pair) with which possibly heterogeneous lists are constructed. The main loop of a Scheme toplevel is written `(print (eval (read)))`. The `read` function reads standard input and constructs a Scheme expression. The `eval` function evaluates the constructed expression and the `print` function prints the result. Scheme has a very useful macro-expansion system which, in association with the `eval` function, permits the easy construction of language extensions. It not only supports interrupting computation (exceptions) but also resuming computation thanks to continuations. A continuation corresponds to a point of computation. The special form `call_cc` launches a computation with the possibility of resuming this one at the level of the current continuation, that is to say of returning to this computation. There are many Scheme implementations. It is even used as a macro language for the GIMP image manipula-

tion software. Scheme is an excellent experimental laboratory for the implementation of new sequential or parallel programming concepts (thanks to continuations).

Languages with delayed evaluation

In contrast with ML or Lisp, languages with delayed evaluation do not compute the parameters of function calls when they are passed, but when evaluation of the body of the function requires it. There is a "lazy" version of ML called Lazy ML but the main representatives of this language family are Miranda and Haskell.

Miranda

Miranda([Tur85]) is a pure functional language. That is to say, without side effects. A Miranda program is a sequence of equations defining functions and data structures.

Link:
http://www.engin.umd.umich.edu/CIS/course.des/cis400/miranda/miranda.html

For example the fib function is defined in this way:

```
fib a = 1, a=0
      = 1, a=1
      = fib(a-1) + fib(a-2), a>1
```

Equations are chosen either through guards (conditional expressions) as above, or by pattern-matching as in the example below:

```
fib 0 = 1
fib 1 = 1
fib a = fib(a-1)+ fib(a-2)
```

These two methods can be mixed.

Functions are higher order and can be partially evaluated. Evaluation is lazy, no subexpression is computed until the moment when its value becomes necessary. Thus, Miranda lists are naturally streams.

Miranda has a concise syntax for infinite structures (lists, sets): `[1..]` represents the list of all the natural numbers. The list of values of the Fibonacci function is written briefly: `fibs = [a | (a,b) <- (1,1),(b,a+b)..]`. Since values are only computed when used, the declaration of `fibs` costs nothing.

Miranda is strongly typed, using a Hindley-Milner type system. Its type discipline is essentially the same as ML's. It accepts the definition of data by the user.

Miranda is the archetype of pure lazy functional languages.

Haskell

The main Haskell language website contains reports of the definition of the language and its libraries, as well as its main implementations.

Link: http://www.haskell.org

Several books are dedicated to functional programming in Haskell, one of the most recent is [Tho99].

This is a language which incorporates almost all of the new concepts of functional languages. It is pure (without side effects), lazy (not strict), equipped with an *ad hoc* polymorphism (for overloading) as well as parametric polymorphism *à la* ML.

Ad hoc **polymorphism** This system is different from the polymorphism seen up to now. In ML a polymorphic function disregards its polymorphic arguments. The treatment is identical for all types. In Haskell it is the opposite. A polymorphic function may have a different behavior according to the type of its polymorphic arguments. This allows function overloading.

The basic idea is to define type classes which group together sets of overloaded functions. A class declaration defines a new class and the operations which it permits. A (class) instance declaration indicates that a certain type is an instance of some class. It includes the definition of the overloaded operations of this class for this type.

For example the `Num` class has the following declaration:

```
class Num a where
  (+)    :: a -> a -> a
  negate :: a -> a
```

Now an instance `Int` of the class `Num` can be declared in this way:

```
instance Num Int where
  x + y    =  addInt x y
  negate x = negateInt x
```

And the instance `Float`:

```
instance Num Float where
  x + y    =  addFloat x y
  negate x = negateFloat x
```

The application of `negate Num` will have a different behavior if the argument is an instance of `Int` or `Float`.

The other advantage of classes derives from inheritance between classes. The descendant class recovers the functions declared by its ancestor. Its instances can modify their behavior.

Other characteristics The other characteristics of the Haskell language are mainly the following:

- a purely functional I/O system using *monads*;
- arrays are built lazily;
- views permit different representations of a single data type.

In fact it contains just about all the high-strung features born of research in the functional language domain. This is its advantage and its disadvantage.

Communication languages

ERLANG

ERLANG is a dynamically typed functional language for concurrent programming. It was developed by the Ericsson corporation in the context of telecommunications applications. It is now open source. The main site for accessing the language is the following:

Link: http://www.erlang.org

It was conceived so that the creation of processes and their communication might be easy. Communications take place by message passing and they can be submitted with delays. It is easy to define protocols via ports. Each process possesses its own definition dictionary. Error management uses an exception mechanism and signals can propagate among processes. Numerous telephony applications have been developed in Erlang, yielding non-negligible savings of development time.

SCOL

The SCOL language is a communication language for constructing 3D worlds. It was developed by the Cryo Networks corporation:

Link: http://www.cryo-networks.com

Its core is close to that of Caml: it is functional, statically typed, parametrically polymorphic with type inference. It is "multimedia" thanks to its API's for sound, 2D, and 3D. The 3D engine is very efficient. SCOL's originality comes from communication between virtual machines by means of channels. A channel is an (environment, network link) pair. The link is a (TCP or UDP) socket.

SCOL's originality lies in having resolved simply the problem of securing downloaded code: only the text of programs circulates on the network. The receiving machine types

the passed program, then executes it, guaranteeing that the code produced does indeed come from an official compiler. To implement such a solution, without sacrificing speed of transmission and reception, the choice of a statically typed functional language was imposed by the conciseness of source code which it supports.

Object-oriented languages: comparison with Java

Although Objective Caml sprang from the functional world, it is necessary to compare its object-oriented extension to an important representative of the object-oriented languages. We pick the Java language which, while similar from the point of view of its implementation, differs strongly in its object model and its type system.

The Java language is an object-oriented language developed by the SUN corporation. The main site for access to the language is the following:

Link: http://java.sun.com

Main characteristics

The Java language is a language with classes. Inheritance is simple and allows redefinition or overloading of inherited methods. Typing is static. An inherited class is in a subtyping relationship with its ancestor class.

