# The Objective Caml system release 2.02

Documentation and user's manual

Xavier Leroy
(with Didier Rémy, Jérôme Vouillon and Damien Doligez)

March 4, 1999

# Contents

# Foreword

This manual documents the release 2.02 of the Objective Caml system. It is organized as follows.

- Part I, “An introduction to Objective Caml”, gives an overview of the language.
- Part II, “The Objective Caml language”, is the reference description of the language.
- Part III, “The Objective Caml tools”, documents the compilers, toplevel system, and programming utilities.
- Part IV, “The Objective Caml library”, describes the modules provided in the standard library.
- Part V, “Appendix”, contains an index of all identifiers defined in the standard library, and an index of keywords.

## Conventions

Objective Caml runs on several operating systems. The parts of this manual that are specific to one operating system are presented as shown below:

**Unix:**
This is material specific to Unix.

**Windows:**
This is material specific to MS Windows (NT and 95).

## License

## Availability

The complete Objective Caml distribution resides on the machine `ftp.inria.fr`. The distribution files can be transferred by anonymous FTP:

| | |
|---|---|
| Host: | `ftp.inria.fr` (Internet address `192.93.2.54`) |
| Login name: | `anonymous` |
| Password: | your e-mail address |
| Directory: | `lang/caml-light` |
| Files: | see the index in file `README` |

More information on the Caml family of languages is also available on the World Wide Web, `http://caml.inria.fr/`.

# Part I

# An introduction to Objective Caml

# Chapter 1

# The core language

This part of the manual is a tutorial introduction to the Objective Caml language. A good familiarity with programming in a conventional languages (say, Pascal or C) is assumed, but no prior exposure to functional languages is required. The present chapter introduces the core language. Chapter 2 deals with the object-oriented features, and chapter 3 with the module system.

## 1.1 Basics

For this overview of Caml, we use the interactive system, which is started by running `ocaml` from the Unix shell, or by launching the `OCamlwin.exe` application under Windows. This tutorial is presented as the transcript of a session with the interactive system: lines starting with `#` represent user input; the system responses are printed below, without a leading `#`.

Under the interactive system, the user types Caml phrases, terminated by `;;`, in response to the `#` prompt, and the system compiles them on the fly, executes them, and prints the outcome of evaluation. Phrases are either simple expressions, or `let` definitions of identifiers (either values or functions).

```
# 1+2*3;;
- : int = 7

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265359

# let square x = x *. x;;
val square : float -> float = <fun>

# square(sin pi) +. square(cos pi);;
- : float = 1
```

The Caml system computes both the value and the type for each phrase. Even function parameters need no explicit type declaration: the system infers their types from their usage in the function. Notice also that integers and floating-point numbers are distinct types, with distinct operators: `+` and `*` operate on integers, but `+.` and `*.` operate on floats.

```
# 1.0 * 2;;
Characters 0-3:
This expression has type float but is here used with type int
```

Recursive functions are defined with the `let rec` binding:

```
# let rec fib n =
#   if n < 2 then 1 else fib(n-1) + fib(n-2);;
val fib : int -> int = <fun>

# fib 10;;
- : int = 89
```

## 1.2 Data types

In addition to integers and floating-point numbers, Caml offers the usual basic data types: booleans, characters, and character strings.

```
# (1 < 2) = false;;
- : bool = false

# 'a';;
- : char = 'a'

# "Hello world";;
- : string = "Hello world"
```

Predefined data structures include tuples, arrays, and lists. General mechanisms for defining your own data structures are also provided. They will be covered in more details later; for now, we concentrate on lists. Lists are either given in extension as a bracketed list of semicolon-separated elements, or built from the empty list `[]` (pronounce "nil") by adding elements in front using the `::` ("cons") operator.

```
# let l = ["is"; "a"; "tale"; "told"; "etc."];;
val l : string list = ["is"; "a"; "tale"; "told"; "etc."]

# "Life" :: l;;
- : string list = ["Life"; "is"; "a"; "tale"; "told"; "etc."]
```

As with all other Caml data structures, lists do not need to be explicitly allocated and deallocated from memory: all memory management is entirely automatic in Caml. Similarly, there is no explicit handling of pointers: the Caml compiler silently introduces pointers where necessary.

As with most Caml data structures, inspecting and destructuring lists is performed by pattern-matching. List patterns have the exact same shape as list expressions, with identifier representing unspecified parts of the list. As an example, here is insertion sort on a list:

```
# let rec sort lst =
#   match lst with
#     [] -> []
#   | head :: tail -> insert head (sort tail)
# and insert elt lst =
#   match lst with
#     [] -> [elt]
#   | head :: tail -> if elt <= head then elt :: lst else head :: insert elt tail
# ;;
val sort : 'a list -> 'a list = <fun>
val insert : 'a -> 'a list -> 'a list = <fun>
```

```
# sort l;;
- : string list = ["a"; "etc."; "is"; "tale"; "told"]
```

The type inferred for `sort`, `'a list -> 'a list`, means that `sort` can actually apply to lists of any type, and returns a list of the same type. The type `'a` is a *type variable*, and stands for any given type. The reason why `sort` can apply to lists of any type is that the comparisons (`=`, `<=`, etc.) are *polymorphic* in Caml: they operate between any two values of the same type. This makes `sort` itself polymorphic over all list types.

```
# sort [6;2;5;3];;
- : int list = [2; 3; 5; 6]
```

```
# sort [3.14; 2.718];;
- : float list = [2.718; 3.14]
```

The `sort` function above does not modify its input list: it builds and returns a new list containing the same elements as the input list, in ascending order. There is actually no way in Caml to modify in-place a list once it is built: we say that lists are *immutable* data structures. Most Caml data structures are immutable, but a few (most notably arrays) are *mutable*, meaning that they can be modified in-place at any time.

## 1.3 Functions as values

Caml is a functional language: functions in the full mathematical sense are supported and can be passed around freely just as any other piece of data. For instance, here is a `deriv` function that takes any float function as argument and returns an approximation of its derivative function:

```
# let deriv f dx = function x -> (f(x +. dx) -. f(x)) /. dx;;
val deriv : (float -> float) -> float -> float -> float = <fun>
```

```
# let cos' = deriv sin 1e-6;;
val cos' : float -> float = <fun>
```

```
# cos' pi;;
- : float = -1.00000000014
```

Even function composition is definable:

```
# let compose f g = function x -> f(g(x));;
val compose : ('a -> 'b) -> ('c -> 'a) -> 'c -> 'b = <fun>
```

```
# let cos2 = compose square cos;;
val cos2 : float -> float = <fun>
```

Functions that take other functions as arguments are called “functionals”, or “higher-order functions”. Functionals are especially useful to provide iterators or similar generic operations over a data structure. For instance, the standard Caml library provides a `List.map` functional that applies a given function to each element of a list, and returns the list of the results:

```
# List.map (function n -> n * 2 + 1) [0;1;2;3;4];;
- : int list = [1; 3; 5; 7; 9]
```

This functional, along with a number of other list and array functionals, is predefined because it is often useful, but there is nothing magic with it: it can easily be defined as follows.

```
# let rec map f l =
#   match l with
#     [] -> []
#   | hd :: tl -> f hd :: map f tl;;
val map : ('a -> 'b) -> 'a list -> 'b list = <fun>
```

## 1.4 Records and variants

User-defined data structures include records and variants. Both are defined with the `type` declaration. Here, we declare a record type to represent rational numbers.

```
# type ratio = {num: int; denum: int};;
type ratio = { num: int; denum: int }
```

```
# let add_ratio r1 r2 =
#   {num = r1.num * r2.denum + r2.num * r1.denum;
#    denum = r1.denum * r2.denum};;
val add_ratio : ratio -> ratio -> ratio = <fun>
```

```
# add_ratio {num=1; denum=3} {num=2; denum=5};;
- : ratio = {num=11; denum=15}
```

The declaration of a variant type lists all possible shapes for values of that type. Each case is identified by a name, called a constructor, which serves both for constructing values of the variant type and inspecting them by pattern-matching. Constructor names are capitalized to distinguish them from variable names (which must start with a lowercase letter). For instance, here is a variant type for doing mixed arithmetic (integers and floats):

```
# type number = Int of int | Float of float | Error;;
type number = | Int of int | Float of float | Error
```

This declaration expresses that a value of type `number` is either an integer, a floating-point number, or the constant `Error` representing the result of an invalid operation (e.g. a division by zero).

Enumerated types are a special case of variant types, where all alternatives are constants:

```
# type sign = Positive | Negative;;
type sign = | Positive | Negative
```

```
# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```

To define arithmetic operations for the `number` type, we use pattern-matching on the two numbers involved:

```
# let add_num n1 n2 =
#   match (n1, n2) with
#     (Int i1, Int i2) ->
#       (* Check for overflow of integer addition *)
#       if sign_int i1 = sign_int i2 && sign_int(i1 + i2) <> sign_int i1
#       then Float(float i1 +. float i2)
#       else Int(i1 + i2)
#   | (Int i1, Float f2) -> Float(float i1 +. f2)
#   | (Float f1, Int i2) -> Float(f1 +. float i2)
#   | (Float f1, Float f2) -> Float(f1 +. f2)
#   | (Error, _) -> Error
#   | (_, Error) -> Error;;
val add_num : number -> number -> number = <fun>

# add_num (Int 123) (Float 3.14159);;
- : number = Float 126.14159
```

The most common usage of variant types is to describe recursive data structures. Consider for example the type of binary trees:

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

This definition reads as follow: a binary tree containing values of type `'a` (an arbitrary type) is either empty, or is a node containing one value of type `'a` and two subtrees containing also values of type `'a`, that is, two `'a btree`.

Operations on binary trees are naturally expressed as recursive functions following the same structure as the type definition itself. For instance, here are functions performing lookup and insertion in ordered binary trees (elements increase from left to right):

```
# let rec member x btree =
#   match btree with
#     Empty -> false
#   | Node(y, left, right) ->
#       if x = y then true else
#       if x < y then member x left else member x right;;
val member : 'a -> 'a btree -> bool = <fun>

# let rec insert x btree =
#   match btree with
#     Empty -> Node(x, Empty, Empty)
#   | Node(y, left, right) ->
#       if x <= y then Node(y, insert x left, right)
#                 else Node(y, left, insert x right);;
val insert : 'a -> 'a btree -> 'a btree = <fun>
```

## 1.5 Imperative features

Though all examples so far were written in purely applicative style, Caml is also equipped with full imperative features. This includes the usual `while` and `for` loops, as well as mutable data structures such as arrays. Arrays are either given in extension between `[|` and `|]` brackets, or allocated and initialized with the `Array.create` function, then filled up later by assignments. For instance, the function below sums two vectors (represented as float arrays) componentwise.

```
# let add_vect v1 v2 =
#   let len = min (Array.length v1) (Array.length v2) in
#   let res = Array.create len 0.0 in
#   for i = 0 to len - 1 do
#     res.(i) <- v1.(i) +. v2.(i)
#   done;
#   res;;
val add_vect : float array -> float array -> float array = <fun>

# add_vect [| 1.0; 2.0 |] [| 3.0; 4.0 |];;
- : float array = [|4; 6|]
```

Record fields can also be modified by assignment, provided they are declared `mutable` in the definition of the record type:

```
# type mutable_point = { mutable x: float; mutable y: float };;
type mutable_point = { mutable x: float; mutable y: float }

# let translate p dx dy =
#   p.x <- p.x +. dx; p.y <- p.y +. dy;;
val translate : mutable_point -> float -> float -> unit = <fun>

# let mypoint = { x = 0.0; y = 0.0 };;
val mypoint : mutable_point = {x=0; y=0}

# translate mypoint 1.0 2.0;;
- : unit = ()

# mypoint;;
- : mutable_point = {x=1; y=2}
```

Caml has no built-in notion of variable – identifiers whose current value can be changed by assignment. (The `let` binding is not an assignment, it introduces a new identifier with a new scope.) However, the standard library provides references, which are mutable indirection cells (or one-element arrays), with operators `!` to fetch the current contents of the reference and `:=` to assign the contents. Variables can then be emulated by `let`-binding a reference. For instance, here is an in-place insertion sort over arrays:

```
# let insertion_sort a =
#   for i = 1 to Array.length a - 1 do
#     let val_i = a.(i) in
#     let j = ref i in
#     while !j > 0 && val_i < a.(!j - 1) do
#       a.(!j) <- a.(!j - 1);
#       j := !j - 1
#     done;
#     a.(!j) <- val_i
#   done;;
val insertion_sort : 'a array -> unit = <fun>
```

References are also useful to write functions that maintain a current state between two calls to the function. For instance, the following pseudo-random number generator keeps the last returned number in a reference:

```
# let current_rand = ref 0;;
val current_rand : int ref = {contents=0}

# let random () =
#   current_rand := !current_rand * 25713 + 1345;
#   !current_rand;;
val random : unit -> int = <fun>
```

Again, there is nothing magic with references: they are implemented as a one-field mutable record, as follows.

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

# let (!) r = r.contents;;
val ! : 'a ref -> 'a = <fun>

# let (:=) r newval = r.contents <- newval;;
val := : 'a ref -> 'a -> unit = <fun>
```

## 1.6 Exceptions

Caml provides exceptions for signalling and handling exceptional conditions. Exceptions can also be used as a general-purpose non-local control structure. Exceptions are declared with the `exception` construct, and signalled with the `raise` operator. For instance, the function below for taking the head of a list uses an exception to signal the case where an empty list is given.

```
# exception Empty_list;;
exception Empty_list

# let head l =
#   match l with
#     [] -> raise Empty_list
#   | hd :: tl -> hd;;
val head : 'a list -> 'a = <fun>

# head [1;2];;
- : int = 1

# head [];;
Uncaught exception: Empty_list
```

Exceptions are used throughout the standard library to signal cases where the library functions cannot complete normally. For instance, the `List.assoc` function, which returns the data associated with a given key in a list of (key, data) pairs, raises the predefined exception `Not_found` when the key does not appear in the list:

```
# List.assoc 1 [(0, "zero"); (1, "one")];;
- : string = "one"
```

```
# List.assoc 2 [(0, "zero"); (1, "one")];;
Uncaught exception: Not_found
```

Exceptions can be trapped with the `try`...`with` construct:

```
# let name_of_binary_digit digit =
#   try
#     List.assoc digit [0, "zero"; 1, "one"]
#   with Not_found ->
#     "not a binary digit";;
val name_of_binary_digit : int -> string = <fun>
```

```
# name_of_binary_digit 0;;
- : string = "zero"
```

```
# name_of_binary_digit (-1);;
- : string = "not a binary digit"
```

The `with` part is actually a regular pattern-matching on the exception value. Thus, several exceptions can be caught by one `try`...`with` construct. Also, finalization can be performed by trapping all exceptions, performing the finalization, then raising again the exception:

```
# let temporarily_set_reference ref newval funct =
#   let oldval = !ref in
#   try
#     ref := newval;
#     let res = funct () in
#     ref := oldval;
#     res
#   with x ->
#     ref := oldval;
#     raise x;;
val temporarily_set_reference : 'a ref -> 'a -> (unit -> 'b) -> 'b = <fun>
```

## 1.7 Symbolic processing of expressions

We finish this introduction with a more complete example representative of the use of Caml for symbolic processing: formal manipulations of arithmetic expressions containing variables. The following variant type describes the expressions we shall manipulate:

```
# type expression =
#     Const of float
#   | Var of string
#   | Sum of expression * expression    (* e1 + e2 *)
#   | Diff of expression * expression   (* e1 - e2 *)
#   | Prod of expression * expression   (* e1 * e2 *)
#   | Quot of expression * expression   (* e1 / e2 *)
# ;;
type expression =
  | Const of float
  | Var of string
  | Sum of expression * expression
  | Diff of expression * expression
  | Prod of expression * expression
  | Quot of expression * expression
```

We first define a function to evaluate an expression given an environment that maps variable names to their values. For simplicity, the environment is represented as an association list.

```
# exception Unbound_variable of string;;
exception Unbound_variable of string
```

```
# let rec eval env exp =
#   match exp with
#     Const c -> c
#   | Var v ->
#       (try List.assoc v env with Not_found -> raise(Unbound_variable v))
#   | Sum(f, g) -> eval env f +. eval env g
#   | Diff(f, g) -> eval env f -. eval env g
#   | Prod(f, g) -> eval env f *. eval env g
#   | Quot(f, g) -> eval env f /. eval env g;;
val eval : (string * float) list -> expression -> float = <fun>
```

```
# eval [("x", 1.0); ("y", 3.14)] (Prod(Sum(Var "x", Const 2.0), Var "y"));;
- : float = 9.42
```

Now for a real symbolic processing, we define the derivative of an expression with respect to a variable dv:

```
# let rec deriv exp dv =
#   match exp with
#     Const c -> Const 0.0
#   | Var v -> if v = dv then Const 1.0 else Const 0.0
#   | Sum(f, g) -> Sum(deriv f dv, deriv g dv)
#   | Diff(f, g) -> Diff(deriv f dv, deriv g dv)
#   | Prod(f, g) -> Sum(Prod(f, deriv g dv), Prod(deriv f dv, g))
#   | Quot(f, g) -> Quot(Diff(Prod(deriv f dv, g), Prod(f, deriv g dv)),
#                        Prod(g, g))
# ;;
val deriv : expression -> string -> expression = <fun>
```

```
# deriv (Quot(Const 1.0, Var "x")) "x";;
- : expression =
Quot
 (Diff (Prod (Const 0, Var "x"), Prod (Const 1, Const 1)),
  Prod (Var "x", Var "x"))
```

## 1.8 Pretty-printing and parsing

As shown in the examples above, the internal representation (also called *abstract syntax*) of expressions quickly becomes hard to read and write as the expressions get larger. We need a printer and a parser to go back and forth between the abstract syntax and the *concrete syntax*, which in the case of expressions is the familiar algebraic notation (e.g. `2*x+1`).

For the printing function, we take into account the usual precedence rules (i.e. `*` binds tighter than `+`) to avoid printing unnecessary parentheses. To this end, we maintain the current operator precedence and print parentheses around an operator only if its precedence is less than the current precedence.

```
# let print_expr exp =
#   (* Local function definitions *)
#   let open_paren prec op_prec =
#     if prec > op_prec then print_string "(" in
#   let close_paren prec op_prec =
#     if prec > op_prec then print_string ")" in
#   let rec print prec exp =     (* prec is the current precedence *)
#     match exp with
#       Const c -> print_float c
#     | Var v -> print_string v
#     | Sum(f, g) ->
#         open_paren prec 0;
#         print 0 f; print_string " + "; print 0 g;
#         close_paren prec 0
#     | Diff(f, g) ->
#         open_paren prec 0;
#         print 0 f; print_string " - "; print 1 g;
#         close_paren prec 0
#     | Prod(f, g) ->
#         open_paren prec 2;
#         print 2 f; print_string " * "; print 2 g;
#         close_paren prec 2
#     | Quot(f, g) ->
#         open_paren prec 2;
#         print 2 f; print_string " / "; print 3 g;
#         close_paren prec 2
#   in print 0 exp;;
val print_expr : expression -> unit = <fun>
```

```
# let e = Sum(Prod(Const 2.0, Var "x"), Const 1.0);;
val e : expression = Sum (Prod (Const 2, Var "x"), Const 1)

# print_expr e; print_newline();;
2 * x + 1
- : unit = ()

# print_expr (deriv e "x"); print_newline();;
2 * 1 + 0 * x + 0
- : unit = ()
```

Parsing (transforming concrete syntax into abstract syntax) is usually more delicate. Caml offers several tools to help write parsers: on the one hand, Caml versions of the lexer generator Lex and the parser generator Yacc (see chapter 11), which handle LALR(1) languages using push-down automata; on the other hand, a predefined type of streams (of characters or tokens) and pattern-matching over streams, which facilitate the writing of recursive-descent parsers for LL(1) languages. An example using `ocamllex` and `ocamlyacc` is given in chapter 11. Here, we will use stream parsers.

```
# open Genlex;;

# let lexer = make_lexer ["("; ")"; "+"; "-"; "*"; "/"];;
val lexer : char Stream.t -> Genlex.token Stream.t = <fun>
```

For the lexical analysis phase (transformation of the input text into a stream of tokens), we use a "generic" lexer provided in the standard library module `Genlex`. The `make_lexer` function takes a list of keywords and returns a lexing function that "tokenizes" an input stream of characters. Tokens are either identifiers, keywords, or literals (integer, floats, characters, strings). Whitespace and comments are skipped.

```
# let token_stream = lexer(Stream.of_string "1.0 +x");;
val token_stream : Genlex.token Stream.t = <abstr>

# Stream.next token_stream;;
- : Genlex.token = Float 1

# Stream.next token_stream;;
- : Genlex.token = Kwd "+"

# Stream.next token_stream;;
- : Genlex.token = Ident "x"
```

The parser itself operates by pattern-matching on the stream of tokens. As usual with recursive descent parsers, we use several intermediate parsing functions to reflect the precedence and associativity of operators. Pattern-matching over streams is more powerful than on regular data structures, as it allows recursive calls to parsing functions inside the patterns, for matching sub-components of the input stream. See chapter 6 for more details.

```
# let rec parse_expr = parser
#     [< e1 = parse_mult; e = parse_more_adds e1 >] -> e
# and parse_more_adds e1 = parser
#     [< 'Kwd "+"; e2 = parse_mult; e = parse_more_adds (Sum(e1, e2)) >] -> e
#   | [< 'Kwd "-"; e2 = parse_mult; e = parse_more_adds (Diff(e1, e2)) >] -> e
#   | [< >] -> e1
# and parse_mult = parser
#     [< e1 = parse_simple; e = parse_more_mults e1 >] -> e
# and parse_more_mults e1 = parser
#     [< 'Kwd "*"; e2 = parse_simple; e = parse_more_mults (Prod(e1, e2)) >] -> e
#   | [< 'Kwd "/"; e2 = parse_simple; e = parse_more_mults (Quot(e1, e2)) >] -> e
#   | [< >] -> e1
# and parse_simple = parser
#     [< 'Ident s >] -> Var s
#   | [< 'Int i >] -> Const(float i)
#   | [< 'Float f >] -> Const f
#   | [< 'Kwd "("; e = parse_expr; 'Kwd ")" >] -> e;;
val parse_expr : Genlex.token Stream.t -> expression = <fun>
val parse_more_adds : expression -> Genlex.token Stream.t -> expression =
  <fun>
val parse_mult : Genlex.token Stream.t -> expression = <fun>
val parse_more_mults : expression -> Genlex.token Stream.t -> expression =
  <fun>
val parse_simple : Genlex.token Stream.t -> expression = <fun>
```

Composing the lexer and parser, we finally obtain a function to read an expression from a character string:

```
# let read_expr s = parse_expr(lexer(Stream.of_string s));;
val read_expr : string -> expression = <fun>
```

```
# read_expr "2*(x+y)";;
- : expression = Prod (Const 2, Sum (Var "x", Var "y"))
```

## 1.9 Standalone Caml programs

All examples given so far were executed under the interactive system. Caml code can also be compiled separately and executed non-interactively using the batch compilers `ocamlc` or `ocamlopt`. The source code must be put in a file with extension `.ml`. It consists of a sequence of phrases, which will be evaluated at runtime in their order of appearance in the source file. Unlike in interactive mode, types and values are not printed automatically; the program must call printing functions explicitly to produce some output. Here is a sample standalone program to print Fibonacci numbers:

```
(* File fib.ml *)
let rec fib n =
  if n < 2 then 1 else fib(n-1) + fib(n-2);;
let main () =
  let arg = int_of_string Sys.argv.(1) in
  print_int(fib arg);
  print_newline();
  exit 0;;
main ();;
```

`Sys.argv` is an array of strings containing the command-line parameters. `Sys.argv.(1)` is thus the first command-line parameter. The program above is compiled and executed with the following shell commands:

```
$ ocamlc -o fib fib.ml
$ ./fib 10
89
$ ./fib 20
10946
```

# Chapter 2

# Objects in Caml

*(Chapter written by Jérôme Vouillon and Didier Rémy)*

This chapter gives an overview of the object-oriented features of Objective Caml.

## 2.1 Classes and objects

The class `point` has one instance variable `x` and two methods `get_x` and `move`. The initial value of the instance variable is `0`. The variable `x` is declared mutable, so the method `move` can change its value.

```
# class point =
#   object
#     val mutable x = 0
#     method get_x = x
#     method move d = x <- x + d
#   end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

We now create a new point `p`.

```
# let p = new point;;
val p : point = <obj>
```

Note that the type of `p` is `point`. This is an abbreviation automatically defined by the class definition above. It stands for the object type `<get_x : int; move : int -> unit>`, listing the methods of class `point` along with their types.

Let us apply some methods to `p`:

```
# p#get_x;;
- : int = 0

# p#move 3;;
- : unit = ()

# p#get_x;;
- : int = 3
```

The evaluation of the body of a class only takes place at object creation time. Therefore, in the following example, the instance variable x is initialized to different values for two different objects.

```
# let x0 = ref 0;;
val x0 : int ref = {contents=0}
```

```
# class point =
#   object
#     val mutable x = incr x0; !x0
#     method get_x = x
#     method move d = x <- x + d
#   end;;
class point :
  object val mutable x : int method get_x : int method move : int -> unit end
```

```
# new point#get_x;;
- : int = 1
```

```
# new point#get_x;;
- : int = 2
```

The class point can also be abstracted over the initial values of points.

```
# class point = fun x_init ->
#   object
#     val mutable x = x_init
#     method get_x = x
#     method move d = x <- x + d
#   end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

As for declaration of functions, the above definition can be abbreviated as:

```
# class point x_init =
#   object
#     val mutable x = x_init
#     method get_x = x
#     method move d = x <- x + d
#   end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```

An instance of the class point is now a function that expects an initial parameter to create a point object:

```
# new point;;
- : int -> point = <fun>
```

```
# let p = new point 7;;
val p : point = <obj>
```

The parameter `x_init` is, of course, visible in the whole body of the definition, including methods. For instance, the method `get_offset` in the class below returns the position of the object to the origin.

```
# class point x_init =
#   object
#     val mutable x = x_init
#     method get_x = x
#     method get_offset = x - x_init
#     method move d = x <- x + d
#   end;;
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

Expressions can be evaluated and bound before defining the object body of the class. This is useful to enforce invariants. For instance, points can be automatically adjusted to grid as follows:

```
# class adjusted_point x_init =
#   let origin = (x_init / 10) * 10 in
#   object
#     val mutable x = origin
#     method get_x = x
#     method get_offset = x - origin
#     method move d = x <- x + d
#   end;;
class adjusted_point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

(One could also raise an exception if the `x_init` coordinate is not on the grid.) In fact, the same effect could here be obtained by calling the definition of class `point` with the value of the `origin`.

```
# class adjusted_point x_init =  point ((x_init / 10) * 10);;
class adjusted_point : int -> point
```

An alternative solution would have been to define the adjustment in a special allocation function:

```
# let new_adjusted_point x_init = new point ((x_init / 10) * 10);;
val new_adjusted_point : int -> point = <fun>
```

However, the former pattern is generally more appropriate, since the code for adjustment is part of the definition of the class and will be inherited.

This ability provides class constructors as can be found in other languages. Several constructors can be defined this way to build objects of the same class but with different initialization patterns.

## 2.2 Reference to self

A method can also send messages to self (that is, the current object). For that, self must be explicitly bound, here to the variable `s` (`s` could be any identifier, even though we will often choose the name `self`.)

```
# class printable_point x_init =
#   object (s)
#     val mutable x = x_init
#     method get_x = x
#     method move d = x <- x + d
#     method print = print_int s#get_x
#   end;;
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end

# let p = new printable_point 7;;
val p : printable_point = <obj>

# p#print;;
7- : unit = ()
```

Dynamically, the variable `s` is bound at the invocation of a method. In particular, when the class `printable_point` will be inherited, the variable `s` will be correctly bound to the object of the subclass.

## 2.3 Initializers

Let-bindings within class definitions are evaluated before the object is constructed. It is also possible to evaluate an expression immediately after the object has been built. Such code is written as an anonymous hidden method called an initializer. Therefore, is can access self and the instance variables.

```
# class printable_point x_init =
#   let origin = (x_init / 10) * 10 in
#   object (self)
#     val mutable x = origin
#     method get_x = x
#     method move d = x <- x + d
#     method print = print_int self#get_x
#     initializer print_string "new point at "; self#print; print_newline()
#   end;;
class printable_point :
  int ->
  object
    val mutable x : int
    method get_x : int
    method move : int -> unit
    method print : unit
  end
```

```
# let p = new printable_point 17;;
new point at 10
val p : printable_point = <obj>
```

Initializers cannot be overridden. On the contrary, all initializers are evaluated sequentially. Initializers are particularly useful to enforce invariants. Another example can be seen in section 4.1.

## 2.4 Virtual methods

It is possible to declare a method without actually defining it, using the keyword `virtual`. This method will be provided latter in subclasses. A class containing virtual methods must be flagged virtual, and cannot be instantiated (that is, no object of this class can be created). It still defines abbreviations (treating virtual methods as other methods.)

```
# class virtual abstract_point x_init =
#   object (self)
#     val mutable x = x_init
#     method virtual get_x : int
#     method get_offset = self#get_x - x_init
#     method virtual move : int -> unit
#   end;;
class virtual abstract_point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method virtual get_x : int
    method virtual move : int -> unit
  end
```

```
# class point x_init =
#   object
#     inherit abstract_point x_init
#     method get_x = x
#     method move d = x <- x + d
#   end;;
```

```
class point :
  int ->
  object
    val mutable x : int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

## 2.5 Private methods

Private methods are methods that do not appear in object interfaces. They can only be invoked from other methods of the same object.

```
# class restricted_point x_init =
#   object (self)
#     val mutable x = x_init
#     method get_x = x
#     method private move d = x <- x + d
#     method bump = self#move 1
#   end;;
class restricted_point :
  int ->
  object
    val mutable x : int
    method bump : unit
    method get_x : int
    method private move : int -> unit
  end
```

```
# let p = new restricted_point 0;;
val p : restricted_point = <obj>
```

```
# p#move 10;;
Characters 0-1:
This expression has type restricted_point
It has no method move
```

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

Private methods are inherited (they are by default visible in subclasses), unless they are hidden by signature matching, as described below.

Private methods can be made public in a subclass.

```
# class point_again x =
#   object (self)
#     inherit restricted_point x
#     method virtual move : _
#   end;;
```

```
class point_again :
  int ->
  object
    val mutable x : int
    method bump : unit
    method get_x : int
    method move : int -> unit
  end
```

The annotation `virtual` here is only used to mentioned a method without providing its definition. An alternative definition is

```
# class point_again x =
#   object (self : < move : _; ..> )
#     inherit restricted_point x
#   end;;
class point_again :
  int ->
  object
    val mutable x : int
    method bump : unit
    method get_x : int
    method move : int -> unit
  end
```

One could think that a private method should remain private in a subclass. However, it since the method is visible in a subclass, it is always possible pick it's code and define a method of the same name that run that code, so yet another (heavier) solution would be:

```
# class point_again x =
#   object (self : < move : _; ..> )
#     inherit restricted_point x as super
#     method move = super#move
#   end;;
class point_again :
  int ->
  object
    val mutable x : int
    method bump : unit
    method get_x : int
    method move : int -> unit
  end
```

## 2.6 Class interfaces

Class interfaces are inferred from class definitions. They may also be defined directly and used to restrict the type of a class. As class declarations, they also define a new type constructor.

```
# class type restricted_point_type =
#   object
#     method get_x : int
#     method bump : unit
# end;;
class type restricted_point_type =
  object method bump : unit method get_x : int end
```

```
# fun (x : restricted_point_type) -> x;;
- : restricted_point_type -> restricted_point_type = <fun>
```

In addition to documentation, these class interfaces can be used to constrain the type of a class. Both instance variables and concrete private methods can be hidden by a class type constraint. Public and virtual methods, however, cannot.

```
# class restricted_point' x = (restricted_point x : restricted_point_type);;
class restricted_point' : int -> restricted_point_type
```

Or, equivalently:

```
# class restricted_point' = (restricted_point : int -> restricted_point_type);;
class restricted_point' : int -> restricted_point_type
```

The interface of a class can also be specified in a module signature, and used to restrict the inferred signature of a module.

```
# module type POINT = sig
#   class restricted_point' : int ->
#     object
#       method get_x : int
#       method bump : unit
#     end
# end;;
module type POINT =
  sig
    class restricted_point' :
      int -> object method bump : unit method get_x : int end
  end
```

```
# module Point : POINT = struct
#   class restricted_point' = restricted_point
# end;;
module Point : POINT
```

## 2.7 Inheritance

We illustrate inheritance by defining a class of colored points that inherits from the class of points. This class has all instance variables and all methods of class `point`, plus a new instance variable `c` and a new method `color`.

```
# class colored_point x (c : string) =
#   object
#     inherit point x
#     val c = c
#     method color = c
#   end;;
class colored_point :
  int ->
  string ->
  object
    val c : string
    val mutable x : int
    method color : string
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end
```

```
# let p' = new colored_point 5 "red";;
val p' : colored_point = <obj>
```

```
# p'#get_x, p'#color;;
- : int * string = 5, "red"
```

A point and a colored point have incompatible types, since a point has no method `color`. However, the function `get_x` below is a generic function applying method `get_x` to any object `p` that has this method (and possibly some others, which are represented by an ellipsis in the type). Thus, it applies to both points and colored points.

```
# let get_succ_x p = p#get_x + 1;;
val get_succ_x : < get_x : int; .. > -> int = <fun>
```

```
# get_succ_x p + get_succ_x p';;
- : int = 8
```

Methods need not be declared previously, as shown by the example:

```
# let set_x p = p#set_x;;
val set_x : < set_x : 'a; .. > -> 'a = <fun>
```

```
# let incr p = set_x p (get_succ_x p);;
val incr : < get_x : int; set_x : int -> 'a; .. > -> 'a = <fun>
```

## 2.8 Multiple inheritance

Multiple inheritance is allowed. Only the last definition of a method is kept: the redefinition in a subclass of a method that was visible in the parent class overrides the definition in the parent class. Previous definitions of a method can be reused by binding the related ancestor. Below, `super` is bound to the ancestor `printable_point`. The name `super` is not actually a variable and can only be used to select a method as in `super#print`.

```
# class printable_colored_point y c =
#   object (self)
#     val c = c
#     method color = c
#     inherit printable_point y as super
#     method print =
#       print_string "(";
#       super#print;
#       print_string ", ";
#       print_string (self#color);
#       print_string ")"
#   end;;
class printable_colored_point :
  int ->
  string ->
  object
    val c : string
    val mutable x : int
    method color : string
    method get_x : int
    method move : int -> unit
    method print : unit
  end
```

```
# let p' = new printable_colored_point 17 "red";;
new point at (10, red)
val p' : printable_colored_point = <obj>
```

```
# p'#print;;
(10, red)- : unit = ()
```

A private method that has been hidden in the parent class is no more visible, and is thus not overridden. This also applies to initializers: all initializers along the class hierarchy are evaluated, in the order they are introduced.

## 2.9 Parameterized classes

Reference cells can also be implemented as objects. The naive definition fails to typecheck:

```
# class ref x_init =
#   object
#     val mutable x = x_init
#     method get = x
#     method set y = x <- y
#   end;;
Characters 5-106:
Some type variables are unbound in this type:
  class ref :
    'a ->
    object val mutable x : 'a method get : 'a method set : 'a -> unit end
The method get has type 'a where 'a is unbound
```

The reason is that at least one of the methods has a polymorphic type (here, the type of the value stored in the reference cell), thus the class should be parametric, or the method type should be constrained to a monomorphic type. A monomorphic instance of the class could be defined by:

```
# class ref (x_init:int) =
#   object
#     val mutable x = x_init
#     method get = x
#     method set y = x <- y
#   end;;
class ref :
  int ->
  object val mutable x : int method get : int method set : int -> unit end
```

A class for polymorphic references must explicitly list the type parameters in its declaration. Class type parameters are always listed between `[` and `]`. The type parameters must also be bound somewhere in the class body by a type constraint.

```
# class ['a] ref x_init =
#   object
#     val mutable x = (x_init : 'a)
#     method get = x
#     method set y = x <- y
#   end;;
class ['a] ref :
  'a -> object val mutable x : 'a method get : 'a method set : 'a -> unit end
```

```
# let r = new ref 1 in r#set 2; (r#get);;
- : int = 2
```

The type parameter in the declaration may actually be constrained in the body of the class definition. In the class type, the actual value of the type parameter is displayed in the `constraint` clause.

```
# class ['a] ref_succ (x_init:'a) =
#   object
#     val mutable x = x_init + 1
#     method get = x
#     method set y = x <- y
#   end;;
class ['a] ref_succ :
  'a ->
  object
    constraint 'a = int
    val mutable x : int
    method get : int
    method set : int -> unit
  end
```

Let us consider a more realistic example. We put an additional type constraint in method `move`, since no free variables must remain uncaptured by a type parameter.

```
# class ['a] circle (c : 'a) =
#   object
#     val mutable center = c
#     method center = center
#     method set_center c = center <- c
#     method move = (center#move : int -> unit)
#   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = < move : int -> unit; .. >
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

An alternate definition of `circle`, using a `constraint` clause in the class definition, is shown below. The type `#point` used below in the `constraint` clause is an abbreviation produced by the definition of class `point`. This abbreviation unifies with the type of any object belonging to a subclass of class `point`. It actually expands to `< get_x : int; move : int -> unit; .. >`. This leads to the following alternate definition of `circle`, which has slightly stronger constraints on its argument, as we now expect `center` to have a method `get_x`.

```
# class ['a] circle (c : 'a) =
#   object
#     constraint 'a = #point
#     val mutable center = c
#     method center = center
#     method set_center c = center <- c
#     method move = center#move
#   end;;
class ['a] circle :
  'a ->
  object
    constraint 'a = #point
    val mutable center : 'a
    method center : 'a
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

The class `colored_circle` is a specialized version of class `circle` which requires the type of the center to unify with `#colored_point`, and adds a method `color`. Note that when specializing a parameterized class, the instance of type parameter must always be explicitly given. It is again written inside `[` and `]`.

```
# class ['a] colored_circle c =
#   object
#     constraint 'a = #colored_point
#     inherit ['a] circle c
#     method color = center#color
#   end;;
class ['a] colored_circle :
  'a ->
  object
    constraint 'a = #colored_point
    val mutable center : 'a
    method center : 'a
    method color : string
    method move : int -> unit
    method set_center : 'a -> unit
  end
```

## 2.10 Using coercions

Subtyping is never implicit. There are, however, two ways to perform subtyping. The most general construction is fully explicit: both the domain and the codomain of the type coercion must be given.

We have seen that points and colored points have incompatible types. For instance, they cannot be mixed in the same list. However, a colored point can be coerced to a point, hiding its `color` method:

```
# let colored_point_to_point cp = (cp : colored_point :> point);;
val colored_point_to_point : colored_point -> point = <fun>

# let p = new point 3 and q = new colored_point 4 "blue";;
val p : point = <obj>
val q : colored_point = <obj>

# let l = [p; (colored_point_to_point q)];;
val l : point list = [<obj>; <obj>]
```

An object of type `t` can be seen as an object of type `t'` only if `t` is a subtype of `t'`. For instance, a point cannot be seen as a colored point.

```
# (p : point :> colored_point);;
Characters 0-28:
Type point = < get_offset : int; get_x : int; move : int -> unit >
is not a subtype of type
  colored_point =
    < color : string; get_offset : int; get_x : int; move : int -> unit >
```

Indeed, backward coercions are unsafe, and should be combined with a type case, possibly raising a runtime error. However, there is no such operation available in the language.

Be aware that subtyping and inheritance are not related. Inheritance is a syntactic relation between classes while subtyping is a semantic relation between types. For instance, the class of colored points could have been defined directly, without inheriting from the class of points; the type of colored points would remain unchanged and thus still be a subtype of points.

The domain of a coercion can usually be omitted. For instance, one can define:

```
# let to_point cp = (cp :> point);;
val to_point :
  < get_offset : int; get_x : int; move : int -> unit; .. > -> point = <fun>
```

In this case, the function `colored_point_to_point` is an instance of the function `to_point`. This is not always true, however. The fully explicit coercion is more precise and is sometimes unavoidable. Here is an example where the shorter form fails:

```
# class virtual c  = object method virtual m : c end;;
class virtual c : object method virtual m : c end
```

```
# class c'  =
#   object (self)
#     inherit c
#     method m = (self :> c)
#     method m' = 1
# end;;
Characters 59-63:
This expression cannot be coerced to type c = < m : c >; it has type
  < m : c; .. >
but is here used with type < m : 'a; .. > as 'a
Type c = < m : c > is not compatible with type 'a
Self type cannot be unified with a closed object type
```

The type of the coercion to type `c` can be seen here:

```
# function x -> (x :> c);;
- : (< m : 'a; .. > as 'a) -> c = <fun>
```

As class `c'` inherits from class `c`, its method `m` must have type `c`. On the other hand, in expression `(self :> c)` the type of `self` and the domain of the coercion above must be unified. That is, the type of the method `m` in `self` (i.e. `c`) is also the type of `self`. So, the type of `self` is `c`. This is a contradiction, as the type of `self` has a method `m'`, whereas type `c` does not.

The desired coercion of type `<m : c;..> -> c` can be obtained by using a fully explicit coercion:

```
# function x -> (x : #c :> c);;
- : #c -> c = <fun>
```

Thus one can define class c' as follows:

```
# class c'  =
#   object (self)
#     inherit c
#     method m = (self : #c :> c)
#     method m' = 1
#   end;;
class c' : object method m : c method m' : int end
```

An alternative is to define class `c` this way (of course this definition is not equivalent to the previous one):

```
# class virtual c = object (_ : 'a) method virtual m : 'a end;;
class virtual c : object ('a) method virtual m : 'a end
```

Then, a coercion operator is not even required.

```
# class c' = object (self) inherit c  method m = self method m' = 1 end;;
class c' : object ('a) method m : 'a method m' : int end
```

Here, the simple coercion operator `(e :> c)` can be used to coerce an object expression `e` from type `c'` to type `c`. Semi-implicit coercions are actually defined so as to work correctly with classes returning `self`.

```
# (new c' :> c);;
- : c = <obj>
```

Another common problem may occur when one tries to define a coercion to a class `c` inside the definition of class `c`. The problem is due to the type abbreviation not being completely defined yet, and so its subtypes are not clearly known. Then, a coercion `(_ : #c :> c)` is taken to be the identity function, as in

```
# function x -> (x :> 'a);;
- : 'a -> 'a = <fun>
```

As a consequence, if the coercion is applied to `self`, as in the following example, the type of `self` is unified with the closed type `c` (a closed object type is an object type without ellipsis). This would constrains the type of self be closed and is thus rejected. Indeed, the type of self cannot be closed: this would prevent any further extension of the class. Therefore, a type error is generated when the unification of this type with another type would result in a closed object type.

```
# class c = object (self) method m = (self : #c :> c) end;;
Characters 36-40:
This expression has type < m : 'a; .. > but is here used with type c = < .. >
Self type cannot escape its class
```

This problem can sometimes be avoided by first defining the abbreviation, using a class type:

```
# class type c0 = object method m : c0 end;;
class type c0 = object method m : c0 end

# class c : c0 = object (self) method m = (self : #c0 :> c0) end;;
class c : c0
```

It is also possible to use a virtual class. Inheriting from this class simultaneously allows to enforce all methods of `c` to have the same type as the methods of `c0`.

```
# class virtual c0 = object method virtual m : c0 end;;
class virtual c0 : object method virtual m : c0 end

# class c = object (self) inherit c0 method m = (self : #c0 :> c0) end;;
class c : object method m : c0 end
```

One could think of defining the type abbreviation directly:

```
# type c1 = <m : c1>;;
type c1 = < m : c1 >
```

However, the abbreviation `#c1` cannot be defined this way (the abbreviation `#c0` is defined from the class `c0`, not from the type `c0`), and should be expanded:

```
# class c = object (self)  method m = (self : <m : c1; ..> as 'a :> c1) end;;
class c : object method m : c1 end
```

## 2.11 Functional objects

It is possible to write a version of class `point` without assignments on the instance variables. The construct `{< ... >}` returns a copy of "self" (that is, the current object), possibly changing the value of some instance variables.

```
# class functional_point y =
#   object
#     val x = y
#     method get_x = x
#     method move d = {< x = x + d >}
#   end;;
class functional_point :
  int ->
  object ('a) val x : int method get_x : int method move : int -> 'a end
# let p = new functional_point 7;;
val p : functional_point = <obj>
# p#get_x;;
- : int = 7
# (p#move 3)#get_x;;
- : int = 10
# p#get_x;;
- : int = 7
```

Note that the type abbreviation `functional_point` is recursive, which can be seen in the class type of `functional_point`: the type of self is `'a` and `'a` appears inside the type of the method `move`.

The above definition of `functional_point` is not equivalent to the following:

```
# class bad_functional_point y =
#   object
#     val x = y
#     method get_x = x
#     method move d = new functional_point (x+d)
#   end;;
```

```
class bad_functional_point :
  int ->
  object
    val x : int
    method get_x : int
    method move : int -> functional_point
  end

# let p = new functional_point 7;;
val p : functional_point = <obj>

# p#get_x;;
- : int = 7

# (p#move 3)#get_x;;
- : int = 10

# p#get_x;;
- : int = 7
```

While objects of either class will behave the same, objects of their subclasses will be different. In a subclass of the latter, the method `move` will keep returning an object of the parent class. On the contrary, in a subclass of the former, the method `move` will return an object of the subclass.

Functional update is often used in conjunction with binary methods as illustrated in section 4.2.1.

## 2.12 Cloning objects

Objects can also be cloned, whether they are functional or imperative. The library function `Oo.copy` makes a shallow copy of an object. That is, it returns an object that is equal to the previous one. The instance variables have been copied but their contents are shared. Assigning a new value to an instance variable of the copy (using a method call) will not affect instance variables of the original, and conversely. A deeper assignment (for example if the instance variable if a reference cell) will of course affect both the original and the copy.

The type of `Oo.copy` is the following:

```
# Oo.copy;;
- : (< .. > as 'a) -> 'a = <fun>
```

The keyword `as` in that type binds the type variable `'a` to the object type `< .. >`. Therefore, `Oo.copy` takes an object with any methods (represented by the ellipsis), and returns an object of the same type. The type of `Oo.copy` is different from type `< .. > -> < .. >` as each ellipsis represents a different set of methods. Ellipsis actually behaves as a type variable.

```
# let p = new point 5;;
val p : point = <obj>

# let q = Oo.copy p;;
val q : point = <obj>

# q#move 7; (p#get_x, q#get_x);;
- : int * int = 5, 12
```

In fact, `Oo.copy p` will behave as `p#copy` assuming that a public method `copy` with body `{< >}` has been defined in the class of `p`.

Objects can be compared using the generic comparison functions (`=`, `<`, ...). Two objects are equal if and only if they are physically equal. In particular, an object and its copy are not equal.

```
# let q = Oo.copy p;;
val q : point = <obj>
```

```
# p = q, p = p;;
- : bool * bool = false, true
```

Cloning and override have a non empty intersection. They are interchangeable when used within an object and without overriding any field:

```
# class copy =
#   object
#     method copy = {< >}
#   end;;
class copy : object ('a) method copy : 'a end
```

```
# class copy =
#   object (self)
#     method copy = Oo.copy self
#   end;;
class copy : object ('a) method copy : 'a end
```

Only the override can be used to actually override fields, and only the `Oo.copy` primitive can be used externally.

Cloning can also be used to provide facilities.

```
# class backup =
#   object (self : 'mytype)
#     val mutable copy = None
#     method save = copy <- Some {< copy = None >}
#     method restore = match copy with Some x -> x | None -> self
#   end;;
class backup :
  object ('a)
    val mutable copy : 'a option
    method restore : 'a
    method save : unit
  end
```

The above definition will only backup one level. Backuping can be added to any class using multiple inheritance.

```
# class ['a] backup_ref x = object inherit ['a] ref x inherit backup end;;
class ['a] backup_ref :
  'a ->
  object ('b)
    val mutable copy : 'b option
    val mutable x : 'a
    method get : 'a
    method restore : 'b
    method save : unit
    method set : 'a -> unit
  end
```

```
# let rec get p n = if n = 0 then p # get else get (p # restore) (n-1);;
val get : (< get : 'b; restore : 'a; .. > as 'a) -> int -> 'b = <fun>
```

```
# let p = new backup_ref 0  in
# p # save; p # set 1; p # save; p # set 2;
# [get p 0; get p 1; get p 2; get p 3; get p 4];;
- : int list = [2; 1; 1; 1; 1]
```

A variant of backups could retain all copies. (We then add a method clear to manually erase all copies.)

```
# class backup =
#   object (self : 'mytype)
#     val mutable copy = None
#     method save = copy <- Some {< >}
#     method restore = match copy with Some x -> x | None -> self
#     method clear = copy <- None
#   end;;
class backup :
  object ('a)
    val mutable copy : 'a option
    method clear : unit
    method restore : 'a
    method save : unit
  end
```

```
# class ['a] backup_ref x = object inherit ['a] ref x inherit backup end;;
class ['a] backup_ref :
  'a ->
  object ('b)
    val mutable copy : 'b option
    val mutable x : 'a
    method clear : unit
    method get : 'a
    method restore : 'b
    method save : unit
    method set : 'a -> unit
  end
```

```
# let p = new backup_ref 0  in
# p # save; p # set 1; p # save; p # set 2;
# [get p 0; get p 1; get p 2; get p 3; get p 4];;
- : int list = [2; 1; 0; 0; 0]
```

## 2.13 Recursive classes

Recursive classes can be used to define objects whose types are mutually recursive.

```
# class window =
#   object
#     val mutable top_widget = (None : widget option)
#     method top_widget = top_widget
#   end
# and widget (w : window) =
#   object
#     val window = w
#     method window = window
#   end;;
class window :
  object
    val mutable top_widget : widget option
    method top_widget : widget option
  end
class widget :
  window -> object val window : window method window : window end
```

Although their types are mutually recursive, the classes `widget` and `window` are themselves independent.

## 2.14 Binary methods

A binary method is a method which takes an argument of the same type as self. The class `comparable` below is a template for classes with a binary method `leq` of type `'a -> bool` where the type variable `'a` is bound to the type of self. Therefore, `#comparable` expands to `< leq : 'a -> bool; .. > as 'a`. We see here that the binder `as` also allows to write recursive types.

```
# class virtual comparable =
#   object (_ : 'a)
#     method virtual leq : 'a -> bool
#   end;;
class virtual comparable : object ('a) method virtual leq : 'a -> bool end
```

We then define a subclass `money` of `comparable`. The class money simply wraps floats as comparable objects. We will extend it below with more operations. There is a type constraint on the class parameter `x` as the primitive `<=` is a polymorphic comparison function in Objective Caml. The `inherit` clause ensures that the type of objects of this class is an instance of `#comparable`.

```
# class money (x : float) =
#   object
#     inherit comparable
#     val repr = x
#     method value = repr
#     method leq p = repr <= p#value
#   end;;
class money :
  float ->
  object ('a)
    val repr : float
    method leq : 'a -> bool
    method value : float
  end
```

Note that the type `money1` is not a subtype of type `comparable`, as the self type appears in contravariant position in the type of method `leq`. Indeed, an object `m` of class `money` has a method `leq` that expects an argument of type `money` since it accesses its `value` method. Considering `m` of type `comparable` would allow to call method `leq` on `m` with an argument that does not have a method `value`, which would be an error.

Objects of class `money2` below also print the value they hold. Similarly the type `money2` is not a subtype of type `money`.

```
# class money2 x =
#   object
#     inherit money x
#     method times k = {< repr = k *. repr >}
#   end;;
class money2 :
  float ->
  object ('a)
    val repr : float
    method leq : 'a -> bool
    method times : float -> 'a
    method value : float
  end
```

It is however possible to define functions that manipulate objects of type either `money` or `money2`: the function `min` will return the minimum of any two objects whose type unifies with `#comparable`. The type of `min` is not the same as `#comparable -> #comparable -> #comparable`, as the abbreviation `#comparable` hides a type variable (an ellipsis). Each occurrence of this abbreviation generates a new variable.

```
# let min (x : #comparable) y =
#   if x#leq y then x else y;;
val min : (#comparable as 'a) -> 'a -> 'a = <fun>
```

This function can be applied to objects of type `money` or `money2`.

```
# (min (new money  1.3) (new money 3.1))#value;;
- : float = 1.3

# (min (new money2 5.0) (new money2 3.14))#value;;
- : float = 3.14
```

More examples of binary methods can be found in sections 4.2.1 and 4.2.3.

Notice the use of functional update for method `times`. Writing `new money2 (k *. repr) >}` instead of `{< repr = k *. repr >}` would not behave well with inheritance: in a subclass `money3` of `money2` the `times` method would return an object of class `money2` but not of class `money3` as would be expected.

The class `money` could naturally carry another binary method. Here is a direct definition:

```
# class money x =
#   object (self : 'a)
#     val repr = x
#     method value = repr
#     method print = print_float repr
#     method times k = {< repr = k *. x >}
#     method leq (p : 'a) = repr <= p#value
#     method plus (p : 'a) = {< repr = x +. p#value >}
#   end;;
class money :
  float ->
  object ('a)
    val repr : float
    method leq : 'a -> bool
    method plus : 'a -> 'a
    method print : unit
    method times : float -> 'a
    method value : float
  end
```

## 2.15 Friends

The above class `money` reveals a problem that often occurs with binary methods. In order to interact with other objects of the same class, the representation of `money` objects must be revealed, using a method such as `value`. If we remove all binary methods (here `plus` and `leq`), the representation can easily be hidden inside objects by removing the method `value` as well. However, this is not possible as long as some binary requires access to the representation on object of the same class but different from self.

```
# class safe_money x =
#   object (self : 'a)
#     val repr = x
#     method print = print_float repr
#     method times k = {< repr = k *. x >}
#   end;;
class safe_money :
  float ->
  object ('a)
    val repr : float
    method print : unit
    method times : float -> 'a
  end
```

Here, the representation of the object is known only to a particular object. To make it available to other objects of the same class, we are forced to make it available to the whole world. However we can easily restrict the visibility of the representation using the module system.

```
# module type MONEY =
#   sig
#     type t
#     class c : float ->
#       object ('a)
#         val repr : t
#         method value : t
#         method print : unit
#         method times : float -> 'a
#         method leq : 'a -> bool
#         method plus : 'a -> 'a
#       end
#   end;;
```

```
# module Euro : MONEY =
#   struct
#     type t = float
#     class c x =
#       object (self : 'a)
#         val repr = x
#         method value = repr
#         method print = print_float repr
#         method times k = {< repr = k *. x >}
#         method leq (p : 'a) = repr <= p#value
#         method plus (p : 'a) = {< repr = x +. p#value >}
#       end
#   end;;
```

Another example of friend functions may be found in section 4.2.3. These examples occur when a group of objects (here objects of the same class) and functions should see each others internal representation, while their representation should be hidden from the outside. The solution is always to define all friends in the same module, give access to the representation and use a signature constraint to make the representation abstract outside of the module.

# Chapter 3

# The module system

This chapter introduces the module system of Objective Caml.

