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 = "93326215443944152681699238856266700490715968264381..."
```

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 = "27048138294215260932671947108075308336779383827810..."
```

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.