Java does not have parameterized classes. One gets two types of polymorphism: *ad hoc* by overloading, and of inclusion by redefinition.

It is multi-threading and supports the development of distributed application whether using sockets or by invoking methods of remote objects (Remote Method Invocation).

The principles of its implementation are close to those of Objective Caml. A Java program is compiled to a virtual machine (JVM). The loading of code is dynamic. The code produced is independent of machine architectures, being interpreted by a virtual machine. The basic datatypes are specified in such a way as to guarantee the same representation on all architectures. The runtime is equipped with a GC.

Java has important class libraries (around 600 with the JDK, which are supplemented by many independent developments). The main libraries concern graphical interfaces and I/O operations integrating communication between machines.

Differences with Objective Caml

The main differences between Java and Objective Caml come from their type system, from redefinition and from overloading of methods. Redefinition of an inherited method must use parameters of exactly the same type. Method overloading supports switching

the method to use according to the types of the method call parameters. In the following example class `B` inherits from class `A`. Class `B` redefines the first version of the `to_string` method, but overloads the second version. Moreover the `eq` method is overloaded since the type of the parameter (here `B`) is not equal to the type of the parameter of the inherited method (here `A`). In the end class `B` has two `eq` methods and two `to_string` methods.

```
class A {
  boolean eq (A o) {  return true;}
  String to_string (int n ) { }
}

class B extends A {
  boolean eq (B o) {  return true;}
  String to_string (int n ) { }
  String to_string (float x, float y)
}
```

Although binding is late, overload resolution, that is determination of the type of the method to use, is carried out on compilation.

The second important difference derives from the possibility of casting the type of an object, as would be done in C. In the following example, two objects `a` and `b` are defined, of class `A` and `B` respectively. Then three variables `c`, `d` and `e` are declared while imposing a type constraint on the affected values.

```
{
  A a = new A ();
  B b = new B ();
  A c = (A) b;
  B d = (B) c;
  B e = (B) a;
}
```

Since the type of `b` is a subtype of the type of `a`, the cast from `b` to `c` is accepted. In this case the type constraint may be omitted. On the other hand the two following constraints require a dynamic type test to be carried out to guarantee that the values in `c` and in `a` do in fact correspond to objects of class `B`. In this program this is true for `c`, but false for `a`. So this last case raises an exception. While this is useful, in particular for graphical interfaces, these type constraints can lead to exceptions being raised during exception due to erroneous use of types. In this Java is a language typed partly statically and partly dynamically. Moreover the absence of parameterized classes quite often obliges one to use this feature to write generic classes.

Future of Objective Caml development

It is difficult for a new language to exist if it is not accompanied by the important development of an application (like Unix for C) or considerable commercial and industrial support (like SUN for JAVA). The intrinsic qualities of the language are rarely enough. Objective Caml has numerous qualities and some defects which we have described in the course of this chapter. For its part, Objective Caml is sustained by INRIA where it was conceived and implemented in the bosom of the CRISTAL project. Born of academic research, Objective Caml is used there as an experimental laboratory for testing new programming paradigms, and an implementation language. It is widely taught in various university programs and preparatory classes. Several thousand students each year learn the concepts of the language and practice it. In this way the Objective Caml language has an important place in the academic world. The teaching of computer science, in France, but also in the United States, creates numerous programmers in this language on a practical as well as a theoretical level.

On the other hand, in industry the movement is less dynamic. To our knowledge, there is not a single commercial application, developed in Objective Caml, sold to the general public and advertising its use of Objective Caml. The only example coming close is that of the SCOL language from Cryo-Networks. There is however a slight agitation in this direction. The first appeals for funding for Objective Caml application startups are appearing. Without hoping for a rapid snowball effect, it is significant that a demand exists for this type of language. And without hoping for a very short-term return on investment either, it is important to take notice of it.

It is now for the language and its development environment to show their relevance. To accompany this phenomenon, it is no doubt necessary to provide certain guarantees as to the evolution of the language. In this capacity, Objective Caml is only just now emerging and must make the choice to venture further out of academia. But this "entry into the world" will only take place if certain rules are followed:

- guaranteeing the survival of developments by assuring upward compatibility in future versions of the language (the difficulty being stability of new elements (objects, etc.));
- specifying the language in conjunction with real developers with a view to future standardization (which will permit the development of several implementations to guarantee the existence of several solutions);
- conceiving a development environment containing a portable graphical interface, a CORBA bus, database interfaces, and especially a more congenial debugging environment.

Some of the points brought up, in particular standardization, can remain within the jurisdiction of academia. Others are only of advantage to industry. Thus everthing will depend on their degree of cooperation. There is a precedent demonstrating that a language can be "free" and still be commercially maintained, as this was the case for the `gnat` compiler of the ADA language and the ACT corporation.

Link: http://www.act-europe.fr

Conclusion

Although computer science has become an industrial activity, in many respects the success of a programming language is a subjective affair. If "the heart has its reasons of which reason knows nothing," then Objective Caml is a reasonable choice for a lover of heart.

It is based on solid theoretical foundations, all while providing a wide spectrum of programming paradigms. If one adds the simplicity of interaction with the language which the toplevel supports, that makes it a language perfectly adapted for teaching.

- Structured types and abstract types support approaching algorithmic problems and their complex data structures, all while abstracting away from problems of memory representation and allocation.
- The functional theoretical model underlying the language supplies a precise introduction to the notions of evaluation and typing which, as a "true programmer", one owes it to oneself to be taught.
- The various programming models can be approached independently of one another: from modular or object-oriented program structure to low-level systems programming, there are few areas where Objective Caml is not useful.
- Its suitability for symbolic programming makes it an excellent support for theoretical courses such as compiling or artifical intelligence.

For these qualities, Objective Caml is often used as the basis of the introductory computer science curriculum as well as for advanced programming courses which make explicit the link between the language's high level of abstraction and its execution. Many teachers have been and remain seduced by the pedagogical advantages of Objective Caml and, by way of consequence, many computer scientists have been schooled in it.

