# Contents

1 **An introduction to OCaml**

1.1 Basics .......................................................... 11
1.2 Data types ..................................................... 12
1.3 Functions as values ......................................... 13
1.4 Records and variants ....................................... 14
1.5 Imperative features ......................................... 16
1.6 Exceptions .................................................... 18
1.7 Symbolic processing of expressions ..................... 19
1.8 Pretty-printing and parsing ................................. 20
1.9 Standalone Caml programs ................................. 23

2 **The module system**

2.1 Structures ..................................................... 25
2.2 Signatures ..................................................... 26
2.3 Functors ........................................................ 27
2.4 Functors and type abstraction ............................. 29
2.5 Modules and separate compilation ....................... 31

3 **Objects in Caml**

3.1 Classes and objects ......................................... 33
3.2 Immediate objects ........................................... 36
3.3 Reference to self ............................................ 37
3.4 Initializers .................................................... 38
3.5 Virtual methods ............................................. 38
3.6 Private methods ............................................. 40
3.7 Class interfaces ............................................. 42
3.8 Inheritance .................................................... 43
3.9 Multiple inheritance ....................................... 44
3.10 Parameterized classes ...................................... 44
3.11 Polymorphic methods ...................................... 47
3.12 Using coercions ............................................. 50
3.13 Functional objects ........................................ 54
3.14 Cloning objects ............................................. 55
3.15 Recursive classes .......................................... 58
<table>
<thead>
<tr>
<th>Section</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>3.16</td>
<td>Binary methods</td>
<td>58</td>
</tr>
<tr>
<td>3.17</td>
<td>Friends</td>
<td>60</td>
</tr>
<tr>
<td>4</td>
<td>Labels and variants</td>
<td>63</td>
</tr>
<tr>
<td>4.1</td>
<td>Labels</td>
<td>63</td>
</tr>
<tr>
<td>4.2</td>
<td>Polymorphic variants</td>
<td>69</td>
</tr>
<tr>
<td>5</td>
<td>Advanced examples with classes and modules</td>
<td>73</td>
</tr>
<tr>
<td>5.1</td>
<td>Extended example: bank accounts</td>
<td>73</td>
</tr>
<tr>
<td>5.2</td>
<td>Simple modules as classes</td>
<td>79</td>
</tr>
<tr>
<td>5.3</td>
<td>The subject/observer pattern</td>
<td>85</td>
</tr>
<tr>
<td>11</td>
<td>The OCaml language</td>
<td>89</td>
</tr>
<tr>
<td>6</td>
<td>The OCaml language</td>
<td>91</td>
</tr>
<tr>
<td>6.1</td>
<td>Lexical conventions</td>
<td>91</td>
</tr>
<tr>
<td>6.2</td>
<td>Values</td>
<td>95</td>
</tr>
<tr>
<td>6.3</td>
<td>Names</td>
<td>97</td>
</tr>
<tr>
<td>6.4</td>
<td>Type expressions</td>
<td>100</td>
</tr>
<tr>
<td>6.5</td>
<td>Constants</td>
<td>103</td>
</tr>
<tr>
<td>6.6</td>
<td>Patterns</td>
<td>103</td>
</tr>
<tr>
<td>6.7</td>
<td>Expressions</td>
<td>106</td>
</tr>
<tr>
<td>6.8</td>
<td>Type and exception definitions</td>
<td>117</td>
</tr>
<tr>
<td>6.9</td>
<td>Classes</td>
<td>120</td>
</tr>
<tr>
<td>6.10</td>
<td>Module types (module specifications)</td>
<td>126</td>
</tr>
<tr>
<td>6.11</td>
<td>Module expressions (module implementations)</td>
<td>130</td>
</tr>
<tr>
<td>6.12</td>
<td>Compilation units</td>
<td>133</td>
</tr>
<tr>
<td>7</td>
<td>Language extensions</td>
<td>135</td>
</tr>
<tr>
<td>7.1</td>
<td>Integer literals for types int32, int64 and nativeint</td>
<td>135</td>
</tr>
<tr>
<td>7.2</td>
<td>Streams and stream parsers</td>
<td>135</td>
</tr>
<tr>
<td>7.3</td>
<td>Recursive definitions of values</td>
<td>135</td>
</tr>
<tr>
<td>7.4</td>
<td>Range patterns</td>
<td>137</td>
</tr>
<tr>
<td>7.5</td>
<td>Assertion checking</td>
<td>137</td>
</tr>
<tr>
<td>7.6</td>
<td>Lazy evaluation</td>
<td>137</td>
</tr>
<tr>
<td>7.7</td>
<td>Local modules</td>
<td>138</td>
</tr>
<tr>
<td>7.8</td>
<td>Recursive modules</td>
<td>138</td>
</tr>
<tr>
<td>7.9</td>
<td>Private types</td>
<td>139</td>
</tr>
<tr>
<td>7.10</td>
<td>Local opens</td>
<td>142</td>
</tr>
<tr>
<td>7.11</td>
<td>Record notations</td>
<td>142</td>
</tr>
<tr>
<td>7.12</td>
<td>Explicit polymorphic type annotations</td>
<td>143</td>
</tr>
<tr>
<td>7.13</td>
<td>Explicit naming of type variables</td>
<td>143</td>
</tr>
<tr>
<td>7.14</td>
<td>First-class modules</td>
<td>144</td>
</tr>
<tr>
<td>7.15</td>
<td>Recovering the type of a module</td>
<td>148</td>
</tr>
<tr>
<td>7.16</td>
<td>Substituting inside a signature</td>
<td>148</td>
</tr>
</tbody>
</table>
III The OCaml tools