## 3.1 Structures

A primary motivation for modules is to package together related definitions (such as the definitions of a data type and associated operations over that type) and enforce a consistent naming scheme for these definitions. This avoids running out of names or accidentally confusing names. Such a package is called a *structure* and is introduced by the `struct`...`end` construct, which contains an arbitrary sequence of definitions. The structure is usually given a name with the `module` binding. Here is for instance a structure packaging together a type of priority queues and their operations:

```
# module PrioQueue =
#   struct
#     type priority = int
#     type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
#     let empty = Empty
#     let rec insert queue prio elt =
#       match queue with
#         Empty -> Node(prio, elt, Empty, Empty)
#       | Node(p, e, left, right) ->
#           if prio <= p
#           then Node(prio, elt, insert right p e, left)
#           else Node(p, e, insert right prio elt, left)
#     exception Queue_is_empty
#     let rec remove_top = function
#         Empty -> raise Queue_is_empty
#       | Node(prio, elt, left, Empty) -> left
#       | Node(prio, elt, Empty, right) -> right
#       | Node(prio, elt, (Node(lprio, lelt, _, _) as left),
#                         (Node(rprio, relt, _, _) as right)) ->
#           if lprio <= rprio
#           then Node(lprio, lelt, remove_top left, right)
#           else Node(rprio, relt, left, remove_top right)
#     let extract = function
#         Empty -> raise Queue_is_empty
#       | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
#   end;;
module PrioQueue :
  sig
    type priority = int
    and 'a queue = | Empty | Node of priority * 'a * 'a queue * 'a queue
    val empty : 'a queue
    val insert : 'a queue -> priority -> 'a -> 'a queue
    exception Queue_is_empty
    val remove_top : 'a queue -> 'a queue
    val extract : 'a queue -> priority * 'a * 'a queue
  end
```

Outside the structure, its components can be referred to using the "dot notation", that is, identifiers qualified by a structure name. For instance, `PrioQueue.insert` in a value context is the function `insert` defined inside the structure `PrioQueue`. Similarly, `PrioQueue.queue` in a type context is the type `queue` defined in `PrioQueue`.

```
# PrioQueue.insert PrioQueue.empty 1 "hello";;
- : string PrioQueue.queue =
PrioQueue.Node (1, "hello", PrioQueue.Empty, PrioQueue.Empty)
```

## 3.2 Signatures

Signatures are interfaces for structures. A signature specifies which components of a structure are accessible from the outside, and with which type. It can be used to hide some components of a structure (e.g. local function definitions) or export some components with a restricted type. For instance, the signature below specifies the three priority queue operations `empty`, `insert` and `extract`, but not the auxiliary function `remove_top`. Similarly, it makes the `queue` type abstract (by not providing its actual representation as a concrete type).

```
# module type PRIOQUEUE =
#   sig
#     type priority = int         (* still concrete *)
#     type 'a queue               (* now abstract *)
#     val empty : 'a queue
#     val insert : 'a queue -> int -> 'a -> 'a queue
#     val extract : 'a queue -> int * 'a * 'a queue
#     exception Queue_is_empty
#   end;;
module type PRIOQUEUE =
  sig
    type priority = int
    and 'a queue
    val empty : 'a queue
    val insert : 'a queue -> int -> 'a -> 'a queue
    val extract : 'a queue -> int * 'a * 'a queue
    exception Queue_is_empty
  end
```

Restricting the `PrioQueue` structure by this signature results in another view of the `PrioQueue` structure where the `remove_top` function is not accessible and the actual representation of priority queues is hidden:

```
# module AbstractPrioQueue = (PrioQueue : PRIOQUEUE);;
module AbstractPrioQueue : PRIOQUEUE

# AbstractPrioQueue.remove_top;;
Characters 0-28:
Unbound value AbstractPrioQueue.remove_top

# AbstractPrioQueue.insert AbstractPrioQueue.empty 1 "hello";;
- : string AbstractPrioQueue.queue = <abstr>
```

The restriction can also be performed during the definition of the structure, as in

```
module PrioQueue = (struct ... end : PRIOQUEUE);;
```

An alternate syntax is provided for the above:

```
module PrioQueue : PRIOQUEUE = struct ... end;;
```

## 3.3 Functors

Functors are "functions" from structures to structures. They are used to express parameterized structures: a structure $A$ parameterized by a structure $B$ is simply a functor $F$ with a formal parameter $B$ (along with the expected signature for $B$) which returns the actual structure $A$ itself. The functor $F$ can then be applied to one or several implementations $B_1 \dots B_n$ of $B$, yielding the corresponding structures $A_1 \dots A_n$.

For instance, here is a structure implementing sets as sorted lists, parameterized by a structure providing the type of the set elements and an ordering function over this type (used to keep the sets sorted):

```
# type comparison = Less | Equal | Greater;;
type comparison = | Less | Equal | Greater

# module type ORDERED_TYPE =
#   sig
#     type t
#     val cmp: t -> t -> comparison
#   end;;
module type ORDERED_TYPE = sig type t val cmp : t -> t -> comparison end

# module Set =
#   functor (Elt: ORDERED_TYPE) ->
#     struct
#       type element = Elt.t
#       type set = element list
#       let empty = []
#       let rec add x s =
#         match s with
#           [] -> [x]
#         | hd::tl ->
#            match Elt.cmp x hd with
#              Equal   -> s         (* x is already in s *)
#            | Less    -> x :: s    (* x is smaller than all elements of s *)
#            | Greater -> hd :: add x tl
#       let rec member x s =
#         match s with
#           [] -> false
#         | hd::tl ->
#             match Elt.cmp x hd with
#               Equal   -> true     (* x belongs to s *)
#             | Less    -> false    (* x is smaller than all elements of s *)
#             | Greater -> member x tl
#     end;;
module Set :
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      and set = element list
      val empty : 'a list
      val add : Elt.t -> Elt.t list -> Elt.t list
      val member : Elt.t -> Elt.t list -> bool
    end
```

By applying the Set functor to a structure implementing an ordered type, we obtain set operations for this type:

```
# module OrderedString =
#   struct
#     type t = string
#     let cmp x y = if x = y then Equal else if x < y then Less else Greater
#   end;;
module OrderedString :
  sig type t = string val cmp : 'a -> 'a -> comparison end
# module StringSet = Set(OrderedString);;
module StringSet :
  sig
    type element = OrderedString.t
    and set = element list
    val empty : 'a list
    val add : OrderedString.t -> OrderedString.t list -> OrderedString.t list
    val member : OrderedString.t -> OrderedString.t list -> bool
  end
# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
```

```
- : bool = false
```

## 3.4 Functors and type abstraction

As in the `PrioQueue` example, it would be good style to hide the actual implementation of the type `set`, so that users of the structure will not rely on sets being lists, and we can switch later to another, more efficient representation of sets without breaking their code. This can be achieved by restricting `Set` by a suitable functor signature:

```
# module type SETFUNCTOR =
#   functor (Elt: ORDERED_TYPE) ->
#     sig
#       type element = Elt.t      (* concrete *)
#       type set                  (* abstract *)
#       val empty : set
#       val add : element -> set -> set
#       val member : element -> set -> bool
#     end;;
module type SETFUNCTOR =
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      and set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end

# module AbstractSet = (Set : SETFUNCTOR);;
module AbstractSet : SETFUNCTOR

# module AbstractStringSet = AbstractSet(OrderedString);;
module AbstractStringSet :
  sig
    type element = OrderedString.t
    and set = AbstractSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# AbstractStringSet.add "gee" AbstractStringSet.empty;;
- : AbstractStringSet.set = <abstr>
```

In an attempt to write the type constraint above more elegantly, one may wish to name the signature of the structure returned by the functor, then use that signature in the constraint:

```
# module type SET =
#   sig
#     type element
#     type set
#     val empty : set
#     val add : element -> set -> set
#     val member : element -> set -> bool
#   end;;
module type SET =
  sig
    type element
    and set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end
```

```
# module WrongSet = (Set : functor(Elt: ORDERED_TYPE) -> SET);;
module WrongSet : functor(Elt : ORDERED_TYPE) -> SET
```

```
# module WrongStringSet = WrongSet(OrderedString);;
module WrongStringSet :
  sig
    type element = WrongSet(OrderedString).element
    and set = WrongSet(OrderedString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end
```

```
# WrongStringSet.add "gee" WrongStringSet.empty;;
Characters 19-24:
This expression has type string but is here used with type
  WrongStringSet.element = WrongSet(OrderedString).element
```

The problem here is that `SET` specifies the type `element` abstractly, so that the type equality between `element` in the result of the functor and `t` in its argument is forgotten. Consequently, `WrongStringSet.element` is not the same type as `string`, and the operations of `WrongStringSet` cannot be applied to strings. As demonstrated above, it is important that the type `element` in the signature `SET` be declared equal to `Elt.t`; unfortunately, this is impossible above since `SET` is defined in a context where `Elt` does not exist. To overcome this difficulty, Objective Caml provides a `with type` construct over signatures that allows to enrich a signature with extra type equalities:

```
# module AbstractSet =
#   (Set : functor(Elt: ORDERED_TYPE) -> (SET with type element = Elt.t));;
module AbstractSet :
  functor(Elt : ORDERED_TYPE) ->
    sig
      type element = Elt.t
      and set
      val empty : set
      val add : element -> set -> set
      val member : element -> set -> bool
    end
```

As in the case of simple structures, an alternate syntax is provided for defining functors and restricting their result:

```
module AbstractSet(Elt: ORDERED_TYPE) : (SET with type element = Elt.t) =
  struct ... end;;
```

Abstracting a type component in a functor result is a powerful technique that provides a high degree of type safety, as we now illustrate. Consider an ordering over character strings that is different from the standard ordering implemented in the `OrderedString` structure. For instance, we compare strings without distinguishing upper and lower case.

```
# module NoCaseString =
#   struct
#     type t = string
#     let cmp s1 s2 =
#       OrderedString.cmp (String.lowercase s1) (String.lowercase s2)
#   end;;
module NoCaseString :
  sig type t = string val cmp : string -> string -> comparison end

# module NoCaseStringSet = AbstractSet(NoCaseString);;
module NoCaseStringSet :
  sig
    type element = NoCaseString.t
    and set = AbstractSet(NoCaseString).set
    val empty : set
    val add : element -> set -> set
    val member : element -> set -> bool
  end

# NoCaseStringSet.add "FOO" AbstractStringSet.empty;;
Characters 26-49:
This expression has type
  AbstractStringSet.set = AbstractSet(OrderedString).set
but is here used with type
  NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Notice that the two types `AbstractStringSet.set` and `NoCaseStringSet.set` are not compatible, and values of these two types do not match. This is the correct behavior: even though both set types contain elements of the same type (strings), both are built upon different orderings of that type, and different invariants need to be maintained by the operations (being strictly increasing for the standard ordering and for the case-insensitive ordering). Applying operations from `AbstractStringSet` to values of type `NoCaseStringSet.set` could give incorrect results, or build lists that violate the invariants of `NoCaseStringSet`.

## 3.5 Modules and separate compilation

All examples of modules so far have been given in the context of the interactive system. However, modules are most useful for large, batch-compiled programs. For these programs, it is a practical necessity to split the source into several files, called compilation units, that can be compiled separately, thus minimizing recompilation after changes.

In Objective Caml, compilation units are special cases of structures and signatures, and the relationship between the units can be explained easily in terms of the module system. A compilation unit *a* comprises two files:

- the implementation file *a*`.ml`, which contains a sequence of definitions, analogous to the inside of a `struct`...`end` construct;
- the interface file *a*`.mli`, which contains a sequence of specifications, analogous to the inside of a `sig`...`end` construct.

Both files define a structure named *A* (same name as the base name *a* of the two files, with the first letter capitalized), as if the following definition was entered at top-level:

```
module A: sig (* contents of file a.mli *) end
        = struct (* contents of file a.ml *) end;;
```

The files defining the compilation units can be compiled separately using the `ocaml -c` command (the `-c` option means “compile only, do not try to link”); this produces compiled interface files (with extension `.cmi`) and compiled object code files (with extension `.cmo`). When all units have been compiled, their `.cmo` files are linked together using the `ocaml` command. For instance, the following commands compile and link a program composed of two compilation units `aux` and `main`:

```
$ ocamlc -c aux.mli                     # produces aux.cmi
$ ocamlc -c aux.ml                      # produces aux.cmo
$ ocamlc -c main.mli                    # produces main.cmi
$ ocamlc -c main.ml                     # produces main.cmo
$ ocamlc -o theprogram aux.cmo main.cmo
```

The program behaves exactly as if the following phrases were entered at top-level:

```
module Aux: sig (* contents of aux.mli *) end
          = struct (* contents of aux.ml *) end;;
module Main: sig (* contents of main.mli *) end
           = struct (* contents of main.ml *) end;;
```

In particular, `Main` can refer to `Aux`: the definitions and declarations contained in `main.ml` and `main.mli` can refer to definition in `aux.ml`, using the `Aux.`*ident* notation, provided these definitions are exported in `aux.mli`.

The order in which the `.cmo` files are given to `ocaml` during the linking phase determines the order in which the module definitions occur. Hence, in the example above, `Aux` appears first and `Main` can refer to it, but `Aux` cannot refer to `Main`.

Notice that only top-level structures can be mapped to separately-compiled files, but not functors nor module types. However, all module-class objects can appear as components of a structure, so the solution is to put the functor or module type inside a structure, which can then be mapped to a file.

# Chapter 4

# Advanced examples with classes and modules

*(Chapter written by Didier Rémy)*

In this chapter, we show some larger examples using objects, classes and modules. We review many of the object features simultaneously on the example of a bank account. We show how modules taken from the standard library can be expressed as classes. Lastly, we describe a programming pattern know of as *virtual types* through the example of window managers.

## 4.1 Extended example: bank accounts

In this section, we illustrate most aspects of Object and inheritance by refining, debugging, and specializing the following initial naive definition of a simple bank account. (We reuse the module `Euro` defined at the end of chapter 2.)

```
# let euro = new Euro.c;;
```

```
# let zero = euro 0.;;
```

```
# let neg x = x#times (-1.);;
```

```
# class account =
#   object
#     val mutable balance = zero
#     method balance = balance
#     method deposit x = balance <- balance # plus x
#     method withdraw x =
#       if x#leq balance then (balance <- balance # plus (neg x); x) else zero
#   end;;
```

```
# let c = new account in c # deposit (euro 100.); c # withdraw (euro 50.);;
```

We now refine this definition with a method to compute interest.

```
# class account_with_interests =
#   object (self)
#     inherit account
#     method private interest = self # deposit (self # balance # times 0.03)
#   end;;
```

We make the method `interest` private, since clearly it should not be called freely from the outside. Here, it is only made accessible to subclasses that will manage monthly or yearly updates of the account.

We should soon fix a bug in the current definition: the deposit method can be used for withdrawing money by depositing negative amounts. We can fix this directly:

```
# class safe_account =
#   object
#     inherit account
#     method deposit x = if zero#leq x then balance <- balance#plus x
#   end;;
```

However, the bug might be fixed more safely by the following definition:

```
# class safe_account =
#   object
#     inherit account as unsafe
#     method deposit x =
#       if zero#leq x then unsafe # deposit x
#       else raise (Invalid_argument "deposit")
#   end;;
```

In particular, this does not require the knowledge of the implementation of the method `deposit`.

To keep trace of operations, we extend the class with a mutable field `history` and a private method `trace` to add an operation in the log. Then each method to be traced is redefined.

```
# type 'a operation = Deposit of 'a | Retrieval of 'a;;
```

```
# class account_with_history =
#   object (self)
#     inherit safe_account as super
#     val mutable history = []
#     method private trace x = history <- x :: history
#     method deposit x = self#trace (Deposit x);  super#deposit x
#     method withdraw x = self#trace (Retrieval x); super#withdraw x
#     method history = List.rev history
#   end;;
```

One may wish to open an account and simultaneously deposit some initial amount. Although the initial implementation did not address this requirement, it can be achieved by using an initializer.

```
# class account_with_deposit x =
#   object
#     inherit account_with_history
#     initializer balance <- x
#   end;;
```

A better alternative is:

```
# class account_with_deposit x =
#   object (self)
#     inherit account_with_history
#     initializer self#deposit x
#   end;;
```

Indeed, the latter is safer since the call to `deposit` will automatically benefit from safety checks and from the trace. Let's test it:

```
# let ccp = new account_with_deposit (euro 100.) in
# let balance = ccp#withdraw (euro 50.) in
# ccp#history;;
```

Closing an account can be done with the following polymorphic function:

```
# let close c = c#withdraw (c#balance);;
```

Of course, this applies to all sorts of accounts.

Finally, we gather several versions of the account into a module `Account` abstracted over some currency.

```
# let today () = (01,01,2000) (* an approximation *)
# module Account (M:MONEY) =
#   struct
#     type m = M.c
#     let m = new M.c
#     let zero = m 0.
#
#     class bank =
#       object (self)
#         val mutable balance = zero
#         method balance = balance
#         val mutable history = []
#         method private trace x = history <- x::history
#         method deposit x =
#           self#trace (Deposit x);
#           if zero#leq x then balance <- balance # plus x
#           else raise (Invalid_argument "deposit")
#         method withdraw x =
#           if x#leq balance then
#             (balance <- balance # plus (neg x); self#trace (Retrieval x); x)
#           else zero
#         method history = List.rev history
#       end
#
#     class type client_view =
#       object
#         method deposit : m -> unit
#         method history : m operation list
#         method withdraw : m -> m
#         method balance : m
#       end
#
#     class virtual check_client x =
#       let y = if (m 100.)#leq x then x
#       else raise (Failure "Insufficient initial deposit") in
#       object (self) initializer self#deposit y end
#
#     module Client (B : sig class bank : client_view end) =
#       struct
#         class account x : client_view =
#           object
#             inherit B.bank
#             inherit check_client x
#           end
#
#         let discount x =
#           let c = new account x in
#           if today() < (1998,10,30) then c # deposit (m 100.); c
#       end
#   end;;
```

This shows the use of modules to group several class definitions that can in fact be thought of as a single unit. This unit would be provided by a bank for both internal and external uses. This is implemented as a functor that abstracts over the currency so that the same code can be used to provide accounts in different currencies.

The class `bank` is the *real* implementation of the bank account (it could have been inlined). This is the one that will be used for further extensions, refinements, etc. Conversely, the client will only be given the client view.

```
# module Euro_account = Account(Euro);;
```

```
# module Client = Euro_account.Client (Euro_account);;
```

```
# new Client.account (new Euro.c 100.);;
```

Hence, the clients do not have direct access to the `balance`, nor the `history` of their own accounts. Their only way to change their balance is to deposit or withdraw money. It is important to give the clients a class and not just the ability to create accounts (such as the promotional `discount` account), so that they can personalize their account. For instance, a client may refine the `deposit` and withdraw methods so as to do his own financial bookkeeping, automatically. On the other hand, the method discount is given as such, with no possibility for further personalization.

It is important that to provide the client's view as a functor Client so that client accounts can still be build after a possible specialization of the bank. The functor Client may remain unchanged and be passed the new definition to initialize a client's view of the extended account.

```
# module Investment_account (M : MONEY) =
#   struct
#     type m = M.c
#     module A = Account(M)
#
#     class bank =
#       object
#         inherit A.bank as super
#         method deposit x =
#           if (new M.c 1000.)#leq x then
#             print_string "Would you like to invest?";
#           super#deposit x
#       end
#
#     module Client = A.Client
#   end;;
```

The functor Client may also be redefined when some new features of the account can be given to the client.

```
# module Internet_account (M : MONEY) =
#   struct
#     type m = M.c
#     module A = Account(M)
#
#     class bank =
#       object
#         inherit A.bank
#         method mail s = print_string s
#       end
#
#     class type client_view =
#       object
#         method deposit : m -> unit
#         method history : m operation list
#         method withdraw : m -> m
#         method balance : m
#         method mail : string -> unit
#       end
#
```

```
#     module Client (B : sig class bank : client_view end) =
#       struct
#         class account x : client_view =
#           object
#             inherit B.bank
#             inherit A.check_client x
#           end
#       end
#   end;;
```

## 4.2 Simple modules as classes

One may wonder whether it is possible to treat primitive types such as integers and strings as objects. Although this is usually uninteresting for integers or strings, there may be some situations where this is desirable. The class `money` above is such an example. We show here how to do it for strings.

### 4.2.1 Strings

A naive definition of strings as objects could be:

```
# class ostring s =
#   object
#      method get n = String.get n
#      method set n c = String.set n c
#      method print = print_string s
#      method copy = new ostring (String.copy s)
#   end;;
```

However, the method `copy` returns an object of the class `string`, and not an objet of the current class. Hence, if the class is further extended, the method `copy` will only return an object of the parent class.

```
# class sub_string s =
#   object
#      inherit ostring s
#      method sub start len = new sub_string (String.sub s  start len)
#   end;;
```

As seen in section 2.14, the solution is to use functional update instead. We need to create an instance variable containing the representation `s` of the string.

```
# class better_string s =
#   object
#      val repr = s
#      method get n = String.get n
#       method set n c = String.set n c
#       method print = print_string repr
#       method copy = {< repr = String.copy repr >}
#       method sub start len = {< repr = String.sub s  start len >}
#    end;;
```

As shown in the inferred type, the methods `copy` and `sub` now return objects of the same type as the one of the class.

Another difficulty is the implementation of the method `concat`. In order to concatenate a string with another string of the same class, one must be able to access the instance variable externally. Thus, a method `repr` returning s must be defined. Here is the correct definition of strings:

```
# class ostring s =
#    object (self : 'mytype)
#       val repr = s
#       method repr = repr
#       method get n = String.get n
#       method set n c = String.set n c
#       method print = print_string repr
#       method copy = {< repr = String.copy repr >}
#       method sub start len = {< repr = String.sub s start len >}
#       method concat (t : 'mytype) = {< repr = repr ^ t#repr >}
#    end;;
```

Another constructor of the class string can be defined to return an uninitialized string of a given length:

```
# class cstring n = ostring (String.create n);;
```

Here, exposing the representation of strings is probably harmless. We do could also hide the representation of strings as we hid the currency in the class `money` of section 2.15.

## Stacks

There is sometimes an alternative between using modules or classes for parametric data types. Indeed, there are situations when the two approaches are quite similar. For instance, a stack can be straightforwardly implemented as a class:

```
# exception Empty;;

# class ['a] stack =
#    object
#      val mutable l = ([] : 'a list)
#      method push x = l <- x::l
#      method pop = match l with [] -> raise Empty | a::l' -> l <- l'; a
#      method clear = l <- []
#      method length = List.length l
#    end;;
```

However, writing a method for iterating over a stack is more problematic. A method `fold` would have type `('b -> 'a -> 'b) -> 'b -> 'b`. Here `'a` is the parameter of the stack. The parameter `'b` is not related to the class `'a stack` but to the argument that will be passed to the method `fold`. The intuition is that method `fold` should be polymorphic, i.e. of type `All ('a) ('b -> 'a -> 'b) -> 'b -> 'b`, which is not currently possible. One possibility would be to make `'b` an extra parameter of class `stack`:

```
# class ['a, 'b] stack2 =
#   object
#     inherit ['a] stack
#     method fold f (x : 'b) = List.fold_left f x l
#   end;;
```

However, the method `fold` of a given object can only be applied to functions that all have the same type:

```
# let s = new stack2;;
```

```
# s#fold (+) 0;;
```

```
# s;;
```

The best solution would be to make method `fold` polymorphic. However, OCaml does not currently allow methods to be polymorphic. Thus, the current solution is to leave the function `fold` outside of the class.

```
# class ['a] stack3 =
#   object
#     inherit ['a] stack
#     method iter f = List.iter (f : 'a -> unit) l
#   end;;
```

```
# let stack_fold (s : 'a #stack3) f x =
#   let accu = ref x in
#   s#iter (fun e -> accu := f !accu e);
#   !accu;;
```

### 4.2.2 Hashtbl

A simplified version of object-oriented hash tables should have the following class type.

```
# class type ['a, 'b] hash_table =
#   object
#     method find : 'a -> 'b
#     method add : 'a -> 'b -> unit
#   end;;
```

A simple implementation, which is quite reasonable for small hastables is to use an association list:

```
# class ['a, 'b] small_hashtbl : ['a, 'b] hash_table =
#   object
#     val mutable table = []
#     method find key = List.assoc key table
#     method add key valeur = table <- (key, valeur) :: table
#   end;;
```

A better implementation, and one that scales up better, is to use a true hash tables… whose elements are small hash tables!

```
# class ['a, 'b] hashtbl size : ['a, 'b] hash_table =
#   object (self)
#     val table = Array.init size (fun i -> new small_hashtbl)
#     method private hash key =
#       (Hashtbl.hash key) mod (Array.length table)
#     method find key = table.(self#hash key) # find key
#     method add key = table.(self#hash key) # add key
#   end;;
```

### 4.2.3 Sets

Implementing sets leads to another difficulty. Indeed, the method `union` needs to be able to access the internal representation of another object of the same class.

This is another instance of friend functions as seen in section 2.15. Indeed, this is the same mechanism used in the module `Set` in the absence of objects.

In the object-oriented version of sets, we only need to add an additional method `tag` to return the representation of a set. Since `set` are parametric. The method `tag` has a parametric type `'a tag`, concrete within the module definition but abstract in its signature. From outside, it will them be guaranteed that two objects with a method `tag` of the same type will share the same representation.

```
# module type SET =
#   sig
#     type 'a tag
#     class ['a] c :
#       object ('b)
#         method is_empty : bool
#         method mem : 'a -> bool
#         method add : 'a -> 'b
#         method union : 'b -> 'b
#         method iter : ('a -> unit) -> unit
#         method tag : 'a tag
#       end
#   end;;
```

```
# module Set : SET =
#   struct
#     let rec merge l1 l2 =
#       match l1 with
#         [] -> l2
#       | h1 :: t1 ->
#           match l2 with
#             [] -> l1
#           | h2 :: t2 ->
#               if h1 < h2 then h1 :: merge t1 l2
#               else if h1 > h2 then h2 :: merge l1 t2
#               else merge t1 l2
#     type 'a tag = 'a list
#     class ['a] c =
#       object (_ : 'b)
#         val repr = ([] : 'a list)
#         method is_empty = (repr = [])
#         method mem x = List.exists ((=) x) repr
#         method add x = {< repr = merge [x] repr >}
#         method union (s : 'b) = {< repr = merge repr s#tag >}
#         method iter (f : 'a -> unit) = List.iter f repr
#         method tag = repr
#       end
#   end;;
```

## 4.3 The subject/observer pattern

The following example, known as the subject/observer pattern, is often presented in the literature as a difficult inheritance problem with inter-connected classes. The general pattern amounts to the definition a pair of two classes that recursively interact with one another.

The class observer has a distinguished method notify that requires two arguments, a subject and an event to execute an action.

```
# class virtual ['subject, 'event] observer =
#   object
#     method virtual notify : 'subject ->  'event -> unit
#   end;;
```

The class subject remembers a list of observers in an instance variable, and has a distinguished method notify_observers to broadcast the message notify to all observers with a particular event e.

```
# class ['observer, 'event] subject =
#   object (self)
#     val mutable observers = ([]:'observer list)
#     method add_observer obs = observers <- (obs :: observers)
#     method notify_observers (e : 'event) =
#         List.iter (fun x -> x#notify self e) observers
#   end;;
```

The difficulty usually relies in defining instances of the pattern above by inheritance. This can be done in a natural and obvious manner in Ocaml, as shown on the following example manipulating windows.

```
# type event = Raise | Resize | Move;;

# let string_of_event = function
#     Raise -> "Raise" | Resize -> "Resize" | Move -> "Move";;

# let count = ref 0;;

# class ['observer] window_subject =
#   let id = count := succ !count; !count in
#   object (self)
#     inherit ['observer, event] subject
#     val mutable position = 0
#     method identity = id
#     method move x = position <- position + x; self#notify_observers Move
#     method draw = Printf.printf "{Position = %d}\n"  position;
#   end;;

# class ['subject] window_observer =
#   object
#     inherit ['subject, event] observer
#     method notify s e = s#draw
#   end;;
```

Unsurprisingly the type of `window` is recursive.

```
# let window = new window_subject;;
```

However, the two classes of `window_subject` and `window_observer` are not mutually recursive.

```
# let window_observer = new window_observer;;

# window#add_observer window_observer;;

# window#move 1;;
```

Classes `window_observer` and `window_subject` can still be extended by inheritance. For instance, one may enrich the `subject` with new behaviors and refined the behavior of the observer.

```
# class ['observer] richer_window_subject =
#   object (self)
#     inherit ['observer] window_subject
#     val mutable size = 1
#     method resize x = size <- size + x; self#notify_observers Resize
#     val mutable top = false
#     method raise = top <- true; self#notify_observers Raise
#     method draw = Printf.printf "{Position = %d; Size = %d}\n"  position size;
#   end;;
```

```
# class ['subject] richer_window_observer =
#   object
#     inherit ['subject] window_observer as super
#     method notify s e = if e <> Raise then s#raise; super#notify s e
#   end;;
```

We can also create a different kind of observer:

```
# class ['subject] trace_observer =
#   object
#     inherit ['subject, event] observer
#     method notify s e =
#       Printf.printf
#         "<Window %d <== %s>\n" s#identity (string_of_event e)
#   end;;
```

and combine them as follows:

```
# let window = new richer_window_subject;;
```

```
# window#add_observer (new richer_window_observer);;
```

```
# window#add_observer (new trace_observer);;
```

```
# window#move 1; window#resize 2;;
```

# Part II

# The Objective Caml language

# Chapter 5

# The Objective Caml language

## Foreword

This document is intended as a reference manual for the Objective Caml language. It lists the language constructs, and gives their precise syntax and informal semantics. It is by no means a tutorial introduction to the language: there is not a single example. A good working knowledge of Caml is assumed.

No attempt has been made at mathematical rigor: words are employed with their intuitive meaning, without further definition. As a consequence, the typing rules have been left out, by lack of the mathematical framework required to express them, while they are definitely part of a full formal definition of the language.

### Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (`like this`). Non-terminal symbols are set in italic font (*like that*). Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components. Curly bracket with a trailing plus sign $\{\ldots\}^+$ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

## 5.1 Lexical conventions

### Blanks

The following characters are considered as blanks: space, newline, horizontal tabulation, carriage return, line feed and form feed. Blanks are ignored, but they separate adjacent identifiers, literals and keywords that would otherwise be confused as one single identifier, literal or keyword.

### Comments

Comments are introduced by the two characters `(*`, with no intervening blanks, and terminated by the characters `*)`, with no intervening blanks. Comments are treated as blank characters. Comments do not occur inside string or character literals. Nested comments are handled correctly.

### Identifiers

$$\begin{array}{rcl} \textit{ident} & ::= & (\textit{letter} \mid \texttt{\_})\ \{\textit{letter} \mid \texttt{0}\ldots\texttt{9} \mid \texttt{\_} \mid \texttt{'}\} \\ \textit{letter} & ::= & \texttt{A}\ldots\texttt{Z} \mid \texttt{a}\ldots\texttt{z} \end{array}$$

Identifiers are sequences of letters, digits, `_` (the underscore character), and `'` (the single quote), starting with a letter or an underscore. Letters contain at least the 52 lowercase and uppercase letters from the ASCII set. The current implementation also recognizes as letters all accented characters from the ISO 8859-1 ("ISO Latin 1") set, and also allows an underscore `_` as the first character of an identifier. All characters in an identifier are meaningful. The current implementation places no limits on the number of characters of an identifier.

### Integer literals

$$\begin{array}{rcl} \textit{integer-literal} & ::= & [\texttt{-}]\ \{\texttt{0}\ldots\texttt{9}\}^{+} \\ & \mid & [\texttt{-}]\ (\texttt{0x} \mid \texttt{0X})\ \{\texttt{0}\ldots\texttt{9} \mid \texttt{A}\ldots\texttt{F} \mid \texttt{a}\ldots\texttt{f}\}^{+} \\ & \mid & [\texttt{-}]\ (\texttt{0o} \mid \texttt{0O})\ \{\texttt{0}\ldots\texttt{7}\}^{+} \\ & \mid & [\texttt{-}]\ (\texttt{0b} \mid \texttt{0B})\ \{\texttt{0}\ldots\texttt{1}\}^{+} \end{array}$$

An integer literal is a sequence of one or more digits, optionally preceded by a minus sign. By default, integer literals are in decimal (radix 10). The following prefixes select a different radix:

| Prefix | Radix |
|---|---|
| `0x`, `0X` | hexadecimal (radix 16) |
| `0o`, `0O` | octal (radix 8) |
| `0b`, `0B` | binary (radix 2) |

(The initial `0` is the digit zero; the `O` for octal is the letter O.) The interpretation of integer literals that fall outside the range of representable integer values is undefined.

### Floating-point literals

$$\begin{array}{rcl} \textit{float-literal} & ::= & [\texttt{-}]\ \{\texttt{0}\ldots\texttt{9}\}^{+}\ [\texttt{.}\ \{\texttt{0}\ldots\texttt{9}\}]\ [(\texttt{e} \mid \texttt{E})\ [\texttt{+} \mid \texttt{-}]\ \{\texttt{0}\ldots\texttt{9}\}^{+}] \end{array}$$

Floating-point decimals consist in an integer part, a decimal part and an exponent part. The integer part is a sequence of one or more digits, optionally preceded by a minus sign. The decimal part is a decimal point followed by zero, one or more digits. The exponent part is the character `e` or `E` followed by an optional `+` or `-` sign, followed by one or more digits. The decimal part or the exponent part can be omitted, but not both to avoid ambiguity with integer literals. The interpretation of floating-point literals that fall outside the range of representable floating-point values is undefined.

### Character literals

$$\begin{array}{rcl} \textit{char-literal} & ::= & \texttt{'}\ \textit{regular-char}\ \texttt{'} \\ & \mid & \texttt{'}\ \texttt{\\}\ (\texttt{\\} \mid \texttt{'} \mid \texttt{n} \mid \texttt{t} \mid \texttt{b} \mid \texttt{r})\ \texttt{'} \\ & \mid & \texttt{'}\ \texttt{\\}\ (\texttt{0}\ldots\texttt{9})\ (\texttt{0}\ldots\texttt{9})\ (\texttt{0}\ldots\texttt{9})\ \texttt{'} \end{array}$$

Character literals are delimited by `'` (single quote) characters. The two single quotes enclose either one character different from `'` and `\`, or one of the escape sequences below:

| Sequence | Character denoted |
|---|---|
| `\\` | backslash (`\`) |
| `\'` | single quote (`'`) |
| `\n` | linefeed (LF) |
| `\r` | return (CR) |
| `\t` | horizontal tabulation (TAB) |
| `\b` | backspace (BS) |
| `\`*ddd* | the character with ASCII code *ddd* in decimal |

**String literals**

```
   string-literal  ::=  " {string-character} "

string-character  ::=  regular-char
                    |  \ (\ | " | n | t | b | r)
                    |  \ (0...9) (0...9) (0...9)
```

String literals are delimited by `"` (double quote) characters. The two double quotes enclose a sequence of either characters different from `"` and `\`, or escape sequences from the table below:

| Sequence | Character denoted |
|---|---|
| `\\` | backslash (`\`) |
| `\"` | double quote (`"`) |
| `\n` | linefeed (LF) |
| `\r` | return (CR) |
| `\t` | horizontal tabulation (TAB) |
| `\b` | backspace (BS) |
| `\`*ddd* | the character with ASCII code *ddd* in decimal |

To allow splitting long string literals across lines, the sequence `\`*newline blanks* (a `\` at end-of-line followed by any number of blanks at the beginning of the next line) is ignored inside string literals.

The current implementation places no restrictions on the length of string literals.

**Prefix and infix symbols**

```
 infix-symbol  ::=  (= | < | > | @ | ^ | | | & | + | - | * | / | $ | %) {operator-char}

prefix-symbol  ::=  (! | ? | ~) {operator-char}

operator-char  ::=  ! | $ | % | & | * | + | - | . | / | : | < | = | > | ? | @ | ^ | | | ~
```

Sequences of "operator characters", such as `<=>` or `!!`, are read as a single token from the *infix-symbol* or *prefix-symbol* class. These symbols are parsed as prefix and infix operators inside expressions, but otherwise behave much as identifiers.

**Keywords**

The identifiers below are reserved as keywords, and cannot be employed otherwise:

```
and         as          assert      asr         begin       class
closed      constraint  do          done        downto      else
end         exception   external    false       for         fun
function    functor     if          in          include     inherit
land        lazy        let         lor         lsl         lsr
lxor        match       method      mod         module      mutable
new         of          open        or          parser      private
rec         sig         struct      then        to          true
try         type        val         virtual     when        while
with
```

The following character sequences are also keywords:

```
#     &     '     (     )     *     ,     ->    ?
.     ..    .(    .[    :     ::    :=    ;     ;;
<-    =     [     [|    [<    {<    ]     |]    >]
>}    _     {     |     }
```

**Ambiguities**

Lexical ambiguities are resolved according to the "longest match" rule: when a character sequence can be decomposed into two tokens in several different ways, the decomposition retained is the one with the longest first token.

## 5.2 Values

This section describes the kinds of values that are manipulated by Caml Light programs.

### 5.2.1 Base values

**Integer numbers**

Integer values are integer numbers from $-2^{30}$ to $2^{30}-1$, that is $-1073741824$ to $1073741823$. The implementation may support a wider range of integer values: on 64-bit platforms, the current implementation supports integers ranging from $-2^{62}$ to $2^{62}-1$.

**Floating-point numbers**

Floating-point values are numbers in floating-point representation. The current implementation uses double-precision floating-point numbers conforming to the IEEE 754 standard, with 53 bits of mantissa and an exponent ranging from $-1022$ to $1023$.

#### Characters

Character values are represented as 8-bit integers between 0 and 255. Character codes between 0 and 127 are interpreted following the ASCII standard. The current implementation interprets character codes between 128 and 255 following the ISO 8859-1 standard.

#### Character strings

String values are finite sequences of characters. The current implementation supports strings containing up to $2^{24} - 6$ characters (16777210 characters).

### 5.2.2 Tuples

Tuples of values are written $(v_1, \ldots, v_n)$, standing for the $n$-tuple of values $v_1$ to $v_n$. The current implementation supports tuple of up to $2^{22} - 1$ elements (4194303 elements).

### 5.2.3 Records

Record values are labeled tuples of values. The record value written $\{label_1 = v_1; \ldots; label_n = v_n\}$ associates the value $v_i$ to the record label $label_i$, for $i = 1 \ldots n$. The current implementation supports records with up to $2^{22} - 1$ fields (4194303 fields).

### 5.2.4 Arrays

Arrays are finite, variable-sized sequences of values of the same type. The current implementation supports arrays containing to $2^{22} - 1$ elements (4194303 elements).

### 5.2.5 Variant values

Variant values are either a constant constructor, or a pair of a non-constant constructor and a value. The former case is written *cconstr*; the latter case is written *ncconstr*$(v)$, where $v$ is said to be the argument of the non-constant constructor *ncconstr*.

The following constants are treated like built-in constant constructors:

| Constant | Constructor |
|---|---|
| `false` | the boolean false |
| `true` | the boolean true |
| `()` | the “unit” value |
| `[]` | the empty list |

The current implementation limits the number of distinct constructors in a given variant type to at most 249.

### 5.2.6 Functions

Functional values are mappings from values to values.

### 5.2.7 Objects

Objects are composed of a hidden internal state which is a record of instance variables, and a set of methods for accessing and modifying these variables. The structure of an object is described by the toplevel class that created it.

## 5.3 Names

Identifiers are used to give names to several classes of language objects and refer to these objects by name later:

- value names (syntactic class *value-name*),
- value constructors (constant – class *cconstr-name* – or non-constant – class *ncconstr-name*),
- type constructors (*typeconstr-name*),
- record labels (*label-name*),
- class names (*class-name*),
- method names (*method-name*),
- instance variable names (*inst-var-name*),
- module names (*module-name*),
- module type names (*modtype-name*).

These nine name spaces are distinguished both by the context and by the capitalization of the identifier: whether the first letter of the identifier is in lowercase (written *lowercase-ident* below) or in uppercase (written *capitalized-ident*). Underscore is considered a lowercase letter for this purpose.

**Naming objects**

*value-name* ::= *lowercase-ident*
  | `(` *operator-name* `)`

*operator-name* ::= *prefix-symbol* | *infix-symbol* | `*` | `=` | `or` | `&` | `:=`

*cconstr-name* ::= *capitalized-ident*
  | `false`
  | `true`
  | `[ ]`
  | `( )`

*ncconstr-name* ::= *capitalized-ident*
  | `::`

*typeconstr-name* ::= *lowercase-ident*

*label-name* ::= *lowercase-ident*

*module-name* ::= *capitalized-ident*

*modtype-name* ::= *ident*

*class-name* ::= *lowercase-ident*

*inst-var-name* ::= *lowercase-ident*

*method-name* ::= *lowercase-ident*

As shown above, prefix and infix symbols as well as some keywords can be used as value names, provided they are written between parentheses. Keywords such as '::' and 'false' are also constructor names. The capitalization rules are summarized in the table below.

| Name space | Case of first letter |
|---|---|
| Values | lowercase |
| Constructors | uppercase |
| Type constructors | lowercase |
| Record labels | lowercase |
| Classes | lowercase |
| Methods | lowercase |
| Modules | uppercase |
| Module types | any |

**Referring to named objects**

| | | |
|---|---|---|
| *value-path* | ::= | *value-name* |
| | \| | *module-path* `.` *lowercase-ident* |
| *cconstr* | ::= | *cconstr-name* |
| | \| | *module-path* `.` *capitalized-ident* |
| *ncconstr* | ::= | *ncconstr-name* |
| | \| | *module-path* `.` *capitalized-ident* |
| *typeconstr* | ::= | *typeconstr-name* |
| | \| | *extended-module-path* `.` *lowercase-ident* |
| *label* | ::= | *label-name* |
| | \| | *module-path* `.` *lowercase-ident* |
| *module-path* | ::= | *module-name* |
| | \| | *module-path* `.` *capitalized-ident* |
| *extended-module-path* | ::= | *module-name* |
| | \| | *extended-module-path* `.` *capitalized-ident* |
| | \| | *extended-module-path* `(` *extended-module-path* `)` |
| *modtype-path* | ::= | *modtype-name* |
| | \| | *extended-module-path* `.` *ident* |
| *class-path* | ::= | *class-name* |
| | \| | *module-path* `.` *lowercase-ident* |

A named object can be referred to either by its name (following the usual static scoping rules for names) or by an access path *prefix* `.` *name*, where *prefix* designates a module and *name* is the name of an object defined in that module. The first component of the path, *prefix*, is either a simple module name or an access path $name_1$ `.` $name_2 \ldots$, in case the defining module is itself nested inside other modules. For referring to type constructors or module types, the *prefix* can also contain simple functor applications (as in the syntactic class *extended-module-path* above), in case the defining module is the result of a functor application.

Instance variable names and method names need not be qualified: the former are local to a class while the latter are global labels.

## 5.4 Type expressions

| | | |
|---|---|---|
| *typexpr* | ::= | `'` *ident* |
| | \| | `(` *typexpr* `)` |
| | \| | *typexpr* `->` *typexpr* |
| | \| | *typexpr* {`*` *typexpr*}$^+$ |
| | \| | *typeconstr* |
| | \| | *typexpr* *typeconstr* |
| | \| | `(` *typexpr* {`,` *typexpr*} `)` *typeconstr* |
| | \| | *typexpr* `as` `'` *ident* |
| | \| | `<` [`..`] `>` |
| | \| | `<` *method-type* {`;` *method-type*} [`;` `..`] `>` |
| | \| | `#` *class-path* |
| | \| | *typexpr* `#` *class-path* |
| | \| | `(` *typexpr* {`,` *typexpr*}`)` `#` *class-path* |
| *method-type* | ::= | *method-name* `:` *typexpr* |

The table below shows the relative precedences and associativity of operators and non-closed type constructions. The constructions with higher precedences come first.

| Operator | Associativity |
|---|---|
| Type constructor application | – |
| `*` | – |
| `->` | right |
| `as` | – |

Type expressions denote types in definitions of data types as well as in type constraints over patterns and expressions.

### Type variables

The type expression `'` *ident* stands for the type variable named *ident*. In data type definitions, type variables are names for the data type parameters. In type constraints, they represent unspecified types that can be instantiated by any type to satisfy the type constraint.

### Parenthesized types

The type expression `(` *typexpr* `)` denotes the same type as *typexpr*.

### Function types

The type expression $typexpr_1$ `->` $typexpr_2$ denotes the type of functions mapping arguments of type $typexpr_1$ to results of type $typexpr_2$.

### Tuple types

The type expression $typexpr_1$ `*` ... `*` $typexpr_n$ denotes the type of tuples whose elements belong to types $typexpr_1, \ldots typexpr_n$ respectively.

### Constructed types

Type constructors with no parameter, as in *typeconstr*, are type expressions.

The type expression *typexpr typeconstr*, where *typeconstr* is a type constructor with one parameter, denotes the application of the unary type constructor *typeconstr* to the type *typexpr*.

The type expression $(typexpr_1, \ldots, typexpr_n)$ *typeconstr*, where *typeconstr* is a type constructor with $n$ parameters, denotes the application of the $n$-ary type constructor *typeconstr* to the types $typexpr_1$ through $typexpr_n$.

### Recursive types

The type expression *typexpr* `as` `'` *ident* denotes the same type as *typexpr*, and also binds the type variable *ident* to type *typexpr* both in *typexpr* and in the remaining part of the type. If the type variable *ident* actually occurs in *typexpr*, a recursive type is created. Recursive types are only allowed when any recursion crosses an object type.

### Object types

An object type `<` *method-type* {`;` *method-type*} `>` is a record of method types.

The type `<` *method-type* {`;` *method-type*} `; .. >` is the type of an object with methods and their associated types are described by $method\text{-}type_1, \ldots, method\text{-}type_n$, and possibly some other methods represented by the ellipsis. This ellipsis actually is a special kind of type variable, named row variable.

### #-types

The type `#` *class-path* is a special kind of abbreviation. This abbreviation unifies with the type of any object belonging to a subclass of class *class-path*. It is handled in a special way as it usually hides a type variable (an ellipsis, representing the methods that may be added in a subclass). In particular, it vanishes when the ellipsis gets instantiated. Each type expression `#` *class-path* defines a new type variable, so type `#` *class-path* `->` `#` *class-path* is usually not the same as type `#` *class-path* `as` `'` *ident* `->` `'` *ident*.

## 5.5 Constants

$$
\begin{array}{rcl}
\textit{constant} & ::= & \textit{integer-literal} \\
& \mid & \textit{float-literal} \\
& \mid & \textit{char-literal} \\
& \mid & \textit{string-literal} \\
& \mid & \textit{cconstr}
\end{array}
$$

The syntactic class of constants comprises literals from the four base types (integers, floating-point numbers, characters, character strings), and constant constructors.

## 5.6 Patterns

```
pattern ::= value-name
         |  _
         |  constant
         |  pattern as value-name
         |  ( pattern )
         |  ( pattern : typexpr )
         |  pattern | pattern
         |  ncconstr pattern
         |  pattern {, pattern}
         |  { label = pattern {; label = pattern} }
         |  [ pattern {; pattern} ]
         |  pattern :: pattern
```

The table below shows the relative precedences and associativity of operators and non-closed pattern constructions. The constructions with higher precedences come first.

| Operator | Associativity |
|---|---|
| Constructor application | – |
| `::` | right |
| `,` | – |
| `\|` | left |
| `as` | – |

Patterns are templates that allow selecting data structures of a given shape, and binding identifiers to components of the data structure. This selection operation is called pattern matching; its outcome is either "this value does not match this pattern", or "this value matches this pattern, resulting in the following bindings of names to values".

### Variable patterns

A pattern that consists in a value name matches any value, binding the name to the value. The pattern `_` also matches any value, but does not bind any name.

Patterns are *linear*: a variable cannot appear several times in a given pattern. In particular, there is no way to test for equality between two parts of a data structure using only a pattern (but `when` guards can be used for this purpose).

### Constant patterns

A pattern consisting in a constant matches the values that are equal to this constant.

### Alias patterns

The pattern $pattern_1$ `as` *value-name* matches the same values as $pattern_1$. If the matching against $pattern_1$ is successful, the name *name* is bound to the matched value, in addition to the bindings performed by the matching against $pattern_1$.

### Parenthesized patterns

The pattern `(` *pattern*$_1$ `)` matches the same values as *pattern*$_1$. A type constraint can appear in a parenthesized pattern, as in `(` *pattern*$_1$ `:` *typexpr* `)`. This constraint forces the type of *pattern*$_1$ to be compatible with *type*.

### "Or" patterns

The pattern *pattern*$_1$ `|` *pattern*$_2$ represents the logical "or" of the two patterns *pattern*$_1$ and *pattern*$_2$. A value matches *pattern*$_1$ `|` *pattern*$_2$ either if it matches *pattern*$_1$ or if it matches *pattern*$_2$. The two sub-patterns *pattern*$_1$ and *pattern*$_2$ must contain no identifiers. Hence no bindings are returned by matching against an "or" pattern.

### Variant patterns

The pattern *ncconstr* *pattern*$_1$ matches all variants whose constructor is equal to *ncconstr*, and whose argument matches *pattern*$_1$.

The pattern *pattern*$_1$ `::` *pattern*$_2$ matches non-empty lists whose heads match *pattern*$_1$, and whose tails match *pattern*$_2$. This pattern behaves like `( :: ) (` *pattern*$_1$ `,` *pattern*$_2$ `)`.

The pattern `[` *pattern*$_1$ `;` ... `;` *pattern*$_n$ `]` matches lists of length $n$ whose elements match *pattern*$_1$ ... *pattern*$_n$, respectively. This pattern behaves like *pattern*$_1$ `::` ... `::` *pattern*$_n$ `::` `[]`.

### Tuple patterns

The pattern *pattern*$_1$ `,` ... `,` *pattern*$_n$ matches $n$-tuples whose components match the patterns *pattern*$_1$ through *pattern*$_n$. That is, the pattern matches the tuple values $(v_1, \ldots, v_n)$ such that *pattern*$_i$ matches $v_i$ for $i = 1, \ldots, n$.

### Record patterns

The pattern `{` *label*$_1$ `=` *pattern*$_1$ `;` ... `;` *label*$_n$ `=` *pattern*$_n$ `}` matches records that define at least the labels *label*$_1$ through *label*$_n$, and such that the value associated to *label*$_i$ match the pattern *pattern*$_i$, for $i = 1, \ldots, n$. The record value can define more labels than *label*$_1$ ... *label*$_n$; the values associated to these extra labels are not taken into account for matching.

## 5.7 Expressions

```
            expr  ::=  value-path
                    |  constant
                    |  ( expr )
                    |  begin expr end
                    |  ( expr : typexpr )
                    |  expr , expr {, expr}
                    |  ncconstr expr
                    |  expr :: expr
                    |  [ expr {; expr} ]
                    |  [| expr {; expr} |]
                    |  { label = expr {; label = expr} }
                    |  expr expr
                    |  prefix-symbol expr
                    |  expr infix-op expr
                    |  expr . label
                    |  expr . label <- expr
                    |  expr .( expr )
                    |  expr .( expr ) <- expr
                    |  expr .[ expr ]
                    |  expr .[ expr ] <- expr
                    |  if expr then expr [else expr]
                    |  while expr do expr done
                    |  for ident = expr (to | downto) expr do expr done
                    |  expr ; expr
                    |  match expr with pattern-matching
                    |  function pattern-matching
                    |  fun multiple-matching
                    |  try expr with pattern-matching
                    |  let [rec] let-binding {and let-binding} in expr
                    |  new class-path
                    |  expr # method-name
                    |  ( expr :> typexpr )
                    |  ( expr : typexpr :> typexpr )
                    |  {< inst-var-name = expr {; inst-var-name = expr} >}

pattern-matching  ::=  pattern [when expr] -> expr {| pattern [when expr] -> expr}

multiple-matching ::=  {pattern}+ [when expr] -> expr


     let-binding  ::=  pattern = expr
                    |  value-name {pattern}+ [: typexpr] = expr

        infix-op  ::=  infix-symbol
                    |  * | = | or | &
```

The table below shows the relative precedences and associativity of operators and non-closed constructions. The constructions with higher precedence come first. For infix and prefix symbols, we write "`*`..." to mean "any symbol starting with `*`".

| Construction or operator | Associativity |
|---|---|
| prefix-symbol | – |
| `.` `.(` `.[` | – |
| function application | left |
| constructor application | – |
| `-` `-.` (prefix) | – |
| `**`... | right |
| `*`... `/`... `%`... `mod` | left |
| `+`... `-`... | left |
| `::` | right |
| `@`... `^`... | right |
| comparisons (`=` `==` `<` etc.), all other infix symbols | left |
| `not` | – |
| `&` `&&` | left |
| `or` `\|\|` | left |
| `,` | – |
| `<-` `:=` | right |
| `if` | – |
| `;` | right |
| `let` `match` `fun` `function` `try` | – |

### 5.7.1 Basic expressions

#### Constants

Expressions consisting in a constant evaluate to this constant.

#### Value paths

Expressions consisting in an access path evaluate to the value bound to this path in the current evaluation environment. The path can be either a value name or an access path to a value component of a module.

#### Parenthesized expressions

The expressions `(` *expr* `)` and `begin` *expr* `end` have the same value as *expr*. Both constructs are semantically equivalent, but it is good style to use `begin`...`end` inside control structures:

```
        if ... then begin ... ; ... end else begin ... ; ... end
```

and `(`...`)` for the other grouping situations.

Parenthesized expressions can contain a type constraint, as in `(` *expr* `:` *type* `)`. This constraint forces the type of *expr* to be compatible with *type*.

Parenthesized expressions can also contain coercions `(` *expr* [`:` *type*] `:>` *type* `)` (see subsection 5.7.5 below).

### Function application

Function application is denoted by juxtaposition of expressions. The expression $expr_1$ $expr_2 \ldots expr_n$ evaluates the expressions $expr_1$ to $expr_n$. The expression $expr_1$ must evaluate to a functional value, which is then applied to the values of $expr_2, \ldots, expr_n$. The order in which the expressions $expr_1, \ldots, expr_n$ are evaluated is not specified.

### Function definition

Two syntactic forms are provided to define functions. The first form is introduced by the keyword `function`:

```
function  pattern₁  ->  expr₁
       |  ...
       |  patternₙ  ->  exprₙ
```

This expression evaluates to a functional value with one argument. When this function is applied to a value $v$, this value is matched against each pattern $pattern_1$ to $pattern_n$. If one of these matchings succeeds, that is, if the value $v$ matches the pattern $pattern_i$ for some $i$, then the expression $expr_i$ associated to the selected pattern is evaluated, and its value becomes the value of the function application. The evaluation of $expr_i$ takes place in an environment enriched by the bindings performed during the matching.

If several patterns match the argument $v$, the one that occurs first in the function definition is selected. If none of the patterns matches the argument, the exception `Match_failure` is raised.

The other form of function definition is introduced by the keyword `fun`:

`fun` $pattern_1 \ldots pattern_n$ `->` *expr*

This expression is equivalent to:

`function` $pattern_1$ `->` ... `function` $pattern_n$ `->` *expr*

That is, the `fun` expression above evaluates to a curried function with $n$ arguments: after applying this function $n$ times to the values $v_1 \ldots v_m$, the values will be matched in parallel against the patterns $pattern_1 \ldots pattern_n$. If the matching succeeds, the function returns the value of *expr* in an environment enriched by the bindings performed during the matchings. If the matching fails, the exception `Match_failure` is raised.

### Guards in pattern-matchings

Cases of a pattern matching (in the `function`, `fun`, `match` and `try` constructs) can include guard expressions, which are arbitrary boolean expressions that must evaluate to `true` for the match case to be selected. Guards occur just before the `->` token and are introduced by the `when` keyword:

```
function  pattern₁ [when cond₁]  ->  expr₁
       |  ...
       |  patternₙ [when condₙ]  ->  exprₙ
```

Matching proceeds as described before, except that if the value matches some pattern *pattern*$_i$ which has a guard *cond*$_i$, then the expression *cond*$_i$ is evaluated (in an environment enriched by the bindings performed during matching). If *cond*$_i$ evaluates to `true`, then *expr*$_i$ is evaluated and its value returned as the result of the matching, as usual. But if *cond*$_i$ evaluates to `false`, the matching is resumed against the patterns following *pattern*$_i$.

**Local definitions**

The `let` and `let rec` constructs bind value names locally. The construct

`let` *pattern*$_1$ `=` *expr*$_1$ `and`... `and` *pattern*$_n$ `=` *expr*$_n$ `in` *expr*

evaluates *expr*$_1$ ... *expr*$_n$ in some unspecified order, then matches their values against the patterns *pattern*$_1$ ... *pattern*$_n$. If the matchings succeed, *expr* is evaluated in the environment enriched by the bindings performed during matching, and the value of *expr* is returned as the value of the whole `let` expression. If one of the matchings fails, the exception `Match_failure` is raised.

An alternate syntax is provided to bind variables to functional values: instead of writing

`let` *ident* `=` `fun` *pattern*$_1$ ... *pattern*$_m$ `->` *expr*

in a `let` expression, one may instead write

`let` *ident* *pattern*$_1$ ... *pattern*$_m$ `=` *expr*

Recursive definitions of names are introduced by `let rec`:

`let rec` *pattern*$_1$ `=` *expr*$_1$ `and`... `and` *pattern*$_n$ `=` *expr*$_n$ `in` *expr*

The only difference with the `let` construct described above is that the bindings of names to values performed by the pattern-matching are considered already performed when the expressions *expr*$_1$ to *expr*$_n$ are evaluated. That is, the expressions *expr*$_1$ to *expr*$_n$ can reference identifiers that are bound by one of the patterns *pattern*$_1$, ..., *pattern*$_n$, and expect them to have the same value as in *expr*, the body of the `let rec` construct.

The recursive definition is guaranteed to behave as described above if the expressions *expr*$_1$ to *expr*$_n$ are function definitions (`fun`... or `function`...), and the patterns *pattern*$_1$ ... *pattern*$_n$ are just value names, as in:

`let rec` *name*$_1$ `= fun`... `and`... `and` *name*$_n$ `= fun`... `in` *expr*

This defines *name*$_1$ ... *name*$_n$ as mutually recursive functions local to *expr*.

The behavior of other forms of `let rec` definitions is implementation-dependent. The current implementation also supports a certain class of recursive definitions of non-functional values, such as

`let rec` *name*$_1$ `= 1 ::` *name*$_2$ `and` *name*$_2$ `= 2 ::` *name*$_1$ `in` *expr*

which binds *name*$_1$ to the cyclic list `1::2::1::2::`..., and *name*$_2$ to the cyclic list `2::1::2::1::`... Informally, the class of accepted definitions consists of those definitions where the defined names occur only inside function bodies or as argument to a data constructor.

### 5.7.2 Control structures

#### Sequence

The expression *expr*$_1$ `;` *expr*$_2$ evaluates *expr*$_1$ first, then *expr*$_2$, and returns the value of *expr*$_2$.

#### Conditional

The expression `if` *expr*$_1$ `then` *expr*$_2$ `else` *expr*$_3$ evaluates to the value of *expr*$_2$ if *expr*$_1$ evaluates to the boolean `true`, and to the value of *expr*$_3$ if *expr*$_1$ evaluates to the boolean `false`.

The `else` *expr*$_3$ part can be omitted, in which case it defaults to `else ()`.

#### Case expression

The expression

```
match expr
 with pattern1  ->  expr1
    | ...
    | patternn  ->  exprn
```

matches the value of *expr* against the patterns *pattern*$_1$ to *pattern*$_n$. If the matching against *pattern*$_i$ succeeds, the associated expression *expr*$_i$ is evaluated, and its value becomes the value of the whole `match` expression. The evaluation of *expr*$_i$ takes place in an environment enriched by the bindings performed during matching. If several patterns match the value of *expr*, the one that occurs first in the `match` expression is selected. If none of the patterns match the value of *expr*, the exception `Match_failure` is raised.

#### Boolean operators

The expression *expr*$_1$ `&` *expr*$_2$ evaluates to `true` if both *expr*$_1$ and *expr*$_2$ evaluate to `true`; otherwise, it evaluates to `false`. The first component, *expr*$_1$, is evaluated first. The second component, *expr*$_2$, is not evaluated if the first component evaluates to `false`. Hence, the expression *expr*$_1$ `&` *expr*$_2$ behaves exactly as

`if` *expr*$_1$ `then` *expr*$_2$ `else false`.

The expression *expr*$_1$ `or` *expr*$_2$ evaluates to `true` if one of *expr*$_1$ and *expr*$_2$ evaluates to `true`; otherwise, it evaluates to `false`. The first component, *expr*$_1$, is evaluated first. The second component, *expr*$_2$, is not evaluated if the first component evaluates to `true`. Hence, the expression *expr*$_1$ `or` *expr*$_2$ behaves exactly as

`if` *expr*$_1$ `then true else` *expr*$_2$.

#### Loops

The expression `while` *expr*$_1$ `do` *expr*$_2$ `done` repeatedly evaluates *expr*$_2$ while *expr*$_1$ evaluates to `true`. The loop condition *expr*$_1$ is evaluated and tested at the beginning of each iteration. The whole `while...done` expression evaluates to the unit value `()`.

The expression `for` *name* `=` $expr_1$ `to` $expr_2$ `do` $expr_3$ `done` first evaluates the expressions $expr_1$ and $expr_2$ (the boundaries) into integer values $n$ and $p$. Then, the loop body $expr_3$ is repeatedly evaluated in an environment where *name* is successively bound to the values $n$, $n+1$, ..., $p-1$, $p$. The loop body is never evaluated if $n > p$.

The expression `for` *name* `=` $expr_1$ `downto` $expr_2$ `do` $expr_3$ `done` evaluates similarly, except that *name* is successively bound to the values $n$, $n-1$, ..., $p+1$, $p$. The loop body is never evaluated if $n < p$.

In both cases, the whole `for` expression evaluates to the unit value `()`.

**Exception handling**

The expression

```
try   expr
with  pattern1  ->  expr1
   |  ...
   |  patternn  ->  exprn
```

evaluates the expression *expr* and returns its value if the evaluation of *expr* does not raise any exception. If the evaluation of *expr* raises an exception, the exception value is matched against the patterns $pattern_1$ to $pattern_n$. If the matching against $pattern_i$ succeeds, the associated expression $expr_i$ is evaluated, and its value becomes the value of the whole `try` expression. The evaluation of $expr_i$ takes place in an environment enriched by the bindings performed during matching. If several patterns match the value of *expr*, the one that occurs first in the `try` expression is selected. If none of the patterns matches the value of *expr*, the exception value is raised again, thereby transparently "passing through" the `try` construct.

## 5.7.3 Operations on data structures

**Products**

The expression $expr_1$ `,` ... `,` $expr_n$ evaluates to the $n$-tuple of the values of expressions $expr_1$ to $expr_n$. The evaluation order for the subexpressions is not specified.

**Variants**

The expression *ncconstr expr* evaluates to the variant value whose constructor is *ncconstr*, and whose argument is the value of *expr*.

For lists, some syntactic sugar is provided. The expression $expr_1$ `::` $expr_2$ stands for the constructor `( :: )` applied to the argument `(` $expr_1$ `,` $expr_2$ `)`, and therefore evaluates to the list whose head is the value of $expr_1$ and whose tail is the value of $expr_2$. The expression `[` $expr_1$ `;` ... `;` $expr_n$ `]` is equivalent to $expr_1$ `::` ... `::` $expr_n$ `::` `[]`, and therefore evaluates to the list whose elements are the values of $expr_1$ to $expr_n$.

**Records**

The expression `{` $label_1$ `=` $expr_1$ `;` ... `;` $label_n$ `=` $expr_n$ `}` evaluates to the record value `{` $label_1$ `=` $v_1$ `;` ... `;` $label_n$ `=` $v_n$ `}`, where $v_i$ is the value of $expr_i$ for $i = 1, \ldots, n$. The labels $label_1$ to $label_n$ must all belong to the same record types; all labels belonging to this record type must appear exactly once in the record expression, though they can appear in any order. The order in which $expr_1$ to $expr_n$ are evaluated is not specified.

The expression $expr_1$ `.` *label* evaluates $expr_1$ to a record value, and returns the value associated to *label* in this record value.

The expression $expr_1$ `.` *label* `<-` $expr_2$ evaluates $expr_1$ to a record value, which is then modified in-place by replacing the value associated to *label* in this record by the value of $expr_2$. This operation is permitted only if *label* has been declared `mutable` in the definition of the record type. The whole expression $expr_1$ `.` *label* `<-` $expr_2$ evaluates to the unit value `()`.

### Arrays

The expression `[|` $expr_1$ `;` ... `;` $expr_n$ `|]` evaluates to a $n$-element array, whose elements are initialized with the values of $expr_1$ to $expr_n$ respectively. The order in which these expressions are evaluated is unspecified.

The expression $expr_1$ `.(` $expr_2$ `)` returns the value of element number $expr_2$ in the array denoted by $expr_1$. The first element has number 0; the last element has number $n-1$, where $n$ is the size of the array. The exception `Invalid_argument` is raised if the access is out of bounds.

The expression $expr_1$ `.(` $expr_2$ `) <-` $expr_3$ modifies in-place the array denoted by $expr_1$, replacing element number $expr_2$ by the value of $expr_3$. The exception `Invalid_argument` is raised if the access is out of bounds. The value of the whole expression is `()`.

### Strings

The expression $expr_1$ `.[` $expr_2$ `]` returns the value of character number $expr_2$ in the string denoted by $expr_1$. The first character has number 0; the last character has number $n-1$, where $n$ is the length of the string. The exception `Invalid_argument` is raised if the access is out of bounds.

The expression $expr_1$ `.[` $expr_2$ `] <-` $expr_3$ modifies in-place the string denoted by $expr_1$, replacing character number $expr_2$ by the value of $expr_3$. The exception `Invalid_argument` is raised if the access is out of bounds. The value of the whole expression is `()`.

## 5.7.4 Operators

Symbols from the class `infix-symbols`, as well as the keywords `*`, `=`, `or` and `&`, can appear in infix position (between two expressions). Symbols from the class `prefix-symbols` can appear in prefix position (in front of an expression).

Infix and prefix symbols do not have a fixed meaning: they are simply interpreted as applications of functions bound to the names corresponding to the symbols. The expression *prefix-symbol expr* is interpreted as the application `(` *prefix-symbol* `)` *expr*. Similarly, the expression $expr_1$ *infix-symbol* $expr_2$ is interpreted as the application `(` *infix-symbol* `)` $expr_1$ $expr_2$.

The table below lists the symbols defined in the initial environment and their initial meaning. (See the description of the standard library module `Pervasive` in chapter 16 for more details). Their meaning may be changed at any time using `let (` *infix-op* `)` $name_1$ $name_2$ `=`...

| Operator | Initial meaning |
|---|---|
| `+` | Integer addition. |
| `-` (infix) | Integer subtraction. |
| `-` (prefix) | Integer negation. |
| `*` | Integer multiplication. |
| `/` | Integer division. Raise `Division_by_zero` if second argument is zero. The result is unspecified if either argument is negative. |
| `mod` | Integer modulus. Raise `Division_by_zero` if second argument is zero. The result is unspecified if either argument is negative. |
| `land` | Bitwise logical “and” on integers. |
| `lor` | Bitwise logical “or on integers. |
| `lxor` | Bitwise logical “exclusive or” on integers. |
| `lsl` | Bitwise logical shift left on integers. |
| `lsr` | Bitwise logical shift right on integers. |
| `asr` | Bitwise arithmetic shift right on integers. |
| `+.` | Floating-point addition. |
| `-.` (infix) | Floating-point subtraction. |
| `-.` (prefix) | Floating-point negation. |
| `*.` | Floating-point multiplication. |
| `/.` | Floating-point division. |
| `**` | Floating-point exponentiation. |
| `@` | List concatenation. |
| `^` | String concatenation. |
| `!` | Dereferencing (return the current contents of a reference). |
| `:=` | Reference assignment (update the reference given as first argument with the value of the second argument). |
| `=` | Structural equality test. |
| `<>` | Structural inequality test. |
| `==` | Physical equality test. |
| `!=` | Physical inequality test. |
| `<` | Test “less than”. |
| `<=` | Test “less than or equal”. |
| `>` | Test “greater than”. |
| `>=` | Test “greater than or equal” |

### 5.7.5 Objects

#### Object creation

When *class-path* evaluates to a class body, `new` *class-path* evaluates to an object containing the instance variables and methods of this class.

When *class-path* evaluates to a class function, `new` *class-path* evaluates to a function expecting the same number of arguments and returning a new object of this class.

### Message sending

The expression *expr* `#` *method-name* invokes the method *method-name* of the object denoted by *expr*.

### Coercion

The type of an object can be coerced (weakened) to a supertype. The expression `(` *expr* `:>` *typexpr* `)` coerces the expression *expr* to type *typexpr*. The expression `(` *expr* `:` $typexpr_1$ `:>` $typexpr_2$ `)` coerces the expression *expr* from type $typexpr_1$ to type $typexpr_2$. The former operator will sometimes fail to coerce an expression *expr* from a type $t_1$ to a type $t_2$ even if type $t_1$ is a subtype of type $t_2$. In this case, the latter operator should be used.

In a class definition, coercion to the type this class defines is the identity, as this type abbreviation is not yet completely defined.

### Object duplication

An object can be duplicated using the library function `Oo.copy` (see section 17.17). Inside a method, the expression `{<` *inst-var-name* `=` *expr* {`;` *inst-var-name* `=` *expr*} `>}` returns a copy of self with the given instance variables replaced by the values of the associated expressions; other instance variables have the same value in the returned object as in self.

## 5.8 Type and exception definitions

### 5.8.1 Type definitions

Type definitions bind type constructors to data types: either variant types, record types, type abbreviations, or abstract data types. They also bind the value constructors and record labels associated with the definition.

```
     type-definition  ::=  type typedef {and typedef}
             typedef  ::=  [type-params] typeconstr-name [type-equation] [type-representation] {constraint}
       type-equation  ::=  = typexpr
 type-representation  ::=  = constr-decl {| constr-decl}
                        |  = { label-decl {; label-decl} }
         type-params  ::=  ' ident
                        |  ( ' ident {, ' ident} )
         constr-decl  ::=  cconstr-name
                        |  ncconstr-name of typexpr
          label-decl  ::=  label-name : typexpr
                        |  mutable label-name : typexpr
          constraint  ::=  constraint ' ident = typexpr
```

Type definitions are introduced by the `type` keyword, and consist in one or several simple definitions, possibly mutually recursive, separated by the `and` keyword. Each simple definition defines one type constructor.

A simple definition consists in a lowercase identifier, possibly preceded by one or several type parameters, and followed by an optional type equation, then an optional type representation, and then a constraint clause. The identifier is the name of the type constructor being defined.

The optional type parameters are either one type variable `'` *ident*, for type constructors with one parameter, or a list of type variables `(`$'$ $\mathit{ident}_1, \ldots,$ `'` $\mathit{ident}_n$`)`, for type constructors with several parameters. These type parameters can appear in the type expressions of the right-hand side of the definition.

The optional type equation `=` *typexpr* makes the defined type equivalent to the type expression *typexpr* on the right of the `=` sign: one can be substituted for the other during typing. If no type equation is given, a new type is generated: the defined type is incompatible with any other type.

The optional type representation describes the data structure representing the defined type, by giving the list of associated constructors (if it is a variant type) or associated labels (if it is a record type). If no type representation is given, nothing is assumed on the structure of the type besides what is stated in the optional type equation.

The type representation `=` *constr-decl* {`|` *constr-decl*} describes a variant type. The constructor declarations $\mathit{constr\text{-}decl}_1, \ldots, \mathit{constr\text{-}decl}_n$ describe the constructors associated to this variant type. The constructor declaration *ncconstr-name* `of` *typexpr* declares the name *ncconstr-name* as a non-constant constructor, whose argument has type *typexpr*. The constructor declaration *cconstr-name* declares the name *cconstr-name* as a constant constructor. Constructor names must be capitalized.

The type representation `= {` *label-decl* {`;` *label-decl*} `}` describes a record type. The label declarations $\mathit{label\text{-}decl}_1, \ldots, \mathit{label\text{-}decl}_n$ describe the labels associated to this record type. The label declaration *label-name* `:` *typexpr* declares *label-name* as a label whose argument has type *typexpr*. The label declaration `mutable` *label-name* `:` *typexpr* behaves similarly; in addition, it allows physical modification over the argument to this label.

The two components of a type definition, the optional equation and the optional representation, can be combined independently, giving rise to four typical situations:

**Abstract type: no equation, no representation.**
When appearing in a module signature, this definition specifies nothing on the type constructor, besides its number of parameters: its representation is hidden and it is assumed incompatible with any other type.

**Type abbreviation: an equation, no representation.**
This defines the type constructor as an abbreviation for the type expression on the right of the `=` sign.

**New variant type or record type: no equation, a representation.**
This generates a new type constructor and defines associated constructors or labels, through which values of that type can be directly built or inspected.

**Re-exported variant type or record type: an equation, a representation.**
In this case, the type constructor is defined as an abbreviation for the type expression given

in the equation, but in addition the constructors or labels given in the representation remain attached to the defined type constructor. The type expression in the equation part must agree with the representation: it must be of the same kind (record or variant) and have exactly the same constructors or labels, in the same order, with the same arguments.

The construct `constraint ' ` *ident* `=` *typexpr* allows to specify type parameters. Any actual type argument corresponding to the type parameter *ident* have to be an instance of *typexpr* (more precisely, *ident* and *typexpr* are unified). Type variables of *typexpr* can appear in the type equation and the type declaration.

### 5.8.2 Exception definitions

*exception-definition* ::= `exception` *constr-decl*

Exception definitions add new constructors to the built-in variant type `exn` of exception values. The constructors are declared as for a definition of a variant type.

## 5.9 Classes

Classes are defined using a small language, similar to the module language.

### 5.9.1 Class types

Class types are the class-level equivalent of type expressions: they specify the general shape and type properties of classes.

```
      class-type ::=
                   | class-body-type
                   | typexpr -> class-type
 class-body-type ::= object [( typexpr )] {class-field-spec} end
                   | class-path
                   | [ typexpr {, typexpr} ] class-path
class-field-spec ::= inherit class-type
                   | val [mutable] inst-var-name : typexpr
                   | method [private] method-name : typexpr
                   | method [private] virtual method-name : typexpr
                   | constraint typexpr = typexpr
```

#### Simple class expressions

The expression *class-path* is equivalent to the class type bound to the name *class-path*. Similarly, the expression `[` $typexpr_1$ `, ...` $typexpr_n$ `]` *class-path* is equivalent to the parametric class type bound to the name *class-path*, in which type parameters have been instanciated to respectively $typexpr_1$, … $typexpr_n$.

### Class function type

The class type expression *typexpr* `->` *class-type* is the type of class functions (functions from values to classes) that take as argument a value of type *typexpr* and return as result a class of type *class-type*.

### Class body type

The class type expression `object` [`(` *typexpr* `)`] {*class-field-spec*} `end` is the type of a class body. It specifies its instance variables and methods. In this type, *typexpr* is match agains self type, therefore provinding a binding for self type.

A class body will match a class body type if it provides definitions for all the components specified in the class type, and these definitions meet the type requirements given in the class type. Furthermore, all methods either virtual or public present in the class body must also be present in the class type (on the other hand, some instance variables and concrete private methods may be omitted). A virtual method will match a concrete method, thus allowing to forget its implementation. An immutable instance variable will match a mutable instance variable.

### Inheritance

The inheritance construct `inherit` *class-type* allows to include methods and instance variables from other classes types. The instance variable and method types from this class type are added into the current class type.

### Instance variable specification

A specification of an instance variable is written `val` [`mutable`] *inst-var-name* `:` *typexpr*, where *inst-var-name* is the name of the instance variable and *typexpr* its expected type. The flag `mutable` indicates whether this instance variable can be physically modified.

An instance variable specification will hide any previous specification of an instance variable of the same name.

### Method specification

A specification of an method is written `method` [`private`] *method-name* `:` *typexpr*, where *method-name* is the name of the method and *typexpr* its expected type. The flag `private` indicates whether the method can be accessed from outside the class.

Several specification for the same method must have compatible types.

### Virtual method specification

Virtual method specification is written `method` [`private`] `virtual` *method-name* `:` *typexpr*, where *method-name* is the name of the method and *typexpr* its expected type.

### Constraints on type parameters

The construct `constraint` *typexpr*$_1$ `=` *typexpr*$_2$ forces the two type expressions to be equals. This is typically used to specify type parameters: they can be that way be bound to a specified type expression.

## 5.9.2 Class expressions

Class expressions are the class-level equivalent of value expressions: they evaluate to classes, thus providing implementations for the specifications expressed in class types.

```
class-expr  ::=  class-path
              |  [ typexpr {, typexpr} ] class-path
              |  ( class-expr )
              |  ( class-expr : class-type )
              |  class-expr expr
              |  fun {pattern}+ -> class-expr
              |  let [rec] let-binding {and let-binding} in class-expr
              |  object [( pattern [: typexpr] )] {class-field} end

class-field ::=  inherit class-expr [as value-name]
              |  val [mutable] inst-var-name = expr
              |  method [private] method-name {pattern} = expr
              |  method [private] virtual method-name : typexpr
              |  constraint typexpr = typexpr
              |  initializer expr
```

### Simple class expressions

The expression *class-path* evaluates to the class bound to the name *class-path*. Similarly, the expression `[` *typexpr*$_1$ `,` ... *typexpr*$_n$ `]` *class-path* evaluates to the parametric class bound to the name *class-path*, in which type parameters have been instanciated to respectively *typexpr*$_1$, ... *typexpr*$_n$.

The expression `(` *class-expr* `)` evaluates to the same module as *class-expr*.

The expression `(` *class-expr* `:` *class-type* `)` checks that *class-type* match the type of *class-expr* (that is, that the implementation *class-expr* meets the type specification *class-type*). The whole expression evaluates to the same class as *class-expr*, except that all components not specified in *class-type* are hidden and can no longer be accessed.

### Class application

Class application is denoted by juxtaposition of expressions. The expression *class-expr expr* evaluates the expressions *class-expr* and *expr*. The expression *class-expr* must evaluate to a functional class value, which is then applied to the value of *expr*.

### Class function

The expression `fun` *pattern* `->` *class-expr* evaluates to a function from values to classes. When this function is applied to a value $v$, this value is matched against the pattern *pattern* and the result is the result of the evaluation of *class-expr* in the extended environment.

The expression

$$\texttt{fun}\ \mathit{pattern}_1 \ldots \mathit{pattern}_n\ \texttt{->}\ \mathit{class\text{-}expr}$$

is a short form for

$$\texttt{fun}\ \mathit{pattern}_1\ \texttt{->} \ldots \texttt{fun}\ \mathit{pattern}_n\ \texttt{->}\ \mathit{expr}$$

### Local definitions

The `let` and `let rec` constructs bind value names locally, as for the core language expressions.

### Class body

The expression `object (` *pattern* [`:` *typexpr*] `)` {*class-field*} `end` denotes a class body. This is the prototype for an object : it lists the instance variables and methods of an objet of this class.

A class body is a class value: it is not evaluated at once. Rather, its components are evaluated each time an object is created.

In a class body, the pattern `(` *pattern* [`:` *typexpr*] `)` is matched against self, therefore provinding a binding for self and self type. Self can only be used in method and initializers.

Self type cannot be a closed object type, so that the class remains extensible.

### Inheritance

The inheritance construct `inherit` *class-expr* allows to reuse methods and instance variables from other classes. The class expression *class-expr* must evaluate to a class body. The instance variables, methods and initializers from this class body are added into the current class. The addition of a method will override any previously defined methods of the same name.

An ancestor can be bound by prepending the construct `as` *value-name* to the inheritance construct above. *value-name* is not a true variable and can only be used to select a method, i.e. in an expression *value-name* `#` *method-name*. This gives access to the method *method-name* as it was defined in the parent class even if it is redefined in the current class.

### Instance variable definition

The definition `val` [`mutable`] *inst-var-name* `=` *expr* adds an instance variable *inst-var-name* whose initial value is the value of expression *expr*. Several variables of the same name can be defined in the same class. The flag `mutable` allows physical modification of this variable by methods.

An instance variables can only be used in the following methods and initializers of the class.

### Method definition

Method definition is written `method` *method-name* `=` *expr*. The definition of a method overrides any previous definition of this method. The method will be public (that is, not private) if any of the definition states so.

A private method, `method private` *method-name* `=` *expr*, is a method that can only be invoked on self (from other methods of the current class as well as of subclasses of the current class). This invocation is performed using the expression *value-name* `#` *method-name*, where *value-name* is directly bound to self at the beginning of the class definition. Private methods do not appear in object types.

Some special expressions are available in method bodies for manipulating instance variables and duplicating self:

| | | |
|---|---|---|
| *expr* | ::= | ... |
| | \| | *inst-var-name* `<-` *expr* |
| | \| | `{<` [*inst-var-name* `=` *expr* {`;` *inst-var-name* `=` *expr*}] `>}` |

The expression *inst-var-name* `<-` *expr* modifies in-place the current object by replacing the value associated to *inst-var-name* by the value of *expr*. Of course, this instance variable must have been declared mutable.

The expression `{<` [*inst-var-name* `=` *expr* {`;` *inst-var-name* `=` *expr*}] `>}` evaluates to a copy of the current object in which the values of instance variables $\textit{inst-var-name}_1, \ldots, \textit{inst-var-name}_n$ have been replaced by the values of the corresponding expressions $\textit{expr}_1, \ldots, \textit{expr}_n$.

### Virtual method definition

Method specification is written `method` [`private`] `virtual` *method-name* `:` *typexpr*. It specifies whether the method is public or private, and gives its type.

### Constraints on type parameters

The construct `constraint` $\textit{typexpr}_1$ `=` $\textit{typexpr}_2$ forces the two type expressions to be equals. This is typically used to specify type parameters: they can be that way be bound to a specified type expression.

### Initializers

A class initializer `initializer` *expr* specifies an expression that will be evaluated when an object will be created from the class, once all the instance variables have been initialized.

## 5.9.3 Class definitions

| | | |
|---|---|---|
| *class-definition* | ::= | `class` *class-binding* {`and` *class-binding*} |
| *class-binding* | ::= | [`virtual`] [`[` *type-parameters* `]`] *class-name* {*pattern*} [`:` *class-type*] `=` *class-expr* |
| *type-parameters* | ::= | `'` *ident* {`,` `'` *ident*} |

A class definition `class` *class-binding* {`and` *class-binding*} is recursive. Each *class-binding* defines a *class-name* that can be used in the whole expression except for inheritance. It can also be used for inheritance, but only in the definitions that follow its own.

A class binding binds the class name *class-name* to the value of expression *class-expr*. It also binds the class type *class-name* to the type of the class, and defines two type abbreviations : *class-name* and `#` *class-name*. The first one is the type of objects of this class, while the second is more general as it unifies with the type of any object belonging to a subclass (see section 5.4).

#### Virtual class

A class must be flagged virtual if one of its methods is virtual (that is, appears in the class type, but is not actually defined). Objects cannot be created from a virtual class.

#### Type parameters

The class type parameters correspond to the ones of the class type and of the two type abbreviations defined by the class binding. They must be bound to actual types in the class definition using type constraints. So that the abbreviations are well-formed, type variables of the inferred type of the class must either be type parameters or be bound in the constraint clause.

### 5.9.4 Class specification

| | | |
|---|---|---|
| *class-specification* | ::= | `class` *class-spec* {`and` *class-spec*} |
| *class-spec* | ::= | [`virtual`] [`[` *type-parameters* `]`] *class-name* `:` *class-type* |

This is the counterpart in signatures of class definitions. A class specification matches a class definition if they have the same type parameters and their types match.

### 5.9.5 Class type definitions

| | | |
|---|---|---|
| *classtype-definition* | ::= | `class type` *classtype-def* {`and` *classtype-def*} |
| *classtype-def* | ::= | [`virtual`] [`[` *type-parameters* `]`] *class-name* `=` *class-body-type* |

A class type definition `class` *class-name* `=` *class-body-type* defines an abbreviation *class-name* for the class body type *class-body-type*. As for class definitions, two type abbreviations *class-name* and `#` *class-name* are also defined. The definition can be parameterized by some type parameters. If any method in the class type body is virtual, the definition must be flagged *virtual*.

Two class type definitions match if they have the same type parameters and the types they expand to match.

## 5.10 Module types (module specifications)

Module types are the module-level equivalent of type expressions: they specify the general shape and type properties of modules.

```
  module-type ::= modtype-path
                | sig {specification [;;]} end
                | functor ( module-name : module-type ) -> module-type
                | module-type with constraint {and constraint}
                | ( module-type )

specification ::= val value-name : typexpr
                | external value-name : typexpr = external-declaration
                | type-definition
                | exception-definition
                | class-specification
                | classtype-definition
                | module module-name : module-type
                | module module-name {( module-name : module-type )} : module-type
                | module type modtype-name
                | module type modtype-name = module-type
                | open module-path
                | include modtype-path

   constraint ::= type [type-parameters] typeconstr = typexp
                | module module-path = extended-module-path
```

### 5.10.1 Simple module types

The expression *modtype-path* is equivalent to the module type bound to the name *modtype-path*. The expression `(` *module-type* `)` denotes the same type as *module-type*.

### 5.10.2 Signatures

Signatures are type specifications for structures. Signatures `sig`...`end` are collections of type specifications for value names, type names, exceptions, module names and module type names. A structure will match a signature if the structure provides definitions (implementations) for all the names specified in the signature (and possibly more), and these definitions meet the type requirements given in the signature.

For compatibility with Caml Light, an optional `;;` is allowed after each specification in a signature. The `;;` has no semantic meaning.

#### Value specifications

A specification of a value component in a signature is written `val` *value-name* `:` *typexpr*, where *value-name* is the name of the value and *typexpr* its expected type.

The form `external` *value-name* `:` *typexpr* `=` *external-declaration* is similar, except that it requires in addition the name to be implemented as the external function specified in *external-declaration* (see chapter 15).

### Type specifications

A specification of one or several type components in a signature is written `type` *typedef* {`and` *typedef*} and consists of a sequence of mutually recursive definitions of type names.

Each type definition in the signature specifies an optional type equation `=` *typexp* and an optional type representation `=` *constr-decl*... or `= {` *label-decl*...`}`. The implementation of the type name in a matching structure must be compatible with the type expression specified in the equation (if given), and have the specified representation (if given). Conversely, users of that signature will be able to rely on the type equation or type representation, if given. More precisely, we have the following four situations:

**Abstract type: no equation, no representation.**
Names that are defined as abstract types in a signature can be implemented in a matching structure by any kind of type definition (provided it has the same number of type parameters). The exact implementation of the type will be hidden to the users of the structure. In particular, if the type is implemented as a variant type or record type, the associated constructors and labels will not be accessible to the users; if the type is implemented as an abbreviation, the type equality between the type name and the right-hand side of the abbreviation will be hidden from the users of the structure. Users of the structure consider that type as incompatible with any other type: a fresh type has been generated.

**Type abbreviation: an equation `=` *typexp*, no representation.**
The type name must be implemented by a type compatible with *typexp*. All users of the structure know that the type name is compatible with *typexp*.

**New variant type or record type: no equation, a representation.**
The type name must be implemented by a variant type or record type with exactly the constructors or labels specified. All users of the structure have access to the constructors or labels, and can use them to create or inspect values of that type. However, users of the structure consider that type as incompatible with any other type: a fresh type has been generated.

**Re-exported variant type or record type: an equation, a representation.**
This case combines the previous two: the representation of the type is made visible to all users, and no fresh type is generated.

### Exception specification

The specification `exception` *constr-decl* in a signature requires the matching structure to provide an exception with the name and arguments specified in the definition, and makes the exception available to all users of the structure.

**Class specifications**

A specification of one or several classes in a signature is written `class` *class-spec* {`and` *class-spec*} and consists of a sequence of mutually recursive definitions of class names.

Class specifications are described more precisely in section 5.9.4.

**Class type specifications**

A specification of one or several classe types in a signature is written `class type` *classtype-def* {`and` *classtype-de* and consists of a sequence of mutually recursive definitions of class type names. Class type specifications are described more precisely in section 5.9.5.

**Module specifications**

A specification of a module component in a signature is written `module` *module-name* `:` *module-type*, where *module-name* is the name of the module component and *module-type* its expected type. Modules can be nested arbitrarily; in particular, functors can appear as components of structures and functor types as components of signatures.

For specifying a module component that is a functor, one may write

`module` *module-name* `(` $name_1$ `:` $module\text{-}type_1$ `)` ... `(` $name_n$ `:` $module\text{-}type_n$ `)` `:` *module-type*

instead of

`module` *module-name* `:` `functor` `(` $name_1$ `:` $module\text{-}type_1$ `)` `->` ... `->` *module-type*

**Module type specifications**

A module type component of a signature can be specified either as a manifest module type or as an abstract module type.

An abstract module type specification `module type` *modtype-name* allows the name *modtype-name* to be implemented by any module type in a matching signature, but hides the implementation of the module type to all users of the signature.

A manifest module type specification `module type` *modtype-name* `=` *module-type* requires the name *modtype-name* to be implemented by the module type *module-type* in a matching signature, but makes the equality between *modtype-name* and *module-type* apparent to all users of the signature.

**Opening a module path**

The expression `open` *module-path* in a signature does not specify any components. It simply affects the parsing of the following items of the signature, allowing components of the module denoted by *module-path* to be referred to by their simple names *name* instead of path accesses *module-path* `.` *name*. The scope of the `open` stops at the end of the signature expression.

**Including a signature**

The expression `include` *modtype-path* in a signature performs textual inclusion of the components of the signature denoted by *modtype-path*. It behaves as if the components of the included signature were copied at the location of the `include`. The *modtype-path* argument must refer to a module type that is a signature,

### 5.10.3 Functor types

The module type expression `functor (` *module-name* `:` *module-type*$_1$ `) ->` *module-type*$_2$ is the type of functors (functions from modules to modules) that take as argument a module of type *module-type*$_1$ and return as result a module of type *module-type*$_2$. The module type *module-type*$_2$ can use the name *module-name* to refer to type components of the actual argument of the functor. No restrictions are placed on the type of the functor argument; in particular, a functor may take another functor as argument ("higher-order" functor).

### 5.10.4 The `with` operator

Assuming *module-type* denotes a signature, the expression *module-type* `with` *constraint* {`and` *constraint*} denotes the same signature where type equations have been added to some of the type specifications, as described by the constraints following the `with` keyword. The constraint `type` [*type-parameters*] *typeconstr* `=` *typexp* adds the type equation `=` *typexp* to the specification of the type component named *typeconstr* of the constrained signature. The constraint `module` *module-path* `=` *extended-module-path* adds type equations to all type components of the sub-structure denoted by *module-path*, making them equivalent to the corresponding type components of the structure denoted by *extended-module-path*.

For instance, if the module type name `S` is bound to the signature

```
        sig type t module M: (sig type u end) end
```

then `S with type t=int` denotes the signature

```
        sig type t=int module M: (sig type u end) end
```

and `S with module M = N` denotes the signature

```
        sig type t module M: (sig type u=N.u end) end
```

A functor taking two arguments of type `S` that share their `t` component is written

```
        functor (A: S) (B: S with type t = A.t) ...
```

## 5.11 Module expressions (module implementations)

Module expressions are the module-level equivalent of value expressions: they evaluate to modules, thus providing implementations for the specifications expressed in module types.

```
module-expr ::= module-path
              | struct {definition [;;]} end
              | functor ( module-name : module-type ) -> module-expr
              | module-expr ( module-expr )
              | ( module-expr )
              | ( module-expr : module-type )

 definition ::= let [rec] let-binding {and let-binding}
              | external value-name : typexpr = external-declaration
              | type-definition
              | exception-definition
              | class-definition
              | classtype-definition
              | module module-name {( module-name : module-type )} [: module-type] = module-expr
              | module type modtype-name = module-type
              | open module-path
```

## 5.11.1 Simple module expressions

The expression *module-path* evaluates to the module bound to the name *module-path*.

The expression `(` *module-expr* `)` evaluates to the same module as *module-expr*.

The expression `(` *module-expr* `:` *module-type* `)` checks that the type of *module-expr* is a subtype of *module-type*, that is, that all components specified in *module-type* are implemented in *module-expr*, and their implementation meets the requirements given in *module-type*. In other terms, it checks that the implementation *module-expr* meets the type specification *module-type*. The whole expression evaluates to the same module as *module-expr*, except that all components not specified in *module-type* are hidden and can no longer be accessed.

## 5.11.2 Structures

Structures `struct`...`end` are collections of definitions for value names, type names, exceptions, module names and module type names. The definitions are evaluated in the order in which they appear in the structure. The scope of the bindings performed by the definitions extend to the end of the structure. As a consequence, a definition may refer to names bound by earlier definitions in the same structure.

For compatibility with toplevel phrases (chapter 8) and with Caml Light, an optional `;;` is allowed after each definition in a structure. The `;;` has no semantic meaning. Also for compatibility, `;;` *expr* is allowed as a component of a structure, meaning `let _ =` *expr*, i.e. evaluate *expr* for its side-effects.

### Value definitions

A value definition `let` [`rec`] *let-binding* {`and` *let-binding*} bind value names in the same way as a `let`...`in`... expression (see section 5.7.1). The value names appearing in the left-hand sides of the bindings are bound to the corresponding values in the right-hand sides.

A value definition `external` *value-name* `:` *typexpr* `=` *external-declaration* implements *value-name* as the external function specified in *external-declaration* (see chapter 15).

### Type definitions

A definition of one or several type components is written `type` *typedef* {`and` *typedef*} and consists of a sequence of mutually recursive definitions of type names.

### Exception definitions

Exceptions are defined with the syntax `exception` *constr-decl*.

### Class definitions

A definition of one or several classes is written `class` *class-binding* {`and` *class-binding*} and consists of a sequence of mutually recursive definitions of class names. Class definitions are described more precisely in section 5.9.3.

### Class type definitions

A definition of one or several classes is written `class type` *classtype-def* {`and` *classtype-def*} and consists of a sequence of mutually recursive definitions of class type names. Class type definitions are described more precisely in section 5.9.5.

### Module definitions

The basic form for defining a module component is `module` *module-name* `=` *module-expr*, which evaluates *module-expr* and binds the result to the name *module-name*.

One can write

`module` *module-name* `:` *module-type* `=` *module-expr*

instead of

`module` *module-name* `=` `(` *module-expr* `:` *module-type* `)`.

Another derived form is

`module` *module-name* `(` *name*$_1$ `:` *module-type*$_1$ `)` ... `(` *name*$_n$ `:` *module-type*$_n$ `)` `=` *module-expr*

which is equivalent to

`module` *module-name* `=` `functor` `(` *name*$_1$ `:` *module-type*$_1$ `)` `->` ... `->` *module-expr*

### Module type definitions

A definition for a module type is written `module type` *modtype-name* `=` *module-type*. It binds the name *modtype-name* to the module type denoted by the expression *module-type*.

#### Opening a module path

The expression `open` *module-path* in a structure does not define any components nor perform any bindings. It simply affects the parsing of the following items of the structure, allowing components of the module denoted by *module-path* to be referred to by their simple names *name* instead of path accesses *module-path* `.` *name*. The scope of the `open` stops at the end of the structure expression.

### 5.11.3 Functors

#### Functor definition

The expression `functor (` *module-name* `:` *module-type* `) ->` *module-expr* evaluates to a functor that takes as argument modules of the type *module-type*$_1$, binds *module-name* to these modules, evaluates *module-expr* in the extended environment, and returns the resulting modules as results. No restrictions are placed on the type of the functor argument; in particular, a functor may take another functor as argument ("higher-order" functor).

#### Functor application

The expression *module-expr*$_1$ `(` *module-expr*$_2$ `)` evaluates *module-expr*$_1$ to a functor and *module-expr*$_2$ to a module, and applies the former to the latter. The type of *module-expr*$_2$ must match the type expected for the arguments of the functor *module-expr*$_1$.

## 5.12 Compilation units

| | | |
|---|---|---|
| *unit-interface* | ::= | {*specification* [`;;`]} |
| *unit-implementation* | ::= | {*definition* [`;;`]} |

Compilation units bridge the module system and the separate compilation system. A compilation unit is composed of two parts: an interface and an implementation. The interface contains a sequence of specifications, just as the inside of a `sig`...`end` signature expression. The implementation contains a sequence of definitions, just as the inside of a `struct`...`end` module expression. A compilation unit also has a name *unit-name*, derived from the names of the files containing the interface and the implementation (see chapter 7 for more details). A compilation unit behaves roughly as the module definition

`module` *unit-name* `: sig` *unit-interface* `end = struct` *unit-implementation* `end`

A compilation unit can refer to other compilation units by their names, as if they were regular modules. For instance, if `U` is a compilation unit that defines a type `t`, other compilation units can refer to that type under the name `U.t`; they can also refer to `U` as a whole structure. Except for names of other compilation units, a unit interface or unit implementation must not have any other free variables. In other terms, the type-checking and compilation of an interface or implementation proceeds in the initial environment

*name*$_1$ `: sig` *interface*$_1$ `end`... *name*$_n$ `: sig` *interface*$_n$ `end`

where *name*$_1$ ... *name*$_N$ are the names of the other compilation units available in the search path (see chapter 7 for more details) and *interface*$_1$ ... *interface*$_n$ are their respective interfaces.

# Chapter 6

# Language extensions

This chapter describes the language features that are implemented in Objective Caml, but not described in the Objective Caml reference manual. In contrast with the fairly stable kernel language that is described in the reference manual, the extensions presented here are still experimental, and may be removed or changed in the future.

## 6.1 Streams and stream parsers

Objective Caml comprises a library type for *streams* (possibly infinite sequences of elements, that are evaluated on demand), and associated stream expressions, to build streams, and stream patterns, to destructure streams. Streams and stream patterns provide a natural approach to the writing of recursive-descent parsers.

Streams are presented by the following extensions to the syntactic classes of expressions:

| | | |
|---|---|---|
| *expr* | ::= | ... |
| | \| | `[< >]` |
| | \| | `[<` *stream-component* {`;` *stream-component*} `>]` |
| | \| | `parser` [*pattern*] *stream-matching* |
| | \| | `match` *expr* `with parser` [*pattern*] *stream-matching* |
| *stream-component* | ::= | `'` *expr* |
| | \| | *expr* |
| *stream-matching* | ::= | *stream-pattern* [*pattern*] `->` *expr* {`\|` *stream-pattern* [*pattern*] `->` *expr*} |
| *stream-pattern* | ::= | `[< >]` |
| | \| | `[<` *stream-pat-comp* {`;` *stream-pat-comp* [`?` *expr*]} `>]` |
| *stream-pat-comp* | ::= | `'` *pattern* [`when` *expr*] |
| | \| | *pattern* `=` *expr* |
| | \| | *ident* |

Stream expressions are bracketed by `[<` and `>]`. They represent the concatenation of their components. The component `'` *expr* represents the one-element stream whose element is the value of *expr*. The component *expr* represents a sub-stream. For instance, if both `s` and `t` are streams of integers, then `[<'1; s; t; '2>]` is a stream of integers containing the element `1`, then the elements of `s`, then those of `t`, and finally `2`. The empty stream is denoted by `[< >]`.

Unlike any other kind of expressions in the language, stream expressions are submitted to lazy evaluation: the components are not evaluated when the stream is built, but only when they are accessed during stream matching. The components are evaluated once, the first time they are accessed; the following accesses reuse the value computed the first time.

Stream patterns, also bracketed by `[<` and `>]`, describe initial segments of streams. In particular, the stream pattern `[< >]` matches all streams. Stream pattern components are matched against the corresponding elements of a stream. The component `'` *pattern* matches the corresponding stream element against the pattern; if followed by *when*, the match is accepted only if the result of the guard expression is `true`. The component *pattern* `=` *expr* applies the function denoted by *expr* to the current stream, then matches the result of the function against *pattern*. Finally, the component *ident* simply binds the identifier to the stream being matched.

Stream matching proceeds destructively: once a component has been matched, it is discarded from the stream (by in-place modification).

Stream matching proceeds in two steps: first, a pattern is selected by matching the stream against the first components of the stream patterns; then, the following components of the selected pattern are checked against the stream. If the following components do not match, the exception `Stream.Error` is raised. There is no backtracking here: stream matching commits to the pattern selected according to the first element. If none of the first components of the stream patterns match, the exception `Stream.Failure` is raised. The `Stream.Parse_failure` exception causes the next alternative to be tried, if it occurs during the matching of the first element of a stream, before matching has committed to one pattern.

The streams hold the count of their elements discarded. The optional *pattern* before the first stream pattern is bound to the stream count before the matching. The one after each stream pattern (optional, too) is bound to the stream count after the matching.

The exception `Parse_error` has a string parameter coming from the optional `?` *expr* after the stream pattern components (its default is the empty string). This expression is evaluated only in case of error.

See *Functional programming using Caml Light* for a more gentle introductions to streams, and for some examples of their use in writing parsers. A more formal presentation of streams, and a discussion of alternate semantics, can be found in *Parsers in ML* by Michel Mauny and Daniel de Rauglaudre, in the proceedings of the 1992 ACM conference on Lisp and Functional Programming.

## 6.2 Range patterns

In patterns, Objective Caml recognizes the form `'` *c* `'` `..` `'` *d* `'` (two character literals separated by `..`) as shorthand for the pattern

$$' c ' \mid ' c_1 ' \mid ' c_2 ' \mid \ldots \mid ' c_n ' \mid ' d '$$

where $c_1$, $c_2$, …, $c_n$ are the characters that occur between *c* and *d* in the ASCII character set. For instance, the pattern `'0'..'9'` matches all characters that are digits.

## 6.3 Assertion checking

Objective Caml supports the `assert` construct to check debugging assertions. The expression `assert` *expr* evaluates the expression *expr* and returns `()` if *expr* evaluates to `true`. Otherwise, the exception `Assert_failure` is raised with the location of *expr* in the source code as argument. As a special case, `assert false` is reduced to `raise (Assert_failure ...)`, which is polymorphic. Assertion checking can be turned off with the `-noassert` compiler option.

## 6.4 Deferred computations

The expression `lazy` *expr* returns a value $v$ of type `Lazy.t` that encapsulates the computation of *expr*. The argument *expr* is not evaluated at this point in the program. Instead, its evaluation will be performed the first time `Lazy.force` is applied to the value $v$, returning the actual value of *expr*. Subsequent applications of `Lazy.force` to $v$ do not evaluate *expr* again.

The expression `lazy` *expr* is equivalent to `ref (Lazy.Delayed (fun () ->` *expr*`))`. For more information, see the description of module `Lazy` in the standard library (section 17.12).

## 6.5 Record copy with update

The expression `{` *expr* `with` $lbl_1$ `=` $expr_1$ `;` … `;` $lbl_n$ `=` $expr_n$ `}` builds a fresh record with fields $lbl_1 \ldots lbl_n$ equal to $expr_1 \ldots expr_n$, and all other fields having the same value as in the record *expr*. In other terms, it returns a shallow copy of the record *expr*, except for the fields $lbl_1 \ldots lbl_n$, which are initialized to $expr_1 \ldots expr_n$. For example:

```
type point = { x : float; y : float; z : float }
let proj p = { p with x = 0.0 }
let proj p = { x = 0.0; y = p.y; z = p.z }
```

The two definitions of `proj` above are equivalent.

## 6.6 Local modules

The expression `let module` *module-name* `=` *module-expr* `in` *expr* locally binds the module expression *module-expr* to the identifier *module-name* during the evaluation of the expression *expr*. It then returns the value of *expr*. For example:

```
let remove_duplicates comparison_fun string_list =
  let module StringSet =
    Set.Make(struct type t = string
                    let compare = comparison_fun end) in
  StringSet.elements
    (List.fold_right StringSet.add string_list StringSet.empty)
```

# Part III

# The Objective Caml tools

# Chapter 7

# Batch compilation (ocamlc)

This chapter describes the Objective Caml batch compiler `ocamlc`, which compiles Caml source files to bytecode object files and link these object files to produce standalone bytecode executable files. These executable files are then run by the bytecode interpreter `ocamlrun`.

## 7.1 Overview of the compiler

The `ocamlc` command has a command-line interface similar to the one of most C compilers. It accepts several types of arguments:

- Arguments ending in `.mli` are taken to be source files for compilation unit interfaces. Interfaces specify the names exported by compilation units: they declare value names with their types, define public data types, declare abstract data types, and so on. From the file *x*`.mli`, the `ocamlc` compiler produces a compiled interface in the file *x*`.cmi`.

- Arguments ending in `.ml` are taken to be source files for compilation unit implementations. Implementations provide definitions for the names exported by the unit, and also contain expressions to be evaluated for their side-effects. From the file *x*`.ml`, the `ocamlc` compiler produces compiled object bytecode in the file *x*`.cmo`.

  If the interface file *x*`.mli` exists, the implementation *x*`.ml` is checked against the corresponding compiled interface *x*`.cmi`, which is assumed to exist. If no interface *x*`.mli` is provided, the compilation of *x*`.ml` produces a compiled interface file *x*`.cmi` in addition to the compiled object code file *x*`.cmo`. The file *x*`.cmi` produced corresponds to an interface that exports everything that is defined in the implementation *x*`.ml`.

- Arguments ending in `.cmo` are taken to be compiled object bytecode. These files are linked together, along with the object files obtained by compiling `.ml` arguments (if any), and the Caml Light standard library, to produce a standalone executable program. The order in which `.cmo` and `.ml` arguments are presented on the command line is relevant: compilation units are initialized in that order at run-time, and it is a link-time error to use a component of a unit before having initialized it. Hence, a given *x*`.cmo` file must come before all `.cmo` files that refer to the unit *x*.

- Arguments ending in `.cma` are taken to be libraries of object bytecode. A library of object bytecode packs in a single file a set of object bytecode files (`.cmo` files). Libraries are built with `ocamlc -a` (see the description of the `-a` option below). The object files contained in the library are linked as regular `.cmo` files (see above), in the order specified when the `.cma` file was built. The only difference is that if an object file contained in a library is not referenced anywhere in the program, then it is not linked in.
- Arguments ending in `.c` are passed to the C compiler, which generates a `.o` object file. This object file is linked with the program if the `-custom` flag is set (see the description of `-custom` below).
- Arguments ending in `.o` or `.a` are assumed to be C object files and libraries. They are passed to the C linker when linking in `-custom` mode (see the description of `-custom` below).

The output of the linking phase is a file containing compiled bytecode that can be executed by the Objective Caml bytecode interpreter: the command named `ocamlrun`. If `caml.out` is the name of the file produced by the linking phase, the command

`ocamlrun caml.out` $arg_1$ $arg_2$ `...` $arg_n$

executes the compiled code contained in `caml.out`, passing it as arguments the character strings $arg_1$ to $arg_n$. (See chapter 9 for more details.)

On most Unix systems, the file produced by the linking phase can be run directly, as in:

`./caml.out` $arg_1$ $arg_2$ `...` $arg_n$

The produced file has the executable bit set, and it manages to launch the bytecode interpreter by itself.

## 7.2 Options

The following command-line options are recognized by `ocamlc`.

`-a` Build a library (`.cma` file) with the object files (`.cmo` files) given on the command line, instead of linking them into an executable file. The name of the library can be set with the `-o` option. The default name is `library.cma`.

`-c` Compile only. Suppress the linking phase of the compilation. Source code files are turned into compiled files, but no executable file is produced. This option is useful to compile modules separately.

`-cc` *ccomp*
Use *ccomp* as the C linker called by `ocamlc -custom` and as the C compiler for compiling `.c` source files.

`-cclib -l`*libname*
Pass the `-l`*libname* option to the C linker when linking in "custom runtime" mode (see the `-custom` option). This causes the given C library to be linked with the program.

`-ccopt` *option*
: Pass the given option to the C compiler and linker, when linking in "custom runtime" mode (see the `-custom` option). For instance, `-ccopt -L`*dir* causes the C linker to search for C libraries in directory *dir*.

`-custom`
: Link in "custom runtime" mode. In the default linking mode, the linker produces bytecode that is intended to be executed with the shared runtime system, `ocamlrun`. In the custom runtime mode, the linker produces an output file that contains both the runtime system and the bytecode for the program. The resulting file is larger, but it can be executed directly, even if the `ocamlrun` command is not installed. Moreover, the "custom runtime" mode enables linking Caml code with user-defined C functions, as described in chapter 15.

`-g`
: Add debugging information while compiling and linking. This option is required in order to be able to debug the program with `ocamldebug` (see chapter 13).

`-i`
: Cause the compiler to print all defined names (with their inferred types or their definitions) when compiling an implementation (`.ml` file). This can be useful to check the types inferred by the compiler. Also, since the output follows the syntax of interfaces, it can help in writing an explicit interface (`.mli` file) for a file: just redirect the standard output of the compiler to a `.mli` file, and edit that file to remove all declarations of unexported names.

`-I` *directory*
: Add the given directory to the list of directories searched for compiled interface files (`.cmi`) and compiled object code files (`.cmo`). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

`-impl` *filename*
: Compile the file *filename* as an implementation file, even if its extension is not `.ml`.

`-intf` *filename*
: Compile the file *filename* as an interface file, even if its extension is not `.mli`.

`-linkall`
: Force all modules contained in libraries to be linked in. If this flag is not given, unreferenced modules are not linked in. When building a library (`-a` flag), setting the `-linkall` flag forces all subsequent links of programs involving that library to link all the modules contained in the library.

`-make-runtime`
: Build a custom runtime system (in the file specified by option `-o`) incorporating the C object files and libraries given on the command line. This custom runtime system can be used later to execute bytecode executables produced with the `ocamlc -use-runtime` *runtime-name* option. See section 15.1.4 for more information.

`-noassert`
: Turn assertion checking off: assertions are not compiled. This flag has no effect when linking already compiled files.

`-o` *exec-file*
: Specify the name of the output file produced by the linker. The default output name is `a.out`, in keeping with the Unix tradition. If the `-a` option is given, specify the name of the library produced. If the `-output-obj` option is given, specify the name of the output file produced.

`-output-obj`
: Cause the linker to produce a C object file instead of a bytecode executable file. This is useful to wrap Caml code as a C library, callable from any C program. See chapter 15, section 15.6.5. The name of the output object file is `camlprog.o` by default; it can be set with the `-o` option.

`-pp` *command*
: Cause the compiler to call the given *command* as a preprocessor for each source file. The output of *command* is redirected to an intermediate file, which is compiled. If there are no compilation errors, the intermediate file is deleted afterwards. The name of this file is built from the basename of the source file with the extension `.ppi` for an interface (`.mli`) file and `.ppo` for an implementation (`.ml`) file.

`-thread`
: Compile or link multithreaded programs, in combination with the `threads` library described in chapter 21. What this option actually does is select a special, thread-safe version of the standard library.

`-unsafe`
: Turn bound checking off on array and string accesses (the `v.(i)` and `s.[i]` constructs). Programs compiled with `-unsafe` are therefore slightly faster, but unsafe: anything can happen if the program accesses an array or string outside of its bounds.

`-use-runtime` *runtime-name*
: Generate a bytecode executable file that can be executed on the custom runtime system *runtime-name*, built earlier with `ocamlc -make-runtime` *runtime-name*. See section 15.1.4 for more information.

`-v`
: Print the version number of the compiler.

`-w` *warning-list*
: Enable or disable warnings according to the argument *warning-list*. The argument is a string of one or several characters, with the following meaning for each character:

  `A/a` enable/disable all warnings

  `F/f` enable/disable warnings for partially applied functions (i.e. `f x;` *expr* where the application `f x` has a function type).

  `M/m` enable/disable warnings for overriden methods.

  `P/p` enable/disable warnings for partial matches (missing cases in pattern matchings).

`S/s` enable/disable warnings for statements that do not have type `unit` (e.g. *expr1*`;` *expr2* when *expr1* does not have type `unit`).

`U/u` enable/disable warnings for unused (redundant) match cases.

`V/v` enable/disable warnings for hidden instance variables.

`X/x` enable/disable all other warnings.

The default setting is `-w A` (all warnings enabled).

## 7.3 Modules and the file system

This short section is intended to clarify the relationship between the names of the modules corresponding to compilation units and the names of the files that contain their compiled interface and compiled implementation.

The compiler always derives the module name by taking the capitalized base name of the source file (`.ml` or `.mli` file). That is, it strips the leading directory name, if any, as well as the `.ml` or `.mli` suffix; then, it set the first letter to uppercase, in order to comply with the requirement that module names must be capitalized. For instance, compiling the file `mylib/misc.ml` provides an implementation for the module named `Misc`. Other compilation units may refer to components defined in `mylib/misc.ml` under the names `Misc.`*name*; they can also do `open Misc`, then use unqualified names *name*.

The `.cmi` and `.cmo` files produced by the compiler have the same base name as the source file. Hence, the compiled files always have their base name equal (modulo capitalization of the first letter) to the name of the module they describe (for `.cmi` files) or implement (for `.cmo` files).

When the compiler encounters a reference to a free module identifier `Mod`, it looks in the search path for a file `mod.cmi` (note lowercasing of first letter) and loads the compiled interface contained in that file. As a consequence, renaming `.cmi` files is not advised: the name of a `.cmi` file must always correspond to the name of the compilation unit it implements. It is admissible to move them to another directory, if their base name is preserved, and the correct `-I` options are given to the compiler. The compiler will flag an error if it loads a `.cmi` file that has been renamed.

Compiled bytecode files (`.cmo` files), on the other hand, can be freely renamed once created. That's because the linker never attempts to find by itself the `.cmo` file that implements a module with a given name: it relies instead on the user providing the list of `.cmo` files by hand.

## 7.4 Common errors

This section describes and explains the most frequently encountered error messages.

**Cannot find file** *filename*

The named file could not be found in the current directory, nor in the directories of the search path. The *filename* is either a compiled interface file (`.cmi` file), or a compiled bytecode file (`.cmo` file). If *filename* has the format *mod*`.cmi`, this means you are trying to compile a file that references identifiers from module *mod*, but you have not yet compiled an interface for module *mod*. Fix: compile *mod*`.mli` or *mod*`.ml` first, to create the compiled interface *mod*`.cmi`.

If *filename* has the format *mod*`.cmo`, this means you are trying to link a bytecode object file that does not exist yet. Fix: compile *mod*`.ml` first.

If your program spans several directories, this error can also appear because you haven't specified the directories to look into. Fix: add the correct `-I` options to the command line.

**Corrupted compiled interface** *filename*

The compiler produces this error when it tries to read a compiled interface file (`.cmi` file) that has the wrong structure. This means something went wrong when this `.cmi` file was written: the disk was full, the compiler was interrupted in the middle of the file creation, and so on. This error can also appear if a `.cmi` file is modified after its creation by the compiler. Fix: remove the corrupted `.cmi` file, and rebuild it.

**This expression has type $t_1$, but is used with type $t_2$**

This is by far the most common type error in programs. Type $t_1$ is the type inferred for the expression (the part of the program that is displayed in the error message), by looking at the expression itself. Type $t_2$ is the type expected by the context of the expression; it is deduced by looking at how the value of this expression is used in the rest of the program. If the two types $t_1$ and $t_2$ are not compatible, then the error above is produced.

In some cases, it is hard to understand why the two types $t_1$ and $t_2$ are incompatible. For instance, the compiler can report that "expression of type `foo` cannot be used with type `foo`", and it really seems that the two types `foo` are compatible. This is not always true. Two type constructors can have the same name, but actually represent different types. This can happen if a type constructor is redefined. Example:

```
type foo = A | B
let f = function A -> 0 | B -> 1
type foo = C | D
f C
```

This result in the error message "expression `C` of type `foo` cannot be used with type `foo`".

**The type of this expression, $t$, contains type variables that cannot be generalized**

Type variables (`'a`, `'b`, ...) in a type $t$ can be in either of two states: generalized (which means that the type $t$ is valid for all possible instantiations of the variables) and not generalized (which means that the type $t$ is valid only for one instantiation of the variables). In a `let` binding `let` *name* `=` *expr*, the type-checker normally generalizes as many type variables as possible in the type of *expr*. However, this leads to unsoundness (a well-typed program can crash) in conjunction with polymorphic mutable data structures. To avoid this, generalization is performed at `let` bindings only if the bound expression *expr* belongs to the class of "syntactic values", which includes constants, identifiers, functions, tuples of syntactic values, etc. In all other cases (for instance, *expr* is a function application), a polymorphic mutable could have been created and generalization is therefore turned off.

Non-generalized type variables in a type cause no difficulties inside a given structure or compilation unit (the contents of a `.ml` file, or an interactive session), but they cannot be allowed inside signatures nor in compiled interfaces (`.cmi` file), because they could be used inconsistently later. Therefore, the compiler flags an error when a structure or compilation unit defines a value *name* whose type contains non-generalized type variables. There are two ways to fix this error:

- Add a type constraint or a `.mli` file to give a monomorphic type (without type variables) to *name*. For instance, instead of writing

  ```
  let sort_int_list = Sort.list (<)
  (* inferred type 'a list -> 'a list, with 'a not generalized *)
  ```

  write

  ```
  let sort_int_list = (Sort.list (<) : int list -> int list);;
  ```

- If you really need *name* to have a polymorphic type, turn its defining expression into a function by adding an extra parameter. For instance, instead of writing

  ```
  let map_length = List.map Array.length
  (* inferred type 'a array list -> int list, with 'a not generalized *)
  ```

  write

  ```
  let map_length lv = List.map Array.length lv
  ```

**Reference to undefined global** *mod*

This error appears when trying to link an incomplete or incorrectly ordered set of files. Either you have forgotten to provide an implementation for the compilation unit named *mod* on the command line (typically, the file named *mod*`.cmo`, or a library containing that file). Fix: add the missing `.ml` or `.cmo` file to the command line. Or, you have provided an implementation for the module named *mod*, but it comes too late on the command line: the implementation of *mod* must come before all bytecode object files that reference *mod*. Fix: change the order of `.ml` and `.cmo` files on the command line.

Of course, you will always encounter this error if you have mutually recursive functions across modules. That is, function `Mod1.f` calls function `Mod2.g`, and function `Mod2.g` calls function `Mod1.f`. In this case, no matter what permutations you perform on the command line, the program will be rejected at link-time. Fixes:

- Put `f` and `g` in the same module.
- Parameterize one function by the other. That is, instead of having

  ```
  mod1.ml:    let f x = ... Mod2.g ...
  mod2.ml:    let g y = ... Mod1.f ...
  ```

  define

  ```
  mod1.ml:    let f g x = ... g ...
  mod2.ml:    let rec g y = ... Mod1.f g ...
  ```

  and link `mod1.cmo` before `mod2.cmo`.

- Use a reference to hold one of the two functions, as in :

```
mod1.ml:    let forward_g =
                ref((fun x -> failwith "forward_g") : <type>)
            let f x = ... !forward_g ...
mod2.ml:    let g y = ... Mod1.f ...
            let _ = Mod1.forward_g := g
```

This will not work if `g` is a polymorphic function, however.

**The external function *f* is not available**

This error appears when trying to link code that calls external functions written in C in "default runtime" mode. As explained in chapter 15, such code must be linked in "custom runtime" mode. Fix: add the `-custom` option, as well as the C libraries and C object files that implement the required external functions.

# Chapter 8

# The toplevel system (ocaml)

This chapter describes the toplevel system for Objective Caml, that permits interactive use of the Objective Caml system through a read-eval-print loop. In this mode, the system repeatedly reads Caml phrases from the input, then typechecks, compile and evaluate them, then prints the inferred type and result value, if any. The system prints a `#` (sharp) prompt before reading each phrase.

Input to the toplevel can span several lines. It is terminated by `;;` (a double-semicolon). The toplevel input consists in one or several toplevel phrases, with the following syntax:

```
      toplevel-input ::= {toplevel-phrase} ;;
     toplevel-phrase ::= definition
                       | expr
                       | # ident directive-argument
          definition ::= let [rec] let-binding {and let-binding}
                       | external value-name : typexpr = external-declaration
                       | type-definition
                       | exception-definition
                       | module module-name [: module-type] = module-expr
                       | module type modtype-name = module-type
                       | open module-path
  directive-argument ::= nothing
                       | string-literal
                       | integer-literal
                       | value-path
```

A phrase can consist of a definition, similar to those found in implementations of compilation units or in `struct`...`end` module expressions. The definition can bind value names, type names, an exception, a module name, or a module type name. The toplevel system performs the bindings, then prints the types and values (if any) for the names thus defined.

A phrase may also consist in a `open` directive (see section 5.11), or a value expression (section 5.7). Expressions are simply evaluated, without performing any bindings, and the value of the expression is printed.

Finally, a phrase can also consist in a toplevel directive, starting with # (the sharp sign). These directives control the behavior of the toplevel; they are listed below in section 8.2.

**Unix:**

The toplevel system is started by the command `ocaml`, as follows:

```
ocaml options                 # interactive mode
ocaml options scriptfile       # script mode
```

If no filename is given on the command line, the toplevel system enters interactive mode: phrases are read on standard input, results are printed on standard output, errors on standard error. End-of-file on standard input terminates `ocaml` (see also the `#quit` directive in section 8.2).

On start-up (before the first phrase is read), if the file `.ocamlinit` exists in the current directory, its contents are read as a sequence of Objective Caml phrases and executed as per the `#use` directive described in section 8.2. The evaluation outcode for each phrase are not displayed.

The toplevel system does not perform line editing, but it can easily be used in conjunction with an external line editor such as `fep`; just run `fep -emacs ocaml` or `fep -vi ocaml`. Another option is to use `ocaml` under Gnu Emacs, which gives the full editing power of Emacs (see the subdirectory `emacs` of the Objective Caml distribution).

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing `ctrl-C` (or, more precisely, by sending the `sigintr` signal to the `ocaml` process). The toplevel then immediately returns to the # prompt.

If a filename is given on the command-line to `ocaml`, the toplevel system enters script mode: the contents of the file are read as a sequence of Objective Caml phrases and executed, as per the `#use` directive (section 8.2). The outcome of the evaluation is not printed. On reaching the end of file, the `ocaml` command exits immediately. No commands are read from standard input.

In script mode, the first line of the script is ignored if it starts with `#!`. Thus, it is theoretically possible to make the script itself executable and put as first line `#!/usr/local/bin/ocaml`, thus calling the toplevel system automatically when the script is run. However, `ocaml` itself is a `#!` script on most installations of Objective Caml, and Unix kernels usually do not handle nested `#!` scripts.

**Windows:**

In addition to the text-only command `ocaml.exe`, which works exactly as under Unix (see above), a graphical user interface for the toplevel is available under the name `ocamlwin.exe`. It should be launched from the Windows file manager or program manager.

The “Terminal” windows is split in two panes. Phrases are entered and edited in the bottom pane. The top pane displays a copy of the input phrases as they are processed by the Caml Light toplevel, interspersed with the toplevel responses. The “Return” key sends the contents of the bottom pane to the Caml Light toplevel. The “Enter” key inserts a newline without sending the contents of the Input window. (This can be configured with the “Preferences” menu item.)

The contents of the input window can be edited at all times, with the standard Windows interface. An history of previously entered phrases is maintained and displayed in a separate window.

To quit the `Camlwin` application, either select “Quit” from the “File” menu, or use the `quit` function described below.

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by selecting the “Interrupt Caml Light” menu item. This goes back to the `#` prompt.

## 8.1 Options

The following command-line options are recognized by the `ocaml` command.

`-I` *directory*
: Add the given directory to the list of directories searched for source and compiled files. By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

  Directories can also be added to the search path once the toplevel is running with the `#directory` directive (section 8.2).

`-unsafe`
: See the corresponding option for `ocamlc`, chapter 7. Turn bound checking off on array and string accesses (the `v.(i)` and `s.[i]` constructs). Programs compiled with `-unsafe` are therefore slightly faster, but unsafe: anything can happen if the program accesses an array or string outside of its bounds.

**Unix:**
: The following environment variables are also consulted:

  `LC_CTYPE`
  : If set to `iso_8859_1`, accented characters (from the ISO Latin-1 character set) in string and character literals are printed as is; otherwise, they are printed as decimal escape sequences (`\`*ddd*).

  `TERM`
  : When printing error messages, the toplevel system attempts to underline visually the location of the error. It consults the `TERM` variable to determines the type of output terminal and look up its capabilities in the terminal database.

## 8.2 Toplevel directives

The following directives control the toplevel behavior, load files in memory, and trace program execution.

`#quit;;`
: Exit the toplevel loop and terminate the `ocaml` command.

`#directory "`*dir-name*`";;`
: Add the given directory to the list of directories searched for source and compiled files.

`#cd "`*dir-name*`";;`
: Change the current working directory.

`#load "`*file-name*`";;`
: Load in memory a bytecode object file (`.cmo` file) produced by the batch compiler `ocamlc`.

`#use "`*file-name*`";;`
: Read, compile and execute source phrases from the given file. This is textual inclusion: phrases are processed just as if they were typed on standard input. The reading of the file stops at the first error encountered.

`#install_printer` *printer-name*`;;`
: This directive registers the function named *printer-name* (a value path) as a printer for objects whose types match the argument type of the function. That is, the toplevel loop will call *printer-name* when it has such an object to print. The printing function *printer-name* must use the `Format` library module to produce its output, otherwise its output will not be correctly located in the values printed by the toplevel loop.

`#remove_printer` *printer-name*`;;`
: Remove the named function from the table of toplevel printers.

`#trace` *function-name*`;;`
: After executing this directive, all calls to the function named *function-name* will be "traced". That is, the argument and the result are displayed for each call, as well as the exceptions escaping out of the function, raised either by the function itself or by another function it calls. If the function is curried, each argument is printed as it is passed to the function.

`#untrace` *function-name*`;;`
: Stop tracing the given function.

`#untrace_all;;`
: Stop tracing all functions traced so far.

`#print_depth` *n*`;;`
: Limit the printing of values to a maximal depth of *n*. The parts of values whose depth exceeds *n* are printed as `...` (ellipsis).

`#print_length` *n*`;;`
: Limit the number of value nodes printed to at most *n*. Remaining parts of values are printed as `...` (ellipsis).

## 8.3 The toplevel and the module system

Toplevel phrases can refer to identifiers defined in compilation units with the same mechanisms as for separately compiled units: either by using qualified names (`Modulename.localname`), or by using the `open` construct and unqualified names (see section 5.3).

However, before referencing another compilation unit, an implementation of that unit must be present in memory. At start-up, the toplevel system contains implementations for all the modules in the the standard library. Implementations for user modules can be entered with the `#load` directive described above. Referencing a unit for which no implementation has been provided results in the error "Reference to undefined global '...'".

Note that entering `open` *mod* merely accesses the compiled interface (`.cmi` file) for *mod*, but does not load the implementation of *mod*, and does not cause any error if no implementation of *mod* has been loaded. The error "reference to undefined global *mod*" will occur only when executing a value or module definition that refers to *mod*.

## 8.4 Common errors

This section describes and explains the most frequently encountered error messages.

**Cannot find file** *filename*

The named file could not be found in the current directory, nor in the directories of the search path.

If *filename* has the format *mod*`.cmi`, this means you have referenced the compilation unit *mod*, but its compiled interface could not be found. Fix: compile *mod*`.mli` or *mod*`.ml` first, to create the compiled interface *mod*`.cmi`.

If *filename* has the format *mod*`.cmo`, this means you are trying to load with `#load` a bytecode object file that does not exist yet. Fix: compile *mod*`.ml` first.

If your program spans several directories, this error can also appear because you haven't specified the directories to look into. Fix: use the `#directory` directive to add the correct directories to the search path.

**This expression has type** $t_1$**, but is used with type** $t_2$

See section 7.4.

**Reference to undefined global** *mod*

You have neglected to load in memory an implementation for a module with `#load`. See section 8.3 above.

## 8.5 Building custom toplevel systems: `ocamlmktop`

The `ocamlmktop` command builds Objective Caml toplevels that contain user code preloaded at start-up.

The `ocamlmktop` command takes as argument a set of `.cmo` and `.cma` files, and links them with the object files that implement the Objective Caml toplevel. The typical use is:

```
        ocamlmktop -o mytoplevel foo.cmo bar.cmo gee.cmo
```

This creates the bytecode file `mytoplevel`, containing the Objective Caml toplevel system, plus the code from the three `.cmo` files. This toplevel is directly executable and is started by:

```
        ./mytoplevel
```

This enters a regular toplevel loop, except that the code from `foo.cmo`, `bar.cmo` and `gee.cmo` is already loaded in memory, just as if you had typed:

```
        #load "foo.cmo";;
        #load "bar.cmo";;
        #load "gee.cmo";;
```

on entrance to the toplevel. The modules `Foo`, `Bar` and `Gee` are not opened, though; you still have to do

```
        open Foo;;
```

yourself, if this is what you wish.

## 8.6 Options

The following command-line options are recognized by `ocamlmktop`.

`-cclib` *libname*
: Pass the `-l`*libname* option to the C linker when linking in "custom runtime" mode. See the corresponding option for `ocamlc`, in chapter 7.

`-ccopt` *option*
: Pass the given option to the C compiler and linker, when linking in "custom runtime" mode. See the corresponding option for `ocamlc`, in chapter 7.

`-custom`
: Link in "custom runtime" mode. See the corresponding option for `ocamlc`, in chapter 7.

`-I` *directory*
: Add the given directory to the list of directories searched for compiled object code files (`.cmo` and `.cma`).

`-o` *exec-file*
: Specify the name of the toplevel file produced by the linker. The default is `a.out`.

# Chapter 9

# The runtime system (ocamlrun)

The `ocamlrun` command executes bytecode files produced by the linking phase of the `ocamlc` command.

## 9.1 Overview

The `ocamlrun` command comprises three main parts: the bytecode interpreter, that actually executes bytecode files; the memory allocator and garbage collector; and a set of C functions that implement primitive operations such as input/output.

The usage for `ocamlrun` is:

> `ocamlrun` *options bytecode-executable* $arg_1$ `...` $arg_n$

The first non-option argument is taken to be the name of the file containing the executable bytecode. (That file is searched in the executable path as well as in the current directory.) The remaining arguments are passed to the Caml Light program, in the string array `Sys.argv`. Element 0 of this array is the name of the bytecode executable file; elements 1 to $n$ are the remaining arguments $arg_1$ to $arg_n$.

As mentioned in chapter 7, in most cases, the bytecode executable files produced by the `ocamlc` command are self-executable, and manage to launch the `ocamlrun` command on themselves automatically. That is, assuming `caml.out` is a bytecode executable file,

> `caml.out` $arg_1$ `...` $arg_n$

works exactly as

> `ocamlrun caml.out` $arg_1$ `...` $arg_n$

Notice that it is not possible to pass options to `ocamlrun` when invoking `caml.out` directly.

## 9.2 Options

The following command-line option is recognized by `ocamlrun`.

`-v` When set, the memory manager prints some verbose messages on standard error.

The following environment variables are also consulted:

`CAMLRUNPARAM`

Set the garbage collection parameters. This variable must be a sequence of parameter specifications. A parameter specification is an option letter followed by an `=` sign, a decimal number, and an optional multiplier. There are seven options, the first six correspond to the fields of the `control` record documented in section 17.9:

**s** (`minor_heap_size`) Size of the minor heap.

**i** (`major_heap_increment`) Minimum size increment for the major heap.

**o** (`space_overhead`) The major GC speed setting.

**O** (`max_overhead`) The heap compaction trigger setting.

**v** (`verbose`) What GC messages to print to stderr. This is a sum of values selected from the following:

- **1** Start of major GC cycle.
- **2** Minor collection and major GC slice.
- **4** Growing and shrinking of the heap.
- **8** Resizing of stacks and memory manager tables.
- **16** Heap compaction.
- **32** Change of GC parameters.
- **64** Computation of major GC slice size.

**l** (`stack_limit`) The limit (in words) of the stack size.

**h** The initial size of the major heap (in words).

The multiplier is `k`, `M`, or `G`, for multiplication by $2^{10}$, $2^{20}$, and $2^{30}$ respectively. For example, on a 32-bit machine, under `bash` the command

```
export CAMLRUNPARAM='s=256k,v=1'
```

tells a subsequent `ocamlrun` to set its initial minor heap size to 1 megabyte and to print a message at the start of each major GC cycle.

`PATH`

List of directories searched to find the bytecode executable file.

## 9.3 Common errors

This section describes and explains the most frequently encountered error messages.

*filename*`: no such file or directory`

If *filename* is the name of a self-executable bytecode file, this means that either that file does not exist, or that it failed to run the `ocamlrun` bytecode interpreter on itself. The second possibility indicates that Objective Caml has not been properly installed on your system.

`Cannot exec camlrun`
: (When launching a self-executable bytecode file.) The `ocamlrun` could not be found in the executable path. Check that Objective Caml has been properly installed on your system.

`Cannot find the bytecode file`
: The file that `ocamlrun` is trying to execute (e.g. the file given as first non-option argument to `ocamlrun`) either does not exist, or is not a valid executable bytecode file.

`Truncated bytecode file`
: The file that `ocamlrun` is trying to execute is not a valid executable bytecode file. Probably it has been truncated or mangled since created. Erase and rebuild it.

`Uncaught exception`
: The program being executed contains a "stray" exception. That is, it raises an exception at some point, and this exception is never caught. This causes immediate termination of the program. The name of the exception is printed, but not its arguments.

`Out of memory`
: The program being executed requires more memory than available. Either the program builds excessively large data structures; or the program contains too many nested function calls, and the stack overflows. In some cases, your program is perfectly correct, it just requires more memory than your machine provides. In other cases, the "out of memory" message reveals an error in your program: non-terminating recursive function, allocation of an excessively large array or string, attempts to build an infinite list or other data structure, ...

  To help you diagnose this error, run your program with the `-v` option to `ocamlrun`. If it displays lots of "`Growing stack`..." messages, this is probably a looping recursive function. If it displays lots of "`Growing heap`..." messages, with the heap size growing slowly, this is probably an attempt to construct a data structure with too many (infinitely many?) cells. If it displays few "`Growing heap`..." messages, but with a huge increment in the heap size, this is probably an attempt to build an excessively large array or string.

# Chapter 10

# Native-code compilation (ocamlopt)

This chapter describes the Objective Caml high-performance native-code compiler `ocamlopt`, which compiles Caml source files to native code object files and link these object files to produce standalone executables.

The native-code compiler is only available on certain platforms. It produces code that runs faster than the bytecode produced by `ocamlc`, at the cost of increased compilation time and executable code size. Compatibility with the bytecode compiler is extremely high: the same source code should run identically when compiled with `ocamlc` and `ocamlopt`.

It is not possible to mix native-code object files produced by `ocamlc` with bytecode object files produced by `ocamlopt`: a program must be compiled entirely with `ocamlopt` or entirely with `ocamlc`. Native-code object files produced by `ocamlopt` cannot be loaded in the toplevel system `ocaml`.

## 10.1 Overview of the compiler

The `ocamlopt` command has a command-line interface very close to that of `ocamlc`. It accepts the same types of arguments:

- Arguments ending in `.mli` are taken to be source files for compilation unit interfaces. Interfaces specify the names exported by compilation units: they declare value names with their types, define public data types, declare abstract data types, and so on. From the file *x*`.mli`, the `ocamlopt` compiler produces a compiled interface in the file *x*`.cmi`. The interface produced is identical to that produced by the bytecode compiler `ocamlc`.

- Arguments ending in `.ml` are taken to be source files for compilation unit implementations. Implementations provide definitions for the names exported by the unit, and also contain expressions to be evaluated for their side-effects. From the file *x*`.ml`, the `ocamlopt` compiler produces two files: *x*`.o`, containing native object code, and *x*`.cmx`, containing extra information for linking and optimization of the clients of the unit. The compiled implementation should always be referred to under the name *x*`.cmx` (when given a `.o` file, `ocamlopt` assumes that it contains code compiled from C, not from Caml).

  The implementation is checked against the interface file *x*`.mli` (if it exists) as described in the manual for `ocamlc` (chapter 7).

- Arguments ending in `.cmx` are taken to be compiled object code. These files are linked together, along with the object files obtained by compiling `.ml` arguments (if any), and the Caml Light standard library, to produce a native-code executable program. The order in which `.cmx` and `.ml` arguments are presented on the command line is relevant: compilation units are initialized in that order at run-time, and it is a link-time error to use a component of a unit before having initialized it. Hence, a given *x*`.cmx` file must come before all `.cmx` files that refer to the unit *x*.
- Arguments ending in `.cmxa` are taken to be libraries of object code. Such a library packs in two files (*lib*`.cmxa` and *lib*`.a`) a set of object files (`.cmx`/`.o` files). Libraries are build with `ocamlopt -a` (see the description of the `-a` option below). The object files contained in the library are linked as regular `.cmx` files (see above), in the order specified when the library was built. The only difference is that if an object file contained in a library is not referenced anywhere in the program, then it is not linked in.
- Arguments ending in `.c` are passed to the C compiler, which generates a `.o` object file. This object file is linked with the program.
- Arguments ending in `.o` or `.a` are assumed to be C object files and libraries. They are linked with the program.

The output of the linking phase is a regular Unix executable file. It does not need `ocamlrun` to run.

## 10.2 Options

The following command-line options are recognized by `ocamlopt`.

`-a` Build a library (`.cmxa`/`.a` file) with the object files (`.cmx`/`.o` files) given on the command line, instead of linking them into an executable file. The name of the library can be set with the `-o` option. The default name is `library.cmxa`.

`-c` Compile only. Suppress the linking phase of the compilation. Source code files are turned into compiled files, but no executable file is produced. This option is useful to compile modules separately.

`-cc` *ccomp*
Use *ccomp* as the C linker called to build the final executable and as the C compiler for compiling `.c` source files.

`-cclib -l`*libname*
Pass the `-l`*libname* option to the linker. This causes the given C library to be linked with the program.

`-ccopt` *option*
Pass the given option to the C compiler and linker. For instance, `-ccopt -L`*dir* causes the C linker to search for C libraries in directory *dir*.

`-compact`
: Optimize the produced code for space rather than for time. This results in slightly smaller but slightly slower programs. The default is to optimize for speed.

`-i`
: Cause the compiler to print all defined names (with their inferred types or their definitions) when compiling an implementation (`.ml` file). This can be useful to check the types inferred by the compiler. Also, since the output follows the syntax of interfaces, it can help in writing an explicit interface (`.mli` file) for a file: just redirect the standard output of the compiler to a `.mli` file, and edit that file to remove all declarations of unexported names.

`-I` *directory*
: Add the given directory to the list of directories searched for compiled interface files (`.cmi`) and compiled object code files (`.cmx`). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, in the order in which they were given on the command line, but before the standard library directory.

`-inline` *n*
: Set aggressiveness of inlining to *n*, where *n* is a positive integer. Specifying `-inline 0` prevents all functions from being inlined, except those whose body is smaller than the call site. Thus, inlining causes no expansion in code size. The default aggressiveness, `-inline 1`, allows slightly larger functions to be inlined, resulting in a slight expansion in code size. Higher values for the `-inline` option cause larger and larger functions to become candidate for inlining, but can result in a serious increase in code size.

`-linkall`
: Forces all modules contained in libraries to be linked in. If this flag is not given, unreferenced modules are not linked in. When building a library (`-a` flag), setting the `-linkall` flag forces all subsequent links of programs involving that library to link all the modules contained in the library.

`-o` *exec-file*
: Specify the name of the output file produced by the linker. The default output name is `a.out`, in keeping with the Unix tradition. If the `-a` option is given, specify the name of the library produced. If the `-output-obj` option is given, specify the name of the output file produced.

`-output-obj`
: Cause the linker to produce a C object file instead of an executable file. This is useful to wrap Caml code as a C library, callable from any C program. See chapter 15, section 15.6.5. The name of the output object file is `camlprog.o` by default; it can be set with the `-o` option.

`-p`
: Generate extra code to write profile information when the program is executed. The profile information can then be examined with the analysis program `gprof`. (See chapter 14 for more information on profiling.) The `-p` option must be given both at compile-time and at link-time. Linking object files not compiled with `-p` is possible, but results in less precise profiling.

**Unix:**

See the Unix manual page for `gprof(1)` for more information about the profiles.

Full support for `gprof` is only available for certain platforms (currently: Intel x86/Linux and Alpha/Digital Unix). On other platforms, the `-p` option will result in a less precise profile (no call graph information, only a time profile).

**Windows:**

The `-p` option does not work under Windows.

`-pp` *command*
: Cause the compiler to call the given *command* as a preprocessor for each source file. The output of *command* is redirected to an intermediate file, which is compiled. If there are no compilation errors, the intermediate file is deleted afterwards. The name of this file is built from the basename of the source file with the extension `.ppi` for an interface (`.mli`) file and `.ppo` for an implementation (`.ml`) file.

`-S` Keep the assembly code produced during the compilation. The assembly code for the source file *x*`.ml` is saved in the file *x*`.s`.

`-thread`
: Compile or link multithreaded programs, in combination with the `threads` library described in chapter 21. What this option actually does is select a special, thread-safe version of the standard library.

`-unsafe`
: Turn bound checking off on array and string accesses (the `v.(i)` and `s.[i]` constructs). Programs compiled with `-unsafe` are therefore faster, but unsafe: anything can happen if the program accesses an array or string outside of its bounds.

`-v` Print the version number of the compiler.

`-w` *warning-list*
: Enable or disable warnings according to the argument *warning-list*. The argument is a string of one or several characters, with the following meaning for each character:

`A/a` enable/disable all warnings

**F/f** enable/disable warnings for partially applied functions (i.e. `f x;` *expr* where the application `f x` has a function type).

**M/m**
: enable/disable warnings for overriden methods.

**P/p**
: enable/disable warnings for partial matches (missing cases in pattern matchings).

**S/s** enable/disable warnings for statements that do not have type `unit` (e.g. *expr1*`;` *expr2* when *expr1* does not have type `unit`).

**U/u**
enable/disable warnings for unused (redundant) match cases.

**V/v**
enable/disable warnings for hidden instance variables.

**X/x**
enable/disable all other warnings.

The default setting is `-w A` (all warnings enabled).

## 10.3 Common errors

The error messages are almost identical to those of `ocamlc`. See section 7.4.

## 10.4 Compatibility with the bytecode compiler

This section lists the known incompatibilities between the bytecode compiler and the native-code compiler. Except on those points, the two compilers should generate code that behave identically.

- The following operations abort the program (either by printing an error message or just via an hardware trap or fatal Unix signal) instead of raising an exception:
  - integer division by zero, modulus by zero;
  - stack overflow;
  - on the Alpha processor only, floating-point operations involving infinite or denormalized numbers (all other processors supported by `ocamlopt` treat these numbers correctly, as per the IEEE 754 standard).

  In particular, notice that stack overflow caused by excessively deep recursion is reported by most Unix kernels as a "segmentation violation" signal.

- Signals are detected only when the program performs an allocation in the heap. That is, if a signal is delivered while in a piece of code that does not allocate, its handler will not be called until the next heap allocation.

The best way to avoid running into those incompatibilities is to *never* trap the `Division_by_zero` and `Stack_overflow` exceptions, thus also treating them as fatal errors with the bytecode compiler as well as with the native-code compiler. Test the divisor before performing the operation instead of trapping the exception afterwards.

# Chapter 11

# Lexer and parser generators (ocamllex, ocamlyacc)

This chapter describes two program generators: `ocamllex`, that produces a lexical analyzer from a set of regular expressions with associated semantic actions, and `ocamlyacc`, that produces a parser from a grammar with associated semantic actions.

These program generators are very close to the well-known `lex` and `yacc` commands that can be found in most C programming environments. This chapter assumes a working knowledge of `lex` and `yacc`: while it describes the input syntax for `ocamllex` and `ocamlyacc` and the main differences with `lex` and `yacc`, it does not explain the basics of writing a lexer or parser description in `lex` and `yacc`. Readers unfamiliar with `lex` and `yacc` are referred to "Compilers: principles, techniques, and tools" by Aho, Sethi and Ullman (Addison-Wesley, 1986), or "Lex & Yacc", by Levine, Mason and Brown (O'Reilly, 1992).

## 11.1 Overview of `ocamllex`

The `ocamllex` command produces a lexical analyzer from a set of regular expressions with attached semantic actions, in the style of `lex`. Assuming the input file is *lexer*`.mll`, executing

```
ocamllex lexer.mll
```

produces Caml code for a lexical analyzer in file *lexer*`.ml`. This file defines one lexing function per entry point in the lexer definition. These functions have the same names as the entry points. Lexing functions take as argument a lexer buffer, and return the semantic attribute of the corresponding entry point.

Lexer buffers are an abstract data type implemented in the standard library module `Lexing`. The functions `Lexing.from_channel`, `Lexing.from_string` and `Lexing.from_function` create lexer buffers that read from an input channel, a character string, or any reading function, respectively. (See the description of module `Lexing` in chapter 16.)

When used in conjunction with a parser generated by `ocamlyacc`, the semantic actions compute a value belonging to the type `token` defined by the generated parsing module. (See the description of `ocamlyacc` below.)

## 11.2 Syntax of lexer definitions

The format of lexer definitions is as follows:

```
{ header }
let ident = regexp ...
rule entrypoint =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint =
  parse ...
and ...
{ trailer }
```

Comments are delimited by `(*` and `*)`, as in Caml.

### 11.2.1 Header and trailer

The *header* and *trailer* sections are arbitrary Caml text enclosed in curly braces. Either or both can be omitted. If present, the header text is copied as is at the beginning of the output file and the trailer text at the end. Typically, the header section contains the `open` directives required by the actions, and possibly some auxiliary functions used in the actions.

### 11.2.2 Naming regular expressions

Between the header and the entry points, one can give names to frequently-occurring regular expressions. This is written `let` *ident* `=` *regexp*. In following regular expressions, the identifier *ident* can be used as shorthand for *regexp*.

### 11.2.3 Entry points

The names of the entry points must be valid identifiers for Caml values (starting with a lowercase letter).

### 11.2.4 Regular expressions

The regular expressions are in the style of `lex`, with a more Caml-like syntax.

' *char* '
: A character constant, with the same syntax as Objective Caml character constants. Match the denoted character.

`_` (Underscore.) Match any character.

`eof` Match the end of the lexer input.
**Note:** On some systems, with interactive input, and end-of-file may be followed by more characters. However, `ocamllex` will not correctly handle regular expressions that contain `eof` followed by something else.

`"` *string* `"`
: A string constant, with the same syntax as Objective Caml string constants. Match the corresponding sequence of characters.

`[` *character-set* `]`
: Match any single character belonging to the given character set. Valid character sets are: single character constants `'` *c* `'`; ranges of characters `'` $c_1$ `'` `-` `'` $c_2$ `'` (all characters between $c_1$ and $c_2$, inclusive); and the union of two or more character sets, denoted by concatenation.

`[` `^` *character-set* `]`
: Match any single character not belonging to the given character set.

*regexp* `*`
: (Repetition.) Match the concatenation of zero or more strings that match *regexp*.

*regexp* `+`
: (Strict repetition.) Match the concatenation of one or more strings that match *regexp*.

*regexp* `?`
: (Option.) Match either the empty string, or a string matching *regexp*.

$regexp_1$ `|` $regexp_2$
: (Alternative.) Match any string that matches either $regexp_1$ or $regexp_2$

$regexp_1$ $regexp_2$
: (Concatenation.) Match the concatenation of two strings, the first matching $regexp_1$, the second matching $regexp_2$.

`(` *regexp* `)`
: Match the same strings as *regexp*.

*ident*
: Reference the regular expression bound to *ident* by an earlier `let` *ident* `=` *regexp* definition.

Concerning the precedences of operators, `*` and `+` have highest precedence, followed by `?`, then concatenation, then `|` (alternation).

### 11.2.5 Actions

The actions are arbitrary Caml expressions. They are evaluated in a context where the identifier `lexbuf` is bound to the current lexer buffer. Some typical uses for `lexbuf`, in conjunction with the operations on lexer buffers provided by the `Lexing` standard library module, are listed below.

`Lexing.lexeme lexbuf`
: Return the matched string.

`Lexing.lexeme_char lexbuf` *n*
: Return the $n^{\text{th}}$ character in the matched string. The first character corresponds to $n = 0$.

`Lexing.lexeme_start lexbuf`
: Return the absolute position in the input text of the beginning of the matched string. The first character read from the input text has position 0.

`Lexing.lexeme_end lexbuf`
: Return the absolute position in the input text of the end of the matched string. The first character read from the input text has position 0.

*entrypoint* `lexbuf`
: (Where *entrypoint* is the name of another entry point in the same lexer definition.) Recursively call the lexer on the given entry point. Useful for lexing nested comments, for example.

## 11.3 Overview of `ocamlyacc`

The `ocamlyacc` command produces a parser from a context-free grammar specification with attached semantic actions, in the style of `yacc`. Assuming the input file is *grammar*`.mly`, executing

```
ocamlyacc options grammar.mly
```

produces Caml code for a parser in the file *grammar*`.ml`, and its interface in file *grammar*`.mli`.

The generated module defines one parsing function per entry point in the grammar. These functions have the same names as the entry points. Parsing functions take as arguments a lexical analyzer (a function from lexer buffers to tokens) and a lexer buffer, and return the semantic attribute of the corresponding entry point. Lexical analyzer functions are usually generated from a lexer specification by the `ocamllex` program. Lexer buffers are an abstract data type implemented in the standard library module `Lexing`. Tokens are values from the concrete type `token`, defined in the interface file *grammar*`.mli` produced by `ocamlyacc`.

## 11.4 Syntax of grammar definitions

Grammar definitions have the following format:

```
%{
  header
%}
  declarations
%%
  rules
%%
  trailer
```

Comments are enclosed between `/*` and `*/` (as in C) in the "declarations" and "rules" sections, and between `(*` and `*)` (as in Caml) in the "header" and "trailer" sections.

### 11.4.1 Header and trailer

The header and the trailer sections are Caml code that is copied as is into file *grammar*`.ml`. Both sections are optional. The header goes at the beginning of the output file; it usually contains `open` directives and auxiliary functions required by the semantic actions of the rules. The trailer goes at the end of the output file.

### 11.4.2 Declarations

Declarations are given one per line. They all start with a `%` sign.

`%token` *symbol* ... *symbol*
: Declare the given symbols as tokens (terminal symbols). These symbols are added as constant constructors for the `token` concrete type.

`%token <` *type* `>` *symbol* ... *symbol*
: Declare the given symbols as tokens with an attached attribute of the given type. These symbols are added as constructors with arguments of the given type for the `token` concrete type. The *type* part is an arbitrary Caml type expression, except that all type constructor names must be fully qualified (e.g. `Modname.typename`) for all types except standard built-in types, even if the proper `open` directives (e.g. `open Modname`) were given in the header section. That's because the header is copied only to the `.ml` output file, but not to the `.mli` output file, while the *type* part of a `%token` declaration is copied to both.

`%start` *symbol* ... *symbol*
: Declare the given symbols as entry points for the grammar. For each entry point, a parsing function with the same name is defined in the output module. Non-terminals that are not declared as entry points have no such parsing function. Start symbols must be given a type with the `%type` directive below.

`%type <` *type* `>` *symbol* ... *symbol*
: Specify the type of the semantic attributes for the given symbols. This is mandatory for start symbols only. Other nonterminal symbols need not be given types by hand: these types will be inferred when running the output files through the Objective Caml compiler (unless the `-s` option is in effect). The *type* part is an arbitrary Caml type expression, except that all type constructor names must be fully qualified, as explained above for `%token`.

`%left` *symbol* ... *symbol*

`%right` *symbol* ... *symbol*

`%nonassoc` *symbol* ... *symbol*
: Associate precedences and associativities to the given symbols. All symbols on the same line are given the same precedence. They have higher precedence than symbols declared before in a `%left`, `%right` or `%nonassoc` line. They have lower precedence than symbols declared

after in a `%left`, `%right` or `%nonassoc` line. The symbols are declared to associate to the left (`%left`), to the right (`%right`), or to be non-associative (`%nonassoc`). The symbols are usually tokens. They can also be dummy nonterminals, for use with the `%prec` directive inside the rules.

### 11.4.3 Rules

The syntax for rules is as usual:

*nonterminal* :
    *symbol* ... *symbol* { *semantic-action* }
  | ...
  | *symbol* ... *symbol* { *semantic-action* }
;

Rules can also contain the `%prec` *symbol* directive in the right-hand side part, to override the default precedence and associativity of the rule with the precedence and associativity of the given symbol.

Semantic actions are arbitrary Caml expressions, that are evaluated to produce the semantic attribute attached to the defined nonterminal. The semantic actions can access the semantic attributes of the symbols in the right-hand side of the rule with the `$` notation: `$1` is the attribute for the first (leftmost) symbol, `$2` is the attribute for the second symbol, etc.

The rules may contain the special symbol `error` to indicate resynchronization points, as in `yacc`.

Actions occurring in the middle of rules are not supported.

### 11.4.4 Error handling

Error recovery is supported as follows: when the parser reaches an error state (no grammar rules can apply), it calls a function named `parse_error` with the string `syntax error` as argument. The default `parse_error` function does nothing and returns, thus initiating error recovery (see below). The user can define a customized `parse_error` function in the header section of the grammar file.

The parser also enters error recovery mode if one of the grammar actions raise the `Parsing.Parse_error` exception.

In error recovery mode, the parser discards states from the stack until it reaches a place where the error token can be shifted. It then discards tokens from the input until it finds three successive tokens that can be accepted, and starts processing with the first of these. If no state can be uncovered where the error token can be shifted, then the parser aborts by raising the `Parsing.Parse_error` exception.

Refer to documentation on `yacc` for more details and guidance in how to use error recovery.

## 11.5 Options

The `ocamlyacc` command recognizes the following options:

`-v` Generate a description of the parsing tables and a report on conflicts resulting from ambiguities in the grammar. The description is put in file *grammar*`.output`.

`-b`*prefix*
: Name the output files *prefix*`.ml`, *prefix*`.mli`, *prefix*`.output`, instead of the default naming convention.

## 11.6 A complete example

The all-time favorite: a desk calculator. This program reads arithmetic expressions on standard input, one per line, and prints their values. Here is the grammar definition:

```
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* highest precedence */
%start main             /* the entry point */
%type <int> main
%%
main:
    expr EOL                { $1 }
;
expr:
    INT                     { $1 }
  | LPAREN expr RPAREN      { $2 }
  | expr PLUS expr          { $1 + $3 }
  | expr MINUS expr         { $1 - $3 }
  | expr TIMES expr         { $1 * $3 }
  | expr DIV expr           { $1 / $3 }
  | MINUS expr %prec UMINUS { - $2 }
;
```

Here is the definition for the corresponding lexer:

```
(* File lexer.mll *)
{
open Parser        (* The type token is defined in parser.mli *)
exception Eof
}
rule token = parse
    [' ' '\t']     { token lexbuf }     (* skip blanks *)
  | ['\n' ]        { EOL }
  | ['0'-'9']+     { INT(int_of_string(Lexing.lexeme lexbuf)) }
  | '+'            { PLUS }
  | '-'            { MINUS }
```

```
        | '*'           { TIMES }
        | '/'           { DIV }
        | '('           { LPAREN }
        | ')'           { RPAREN }
        | eof           { raise Eof }
```

Here is the main program, that combines the parser with the lexer:

```
(* File calc.ml *)
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
    while true do
      let result = Parser.main Lexer.token lexbuf in
        print_int result; print_newline(); flush stdout
    done
  with Lexer.Eof ->
    exit 0
```

To compile everything, execute:

```
ocamllex lexer.mll       # generates lexer.ml
ocamlyacc parser.mly     # generates parser.ml and parser.mli
ocamlc -c parser.mli
ocamlc -c lexer.ml
ocamlc -c parser.ml
ocamlc -c calc.ml
ocamlc -o calc lexer.cmo parser.cmo calc.cmo
```

## 11.7 Common errors

**ocamllex: transition table overflow, automaton is too big**

The deterministic automata generated by `ocamllex` are limited to at most 32767 transitions. The message above indicates that your lexer definition is too complex and overflows this limit. This is commonly caused by lexer definitions that have separate rules for each of the alphabetic keywords of the language, as in the following example.

```
rule token = parse
  "keyword1"   { KWD1 }
| "keyword2"   { KWD2 }
| ...
| "keyword100" { KWD100 }
| ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_'] *
               { IDENT(Lexing.lexeme lexbuf) }
```

To keep the generated automata small, rewrite those definitions with only one general "identifier" rule, followed by a hashtable lookup to separate keywords from identifiers:

```
{ let keyword_table = Hashtbl.create 53
  let _ =
    List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
              [ "keyword1", KWD1;
                "keyword2", KWD2; ...
                "keyword100", KWD100 ]
}
rule token = parse
  ['A'-'Z' 'a'-'z'] ['A'-'Z' 'a'-'z' '0'-'9' '_'] *
               { let id = Lexing.lexeme lexbuf in
                 try
                   Hashtbl.find keyword_table s
                 with Not_found ->
                   IDENT s }
```

# Chapter 12

# Dependency generator (ocamldep)

The `ocamldep` command scans a set of Objective Caml source files (`.ml` and `.mli` files) for references to external compilation units, and outputs dependency lines in a format suitable for the `make` utility. This ensures that `make` will compile the source files in the correct order, and recompile those files that need to when a source file is modified.

The typical usage is:

> `ocamldep` *options* `*.mli *.ml > .depend`

where `*.mli *.ml` expands to all source files in the current directory and `.depend` is the file that should contain the dependencies. (See below for a typical `Makefile`.)

Dependencies are generated both for compiling with the bytecode compiler `ocamlc` and with the native-code compiler `ocamlopt`.

## 12.1 Options

The following command-line option is recognized by `ocamldep`.

`-I` *directory*
: Add the given directory to the list of directories searched for source files. If a source file `foo.ml` mentions an external compilation unit `Bar`, a dependency on that unit's interface `bar.cmi` is generated only if the source for `bar` is found in the current directory or in one of the directories specified with `-I`. Otherwise, `Bar` is assumed to be a module form the standard library, and no dependencies are generated. For programs that span multiple directories, it is recommended to pass `ocamldep` the same `-I` options that are passed to the compiler.

## 12.2 A typical Makefile

Here is a template `Makefile` for a Objective Caml program.

```
OCAMLC=ocamlc
OCAMLOPT=ocamlopt
OCAMLDEP=ocamldep
```

```
INCLUDES=                 # all relevant -I options here
OCAMLFLAGS=$(INCLUDES)    # add other options for ocamlc here
OCAMLOPTFLAGS=$(INCLUDES) # add other options for ocamlopt here

# prog1 should be compiled to bytecode, and is composed of three
# units: mod1, mod2 and mod3.

# The list of object files for prog1
PROG1_OBJS=mod1.cmo mod2.cmo mod3.cmo

prog1: $(PROG1_OBJS)
        $(OCAMLC) -o prog1 $(OCAMLFLAGS) $(PROG1_OBJS)

# prog2 should be compiled to native-code, and is composed of two
# units: mod4 and mod5.

# The list of object files for prog2
PROG2_OBJS=mod4.cmx mod5.cmx

prog2: $(PROG2_OBJS)
        $(OCAMLOPT) -o prog2 $(OCAMLFLAGS) $(PROG2_OBJS)

# Common rules
.SUFFIXES: .ml .mli .cmo .cmi .cmx

.ml.cmo:
        $(OCAMLC) $(OCAMLFLAGS) -c $<

.mli.cmi:
        $(OCAMLC) $(OCAMLFLAGS) -c $<

.ml.cmx:
        $(OCAMLOPT) $(OCAMLOPTFLAGS) -c $<

# Clean up
clean:
        rm -f prog1 prog2
        rm -f *.cm[iox]

# Dependencies
depend:
        $(OCAMLDEP) $(INCLUDES) *.mli *.ml > .depend

include .depend
```

# Chapter 13

# The debugger (ocamldebug)

This chapter describes the Objective Caml source-level replay debugger `ocamldebug`.

**Unix:**
The debugger is available on Unix systems that provides BSD sockets.

**Windows:**
The debugger is not available.

## 13.1 Compiling for debugging

Before the debugger can be used, the program must be compiled and linked with the `-g` option: all `.cmo` and `.cma` files that are part of the program should have been created with `ocamlc -g`, and they must be linked together with `ocamlc -g`.

Compiling with `-g` entails no penalty on the running time of programs: object files and bytecode executable files are bigger and take longer to produce, but the executable files run at exactly the same speed as if they had been compiled without `-g`.

## 13.2 Invocation

### 13.2.1 Starting the debugger

The Objective Caml debugger is invoked by running the program `ocamldebug` with the name of the bytecode executable file as first argument:

`ocamldebug [`*options*`]` *program* `[`*arguments*`]`

The arguments following *program* are optional, and are passed as command-line arguments to the program being debugged. (See also the `set arguments` command.)

The following command-line options are recognized:

`-I` *directory*
Add *directory* to the list of directories searched for source files and compiled files. (See also the `directory` command.)

`-s` *socket*
: Use *socket* for communicating with the debugged program. See the description of the command `set socket` (section 13.8.6) for the format of *socket*.

`-c` *count*
: Set the maximum number of simultaneously live checkpoints to *count*.

`-cd` *directory*
: Run the debugger program from the working directory *directory*, instead of the current directory. (See also the `cd` command.)

`-emacs`
: Tell the debugger it is executed under Emacs. (See section 13.10 for information on how to run the debugger under Emacs.)

### 13.2.2 Exiting the debugger

The command `quit` exits the debugger. You can also exit the debugger by typing an end-of-file character (usually `ctrl-D`).

Typing an interrupt character (usually `ctrl-C`) will not exit the debugger, but will terminate the action of any debugger command that is in progress and return to the debugger command level.

## 13.3 Commands

A debugger command is a single line of input. It starts with a command name, which is followed by arguments depending on this name. Examples:

```
run
goto 1000
set arguments arg1 arg2
```

A command name can be truncated as long as there is no ambiguity. For instance, `go 1000` is understood as `goto 1000`, since there are no other commands whose name starts with `go`. For the most frequently used commands, ambiguous abbreviations are allowed. For instance, `r` stands for `run` even though there are others commands starting with `r`. You can test the validity of an abbreviation using the `help` command.

If the previous command has been successful, a blank line (typing just `RET`) will repeat it.

### 13.3.1 Getting help

The Objective Caml debugger has a simple on-line help system, which gives a brief description of each command and variable.

`help`
: Print the list of commands.

`help` *command*
: Give help about the command *command*.

`help set` *variable*, `help show` *variable*
: Give help about the variable *variable*. The list of all debugger variables can be obtained with `help set`.

`help info` *topic*
: Give help about *topic*. Use `help info` to get a list of known topics.

### 13.3.2 Accessing the debugger state

`set` *variable value*
: Set the debugger variable *variable* to the value *value*.

`show` *variable*
: Print the value of the debugger variable *variable*.

`info` *subject*
: Give information about the given subject. For instance, `info breakpoints` will print the list of all breakpoints.

## 13.4 Executing a program

### 13.4.1 Events

Events are "interesting" locations in the source code, corresponding to the beginning or end of evaluation of "interesting" sub-expressions. Events are the unit of single-stepping (stepping goes to the next or previous event encountered in the program execution). Also, breakpoints can only be set at events. Thus, events play the role of line numbers in debuggers for conventional languages.

During program execution, a counter is incremented at each event encountered. The value of this counter is referred as the *current time*. Thanks to reverse execution, it is possible to jump back and forth to any time of the execution.

Here is where the debugger events (written ⋈) are located in the source code:

- Following a function application:

  ```
  (f arg)⋈
  ```

- On entrance to a function:

  ```
  fun x y z -> ⋈ ...
  ```

- On each case of a pattern-matching definition (function, `match`...`with` construct, `try`...`with` construct):

  ```
  function pat1 -> ⋈ expr1
         | ...
         | patN -> ⋈ exprN
  ```

- Between subexpressions of a sequence:

```
expr1; ⋈ expr2; ⋈ ...; ⋈ exprN
```

- In the two branches of a conditional expression:

```
if cond then ⋈ expr1 else ⋈ expr2
```

- At the beginning of each iteration of a loop:

```
while cond do ⋈ body done
for i = a to b do ⋈ body done
```

Exceptions: A function application followed by a function return is replaced by the compiler by a jump (tail-call optimization). In this case, no event is put after the function application.

### 13.4.2 Starting the debugged program

The debugger starts executing the debugged program only when needed. This allows setting breapoints or assigning debugger variables before execution starts. There are several ways to start execution:

`run` Run the program until a breakpoint is hit, or the program terminates.

`step 0`
: Load the program and stop on the first event.

`goto` *time*
: Load the program and execute it until the given time. Useful when you already know approximately at what time the problem appears. Also useful to set breakpoints on function values that have not been computed at time 0 (see section 13.5).

The execution of a program is affected by certain information it receives when the debugger starts it, such as the command-line arguments to the program and its working directory. The debugger provides commands to specify this information (`set arguments` and `cd`). These commands must be used before program execution starts. If you try to change the arguments or the working directory after starting your program, the debugger will kill the program (after asking for confirmation).

### 13.4.3 Running the program

The following commands execute the program forward or backward, starting at the current time. The execution will stop either when specified by the command or when a breakpoint is encountered.

`run` Execute the program forward from current time. Stops at next breakpoint or when the program terminates.

`reverse`
: Execute the program backward from current time. Mostly useful to go to the last breakpoint encountered before the current time.

`step` [*count*]
: Run the program and stop at the next event. With an argument, do it *count* times.

`backstep` [*count*]
: Run the program backward and stop at the previous event. With an argument, do it *count* times.

`next` [*count*]
: Run the program and stop at the next event, skipping over function calls. With an argument, do it *count* times.

`previous` [*count*]
: Run the program backward and stop at the previous event, skipping over function calls. With an argument, do it *count* times.

`finish`
: Run the program until the current function returns.

`start`
: Run the program backward and stop at the first event before the current function invocation.

### 13.4.4 Time travel

You can jump directly to a given time, without stopping on breakpoints, using the `goto` command.

As you move through the program, the debugger maintains an history of the successive times you stop at. The `last` command can be used to revisit these times: each `last` command moves one step back through the history. That is useful mainly to undo commands such as `step` and `next`.

`goto` *time*
: Jump to the given time.

`last` [*count*]
: Go back to the latest time recorded in the execution history. With an argument, do it *count* times.

`set history` *size*
: Set the size of the execution history.

### 13.4.5 Killing the program

`kill`
: Kill the program being executed. This command is mainly useful if you wish to recompile the program without leaving the debugger.

## 13.5 Breakpoints

A breakpoint causes the program to stop whenever a certain point in the program is reached. It can be set in several ways using the `break` command. Breakpoints are assigned numbers when set, for further reference. The most comfortable way to set breakpoints is through the Emacs interface (see section 13.10).

`break`
: Set a breakpoint at the current position in the program execution. The current position must be on an event (i.e., neither at the beginning, nor at the end of the program).

`break` *function*
: Set a breakpoint at the beginning of *function*. This works only when the functional value of the identifier *function* has been computed and assigned to the identifier. Hence this command cannot be used at the very beginning of the program execution, when all identifiers are still undefined; use `goto` *time* to advance execution until the functional value is available.

`break @` [*module*] *line*
: Set a breakpoint in module *module* (or in the current module if *module* is not given), at the first event of line *line*.

`break @` [*module*] *line column*
: Set a breakpoint in module *module* (or in the current module if *module* is not given), at the event closest to line *line*, column *column*.

`break @` [*module*] `#` *character*
: Set a breakpoint in module *module* at the event closest to character number *character*.

`break` *address*
: Set a breakpoint at the code address *address*.

`delete` [*breakpoint-numbers*]
: Delete the specified breakpoints. Without argument, all breakpoints are deleted (after asking for confirmation).

`info breakpoints`
: Print the list of all breakpoints.

## 13.6 The call stack

Each time the program performs a function application, it saves the location of the application (the return address) in a block of data called a stack frame. The frame also contains the local variables of the caller function. All the frames are allocated in a region of memory called the call stack. The command `backtrace` (or `bt`) displays parts of the call stack.

At any time, one of the stack frames is "selected" by the debugger; several debugger commands refer implicitly to the selected frame. In particular, whenever you ask the debugger for the value of a local variable, the value is found in the selected frame. The commands `frame`, `up` and `down` select whichever frame you are interested in.

When the program stops, the debugger automatically selects the currently executing frame and describes it briefly as the `frame` command does.

`frame`
: Describe the currently selected stack frame.

`frame` *frame-number*
: Select a stack frame by number and describe it. The frame currently executing when the program stopped has number 0; its caller has number 1; and so on up the call stack.

`backtrace` [*count*], `bt` [*count*]
: Print the call stack. This is useful to see which sequence of function calls led to the currently executing frame. With a positive argument, print only the innermost *count* frames. With a negative argument, print only the outermost -*count* frames.

`up` [*count*]
: Select and display the stack frame just "above" the selected frame, that is, the frame that called the selected frame. An argument says how many frames to go up.

`down` [*count*]
: Select and display the stack frame just "below" the selected frame, that is, the frame that was called by the selected frame. An argument says how many frames to go down.

## 13.7 Examining variable values

The debugger can print the current value of simple expressions. The expressions can involve program variables: all the identifiers that are in scope at the selected program point can be accessed.

Expressions that can be printed are a subset of Objective Caml expressions, as described by the following grammar:

```
expr ::= lowercase-ident
       | {uppercase-ident .} lowercase-ident
       | *
       | $ integer
       | expr . lowercase-ident
       | expr .( integer )
       | expr .[ integer ]
       | ! expr
       | ( expr )
```

The first two cases refer to a value identifier, either unqualified or qualified by the path to the structure that define it. `*` refers to the result just computed (typically, the value of a function application), and is valid only if the selected event is an "after" event (typically, a function application). `$` *integer* refer to a previously printed value. The remaining four forms select part of an expression: respectively, a record field, an array element, a string element, and the current contents of a reference.

`print` *variables*
: Print the values of the given variables. `print` can be abbreviated as `p`.

`display` *variables*
: Same as `print`, but limit the depth of printing to 1. Useful to browse large data structures without printing them in full. `display` can be abbreviated as `d`.

When printing a complex expression, a name of the form `$`*integer* is automatically assigned to its value. Such names are also assigned to parts of the value that cannot be printed because the maximal printing depth is exceeded. Named values can be printed later on with the commands `p $`*integer* or `d $`*integer*. Named values are valid only as long as the program is stopped. They are forgotten as soon as the program resumes execution.

`set print_depth` *d*
: Limit the printing of values to a maximal depth of *d*.

`set print_length` *l*
: Limit the printing of values to at most *l* nodes printed.

## 13.8 Controlling the debugger

### 13.8.1 Setting the program name and arguments

`set program` *file*
: Set the program name to *file*.

`set arguments` *arguments*
: Give *arguments* as command-line arguments for the program.

A shell is used to pass the arguments to the debugged program. You can therefore use wildcards, shell variables, and file redirections inside the arguments. To debug programs that read from standard input, it is recommended to redirect their input from a file (using `set arguments < input-file`), otherwise input to the program and input to the debugger are not properly separated, and inputs are not properly replayed when running the program backwards.

### 13.8.2 How programs are loaded

The `loadingmode` variable controls how the program is executed.

`set loadingmode direct`
: The program is run directly by the debugger. This is the default mode.

`set loadingmode runtime`
: The debugger execute the Objective Caml runtime `camlrun` on the program. Rarely useful; moreover it prevents the debugging of programs compiled in “custom runtime” mode.

`set loadingmode manual`
: The user starts manually the program, when asked by the debugger. Allows remote debugging (see section 13.8.6).

### 13.8.3 Search path for files

The debugger searches for source files and compiled interface files in a list of directories, the search path. The search path initially contains the current directory `.` and the standard library directory. The `directory` command adds directories to the path.

Whenever the search path is modified, the debugger will clear any information it may have cached about the files.

`directory` *directorynames*
: Add the given directories to the search path. These directories are added at the front, and will therefore be searched first.

`directory`
: Reset the search path. This requires confirmation.

### 13.8.4 Working directory

Each time a program is started in the debugger, it inherits its working directory from the current working directory of the debugger. This working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in the debugger with the `cd` command or the `-cd` command-line option.

`cd` *directory*
: Set the working directory for `camldebug` to *directory*.

`pwd` Print the working directory for `camldebug`.

### 13.8.5 Turning reverse execution on and off

In some cases, you may want to turn reverse execution off. This speeds up the program execution, and is also sometimes useful for interactive programs.

Normally, the debugger takes checkpoints of the program state from time to time. That is, it makes a copy of the current state of the program (using the Unix system call `fork`). If the variable *checkpoints* is set to `off`, the debugger will not take any checkpoints.

`set checkpoints` *on/off*
: Select whether the debugger makes checkpoints or not.

### 13.8.6 Communication between the debugger and the program

The debugger communicate with the program being debugged through a Unix socket. You may need to change the socket name, for example if you need to run the debugger on a machine and your program on another.

`set socket` *socket*
: Use *socket* for communication with the program. *socket* can be either a file name, or an Internet port specification *host*:*port*, where *host* is a host name or an Internet address in dot notation, and *port* is a port number on the host.

On the debugged program side, the socket name is passed either by the `-D` command line option to `camlrun`, or through the `CAML_DEBUG_SOCKET` environment variable.

### 13.8.7 Fine-tuning the debugger

Several variables enables to fine-tune the debugger. Reasonable defaults are provided, and you should normally not have to change them.

`set processcount` *count*
: Set the maximum number of checkpoints to *count*. More checkpoints facilitate going far back in time, but use more memory and create more Unix processes.

As checkpointing is quite expensive, it must not be done too often. On the other hand, backward execution is faster when checkpoints are taken more often. In particular, backward single-stepping is more responsive when many checkpoints have been taken just before the current time. To fine-tune the checkpointing strategy, the debugger does not take checkpoints at the same frequency for long displacements (e.g. `run`) and small ones (e.g. `step`). The two variables `bigstep` and `smallstep` contain the number of events between two checkpoints in each case.

`set bigstep` *count*
: Set the number of events between two checkpoints for long displacements.

`set smallstep` *count*
: Set the number of events between two checkpoints for small displacements.

The following commands display information on checkpoints and events:

`info checkpoints`
: Print a list of checkpoints.

`info events` [*module*]
: Print the list of events in the given module (the current module, by default).

### 13.8.8 User-defined printers

Just as in the toplevel system (section 8.2), the user can register functions for printing values of certain types. For technical reasons, the debugger cannot call printing functions that reside in the program being debugged. The code for the printing functions must therefore be loaded explicitly in the debugger.

`load_printer "`*file-name*`"`
: Load in the debugger the indicated `.cmo` or `.cma` object file. The file is loaded in an environment consisting only of the Objective Caml standard library plus the definitions provided by object files previously loaded using `load_printer`. If this file depends on other object files not yet loaded, the debugger automatically loads them if it is able to find them in the search path. The loaded file does not have direct access to the modules of the program being debugged.

`install_printer` *printer-name*
: Register the function named *printer-name* (a value path) as a printer for objects whose types match the argument type of the function. That is, the debugger will call *printer-name* when it has such an object to print. The printing function *printer-name* must use the `Format` library module to produce its output, otherwise its output will not be correctly located in the values printed by the toplevel loop.

  The value path *printer-name* must refer to one of the functions defined by the object files loaded using `load_printer`. It cannot reference the functions of the program being debugged.

`remove_printer` *printer-name*
: Remove the named function from the table of value printers.

## 13.9 Miscellaneous commands

`list` [*module*] [*beginning*] [*end*]
: List the source of module *module*, from line number *beginning* to line number *end*. By default, 20 lines of the current module are displayed, starting 10 lines before the current position.

`source` *filename*
: Read debugger commands from the script *filename*.

## 13.10 Running the debugger under Emacs

The most user-friendly way to use the debugger is to run it under Emacs. See the file `emacs/README` in the distribution for information on how to load the Emacs Lisp files for Caml support.

The Caml debugger is started under Emacs by the command `M-x camldebug`, with argument the name of the executable file *progname* to debug. Communication with the debugger takes place in an Emacs buffer named `*camldebug-`*progname*`*`. The editing and history facilities of Shell mode are available for interacting with the debugger.

In addition, Emacs displays the source files containing the current event (the current position in the program execution) and highlights the location of the event. This display is updated synchronously with the debugger action.

The following bindings for the most common debugger commands are available in the `*camldebug-`*progname*`*` buffer:

`C-c C-s`
: (command `step`): execute the program one step forward.

`C-c C-k`
: (command `backstep`): execute the program one step backward.

`C-c C-n`
: (command `next`): execute the program one step forward, skipping over function calls.

**Middle mouse button**
: (command `display`): display named value. `$`*n* under mouse cursor (support incremental browsing of large data structures).

`C-c C-p`
: (command `print`): print value of identifier at point.

`C-c C-d`
: (command `display`): display value of identifier at point.

`C-c C-r`
: (command `run`): execute the program forward to next breakpoint.

`C-c C-v`
: (command `reverse`): execute the program backward to latest breakpoint.

`C-c C-l`
: (command `last`): go back one step in the command history.

`C-c C-t`
: (command `backtrace`): display backtrace of function calls.

`C-c C-f`
: (command `finish`): run forward till the current function returns.

`C-c <`
: (command `up`): select the stack frame below the current frame.

`C-c >`
: (command `down`): select the stack frame above the current frame.

In all buffers in Caml editing mode, the following debugger commands are also available:

`C-x C-a C-b`
: (command `break`): set a breakpoint at event closest to point

`C-x C-a C-p`
: (command `print`): print value of identifier at point

`C-x C-a C-d`
: (command `display`): display value of identifier at point

# Chapter 14

# Profiling (ocamlprof)

This chapter describes how the execution of Objective Caml programs can be profiled, by recording how many times functions are called, branches of conditionals are taken, . . .

## 14.1 Compiling for profiling

Before profiling an execution, the program must be compiled in profiling mode, using the `ocamlcp` front-end to the `ocamlc` compiler (see chapter 7). When compiling modules separately, `ocamlcp` must be used when compiling the modules (production of `.cmo` files), and can also be used (though this is not strictly necessary) when linking them together.

The amount of profiling information can be controlled through the `-p` option to `ocamlcp`, followed by one or several letters indicating which parts of the program should be profiled:

| | |
|---|---|
| `a` | all options |
| `f` | function calls : a count point is set at the beginning of function bodies |
| `i` | **if . . . then . . . else . . .** : count points are set in both **then** branch and **else** branch |
| `l` | **while, for** loops: a count point is set at the beginning of the loop body |
| `m` | **match** branches: a count point is set at the beginning of the body of each branch |
| `t` | **try . . . with . . .** branches: a count point is set at the beginning of the body of each branch |

For instance, compiling with `ocamlcp -pfilm` profiles function calls, if. . . then. . . else. . . , loops and pattern matching.

Calling `ocamlcp` without the `-p` option defaults to `-p fm`, meaning that only function calls and pattern matching are profiled.

## 14.2 Profiling an execution

Running a bytecode executable file that has been compiled with `ocamlcp` records the execution counts for the specified parts of the program and saves them in a file called `ocamlprof.dump` in the current directory.

The `ocamlprof.dump` file is written only if the program terminates normally (by calling `exit` or by falling through). It is not written if the program terminates with an `uncaught exception`.

If a compatible dump file already exists in the current directory, then the profiling information is accumulated in this dump file. This allows, for instance, the profiling of several executions of a program on different inputs.

## 14.3 Printing profiling information

The `ocamlprof` command produces a source listing of the program modules where execution counts have been inserted as comments. For instance,

```
ocamlprof foo.ml
```

prints the source code for the `foo` module, with comments indicating how many times the functions in this module have been called. Naturally, this information is accurate only if the source file has not been modified since the profiling execution took place.

The following options are recognized by `ocamlprof`:

`-f` *dumpfile*
: Specifies an alternate dump file of profiling information

`-F` *string*
: Specifies an additional string to be output with profiling information. By default, `ocamlprof` will annotate programs with comments of the form `(*` *n* `*)` where *n* is the counter value for a profiling point. With option `-F s`, the annotation will be `(*` *s n* `*)`.

## 14.4 Time profiling

Profiling with `ocamlprof` only records execution counts, not the actual time spent into each function. There is currently no way to perform time profiling on bytecode programs generated by `ocamlc`.

Native-code programs generated by `ocamlopt` can be profiled for time and execution counts using the `-p` option and the standard Unix profiler `gprof`. Just add the `-p` option when compiling and linking the program:

```
ocamlopt -o myprog -p other-options files
./myprog
gprof myprog
```

Caml function names in the output of `gprof` have the following format:

*Module-name_function-name_unique-number*

Other functions shown are either parts of the Caml run-time system or external C functions linked with the program.

The output of `gprof` is described in the Unix manual page for `gprof(1)`. It generally consists of two parts: a “flat” profile showing the time spent in each function and the number of invocation of each function, and a "hierarchical" profile based on the call graph. Currently, only the Intel x86/Linux and Alpha/Digital Unix ports of `ocamlopt` support the two profiles. On other platforms, `gprof` will report only the "flat" profile with just time information. When reading the output of `gprof`, keep in mind that the accumulated times computed by `gprof` are based on heuristics and may not be exact.

Chapter 15

# Interfacing C with Objective Caml

This chapter describes how user-defined primitives, written in C, can be linked with Caml code and called from Caml functions.

## 15.1 Overview and compilation information

### 15.1.1 Declaring primitives

User primitives are declared in an implementation file or `struct`...`end` module expression using the `external` keyword:

> `external` *name* `:` *type* `=` *C-function-name*

This defines the value name *name* as a function with type *type* that executes by calling the given C function. For instance, here is how the `input` primitive is declared in the standard library module `Pervasives`:

```
external input : in_channel -> string -> int -> int -> int
               = "input"
```

Primitives with several arguments are always curried. The C function does not necessarily have the same name as the ML function.

External functions thus defined can be specified in interface files or `sig`...`end` signatures either as regular values

> `val` *name* `:` *type*

thus hiding their implementation as a C function, or explicitly as "manifest" external functions

> `external` *name* `:` *type* `=` *C-function-name*

The latter is slightly more efficient, as it allows clients of the module to call directly the C function instead of going through the corresponding Caml function.

The arity (number of arguments) of a primitive is automatically determined from its Caml type in the `external` declaration, by counting the number of function arrows in the type. For instance, `input` above has arity 4, and the `input` C function is called with four arguments. Similarly,

```
external input2 : in_channel * string * int * int -> int = "input2"
```

has arity 1, and the `input2` C function receives one argument (which is a quadruple of Caml values).

Type abbreviations are not expanded when determining the arity of a primitive. For instance,

```
type int_endo = int -> int
external f : int_endo -> int_endo = "f"
external g : (int -> int) -> (int -> int) = "f"
```

`f` has arity 1, but `g` has arity 2. This allows a primitive to return a functional value (as in the `f` example above): just remember to name the functional return type in a type abbreviation.

### 15.1.2 Implementing primitives

User primitives with arity $n \leq 5$ are implemented by C functions that take $n$ arguments of type `value`, and return a result of type `value`. The type `value` is the type of the representations for Caml values. It encodes objects of several base types (integers, floating-point numbers, strings, ...), as well as Caml data structures. The type `value` and the associated conversion functions and macros are described in details below. For instance, here is the declaration for the C function implementing the `input` primitive:

```
value input(value channel, value buffer, value offset, value length)
{
 ...
}
```

When the primitive function is applied in a Caml program, the C function is called with the values of the expressions to which the primitive is applied as arguments. The value returned by the function is passed back to the Caml program as the result of the function application.

User primitives with arity greater than 5 should be implemented by two C functions. The first function, to be used in conjunction with the bytecode compiler `ocamlc`, receives two arguments: a pointer to an array of Caml values (the values for the arguments), and an integer which is the number of arguments provided. The other function, to be used in conjunction with the native-code compiler `ocamlopt`, takes its arguments directly. For instance, here are the two C functions for the 7-argument primitive `Nat.add_nat`:

```
value add_nat_native(value nat1, value ofs1, value len1,
                     value nat2, value ofs2, value len2,
                     value carry_in)
{
  ...
}
value add_nat_bytecode(value * argv, int argn)
{
  return add_nat_native(argv[0], argv[1], argv[2], argv[3],
                        argv[4], argv[5], argv[6]);
}
```

The names of the two C functions must be given in the primitive declaration, as follows:

```
external name : type =
         bytecode-C-function-name native-code-C-function-name
```

For instance, in the case of `add_nat`, the declaration is:

```
external add_nat: nat -> int -> int -> nat -> int -> int -> int -> int
                = "add_nat_bytecode" "add_nat_native"
```

Implementing a user primitive is actually two separate tasks: on the one hand, decoding the arguments to extract C values from the given Caml values, and encoding the return value as a Caml value; on the other hand, actually computing the result from the arguments. Except for very simple primitives, it is often preferable to have two distinct C functions to implement these two tasks. The first function actually implements the primitive, taking native C values as arguments and returning a native C value. The second function, often called the "stub code", is a simple wrapper around the first function that converts its arguments from Caml values to C values, call the first function, and convert the returned C value to Caml value. For instance, here is the stub code for the `input` primitive:

```
value input(value channel, value buffer, value offset, value length)
{
  return Val_long(getblock((struct channel *) channel,
                           &Byte(buffer, Long_val(offset)),
                           Long_val(length)));
}
```

(Here, `Val_long`, `Long_val` and so on are conversion macros for the type `value`, that will be described later.) The hard work is performed by the function `getblock`, which is declared as:

```
long getblock(struct channel * channel, char * p, long n)
{
  ...
}
```

To write C code that operates on Objective Caml values, the following include files are provided:

| Include file | Provides |
|---|---|
| `caml/mlvalues.h` | definition of the `value` type, and conversion macros |
| `caml/alloc.h` | allocation functions (to create structured Caml objects) |
| `caml/memory.h` | miscellaneous memory-related functions and macros (for GC interface, in-place modification of structures, etc). |
| `caml/fail.h` | functions for raising exceptions (see section 15.4.6) |
| `caml/callback.h` | callback from C to Caml (see section 15.6). |

These files reside in the `caml/` subdirectory of the Objective Caml standard library directory (usually `/usr/local/lib/ocaml`).

### 15.1.3 Linking C code with Caml code

The Objective Caml runtime system comprises three main parts: the bytecode interpreter, the memory manager, and a set of C functions that implement the primitive operations. Some bytecode instructions are provided to call these C functions, designated by their offset in a table of functions (the table of primitives).

In the default mode, the Caml linker produces bytecode for the standard runtime system, with a standard set of primitives. References to primitives that are not in this standard set result in the "unavailable C primitive" error.

In the "custom runtime" mode, the Caml linker scans the object files and determines the set of required primitives. Then, it builds a suitable runtime system, by calling the native code linker with:

- the table of the required primitives
- a library that provides the bytecode interpreter, the memory manager, and the standard primitives
- libraries and object code files (`.o` files) mentioned on the command line for the Caml linker, that provide implementations for the user's primitives.

This builds a runtime system with the required primitives. The Caml linker generates bytecode for this custom runtime system. The bytecode is appended to the end of the custom runtime system, so that it will be automatically executed when the output file (custom runtime + bytecode) is launched.

To link in "custom runtime" mode, execute the `ocamlc` command with:

- the `-custom` option
- the names of the desired Caml object files (`.cmo` and `.cma` files)
- the names of the C object files and libraries (`.o` and `.a` files) that implement the required primitives. (Under Unix, a library named `lib`*name*`.a` residing in one of the standard library directories can also be specified as `-cclib -l`*name*.)

If you are using the native-code compiler `ocamlopt`, the `-custom` flag is not needed, as the final linking phase of `ocamlopt` always builds a standalone executable. To build a mixed Caml/C executable, execute the `ocamlopt` command with:

- the names of the desired Caml native object files (`.cmx` and `.cmxa` files)
- the names of the C object files and libraries (`.o` and `.a` files) that implement the required primitives.

### 15.1.4 Building standalone custom runtime systems

It is sometimes inconvenient to build a custom runtime system each time Caml code is linked with C libraries, like `ocamlc -custom` does. For one thing, the building of the runtime system is slow on some systems (that have bad linkers or slow remote file systems); for another thing, the platform-independence of bytecode files is lost, forcing to perform one `ocamlc -custom` link per platform of interest.

An alternative to `ocamlc -custom` is to build separately a custom runtime system integrating the desired C libraries, then generate "pure" bytecode executables (not containing their own runtime system) that can run on this custom runtime. This is achieved by the `-make_runtime` and `-use_runtime` flags to `ocamlc`. For example, to build a custom runtime system integrating the C parts of the "unix" and "threads" libraries, do:

```
        ocamlc -make-runtime -o /home/me/ocamlunixrun unix.cma threads.cma \
                -cclib -lunix -cclib -lthreads
```

To generate a bytecode executable that runs on this runtime system, do:

```
        ocamlc -use-runtime /home/me/ocamlunixrun -o myprog \
                unix.cma threads.cma your .cmo and .cma files
```

The bytecode executable `myprog` can then be launched as usual: `myprog` *args* or `/home/me/ocamlunixrun myprog` *args*.

Notice that the bytecode libraries `unix.cma` and `threads.cma` must be given twice: when building the runtime system (so that `ocamlc` knows which C primitives from `-lunix` and `-lthreads` are required) and also when building the bytecode executable (so that the bytecode from `unix.cma` and `threads.cma` is actually linked in).

## 15.2 The `value` type

All Caml objects are represented by the C type `value`, defined in the include file `caml/mlvalues.h`, along with macros to manipulate values of that type. An object of type `value` is either:

- an unboxed integer
- a pointer to a block inside the heap (such as the blocks allocated through one of the `alloc_*` functions below)
- a pointer to an object outside the heap (e.g., a pointer to a block allocated by `malloc`, or to a C variable).

### 15.2.1 Integer values

Integer values encode 31-bit signed integers (63-bit on 64-bit architectures). They are unboxed (unallocated).

### 15.2.2 Blocks

Blocks in the heap are garbage-collected, and therefore have strict structure constraints. Each block includes a header containing the size of the block (in words), and the tag of the block. The tag governs how the contents of the blocks are structured. A tag lower than `No_scan_tag` indicates a structured block, containing well-formed values, which is recursively traversed by the garbage collector. A tag greater than or equal to `No_scan_tag` indicates a raw block, whose contents are not scanned by the garbage collector. For the benefits of ad-hoc polymorphic primitives such as equality and structured input-output, structured and raw blocks are further classified according to their tags as follows:

| Tag | Contents of the block |
|---|---|
| 0 to `No_scan_tag` − 1 | A structured block (an array of Caml objects). Each field is a `value`. |
| `Closure_tag` | A closure representing a functional value. The first word is a pointer to a piece of code, the remaining words are `value` containing the environment. |
| `String_tag` | A character string. |
| `Double_tag` | A double-precision floating-point number. |
| `Double_array_tag` | An array or record of double-precision floating-point numbers. |
| `Abstract_tag` | A block representing an abstract datatype. |
| `Final_tag` | A block representing an abstract datatype with a "finalization" function, to be called when the block is deallocated. |

### 15.2.3 Pointers outside the heap

Any word-aligned pointer to an address outside the heap can be safely cast to and from the type `value`. This includes pointers returned by `malloc`, and pointers to C variables (of size at least one word) obtained with the `&` operator.

## 15.3 Representation of Caml data types

This section describes how Caml data types are encoded in the `value` type.

### 15.3.1 Atomic types

| Caml type | Encoding |
|---|---|
| `int` | Unboxed integer values. |
| `char` | Unboxed integer values (ASCII code). |
| `float` | Blocks with tag `Double_tag`. |
| `string` | Blocks with tag `String_tag`. |

### 15.3.2 Tuples and records

Tuples are represented by pointers to blocks, with tag 0.

Records are also represented by zero-tagged blocks. The ordering of labels in the record type declaration determines the layout of the record fields: the value associated to the label declared first is stored in field 0 of the block, the value associated to the label declared next goes in field 1, and so on.

As an optimization, records whose fields all have static type `float` are represented as arrays of floating-point numbers, with tag `Double_array_tag`. (See the section below on arrays.)

### 15.3.3 Arrays

Arrays of integers and pointers are represented like tuples, that is, as pointers to blocks tagged 0. They are accessed with the `Field` macro for reading and the `modify` function for writing.

Arrays of floating-point numbers (type `float array`) have a special, unboxed, more efficient representation. These arrays are represented by pointers to blocks with tag `Double_array_tag`. They should be accessed with the `Double_field` and `Store_double_field` macros.

### 15.3.4 Concrete types

Constructed terms are represented either by unboxed integers (for constant constructors) or by blocks whose tag encode the constructor (for non-constant constructors). The constant constructors and the non-constant constructors for a given concrete type are numbered separately, starting from 0, in the order in which they appear in the concrete type declaration. Constant constructors are represented by unboxed integers equal to the constructor number. Non-constant constructors declared with a $n$-tuple as argument are represented by a block of size $n$, tagged with the constructor number; the $n$ fields contain the components of its tuple argument. Other non-constant constructors are represented by a block of size 1, tagged with the constructor number; the field 0 contains the value of the constructor argument. Example:

| Constructed term | Representation |
|---|---|
| `()` | `Val_int(0)` |
| `false` | `Val_int(0)` |
| `true` | `Val_int(1)` |
| `[]` | `Val_int(0)` |
| `h::t` | Block with size = 2 and tag = 0; first field contains `h`, second field `t` |

As a convenience, `caml/mlvalues.h` defines the macros `Val_unit`, `Val_false` and `Val_true` to refer to `()`, `false` and `true`.

### 15.3.5 Objects

Objects are represented as zero-tagged blocks. The first field of the block refers to the object class and associated method suite, in a format that cannot easily be exploited from C. The remaining fields of the object contain the values of the instance variables of the object. Instance variables are stored in the order in which they appear in the class definition (taking inherited classes into account).

## 15.4 Operations on values

### 15.4.1 Kind tests

- `Is_long(`$v$`)` is true if value $v$ is an immediate integer, false otherwise
- `Is_block(`$v$`)` is true if value $v$ is a pointer to a block, and false if it is an immediate integer.

### 15.4.2 Operations on integers

- `Val_long(`$l$`)` returns the value encoding the `long int` $l$.
- `Long_val(`$v$`)` returns the `long int` encoded in value $v$.
- `Val_int(`$i$`)` returns the value encoding the `int` $i$.
- `Int_val(`$v$`)` returns the `int` encoded in value $v$.
- `Val_bool(`$x$`)` returns the Caml boolean representing the truth value of the C integer $x$.
- `Bool_val(`$v$`)` returns 0 if $v$ is the Caml boolean `false`, 1 if $v$ is `true`.
- `Val_true`, `Val_false` represent the Caml booleans `true` and `false`.

### 15.4.3 Accessing blocks

- `Wosize_val(`$v$`)` returns the size of the block $v$, in words, excluding the header.
- `Tag_val(`$v$`)` returns the tag of the block $v$.
- `Field(`$v$`, `$n$`)` returns the value contained in the $n^{\text{th}}$ field of the structured block $v$. Fields are numbered from 0 to `Wosize_val`$(v) - 1$.
- `Store_field(`$b$`, `$n$`, `$v$`)` stores the value $v$ in the field number $n$ of value $b$, which must be a structured block.
- `Code_val(`$v$`)` returns the code part of the closure $v$.
- `string_length(`$v$`)` returns the length (number of characters) of the string $v$.
- `Byte(`$v$`, `$n$`)` returns the $n^{\text{th}}$ character of the string $v$, with type `char`. Characters are numbered from 0 to `string_length`$(v) - 1$.
- `Byte_u(`$v$`, `$n$`)` returns the $n^{\text{th}}$ character of the string $v$, with type `unsigned char`. Characters are numbered from 0 to `string_length`$(v) - 1$.
- `String_val(`$v$`)` returns a pointer to the first byte of the string $v$, with type `char *`. This pointer is a valid C string: there is a null character after the last character in the string. However, Caml strings can contain embedded null characters, that will confuse the usual C functions over strings.
- `Double_val(`$v$`)` returns the floating-point number contained in value $v$, with type `double`.
- `Double_field(`$v$`, `$n$`)` returns the $n^{\text{th}}$ element of the array of floating-point numbers $v$ (a block tagged `Double_array_tag`).
- `Store_double_field(`$v$`, `$n$`, `$d$`)` stores the double precision floating-point number $d$ in the $n^{\text{th}}$ element of the array of floating-point numbers $v$.

The expressions `Field(`$v$`, `$n$`)`, `Byte(`$v$`, `$n$`)` and `Byte_u(`$v$`, `$n$`)` are valid l-values. Hence, they can be assigned to, resulting in an in-place modification of value $v$. Assigning directly to `Field(`$v$`, `$n$`)` must be done with care to avoid confusing the garbage collector (see below).

### 15.4.4 Allocating blocks

#### Simple interface

- `Atom(`*t*`)` returns an "atom" (zero-sized block) with tag *t*. Zero-sized blocks are preallocated outside of the heap. It is incorrect to try and allocate a zero-sized block using the functions below. For instance, `Atom(0)` represents the empty array.
- `alloc(`*n*`, `*t*`)` returns a fresh block of size *n* with tag *t*. If *t* is less than `No_scan_tag`, then the fields of the block are initialized with a valid value in order to satisfy the GC constraints.
- `alloc_tuple(`*n*`)` returns a fresh block of size *n* words, with tag 0.
- `alloc_string(`*n*`)` returns a string value of length *n* characters. The string initially contains garbage.
- `copy_string(`*s*`)` returns a string value containing a copy of the null-terminated C string *s* (a `char *`).
- `copy_double(`*d*`)` returns a floating-point value initialized with the `double` *d*.
- `alloc_array(`*f*`, `*a*`)` allocates an array of values, calling function *f* over each element of the input array *a* to transform it into a value. The array *a* is an array of pointers terminated by the null pointer. The function *f* receives each pointer as argument, and returns a value. The zero-tagged block returned by `alloc_array(`*f*`, `*a*`)` is filled with the values returned by the successive calls to *f*. (This function must not be used to build an array of floating-point numbers.)
- `copy_string_array(`*p*`)` allocates an array of strings, copied from the pointer to a string array *p* (a `char **`).

#### Low-level interface

The following functions are slightly more efficient than `alloc`, but also much more difficult to use.

From the standpoint of the allocation functions, blocks are divided according to their size as zero-sized blocks, small blocks (with size less than or equal to `Max_young_wosize`), and large blocks (with size greater than `Max_young_wosize`). The constant `Max_young_wosize` is declared in the include file `mlvalues.h`. It is guaranteed to be at least 64 (words), so that any block with constant size less than or equal to 64 can be assumed to be small. For blocks whose size is computed at run-time, the size must be compared against `Max_young_wosize` to determine the correct allocation procedure.

- `alloc_small(`*n*`, `*t*`)` returns a fresh small block of size $n \leq$ `Max_young_wosize` words, with tag *t*. If this block is a structured block (i.e. if $t <$ `No_scan_tag`), then the fields of the block (initially containing garbage) must be initialized with legal values (using direct assignment to the fields of the block) before the next allocation.
- `alloc_shr(`*n*`, `*t*`)` returns a fresh block of size *n*, with tag *t*. The size of the block can be greater than `Max_young_wosize`. (It can also be smaller, but in this case it is more

efficient to call `alloc_small` instead of `alloc_shr`.) If this block is a structured block (i.e. if $t <$ `No_scan_tag`), then the fields of the block (initially containing garbage) must be initialized with legal values (using the `initialize` function described below) before the next allocation.

### 15.4.5 Finalized blocks

Blocks with tag `Final_tag` have an attached C finalization function that is called when the block becomes unreachable and is about to be reclaimed. The finalization function occupies the first word of the allocated block; the remaining words can contain arbitrary raw data (but not Caml pointers, since `Final_tag` is greater than `No_scan_tag`.

Finalized blocks must be allocated via the `alloc_final` function. `alloc_final(`*n*`,` *f*`,` *used*`,` *max*`)` returns a fresh finalized block of size *n* words, with finalization function *f*.

The two parameters *used* and *max* are used to control the speed of garbage collection when the finalized object contains pointers to out-of-heap resources. Generally speaking, the Caml incremental major collector adjusts its speed relative to the allocation rate of the program. The faster the program allocates, the harder the GC works in order to reclaim quickly unreachable blocks and avoid having large amount of "floating garbage" (unreferenced objects that the GC has not yet collected).

Normally, the allocation rate is measured by counting the in-heap size of allocated blocks. However, it often happens that finalized objects contain pointers to out-of-heap memory blocks and other resources (such as file descriptors, X Windows bitmaps, etc.). For those blocks, the in-heap size of blocks is not a good measure of the quantity of resources allocated by the program.

The two arguments *used* and *max* give the GC an idea of how much out-of-heap resources are consumed by the finalized block being allocated: you give the amount of resources allocated to this object as parameter *used*, and the maximum amount that you want to see in floating garbage as parameter *max*. The units are arbitrary: the GC cares only about the ratio $used/max$.

For instance, if you are allocating a finalized block holding an X Windows bitmap of $w$ by $h$ pixels, and you'd rather not have more than 1 mega-pixels of unreclaimed bitmaps, specify $used = w * h$ and $max = 1000000$.

If your finalized blocks contain no pointers to out-of-heap resources, or if the previous discussion made little sense to you, just take $used = 0$ and $max = 1$. But if you later find that the finalization functions are not called "often enough", consider increasing the $used/max$ ratio.

### 15.4.6 Raising exceptions

Two functions are provided to raise two standard exceptions:

- `failwith(`*s*`)`, where *s* is a null-terminated C string (with type `char *`), raises exception `Failure` with argument *s*.

- `invalid_argument(`*s*`)`, where *s* is a null-terminated C string (with type `char *`), raises exception `Invalid_argument` with argument *s*.

Raising arbitrary exceptions from C is more delicate: the exception identifier is dynamically allocated by the Caml program, and therefore must be communicated to the C function using the registration facility described below in section 15.6.3. Once the exception identifier is recovered in C, the following functions actually raise the exception:

- `raise_constant(`*id*`)` raises the exception *id* with no argument;
- `raise_with_arg(`*id*`, `*v*`)` raises the exception *id* with the Caml value *v* as argument;
- `raise_with_string(`*id*`, `*s*`)`, where *s* is a null-terminated C string, raises the exception *id* with a copy of the C string *s* as argument.

## 15.5 Living in harmony with the garbage collector

Unused blocks in the heap are automatically reclaimed by the garbage collector. This requires some cooperation from C code that manipulates heap-allocated blocks.

### 15.5.1 Simple interface

All the macros described in this section are declared in the `memory.h` header file.

**Rule 1** *A function that has parameters or local variables of type* `value` *must begin with a call to one of the* `CAMLparam` *macros and return with* `CAMLreturn`*.*

There are six `CAMLparam` macros: `CAMLparam0` to `CAMLparam5`, which take zero to five arguments respectively. If your function has fewer than 5 parameters of type `value`, use the corresponding macros with these parameters as arguments. If your function has more than 5 parameters of type `value`, use `CAMLparam5` with five of these parameters, and use one or more calls to the `CAMLxparam` macros for the remaining parameters (`CAMLxparam0` to `CAMLxparam5`).

The macro `CAMLreturn` is used as a direct replacement for the C keyword `return`. All occurences of `return` must be replaced by `CAMLreturn`, including the implicit `return` at the end of a procedure (`void`-returning function).

Example:

```
void foo (value v1, value v2, value v3)
{
  CAMLparam3 (v1, v2, v3);
  ...
  CAMLreturn;
}
```

**Note:** if your function is a primitive with more than 5 arguments for use with the byte-code runtime, its arguments are not `value`s and must not be declared (they have types `value *` and `int`).

**Rule 2** *Local variables of type* `value` *must be declared with one of the* `CAMLlocal` *macros. Arrays of* `value`*s are declared with* `CAMLlocalN`*.*

The macros `CAMLlocal1` to `CAMLlocal5` declare and initialize one to five local variables of type `value`. The variable names are given as arguments to the macros. `CAMLlocalN(`*x*`,` *n*`)` declares and initializes a local variable of type `value [`*n*`]`. You can use several calls to these macros if you have more than 5 local variables. You can also use them in nested C blocks within the function.

Example:

```
value bar (value v1, value v2, value v3)
{
  CAMLparam3 (v1, v2, v3);
  CAMLlocal1 (result);
  result = alloc (3, 0);
  ...
  CAMLreturn result;
}
```

**Rule 3** *Assignments to the fields of structured blocks must be done with the* `Store_field` *macro.*

`Store_field (`*b*`,` *n*`,` *v*`)` stores the value *v* in the field number *n* of value *b*, which must be a block (i.e. `Is_block(`*b*`)` must be true).

Example:

```
value bar (value v1, value v2, value v3)
{
  CAMLparam3 (v1, v2, v3);
  CAMLlocal1 (result);
  result = alloc (3, 0);
  Store_field (result, 0, v1);
  Store_field (result, 1, v2);
  Store_field (result, 2, v3);
  CAMLreturn result;
}
```

**Rule 4** *Global variables containing values must be registered with the garbage collector using the* `register_global_root` *function.*

Registration of a global variable `v` is achieved by calling `register_global_root(&v)` just before a valid value is stored in `v` for the first time.

A registered global variable `v` can be un-registered by calling `remove_global_root(&v)`.

**Note:** The `CAML` macros use identifiers (local variables, type identifiers, structure tags) that start with `caml__`. Do not use any identifier starting with `caml__` in your programs.

### 15.5.2 Low-level interface

We now give the GC rules corresponding to the low-level allocation functions `alloc_small` and `alloc_shr`. You can ignore those rules if you stick to the simplified allocation function `alloc`.

**Rule 5** *After a structured block (a block with tag less than* `No_scan_tag`*) is allocated with the low-level functions, all fields of this block must be filled with well-formed values before the next allocation operation. If the block has been allocated with* `alloc_small`*, filling is performed by direct assignment to the fields of the block:*

`Field(`$v$`, `$n$`) = `$v_n$`;`

*If the block has been allocated with* `alloc_shr`*, filling is performed through the* `initialize` *function:*

`initialize(&Field(`$v$`, `$n$`), `$v_n$`);`

The next allocation can trigger a garbage collection. The garbage collector assumes that all structured blocks contain well-formed values. Newly created blocks contain random data, which generally do not represent well-formed values.

If you really need to allocate before the fields can receive their final value, first initialize with a constant value (e.g. `Val_long(0)`), then allocate, then modify the fields with the correct value (see rule 6).

**Rule 6** *Direct assignment to a field of a block, as in*

`Field(`$v$`, `$n$`) = `$w$`;`

*is safe only if $v$ is a block newly allocated by* `alloc_small`*; that is, if no allocation took place between the allocation of $v$ and the assignment to the field. In all other cases, never assign directly. If the block has just been allocated by* `alloc_shr`*, use* `initialize` *to assign a value to a field for the first time:*

`initialize(&Field(`$v$`, `$n$`), `$w$`);`

*Otherwise, you are updating a field that previously contained a well-formed value; then, call the* `modify` *function:*

`modify(&Field(`$v$`, `$n$`), `$w$`);`

To illustrate the rules above, here is a C function that builds and returns a list containing the two integers given as parameters. First, we write it using the simplified allocation functions:

```
value alloc_list_int(int i1, int i2)
{
  CAMLparam0;
  CAMLlocal2 (result, r);

  r = alloc(2, 0);                        /* Allocate a cons cell */
  Store_field(r, 0, Val_int(i2));         /* car = the integer i2 */
  Store_field(r, 1, Val_int(0));          /* cdr = the empty list [] */
  result = alloc(2, 0);                   /* Allocate the other cons cell */
  Store_field(result, 0, Val_int(i1));    /* car = the integer i1 */
  Store_field(result, 1, r);              /* cdr = the first cons cell */
  CAMLreturn result;
}
```

Here, the registering of `result` is not strictly needed, because no allocation takes place after it gets its value, but it's easier and safer to simply register all the local variables that have type `value`.

Here is the same function written using the low-level allocation functions. We notice that the cons cells are small blocks and can be allocated with `alloc_small`, and filled by direct assignments on their fields.

```
value alloc_list_int(int i1, int i2)
{
  CAMLparam0;
  CAMLlocal2 (result, r);

  r = alloc_small(2, 0);                  /* Allocate a cons cell */
  Field(r, 0) = Val_int(i2);              /* car = the integer i2 */
  Field(r, 1) = Val_int(0);               /* cdr = the empty list [] */
  result = alloc_small(2, 0);             /* Allocate the other cons cell */
  Field(result, 0) = Val_int(i1);         /* car = the integer i1 */
  Field(result, 1) = r;                   /* cdr = the first cons cell */
  CAMLreturn result;
}
```

In the two examples above, the list is built bottom-up. Here is an alternate way, that proceeds top-down. It is less efficient, but illustrates the use of `modify`.

```
value alloc_list_int(int i1, int i2)
{
  CAMLparam0;
  CAMLlocal2 (tail, r);

  r = alloc_small(2, 0);                  /* Allocate a cons cell */
  Field(r, 0) = Val_int(i1);              /* car = the integer i1 */
  Field(r, 1) = Val_int(0);               /* A dummy value
  tail = alloc_small(2, 0);               /* Allocate the other cons cell */
  Field(tail, 0) = Val_int(i2);           /* car = the integer i2 */
  Field(tail, 1) = Val_int(0);            /* cdr = the empty list [] */
  modify(&Field(r, 1), tail);             /* cdr of the result = tail */
  return r;
}
```

It would be incorrect to perform `Field(r, 1) = tail` directly, because the allocation of `tail` has taken place since `r` was allocated. `tail` is not registered as a root because there is no allocation between the assignment where it takes its value and the `modify` statement that uses the value.

## 15.6 Callbacks from C to Caml

So far, we have described how to call C functions from Caml. In this section, we show how C functions can call Caml functions, either as callbacks (Caml calls C which calls Caml), or because the main program is written in C.

### 15.6.1 Applying Caml closures from C

C functions can apply Caml functional values (closures) to Caml values. The following functions are provided to perform the applications:

- `callback(`*f, a*`)` applies the functional value *f* to the value *a* and return the value returned by *f*.
- `callback2(`*f, a, b*`)` applies the functional value *f* (which is assumed to be a curried Caml function with two arguments) to *a* and *b*.
- `callback3(`*f, a, b, c*`)` applies the functional value *f* (a curried Caml function with three arguments) to *a*, *b* and *c*.
- `callbackN(`*f, n, args*`)` applies the functional value *f* to the *n* arguments contained in the array of values *args*.

If the function *f* does not return, but raises an exception that escapes the scope of the application, then this exception is propagated to the next enclosing Caml code, skipping over the C code. That is, if a Caml function *f* calls a C function *g* that calls back a Caml function *h* that raises a stray exception, then the execution of *g* is interrupted and the exception is propagated back into *f*.

If the C code wishes to catch exceptions escaping the Caml function, it can use the functions `callback_exn`, `callback2_exn`, `callback3_exn`, `callbackN_exn`. These functions take the same arguments as their non-`_exn` counterparts, but catch escaping exceptions and return them to the C code. The return value *v* of the `callback*_exn` functions must be tested with the macro `Is_exception_result(`*v*`)`. If the macro returns "false", no exception occured, and *v* is the value returned by the Caml function. If `Is_exception_result(`*v*`)` returns "true", an exception escaped, and its value (the exception descriptor) can be recovered using `Extract_exception(`*v*`)`.

### 15.6.2 Registering Caml closures for use in C functions

The main difficulty with the `callback` functions described above is obtaining a closure to the Caml function to be called. For this purpose, Objective Caml provides a simple registration mechanism, by which Caml code can register Caml functions under some global name, and then C code can retrieve the corresponding closure by this global name.

On the Caml side, registration is performed by evaluating `Callback.register` *n v*. Here, *n* is the global name (an arbitrary string) and *v* the Caml value. For instance:

```
let f x = print_string "f is applied to "; print_int n; print_newline()
let _ = Callback.register "test function" f
```

On the C side, a pointer to the value registered under name *n* is obtained by calling `caml_named_value(`*n*`)`. The returned pointer must then be dereferenced to recover the actual Caml value. If no value is registered under the name *n*, the null pointer is returned. For example, here is a C wrapper that calls the Caml function `f` above:

```
void call_caml_f(int arg)
{
    callback(*caml_named_value("test function"), Val_int(arg));
}
```

The pointer returned by `caml_named_value` is constant and can safely be cached in a C variable to avoid repeated name lookups. On the other hand, the value pointed to can change during garbage collection and must always be recomputed at the point of use. Here is a more efficient variant of `call_caml_f` above that calls `caml_named_value` only once:

```
void call_caml_f(int arg)
{
    static value * closure_f = NULL;
    if (closure_f == NULL) {
        /* First time around, look up by name */
        closure_f = caml_named_value("test function");
    }
    callback(*closure_f, Val_int(arg));
}
```

### 15.6.3 Registering Caml exceptions for use in C functions

The registration mechanism described above can also be used to communicate exception identifiers from Caml to C. The Caml code registers the exception by evaluating `Callback.register_exception` *n exn*, where *n* is an arbitrary name and *exn* is an exception value of the exception to register. For example:

```
exception Error of string
let _ = Callback.register_exception "test exception" (Error "any string")
```

The C code can then recover the exception identifier using `caml_named_value` and pass it as first argument to the functions `raise_constant`, `raise_with_arg`, and `raise_with_string` (described in section 15.4.6) to actually raise the exception. For example, here is a C function that raises the `Error` exception with the given argument:

```
void raise_error(char * msg)
{
    raise_with_string(*caml_named_value("test exception"), msg);
}
```

### 15.6.4 Main program in C

In normal operation, a mixed Caml/C program starts by executing the Caml initialization code, which then may proceed to call C functions. We say that the main program is the Caml code. In some applications, it is desirable that the C code plays the role of the main program, calling Caml functions when needed. This can be achieved as follows:

- The C part of the program must provide a `main` function, which will override the default `main` function provided by the Caml runtime system. Execution will start in the user-defined `main` function just like for a regular C program.
- At some point, the C code must call `caml_main(argv)` to initialize the Caml code. The `argv` argument is a C array of strings (type `char **`) which represents the command-line arguments,

as passed as second argument to `main`. The Caml array `Sys.argv` will be initialized from this parameter. For the bytecode compiler, `argv[0]` and `argv[1]` are also consulted to find the file containing the bytecode.

- The call to `caml_main` initializes the Caml runtime system, loads the bytecode (in the case of the bytecode compiler), and executes the initialization code of the Caml program. Typically, this initialization code registers callback functions using `Callback.register`. Once the Caml initialization code is complete, control returns to the C code that called `caml_main`.
- The C code can then invoke Caml functions using the callback mechanism (see section 15.6.1).

### 15.6.5 Embedding the Caml code in the C code

The bytecode compiler in custom runtime mode (`ocamlc -custom`) normally appends the bytecode to the executable file containing the custom runtime. This has two consequences. First, the final linking step must be performed by `ocamlc`. Second, the Caml runtime library must be able to find the name of the executable file from the command-line arguments. When using `caml_main(argv)` as in section 15.6.4, this means that `argv[0]` or `argv[1]` must contain the executable file name.

An alternative is to embed the bytecode in the C code. The `-output-obj` option to `ocamlc` is provided for this purpose. It causes the `ocamlc` compiler to output a C object file (`.o` file) containing the bytecode for the Caml part of the program, as well as a `caml_startup` function. The C object file produced by `ocamlc -output-obj` can then be linked with C code using the standard C compiler, or stored in a C library.

The `caml_startup` function must be called from the main C program in order to initialize the Caml runtime and execute the Caml initialization code. Just like `caml_main`, it takes one `argv` parameter containing the command-line parameters. Unlike `caml_main`, this `argv` parameter is used only to initialize `Sys.argv`, but not for finding the name of the executable file.

The native-code compiler `ocamlopt` also supports the `-output-obj` option, causing it to output a C object file containing the native code for all Caml modules on the command-line, as well as the Caml startup code. Initialization is performed by calling `caml_startup` as in the case of the bytecode compiler.

For the final linking phase, in addition to the object file produced by `-output-obj`, you will have to provide the Objective Caml runtime library (`libcamlrun.a` for bytecode, `libasmrun.a` for native-code), as well as all C libraries that are required by the Caml libraries used. For instance, assume the Caml part of your program uses the Unix library. With `ocamlc`, you should do:

```
ocamlc -output-obj -o camlcode.o unix.cma other .cmo and .cma files
cc -o myprog C objects and libraries \
   camlcode.o -L/usr/local/lib/ocaml -lunix -lcamlrun
```

With `ocamlopt`, you should do:

```
ocamlopt -output-obj -o camlcode.o unix.cmxa other .cmx and .cmxa files
cc -o myprog C objects and libraries \
   camlcode.o -L/usr/local/lib/ocaml -lunix -lasmrun
```

**Warning:** On some ports, special options are required on the final linking phase that links together the object file produced by the `-output-obj` option and the remainder of the program. Those options are shown in the configuration file `config/Makefile` generated during compilation of Objective Caml, as the variables `BYTECCLINKOPTS` (for object files produced by `ocamlc -output-obj`) and `NATIVECCLINKOPTS` (for object files produced by `ocamlopt -output-obj`). Currently, the only ports that require special attention are:

- Digital Unix on the Alpha: object files produced by `ocamlc -output-obj` must be linked with the `-taso` option. This is not necessary for object files produced by `ocamlopt -output-obj`.
- Windows NT: the object file produced by Objective Caml have been compiled with the `/MT` flag, and therefore all other object files linked with it should also be compiled with `/MT`.

## 15.7 A complete example

This section outlines how the functions from the Unix `curses` library can be made available to Objective Caml programs. First of all, here is the interface `curses.mli` that declares the `curses` primitives and data types:

```
type window                   (* The type "window" remains abstract *)
external initscr: unit -> window = "curses_initscr"
external endwin: unit -> unit = "curses_endwin"
external refresh: unit -> unit = "curses_refresh"
external wrefresh : window -> unit = "curses_wrefresh"
external newwin: int -> int -> int -> int -> window = "curses_newwin"
external mvwin: window -> int -> int -> unit = "curses_mvwin"
external addch: char -> unit = "curses_addch"
external mvwaddch: window -> int -> int -> char -> unit = "curses_mvwaddch"
external addstr: string -> unit = "curses_addstr"
external mvwaddstr: window -> int -> int -> string -> unit = "curses_mvwaddstr"
(* lots more omitted *)
```

To compile this interface:

```
        ocamlc -c curses.mli
```

To implement these functions, we just have to provide the stub code; the core functions are already implemented in the `curses` library. The stub code file, `curses.o`, looks like:

```
#include <curses.h>
#include <mlvalues.h>

value curses_initscr(value unit)
{
  return (value) initscr();     /* OK to coerce directly from WINDOW * to value
                                   since that's a block created by malloc() */
}
```

```
value curses_wrefresh(value win)
{
  wrefresh((WINDOW *) win);
  return Val_unit;
}

value curses_newwin(value nlines, value ncols, value x0, value y0)
{
  return (value) newwin(Int_val(nlines), Int_val(ncols),
                        Int_val(x0), Int_val(y0));
}

value curses_addch(value c)
{
  addch(Int_val(c));            /* Characters are encoded like integers */
  return Val_unit;
}

value curses_addstr(value s)
{
  addstr(String_val(s));
  return Val_unit;
}

/* This goes on for pages. */
```

The file `curses.c` can be compiled with:

```
        cc -c -I/usr/local/lib/ocaml curses.c
```

or, even simpler,

```
        ocamlc -c curses.c
```

(When passed a `.c` file, the `ocamlc` command simply calls the C compiler on that file, with the right `-I` option.)

Now, here is a sample Caml program `test.ml` that uses the `curses` module:

```
open Curses
let main_window = initscr () in
let small_window = newwin 10 5 20 10 in
  mvwaddstr main_window 10 2 "Hello";
  mvwaddstr small_window 4 3 "world";
  refresh();
  for i = 1 to 100000 do () done;
  endwin()
```

To compile this program, run:

```
        ocamlc -c test.ml
```

Finally, to link everything together:

```
        ocamlc -custom -o test test.cmo curses.o -cclib -lcurses
```

## 15.8 Advanced example with callbacks

This section illustrates the callback facilities described in section 15.6. We are going to package some Caml functions in such a way that they can be linked with C code and called from C just like any C functions. The Caml functions are defined in the following `mod.ml` Caml source:

```
(* File mod.ml -- some ''useful'' Caml functions *)

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

let format_result n = Printf.sprintf "Result is: %d\n" n

(* Export those two functions to C *)

let _ = Callback.register "fib" fib
let _ = Callback.register "format_result" format_result
```

Here is the C stub code for calling these functions from C:

```
/* File modwrap.c -- wrappers around the Caml functions */

#include <stdio.h>
#include <string.h>
#include <caml/mlvalues.h>
#include <caml/callback.h>

int fib(int n)
{
  static value * fib_closure = NULL;
  if (fib_closure == NULL) fib_closure = caml_named_value("fib");
  return Int_val(callback(*fib_closure, Val_int(n)));
}

char * format_result(int n)
{
  static value * format_result_closure = NULL;
  if (format_result_closure == NULL)
    format_result_closure = caml_named_value("format_result");
  return strdup(String_val(callback(*format_result_closure, Val_int(n))));
  /* We copy the C string returned by String_val to the C heap
     so that it remains valid after garbage collection. */
}
```

We now compile the Caml code to a C object file and put it in a C library along with the stub code in `modwrap.c` and the Caml runtime system:

```
        ocamlc -custom -output-obj -o modcaml.o mod.ml
        ocamlc -c modwrap.c
        cp /usr/local/lib/ocaml/libcamlrun.a mod.a
        ar r mod.a modcaml.o modwrap.o
```

(One can also use `ocamlopt -output-obj` instead of `ocamlc -custom -output-obj`. In this case, replace `libcamlrun.a` (the bytecode runtime library) by `libasmrun.a` (the native-code runtime library).)

Now, we can use the two fonctions `fib` and `format_result` in any C program, just like regular C functions. Just remember to call `caml_startup` once before.