One of the first causes for satisfaction in Objective Caml development is how comfortable it is to use. The compiler loads rapidly and its static type inference lets nothing

escape. Other static analyses of the code give the programmer precious indices of anomalies if not errors: incomplete pattern-matching is signaled, partial application of a function in a sequence is detected, etc. To this first cause of satisfaction is added a second: the compiler very rapidly generates efficient code.

Compiler performance, conciseness of expression of functional programming, quality and diversity of libraries make Objective Caml a language perfectly adapted to the needs of "disposable software". But it would be diminishing it to restrict it to this single application domain. For these same reasons, Objective Caml is a precious tool for experimentation and application prototyping. Moreover, when the structuring mechanisms of modules and objects come to be added to the features already mentioned, the language opens the way to the conception and development of finished applications.

Finally, Objective Caml and its developer community form a milieu which reacts quickly to innovation in the area of programming. The free availability and the distribution of the source code of the language offer emerging concepts a terrain for experimentation.

Learning Objective Caml requires a certain effort from the programmer familiar with other languages. And this, as well as the object of study is in constant evolution. We hope that without masking the complexity of certain concepts, this book will facilitate this phase of learning and can thus accelerate the return on investment for the Objective Caml application developer.

Part V

Appendices

A

Cyclic Types

Objective Caml's type system would be much simpler if the language were purely functionnal. Alas, language extensions entail extensions to the type language, and to the inference mechanism, of which we saw the illustration with the weak type variables (see page 74), made unavoidable by imperative extensions.

Object typing introduces the notion of *cyclic type*, associated with the keyword **as** (see page 454), which can be used independently of any concept of object oriented programming. The present appendix describes this extension of the type language, available through an option of the compiler.

Cyclic types

In Objective Caml, it is possible to declare recursive data structures: such a structure may contain a value with precisely the same structure.

```
# type sum_ex1 = Ctor of sum_ex1 ;;
type sum_ex1 = | Ctor of sum_ex1

# type record_ex1 = { field : record_ex1 } ;;
type record_ex1 = { field: record_ex1 }
```

How to build values with such types is not obvious, since we need a value before building one! The recursive declaration of values allows to get out of this vicious circle.

```
# let rec sum_val = Ctor sum_val ;;
val sum_val : sum_ex1 = Ctor (Ctor (Ctor (Ctor (Ctor ...))))

# let rec val_record_1 = { field = val_record_2 }
```

```
  and     val_record_2 = { field = val_record_1 } ;;
val val_record_1 : record_ex1 = {field={field={field={field={field=...}}}}}
val val_record_2 : record_ex1 = {field={field={field={field={field=...}}}}}
```

Arbitrary planar trees can be represented by such a data structure.

```
# type 'a tree = Vertex of 'a * 'a tree list ;;
type 'a tree = | Vertex of 'a * 'a tree list
# let height_1 = Vertex (0,[]) ;;
val height_1 : int tree = Vertex (0, [])
# let height_2 = Vertex (0,[ Vertex (1,[]); Vertex (2,[]); Vertex (3,[]) ]) ;;
val height_2 : int tree =
  Vertex (0, [Vertex (1, []); Vertex (2, []); Vertex (3, [])])
# let height_3 = Vertex (0,[ height_2; height_1 ]) ;;
val height_3 : int tree =
  Vertex
   (0,
    [Vertex (0, [Vertex (...); Vertex (...); Vertex (...)]); Vertex (0, [])])

(* same with a record *)
# type 'a tree_rec = { label:'a ; sons:'a tree_rec list } ;;
type 'a tree_rec = { label: 'a; sons: 'a tree_rec list }
# let hgt_rec_1 = { label=0; sons=[] } ;;
val hgt_rec_1 : int tree_rec = {label=0; sons=[]}
# let hgt_rec_2 = { label=0; sons=[hgt_rec_1] } ;;
val hgt_rec_2 : int tree_rec = {label=0; sons=[{label=0; sons=[]}]}
```

We might think that an enumerated type with only one constructor is not useful, but by default, Objective Caml does not accept recursive type abbreviations.

```
# type 'a tree = 'a * 'a tree list ;;
Characters 7-34:
The type abbreviation tree is cyclic
```

We can define values with such a structure, but they do not have the same type.

```
# let tree_1 = (0,[]) ;;
val tree_1 : int * 'a list = 0, []
# let tree_2 = (0,[ (1,[]); (2,[]); (3,[]) ]) ;;
val tree_2 : int * (int * 'a list) list = 0, [1, []; 2, []; 3, []]
# let tree_3 = (0,[ tree_2; tree_1 ]) ;;
val tree_3 : int * (int * (int * 'a list) list) list =
  0, [0, [...; ...; ...]; 0, []]
```

In the same way, Objective Caml is not able to infer a type for a function whose argument is a value of this form.

```
# let max_list = List.fold_left max 0 ;;
val max_list : int list -> int = <fun>
```

```
# let rec height = function
      Vertex (_,[]) → 1
    | Vertex (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : 'a tree -> int = <fun>


# let rec height2 = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height2 sons)) ;;
Characters 95-99:
This expression has type 'a list but is here used with type
  ('b * 'a list) list
```

The error message tells us that the function `height2` could be typed, if we had type equality between *'a* and *'b * 'a list*, and precisely this equality was denied to us in the declaration of the type abbreviation *tree*.

However, object typing allows to build values, whose type is *cyclic*. Let us consider the following function, and try to guess its type.

```
# let f x = x#copy = x ;;
```

The type of `x` is a class with method `copy`. The type of this method should be the same as that of `x`, since equality is tested between them. So, if *foo* is the type of `x`, it has the form: *< copy : foo ; .. >*. From what has been said above, the type of this function is cyclic, and it should be rejected; but it is not:

```
# let f x = x#copy = x ;;
val f : (< copy : 'a; .. > as 'a) -> bool = <fun>
```

Objective Caml does accept this function, and notes the type cyclicity using **as**, which identifies *'a* with a type containing *'a*.