8 Batch compilation (ocamlc)
  8.1 Overview of the compiler .............................................. 153
  8.2 Options ........................................................................ 154
  8.3 Modules and the file system ............................................. 161
  8.4 Common errors ............................................................... 162

9 The toplevel system (ocaml)
  9.1 Options ........................................................................ 166
  9.2 Toplevel directives ........................................................ 168
  9.3 The toplevel and the module system ................................. 170
  9.4 Common errors ............................................................... 170
  9.5 Building custom toplevel systems: ocamlmktop .................. 171
  9.6 Options ........................................................................ 171

10 The runtime system (ocamlrun)
  10.1 Overview ...................................................................... 173
  10.2 Options ........................................................................ 174
  10.3 Dynamic loading of shared libraries ................................. 176
  10.4 Common errors ............................................................... 176

11 Native-code compilation (ocamlopt)
  11.1 Overview of the compiler ................................................ 179
  11.2 Options ........................................................................ 180
  11.3 Common errors ............................................................... 188
  11.4 Running executables produced by ocamlopt ...................... 188
  11.5 Compatibility with the bytecode compiler ......................... 188

12 Lexer and parser generators (ocamllex, ocamlyacc)
  12.1 Overview of ocamllex ..................................................... 191
  12.2 Syntax of lexer definitions .............................................. 192
  12.3 Overview of ocamlyacc .................................................. 196
  12.4 Syntax of grammar definitions ........................................ 196
  12.5 Options ........................................................................ 199
  12.6 A complete example ...................................................... 199
  12.7 Common errors ............................................................... 201

13 Dependency generator (ocamldep)
  13.1 Options ........................................................................ 203
  13.2 A typical Makefile .......................................................... 204
IV The OCaml library

19 The core library

19.1 Built-in types and predefined exceptions

19.2 Module Pervasives: The initially opened module.

20 The standard library

20.1 Module Arg: Parsing of command line arguments.

20.2 Module Array: Array operations.

20.3 Module Buffer: Extensible string buffers.

20.4 Module Callback: Registering Caml values with the C runtime.

20.5 Module Char: Character operations.

20.6 Module Complex: Complex numbers.

20.7 Module Digest: MD5 message digest.

20.8 Module Filename: Operations on file names.

20.9 Module Format: Pretty printing.

20.10 Module Gc: Memory management control and statistics; finalised values.

20.11 Module Genlex: A generic lexical analyzer.

20.12 Module Hashtbl: Hash tables and hash functions.

20.13 Module Int32: 32-bit integers.

20.14 Module Int64: 64-bit integers.

20.15 Module Lazy: Deferred computations.

20.16 Module Lexing: The run-time library for lexers generated by ocamllex.

20.17 Module List: List operations.

20.18 Module Map: Association tables over ordered types.

20.19 Module Marshal: Marshaling of data structures.


20.21 Module Oo: Operations on objects.

20.22 Module Parsing: The run-time library for parsers generated by ocamlyacc.

20.23 Module Printexc: Facilities for printing exceptions.

20.24 Module Printf: Formatted output functions.

20.25 Module Queue: First-in first-out queues.


20.27 Module Scanf: Formatted input functions.

20.28 Module Set: Sets over ordered types.

20.29 Module Sort: Sorting and merging lists.

20.30 Module Stack: Last-in first-out stacks.

20.31 Module StdLabels: Standard labeled libraries.

20.32 Module Stream: Streams and parsers.

20.33 Module String: String operations.

20.34 Module Sys: System interface.

20.35 Module Weak: Arrays of weak pointers and hash tables of weak pointers.
21 The unix library: Unix system calls
   21.1 Module Unix: Interface to the Unix system
   21.2 Module UnixLabels: labeled version of the interface

22 The num library: arbitrary-precision rational arithmetic
   22.1 Module Num: Operation on arbitrary-precision numbers
   22.2 Module Big_int: Operations on arbitrary-precision integers
   22.3 Module Arith_status: Flags that control rational arithmetic