```
/* File main.c -- a sample client for the Caml functions */

#include <stdio.h>

int main(int argc, char ** argv)
{
  int result;

  /* Initialize Caml code */
  caml_startup(argv);
  /* Do some computation */
  result = fib(10);
  printf("fib(10) = %s\n", format_result(result));
  return 0;
}
```

To build the whole program, just invoke the C compiler as follows:

```
        cc -o prog main.c mod.a
```

# Part IV

# The Objective Caml library

# Chapter 16

# The core library

This chapter describes the functions provided by the Caml Light core library module: module `Pervasives`. This module is special in two ways:

- It is automatically linked with the user's object code files by the `ocamlc` command (chapter 7).
- It is automatically "opened" when a compilation starts, or when the toplevel system is launched. Hence, it is possible to use unqualified identifiers to refer to the functions provided by the `Pervasives` module, without adding a `open Pervasives` directive.

## Conventions

The declarations from the signature of the `Pervasives` module are printed one by one in typewriter font, followed by a short comment. All modules and the identifiers they export are indexed at the end of this report.

## 16.1 Module `Pervasives`: the initially opened module

This module provides the built-in types (numbers, booleans, strings, exceptions, references, lists, arrays, input-output channels, ...) and the basic operations over these types.

This module is automatically opened at the beginning of each compilation. All components of this module can therefore be referred by their short name, without prefixing them by `Pervasives`.

### Predefined types

`type int`

The type of integer numbers.

`type char`

The type of characters.

`type string`

The type of character strings.

`type float`

The type of floating-point numbers.

`type bool`

The type of booleans (truth values).

`type unit = ()`

The type of the unit value.

`type exn`

The type of exception values.

`type 'a array`

The type of arrays whose elements have type `'a`.

`type 'a list = [] | :: of 'a * 'a list`

The type of lists whose elements have type `'a`.

`type 'a option = None | Some of 'a`

The type of optional values.

`type ('a, 'b, 'c) format`

The type of format strings. `'a` is the type of the parameters of the format, `'c` is the result type for the `printf`-style function, and `'b` is the type of the first argument given to `%a` and `%t` printing functions (see module `Printf`).

## Exceptions

`val raise : exn -> 'a`

Raise the given exception value

`exception Match_failure of (string * int * int)`

Exception raised when none of the cases of a pattern-matching apply. The arguments are the location of the pattern-matching in the source code (file name, position of first character, position of last character).

`exception Assert_failure of (string * int * int)`

Exception raised when an assertion fails. The arguments are the location of the pattern-matching in the source code (file name, position of first character, position of last character).

`exception Invalid_argument of string`

Exception raised by library functions to signal that the given arguments do not make sense.

`exception Failure of string`

Exception raised by library functions to signal that they are undefined on the given arguments.

`exception Not_found`

Exception raised by search functions when the desired object could not be found.

`exception Out_of_memory`

Exception raised by the garbage collector when there is insufficient memory to complete the computation.

`exception Stack_overflow`

Exception raised by the bytecode interpreter when the evaluation stack reaches its maximal size. This often indicates infinite or excessively deep recursion in the user's program.

`exception Sys_error of string`

Exception raised by the input/output functions to report an operating system error.

`exception End_of_file`

Exception raised by input functions to signal that the end of file has been reached.

`exception Division_by_zero`

Exception raised by division and remainder operations when their second argument is null.

`exception Exit`

This exception is not raised by any library function. It is provided for use in your programs.

`exception Sys_blocked_io`

A special case of `Sys_error` raised when no I/O is possible on a non-blocking I/O channel.

`val invalid_arg: string -> 'a`

Raise exception `Invalid_argument` with the given string.

`val failwith: string -> 'a`

Raise exception `Failure` with the given string.

## Comparisons

```
val (=) : 'a -> 'a -> bool
```

`e1 = e2` tests for structural equality of `e1` and `e2`. Mutable structures (e.g. references and arrays) are equal if and only if their current contents are structurally equal, even if the two mutable objects are not the same physical object. Equality between functional values raises `Invalid_argument`. Equality between cyclic data structures may not terminate.

```
val (<>) : 'a -> 'a -> bool
```

Negation of `(=)`.

```
val (<) : 'a -> 'a -> bool
val (>) : 'a -> 'a -> bool
val (<=) : 'a -> 'a -> bool
val (>=) : 'a -> 'a -> bool
```

Structural ordering functions. These functions coincide with the usual orderings over integers, characters, strings and floating-point numbers, and extend them to a total ordering over all types. The ordering is compatible with `(=)`. As in the case of `(=)`, mutable structures are compared by contents. Comparison between functional values raises `Invalid_argument`. Comparison between cyclic structures may not terminate.

```
val compare: 'a -> 'a -> int
```

`compare x y` returns `0` if `x=y`, a negative integer if `x<y`, and a positive integer if `x>y`. The same restrictions as for `=` apply. `compare` can be used as the comparison function required by the `Set` and `Map` modules.

```
val min: 'a -> 'a -> 'a
```

Return the smaller of the two arguments.

```
val max: 'a -> 'a -> 'a
```

Return the greater of the two arguments.

```
val (==) : 'a -> 'a -> bool
```

`e1 == e2` tests for physical equality of `e1` and `e2`. On integers and characters, it is the same as structural equality. On mutable structures, `e1 == e2` is true if and only if physical modification of `e1` also affects `e2`. On non-mutable structures, the behavior of `(==)` is implementation-dependent, except that `e1 == e2` implies `e1 = e2`.

```
val (!=) : 'a -> 'a -> bool
```

Negation of `(==)`.

### Boolean operations

```
val not : bool -> bool
```

The boolean negation.

```
val (&&) : bool -> bool -> bool
val (&) : bool -> bool -> bool
```

The boolean "and". Evaluation is sequential, left-to-right: in `e1 && e2`, `e1` is evaluated first, and if it returns `false`, `e2` is not evaluated at all.

```
val (||) : bool -> bool -> bool
val (or) : bool -> bool -> bool
```

The boolean "or". Evaluation is sequential, left-to-right: in `e1 || e2`, `e1` is evaluated first, and if it returns `true`, `e2` is not evaluated at all.

### Integer arithmetic

Integers are 31 bits wide (or 63 bits on 64-bit processors). All operations are taken modulo $2^{31}$ (or $2^{63}$). They do not fail on overflow.

```
val (~-) : int -> int
```

Unary negation. You can also write `-e` instead of `~-e`.

```
val succ : int -> int
```

`succ x` is `x+1`.

```
val pred : int -> int
```

`pred x` is `x-1`.

```
val (+) : int -> int -> int
```

Integer addition.

```
val (-) : int -> int -> int
```

Integer subtraction.

```
val (*) : int -> int -> int
```

Integer multiplication.

```
val (/) : int -> int -> int
```

Integer division. Raise `Division_by_zero` if the second argument is 0.

`val (mod) : int -> int -> int`

Integer remainder. If `x >= 0` and `y > 0`, the result of `x mod y` satisfies the following properties: `0 <= x mod y < y` and `x = (x / y) * y + x mod y`. If `y = 0`, `x mod y` raises `Division_by_zero`. If `x < 0` or `y < 0`, the result of `x mod y` is not specified and depends on the platform.

`val abs : int -> int`

Return the absolute value of the argument.

```
val max_int: int
val min_int: int
```

The greatest and smallest representable integers.

**Bitwise operations**

`val (land) : int -> int -> int`

Bitwise logical and.

`val (lor) : int -> int -> int`

Bitwise logical or.

`val (lxor) : int -> int -> int`

Bitwise logical exclusive or.

`val lnot: int -> int`

Bitwise logical negation.

`val (lsl) : int -> int -> int`

`n lsl m` shifts `n` to the left by `m` bits.

`val (lsr) : int -> int -> int`

`n lsr m` shifts `n` to the right by `m` bits. This is a logical shift: zeroes are inserted regardless of the sign of `n`.

`val (asr) : int -> int -> int`

`n asr m` shifts `n` to the right by `m` bits. This is an arithmetic shift: the sign bit of `n` is replicated.

## Floating-point arithmetic

On most platforms, Caml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers. Floating-point operations do not fail on overflow or underflow, but return denormal numbers.

```
val (~-.) : float -> float
```

Unary negation. You can also write `-.e` instead of `~-.e`.

```
val (+.) : float -> float -> float
```

Floating-point addition

```
val (-.) : float -> float -> float
```

Floating-point subtraction

```
val (*.) : float -> float -> float
```

Floating-point multiplication

```
val (/.) : float -> float -> float
```

Floating-point division.

```
val (**) : float -> float -> float
```

Exponentiation

```
val sqrt : float -> float
```

Square root

```
val exp : float -> float
val log : float -> float
val log10 : float -> float
```

Exponential, natural logarithm, base 10 logarithm.

```
val cos : float -> float
val sin : float -> float
val tan : float -> float
val acos : float -> float
val asin : float -> float
val atan : float -> float
val atan2 : float -> float -> float
```

The usual trignonmetric functions

```
val cosh : float -> float
val sinh : float -> float
val tanh : float -> float
```

The usual hyperbolic trigonometric functions

```
val ceil : float -> float
val floor : float -> float
```

Round the given float to an integer value. `floor f` returns the greatest integer value less than or equal to `f`. `ceil f` returns the least integer value greater than or equal to `f`.

```
val abs_float : float -> float
```

Return the absolute value of the argument.

```
val mod_float : float -> float -> float
```

`mod_float a b` returns the remainder of `a` with respect to `b`. The returned value is `a -. n *. b`, where `n` is the quotient `a /. b` rounded towards zero to an integer.

```
val frexp : float -> float * int
```

`frexp f` returns the pair of the significant and the exponent of `f`. When `f` is zero, the significant `x` and the exponent `n` of `f` are equal to zero. When `f` is non-zero, they are defined by `f = x *. 2 ** n` and `0.5 <= x < 1.0`.

```
val ldexp : float -> int -> float
```

`ldexp x n` returns `x *. 2 ** n`.

```
val modf : float -> float * float
```

`modf f` returns the pair of the fractional and integral part of `f`.

```
val float : int -> float
val float_of_int : int -> float
```

Convert an integer to floating-point.

```
val truncate : float -> int
val int_of_float : float -> int
```

Truncate the given floating-point number to an integer. The result is unspecified if it falls outside the range of representable integers.

## String operations

More string operations are provided in module `String`.

```
val (^) : string -> string -> string
```

String concatenation.

### Character operations

More character operations are provided in module `Char`.

`val int_of_char : char -> int`

Return the ASCII code of the argument.

`val char_of_int : int -> char`

Return the character with the given ASCII code. Raise `Invalid_argument "char_of_int"` if the argument is outside the range 0–255.

### Unit operations

`val ignore : 'a -> unit`

Discard the value of its argument and return `()`. For instance, `ignore(f x)` discards the result of the side-effecting function `f`. It is equivalent to `f x; ()`, except that no warning is generated.

### String conversion functions

`val string_of_bool : bool -> string`

Return the string representation of a boolean.

`val bool_of_string : string -> bool`

Convert the given string to a boolean. Raise `Invalid_argument "bool_of_string"` if the string is not `"true"` or `"false"`.

`val string_of_int : int -> string`

Return the string representation of an integer, in decimal.

`val int_of_string : string -> int`

Convert the given string to an integer. The string is read in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b` respectively. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

`val string_of_float : float -> string`

Return the string representation of a floating-point number.

`val float_of_string : string -> float`

Convert the given string to a float. The result is unspecified if the given string is not a valid representation of a float.

## Pair operations

`val fst : 'a * 'b -> 'a`

Return the first component of a pair.

`val snd : 'a * 'b -> 'b`

Return the second component of a pair.

## List operations

More list operations are provided in module `List`.

`val (@) : 'a list -> 'a list -> 'a list`

List concatenation.

## Input/output

```
type in_channel
type out_channel
```

The types of input channels and output channels.

```
val stdin : in_channel
val stdout : out_channel
val stderr : out_channel
```

The standard input, standard output, and standard error output for the process.

### Output functions on standard output

`val print_char : char -> unit`

Print a character on standard output.

`val print_string : string -> unit`

Print a string on standard output.

`val print_int : int -> unit`

Print an integer, in decimal, on standard output.

`val print_float : float -> unit`

Print a floating-point number, in decimal, on standard output.

`val print_endline : string -> unit`

Print a string, followed by a newline character, on standard output.

`val print_newline : unit -> unit`

Print a newline character on standard output, and flush standard output. This can be used to simulate line buffering of standard output.

### Output functions on standard error

`val prerr_char : char -> unit`

Print a character on standard error.

`val prerr_string : string -> unit`

Print a string on standard error.

`val prerr_int : int -> unit`

Print an integer, in decimal, on standard error.

`val prerr_float : float -> unit`

Print a floating-point number, in decimal, on standard error.

`val prerr_endline : string -> unit`

Print a string, followed by a newline character on standard error and flush standard error.

`val prerr_newline : unit -> unit`

Print a newline character on standard error, and flush standard error.

### Input functions on standard input

`val read_line : unit -> string`

Flush standard output, then read characters from standard input until a newline character is encountered. Return the string of all characters read, without the newline character at the end.

`val read_int : unit -> int`

Flush standard output, then read one line from standard input and convert it to an integer. Raise `Failure "int_of_string"` if the line read is not a valid representation of an integer.

`val read_float : unit -> float`

Flush standard output, then read one line from standard input and convert it to a floating-point number. The result is unspecified if the line read is not a valid representation of a floating-point number.

### General output functions

```
type open_flag =
    Open_rdonly | Open_wronly | Open_append
  | Open_creat | Open_trunc | Open_excl
  | Open_binary | Open_text | Open_nonblock
```

Opening modes for `open_out_gen` and `open_in_gen`.

`Open_rdonly`: open for reading.

`Open_wronly`: open for writing.

`Open_append`: open for appending.

`Open_creat`: create the file if it does not exist.

`Open_trunc`: empty the file if it already exists.

`Open_excl`: fail if the file already exists.

`Open_binary`: open in binary mode (no conversion).

`Open_text`: open in text mode (may perform conversions).

`Open_nonblock`: open in non-blocking mode.

```
val open_out : string -> out_channel
```

Open the named file for writing, and return a new output channel on that file, positionned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists. Raise `Sys_error` if the file could not be opened.

```
val open_out_bin : string -> out_channel
```

Same as `open_out`, but the file is opened in binary mode, so that no translation takes place during writes. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_out`.

```
val open_out_gen : open_flag list -> int -> string -> out_channel
```

`open_out_gen mode rights filename` opens the file named `filename` for writing, as above. The extra argument `mode` specify the opening mode. The extra argument `rights` specifies the file permissions, in case the file must be created. `open_out` and `open_out_bin` are special cases of this function.

```
val flush : out_channel -> unit
```

Flush the buffer associated with the given output channel, performing all pending writes on that channel. Interactive programs must be careful about flushing standard output and standard error at the right time.

```
val output_char : out_channel -> char -> unit
```

Write the character on the given output channel.

`val output_string : out_channel -> string -> unit`

Write the string on the given output channel.

`val output : out_channel -> string -> int -> int -> unit`

`output chan buff ofs len` writes `len` characters from string `buff`, starting at offset `ofs`, to the output channel `chan`. Raise `Invalid_argument "output"` if `ofs` and `len` do not designate a valid substring of `buff`.

`val output_byte : out_channel -> int -> unit`

Write one 8-bit integer (as the single character with that code) on the given output channel. The given integer is taken modulo 256.

`val output_binary_int : out_channel -> int -> unit`

Write one integer in binary format on the given output channel. The only reliable way to read it back is through the `input_binary_int` function. The format is compatible across all machines for a given version of Objective Caml.

`val output_value : out_channel -> 'a -> unit`

Write the representation of a structured value of any type to a channel. Circularities and sharing inside the value are detected and preserved. The object can be read back, by the function `input_value`. See the description of module `Marshal` for more information. `output_value` is equivalent to `Marshal.to_channel` with an empty list of flags.

`val seek_out : out_channel -> int -> unit`

`seek_out chan pos` sets the current writing position to `pos` for channel `chan`. This works only for regular files. On files of other kinds (such as terminals, pipes and sockets), the behavior is unspecified.

`val pos_out : out_channel -> int`

Return the current writing position for the given channel.

`val out_channel_length : out_channel -> int`

Return the total length (number of characters) of the given channel. This works only for regular files. On files of other kinds, the result is meaningless.

`val close_out : out_channel -> unit`

Close the given channel, flushing all buffered write operations. The behavior is unspecified if any of the functions above is called on a closed channel.

`val set_binary_mode_out : out_channel -> bool -> unit`

`set_binary_mode_out oc true` sets the channel `oc` to binary mode: no translations take place during output. `set_binary_mode_out oc false` sets the channel `oc` to text mode: depending on the operating system, some translations may take place during output. For instance, under Windows, end-of-lines will be translated from `\n` to `\r\n`. This function has no effect under operating systems that do not distinguish between text mode and binary mode.

### General input functions

`val open_in : string -> in_channel`

Open the named file for reading, and return a new input channel on that file, positionned at the beginning of the file. Raise `Sys_error` if the file could not be opened.

`val open_in_bin : string -> in_channel`

Same as `open_in`, but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_in`.

`val open_in_gen : open_flag list -> int -> string -> in_channel`

`open_in_gen mode rights filename` opens the file named `filename` for reading, as above. The extra arguments `mode` and `rights` specify the opening mode and file permissions. `open_in` and `open_in_bin` are special cases of this function.

`val input_char : in_channel -> char`

Read one character from the given input channel. Raise `End_of_file` if there are no more characters to read.

`val input_line : in_channel -> string`

Read characters from the given input channel, until a newline character is encountered. Return the string of all characters read, without the newline character at the end. Raise `End_of_file` if the end of the file is reached at the beginning of line.

`val input : in_channel -> string -> int -> int -> int`

`input chan buff ofs len` attempts to read `len` characters from channel `chan`, storing them in string `buff`, starting at character number `ofs`. It returns the actual number of characters read, between 0 and `len` (inclusive). A return value of 0 means that the end of file was reached. A return value between 0 and `len` exclusive means that no more characters were available at that time; `input` must be called again to read the remaining characters, if desired. Exception `Invalid_argument "input"` is raised if `ofs` and `len` do not designate a valid substring of `buff`.

`val really_input : in_channel -> string -> int -> int -> unit`

`really_input chan buff ofs len` reads `len` characters from channel `chan`, storing them in string `buff`, starting at character number `ofs`. Raise `End_of_file` if the end of file is reached before `len` characters have been read. Raise `Invalid_argument "really_input"` if `ofs` and `len` do not designate a valid substring of `buff`.

`val input_byte : in_channel -> int`

Same as `input_char`, but return the 8-bit integer representing the character. Raise `End_of_file` if an end of file was reached.

`val input_binary_int : in_channel -> int`

Read an integer encoded in binary format from the given input channel. See `output_binary_int`. Raise `End_of_file` if an end of file was reached while reading the integer.

`val input_value : in_channel -> 'a`

Read the representation of a structured value, as produced by `output_value`, and return the corresponding value. This function is identical to `Marshal.from_channel`; see the description of module `Marshal` for more information, in particular concerning the lack of type safety.

`val seek_in : in_channel -> int -> unit`

`seek_in chan pos` sets the current reading position to `pos` for channel `chan`. This works only for regular files. On files of other kinds, the behavior is unspecified.

`val pos_in : in_channel -> int`

Return the current reading position for the given channel.

`val in_channel_length : in_channel -> int`

Return the total length (number of characters) of the given channel. This works only for regular files. On files of other kinds, the result is meaningless.

`val close_in : in_channel -> unit`

Close the given channel. Anything can happen if any of the functions above is called on a closed channel.

`val set_binary_mode_in : in_channel -> bool -> unit`

`set_binary_mode_in ic true` sets the channel `ic` to binary mode: no translations take place during input. `set_binary_mode_out ic false` sets the channel `ic` to text mode: depending on the operating system, some translations may take place during input. For instance, under Windows, end-of-lines will be translated from `\r\n` to `\n`. This function has no effect under operating systems that do not distinguish between text mode and binary mode.

## References

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

The type of references (mutable indirection cells) containing a value of type `'a`.

`val ref : 'a -> 'a ref`

Return a fresh reference containing the given value.

`val (!) : 'a ref -> 'a`

`!r` returns the current contents of reference `r`. Equivalent to `fun r -> r.contents`.

`val (:=) : 'a ref -> 'a -> unit`

`r := a` stores the value of `a` in reference `r`. Equivalent to `fun r v -> r.contents <- v`.

`val incr : int ref -> unit`

Increment the integer contained in the given reference. Equivalent to `fun r -> r := succ !r`.

`val decr : int ref -> unit`

Decrement the integer contained in the given reference. Equivalent to `fun r -> r := pred !r`.

## Program termination

`val exit : int -> 'a`

Flush all pending writes on `stdout` and `stderr`, and terminate the process, returning the given status code to the operating system (usually 0 to indicate no errors, and a small positive integer to indicate failure.) An implicit `exit 0` is performed each time a program terminates normally (but not if it terminates because of an uncaught exception).

`val at_exit: (unit -> unit) -> unit`

Register the given function to be called at program termination time. The functions registered with `at_exit` will be called when the program executes `exit`. They will not be called if the program terminates because of an uncaught exception. The functions are called in “last in, first out” order: the function most recently added with `at_exit` is called first.

# Chapter 17

# The standard library

This chapter describes the functions provided by the Caml Light standard library. The modules from the standard library are automatically linked with the user's object code files by the `ocamlc` command. Hence, these modules can be used in standalone programs without having to add any `.cmo` file on the command line for the linking phase. Similarly, in interactive use, these globals can be used in toplevel phrases without having to load any `.cmo` file in memory.

Unlike the `Pervasive` module from the core library, the modules from the standard library are not automatically "opened" when a compilation starts, or when the toplevel system is launched. Hence it is necessary to use qualified identifiers to refer to the functions provided by these modules, or to add `open` directives.

## Conventions

For easy reference, the modules are listed below in alphabetical order of module names. For each module, the declarations from its signature are printed one by one in typewriter font, followed by a short comment. All modules and the identifiers they export are indexed at the end of this report.

## Overview

Here is a short listing, by theme, of the standard library modules.

**Data structures:**

| | | |
|---|---|---|
| `Char` | p. 212 | character operations |
| `String` | p. 246 | string operations |
| `Array` | p. 208 | array operations |
| `List` | p. 231 | list operations |
| `Sort` | p. 243 | sorting and merging lists |
| `Hashtbl` | p. 226 | hash tables and hash functions |
| `Random` | p. 241 | pseudo-random number generator |
| `Set` | p. 241 | sets over ordered types |
| `Map` | p. 234 | association tables over ordered types |
| `Oo` | p. 237 | useful functions on objects |
| `Stack` | p. 244 | last-in first-out stacks |
| `Queue` | p. 240 | first-in first-out queues |
| `Buffer` | p. 210 | string buffers that grow on demand |
| `Lazy` | p. 229 | delayed evaluation |
| `Weak` | p. 251 | references that don't prevent objects from being garbage-collected |

**Input/output:**

| | | |
|---|---|---|
| `Format` | p. 214 | pretty printing |
| `Marshal` | p. 235 | marshaling of data structures |
| `Printf` | p. 239 | formatting printing functions |
| `Digest` | p. 212 | MD5 message digest |

**Parsing:**

| | | |
|---|---|---|
| `Genlex` | p. 226 | a generic lexer over streams |
| `Lexing` | p. 229 | the run-time library for lexers generated by `camllex` |
| `Parsing` | p. 238 | the run-time library for parsers generated by `camlyacc` |
| `Stream` | p. 244 | basic functions over streams |

**System interface:**

| | | |
|---|---|---|
| `Arg` | p. 206 | parsing of command line arguments |
| `Callback` | p. 212 | registering Caml functions to be called from C |
| `Filename` | p. 213 | operations on file names |
| `Gc` | p. 223 | memory management control and statistics |
| `Printexc` | p. 238 | a catch-all exception handler |
| `Sys` | p. 248 | system interface |

## 17.1 Module `Arg`: parsing of command line arguments

This module provides a general mechanism for extracting options and arguments from the command line to the program.

Syntax of command lines: A keyword is a character string starting with a `-`. An option is a keyword alone or followed by an argument. The types of keywords are: `Unit`, `Set`, `Clear`, `String`, `Int`, `Float`, and `Rest`. `Unit`, `Set` and `Clear` keywords take no argument. `String`, `Int`, and `Float` keywords take the following word on the command line as an argument. A `Rest` keyword takes the remaining of the command line as (string) arguments. Arguments not preceded by a keyword are called anonymous arguments.

Examples (`cmd` is assumed to be the command name):

| | |
|---|---|
| `cmd -flag` | (a unit option) |
| `cmd -int 1` | (an int option with argument `1`) |
| `cmd -string foobar` | (a string option with argument `"foobar"`) |
| `cmd -float 12.34` | (a float option with argument `12.34`) |
| `cmd a b c` | (three anonymous arguments: `"a"`, `"b"`, and `"c"`) |
| `cmd a b -- c d` | (two anonymous arguments and a rest option with two arguments) |

```
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 concrete type describing the behavior associated with a keyword.

```
val parse : (string * spec * string) list -> (string -> unit) -> string -> unit
```

`Arg.parse speclist anonfun usage_msg` parses the command line. `speclist` is a list of triples `(key, spec, doc)`. `key` is the option keyword, it must start with a `'-'` character. `spec` gives the option type and the function to call when this option is found on the command line. `doc` is a one-line description of this option. `anonfun` is called on anonymous arguments. The functions in `spec` and `anonfun` are called in the same order as their arguments appear on the command line.

If an error occurs, `Arg.parse` exits the program, after printing an error message as follows:

The reason for the error: unknown option, invalid or missing argument, etc.

`usage_msg`

The list of options, each followed by the corresponding `doc` string.

For the user to be able to specify anonymous arguments starting with a `-`, include for example `("-", String anonfun, doc)` in `speclist`.

By default, `parse` recognizes a unit option `-help`, which will display `usage_msg` and the list of options, and exit the program. You can override this behaviour by specifying your own `-help` option in `speclist`.

`exception Bad of string`

Functions in `spec` or `anonfun` can raise `Arg.Bad` with an error message to reject invalid arguments.

`val usage: (string * spec * string) list -> string -> unit`

`Arg.usage speclist usage_msg` prints an error message including the list of valid options. This is the same message that `Arg.parse` prints in case of error. `speclist` and `usage_msg` are the same as for `Arg.parse`.

`val current: int ref;;`

Position (in `Sys.argv`) of the argument being processed. You can change this value, e.g. to force `Arg.parse` to skip some arguments.

## 17.2 Module `Array`: array operations

`val length : 'a array -> int`

Return the length (number of elements) of the given array.

`val get: 'a array -> int -> 'a`

`Array.get a n` returns the element number `n` of array `a`. The first element has number 0. The last element has number `Array.length a - 1`. Raise `Invalid_argument "Array.get"` if `n` is outside the range 0 to `(Array.length a - 1)`. You can also write `a.(n)` instead of `Array.get a n`.

`val set: 'a array -> int -> 'a -> unit`

`Array.set a n x` modifies array `a` in place, replacing element number `n` with `x`. Raise `Invalid_argument "Array.set"` if `n` is outside the range 0 to `Array.length a - 1`. You can also write `a.(n) <- x` instead of `Array.set a n x`.

```
val make: int -> 'a -> 'a array
val create: int -> 'a -> 'a array
```

`Array.make n x` returns a fresh array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x` (in the sense of the `==` predicate). Consequently, if `x` is mutable, it is shared among all elements of the array, and modifying `x` through one of the array entries will modify all other entries at the same time.

`val init: int -> (int -> 'a) -> 'a array`

`Array.init n f` returns a fresh array of length `n`, with element number `i` initialized to the result of `f i`. In other terms, `Array.init n f` tabulates the results of `f` applied to the integers `0` to `n-1`.

```
val make_matrix: int -> int -> 'a -> 'a array array
val create_matrix: int -> int -> 'a -> 'a array array
```

`Array.make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element (`x,y`) of a matrix `m` is accessed with the notation `m.(x).(y)`.

```
val append: 'a array -> 'a array -> 'a array
```

`Array.append v1 v2` returns a fresh array containing the concatenation of the arrays `v1` and `v2`.

```
val concat: 'a array list -> 'a array
```

Same as `Array.append`, but catenates a list of arrays.

```
val sub: 'a array -> int -> int -> 'a array
```

`Array.sub a start len` returns a fresh array of length `len`, containing the elements number `start` to `start + len - 1` of array `a`. Raise `Invalid_argument "Array.sub"` if `start` and `len` do not designate a valid subarray of `a`; that is, if `start < 0`, or `len < 0`, or `start + len > Array.length a`.

```
val copy: 'a array -> 'a array
```

`Array.copy a` returns a copy of `a`, that is, a fresh array containing the same elements as `a`.

```
val fill: 'a array -> int -> int -> 'a -> unit
```

`Array.fill a ofs len x` modifies the array `a` in place, storing `x` in elements number `ofs` to `ofs + len - 1`. Raise `Invalid_argument "Array.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

```
val blit: 'a array -> int -> 'a array -> int -> int -> unit
```

`Array.blit v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap. Raise `Invalid_argument "Array.blit"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.

```
val to_list: 'a array -> 'a list
```

`Array.to_list a` returns the list of all the elements of `a`.

```
val of_list: 'a list -> 'a array
```

`Array.of_list l` returns a fresh array containing the elements of `l`.

`val iter: ('a -> unit) -> 'a array -> unit`

`Array.iter f a` applies function `f` in turn to all the elements of `a`. It is equivalent to `f a.(0); f a.(1); ...; f a.(Array.length a - 1); ()`.

`val map: ('a -> 'b) -> 'a array -> 'b array`

`Array.map f a` applies function `f` to all the elements of `a`, and builds an array with the results returned by `f`: `[| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |]`.

`val iteri: (int -> 'a -> unit) -> 'a array -> unit`
`val mapi: (int -> 'a -> 'b) -> 'a array -> 'b array`

Same as `Array.iter` and `Array.map` respectively, but the function is applied to the index of the element as first argument, and the element itself as second argument.

`val fold_left: ('a -> 'b -> 'a) -> 'a -> 'b array -> 'a`

`Array.fold_left f x a` computes `f (... (f (f x a.(0)) a.(1)) ...) a.(n-1)`, where `n` is the length of the array `a`.

`val fold_right: ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a`

`Array.fold_right f a x` computes `f a.(0) (f a.(1) ( ... (f a.(n-1) x) ...))`, where `n` is the length of the array `a`.

## 17.3 Module `Buffer`: extensible string buffers

This module implements string buffers that automatically expand as necessary. It provides accumulative concatenation of strings in quasi-linear time (instead of quadratic time when strings are concatenated pairwise).

`type t`

The abstract type of buffers.

`val create : int -> t`

`create n` returns a fresh buffer, initially empty. The `n` parameter is the initial size of the internal string that holds the buffer contents. That string is automatically reallocated when more than `n` characters are stored in the buffer, but shrinks back to `n` characters when `reset` is called. For best performance, `n` should be of the same order of magnitude as the number of characters that are expected to be stored in the buffer (for instance, 80 for a buffer that holds one output line). Nothing bad will happen if the buffer grows beyond that limit, however. In doubt, take `n = 16` for instance.

`val contents : t -> string`

Return a copy of the current contents of the buffer. The buffer itself is unchanged.

`val length : t -> int`

Return the number of characters currently contained in the buffer.

`val clear : t -> unit`

Empty the buffer.

`val reset : t -> unit`

Empty the buffer and deallocate the internal string holding the buffer contents, replacing it with the initial internal string of length `n` that was allocated by `create n`. For long-lived buffers that may have grown a lot, `reset` allows faster reclaimation of the space used by the buffer.

`val add_char : t -> char -> unit`

`add_char b c` appends the character `c` at the end of the buffer `b`.

`val add_string : t -> string -> unit`

`add_string b s` appends the string `s` at the end of the buffer `b`.

`val add_substring : t -> string -> int -> int -> unit`

`add_substring b s ofs len` takes `len` characters from offset `ofs` in string `s` and appends them at the end of the buffer `b`.

`val add_buffer : t -> t -> unit`

`add_buffer b1 b2` appends the current contents of buffer `b2` at the end of buffer `b1`. `b2` is not modified.

`val add_channel : t -> in_channel -> int -> unit`

`add_channel b ic n` reads exactly `n` character from the input channel `ic` and stores them at the end of buffer `b`. Raise `End_of_file` if the channel contains fewer than `n` characters.

`val output_buffer : out_channel -> t -> unit`

`output_buffer oc b` writes the current contents of buffer `b` on the output channel `oc`.

## 17.4 Module `Callback`: registering Caml values with the C runtime

This module allows Caml values to be registered with the C runtime under a symbolic name, so that C code can later call back registered Caml functions, or raise registered Caml exceptions.

```
val register: string -> 'a -> unit
```

`Callback.register n v` registers the value `v` under the name `n`. C code can later retrieve a handle to `v` by calling `caml_named_value(n)`.

```
val register_exception: string -> exn -> unit
```

`Callback.register_exception n exn` registers the exception contained in the exception value `exn` under the name `n`. C code can later retrieve a handle to the exception by calling `caml_named_value(n)`. The exception value thus obtained is suitable for passign as first argument to `raise_constant` or `raise_with_arg`.

## 17.5 Module `Char`: character operations

```
val code : char -> int
```

Return the ASCII code of the argument.

```
val chr: int -> char
```

Return the character with the given ASCII code. Raise `Invalid_argument "Char.chr"` if the argument is outside the range 0–255.

```
val escaped : char -> string
```

Return a string representing the given character, with special characters escaped following the lexical conventions of Objective Caml.

```
val lowercase: char -> char
val uppercase: char -> char
```

Convert the given character to its equivalent lowercase or uppercase character, respectively.

## 17.6 Module `Digest`: MD5 message digest

This module provides functions to compute 128-bit "digests" of arbitrary-length strings or files. The digests are of cryptographic quality: it is very hard, given a digest, to forge a string having that digest. The algorithm used is MD5.

`type t = string`

The type of digests: 16-character strings.

`val string: string -> t`

Return the digest of the given string.

`val substring: string -> int -> int -> t`

`Digest.substring s ofs len` returns the digest of the substring of `s` starting at character number `ofs` and containing `len` characters.

`val channel: in_channel -> int -> t`

`Digest.channel ic len` reads `len` characters from channel `ic` and returns their digest.

`val file: string -> t`

Return the digest of the file whose name is given.

`val output: out_channel -> t -> unit`

Write a digest on the given output channel.

`val input: in_channel -> t`

Read a digest from the given input channel.

## 17.7 Module `Filename`: operations on file names

`val current_dir_name : string`

The conventional name for the current directory (e.g. `.` in Unix).

`val concat : string -> string -> string`

`concat dir file` returns a file name that designates file `file` in directory `dir`.

`val is_relative : string -> bool`

Return `true` if the file name is relative to the current directory, `false` if it is absolute (i.e. in Unix, starts with `/`.

`val is_implicit : string -> bool`

Return `true` if the file name is relative and does not start with an explicit reference to the current directory (`./` or `../` in Unix), `false` if it starts with an explicit reference to the root directory or the current directory.

```
val check_suffix : string -> string -> bool
```

`check_suffix name suff` returns `true` if the filename `name` ends with the suffix `suff`.

```
val chop_suffix : string -> string -> string
```

`chop_suffix name suff` removes the suffix `suff` from the filename `name`. The behavior is undefined if `name` does not end with the suffix `suff`.

```
val chop_extension : string -> string
```

Return the given file name without its extension. The extension is the shortest suffix starting with a period, `.xyz` for instance. Raise `Invalid_argument` if the given name does not contain a period.

```
val basename : string -> string
val dirname : string -> string
```

Split a file name into directory name / base file name.
`concat (dirname name) (basename name)` returns a file name which is equivalent to `name`. Moreover, after setting the current directory to `dirname name` (with `Sys.chdir`), references to `basename name` (which is a relative file name) designate the same file as `name` before the call to `Sys.chdir`.

```
val temp_file: string -> string -> string
```

`temp_file prefix suffix` returns the name of a non-existent temporary file in the temporary directory. The base name of the temporary file is formed by concatenating `prefix`, then a suitably chosen integer number, then `suffix`. Under Unix, the temporary directory is `/tmp` by default; if set, the value of the environment variable `TMPDIR` is used instead. Under Windows, the name of the temporary directory is the value of the environment variable `TEMP`, or `C:\temp` by default. Under MacOS, the name of the temporary directory is given by the environment variable `TempFolder`; if not set, temporary files are created in the current directory.

## 17.8 Module `Format`: pretty printing

This module implements a 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.

You may consider this module as providing an extension to the `printf` facility to provide automatic line breaking. The addition of pretty-printing annotations to your regular `printf` formats gives you fancy indentation and line breaks. Pretty-printing annotations are described below in the documentation of the function `fprintf`.

You may also use the explicit box management and printing functions provided by this module. This style is more basic but more verbose than the `fprintf` concise formats.

For instance, the sequence
`open_box (); print_string "x ="; print_space (); print_int 1; close_box ()`
that prints `x = 1` within a pretty-printing box, can be abbreviated as
`printf "@[%s@ %i@]" "x =" 1`.

Rule of thumb for casual users of this library:

use simple boxes (as obtained by `open_box 0`);

use simple break hints (as obtained by `print_cut ()` that outputs a simple break hint, or by `print_space ()` that outputs a space indicating a break hint);

once a box is opened, display its material with basic printing functions (e. g. `print_int` and `print_string`);

when the material for a box has been printed, call `close_box ()` to close the box;

at the end of your routine, evaluate `print_newline ()` to close all remaining boxes and flush the pretty-printer.

The behaviour of pretty-printing commands is unspecified if there is no opened pretty-printing box. Each box opened via one of the `open_` functions below must be closed using `close_box` for proper formatting. Otherwise, some of the material printed in the boxes may not be output, or may be formatted incorrectly.

In case of interactive use, the system closes all opened boxes and flushes all pending text (as with the `print_newline` function) after each phrase. Each phrase is therefore executed in the initial state of the pretty-printer.

## Boxes

`val open_box : int -> unit;;`

`open_box d` opens a new pretty-printing box with offset `d`. This box is the general purpose pretty-printing box. Material in this box is displayed “horizontal or vertical”: break hints inside the box may lead to a new line, if there is no more room on the line to print the remainder of the box, or if a new line may lead to a new indentation (demonstrating the indentation of the box). When a new line is printed in the box, `d` is added to the current indentation.

`val close_box : unit -> unit;;`

Close the most recently opened pretty-printing box.

## Formatting functions

```
val print_string : string -> unit;;
```

`print_string str` prints `str` in the current box.

```
val print_as : int -> string -> unit;;
```

`print_as len str` prints `str` in the current box. The pretty-printer formats `str` as if it were of length `len`.

```
val print_int : int -> unit;;
```

Print an integer in the current box.

```
val print_float : float -> unit;;
```

Print a floating point number in the current box.

```
val print_char : char -> unit;;
```

Print a character in the current box.

```
val print_bool : bool -> unit;;
```

Print an boolean in the current box.

## Break hints

```
val print_space : unit -> unit;;
```

`print_space ()` is used to separate items (typically to print a space between two words). It indicates that the line may be split at this point. It either prints one space or splits the line. It is equivalent to `print_break 1 0`.

```
val print_cut : unit -> unit;;
```

`print_cut ()` is used to mark a good break position. It indicates that the line may be split at this point. It either prints nothing or splits the line. This allows line splitting at the current point, without printing spaces or adding indentation. It is equivalent to `print_break 0 0`.

```
val print_break : int -> int -> unit;;
```

Insert a break hint in a pretty-printing box. `print_break nspaces offset` indicates that the line may be split (a newline character is printed) at this point, if the contents of the current box does not fit on one line. If the line is split at that point, `offset` is added to the current indentation. If the line is not split, `nspaces` spaces are printed.

```
val print_flush : unit -> unit;;
```

Flush the pretty printer: all opened boxes are closed, and all pending text is displayed.

`val print_newline : unit -> unit;;`

Equivalent to `print_flush` followed by a new line.

`val force_newline : unit -> unit;;`

Force a newline in the current box. Not the normal way of pretty-printing, you should prefer break hints.

`val print_if_newline : unit -> unit;;`

Execute the next formatting command if the preceding line has just been split. Otherwise, ignore the next formatting command.

### Margin

`val set_margin : int -> unit;;`

`set_margin d` sets the value of the right margin to `d` (in characters): this value is used to detect line overflows that leads to split lines. Nothing happens if `d` is smaller than 2 or bigger than 999999999.

`val get_margin : unit -> int;;`

Return the position of the right margin.

### Maximum indentation limit

`val set_max_indent : int -> unit;;`

`set_max_indent d` sets the value of the maximum indentation limit to `d` (in characters): once this limit is reached, boxes are rejected to the left, if they do not fit on the current line. Nothing happens if `d` is smaller than 2 or bigger than 999999999.

`val get_max_indent : unit -> int;;`

Return the value of the maximum indentation limit (in characters).

### Formatting depth: maximum number of boxes allowed before ellipsis

`val set_max_boxes : int -> unit;;`

`set_max_boxes max` sets the maximum number of boxes simultaneously opened. Material inside boxes nested deeper is printed as an ellipsis (more precisely as the text returned by `get_ellipsis_text ()`). Nothing happens if `max` is not greater than 1.

`val get_max_boxes : unit -> int;;`

Return the maximum number of boxes allowed before ellipsis.

`val over_max_boxes : unit -> bool;;`

Test the maximum number of boxes allowed have already been opened.

## Advanced formatting

`val open_hbox : unit -> unit;;`

`open_hbox ()` opens a new pretty-printing box. This box is “horizontal”: the line is not split in this box (new lines may still occur inside boxes nested deeper).

`val open_vbox : int -> unit;;`

`open_vbox d` opens a new pretty-printing box with offset `d`. This box is “vertical”: every break hint inside this box leads to a new line. When a new line is printed in the box, `d` is added to the current indentation.

`val open_hvbox : int -> unit;;`

`open_hvbox d` opens a new pretty-printing box with offset `d`. This box is “horizontal-vertical”: it behaves as an “horizontal” box if it fits on a single line, otherwise it behaves as a “vertical” box. When a new line is printed in the box, `d` is added to the current indentation.

`val open_hovbox : int -> unit;;`

`open_hovbox d` opens a new pretty-printing box with offset `d`. This box is “horizontal or vertical”: break hints inside this box may lead to a new line, if there is no more room on the line to print the remainder of the box. When a new line is printed in the box, `d` is added to the current indentation.

## Tabulations

`val open_tbox : unit -> unit;;`

Open a tabulation box.

`val close_tbox : unit -> unit;;`

Close the most recently opened tabulation box.

`val print_tbreak : int -> int -> unit;;`

Break hint in a tabulation box. `print_tbreak spaces offset` moves the insertion point to the next tabulation (`spaces` being added to this position). Nothing occurs if insertion point is already on a tabulation mark. If there is no next tabulation on the line, then a newline is printed and the insertion point moves to the first tabulation of the box. If a new line is printed, `offset` is added to the current indentation.

`val set_tab : unit -> unit;;`

Set a tabulation mark at the current insertion point.

`val print_tab : unit -> unit;;`

`print_tab ()` is equivalent to `print_tbreak (0,0)`.

### Ellipsis

```
val set_ellipsis_text : string -> unit;;
```

Set the text of the ellipsis printed when too many boxes are opened (a single dot, `.`, by default).

```
val get_ellipsis_text : unit -> string;;
```

Return the text of the ellipsis.

### Redirecting formatter output

```
val set_formatter_out_channel : out_channel -> unit;;
```

Redirect the pretty-printer output to the given channel.

```
val set_formatter_output_functions :
      (string -> int -> int -> unit) -> (unit -> unit) -> unit;;
```

`set_formatter_output_functions out flush` redirects the pretty-printer output to the functions `out` and `flush`. The `out` function performs the pretty-printer output. It is called with a string `s`, a start position `p`, and a number of characters `n`; it is supposed to output characters `p` to `p+n-1` of `s`. The `flush` function is called whenever the pretty-printer is flushed using `print_flush` or `print_newline`.

```
val get_formatter_output_functions :
        unit -> (string -> int -> int -> unit) * (unit -> unit);;
```

Return the current output functions of the pretty-printer.

### Changing the meaning of indentation and line breaking

```
val set_all_formatter_output_functions :
      (string -> int -> int -> unit) -> (unit -> unit) ->
      (unit -> unit) -> (int -> unit) -> unit;;
```

`set_all_formatter_output_functions out flush outnewline outspace` redirects the pretty-printer output to the functions `out` and `flush` as described in `set_formatter_output_functions`. In addition, the pretty-printer function that outputs a newline is set to the function `outnewline` and the function that outputs indentation spaces is set to the function `outspace`. This way, you can change the meaning of indentation (which can be something else than just printing a space character) and the meaning of new lines opening (which can be connected to any other action needed by the application at hand). The two functions `outspace` and `outnewline` are normally connected to `out` and `flush`: respective default values for `outspace` and `outnewline` are `out (String.make n ' ') 0 n` and `out "\n" 0 1`.

```
val get_all_formatter_output_functions : unit ->
        (string -> int -> int -> unit) * (unit -> unit) *
        (unit -> unit) * (int -> unit);;
```

Return the current output functions of the pretty-printer, including line breaking and indentation functions.

## Multiple formatted output

```
type formatter;;
```

Abstract data type corresponding to a pretty-printer and all its machinery. Defining new pretty-printers permits the output of material in parallel on several channels. Parameters of the pretty-printer are local to the pretty-printer: margin, maximum indentation limit, maximum number of boxes simultaneously opened, ellipsis, and so on, are specific to each pretty-printer and may be fixed independently. Given an output channel `oc`, a new formatter writing to that channel is obtained by calling `formatter_of_out_channel oc`. Alternatively the `make_formatter` function allocates a new formatter with explicit output and flushing functions (convenient to output material to strings for instance).

```
val formatter_of_out_channel : out_channel -> formatter;;
```

`formatter_of_out_channel oc` returns a new formatter that writes to the corresponding channel `oc`.

```
val std_formatter : formatter;;
```

The standard formatter used by the formatting functions above. It is defined as `formatter_of_out_channel stdout`.

```
val err_formatter : formatter;;
```

A formatter to use with formatting functions below for output to standard error. It is defined as `formatter_of_out_channel stderr`.

```
val formatter_of_buffer : Buffer.t -> formatter;;
```

`formatter_of_buffer b` returns a new formatter writing to buffer `b`. As usual, the formatter has to be flushed at the end of pretty printing, using `pp_print_flush` or `pp_print_newline`, to display all the pending material. In this case the buffer is also flushed using `Buffer.flush`.

```
val stdbuf : Buffer.t;;
```

The string buffer in which `str_formatter` writes.

```
val str_formatter : formatter;;
```

A formatter to use with formatting functions below for output to the `stdbuf` string buffer.

```
val flush_str_formatter : unit -> string;;
```

Returns the material printed with `str_formatter`, flushes the formatter and reset the corresponding buffer. `str_formatter` is defined as `formatter_of_buffer stdbuf`.

```
val make_formatter :
    (string -> int -> int -> unit) -> (unit -> unit) -> formatter;;
```

`make_formatter out flush` returns a new formatter that writes according to the output function `out`, and the flushing function `flush`. Hence, a formatter to out channel `oc` is returned by `make_formatter (output oc) (fun () -> flush oc)`.

```
val pp_open_hbox : formatter -> unit -> unit;;
val pp_open_vbox : formatter -> int -> unit;;
val pp_open_hvbox : formatter -> int -> unit;;
val pp_open_hovbox : formatter -> int -> unit;;
val pp_open_box : formatter -> int -> unit;;
val pp_close_box : formatter -> unit -> unit;;
val pp_print_string : formatter -> string -> unit;;
val pp_print_as : formatter -> int -> string -> unit;;
val pp_print_int : formatter -> int -> unit;;
val pp_print_float : formatter -> float -> unit;;
val pp_print_char : formatter -> char -> unit;;
val pp_print_bool : formatter -> bool -> unit;;
val pp_print_break : formatter -> int -> int -> unit;;
val pp_print_cut : formatter -> unit -> unit;;
val pp_print_space : formatter -> unit -> unit;;
val pp_force_newline : formatter -> unit -> unit;;
val pp_print_flush : formatter -> unit -> unit;;
val pp_print_newline : formatter -> unit -> unit;;
val pp_print_if_newline : formatter -> unit -> unit;;
val pp_open_tbox : formatter -> unit -> unit;;
val pp_close_tbox : formatter -> unit -> unit;;
val pp_print_tbreak : formatter -> int -> int -> unit;;
val pp_set_tab : formatter -> unit -> unit;;
val pp_print_tab : formatter -> unit -> unit;;
val pp_set_margin : formatter -> int -> unit;;
val pp_get_margin : formatter -> unit -> int;;
val pp_set_max_indent : formatter -> int -> unit;;
val pp_get_max_indent : formatter -> unit -> int;;
val pp_set_max_boxes : formatter -> int -> unit;;
val pp_get_max_boxes : formatter -> unit -> int;;
val pp_over_max_boxes : formatter -> unit -> bool;;
val pp_set_ellipsis_text : formatter -> string -> unit;;
val pp_get_ellipsis_text : formatter -> unit -> string;;
val pp_set_formatter_out_channel : formatter -> out_channel -> unit;;
val pp_set_formatter_output_functions : formatter ->
```

```
        (string -> int -> int -> unit) -> (unit -> unit) -> unit;;
val pp_get_formatter_output_functions :
        formatter -> unit -> (string -> int -> int -> unit) * (unit -> unit);;
val pp_set_all_formatter_output_functions : formatter ->
      (string -> int -> int -> unit) -> (unit -> unit) ->
      (unit -> unit) -> (int -> unit) -> unit;;
val pp_get_all_formatter_output_functions : formatter -> unit ->
      (string -> int -> int -> unit) * (unit -> unit) *
      (unit -> unit) * (int -> unit);;
```

The basic functions to use with formatters. These functions are the basic ones: usual functions operating on the standard formatter are defined via partial evaluation of these primitives. For instance, `print_string` is equal to `pp_print_string std_formatter`.

```
val fprintf : formatter -> ('a, formatter, unit) format -> 'a;;
```

`fprintf ff format arg1 ... argN` formats the arguments `arg1` to `argN` according to the format string `format`, and outputs the resulting string on the formatter `ff`. The format is a character string which contains three types of objects: plain characters and conversion specifications as specified in the `printf` module, and pretty-printing indications. The pretty-printing indication characters are introduced by a `@` character, and their meanings are:

`@[`: open a pretty-printing box. The type and offset of the box may be optionally specified with the following syntax: the `<` character, followed by an optional box type indication, then an optional integer offset, and the closing `>` character. Box type is one of `h`, `v`, `hv`, `b`, or `hov`, which stand respectively for an horizontal box, a vertical box, an "horizontal-vertical" box, or an "horizontal or vertical" box (`b` standing for an "horizontal or vertical" box demonstrating indentation and `hov` standing for a regular"horizontal or vertical" box). For instance, `@[<hov 2>` opens an "horizontal or vertical" box with indentation 2.

`@]`: close the most recently opened pretty-printing box.

`@,`: output a good break as with `print_cut ()`.

`@ `: output a space, as with `print_space ()`.

`@\n`: force a newline, as with `force_newline ()`.

`@;`: output a good break as with `print_break`. The `nspaces` and `offset` parameters of the break may be optionally specified with the following syntax: the `<` character, followed by an integer `nspaces` value, then an integer offset, and a closing `>` character.

`@?`: flush the pretty printer as with `print_flush ()`.

`@.`: flush the pretty printer and output a new line, as with `print_newline ()`.

`@<n>`: print the following item as if it were of length `n`. Hence, `printf "@<0>%s" arg` is equivalent to `print_as 0 arg`. If `@<n>` is not followed by a conversion specification, then the following character of the format is printed as if it were of length `n`.

`@@`: print a plain `@` character.

Example: `printf "@[%s@ %d@]" "x =" 1` is equivalent to
`open_box (); print_string "x ="; print_space (); print_int 1; close_box ()`. It prints `x = 1` within a pretty-printing box.

```
val bprintf: Buffer.t -> ('a, Buffer.t, unit) format -> 'a;;
```

Same as `fprintf`, but instead of printing on a formatter, writes into the buffer argument.

```
val printf : ('a, formatter, unit) format -> 'a;;
```

Same as `fprintf`, but output on `std_formatter`.

```
val eprintf: ('a, formatter, unit) format -> 'a;;
```

Same as `fprintf`, but output on `err_formatter`.

```
val sprintf: ('a, unit, string) format -> 'a;;
```

Same as `printf`, but instead of printing on a formatter, return a string containing the result of formatting the arguments.

## 17.9 Module `Gc`: memory management control and statistics

```
type stat = {
  minor_words : int;
  promoted_words : int;
  major_words : int;
  minor_collections : int;
  major_collections : int;
  heap_words : int;
  heap_chunks : int;
  live_words : int;
  live_blocks : int;
  free_words : int;
  free_blocks : int;
  largest_free : int;
  fragments : int;
  compactions : int
}
```

The memory management counters are returned in a `stat` record. The fields of this record are:

`minor_words` Number of words allocated in the minor heap since the program was started.

`promoted_words` Number of words allocated in the minor heap that survived a minor collection and were moved to the major heap since the program was started.

`major_words` Number of words allocated in the major heap, including the promoted words, since the program was started.

`minor_collections` Number of minor collections since the program was started.

`major_collections` Number of major collection cycles, not counting the current cycle, since the program was started.

`heap_words` Total number of words in the major heap.

`heap_chunks` Number of times the major heap size was increased since the program was started.

`live_words` Number of words of live data in the major heap, including the header words.

`live_blocks` Number of live objects in the major heap.

`free_words` Number of words in the free list.

`free_blocks` Number of objects in the free list.

`largest_free` Size (in words) of the largest object in the free list.

`fragments` Number of wasted words due to fragmentation. These are 1-words free blocks placed between two live objects. They cannot be inserted in the free list, thus they are not available for allocation.

`compactions` Number of heap compactions since the program was started.

The total amount of memory allocated by the program since it was started is (in words) `minor_words + major_words - promoted_words`. Multiply by the word size (4 on a 32-bit machine, 8 on a 64-bit machine) to get the number of bytes.

```
type control = {
  mutable minor_heap_size : int;
  mutable major_heap_increment : int;
  mutable space_overhead : int;
  mutable verbose : int;
  mutable max_overhead : int;
  mutable stack_limit : int
}
```

The GC parameters are given as a `control` record. The fields are:

`minor_heap_size` The size (in words) of the minor heap. Changing this parameter will trigger a minor collection. Default: 32k.

`major_heap_increment` The minimum number of words to add to the major heap when increasing it. Default: 62k.

`space_overhead` The major GC speed is computed from this parameter. This is the memory that will be "wasted" because the GC does not immediatly collect unreachable objects. It is expressed as a percentage of the memory used for live data. The GC will work more (use more CPU time and collect objects more eagerly) if `space_overhead` is smaller. The computation of the GC speed assumes that the amount of live data is constant. Default: 42.

`max_overhead` Heap compaction is triggered when the estimated amount of free memory is more than `max_overhead` percent of the amount of live data. If `max_overhead` is set to 0, heap compaction is triggered at the end of each major GC cycle (this setting is intended for testing purposes only). If `max_overhead >= 1000000`, compaction is never triggered. Default: 1000000.

`verbose` This value controls the GC messages on standard error output. It is a sum of some of the following flags, to print messages on the corresponding events:

`1` Start of major GC cycle.

`2` Minor collection and major GC slice.

`4` Growing and shrinking of the heap.

`8` Resizing of stacks and memory manager tables.

`16` Heap compaction.

`32` Change of GC parameters.

`64` Computation of major GC slice size. Default: 0.

`stack_limit` The maximum size of the stack (in words). This is only relevant to the byte-code runtime, as the native code runtime uses the operating system's stack. Default: 256k.

`val stat : unit -> stat`

Return the current values of the memory management counters in a `stat` record.

`val print_stat : out_channel -> unit`

Print the current values of the memory management counters (in human-readable form) into the channel argument.

`val get : unit -> control`

Return the current values of the GC parameters in a `control` record.

`val set : control -> unit`

`set r` changes the GC parameters according to the `control` record `r`. The normal usage is:

```
let r = Gc.get () in    (* Get the current parameters. *)
  r.verbose <- 13;      (* Change some of them. *)
  Gc.set r              (* Set the new values. *)
```

`val minor : unit -> unit`

Trigger a minor collection.

`val major : unit -> unit`

Finish the current major collection cycle.

`val full_major : unit -> unit`

Finish the current major collection cycle and perform a complete new cycle. This will collect all currently unreachable objects.

`val compact : unit -> unit = "gc_compaction";;`

Perform a full major collection and compact the heap. Note that heap compaction is a lengthy operation.

## 17.10 Module `Genlex`: a generic lexical analyzer

This module implements a simple "standard" lexical analyzer, presented as a function from character streams to token streams. It implements roughly the lexical conventions of Caml, but is parameterized by the set of keywords of your language.

```
type token =
    Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char
```

The type of tokens. The lexical classes are: `Int` and `Float` for integer and floating-point numbers; `String` for string literals, enclosed in double quotes; `Char` for character literals, enclosed in single quotes; `Ident` for identifiers (either sequences of letters, digits, underscores and quotes, or sequences of "operator characters" such as `+`, `*`, etc); and `Kwd` for keywords (either identifiers or single "special characters" such as `(`, `}`, etc).

```
val make_lexer: string list -> (char Stream.t -> token Stream.t)
```

Construct the lexer function. The first argument is the list of keywords. An identifier `s` is returned as `Kwd s` if `s` belongs to this list, and as `Ident s` otherwise. A special character `s` is returned as `Kwd s` if `s` belongs to this list, and cause a lexical error (exception `Parse_error`) otherwise. Blanks and newlines are skipped. Comments delimited by `(*` and `*)` are skipped as well, and can be nested.