In fact, the problems are the same, but by default, Objective Caml will not accept such types unless objects are concerned. The function `height` is typable if it gives a cyclicity on the type of an object.

```
# let rec height a = match a#sons with
      [] → 1
    | l → 1 + (max_list (List.map height l)) ;;
val height : (< sons : 'a list; .. > as 'a) -> int = <fun>
```

Option `-rectypes`

With a compiler option, we can avoid this restriction to objects in cyclic types.

```
$ ocamlc -rectypes  ...
$ ocamlopt -rectypes  ...
$ ocaml -rectypes
```

If we take up the above examples in a toplevel started with this option, here is what we get.

```
# type 'a tree = 'a * 'a tree list ;;
type 'a tree = 'a * 'a tree list

# let rec height = function
      (_,[]) → 1
    | (_,sons) → 1 + (max_list (List.map height sons)) ;;
val height : ('b * 'a list as 'a) -> int = <fun>
```

The values `tree_1`, `tree_2` and `tree_3` previously defined don't have the same type, but they all have a type compatible with that of `height`.

```
# height tree_1 ;;
- : int = 1
# height tree_2 ;;
- : int = 2
# height tree_3 ;;
- : int = 3
```

The keyword **as** belongs to the type language, and as such, it can be used in a type declaration.

Syntax : **type** *nom* **=** *typedef* **as** *'var* **;;**

We can use this syntax to define type *tree*.

```
# type 'a tree = ( 'a * 'vertex list ) as 'vertex ;;
type 'a tree = 'a * 'a tree list
```

Warning If this mode may be useful in some cases, it tends to accept the typing of too many values, giving them types that are not easy to read.

Without the option `-rectypes`, the function below would have been rejected by the typing system.

```
# let inclus l1 l2 =
    let rec mem x = function
        [] → false
      | a::l → (l=x) || (mem x a) (* an error on purpose: a and l inverted *)
    in List.for_all (fun x → mem x l2) l1  ;;
val inclus : ('a list as 'a) list list -> ('b list as 'b) -> bool = <fun>
```

Although a quick examination of the type allows to conclude to an error, we no longer have an error message to help us locating this error.

B

Objective Caml 3.04

Independently of the development of Objective Caml, several extensions of the language appeared. One of these, named Olabl, was integrated with Objective Caml, starting with version 3.00.

This appendix describes briefly the new features offered in the current version of Objective Caml at the time of this writing, that is. Objective Caml 3.04. This version can be found on the CD-ROM accompanying this book. The new features include:

- *labels*;
- *optional arguments*;
- *polymorphic constructors*;
- the `ocamlbrowser` IDE;
- the `LablTk` library.

The reader is referred to the Objective Caml reference manual for a more detailed description of these features.

Language Extensions

Objective Caml 3.04 brings three language extensions to Objective Caml: labels, optional arguments, and polymorphic constructors. These extensions preserve backward compatibility with the original language: a program written for version 2.04 keeps the same semantics in version 3.04.

Labels

A label is an annotation for the arguments of a function in its declaration and its application. It is presented as a separate identifier of the function parameter (formal or actual), enclosed between an initial symbol `'~'` and a final symbol `':'`.

Labels can appear in the declarations of functions:

Syntax : **let** *f* ~*label*:*p* **=** *exp*

in the anonymous declarations with the keyword **fun** :

Syntax : **fun** ~*label*:*p* **->** *exp*

and in the actual parameter of a function:

Syntax : **(** *f* ~*label*:*exp* **)**

Labels in types The labels given to arguments of a functional expression appear in its type and annotate the types of the arguments to which they refer. (The `'~'` symbol in front of the label is omitted in types.)

```
# let add ~op1:x ~op2:y = x + y ;;
val add : op1:int -> op2:int -> int = <fun>

# let mk_triplet ~arg1:x ~arg2:y ~arg3:z = (x,y,z) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

If one wishes to give the same identifier to the label and the variable, as in `~x:x`, it is unnecessary to repeat the identifier; the shorter syntax `~x` can be used instead.

Syntax : **fun** ~*p* –> *exp*

```
# let mk_triplet ~arg1 ~arg2 ~arg3 = (arg1,arg2,arg3) ;;
val mk_triplet : arg1:'a -> arg2:'b -> arg3:'c -> 'a * 'b * 'c = <fun>
```

It is not possible to define labels in a declaration of a function by pattern matching; consequently the keyword **function** cannot be used for a function with a label.

```
# let f = function ~arg:x → x ;;
Toplevel input:
#
  let f = function ~arg:x -> x ;;
                   ^^^^^
Syntax error
# let f = fun ~arg:x → x ;;
val f : arg:'a -> 'a = <fun>
```

Labels in function applications When a function is defined with labeled parameters, applications of this function require that matching labels are provided on the function arguments.

```
# mk_triplet ~arg1:'1' ~arg2:2 ~arg3:3.0 ;;
- : char * int * float = '1', 2, 3
# mk_triplet '1' 2 3.0 ;;
- : char * int * float = '1', 2, 3
```

A consequence of this requirement is that the order of arguments having a label does not matter, since one can identify them by their label. Thus, labeled arguments to a function can be "commuted", that is, passed in an order different from the function definition.

```
# mk_triplet ~arg2:2  ~arg1:'1'  ~arg3:3.0 ;;
- : char * int * float = '1', 2, 3
```

This feature is particularly useful for making a partial application on an argument that is not the first in the declaration.

```
# let triplet_0_0 = mk_triplet ~arg2:0 ~arg3:0 ;;
val triplet_0_0 : arg1:'a -> 'a * int * int = <fun>
# triplet_0_0 ~arg1:2 ;;
- : int * int * int = 2, 0, 0
```

Arguments that have no label, or that have the same label as another argument, do not commute. In such a case, the application uses the first argument that has the given label.

```
# let test ~arg1:_ ~arg2:_  _  ~arg2:_  _ = () ;;
val test : arg1:'a -> arg2:'b -> 'c -> arg2:'d -> 'e -> unit = <fun>

# test ~arg2:() ;;  (* the first arg2: in the declaration *)
- : arg1:'a -> 'b -> arg2:'c -> 'd -> unit = <fun>

# test () ;;       (* the first unlabeled argument in the declaration *)
- : arg1:'a -> arg2:'b -> arg2:'c -> 'd -> unit = <fun>
```

Legibility of code Besides allowing re-ordering of function arguments, labels are also very useful to make the function interface more explicit. Consider for instance the `String.sub` standard library function.

```
# String.sub ;;
- : string -> int -> int -> string = <fun>
```

In the type of this function, nothing indicates that the first integer argument is a character position, while the second is the length of the string to be extracted. Objective Caml 3.04 provides a "labelized" version of this function, where the purpose of the different function arguments have been made explicit using labels.

```
# StringLabels.sub ;;
- : string -> pos:int -> len:int -> string = <fun>
```
Clearly, the function `StringLabels.sub` takes as arguments a string, the position of the first character, and the length of the string to be extracted.

Objective Caml 3.04 provides "labelized" versions of many standard library functions in the modules `ArrayLabels`, `ListLabels`, `StringLabels`, `UnixLabels`, and `MoreLabels`. Table B.1 gives the labeling conventions that were used.

label	significance
pos:	a position in a string or array
len:	a length
buf:	a string used as buffer
src:	the source of an operation
dst:	the destination of an operation
init:	the initial value for an iterator
cmp:	a comparison function
mode:	an operation mode or a flag list

Figure B.1: Conventions for labels

Optional arguments

Objective Caml 3.04 allows the definition of functions with labeled *optional* arguments. Such arguments are defined with a default value (the value given to the parameter if the application does not give any other explicitly).

Syntax : **fun ?***name*: **(** *p* **=** *exp*$_1$**)** $\rightarrow$ *exp*$_2$

As in the case of regular labels, the argument label can be omitted if it is identical to the argument identifier:

Syntax : **fun ?(** *name* **=** *exp*$_1$**)** $\rightarrow$ *exp*$_2$

Optional arguments appear in the function type prefixed with the **?** symbol.

```
# let sp_incr ?inc:(x=1) y =  y := !y + x ;;
val sp_incr : ?inc:int -> int ref -> unit = <fun>
```
The function `sp_incr` behaves like the function `incr` from the Pervasives module.
```
# let v = ref 4  in  sp_incr v ; v ;;
- : int ref = {contents = 5}
```
However, one can specify a different increment from the default.
```
# let v = ref 4  in  sp_incr ~inc:3 v ; v ;;
- : int ref = {contents = 7}
```

A function is applied by giving the default value to all the optional parameters until the actual parameter is passed by the application. If the argument of the call is given without a label, it is considered as being the first non-optional argument of the function.

```
# let f ?(x1=0) ?(x2=0) x3 x4 = 1000*x1+100*x2+10*x3+x4 ;;
val f : ?x1:int -> ?x2:int -> int -> int -> int = <fun>
# f 3 ;;
- : int -> int = <fun>
# f 3 4 ;;
- : int = 34
# f ~x1:1 3 4 ;;
- : int = 1034
# f ~x2:2 3 4 ;;
- : int = 234
```

An optional argument can be given without a default value, in this case it is considered in the body of the function as being of the type *'a option*; `None` is its default value.

Syntax : **fun ?***name*:*p* -> *exp*

```
# let print_integer ?file:opt_f n =
    match opt_f with
        None → print_int n
      | Some f → let fic = open_out f in
                 output_string fic (string_of_int n) ;
                 output_string fic "\n" ;
                 close_out fic ;;
val print_integer : ?file:string -> int -> unit = <fun>
```

By default, the function `print_integer` displays its argument on standard output. If it receives a file name with the label `file`, it outputs its integer argument to that file instead.

Note

If the last parameter of a function is optional, it will have to be applied explicitly.

```
# let test ?x ?y  n  ?a ?b  =  n ;;
val test : ?x:'a -> ?y:'b -> 'c -> ?a:'d -> ?b:'e -> 'c = <fun>
# test 1 ;;
- : ?a:'_a -> ?b:'_b -> int = <fun>
# test 1 ~b:'x' ;;
- : ?a:'_a -> int = <fun>
# test 1 ~a:() ~b:'x' ;;
- : int = 1
```

Labels and objects

Labels can be used for the parameters of a method or an object's constructor.

```
# class point ?(x=0) ?(y=0) (col : Graphics.color) =
    object
      val pos = (x,y)
      val color = col
      method print ?dest:(file=stdout) () =
          output_string file "point (" ;
          output_string file (string_of_int (fst pos)) ;
          output_string file "," ;
          output_string file (string_of_int (snd pos)) ;
          output_string file ")\n"
    end ;;
class point :
  ?x:int ->
  ?y:int ->
  Graphics.color ->
  object
    method print : ?dest:out_channel -> unit -> unit
    val color : Graphics.color
    val pos : int * int
  end

# let obj1 = new point ~x:1 ~y:2 Graphics.white
  in obj1#print () ;;
point (1,2)
- : unit = ()
# let obj2 = new point Graphics.black
  in obj2#print () ;;
point (0,0)
- : unit = ()
```

Labels and optional arguments provide an alternative to method and constructor overloading often found in object-oriented languages, but missing from Objective Caml.

This emulation of overloading has some limitations. In particular, it is necessary that at least one of the arguments is not optional. A dummy argument of type *unit* can always be used.