Example: a lexer suitable for a desk calculator is obtained by

```
let lexer = make_lexer ["+";"-";"*";"/";"let";"="; "("; ")"]
```

The associated parser would be a function from `token stream` to, for instance, `int`, and would have rules such as:

```
let parse_expr = parser
       [< 'Int n >] -> n
     | [< 'Kwd "("; n = parse_expr; 'Kwd ")" >] -> n
     | [< n1 = parse_expr; n2 = parse_remainder n1 >] -> n2
and parse_remainder n1 = parser
       [< 'Kwd "+"; n2 = parse_expr >] -> n1+n2
     | ...
```

## 17.11 Module `Hashtbl`: hash tables and hash functions

Hash tables are hashed association tables, with in-place modification.

## Generic interface

`type ('a, 'b) t`

The type of hash tables from type `'a` to type `'b`.

`val create : int -> ('a,'b) t`

`Hashtbl.create n` creates a new, empty hash table, with initial size `n`. The table grows as needed, so `n` is just an initial guess. Better results are said to be achieved when `n` is a prime number. Raise `Invalid_argument` if `n` is less than 1.

`val clear : ('a, 'b) t -> unit`

Empty a hash table.

`val add : ('a, 'b) t -> 'a -> 'b -> unit`

`Hashtbl.add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashtbl.remove tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists.)

`val find : ('a, 'b) t -> 'a -> 'b`

`Hashtbl.find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

`val find_all : ('a, 'b) t -> 'a -> 'b list`

`Hashtbl.find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

`val mem :  ('a, 'b) t -> 'a -> bool`

`Hashtbl.mem tbl x` checks if `x` is bound in `tbl`.

`val remove : ('a, 'b) t -> 'a -> unit`

`Hashtbl.remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

`val iter : ('a -> 'b -> unit) -> ('a, 'b) t -> unit`

`Hashtbl.iter f tbl` applies `f` to all bindings in table `tbl`. `f` receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to `f` is unspecified. Each binding is presented exactly once to `f`.

### Functorial interface

```
module type HashedType =
  sig
    type t
    val equal: t -> t -> bool
    val hash: t -> int
  end
```

The input signature of the functor `Hashtbl.Make`. `t` is the type of keys. `equal` is the equality predicate used to compare keys. `hash` is a hashing function on keys, returning a non-negative integer. It must be such that if two keys are equal according to `equal`, then they must have identical hash values as computed by `hash`. Examples: suitable (`equal`, `hash`) pairs for arbitrary key types include (`(=)`, `Hashtbl.hash`) for comparing objects by structure, and (`(==)`, `Hashtbl.hash`) for comparing objects by addresses (e.g. for mutable or cyclic keys).

```
module type S =
  sig
    type key
    type 'a 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
module Make(H: HashedType): (S with type key = H.t)
```

The functor `Hashtbl.Make` returns a structure containing a type `key` of keys and a type `'a t` of hash tables associating data of type `'a` to keys of type `key`. The operations perform similarly to those of the generic interface, but use the hashing and equality functions specified in the functor argument `H` instead of generic equality and hashing.

### The polymorphic hash primitive

```
val hash : 'a -> int
```

`Hashtbl.hash x` associates a positive integer to any value of any type. It is guaranteed that if `x = y`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

```
val hash_param : int -> int -> 'a -> int
```

Hashtbl.hash_param n m x computes a hash value for x, with the same properties as for hash. The two extra parameters n and m give more precise control over hashing. Hashing performs a depth-first, right-to-left traversal of the structure x, stopping after n meaningful nodes were encountered, or m nodes, meaningful or not, were encountered. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of m and n means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters m and n govern the tradeoff between accuracy and speed.

## 17.12 Module Lazy: deferred computations.

```
type 'a status =
  | Delayed of (unit -> 'a)
  | Value of 'a
  | Exception of exn
;;
type 'a t = 'a status ref;;
```

A value of type 'a Lazy.t is a deferred computation (also called a suspension) that computes a result of type 'a. The expression lazy (expr) returns a suspension that computes expr.

```
val force: 'a t -> 'a;;
```

Lazy.force x computes the suspension x and returns its result. If the suspension was already computed, Lazy.force x returns the same value again. If it raised an exception, the same exception is raised again.

## 17.13 Module Lexing: the run-time library for lexers generated by ocamllex

### Lexer buffers

```
type lexbuf =
  { refill_buff : lexbuf -> unit;
    mutable lex_buffer : string;
    mutable lex_buffer_len : int;
    mutable lex_abs_pos : int;
    mutable lex_start_pos : int;
    mutable lex_curr_pos : int;
    mutable lex_last_pos : int;
    mutable lex_last_action : int;
    mutable lex_eof_reached : bool }
```

The type of lexer buffers. A lexer buffer is the argument passed to the scanning functions defined by the generated scanners. The lexer buffer holds the current state of the scanner, plus a function to refill the buffer from the input.

```
val from_channel : in_channel -> lexbuf
```

Create a lexer buffer on the given input channel. `Lexing.from_channel inchan` returns a lexer buffer which reads from the input channel `inchan`, at the current reading position.

```
val from_string : string -> lexbuf
```

Create a lexer buffer which reads from the given string. Reading starts from the first character in the string. An end-of-input condition is generated when the end of the string is reached.

```
val from_function : (string -> int -> int) -> lexbuf
```

Create a lexer buffer with the given function as its reading method. When the scanner needs more characters, it will call the given function, giving it a character string `s` and a character count `n`. The function should put `n` characters or less in `s`, starting at character number 0, and return the number of characters provided. A return value of 0 means end of input.

### Functions for lexer semantic actions

The following functions can be called from the semantic actions of lexer definitions (the ML code enclosed in braces that computes the value returned by lexing functions). They give access to the character string matched by the regular expression associated with the semantic action. These functions must be applied to the argument `lexbuf`, which, in the code generated by `ocamllex`, is bound to the lexer buffer passed to the parsing function.

```
val lexeme : lexbuf -> string
```

`Lexing.lexeme lexbuf` returns the string matched by the regular expression.

```
val lexeme_char : lexbuf -> int -> char
```

`Lexing.lexeme_char lexbuf i` returns character number `i` in the matched string.

```
val lexeme_start : lexbuf -> int
```

`Lexing.lexeme_start lexbuf` returns the position in the input stream of the first character of the matched string. The first character of the stream has position 0.

```
val lexeme_end : lexbuf -> int
```

`Lexing.lexeme_end lexbuf` returns the position in the input stream of the character following the last character of the matched string. The first character of the stream has position 0.

## 17.14 Module `List`: list operations

`val length : 'a list -> int`

Return the length (number of elements) of the given list.

`val hd : 'a list -> 'a`

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

`val tl : 'a list -> 'a list`

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

`val nth : 'a list -> int -> 'a`

Return the n-th element of the given list. The first element (head of the list) is at position 0. Raise `Failure "nth"` if the list is too short.

`val rev : 'a list -> 'a list`

List reversal.

`val append : 'a list -> 'a list -> 'a list`

Catenate two lists. Same function as the infix operator `@`.

`val rev_append : 'a list -> 'a list -> 'a list`

`List.rev_append l1 l2` reverses `l1` and catenates it to `l2`. This is equivalent to `List.rev l1 @ l2`, but is more efficient as no intermediate lists are built.

```
val concat  : 'a list list -> 'a list
val flatten : 'a list list -> 'a list
```

Catenate (flatten) a list of lists.

### Iterators

`val iter : ('a -> unit) -> 'a list -> unit`

`List.iter f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`. It is equivalent to `begin f a1; f a2; ...; f an; () end`.

`val map : ('a -> 'b) -> 'a list -> 'b list`

`List.map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

`val fold_left : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

`List.fold_left f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

`val fold_right : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

`List.fold_right f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`.

### Iterators on two lists

`val iter2 : ('a -> 'b -> unit) -> 'a list -> 'b list -> unit`

`List.iter2 f [a1; ...; an] [b1; ...; bn]` calls in turn `f a1 b1; ...; f an bn`. Raise `Invalid_argument` if the two lists have different lengths.

`val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`List.map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`. Raise `Invalid_argument` if the two lists have different lengths.

`val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a`

`List.fold_left2 f a [b1; ...; bn] [c1; ...; cn]` is `f (... (f (f a b1 c1) b2 c2) ...) bn cn`. Raise `Invalid_argument` if the two lists have different lengths.

`val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c`

`List.fold_right2 f [a1; ...; an] [b1; ...; bn] c` is `f a1 b1 (f a2 b2 (... (f an bn c) ...))`. Raise `Invalid_argument` if the two lists have different lengths.

### List scanning

`val for_all : ('a -> bool) -> 'a list -> bool`

`for_all p [a1; ...; an]` checks if all elements of the list satisfy the predicate `p`. That is, it returns `(p a1) && (p a2) && ... && (p an)`.

`val exists : ('a -> bool) -> 'a list -> bool`

`exists p [a1; ...; an]` checks if at least one element of the list satisfies the predicate `p`. That is, it returns `(p a1) || (p a2) || ... || (p an)`.

`val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`
`val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool`

Same as `for_all` and `exists`, but for a two-argument predicate. Raise `Invalid_argument` if the two lists have different lengths.

`val mem : 'a -> 'a list -> bool`

`mem a l` is true if and only if `a` is equal to an element of `l`.

`val memq : 'a -> 'a list -> bool`

Same as `mem`, but uses physical equality instead of structural equality to compare list elements.

### List searching

```
val find : ('a -> bool) -> 'a list -> 'a
```

> `find p l` returns the first element of the list `l` that satisfies the predicate `p`. Raise `Not_found` if there is no value that satisfies `p` in the list `l`.

```
val filter : ('a -> bool) -> 'a list -> 'a list
val find_all : ('a -> bool) -> 'a list -> 'a list
```

> `filter p l` returns all the elements of the list `l` that satisfies the predicate `p`. `find_all` is another name for `filter`.

```
val partition : ('a -> bool) -> 'a list -> 'a list * 'a list
```

> `partition p l` returns a pair of lists `(l1, l2)`, where `l1` is the list of all the elements of `l` that satisfy the predicate `p`, and `l2` is the list of all the elements of `l` that do not satisfy `p`.

### Association lists

```
val assoc : 'a -> ('a * 'b) list -> 'b
```

> `assoc a l` returns the value associated with key `a` in the list of pairs `l`. That is, `assoc a [ ...; (a,b); ...] = b` if `(a,b)` is the leftmost binding of `a` in list `l`. Raise `Not_found` if there is no value associated with `a` in the list `l`.

```
val assq : 'a -> ('a * 'b) list -> 'b
```

> Same as `assoc`, but uses physical equality instead of structural equality to compare keys.

```
val mem_assoc : 'a -> ('a * 'b) list -> bool
```

> Same as `assoc`, but simply return true if a binding exists, and false if no bindings exist for the given key.

```
val mem_assq : 'a -> ('a * 'b) list -> bool
```

> Same as `mem_assoc`, but uses physical equality instead of structural equality to compare keys.

```
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
```

> `remove_assoc a l` returns the list of pairs `l` without the first pair with key `a`, if any.

```
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
```

> Same as `remove_assq`, but uses physical equality instead of structural equality to compare keys.

**Lists of pairs**

```
val split : ('a * 'b) list -> 'a list * 'b list
```

Transform a list of pairs into a pair of lists: `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`

```
val combine : 'a list -> 'b list -> ('a * 'b) list
```

Transform a pair of lists into a list of pairs: `combine ([a1; ...; an], [b1; ...; bn])` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument` if the two lists have different lengths.

## 17.15 Module `Map`: association tables over ordered types

This module implements applicative association tables, also known as finite maps or dictionaries, given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses balanced binary trees, and therefore searching and insertion take time logarithmic in the size of the map.

```
module type OrderedType =
  sig
    type t
    val compare: t -> t -> int
  end
```

The input signature of the functor `Map.Make`. `t` is the type of the map keys. `compare` is a total ordering function over the keys. This is a two-argument function `f` such that `f e1 e2` is zero if the keys `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Example: a suitable ordering function is the generic structural comparison function `compare`.

```
module type S =
  sig
    type key
```

The type of the map keys.

```
    type 'a t
```

The type of maps from type `key` to type `'a`.

```
    val empty: 'a t
```

The empty map.

```
    val add: key -> 'a -> 'a t -> 'a t
```

`add x y m` returns a map containing the same bindings as `m`, plus a binding of `x` to `y`. If `x` was already bound in `m`, its previous binding disappears.

```
    val find: key -> 'a t -> 'a
```

find x m returns the current binding of x in m, or raises Not_found if no such binding exists.

```
    val remove: key -> 'a t -> 'a t
```

remove x m returns a map containing the same bindings as m, except for x which is unbound in the returned map.

```
    val mem:  key -> 'a t -> bool
```

mem x m returns true if m contains a binding for m, and false otherwise.

```
    val iter: (key -> 'a -> unit) -> 'a t -> unit
```

iter f m applies f to all bindings in map m. f receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to f is unspecified. Only current bindings are presented to f: bindings hidden by more recent bindings are not passed to f.

```
    val map: ('a -> 'b) -> 'a t -> 'b t
```

map f m returns a map with same domain as m, where the associated value a of all bindings of m has been replaced by the result of the application of f to a. The order in which the associated values are passed to f is unspecified.

```
    val fold: (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
```

fold f m a computes (f kN dN ... (f k1 d1 a)...), where k1 ... kN are the keys of all bindings in m, and d1 ... dN are the associated data. The order in which the bindings are presented to f is unspecified.

```
  end
module Make(Ord: OrderedType): (S with type key = Ord.t)
```

Functor building an implementation of the map structure given a totally ordered type.

## 17.16 Module Marshal: marshaling of data structures

This module provides functions to encode arbitrary data structures as sequences of bytes, which can then be written on a file or sent over a pipe or network connection. The bytes can then be read back later, possibly in another process, and decoded back into a data structure. The format for the byte sequences is compatible across all machines for a given version of Objective Caml.

Warning: marshaling is currently not type-safe. The type of marshaled data is not transmitted along the value of the data, making it impossible to check that the data read back possesses the type expected by the context. In particular, the result type of the Marshal.from_* functions is given as 'a, but this is misleading: the returned Caml value does not possess type 'a for all 'a; it has one, unique type which cannot be determined at compile-type. The programmer should explicitly give the expected type of the returned value, using the following syntax: `(Marshal.from_channel chan : type)`. Anything can happen at run-time if the object in the file does not belong to the given type.

The representation of marshaled values is not human-readable, and uses bytes that are not printable characters. Therefore, input and output channels used in conjunction with `Marshal.to_channel` and `Marshal.from_channel` must be opened in binary mode, using e.g. `open_out_bin` or `open_in_bin`; channels opened in text mode will cause unmarshaling errors on platforms where text channels behave differently than binary channels, e.g. Windows.

```
type extern_flags =
    No_sharing                                    (* Don't preserve sharing *)
  | Closures                                      (* Send function closures *)
```

The flags to the `Marshal.to_*` functions below.

```
val to_channel: out_channel -> 'a -> extern_flags list -> unit
```

`Marshal.to_channel chan v flags` writes the representation of `v` on channel `chan`. The `flags` argument is a possibly empty list of flags that governs the marshaling behavior with respect to sharing and functional values.

If `flags` does not contain `Marshal.No_sharing`, circularities and sharing inside the value `v` are detected and preserved in the sequence of bytes produced. In particular, this guarantees that marshaling always terminates. Sharing between values marshaled by successive calls to `Marshal.to_channel` is not detected, though. If `flags` contains `Marshal.No_sharing`, sharing is ignored. This results in faster marshaling if `v` contains no shared substructures, but may cause slower marshaling and larger byte representations if `v` actually contains sharing, or even non-termination if `v` contains cycles.

If `flags` does not contain `Marshal.Closures`, marshaling fails when it encounters a functional value inside `v`: only "pure" data structures, containing neither functions nor objects, can safely be transmitted between different programs. If `flags` contains `Marshal.Closures`, functional values will be marshaled as a position in the code of the program. In this case, the output of marshaling can only be read back in processes that run exactly the same program, with exactly the same compiled code. (This is checked at un-marshaling time, using an MD5 digest of the code transmitted along with the code position.)

```
val to_string: 'a -> extern_flags list -> string
```

`Marshal.to_string v flags` returns a string containing the representation of `v` as a sequence of bytes. The `flags` argument has the same meaning as for `Marshal.to_channel`.

```
val to_buffer: string -> int -> int -> 'a -> extern_flags list -> int
```

`Marshal.to_buffer buff ofs len v flags` marshals the value `v`, storing its byte representation in the string `buff`, starting at character number `ofs`, and writing at most `len` characters. It returns the number of characters actually written to the string. If the byte representation of `v` does not fit in `len` characters, the exception `Failure` is raised.

```
val from_channel: in_channel -> 'a
```

`Marshal.from_channel chan` reads from channel `chan` the byte representation of a structured value, as produced by one of the `Marshal.to_*` functions, and reconstructs and returns the corresponding value.

```
val from_string: string -> int -> 'a
```

`Marshal.from_string buff ofs` unmarshals a structured value like `Marshal.from_channel` does, except that the byte representation is not read from a channel, but taken from the string `buff`, starting at position `ofs`.

```
val header_size : int
val data_size : string -> int -> int
val total_size : string -> int -> int
```

The bytes representing a marshaled value are composed of a fixed-size header and a variable-sized data part, whose size can be determined from the header. `Marshal.header_size` is the size, in characters, of the header. `Marshal.data_size buff ofs` is the size, in characters, of the data part, assuming a valid header is stored in `buff` starting at position `ofs`. Finally, `Marshal.total_size buff ofs` is the total size, in characters, of the marshaled value. Both `Marshal.data_size` and `Marshal.total_size` raise `Failure` if `buff`, `ofs` does not contain a valid header.

To read the byte representation of a marshaled value into a string buffer, the program needs to read first `Marshal.header_size` characters into the buffer, then determine the length of the remainder of the representation using `Marshal.data_size`, make sure the buffer is large enough to hold the remaining data, then read it, and finally call `Marshal.from_string` to unmarshal the value.

## 17.17 Module `Oo`: object-oriented extension

```
val copy : < .. > as 'a -> 'a
```

`Oo.copy o` returns a copy of object `o`, that is a fresh object with the same methods and instance variables as `o`

## 17.18 Module `Parsing`: the run-time library for parsers generated by `ocamlyacc`

```
val symbol_start : unit -> int
val symbol_end : unit -> int
```

`symbol_start` and `symbol_end` are to be called in the action part of a grammar rule only. They return the position of the string that matches the left-hand side of the rule: `symbol_start()` returns the position of the first character; `symbol_end()` returns the position of the last character, plus one. The first character in a file is at position 0.

```
val rhs_start: int -> int
val rhs_end: int -> int
```

Same as `symbol_start` and `symbol_end`, but return the position of the string matching the `n`th item on the right-hand side of the rule, where `n` is the integer parameter to `lhs_start` and `lhs_end`. `n` is 1 for the leftmost item.

```
val clear_parser : unit -> unit
```

Empty the parser stack. Call it just after a parsing function has returned, to remove all pointers from the parser stack to structures that were built by semantic actions during parsing. This is optional, but lowers the memory requirements of the programs.

```
exception Parse_error
```

Raised when a parser encounters a syntax error. Can also be raised from the action part of a grammar rule, to initiate error recovery.

## 17.19 Module `Printexc`: a catch-all exception handler

```
val catch: ('a -> 'b) -> 'a -> 'b
```

`Printexc.catch fn x` applies `fn` to `x` and returns the result. If the evaluation of `fn x` raises any exception, the name of the exception is printed on standard error output, and the programs aborts with exit code 2. Typical use is `Printexc.catch main ()`, where `main`, with type `unit->unit`, is the entry point of a standalone program. This catches and reports any exception that escapes the program.

```
val print: ('a -> 'b) -> 'a -> 'b
```

Same as `catch`, but re-raise the stray exception after printing it, instead of aborting the program.

```
val to_string : exn -> string
```

`Printexc.to_string e` returns a string representation of `e`.

## 17.20 Module `Printf`: formatting printing functions

`val fprintf: out_channel -> ('a, out_channel, unit) format -> 'a`

> `fprintf outchan format arg1 ... argN` formats the arguments `arg1` to `argN` according to the format string `format`, and outputs the resulting string on the channel `outchan`.
>
> The format is a character string which contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which causes conversion and printing of one argument.
>
> Conversion specifications consist in the `%` character, followed by optional flags and field widths, followed by one conversion character. The conversion characters and their meanings are:
>
> `d` or `i`: convert an integer argument to signed decimal
>
> `u`: convert an integer argument to unsigned decimal
>
> `x`: convert an integer argument to unsigned hexadecimal, using lowercase letters.
>
> `X`: convert an integer argument to unsigned hexadecimal, using uppercase letters.
>
> `s`: insert a string argument
>
> `c`: insert a character argument
>
> `f`: convert a floating-point argument to decimal notation, in the style `dddd.ddd`
>
> `e` or `E`: convert a floating-point argument to decimal notation, in the style `d.ddd e+-dd` (mantissa and exponent)
>
> `g` or `G`: convert a floating-point argument to decimal notation, in style `f` or `e`, `E` (whichever is more compact)
>
> `b`: convert a boolean argument to the string `true` or `false`
>
> `a`: user-defined printer. Takes two arguments and apply the first one to `outchan` (the current output channel) and to the second argument. The first argument must therefore have type `out_channel -> 'b -> unit` and the second `'b`. The output produced by the function is therefore inserted in the output of `fprintf` at the current point.
>
> `t`: same as `%a`, but takes only one argument (with type `out_channel -> unit`) and apply it to `outchan`.
>
> `%`: take no argument and output one `%` character.
>
> Refer to the C library `printf` function for the meaning of flags and field width specifiers.
>
> If too few arguments are provided, printing stops just before converting the first missing argument.

`val printf: ('a, out_channel, unit) format -> 'a`

> Same as `fprintf`, but output on `stdout`.

`val eprintf: ('a, out_channel, unit) format -> 'a`

> Same as `fprintf`, but output on `stderr`.

`val sprintf: ('a, unit, string) format -> 'a`

Same as `fprintf`, but instead of printing on an output channel, return a string containing the result of formatting the arguments.

`val bprintf: Buffer.t -> ('a, Buffer.t, unit) format -> 'a`

Same as `fprintf`, but instead of printing on an output channel, append the formatted arguments to the given extensible buffer (see module `Buffer`).

## 17.21 Module `Queue`: first-in first-out queues

This module implements queues (FIFOs), with in-place modification.

`type 'a t`

The type of queues containing elements of type `'a`.

`exception Empty`

Raised when `take` is applied to an empty queue.

`val create: unit -> 'a t`

Return a new queue, initially empty.

`val add: 'a -> 'a t -> unit`

`add x q` adds the element `x` at the end of the queue `q`.

`val take: 'a t -> 'a`

`take q` removes and returns the first element in queue `q`, or raises `Empty` if the queue is empty.

`val peek: 'a t -> 'a`

`peek q` returns the first element in queue `q`, without removing it from the queue, or raises `Empty` if the queue is empty.

`val clear : 'a t -> unit`

Discard all elements from a queue.

`val length: 'a t -> int`

Return the number of elements in a queue.

`val iter: ('a -> unit) -> 'a t -> unit`

`iter f q` applies `f` in turn to all elements of `q`, from the least recently entered to the most recently entered. The queue itself is unchanged.

## 17.22 Module `Random`: pseudo-random number generator

```
val init : int -> unit
```

Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

```
val full_init : int array -> unit
```

Same as `init` but takes more data as seed.

```
val bits : unit -> int
```

Return 30 random bits in a nonnegative integer.

```
val int : int -> int
```

`Random.int bound` returns a random integer between 0 (inclusive) and `bound` (exclusive). `bound` must be more than 0 and less than $2^{30}$.

```
val float : float -> float
```

`Random.float bound` returns a random floating-point number between 0 (inclusive) and `bound` (exclusive). If `bound` is negative, the result is negative. If `bound` is 0, the result is 0.

## 17.23 Module `Set`: sets over ordered types

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

```
module type OrderedType =
  sig
    type t
    val compare: t -> t -> int
  end
```

The input signature of the functor `Set.Make`. `t` is the type of the set elements. `compare` is a total ordering function over the set elements. This is a two-argument function `f` such that `f e1 e2` is zero if the elements `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Example: a suitable ordering function is the generic structural comparison function `compare`.

```
module type S =
  sig
    type elt
```

The type of the set elements.

```
type t
```

The type of sets.

```
val empty: t
```

The empty set.

```
val is_empty: t -> bool
```

Test whether a set is empty or not.

```
val mem: elt -> t -> bool
```

`mem x s` tests whether `x` belongs to the set `s`.

```
val add: elt -> t -> t
```

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

```
val singleton: elt -> t
```

`singleton x` returns the one-element set containing only `x`.

```
val remove: elt -> t -> t
```

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

```
val union: t -> t -> t
val inter: t -> t -> t
val diff: t -> t -> t
```

Union, intersection and set difference.

```
val compare: t -> t -> int
```

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

```
val equal: t -> t -> bool
```

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain equal elements.

```
val subset: t -> t -> bool
```

`subset s1 s2` tests whether the set `s1` is a subset of the set `s2`.

```
val iter: (elt -> unit) -> t -> unit
```

`iter f s` applies `f` in turn to all elements of `s`. The order in which the elements of `s` are presented to `f` is unspecified.

```
val fold: (elt -> 'a -> 'a) -> t -> 'a -> 'a
```

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is unspecified.

```
val cardinal: t -> int
```

Return the number of elements of a set.

```
val elements: t -> elt list
```

Return the list of all elements of the given set. The returned list is sorted in increasing order with respect to the ordering `Ord.compare`, where `Ord` is the argument given to `Set.Make`.

```
val min_elt: t -> elt
```

Return the smallest element of the given set (with respect to the `Ord.compare` ordering), or raise `Not_found` if the set is empty.

```
val max_elt: t -> elt
```

Same as `min_elt`, but returns the largest element of the given set.

```
val choose: t -> elt
```

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.

```
  end
module Make(Ord: OrderedType): (S with type elt = Ord.t)
```

Functor building an implementation of the set structure given a totally ordered type.

## 17.24 Module `Sort`: sorting and merging lists

```
val list : ('a -> 'a -> bool) -> 'a list -> 'a list
```

Sort a list in increasing order according to an ordering predicate. The predicate should return `true` if its first argument is less than or equal to its second argument.

```
val array : ('a -> 'a -> bool) -> 'a array -> unit
```

Sort an array in increasing order according to an ordering predicate. The predicate should return `true` if its first argument is less than or equal to its second argument. The array is sorted in place.

```
val merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list
```

Merge two lists according to the given predicate. Assuming the two argument lists are sorted according to the predicate, `merge` returns a sorted list containing the elements from the two lists. The behavior is undefined if the two argument lists were not sorted.

## 17.25 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.

## 17.26 Module `Stream`: streams and parsers

`type 'a t`

The type of streams holding values of type `'a`.

`exception Failure;;`

Raised by parsers when none of the first components of the stream patterns is accepted.

`exception Error of string;;`

Raised by parsers when the first component of a stream pattern is accepted, but one of the following components is rejected.

### Stream builders

Warning: these functions create streams with fast access; it is illegal to mix them with streams built with `[< >]`; would raise `Failure` when accessing such mixed streams.

```
val from : (int -> 'a option) -> 'a t;;
```

`Stream.from f` returns a stream built from the function `f`. To create a new stream element, the function `f` is called with the current stream count. The user function `f` must return either `Some <value>` for a value or `None` to specify the end of the stream.

```
val of_list : 'a list -> 'a t;;
```

Return the stream holding the elements of the list in the same order.

```
val of_string : string -> char t;;
```

Return the stream of the characters of the string parameter.

```
val of_channel : in_channel -> char t;;
```

Return the stream of the characters read from the input channel.

### Stream iterator

```
val iter : ('a -> unit) -> 'a t -> unit;;
```

`Stream.iter f s` scans the whole stream s, applying function `f` in turn to each stream element encountered.

### Predefined parsers

```
val next : 'a t -> 'a;;
```

Return the first element of the stream and remove it from the stream. Raise `Stream.Failure` if the stream is empty.

```
val empty : 'a t -> unit;;
```

Return `()` if the stream is empty, else raise `Stream.Failure`.

#### Useful functions

`val peek : 'a t -> 'a option;;`

Return `Some` of "the first element" of the stream, or `None` if the stream is empty.

`val junk : 'a t -> unit;;`

Remove the first element of the stream, possibly unfreezing it before.

`val count : 'a t -> int = "%field0";;`

Return the current count of the stream elements, i.e. the number of the stream elements discarded.

`val npeek : int -> 'a t -> 'a list;;`

`npeek n` returns the list of the `n` first elements of the stream, or all its remaining elements if less than `n` elements are available.

## 17.27 Module `String`: string operations

`val length : string -> int`

Return the length (number of characters) of the given string.

`val get : string -> int -> char`

`String.get s n` returns character number `n` in string `s`. The first character is character number 0. The last character is character number `String.length s - 1`. Raise `Invalid_argument` if `n` is outside the range 0 to `(String.length s - 1)`. You can also write `s.[n]` instead of `String.get s n`.

`val set : string -> int -> char -> unit`

`String.set s n c` modifies string `s` in place, replacing the character number `n` by `c`. Raise `Invalid_argument` if `n` is outside the range 0 to `(String.length s - 1)`. You can also write `s.[n] <- c` instead of `String.set s n c`.

`val create : int -> string`

`String.create n` returns a fresh string of length `n`. The string initially contains arbitrary characters.

`val make : int -> char -> string`

`String.make n c` returns a fresh string of length `n`, filled with the character `c`.

`val copy : string -> string`

Return a copy of the given string.

`val sub : string -> int -> int -> string`

`String.sub s start len` returns a fresh string of length `len`, containing the characters number `start` to `start + len - 1` of string `s`. Raise `Invalid_argument` if `start` and `len` do not designate a valid substring of `s`; that is, if `start < 0`, or `len < 0`, or `start + len > String.length s`.

`val fill : string -> int -> int -> char -> unit`

`String.fill s start len c` modifies string `s` in place, replacing the characters number `start` to `start + len - 1` by `c`. Raise `Invalid_argument` if `start` and `len` do not designate a valid substring of `s`.

`val blit : string -> int -> string -> int -> int -> unit`

`String.blit src srcoff dst dstoff len` copies `len` characters from string `src`, starting at character number `srcoff`, to string `dst`, starting at character number `dstoff`. It works correctly even if `src` and `dst` are the same string, and the source and destination chunks overlap. Raise `Invalid_argument` if `srcoff` and `len` do not designate a valid substring of `src`, or if `dstoff` and `len` do not designate a valid substring of `dst`.

`val concat : string -> string list -> string`

`String.concat sep sl` catenates the list of strings `sl`, inserting the separator string `sep` between each.

`val escaped: string -> string`

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Objective Caml.

`val index: string -> char -> int`

`String.index s c` returns the position of the leftmost occurrence of character `c` in string `s`. Raise `Not_found` if `c` does not occur in `s`.

`val rindex: string -> char -> int`

`String.rindex s c` returns the position of the rightmost occurrence of character `c` in string `s`. Raise `Not_found` if `c` does not occur in `s`.

```
val index_from: string -> int -> char -> int
val rindex_from: string -> int -> char -> int
```

Same as `String.index` and `String.rindex`, but start searching at the character position given as second argument. `String.index s c` is equivalent to `String.index_from s 0 c`, and `String.rindex s c` to `String.rindex_from s (String.length s - 1) c`.

`val contains : string -> char -> bool`

`String.contains s c` tests if character `c` appears in the string `s`.

`val contains_from : string -> int -> char -> bool`

> `String.contains_from s start c` tests if character `c` appears in the substring of `s` starting from `start` to the end of `s`. Raise `Invalid_argument` if `start` is not a valid index of `s`.

`val rcontains_from : string -> int -> char -> bool`

> `String.rcontains_from s stop c` tests if character `c` appears in the substring of `s` starting from the beginning of `s` to index `stop`. Raise `Invalid_argument` if `stop` is not a valid index of `s`.

`val uppercase: string -> string`

> Return a copy of the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val lowercase: string -> string`

> Return a copy of the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.

`val capitalize: string -> string`

> Return a copy of the argument, with the first letter set to uppercase.

`val uncapitalize: string -> string`

> Return a copy of the argument, with the first letter set to lowercase.

## 17.28 Module `Sys`: system interface

`val argv: string array`

> The command line arguments given to the process. The first element is the command name used to invoke the program. The following elements are the command-line arguments given to the program.

`val file_exists: string -> bool`

> Test if a file with the given name exists.

`val remove: string -> unit`

> Remove the given file name from the file system.

`val rename : string -> string -> unit`

> Rename a file. The first argument is the old name and the second is the new name.

`val getenv: string -> string`

Return the value associated to a variable in the process environment. Raise `Not_found` if the variable is unbound.

`val command: string -> int`

Execute the given shell command and return its exit code.

`val time: unit -> float`

Return the processor time, in seconds, used by the program since the beginning of execution.

`val chdir: string -> unit`

Change the current working directory of the process.

`val getcwd: unit -> string`

Return the current working directory of the process.

`val interactive: bool ref`

This reference is initially set to `false` in standalone programs and to `true` if the code is being executed under the interactive toplevel system `ocaml`.

`val os_type: string`

Operating system currently executing the Caml program. One of `"Unix"`, `"Win32"`, or `"MacOS"`.

`val word_size: int`

Size of one word on the machine currently executing the Caml program, in bits: 32 or 64.

`val max_string_length: int`

Maximum length of a string.

`val max_array_length: int`

Maximum length of an array.

## Signal handling

```
type signal_behavior =
    Signal_default
  | Signal_ignore
  | Signal_handle of (int -> unit)
```

What to do when receiving a signal:

`Signal_default`: take the default behavior (usually: abort the program)

`Signal_ignore`: ignore the signal

`Signal_handle f`: call function `f`, giving it the signal number as argument.

```
val signal: int -> signal_behavior -> signal_behavior
```

Set the behavior of the system on receipt of a given signal. The first argument is the signal number. Return the behavior previously associated with the signal.

```
val set_signal: int -> signal_behavior -> unit
```

Same as `signal` but return value is ignored.

```
val sigabrt: int   (* Abnormal termination *)
val sigalrm: int   (* Timeout *)
val sigfpe: int    (* Arithmetic exception *)
val sighup: int    (* Hangup on controlling terminal *)
val sigill: int    (* Invalid hardware instruction *)
val sigint: int    (* Interactive interrupt (ctrl-C) *)
val sigkill: int   (* Termination (cannot be ignored) *)
val sigpipe: int   (* Broken pipe *)
val sigquit: int   (* Interactive termination *)
val sigsegv: int   (* Invalid memory reference *)
val sigterm: int   (* Termination *)
val sigusr1: int   (* Application-defined signal 1 *)
val sigusr2: int   (* Application-defined signal 2 *)
val sigchld: int   (* Child process terminated *)
val sigcont: int   (* Continue *)
val sigstop: int   (* Stop *)
val sigtstp: int   (* Interactive stop *)
val sigttin: int   (* Terminal read from background process *)
val sigttou: int   (* Terminal write from background process *)
val sigvtalrm: int (* Timeout in virtual time *)
val sigprof: int   (* Profiling interrupt *)
```

Signal numbers for the standard POSIX signals.

```
exception Break
```

Exception raised on interactive interrupt if `catch_break` is on.

```
val catch_break: bool -> unit
```

`catch_break` governs whether interactive interrupt (ctrl-C) terminates the program or raises the `Break` exception. Call `catch_break true` to enable raising `Break`, and `catch_break false` to let the system terminate the program on user interrupt.

## 17.29 Module `Weak`: arrays of weak pointers

```
type 'a t;;
```

The type of arrays of weak pointers (weak arrays). A weak pointer is an object that the garbage collector may erase at any time. A weak pointer is said to be full if it points to an object, empty if the object was erased by the GC.

```
val create : int -> 'a t;;
```

`Weak.create n` returns a new weak array of length `n`. All the pointers are initially empty.

```
val length : 'a t -> int;;
```

`Weak.length ar` returns the length (number of elements) of `ar`.

```
val set : 'a t -> int -> 'a option -> unit;;
```

`Weak.set ar n (Some el)` sets the `n`th cell of `ar` to be a (full) pointer to `el`; `Weak.set ar n None` sets the `n`th cell of `ar` to empty. Raise `Invalid_argument "Weak.set"` if `n` is not in the range 0 to `Weak.length a - 1`.

```
val get : 'a t -> int -> 'a option;;
```

`Weak.get ar n` returns None if the `n`th cell of `ar` is empty, `Some x` (where `x` is the object) if it is full. Raise `Invalid_argument "Weak.get"` if `n` is not in the range 0 to `Weak.length a - 1`.

```
val check: 'a t -> int -> bool;;
```

`Weak.check ar n` returns `true` if the `n`th cell of `ar` is full, `false` if it is empty. Note that even if `Weak.check ar n` returns `true`, a subsequent `Weak.get ar n` can return `None`.

```
val fill: 'a t -> int -> int -> 'a option -> unit;;
```

`Weak.fill ar ofs len el` sets to `el` all pointers of `ar` from `ofs` to `ofs + len - 1`. Raise `Invalid_argument "Weak.fill"` if `ofs` and `len` do not designate a valid subarray of `a`.

```
val blit : 'a t -> int -> 'a t -> int -> int -> unit;;
```

`Weak.blit ar1 off1 ar2 off2 len` copies `len` weak pointers from `ar1` (starting at `off1`) to `ar2` (starting at `off2`). It works correctly even if `ar1` and `ar2` are the same. Raise `Invalid_argument "Weak.blit"` if `off1` and `len` do not designate a valid subarray of `ar1`, or if `off2` and `len` do not designate a valid subarray of `ar2`.

# Chapter 18

# The unix library: Unix system calls

The `unix` library makes many Unix system calls and system-related library functions available to Objective Caml programs. This chapter describes briefly the functions provided. Refer to sections 2 and 3 of the Unix manual for more details on the behavior of these functions.

Not all functions are provided by all Unix variants. If some functions are not available, they will raise `Invalid_arg` when called.

Programs that use the `unix` library must be linked in "custom runtime" mode, as follows:

```
ocamlc -custom other options unix.cma other files -cclib -lunix
ocamlopt other options unix.cmxa other files -cclib -lunix
```

For interactive use of the `unix` library, do:

```
ocamlmktop -custom -o mytop unix.cma -cclib -lunix
./mytop
```

**Windows:**

A fairly complete emulation of the Unix system calls is provided in the Windows version of Objective Caml. The end of this chapter gives more information on the functions that are not supported under Windows.

## 18.1 Module `Unix`: interface to the Unix system

### Error report

```
type error =
```

Errors defined in the POSIX standard

```
    E2BIG               (* Argument list too long *)
  | EACCES              (* Permission denied *)
  | EAGAIN              (* Resource temporarily unavailable; try again *)
  | EBADF               (* Bad file descriptor *)
  | EBUSY               (* Resource unavailable *)
  | ECHILD              (* No child process *)
```

```
| EDEADLK            (* Resource deadlock would occur *)
| EDOM               (* Domain error for math functions, etc. *)
| EEXIST             (* File exists *)
| EFAULT             (* Bad address *)
| EFBIG              (* File too large *)
| EINTR              (* Function interrupted by signal *)
| EINVAL             (* Invalid argument *)
| EIO                (* Hardware I/O error *)
| EISDIR             (* Is a directory *)
| EMFILE             (* Too many open files by the process *)
| EMLINK             (* Too many links *)
| ENAMETOOLONG       (* Filename too long *)
| ENFILE             (* Too many open files in the system *)
| ENODEV             (* No such device *)
| ENOENT             (* No such file or directory *)
| ENOEXEC            (* Not an executable file *)
| ENOLCK             (* No locks available *)
| ENOMEM             (* Not enough memory *)
| ENOSPC             (* No space left on device *)
| ENOSYS             (* Function not supported *)
| ENOTDIR            (* Not a directory *)
| ENOTEMPTY          (* Directory not empty *)
| ENOTTY             (* Inappropriate I/O control operation *)
| ENXIO              (* No such device or address *)
| EPERM              (* Operation not permitted *)
| EPIPE              (* Broken pipe *)
| ERANGE             (* Result too large *)
| EROFS              (* Read-only file system *)
| ESPIPE             (* Invalid seek e.g. on a pipe *)
| ESRCH              (* No such process *)
| EXDEV              (* Invalid link *)
```

Additional errors, mostly BSD

```
| EWOULDBLOCK        (* Operation would block *)
| EINPROGRESS        (* Operation now in progress *)
| EALREADY           (* Operation already in progress *)
| ENOTSOCK           (* Socket operation on non-socket *)
| EDESTADDRREQ       (* Destination address required *)
| EMSGSIZE           (* Message too long *)
| EPROTOTYPE         (* Protocol wrong type for socket *)
| ENOPROTOOPT        (* Protocol not available *)
| EPROTONOSUPPORT    (* Protocol not supported *)
| ESOCKTNOSUPPORT    (* Socket type not supported *)
| EOPNOTSUPP         (* Operation not supported on socket *)
| EPFNOSUPPORT       (* Protocol family not supported *)
```

```
  | EAFNOSUPPORT        (* Address family not supported by protocol family *)
  | EADDRINUSE          (* Address already in use *)
  | EADDRNOTAVAIL       (* Can't assign requested address *)
  | ENETDOWN            (* Network is down *)
  | ENETUNREACH         (* Network is unreachable *)
  | ENETRESET           (* Network dropped connection on reset *)
  | ECONNABORTED        (* Software caused connection abort *)
  | ECONNRESET          (* Connection reset by peer *)
  | ENOBUFS             (* No buffer space available *)
  | EISCONN             (* Socket is already connected *)
  | ENOTCONN            (* Socket is not connected *)
  | ESHUTDOWN           (* Can't send after socket shutdown *)
  | ETOOMANYREFS        (* Too many references: can't splice *)
  | ETIMEDOUT           (* Connection timed out *)
  | ECONNREFUSED        (* Connection refused *)
  | EHOSTDOWN           (* Host is down *)
  | EHOSTUNREACH        (* No route to host *)
  | ELOOP               (* Too many levels of symbolic links *)
```

All other errors are mapped to EUNKNOWNERR

```
  | EUNKNOWNERR of int  (* Unknown error *)
```

The type of error codes.

```
exception Unix_error of error * string * string
```

Raised by the system calls below when an error is encountered. The first component is the error code; the second component is the function name; the third component is the string parameter to the function, if it has one, or the empty string otherwise.

```
val error_message : error -> string
```

Return a string describing the given error code.

```
val handle_unix_error : ('a -> 'b) -> 'a -> 'b
```

`handle_unix_error f x` applies `f` to `x` and returns the result. If the exception `Unix_error` is raised, it prints a message describing the error and exits with code 2.

## Access to the process environment

```
val environment : unit -> string array
```

Return the process environment, as an array of strings with the format "variable=value".

```
val getenv: string -> string
```

Return the value associated to a variable in the process environment. Raise `Not_found` if the variable is unbound. (This function is identical to `Sys.getenv`.)

```
val putenv: string -> string -> unit
```

`Unix.putenv name value` sets the value associated to a variable in the process environment. `name` is the name of the environment variable, and `value` its new associated value.

## Process handling

```
type process_status =
    WEXITED of int
  | WSIGNALED of int
  | WSTOPPED of int
```

The termination status of a process. `WEXITED` means that the process terminated normally by `exit`; the argument is the return code. `WSIGNALED` means that the process was killed by a signal; the argument is the signal number. `WSTOPPED` means that the process was stopped by a signal; the argument is the signal number.

```
type wait_flag =
    WNOHANG
  | WUNTRACED
```

Flags for `waitopt` and `waitpid`. `WNOHANG` means do not block if no child has died yet, but immediately return with a pid equal to 0. `WUNTRACED` means report also the children that receive stop signals.

```
val execv : string -> string array -> unit
```

`execv prog args` execute the program in file `prog`, with the arguments `args`, and the current process environment.

```
val execve : string -> string array -> string array -> unit
```

Same as `execv`, except that the third argument provides the environment to the program executed.

```
val execvp : string -> string array -> unit
val execvpe : string -> string array -> string array -> unit
```

Same as `execv` and `execvp` respectively, except that the program is searched in the path.

```
val fork : unit -> int
```

Fork a new process. The returned integer is 0 for the child process, the pid of the child process for the parent process.

```
val wait : unit -> int * process_status
```

Wait until one of the children processes die, and return its pid and termination status.

```
val waitpid : wait_flag list -> int -> int * process_status
```

Same as `wait`, but waits for the process whose pid is given. A pid of `-1` means wait for any child. A pid of `0` means wait for any child in the same process group as the current process. Negative pid arguments represent process groups. The list of options indicates whether `waitpid` should return immediately without waiting, or also report stopped children.

```
val system : string -> process_status
```

Execute the given command, wait until it terminates, and return its termination status. The string is interpreted by the shell `/bin/sh` and therefore can contain redirections, quotes, variables, etc. The result `WEXITED 127` indicates that the shell couldn't be executed.

```
val getpid : unit -> int
```

Return the pid of the process.

```
val getppid : unit -> int
```

Return the pid of the parent process.

```
val nice : int -> int
```

Change the process priority. The integer argument is added to the "nice" value. (Higher values of the "nice" value mean lower priorities.) Return the new nice value.

## Basic file input/output

```
type file_descr
```

The abstract type of file descriptors.

```
val stdin : file_descr
val stdout : file_descr
val stderr : file_descr
```

File descriptors for standard input, standard output and standard error.

```
type open_flag =
    O_RDONLY                           (* Open for reading *)
  | O_WRONLY                           (* Open for writing *)
  | O_RDWR                             (* Open for reading and writing *)
  | O_NONBLOCK                         (* Open in non-blocking mode *)
  | O_APPEND                           (* Open for append *)
  | O_CREAT                            (* Create if nonexistent *)
  | O_TRUNC                            (* Truncate to 0 length if existing *)
  | O_EXCL                             (* Fail if existing *)
```

The flags to `open`.

`type file_perm = int`

The type of file access rights.

`val openfile : string -> open_flag list -> file_perm -> file_descr`

Open the named file with the given flags. Third argument is the permissions to give to the file if it is created. Return a file descriptor on the named file.

`val close : file_descr -> unit`

Close a file descriptor.

`val read : file_descr -> string -> int -> int -> int`

`read fd buff ofs len` reads `len` characters from descriptor `fd`, storing them in string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually read.

`val write : file_descr -> string -> int -> int -> int`

`write fd buff ofs len` writes `len` characters to descriptor `fd`, taking them from string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually written.

### Interfacing with the standard input/output library.

`val in_channel_of_descr : file_descr -> in_channel`

Create an input channel reading from the given descriptor. The channel is initially in binary mode; use `set_binary_mode_in ic false` if text mode is desired.

`val out_channel_of_descr : file_descr -> out_channel`

Create an output channel writing on the given descriptor. The channel is initially in binary mode; use `set_binary_mode_out oc false` if text mode is desired.

`val descr_of_in_channel : in_channel -> file_descr`

Return the descriptor corresponding to an input channel.

`val descr_of_out_channel : out_channel -> file_descr`

Return the descriptor corresponding to an output channel.

## Seeking and truncating

```
type seek_command =
    SEEK_SET
  | SEEK_CUR
  | SEEK_END
```

Positioning modes for `lseek`. `SEEK_SET` indicates positions relative to the beginning of the file, `SEEK_CUR` relative to the current position, `SEEK_END` relative to the end of the file.

```
val lseek : file_descr -> int -> seek_command -> int
```

Set the current position for a file descriptor

```
val truncate : string -> int -> unit
```

Truncates the named file to the given size.

```
val ftruncate : file_descr -> int -> unit
```

Truncates the file corresponding to the given descriptor to the given size.

## File statistics

```
type file_kind =
    S_REG                                (* Regular file *)
  | S_DIR                                (* Directory *)
  | S_CHR                                (* Character device *)
  | S_BLK                                (* Block device *)
  | S_LNK                                (* Symbolic link *)
  | S_FIFO                               (* Named pipe *)
  | S_SOCK                               (* Socket *)
type stats =
  { st_dev : int;                        (* Device number *)
    st_ino : int;                        (* Inode number *)
    st_kind : file_kind;                 (* Kind of the file *)
    st_perm : file_perm;                 (* Access rights *)
    st_nlink : int;                      (* Number of links *)
    st_uid : int;                        (* User id of the owner *)
    st_gid : int;                        (* Group id of the owner *)
    st_rdev : int;                       (* Device minor number *)
    st_size : int;                       (* Size in bytes *)
    st_atime : float;                    (* Last access time *)
    st_mtime : float;                    (* Last modification time *)
    st_ctime : float }                   (* Last status change time *)
```

The informations returned by the `stat` calls.

`val stat : string -> stats`

Return the information for the named file.

`val lstat : string -> stats`

Same as `stat`, but in case the file is a symbolic link, return the information for the link itself.

`val fstat : file_descr -> stats`

Return the information for the file associated with the given descriptor.

### Operations on file names

`val unlink : string -> unit`

Removes the named file

`val rename : string -> string -> unit`

`rename old new` changes the name of a file from `old` to `new`.

`val link : string -> string -> unit`

`link source dest` creates a hard link named `dest` to the file named `new`.

### File permissions and ownership

```
type access_permission =
    R_OK                                (* Read permission *)
  | W_OK                                (* Write permission *)
  | X_OK                                (* Execution permission *)
  | F_OK                                (* File exists *)
```

Flags for the `access` call.

`val chmod : string -> file_perm -> unit`

Change the permissions of the named file.

`val fchmod : file_descr -> file_perm -> unit`

Change the permissions of an opened file.

`val chown : string -> int -> int -> unit`

Change the owner uid and owner gid of the named file.

`val fchown : file_descr -> int -> int -> unit`

Change the owner uid and owner gid of an opened file.

`val umask : int -> int`

Set the process creation mask, and return the previous mask.

`val access : string -> access_permission list -> unit`

Check that the process has the given permissions over the named file. Raise `Unix_error` otherwise.

## Operations on file descriptors

```
val dup : file_descr -> file_descr
```

Return a new file descriptor referencing the same file as the given descriptor.

```
val dup2 : file_descr -> file_descr -> unit
```

`dup2 fd1 fd2` duplicates `fd1` to `fd2`, closing `fd2` if already opened.

```
val set_nonblock : file_descr -> unit
val clear_nonblock : file_descr -> unit
```

Set or clear the "non-blocking" flag on the given descriptor. When the non-blocking flag is set, reading on a descriptor on which there is temporarily no data available raises the `EAGAIN` or `EWOULDBLOCK` error instead of blocking; writing on a descriptor on which there is temporarily no room for writing also raises `EAGAIN` or `EWOULDBLOCK`.

```
val set_close_on_exec : file_descr -> unit
val clear_close_on_exec : file_descr -> unit
```

Set or clear the "close-on-exec" flag on the given descriptor. A descriptor with the close-on-exec flag is automatically closed when the current process starts another program with one of the `exec` functions.

## Directories

```
val mkdir : string -> file_perm -> unit
```

Create a directory with the given permissions.

```
val rmdir : string -> unit
```

Remove an empty directory.

```
val chdir : string -> unit
```

Change the process working directory.

```
val getcwd : unit -> string
```

Return the name of the current working directory.

```
val chroot : string -> unit
```

Change the process root directory.

```
type dir_handle
```

The type of descriptors over opened directories.

`val opendir : string -> dir_handle`

Open a descriptor on a directory

`val readdir : dir_handle -> string`

Return the next entry in a directory. Raise `End_of_file` when the end of the directory has been reached.

`val rewinddir : dir_handle -> unit`

Reposition the descriptor to the beginning of the directory

`val closedir : dir_handle -> unit`

Close a directory descriptor.

### Pipes and redirections

`val pipe : unit -> file_descr * file_descr`

Create a pipe. The first component of the result is opened for reading, that's the exit to the pipe. The second component is opened for writing, that's the entrance to the pipe.

`val mkfifo : string -> file_perm -> unit`

Create a named pipe with the given permissions.

### High-level process and redirection management

```
val create_process :
  string -> string array -> file_descr -> file_descr -> file_descr -> int
```

`create_process prog args new_stdin new_stdout new_stderr` forks a new process that executes the program in file `prog`, with arguments `args`. The pid of the new process is returned immediately; the new process executes concurrently with the current process. The standard input and outputs of the new process are connected to the descriptors `new_stdin`, `new_stdout` and `new_stderr`. Passing e.g. `stdout` for `new_stdout` prevents the redirection and causes the new process to have the same standard output as the current process. The executable file `prog` is searched in the path. The new process has the same environment as the current process. All file descriptors of the current process are closed in the new process, except those redirected to standard input and outputs.

```
val create_process_env :
  string -> string array -> string array ->
  file_descr -> file_descr -> file_descr -> int
```

`create_process_env prog args env new_stdin new_stdout new_stderr` works as `create_process`, except that the extra argument `env` specifies the environment passed to the program.

```
val open_process_in: string -> in_channel
val open_process_out: string -> out_channel
val open_process: string -> in_channel * out_channel
```

High-level pipe and process management. These functions run the given command in parallel with the program, and return channels connected to the standard input and/or the standard output of the command. The command is interpreted by the shell `/bin/sh` (cf. `system`). Warning: writes on channels are buffered, hence be careful to call `flush` at the right times to ensure correct synchronization.

```
val close_process_in: in_channel -> process_status
val close_process_out: out_channel -> process_status
val close_process: in_channel * out_channel -> process_status
```

Close channels opened by `open_process_in`, `open_process_out` and `open_process`, respectively, wait for the associated command to terminate, and return its termination status.

## Symbolic links

```
val symlink : string -> string -> unit
```

`symlink source dest` creates the file `dest` as a symbolic link to the file `source`.

```
val readlink : string -> string
```

Read the contents of a link.

## Polling

```
val select :
  file_descr list -> file_descr list -> file_descr list -> float ->
        file_descr list * file_descr list * file_descr list
```

Wait until some input/output operations become possible on some channels. The three list arguments are, respectively, a set of descriptors to check for reading (first argument), for writing (second argument), or for exceptional conditions (third argument). The fourth argument is the maximal timeout, in seconds; a negative fourth argument means no timeout (unbounded wait). The result is composed of three sets of descriptors: those ready for reading (first component), ready for writing (second component), and over which an exceptional condition is pending (third component).

### Locking

```
type lock_command =
    F_ULOCK       (* Unlock a region *)
  | F_LOCK        (* Lock a region for writing, and block if already locked *)
  | F_TLOCK       (* Lock a region for writing, or fail if already locked *)
  | F_TEST        (* Test a region for other process locks *)
  | F_RLOCK       (* Lock a region for reading, and block if already locked *)
  | F_TRLOCK      (* Lock a region for reading, or fail if already locked *)
```

Commands for `lockf`.

```
val lockf : file_descr -> lock_command -> int -> unit
```

`lockf fd cmd size` puts a lock on a region of the file opened as `fd`. The region starts at the current read/write position for `fd` (as set by `lseek`), and extends `size` bytes forward if `size` is positive, `size` bytes backwards if `size` is negative, or to the end of the file if `size` is zero. A write lock (set with `F_LOCK` or `F_TLOCK`) prevents any other process from acquiring a read or write lock on the region. A read lock (set with `F_RLOCK` or `F_TRLOCK`) prevents any other process from acquiring a write lock on the region, but lets other processes acquire read locks on it.