```
# class number ?integer ?real () =
    object
      val mutable value = 0.0
      method print = print_float value
      initializer
        match (integer,real) with
            (None,None) | (Some _,Some _) → failwith "incorrect number"
          | (None,Some f) → value <- f
```

```
          | (Some n,None) → value <- float_of_int n
    end ;;
class number :
  ?integer:int ->
  ?real:float ->
  unit -> object method print : unit val mutable value : float end

# let n1 = new number ~integer:1 () ;;
val n1 : number = <obj>
# let n2 = new number ~real:1.0 () ;;
val n2 : number = <obj>
```

Polymorphic variants

The variant types of Objective Caml have two principal limitations. First, it is not possible to extend a variant type with a new constructor. Also, a constructor can belong to only one type. Objective Caml 3.04 features an alternate kind of variant types, called *polymorphic variants* that do not have these two constraints.

Constructors for polymorphic variants are prefixed with a **‘** (backquote) character, to distinguish them from regular constructors. Apart from this, the syntactic constraints on polymorphic constructors are the same as for other constructors. In particular, the identifier used to build the constructor must begin with a capital letter.

Syntax : **‘***Name*

ou

Syntax : **‘***Name type*

A group of polymorphic variant constructors forms a type, but this type does not need to be declared before using the constructors.

```
# let x = ‘Integer 3 ;;
val x : [> ‘Integer of int] = ‘Integer 3
```

The type of `x` with the symbol **[>** indicates that the type contains at least the constructor ‘`Integer` *`int`*.

```
# let int_of = function
      ‘Integer n → n
    | ‘Real r → int_of_float r ;;
val int_of : [< ‘Integer of int | ‘Real of float] -> int = <fun>
```

Conversely, the symbol **[<** indicates that the argument of `int_of` belongs to the type that contains at most the constructors ‘`Integer` *`int`* and ‘`Real` *`float`*.

It is also possible to define a polymorphic variant type by enumerating its constructors:

Syntax : **type** *t* **= [** ‘*Name*$_1$ **|** ‘*Name*$_2$ **|** ... **|** ‘*Name*$_n$ **]**

or for parameterized types:

Syntax : **type** *('a,'b,...) t* **= [** ‘*Name*$_1$ **|** ‘*Name*$_2$ **|** ... **|** ‘*Name*$_n$ **]**

```
# type value = [ ‘Integer of int | ‘Real of float ] ;;
type value = [ ‘Integer of int | ‘Real of float]
```

Constructors of polymorphic variants can take arguments of different types.

```
# let v1 = ‘Number 2
  and v2 = ‘Number 2.0 ;;
val v1 : [> ‘Number of int] = ‘Number 2
val v2 : [> ‘Number of float] = ‘Number 2
```

However, v1 and v2 have different types.

```
# v1=v2 ;;
Toplevel input:
#
  v1=v2 ;;
     ^^
This expression has type [> ‘Number of float] but is here used with type
  [> ‘Number of int]
```

More generally, the constraints on the type of arguments for polymorphic variant constructors are accumulated in their type by the annotation **&**.

```
# let test_nul_integer = function ‘Number n → n=0
  and test_nul_real = function ‘Number r → r=0.0 ;;
val test_nul_integer : [< ‘Number of int] -> bool = <fun>
val test_nul_real : [< ‘Number of float] -> bool = <fun>
# let test_nul x = (test_nul_integer x) || (test_nul_real x) ;;
val test_nul : [< ‘Number of float & int] -> bool = <fun>
```

The type of test_nul indicates that the only values accepted by this function are those with the constructor ‘Number and an argument which is at the same time of type *int* and of *float*. That is, the only acceptable values are of type *'a*!

```
# let f () = test_nul (failwith "returns a value of type 'a") ;;
val f : unit -> bool = <fun>
```

The types of the polymorphic variant constructor are themselves likely to be polymorphic.

```
# let id = function ‘Ctor → ‘Ctor ;;
val id : [< ‘Ctor] -> [> ‘Ctor] = <fun>
```

The type of the value returned from id is “the group of constructors that contains at least ‘Ctor” therefore it is a polymorphic type which can instantiate to a more precise type. In the same way, the argument of id is “the group of constructors that contains

no more than ‘Ctor” which is also likely to be specified. Consequently, they follow the general polymorphic type mechanism of Objective Caml knowing that they are likely to be weakened.

```
# let v = id ‘Ctor ;;
val v : _[> ‘Ctor] = ‘Ctor
```

v, the result of the application is not polymorphic (as denoted by the character _ in the name of the type variable).

```
# id v ;;
- : _[> ‘Ctor] = ‘Ctor
```

v is monomorphic and its type is a sub-type of “contains at least the constructor ‘Ctor”. Applying it with id will force its type to be a sub-type of“contains no more than the constructor ‘Ctor”. Logically, it must now have the type “contains exactly ‘Ctor”. Let us check.

```
# v ;;
- : [ ‘Ctor] = ‘Ctor
```

As with object types, the types of polymorphic variant constructors can be open.

```
# let is_integer = function
      ‘Integer (n : int) → true
    | _ → false ;;
val is_integer : [> ‘Integer of int] -> bool = <fun>
# is_integer (‘Integer 3) ;;
- : bool = true
# is_integer ‘Other ;;
- : bool = false
```

All the constructors are accepted, but the constructor ‘Integer must have an integer argument.

```
# is_integer (‘Integer 3.0) ;;
Toplevel input:
#
  is_integer (‘Integer 3.0) ;;
              ^^^^^^^^^^^^
This expression has type [> ‘Integer of float] but is here used with type
  [> ‘Integer of int]
```

As with object types, the type of a constructor can be cyclic.

```
# let rec long = function ‘Rec x → 1 + (long x) ;;
val long : ([< ‘Rec of ’a] as ’a) -> int = <fun>
```

Finally, let us note that the type can be at the same time a sub-group and one of a group of constructors. Starting with a a simple example:

```
# let ex1 = function ‘C1 → ‘C2 ;;
val ex1 : [< ‘C1] -> [> ‘C2] = <fun>
```

Now we identify the input and output types of the example by a second pattern.

```
# let ex2 = function ‘C1 → ‘C2 | x → x ;;
val ex2 : ([> ‘C2 | ‘C1] as ’a) -> ’a = <fun>
```

We thus obtain the open type which contains at least ‘C2 since the return type contains at least ‘C2.

```
# ex2 ( ‘C1 : [> ‘C1 ] ) ;;  (* is a subtype of [<‘C2|‘C1| .. >‘C2] *)
```

```
- : _[> `C2 | `C1] = `C2
# ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
Toplevel input:
# ex2 ( `C1 : [ `C1 ] ) ;;   (* is not a subtype of [<`C2|`C1| .. >`C2] *)
        ^^^
This expression has type [ `C1] but is here used with type [> `C2 | `C1]
```

`LablTk` *Library*

The interface to Tcl/Tk was integrated in the distribution of Objective Caml 3.04, and is available for Unix and Windows. The installation provides one new command: `labltk`, which launches a toplevel interactive loop integrating the `LablTk` library.

The `LablTk` library defines a large number of modules, and heavily uses the language extensions of Objective Caml 3.04. A detailed presentation of this module falls outside the scope of this appendix, and we invite the interested reader to refer to the documentation of Objective Caml 3.04.

There is also an interface with Gtk, written in class-based style, but it is not yet part of the Objective Caml distribution. It should be compatible with Unix and Windows.

OCamlBrowser

OcamlBrowser is a code browser for Objective Caml, providing a `LablTk`-based graphical user interface. It integrates a "navigator" allowing to browse various modules, to look at their contents (names of values and types), and to edit them.

When launching OCamlBrowser by the command `ocamlbrowser`, the list of all the compiled modules available (see figure B.2) is displayed. One can add more modules by specifying a path to find them. From the menu `File`, one can launch a toplevel interactive loop or an editor in a new window.

When one of the modules is clicked on, a new window opens to display its contents (see figure B.3). By selecting a value, its type appears in bottom of the window.

In the main window, one can search on the name of a function. The result appears in a new window. The figure B.4 shows the result of a search on the word `create`.

There are other possibilities that we let the user discover.

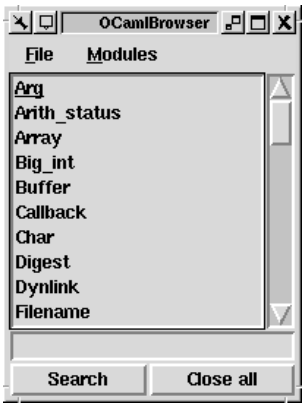


Figure B.2: OCamlBrowser : the main window

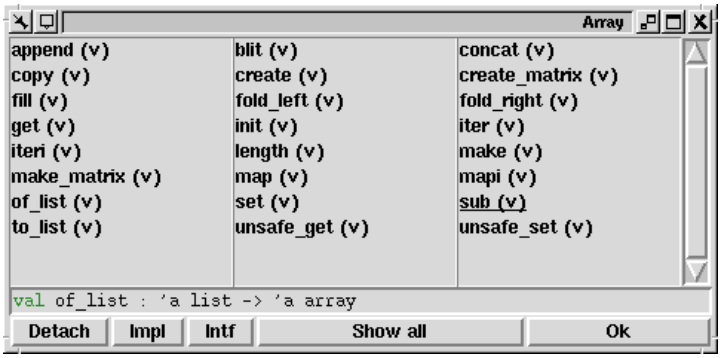


Figure B.3: OCamlBrowser : module contents

Figure B.4: OCamlBrowser : search for `create`

Bibliography

[AC96] María-Virginia Aponte and Giuseppe Castagna. Programmation modulaire avec surcharge et liaison tardive. In *Journées Francophones des Langages Applicatifs*. INRIA, January 1996.

[AHU83] Alfred Aho, John Hopcroft, and Jeffrey Ullman. *Data Structures and Algorithms*. Addison-Wesley, 1983.

[And91] G. Andrews. *Concurrent Programming : Principles and practices*. Benjamin Cumming, 1991.

[Ari90] Ben Ari. *Principles of Concurrent and Distributed Programming*. Prentice Hall, second edition, 1990.

[ASS96] Harold Abelson, Gerald Jay Sussman, and Julie Sussman. *Structure and Interpretation of Computer Programs*. MIT Press, second edition, 1996.

[ASU86] Alfred V. Aho, Ravi Sethi, and Jeffrey D. Ullman. *Compilers: principles, techniques, and tools*. Addison-Wesley, 1986.

[BLH00] Olivier Ballereau, Frédéric Loulergue, and Gaétan Hains. High level BSP programming: BSML and BSlambda. In Stephen Gilmore, editor, *Trends in Functional Programming, volume 2*. Intellect, 2000.

[CC92] Emmanuel Chailloux and Guy Cousineau. Programming images in ML. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1992.

[CCM87] Guy Cousineau, Pierre-Louis Curien, and Michel Mauny. The Categorical Abstract Machine. *Science of Computer Programming*, 8:173–202, 1987.

[CDM98] Rémy Card, Éric Dumas, and Franck Mével. *The Linux Kernel Book*. Wiley, John & Sons, 1998.

[CKL96] Emmanuel Chailloux, Laurent Kirsch, and Stéphane Lucas. Caml2sml, un outil d'aide à la traduction de Caml vers Sml. In *Journées Francophones des Langages Applicatifs*. INRIA, January 1996.

[CL99] Sylvain Conchon and Fabrice Le Fessant. JoCaml: mobile agents for Objective-Caml. In *International Symposium on Agent Systems and Applications*, 1999.

[CM98] Guy Cousineau and Michel Mauny. *The functional approach to programming*. Cambridge University Press, 1998.

[CP95] Paul Caspi and Marc Pouzet. A functional extension to LUSTRE. In *8th International Symposium on Languages for Intensional Programming*, Sydney, May 1995. World Scientific.

[CS94] Emmanuel Chailloux and Ascánder Suárez. mlPicTeX, a picture environment for LaTeX. In *Proceedings of the ACM SIGPLAN Workshop on ML and its Applications*, 1994.

[DDLP98] Marco Danelutto, Roberto Di Cosmo, Xavier Leroy, and Susanna Pelagatti. Parallel functional programming with skeletons: the ocamlp3l experiment. In *ML Workshop*. ACM SIGPLAN, 1998.

[DEMN98] Roland Ducournau, Jérôme Euzenat, Gérald Masini, and Amedeo Napoli, editors. *Langages et modèles à objets: état et perspectives de la recherche*. INRIA, 1998.

[Eng98] Emmanuel Engel. *Extensions sûres et praticables du système de types de ML en présence d'un langage de modules et de traits impératifs*. PhD thesis, Université Paris-Sud, Orsay, France, mai 1998.