### Signals

```
val kill : int -> int -> unit
```

`kill pid sig` sends signal number `sig` to the process with id `pid`.

```
type sigprocmask_command = SIG_SETMASK | SIG_BLOCK | SIG_UNBLOCK
val sigprocmask: sigprocmask_command -> int list -> int list
```

`sigprocmask cmd sigs` changes the set of blocked signals. If `cmd` is `SIG_SETMASK`, blocked signals are set to those in the list `sigs`. If `cmd` is `SIG_BLOCK`, the signals in `sigs` are added to the set of blocked signals. If `cmd` is `SIG_UNBLOCK`, the signals in `sigs` are removed from the set of blocked signals. `sigprocmask` returns the set of previously blocked signals.

```
val sigpending: unit -> int list
```

Return the set of blocked signals that are currently pending.

```
val sigsuspend: int list -> unit
```

`sigsuspend sigs` atomically sets the blocked signals to `sig` and waits for a non-ignored, non-blocked signal to be delivered. On return, the blocked signals are reset to their initial value.

```
val pause : unit -> unit
```

Wait until a non-ignored, non-blocked signal is delivered.

## Time functions

```
type process_times =
  { tms_utime : float;          (* User time for the process *)
    tms_stime : float;          (* System time for the process *)
    tms_cutime : float;         (* User time for the children processes *)
    tms_cstime : float }        (* System time for the children processes *)
```

The execution times (CPU times) of a process.

```
type tm =
  { tm_sec : int;                       (* Seconds 0..59 *)
    tm_min : int;                       (* Minutes 0..59 *)
    tm_hour : int;                      (* Hours 0..23 *)
    tm_mday : int;                      (* Day of month 1..31 *)
    tm_mon : int;                       (* Month of year 0..11 *)
    tm_year : int;                      (* Year - 1900 *)
    tm_wday : int;                      (* Day of week (Sunday is 0) *)
    tm_yday : int;                      (* Day of year 0..365 *)
    tm_isdst : bool }                   (* Daylight time savings in effect *)
```

The type representing wallclock time and calendar date.

```
val time : unit -> float
```

Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

```
val gettimeofday : unit -> float
```

Same as `time`, but with resolution better than 1 second.

```
val gmtime : float -> tm
```

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes Greenwich meridian time zone, also known as UTC.

```
val localtime : float -> tm
```

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes the local time zone.

```
val mktime : tm -> float * tm
```

Convert a date and time, specified by the `tm` argument, into a time in seconds, as returned by `time`. Also return a normalized copy of the given `tm` record, with the `tm_wday`, `tm_yday`, and `tm_isdst` fields recomputed from the other fields. The `tm` argument is interpreted in the local time zone.

```
val alarm : int -> int
```

Schedule a `SIGALRM` signals after the given number of seconds.

```
val sleep : int -> unit
```

Stop execution for the given number of seconds.

```
val times : unit -> process_times
```

Return the execution times of the process.

```
val utimes : string -> float -> float -> unit
```

Set the last access time (second arg) and last modification time (third arg) for a file. Times are expressed in seconds from 00:00:00 GMT, Jan. 1, 1970.

```
type interval_timer =
    ITIMER_REAL
  | ITIMER_VIRTUAL
  | ITIMER_PROF
```

The three kinds of interval timers. `ITIMER_REAL` decrements in real time, and sends the signal `SIGALRM` when expired. `ITIMER_VIRTUAL` decrements in process virtual time, and sends `SIGVTALRM` when expired. `ITIMER_PROF` (for profiling) decrements both when the process is running and when the system is running on behalf of the process; it sends `SIGPROF` when expired.

```
type interval_timer_status =
  { it_interval: float;                (* Period *)
    it_value: float }                  (* Current value of the timer *)
```

The type describing the status of an interval timer

```
val getitimer: interval_timer -> interval_timer_status
```

Return the current status of the given interval timer.

```
val setitimer:
  interval_timer -> interval_timer_status -> interval_timer_status
```

`setitimer t s` sets the interval timer `t` and returns its previous status. The `s` argument is interpreted as follows: `s.it_value`, if nonzero, is the time to the next timer expiration; `s.it_interval`, if nonzero, specifies a value to be used in reloading it_value when the timer expires. Setting `s.it_value` to zero disable the timer. Setting `s.it_interval` to zero causes the timer to be disabled after its next expiration.

## User id, group id

```
val getuid : unit -> int
```

Return the user id of the user executing the process.

```
val geteuid : unit -> int
```

Return the effective user id under which the process runs.

```
val setuid : int -> unit
```

Set the real user id and effective user id for the process.

```
val getgid : unit -> int
```

Return the group id of the user executing the process.

```
val getegid : unit -> int
```

Return the effective group id under which the process runs.

```
val setgid : int -> unit
```

Set the real group id and effective group id for the process.

```
val getgroups : unit -> int array
```

Return the list of groups to which the user executing the process belongs.

```
type passwd_entry =
  { pw_name : string;
    pw_passwd : string;
    pw_uid : int;
    pw_gid : int;
    pw_gecos : string;
    pw_dir : string;
    pw_shell : string }
```

Structure of entries in the `passwd` database.

```
type group_entry =
  { gr_name : string;
    gr_passwd : string;
    gr_gid : int;
    gr_mem : string array }
```

Structure of entries in the `groups` database.

```
val getlogin : unit -> string
```

Return the login name of the user executing the process.

```
val getpwnam : string -> passwd_entry
```

Find an entry in `passwd` with the given name, or raise `Not_found`.

```
val getgrnam : string -> group_entry
```

Find an entry in `group` with the given name, or raise `Not_found`.

```
val getpwuid : int -> passwd_entry
```

Find an entry in `passwd` with the given user id, or raise `Not_found`.

```
val getgrgid : int -> group_entry
```

Find an entry in `group` with the given group id, or raise `Not_found`.

## Internet addresses

```
type inet_addr
```

The abstract type of Internet addresses.

```
val inet_addr_of_string : string -> inet_addr
val string_of_inet_addr : inet_addr -> string
```

Conversions between string with the format `XXX.YYY.ZZZ.TTT` and Internet addresses. `inet_addr_of_string` raises `Failure` when given a string that does not match this format.

```
val inet_addr_any : inet_addr
```

A special Internet address, for use only with `bind`, representing all the Internet addresses that the host machine possesses.

## Sockets

```
type socket_domain =
    PF_UNIX                             (* Unix domain *)
  | PF_INET                             (* Internet domain *)
```

The type of socket domains.

```
type socket_type =
    SOCK_STREAM                         (* Stream socket *)
  | SOCK_DGRAM                          (* Datagram socket *)
  | SOCK_RAW                            (* Raw socket *)
  | SOCK_SEQPACKET                      (* Sequenced packets socket *)
```

The type of socket kinds, specifying the semantics of communications.

```
type sockaddr =
    ADDR_UNIX of string
  | ADDR_INET of inet_addr * int
```

The type of socket addresses. `ADDR_UNIX name` is a socket address in the Unix domain; `name` is a file name in the file system. `ADDR_INET(addr,port)` is a socket address in the Internet domain; `addr` is the Internet address of the machine, and `port` is the port number.

```
val socket : socket_domain -> socket_type -> int -> file_descr
```

Create a new socket in the given domain, and with the given kind. The third argument is the protocol type; 0 selects the default protocol for that kind of sockets.

```
val socketpair :
        socket_domain -> socket_type -> int -> file_descr * file_descr
```

Create a pair of unnamed sockets, connected together.

```
val accept : file_descr -> file_descr * sockaddr
```

Accept connections on the given socket. The returned descriptor is a socket connected to the client; the returned address is the address of the connecting client.

```
val bind : file_descr -> sockaddr -> unit
```

Bind a socket to an address.

```
val connect : file_descr -> sockaddr -> unit
```

Connect a socket to an address.

```
val listen : file_descr -> int -> unit
```

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

```
type shutdown_command =
    SHUTDOWN_RECEIVE                    (* Close for receiving *)
  | SHUTDOWN_SEND                       (* Close for sending *)
  | SHUTDOWN_ALL                        (* Close both *)
```

The type of commands for `shutdown`.

```
val shutdown : file_descr -> shutdown_command -> unit
```

Shutdown a socket connection. `SHUTDOWN_SEND` as second argument causes reads on the other end of the connection to return an end-of-file condition. `SHUTDOWN_RECEIVE` causes writes on the other end of the connection to return a closed pipe condition (`SIGPIPE` signal).

```
val getsockname : file_descr -> sockaddr
```

Return the address of the given socket.

```
val getpeername : file_descr -> sockaddr
```

Return the address of the host connected to the given socket.

```
type msg_flag =
    MSG_OOB
  | MSG_DONTROUTE
  | MSG_PEEK
```

The flags for `recv`, `recvfrom`, `send` and `sendto`.

```
val recv : file_descr -> string -> int -> int -> msg_flag list -> int
val recvfrom :
        file_descr -> string -> int -> int -> msg_flag list -> int * sockaddr
```

Receive data from an unconnected socket.

```
val send : file_descr -> string -> int -> int -> msg_flag list -> int
val sendto :
        file_descr -> string -> int -> int -> msg_flag list -> sockaddr -> int
```

Send data over an unconnected socket.

```
type socket_option =
    SO_DEBUG               (* Record debugging information *)
  | SO_BROADCAST           (* Permit sending of broadcast messages *)
  | SO_REUSEADDR           (* Allow reuse of local addresses for bind *)
  | SO_KEEPALIVE           (* Keep connection active *)
  | SO_DONTROUTE           (* Bypass the standard routing algorithms *)
  | SO_OOBINLINE           (* Leave out-of-band data in line *)
```

The socket options settable with `setsockopt`.

```
val getsockopt : file_descr -> socket_option -> bool
```

Return the current status of an option in the given socket.

```
val setsockopt : file_descr -> socket_option -> bool -> unit
```

Set or clear an option in the given socket.

## High-level network connection functions

```
val open_connection : sockaddr -> in_channel * out_channel
```

Connect to a server at the given address. Return a pair of buffered channels connected to the server. Remember to call `flush` on the output channel at the right times to ensure correct synchronization.

```
val shutdown_connection : in_channel -> unit
```

"Shut down" a connection established with `open_connection`; that is, transmit an end-of-file condition to the server reading on the other side of the connection.

```
val establish_server : (in_channel -> out_channel -> 'a) -> sockaddr -> unit
```

Establish a server on the given address. The function given as first argument is called for each connection with two buffered channels connected to the client. A new process is created for each connection. The function `establish_server` never returns normally.

## Host and protocol databases

```
type host_entry =
  { h_name : string;
    h_aliases : string array;
    h_addrtype : socket_domain;
    h_addr_list : inet_addr array }
```

Structure of entries in the `hosts` database.

```
type protocol_entry =
  { p_name : string;
    p_aliases : string array;
    p_proto : int }
```

Structure of entries in the `protocols` database.

```
type service_entry =
  { s_name : string;
    s_aliases : string array;
    s_port : int;
    s_proto : string }
```

Structure of entries in the `services` database.

```
val gethostname : unit -> string
```

Return the name of the local host.

```
val gethostbyname : string -> host_entry
```

Find an entry in `hosts` with the given name, or raise `Not_found`.

```
val gethostbyaddr : inet_addr -> host_entry
```

Find an entry in `hosts` with the given address, or raise `Not_found`.

```
val getprotobyname : string -> protocol_entry
```

Find an entry in `protocols` with the given name, or raise `Not_found`.

```
val getprotobynumber : int -> protocol_entry
```

Find an entry in `protocols` with the given protocol number, or raise `Not_found`.

```
val getservbyname : string -> string -> service_entry
```

Find an entry in `services` with the given name, or raise `Not_found`.

```
val getservbyport : int -> string -> service_entry
```

Find an entry in `services` with the given service number, or raise `Not_found`.

## Terminal interface

The following functions implement the POSIX standard terminal interface. They provide control over asynchronous communication ports and pseudo-terminals. Refer to the `termios` man page for a complete description.

```
type terminal_io = {
```

Input modes:

```
    mutable c_ignbrk: bool;  (* Ignore the break condition. *)
    mutable c_brkint: bool;  (* Signal interrupt on break condition. *)
    mutable c_ignpar: bool;  (* Ignore characters with parity errors. *)
    mutable c_parmrk: bool;  (* Mark parity errors. *)
    mutable c_inpck: bool;   (* Enable parity check on input. *)
    mutable c_istrip: bool;  (* Strip 8th bit on input characters. *)
    mutable c_inlcr: bool;   (* Map NL to CR on input. *)
    mutable c_igncr: bool;   (* Ignore CR on input. *)
    mutable c_icrnl: bool;   (* Map CR to NL on input. *)
    mutable c_ixon: bool;    (* Recognize XON/XOFF characters on input. *)
    mutable c_ixoff: bool;   (* Emit XON/XOFF chars to control input flow. *)
```

Output modes:

```
    mutable c_opost: bool;   (* Enable output processing. *)
```

Control modes:

```
    mutable c_obaud: int;    (* Output baud rate (0 means close connection).*)
    mutable c_ibaud: int;    (* Input baud rate. *)
    mutable c_csize: int;    (* Number of bits per character (5-8). *)
    mutable c_cstopb: int;   (* Number of stop bits (1-2). *)
    mutable c_cread: bool;   (* Reception is enabled. *)
    mutable c_parenb: bool;  (* Enable parity generation and detection. *)
    mutable c_parodd: bool;  (* Specify odd parity instead of even. *)
    mutable c_hupcl: bool;   (* Hang up on last close. *)
    mutable c_clocal: bool;  (* Ignore modem status lines. *)
```

Local modes:

```
    mutable c_isig: bool;    (* Generate signal on INTR, QUIT, SUSP. *)
    mutable c_icanon: bool;  (* Enable canonical processing
                                (line buffering and editing) *)
    mutable c_noflsh: bool;  (* Disable flush after INTR, QUIT, SUSP. *)
    mutable c_echo: bool;    (* Echo input characters. *)
    mutable c_echoe: bool;   (* Echo ERASE (to erase previous character). *)
    mutable c_echok: bool;   (* Echo KILL (to erase the current line). *)
    mutable c_echonl: bool;  (* Echo NL even if c_echo is not set. *)
```

Control characters:

```
    mutable c_vintr: char;   (* Interrupt character (usually ctrl-C). *)
    mutable c_vquit: char;   (* Quit character (usually ctrl-\). *)
    mutable c_verase: char;  (* Erase character (usually DEL or ctrl-H). *)
    mutable c_vkill: char;   (* Kill line character (usually ctrl-U). *)
    mutable c_veof: char;    (* End-of-file character (usually ctrl-D). *)
    mutable c_veol: char;    (* Alternate end-of-line char. (usually none). *)
    mutable c_vmin: int;     (* Minimum number of characters to read
                                before the read request is satisfied. *)
    mutable c_vtime: int;    (* Maximum read wait (in 0.1s units). *)
    mutable c_vstart: char;  (* Start character (usually ctrl-Q). *)
    mutable c_vstop: char    (* Stop character (usually ctrl-S). *)
  }
val tcgetattr: file_descr -> terminal_io
```

Return the status of the terminal referred to by the given file descriptor.

```
type setattr_when = TCSANOW | TCSADRAIN | TCSAFLUSH
val tcsetattr: file_descr -> setattr_when -> terminal_io -> unit
```

Set the status of the terminal referred to by the given file descriptor. The second argument indicates when the status change takes place: immediately (`TCSANOW`), when all pending output has been transmitted (`TCSADRAIN`), or after flushing all input that has been received but not read (`TCSAFLUSH`). `TCSADRAIN` is recommended when changing the output parameters; `TCSAFLUSH`, when changing the input parameters.

```
val tcsendbreak: file_descr -> int -> unit
```

Send a break condition on the given file descriptor. The second argument is the duration of the break, in 0.1s units; 0 means standard duration (0.25s).

```
val tcdrain: file_descr -> unit
```

Waits until all output written on the given file descriptor has been transmitted.

```
type flush_queue = TCIFLUSH | TCOFLUSH | TCIOFLUSH
val tcflush: file_descr -> flush_queue -> unit
```

Discard data written on the given file descriptor but not yet transmitted, or data received but not yet read, depending on the second argument: `TCIFLUSH` flushes data received but not read, `TCOFLUSH` flushes data written but not transmitted, and `TCIOFLUSH` flushes both.

```
type flow_action = TCOOFF | TCOON | TCIOFF | TCION
val tcflow: file_descr -> flow_action -> unit
```

Suspend or restart reception or transmission of data on the given file descriptor, depending on the second argument: `TCOOFF` suspends output, `TCOON` restarts output, `TCIOFF` transmits a STOP character to suspend input, and `TCION` transmits a START character to restart input.

```
val setsid : unit -> int
```

Put the calling process in a new session and detach it from its controlling terminal.

**Windows:**

Below is a list of the functions that are not implemented, or only partially implemented, under Windows. Functions not mentioned are fully implemented and behave as described previously in this chapter.

| Functions | Comment |
|---|---|
| `fork` | not implemented, use `create_process` or threads |
| `wait` | not implemented, use `waitpid` |
| `waitpid` | can only wait for a given PID, not any child process |
| `getppid` | not implemented (meaningless under Windows) |
| `nice` | not implemented |
| `truncate`, `ftruncate` | not implemented |
| `lstat`, `fstat` | not implemented |
| `link`, `symlink`, `readlink` | not implemented (no links under Windows) |
| `chmod`, `fchmod` | not implemented |
| `chown`, `fchown` | not implemented (make no sense on a DOS file system) |
| `umask` | not implemented |
| `set_nonblock`, `clear_nonblock` | implemented as dummy functions; use threads instead of non-blocking I/O |
| `rewinddir` | not implemented; re-open the directory instead |
| `mkfifo` | not implemented |
| `select` | implemented, but works only for sockets; use threads if you need to wait on other kinds of file descriptors |
| `lockf` | not implemented |
| `kill`, `pause` | not implemented (no inter-process signals in Windows) |
| `alarm`, `times` | not implemented |
| `getitimer`, `setitimer` | not implemented |
| `getuid`, `getgid` | always return 1 |
| `getgid`, `getegid`, `getgroups` | not implemented |
| `setuid`, `setgid` | not implemented |
| `getpwnam`, `getpwuid` | always raise `Not_found` |
| `getgrnam`, `getgrgid` | always raise `Not_found` |
| type `socket_domain` | the domain `PF_UNIX` is not supported; `PF_INET` is fully supported |
| `establish_server` | not implemented; use threads |
| terminal functions (`tc*`) | not implemented |

# Chapter 19

# The num library: arbitrary-precision rational arithmetic

The `num` library implements exact-precision rational arithmetic. It is built upon the state-of-the-art BigNum arbitrary-precision integer arithmetic package, and therefore achieves very high performance.

The functions provided in this library are fully documented in *The CAML Numbers Reference Manual* by Valérie Ménissier-Morain, technical report 141, INRIA, july 1992 (available by anonymous FTP from `ftp.inria.fr`, directory `INRIA/publications/RT`, file `RT-0141.ps.Z`). A summary of the functions is given below.

Programs that use the `num` library must be linked in "custom runtime" mode, as follows:

```
ocamlc -custom other options nums.cma other files -cclib -lnums
ocamlopt other options nums.cmxa other files -cclib -lnums
```

For interactive use of the `nums` library, do:

```
ocamlmktop -custom -o mytop nums.cma -cclib -lnums
./mytop
```

## 19.1 Module Num: operation on arbitrary-precision numbers

```
open Nat
open Big_int
open Ratio
```

Numbers (type `num`) are arbitrary-precision rational numbers, plus the special elements `1/0` (infinity) and `0/0` (undefined).

```
type num = Int of int | Big_int of big_int | Ratio of ratio
```

The type of numbers.

Arithmetic operations

```
val (+/) : num -> num -> num
val add_num : num -> num -> num
```

Addition

```
val minus_num : num -> num
```

Unary negation.

```
val (-/) : num -> num -> num
val sub_num : num -> num -> num
```

Subtraction

```
val (*/) : num -> num -> num
val mult_num : num -> num -> num
```

Multiplication

```
val square_num : num -> num
```

Squaring

```
val (//) : num -> num -> num
val div_num : num -> num -> num
```

Division

```
val quo_num : num -> num -> num
val mod_num : num -> num -> num
```

Euclidean division: quotient and remainder

```
val (**/) : num -> num -> num
val power_num : num -> num -> num
```

Exponentiation

```
val is_integer_num : num -> bool
```

Test if a number is an integer

```
val integer_num : num -> num
val floor_num : num -> num
val round_num : num -> num
val ceiling_num : num -> num
```

Approximate a number by an integer. `floor_num n` returns the largest integer smaller or equal to `n`. `ceiling_num n` returns the smallest integer bigger or equal to `n`. `integer_num n` returns the integer closest to `n`. In case of ties, rounds towards zero. `round_num n` returns the integer closest to `n`. In case of ties, rounds off zero.

```
val sign_num : num -> int
```

Return `-1`, `0` or `1` according to the sign of the argument.

```
val (=/) : num -> num -> bool
val (</) : num -> num -> bool
val (>/) : num -> num -> bool
val (<=/) : num -> num -> bool
val (>=/) : num -> num -> bool
val (<>/) : num -> num -> bool
val eq_num : num -> num -> bool
val lt_num : num -> num -> bool
val le_num : num -> num -> bool
val gt_num : num -> num -> bool
val ge_num : num -> num -> bool
```

Usual comparisons between numbers

```
val compare_num : num -> num -> int
```

Return `-1`, `0` or `1` if the first argument is less than, equal to, or greater than the second argument.

```
val max_num : num -> num -> num
val min_num : num -> num -> num
```

Return the greater (resp. the smaller) of the two arguments.

```
val abs_num : num -> num
```

Absolute value.

```
val succ_num: num -> num
```

`succ n` is `n+1`

```
val pred_num: num -> num
```

`pred n` is `n-1`

```
val incr_num: num ref -> unit
```

`incr r` is `r:=!r+1`, where `r` is a reference to a number.

```
val decr_num: num ref -> unit
```

`decr r` is `r:=!r-1`, where `r` is a reference to a number.

Coercions with strings

```
val string_of_num : num -> string
```

Convert a number to a string, using fractional notation.

```
val approx_num_fix : int -> num -> string
val approx_num_exp : int -> num -> string
```

Approximate a number by a decimal. The first argument is the required precision. The second argument is the number to approximate. `approx_fix` uses decimal notation; the first argument is the number of digits after the decimal point. `approx_exp` uses scientific (exponential) notation; the first argument is the number of digits in the mantissa.

```
val num_of_string : string -> num
```

Convert a string to a number.

Coercions between numerical types

```
val int_of_num : num -> int
val num_of_int : int -> num
val nat_of_num : num -> nat
val num_of_nat : nat -> num
val num_of_big_int : big_int -> num
val big_int_of_num : num -> big_int
val ratio_of_num : num -> ratio
val num_of_ratio : ratio -> num
val float_of_num : num -> float
```

## 19.2 Module `Arith_status`: flags that control rational arithmetic

```
val arith_status: unit -> unit
```

Print the current status of the arithmetic flags.

```
val get_error_when_null_denominator : unit -> bool
val set_error_when_null_denominator : bool -> unit
```

Get or set the flag `null_denominator`. When on, attempting to create a rational with a null denominator raises an exception. When off, rationals with null denominators are accepted. Initially: on.

```
val get_normalize_ratio : unit -> bool
val set_normalize_ratio : bool -> unit
```

Get or set the flag `normalize_ratio`. When on, rational numbers are normalized after each operation. When off, rational numbers are not normalized until printed. Initially: off.

```
val get_normalize_ratio_when_printing : unit -> bool
val set_normalize_ratio_when_printing : bool -> unit
```

Get or set the flag `normalize_ratio_when_printing`. When on, rational numbers are normalized before being printed. When off, rational numbers are printed as is, without normalization. Initially: on.

```
val get_approx_printing : unit -> bool
val set_approx_printing : bool -> unit
```

Get or set the flag `approx_printing`. When on, rational numbers are printed as a decimal approximation. When off, rational numbers are printed as a fraction. Initially: off.

```
val get_floating_precision : unit -> int
val set_floating_precision : int -> unit
```

Get or set the parameter `floating_precision`. This parameter is the number of digits displayed when `approx_printing` is on. Initially: 12.

# Chapter 20

# The str library: regular expressions and string processing

The `str` library provides high-level string processing functions, some based on regular expressions. It is intended to support the kind of file processing that is usually performed with scripting languages such as `awk`, `perl` or `sed`.

Programs that use the `str` library must be linked in "custom runtime" mode, as follows:

```
ocamlc -custom other options str.cma other files -cclib -lstr
ocamlopt other options str.cmxa other files -cclib -lstr
```

For interactive use of the `str` library, do:

```
ocamlmktop -custom -o mytop str.cma -cclib -lstr
./mytop
```

## 20.1 Module `Str`: regular expressions and high-level string processing

### Regular expressions

`type regexp`

The type of compiled regular expressions.

`val regexp: string -> regexp`

Compile a regular expression. The syntax for regular expressions is the same as in Gnu Emacs. The special characters are `$^.*+?[]`. The following constructs are recognized:

| | |
|---|---|
| `.` | matches any character except newline |
| `*` | (postfix) matches the previous expression zero, one or several times |
| `+` | (postfix) matches the previous expression one or several times |
| `?` | (postfix) matches the previous expression once or not at all |

| | |
|---|---|
| `[..]` | character set; ranges are denoted with `-`, as in `[a-z]`; an initial `^`, as in `[^0-9]`, complements the set |
| `^` | matches at beginning of line |
| `$` | matches at end of line |
| `\|` | (infix) alternative between two expressions |
| `\(..\)` | grouping and naming of the enclosed expression |
| `\1` | the text matched by the first `\(...\)` expression (`\2` for the second expression, etc) |
| `\b` | matches word boundaries |
| `\` | quotes special characters. |

```
val regexp_case_fold: string -> regexp
```

Same as `regexp`, but the compiled expression will match text in a case-insensitive way: uppercase and lowercase letters will be considered equivalent.

```
val quote: string -> string
```

`Str.quote s` returns a regexp string that matches exactly `s` and nothing else.

```
val regexp_string: string -> regexp
val regexp_string_case_fold: string -> regexp
```

`Str.regexp_string s` returns a regular expression that matches exactly `s` and nothing else. `Str.regexp_string_case_fold` is similar, but the regexp matches in a case-insensitive way.

## String matching and searching

```
val string_match: regexp -> string -> int -> bool
```

`string_match r s start` tests whether the characters in `s` starting at position `start` match the regular expression `r`. The first character of a string has position `0`, as usual.

```
val search_forward: regexp -> string -> int -> int
```

`search_forward r s start` searchs the string `s` for a substring matching the regular expression `r`. The search starts at position `start` and proceeds towards the end of the string. Return the position of the first character of the matched substring, or raise `Not_found` if no substring matches.

```
val search_backward: regexp -> string -> int -> int
```

Same as `search_forward`, but the search proceeds towards the beginning of the string.

```
val string_partial_match: regexp -> string -> int -> bool
```

Similar to `string_match`, but succeeds whenever the argument string is a prefix of a string that matches. This includes the case of a true complete match.

```
val matched_string: string -> string
```

`matched_string s` returns the substring of `s` that was matched by the latest `string_match`, `search_forward` or `search_backward`. The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

```
val match_beginning: unit -> int
val match_end: unit -> int
```

`match_beginning()` returns the position of the first character of the substring that was matched by `string_match`, `search_forward` or `search_backward`. `match_end()` returns the position of the character following the last character of the matched substring.

```
val matched_group: int -> string -> string
```

`matched_group n s` returns the substring of `s` that was matched by the `n`th group `\(...\)` of the regular expression during the latest `string_match`, `search_forward` or `search_backward`. The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

```
val group_beginning: int -> int
val group_end: int -> int
```

`group_beginning n` returns the position of the first character of the substring that was matched by the `n`th group of the regular expression. `group_end n` returns the position of the character following the last character of the matched substring.

## Replacement

```
val global_replace: regexp -> string -> string -> string
```

`global_replace regexp repl s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by `repl`. The replacement text `repl` can contain `\1`, `\2`, etc; these sequences will be replaced by the text matched by the corresponding group in the regular expression. `\0` stands for the text matched by the whole regular expression.

```
val replace_first: regexp -> string -> string -> string
```

Same as `global_replace`, except that only the first substring matching the regular expression is replaced.

```
val global_substitute: regexp -> (string -> string) -> string -> string
```

`global_substitute regexp subst s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by the result of function `subst`. The function `subst` is called once for each matching substring, and receives `s` (the whole text) as argument.

```
val substitute_first: regexp -> (string -> string) -> string -> string
```

Same as `global_substitute`, except that only the first substring matching the regular expression is replaced.

```
val replace_matched : string -> string -> string
```

`replace_matched repl s` returns the replacement text `repl` in which `\1`, `\2`, etc. have been replaced by the text matched by the corresponding groups in the most recent matching operation. `s` must be the same string that was matched during this matching operation.

## Splitting

```
val split: regexp -> string -> string list
```

`split r s` splits `s` into substrings, taking as delimiters the substrings that match `r`, and returns the list of substrings. For instance, `split (regexp "[ \t]+") s` splits `s` into blank-separated words. An occurrence of the delimiter at the beginning and at the end of the string is ignored.

```
val bounded_split: regexp -> string -> int -> string list
```

Same as `split`, but splits into at most `n` substrings, where `n` is the extra integer parameter.

```
val split_delim: regexp -> string -> string list
val bounded_split_delim: regexp -> string -> int -> string list
```

Same as `split` and `bounded_split`, but occurrences of the delimiter at the beginning and at the end of the string are recognized and returned as empty strings in the result. For instance, `split_delim (regexp " ") " abc "` returns `[""; "abc"; ""]`, while `split` with the same arguments returns `["abc"]`.

```
type split_result = Text of string | Delim of string
val full_split: regexp -> string -> split_result list
val bounded_full_split: regexp -> string -> int -> split_result list
```

Same as `split_delim` and `bounded_split_delim`, but returns the delimiters as well as the substrings contained between delimiters. The former are tagged `Delim` in the result list; the latter are tagged `Text`. For instance, `full_split (regexp "[{}]") "{ab}"` returns `[Delim "{"; Text "ab"; Delim "}"]`.

## Extracting substrings

```
val string_before: string -> int -> string
```

`string_before s n` returns the substring of all characters of `s` that precede position `n` (excluding the character at position `n`).

`val string_after: string -> int -> string`

> `string_after s n` returns the substring of all characters of `s` that follow position `n` (including the character at position `n`).

`val first_chars: string -> int -> string`

> `first_chars s n` returns the first `n` characters of `s`. This is the same function as `string_before`.

`val last_chars: string -> int -> string`

> `last_chars s n` returns the last `n` characters of `s`.

# Chapter 21

# The threads library

The threads library allows concurrent programming in Objective Caml. It provides multiple threads of control (also called lightweight processes) that execute concurrently in the same memory space. Threads communicate by in-place modification of shared data structures, or by sending and receiving data on communication channels.

The threads library is implemented by time-sharing on a single processor. It will not take advantage of multi-processor machines. Using this library will therefore never make programs run faster. However, many programs are easier to write when structured as several communicating processes.

**Unix:**

Programs that use the threads library must be linked as follows:

```
ocamlc -thread -custom other options threads.cma other files -cclib -lthreads
```

The -thread option selects a special, thread-safe version of the standard library (see chapter 7). The -thread option must also be given when compiling any source file that references modules from the thread library (Thread, Mutex, ...).

The default thread implementation cannot be used in native-code programs compiled with ocamlopt. If your operating system provides POSIX 1003.1c compliant threads, you can select an alternate implementation when configuring Objective Caml (use the -with-pthread option to configure) which also supports native-code programs. Programs that use this alternate implementation of the threads library must be linked as follows:

```
ocamlc -thread -custom other options threads.cma other files \
         -cclib -lthreads -cclib -lunix -cclib -lpthread
ocamlopt -thread other options threads.cmxa other files \
        -cclib -lthreadsnat -cclib -lunix -cclib -lpthread
```

Depending on the operating system, extra system libraries can be necessary. For instance, under Solaris 2.5, add -cclib -lposix4 at the end of the command line.

**Windows:**

Programs that use the threads library must be linked as follows:

```
ocamlc -thread -custom other options threads.cma other files -cclib -lthreads
```

All object files on the command line must also have been compiled with the `-thread` option, which selects a special, thread-safe version of the standard library (see chapter 7).

## 21.1 Module `Thread`: lightweight threads

`type t`

The type of thread handles.

### Thread creation and termination

`val create : ('a -> 'b) -> 'a -> t`

`Thread.create funct arg` creates a new thread of control, in which the function application `funct arg` is executed concurrently with the other threads of the program. The application of `Thread.create` returns the handle of the newly created thread. The new thread terminates when the application `funct arg` returns, either normally or by raising an uncaught exception. In the latter case, the exception is printed on standard error, but not propagated back to the parent thread. Similarly, the result of the application `funct arg` is discarded and not directly accessible to the parent thread.

`val self : unit -> t`

Return the thread currently executing.

`val id : t -> int`

Return the identifier of the given thread. A thread identifier is an integer that identifies uniquely the thread. It can be used to build data structures indexed by threads.

`val exit : unit -> unit`

Terminate prematurely the currently executing thread.

`val kill : t -> unit`

Terminate prematurely the thread whose handle is given. This functionality is available only with bytecode-level threads.

## Suspending threads

```
val delay: float -> unit
```

delay d suspends the execution of the calling thread for d seconds. The other program threads continue to run during this time.

```
val join : t -> unit
```

join th suspends the execution of the calling thread until the thread th has terminated.

```
val wait_read : Unix.file_descr -> unit
val wait_write : Unix.file_descr -> unit
```

Suspend the execution of the calling thread until at least one character is available for reading (wait_read) or one character can be written without blocking (wait_write) on the given Unix file descriptor.

```
val wait_timed_read : Unix.file_descr -> float -> bool
val wait_timed_write : Unix.file_descr -> float -> bool
```

Same as wait_read and wait_write, but wait for at most the amount of time given as second argument (in seconds). Return true if the file descriptor is ready for input/output and false if the timeout expired.

```
val 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
```

Suspend the execution of the calling thead until input/output becomes possible on the given Unix file descriptors. The arguments and results have the same meaning as for Unix.select.

```
val wait_pid : int -> int * Unix.process_status
```

wait_pid p suspends the execution of the calling thread until the Unix process specified by the process identifier p terminates. A pid p of -1 means wait for any child. A pid of 0 means wait for any child in the same process group as the current process. Negative pid arguments represent process groups. Returns the pid of the child caught and its termination status, as per Unix.wait.

```
val wait_signal : int list -> int
```

wait_signal sigs suspends the execution of the calling thread until the process receives one of the signals specified in the list sigs. It then returns the number of the signal received. Signal handlers attached to the signals in sigs will not be invoked. Do not call wait_signal concurrently from several threads on the same signals.

## 21.2 Module Mutex: locks for mutual exclusion

Mutexes (mutual-exclusion locks) are used to implement critical sections and protect shared mutable data structures against concurrent accesses. The typical use is (if `m` is the mutex associated with the data structure `D`):

```
Mutex.lock m;
(* Critical section that operates over D *);
Mutex.unlock m
```

`type t`

The type of mutexes.

`val create: unit -> t`

Return a new mutex.

`val lock: t -> unit`

Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.

`val try_lock: t -> bool`

Same as `try_lock`, but does not suspend the calling thread if the mutex is already locked: just return `false` immediately in that case. If the mutex is unlocked, lock it and return `true`.

`val unlock: t -> unit`

Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.

## 21.3 Module Condition: condition variables to synchronize between threads

Condition variables are used when one thread wants to wait until another thread has finished doing something: the former thread “waits” on the condition variable, the latter thread “signals” the condition when it is done. Condition variables should always be protected by a mutex. The typical use is (if `D` is a shared data structure, `m` its mutex, and `c` is a condition variable):

```
Mutex.lock m;
while (* some predicate P over D is not satisfied *) do
  Condition.wait c m
done;
(* Modify D *)
if (* the predicate P over D is now satified *) then Condition.signal c;
Mutex.unlock m
```

`type t`

The type of condition variables.

`val create: unit -> t`

Return a new condition variable.

`val wait: t -> Mutex.t -> unit`

`wait c m` atomically unlocks the mutex `m` and suspends the calling process on the condition variable `c`. The process will restart after the condition variable `c` has been signalled. The mutex `m` is locked again before `wait` returns.

`val signal: t -> unit`

`signal c` restarts one of the processes waiting on the condition variable `c`.

`val broadcast: t -> unit`

`broadcast c` restarts all processes waiting on the condition variable `c`.

## 21.4 Module `Event`: first-class synchronous communication

This module implements synchronous interprocess communications over channels. As in John Reppy's Concurrent ML system, the communication events are first-class values: they can be built and combined independently before being offered for communication.

`type 'a channel`

The type of communication channels carrying values of type `'a`.

`val new_channel: unit -> 'a channel`

Return a new channel.

`type 'a event`

The type of communication events returning a result of type `'a`.

`val send: 'a channel -> 'a -> unit event`

`send ch v` returns the event consisting in sending the value `v` over the channel `ch`. The result value of this event is `()`.

`val receive: 'a channel -> 'a event`

`receive ch` returns the event consisting in receiving a value from the channel `ch`. The result value of this event is the value received.

```
val always: 'a -> 'a event
```

`always v` returns an event that is always ready for synchronization. The result value of this event is `v`.

```
val choose: 'a event list -> 'a event
```

`choose evl` returns the event that is the alternative of all the events in the list `evl`.

```
val wrap: 'a event -> ('a -> 'b) -> 'b event
```

`wrap ev fn` returns the event that performs the same communications as `ev`, then applies the post-processing function `fn` on the return value.

```
val guard: (unit -> 'a event) -> 'a event
```

`guard fn` returns the event that, when synchronized, computes `fn()` and behaves as the resulting event. This allows to compute events with side-effects at the time of the synchronization operation.

```
val sync: 'a event -> 'a
```

"Synchronize" on an event: offer all the communication possibilities specified in the event to the outside world, and block until one of the communications succeed. The result value of that communication is returned.

```
val select: 'a event list -> 'a
```

"Synchronize" on an alternative of events. `select evl` is shorthand for `sync(choose evl)`.

```
val poll: 'a event -> 'a option
```

Non-blocking version of `sync`: offer all the communication possibilities specified in the event to the outside world, and if one can take place immediately, perform it and return `Some r` where `r` is the result value of that communication. Otherwise, return `None` without blocking.

## 21.5 Module `ThreadUnix`: thread-compatible system calls

This module reimplements some of the functions from `Unix` so that they only block the calling thread, not all threads in the program, if they cannot complete immediately. See the documentation of the `Unix` module for more precise descriptions of the functions below.

### Process handling

```
val execv : string -> string array -> unit
val execve : string -> string array -> string array -> unit
val execvp : string -> string array -> unit
val wait : unit -> int * Unix.process_status
val waitpid : Unix.wait_flag list -> int -> int * Unix.process_status
val system : string -> Unix.process_status
```

## Basic input/output

```
val read : Unix.file_descr -> string -> int -> int -> int
val write : Unix.file_descr -> string -> int -> int -> int
```

## Input/output with timeout

```
val timed_read : Unix.file_descr -> string -> int -> int -> float -> int
val timed_write : Unix.file_descr -> string -> int -> int -> float -> int
```

Behave as `read` and `write`, except that `Unix_error(ETIMEDOUT,_,_)` is raised if no data is available for reading or ready for writing after `d` seconds. The delay `d` is given in the fifth argument, in seconds.

## Polling

```
val 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
```

## Pipes and redirections

```
val pipe : unit -> Unix.file_descr * Unix.file_descr
val open_process_out: string -> out_channel
val open_process: string -> in_channel * out_channel
```

## Time

```
val sleep : int -> unit
```

## Sockets

```
val socket : Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr
val socketpair : Unix.socket_domain -> Unix.socket_type -> int ->
                 Unix.file_descr * Unix.file_descr
val accept : Unix.file_descr -> Unix.file_descr * Unix.sockaddr
val connect : Unix.file_descr -> Unix.sockaddr -> unit
val recv : Unix.file_descr -> string -> int -> int -> Unix.msg_flag list -> int
val recvfrom : Unix.file_descr -> string -> int -> int ->
               Unix.msg_flag list -> int * Unix.sockaddr
val send : Unix.file_descr -> string -> int -> int ->
           Unix.msg_flag list -> int
val sendto : Unix.file_descr -> string -> int -> int ->
             Unix.msg_flag list -> Unix.sockaddr -> int
val open_connection : Unix.sockaddr -> in_channel * out_channel
val establish_server :
      (in_channel -> out_channel -> 'a) -> Unix.sockaddr -> unit
```

# Chapter 22

# The graphics library

The `graphics` library provides a set of portable drawing primitives. Drawing takes place in a separate window that is created when `open_graph` is called.

**Unix:**

This library is implemented under the X11 windows system. Programs that use the `graphics` library must be linked as follows:

```
        ocamlc -custom other options graphics.cma other files \
               -cclib -lgraphics -cclib -lX11
```

For interactive use of the `graphics` library, do:

```
        ocamlmktop -custom -o mytop graphics.cma -cclib -lgraphics \
                   -cclib -lX11
        ./mytop
```

Here are the graphics mode specifications supported by `open_graph` on the X11 implementation of this library: the argument to `open_graph` has the format `"`*display-name geometry*`"`, where *display-name* is the name of the X-windows display to connect to, and *geometry* is a standard X-windows geometry specification. The two components are separated by a space. Either can be omitted, or both. Examples:

`open_graph "foo:0"`
: connects to the display `foo:0` and creates a window with the default geometry

`open_graph "foo:0 300x100+50-0"`
: connects to the display `foo:0` and creates a window 300 pixels wide by 100 pixels tall, at location $(50, 0)$

`open_graph " 300x100+50-0"`
: connects to the default display and creates a window 300 pixels wide by 100 pixels tall, at location $(50, 0)$

`open_graph ""`
: connects to the default display and creates a window with the default geometry.

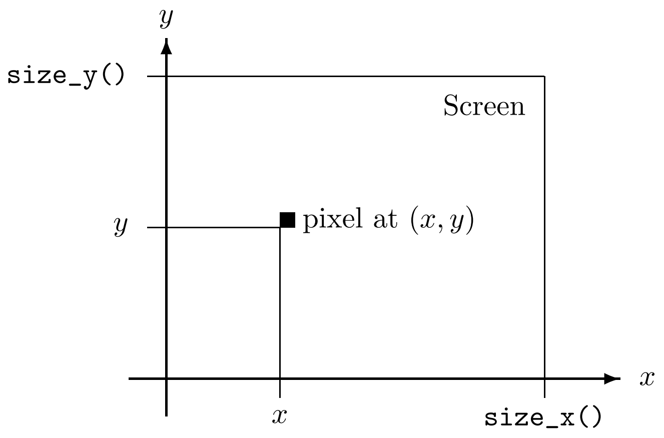
**Windows:**

This library is available only under the toplevel application `ocamlwin.exe`. Before using it, the Caml part of this library must be loaded in-core, either by typing

```
        #load "graphics.cmo";;
```

in the input windows or by using the “Load” entry of the “File” menu.

The screen coordinates are interpreted as shown in the figure below. Notice that the coordinate system used is the same as in mathematics: $y$ increases from the bottom of the screen to the top of the screen, and angles are measured counterclockwise (in degrees). Drawing is clipped to the screen.



## 22.1 Module `Graphics`: machine-independent graphics primitives

`exception Graphic_failure of string`

Raised by the functions below when they encounter an error.

### Initializations

`val open_graph: string -> unit`

Show the graphics window or switch the screen to graphic mode. The graphics window is cleared and the current point is set to (0, 0). The string argument is used to pass optional information on the desired graphics mode, the graphics window size, and so on. Its interpretation is implementation-dependent. If the empty string is given, a sensible default is selected.

`val close_graph: unit -> unit`

Delete the graphics window or switch the screen back to text mode.

`val clear_graph : unit -> unit`

Erase the graphics window.

```
val size_x : unit -> int
val size_y : unit -> int
```

Return the size of the graphics window. Coordinates of the screen pixels range over `0 .. size_x()-1` and `0 .. size_y()-1`. Drawings outside of this rectangle are clipped, without causing an error. The origin (0,0) is at the lower left corner.

## Colors

```
type color = int
```

A color is specified by its R, G, B components. Each component is in the range `0..255`. The three components are packed in an `int`: `0xRRGGBB`, where `RR` are the two hexadecimal digits for the red component, `GG` for the green component, `BB` for the blue component.

```
val rgb: int -> int -> int -> color
```

`rgb r g b` returns the integer encoding the color with red component `r`, green component `g`, and blue component `b`. `r`, `g` and `b` are in the range `0..255`.

```
val set_color : color -> unit
```

Set the current drawing color.

```
val black : color
val white : color
val red : color
val green : color
val blue : color
val yellow : color
val cyan : color
val magenta : color
```

Some predefined colors.

```
val background: color
val foreground: color
```

Default background and foreground colors (usually, either black foreground on a white background or white foreground on a black background). `clear_graph` fills the screen with the `background` color. The initial drawing color is `foreground`.

### Point and line drawing

`val plot : int -> int -> unit`

Plot the given point with the current drawing color.

`val point_color : int -> int -> color`

Return the color of the given point.

`val moveto : int -> int -> unit`

Position the current point.

`val current_point : unit -> int * int`

Return the position of the current point.

`val lineto : int -> int -> unit`

Draw a line with endpoints the current point and the given point, and move the current point to the given point.

`val draw_arc : int -> int -> int -> int -> int -> int -> unit`

`draw_arc x y rx ry a1 a2` draws an elliptical arc with center `x,y`, horizontal radius `rx`, vertical radius `ry`, from angle `a1` to angle `a2` (in degrees). The current point is unchanged.

`val draw_ellipse : int -> int -> int -> int -> unit`

`draw_ellipse x y rx ry` draws an ellipse with center `x,y`, horizontal radius `rx` and vertical radius `ry`. The current point is unchanged.

`val draw_circle : int -> int -> int -> unit`

`draw_circle x y r` draws a circle with center `x,y` and radius `r`. The current point is unchanged.

`val set_line_width : int -> unit`

Set the width of points and lines drawn with the functions above. Under X Windows, `set_line_width 0` selects a width of 1 pixel and a faster, but less precise drawing algorithm than the one used when `set_line_width 1` is specified.

## Text drawing

```
val draw_char : char -> unit
val draw_string : string -> unit
```

Draw a character or a character string with lower left corner at current position. After drawing, the current position is set to the lower right corner of the text drawn.

```
val set_font : string -> unit
val set_text_size : int -> unit
```

Set the font and character size used for drawing text. The interpretation of the arguments to `set_font` and `set_text_size` is implementation-dependent.

```
val text_size : string -> int * int
```

Return the dimensions of the given text, if it were drawn with the current font and size.

## Filling

```
val fill_rect : int -> int -> int -> int -> unit
```

`fill_rect x y w h` fills the rectangle with lower left corner at `x,y`, width `w` and height `h`, with the current color.

```
val fill_poly : (int * int) array -> unit
```

Fill the given polygon with the current color. The array contains the coordinates of the vertices of the polygon.

```
val fill_arc : int -> int -> int -> int -> int -> int -> unit
```

Fill an elliptical pie slice with the current color. The parameters are the same as for `draw_arc`.

```
val fill_ellipse : int -> int -> int -> int -> unit
```

Fill an ellipse with the current color. The parameters are the same as for `draw_ellipse`.

```
val fill_circle : int -> int -> int -> unit
```

Fill a circle with the current color. The parameters are the same as for `draw_circle`.

## Images

`type image`

The abstract type for images, in internal representation. Externally, images are represented as matrices of colors.

`val transp : color`

In matrices of colors, this color represent a “transparent” point: when drawing the corresponding image, all pixels on the screen corresponding to a transparent pixel in the image will not be modified, while other points will be set to the color of the corresponding point in the image. This allows superimposing an image over an existing background.

`val make_image : color array array -> image`

Convert the given color matrix to an image. Each sub-array represents one horizontal line. All sub-arrays must have the same length; otherwise, exception `Graphic_failure` is raised.

`val dump_image : image -> color array array`

Convert an image to a color matrix.

`val draw_image : image -> int -> int -> unit`

Draw the given image with lower left corner at the given point.

`val get_image : int -> int -> int -> int -> image`

Capture the contents of a rectangle on the screen as an image. The parameters are the same as for `fill_rect`.

`val create_image : int -> int -> image`

`create_image w h` returns a new image `w` pixels wide and `h` pixels tall, to be used in conjunction with `blit_image`. The initial image contents are random, except that no point is transparent.

`val blit_image : image -> int -> int -> unit`

`blit_image img x y` copies screen pixels into the image `img`, modifying `img` in-place. The pixels copied are those inside the rectangle with lower left corner at `x,y`, and width and height equal to those of the image. Pixels that were transparent in `img` are left unchanged.

## Mouse and keyboard events

```
type status =
  { mouse_x : int;              (* X coordinate of the mouse *)
    mouse_y : int;              (* Y coordinate of the mouse *)
    button : bool;              (* true if a mouse button is pressed *)
    keypressed : bool;          (* true if a key has been pressed *)
    key : char }                (* the character for the key pressed *)
```

To report events.

```
type event =
    Button_down                 (* A mouse button is pressed *)
  | Button_up                   (* A mouse button is released *)
  | Key_pressed                 (* A key is pressed *)
  | Mouse_motion                (* The mouse is moved *)
  | Poll                        (* Don't wait; return immediately *)
```

To specify events to wait for.

`val wait_next_event : event list -> status`

Wait until one of the events specified in the given event list occurs, and return the status of the mouse and keyboard at that time. If `Poll` is given in the event list, return immediately with the current status. If the mouse cursor is outside of the graphics window, the `mouse_x` and `mouse_y` fields of the event are outside the range `0..size_x()-1, 0..size_y()-1`. Keypresses are queued, and dequeued one by one when the `Key_pressed` event is specified.

## Mouse and keyboard polling

`val mouse_pos : unit -> int * int`

Return the position of the mouse cursor, relative to the graphics window. If the mouse cursor is outside of the graphics window, `mouse_pos()` returns a point outside of the range `0..size_x()-1, 0..size_y()-1`.

`val button_down : unit -> bool`

Return `true` if the mouse button is pressed, `false` otherwise.

`val read_key : unit -> char`

Wait for a key to be pressed, and return the corresponding character. Keypresses are queued.

`val key_pressed : unit -> bool`

Return `true` if a keypress is available; that is, if `read_key` would not block.

## Sound

`val sound : int -> int -> unit`

`sound freq dur` plays a sound at frequency `freq` (in hertz) for a duration `dur` (in milliseconds).

# Chapter 23

# The dbm library: access to NDBM databases

The `dbm` library provides access to NDBM databases under Unix. NDBM databases maintain key/data associations, where both the key and the data are arbitrary strings. They support fairly large databases (several gigabytes) and can retrieve a keyed item in one or two file system accesses. Refer to the Unix manual pages for more information.

**Unix:**

Programs that use the `dbm` library must be linked in "custom runtime" mode, as follows:

```
ocamlc -custom other options dbm.cma other files -cclib -lmldbm -cclib -lndbm
ocamlopt other options dbm.cmxa other files -cclib -lmldbm -cclib -lndbm
```

For interactive use of the `dbm` library, do:

```
ocamlmktop -custom -o mytop dbm.cma -cclib -lmldbm -cclib -lndbm
./mytop
```

Depending on the Unix system used, the `-cclib -lndbm` option is not always necessary, or the library may have another name than `-lndbm`.

**Windows:**

This library is not available.

## 23.1 Module `Dbm`: interface to the NDBM databases

```
type t
```

The type of file descriptors opened on NDBM databases.

```
type open_flag =
  Dbm_rdonly | Dbm_wronly | Dbm_rdwr | Dbm_create
```

Flags for opening a database (see `opendbm`).

`exception Dbm_error of string`

Raised by the following functions when an error is encountered.

`val opendbm : string -> open_flag list -> int -> t`

Open a descriptor on an NDBM database. The first argument is the name of the database (without the `.dir` and `.pag` suffixes). The second argument is a list of flags: `Dbm_rdonly` opens the database for reading only, `Dbm_wronly` for writing only, `Dbm_rdwr` for reading and writing; `Dbm_create` causes the database to be created if it does not already exist. The third argument is the permissions to give to the database files, if the database is created.

`val close : t -> unit`

Close the given descriptor.

`val find : t -> string -> string`

`find db key` returns the data associated with the given `key` in the database opened for the descriptor `db`. Raise `Not_found` if the `key` has no associated data.

`val add : t -> string -> string -> unit`

`add db key data` inserts the pair (`key`, `data`) in the database `db`. If the database already contains data associated with `key`, raise `Dbm_error "Entry already exists"`.

`val replace : t -> string -> string -> unit`

`replace db key data` inserts the pair (`key`, `data`) in the database `db`. If the database already contains data associated with `key`, that data is discarded and silently replaced by the new `data`.

`val remove : t -> string -> unit`

`remove db key data` removes the data associated with `key` in `db`. If `key` has no associated data, raise `Dbm_error "dbm_delete"`.

```
val firstkey : t -> string
val nextkey : t -> string
```

Enumerate all keys in the given database, in an unspecified order. `firstkey db` returns the first key, and repeated calls to `nextkey db` return the remaining keys. `Not_found` is raised when all keys have been enumerated.

`val iter : (string -> string -> 'a) -> t -> unit`

`iter f db` applies `f` to each (`key`, `data`) pair in the database `db`. `f` receives `key` as first argument and `data` as second argument.

# Chapter 24

# The dynlink library: dynamic loading and linking of object files

The `dynlink` library supports type-safe dynamic loading and linking of bytecode object files (`.cmo` and `.cma` files) in a running bytecode program. Type safety is ensured by limiting the set of modules from the running program that the loaded object file can access, and checking that the running program and the loaded object file have been compiled against the same interfaces for these modules.

Programs that use the `dynlink` library simply need to link `dynlink.cma` with their object files and other libraries. Linking in "custom runtime" mode is not necessary. Dynamic linking is available only to bytecode programs compiled with `ocamlc`, not to native-code programs compiled with `ocamlopt`.

## 24.1 Module `Dynlink`: dynamic loading of bytecode object files

```
val init : unit -> unit
```

Initialize the library. Must be called before `loadfile`.

```
val loadfile : string -> unit
```

Load the given bytecode object file and link it. All toplevel expressions in the loaded compilation unit are evaluated. No facilities are provided to access value names defined by the unit. Therefore, the unit must register itself its entry points with the main program, e.g. by modifying tables of functions.

```
val loadfile_private : string -> unit
```

Same as `loadfile`, except that the module loaded is not made available to other modules dynamically loaded afterwards.

```
val add_interfaces : string list -> string list -> unit
```

`add_interfaces units path` grants dynamically-linked object files access to the compilation units named in list `units`. The interfaces (`.cmi` files) for these units are searched in `path` (a list of directory names). Initially, dynamically-linked object files do not have access to any of the compilation units composing the running program, not even the standard library. `add_interfaces` or `add_available_units` (see below) must be called to grant access to some of the units.

```
val add_available_units : (string * Digest.t) list -> unit
```

Same as `add_interfaces`, but instead of searching `.cmi` files to find the unit interfaces, uses the interface digests given for each unit. This way, the `.cmi` interface files need not be available at run-time. The digests can be extracted from `.cmi` files using the `extract_crc` program installed in the Objective Caml standard library directory.

```
val clear_available_units : unit -> unit
```

Clear the list of compilation units accessible to dynamically-linked programs.

```
val allow_unsafe_modules : bool -> unit
```

Govern whether unsafe object files are allowed to be dynamically linked. A compilation unit is "unsafe" if it contains declarations of external functions, which can break type safety. By default, dynamic linking of unsafe object files is not allowed.

```
type linking_error =
    Undefined_global of string
  | Unavailable_primitive of string
type error =
    Not_a_bytecode_file of string
  | Inconsistent_import of string
  | Unavailable_unit of string
  | Unsafe_file
  | Linking_error of string * linking_error
  | Corrupted_interface of string
exception Error of error
```

Errors in dynamic linking are reported by raising the `Error` exception with a description of the error.

```
val error_message: error -> string
```

Convert an error description to a printable message.

# Part V

# Appendix

## Index to the library

## Index of keywords