[FC95] Christian Foisy and Emmanuel Chailloux. Caml Flight: a Portable SPMD Extension of ML for Distributed Memory Multiprocessors. In *Conference on High Performance Functional Computing*, April 1995.

[FF98] Robert B. Findler and Matthew Flatt. Modular Object-Oriented Programming with Units and Mixins. In *International Conference on Functional Programming*. ACM, 1998.

[FW00] Jun Furuse and Pierre Weis. Entrées/Sorties de valeurs en Caml. In *JFLA'2000 : Journées Francophones des Langages Applicatifs*, Mont Saint-Michel, January 2000. INRIA.

[GHJV95] Erich Gamma, Richard Helm, Ralph Johnson, and John Vlissides. *Design Patterns*. Addison-Wesley, 1995.

[HF+96] Pieter Hartel, Marc Feeley, et al. Benchmarking implementations of functional languages with "Pseudoknot", a float-intensive benchmark. *Journal of Functional Programming*, 6(4), 1996.

[HS94] Samuel P. Harbison and Guy L. Steele. *C: A reference manual*. Prentice-Hall, fourth edition, 1994.

[Hui97] Christian Huitema. *IPv6 – The New Internet Protocol*. Prentice Hall, 1997.

[Jon98] Richard Jones. *Garbage Collection: Algorithms for Automatic Dynamic Memory Management*. John Wiley & Sons, 1998.

[Ler90] Xavier Leroy. The ZINC experiment: an economical implementation of the ML language. Technical report 117, INRIA, 1990.

[Ler92] Xavier Leroy. Programmation du système Unix en Caml Light. Technical report 147, INRIA, 1992.

[LMB92] John R. Levine, Tony Mason, and Doug Brown. *Lex & Yacc*. O'Reilly, second edition, 1992.

[LRVD99] Xavier Leroy, Didier Rémy, Jérôme Vouillon, and Damien Doligez. The objective caml system release 2.04. Technical report, INRIA, December 1999.

[MdR92] Michel Mauny and Daniel de Rauglaudre. Parser in ML. Research report 1659, INRIA, avril 1992.

[MNC+91] Gérald Masini, Amedeo Napoli, Dominique Colnet, Daniel Léonard, and Karl Tombre. *Object-Oriented Languages*. Academic Press, New York, 1991.

[MT91] Robin Milner and Mads Tofte. *Commentary on Standard ML*. MIT Press, 1991.

[MTH97] Robin Milner, Mads Tofte, and Robert Harper. *The Definition of Standard ML (revised)*. MIT Press, 1997.

[Rep99] John Reppy. *Concurrent Programming in ML*. Cambridge University Press, 1999.

[Rob89] Eric S. Robert. Implementing exceptions in C. Technical Report SRC-40, Digital Equipment, 1989.

[Rou96] François Rouaix. A Web navigator with applets in Caml. In *Proceedings of the 5th International World Wide Web Conference, in Computer Networks and Telecommunications Networking*, volume 28, pages 1365–1371. Elsevier, May 1996.

[RV98] Didier Rémy and Jérôme Vouillon. Objective ML: An effective object-oriented extension to ML. *Theory And Practice of Object Systems*, 4(1):27–50, 1998. A preliminary version appeared in the proceedings of the 24th ACM Conference on Principles of Programming Languages, 1997.

[Sed88] Robert Sedgewick. *Algorithms*. Addison-Wesley, second edition, 1988.

[Ste92] W. Richard Stevens. *Advanced Programming in the UNIX Environment*. Addison-Wesley, 1992.

[Tho99] Simon Thompson. *Haskell: The Craft of Functional Programming*. Addison Wesley, seconde edition, 1999.

[Tur85] David A. Turner. Miranda: A non-strict functional language with polymorphic types. In J. Jouannaud, editor, *Proceedings International Conference on Functional Programming Languages and Computer Architecture*, volume 201 of *Lecture Notes in Computer Science*, pages 1–16, New York, NY, September 1985. Springer-Verlag.

[Wil92] Paul. R. Wilson. Uniprocessor garbage collection techniques. In *International Workshop on Memory Management*, number 637 in LNCS, pages 1–42. Springer-Verlag, 1992.

[Wri95] Andrew K. Wright. Simple imperative polymorphism. *Lisp and Symbolic Computation*, 8(4):343–356, 1995.

Index of concepts

— A —

— B —

— C —

— D —

— E —

— F —

— O —

— P —

— R —

— S —

Index of language elements

— Symboles —

— A —

— B —

— C —

— D —

— E —

— F —

— G —

— H —

— I —

— K —

— L —

— M —

— N —

— O —

— P —

— Q —

— R —

— S —

— T —

— U —

— V —

— W —

Liste de diffusion par messagerie électronique

Si vous souhaitez recevoir périodiquement les annonces de nouveaux produits O'Reilly en français, il vous suffit de souscrire un abonnement à la liste d'annonces

```
parutions-oreilly
```

Merci d'expédier en ce cas un message électronique à `majordomo@ora.de` contenant (dans le corps du message) :

```
subscribe parutions-oreilly votre_adresse_email
```

Exemple :

```
subscribe parutions-oreilly jean.dupond@ici.fr
```

Cette liste ne véhicule que des annonces et non des *discussions*, vous ne pourrez par conséquent pas y poster. Son volume ne dépasse pas quatre messages par mois.

En cas de problème technique écrire à

```
parutions-oreilly-owner@ora.de
```

Site Web

Notre site Web `http://www.editions-oreilly.fr/` diffuse diverses informations :

- le catalogue des produits proposés par les éditions O'Reilly,
- les *errata* de nos ouvrages,
- des archives abritant les exemples,
- la liste des revendeurs.