23 The str library: regular expressions and string processing
   23.1 Module Str: Regular expressions and high-level string processing

24 The threads library
   24.1 Module Thread: Lightweight threads for Posix 1003.1c and Win32
   24.2 Module Mutex: Locks for mutual exclusion
   24.3 Module Condition: Condition variables to synchronize between threads
   24.4 Module Event: First-class synchronous communication
   24.5 Module ThreadUnix: Thread-compatible system calls

25 The graphics library
   25.1 Module Graphics: Machine-independent graphics primitives

26 The dbm library: access to NDBM databases
   26.1 Module Dbm: Interface to the NDBM database

27 The dynlink library: dynamic loading and linking of object files
   27.1 Module Dynlink: Dynamic loading of object files

28 The LablTk library: Tcl/Tk GUI interface
   28.1 Module Tk: Basic functions and types for LablTk

29 The bigarray library
   29.1 Module Bigarray: Large, multi-dimensional, numerical arrays
   29.2 Big arrays in the Caml-C interface

V Appendix
   Index to the library
   Index of keywords
Foreword

This manual documents the release 3.12 of the OCaml system. It is organized as follows.

- Part I “An introduction to OCaml”, gives an overview of the language.
- Part II “The OCaml language”, is the reference description of the language.
- Part III “The OCaml tools”, documents the compilers, toplevel system, and programming utilities.
- Part IV “The OCaml 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

OCaml 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 the Unix family of operating systems, including Linux and MacOS X.

Windows:
This is material specific to Microsoft Windows (2000, XP, Vista).

License

The OCaml system is copyright © 1996–2011 Institut National de Recherche en Informatique et en Automatique (INRIA). INRIA holds all ownership rights to the OCaml system.

The OCaml system is open source and can be freely redistributed. See the file LICENSE in the distribution for licensing information.

The present documentation is copyright © 2011 Institut National de Recherche en Informatique et en Automatique (INRIA). The OCaml documentation and user’s manual may be reproduced and distributed in whole or in part, subject to the following conditions:

- The copyright notice above and this permission notice must be preserved complete on all complete or partial copies.
• Any translation or derivative work of the OCaml documentation and user’s manual must be approved by the authors in writing before distribution.

• If you distribute the OCaml documentation and user’s manual in part, instructions for obtaining the complete version of this manual must be included, and a means for obtaining a complete version provided.

• Small portions may be reproduced as illustrations for reviews or quotes in other works without this permission notice if proper citation is given.

Availability

The complete OCaml distribution can be accessed via the Web site http://caml.inria.fr/. This Web site contains a lot of additional information on OCaml.
Part I

An introduction to OCaml
Chapter 1

The core language

This part of the manual is a tutorial introduction to the OCaml 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 3 deals with the object-oriented features, and chapter 2 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).

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

# let pi = 4.0 *. atan 1.0;;
val pi : float = 3.14159265358979312

# 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.

```caml
# 1.0 * 2;;
Error: This expression has type float but an expression was expected of type int
```
Recursive functions are defined with the `let rec` binding:

```ocaml
# let rec fib n =  
#   if n < 2 then n else fib(n-1) + fib(n-2);; 
val fib : int -> int = <fun>
# fib 10;; 
- : int = 55
```

### 1.2 Data types

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

```ocaml
# (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.

```ocaml
# 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:

```ocaml
# 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
```
Chapter 1. The core language

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

# sin' pi;;
- : float = -1.00000000013961143

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};;

val ratio : 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;;
val number : 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;;
val sign : sign = Positive | Negative

To define arithmetic operations for the number type, we use pattern-matching on the two numbers involved:
Chapter 1. The core language

# 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.

```caml
# 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:

```caml
# 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:

```caml
# 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
```
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:

```ocaml
# 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.

```ocaml
# type 'a ref = { mutable contents: 'a };;
val ( ! ) : 'a ref -> 'a = <fun>
# let (:=) r newval = r.contents <- newval;;
val ( := ) : 'a ref -> 'a -> unit = <fun>
```

In some special cases, you may need to store a polymorphic function in a data structure, keeping its polymorphism. Without user-provided type annotations, this is not allowed, as polymorphism is only introduced on a global level. However, you can give explicitly polymorphic types to record fields.

```ocaml
# type idref = { mutable id: 'a. 'a -> 'a };;
val idref : idref = {id = <fun>}
# let r = {id = fun x -> x};;
val r : idref = {id = <fun>}
# let g s = (s.id 1, s.id true);;
val g : idref -> int * bool = <fun>
# r.id <- (fun x -> print_string "called id\n"; x);;
- : unit = ()
# g r;;
called id
called id
- : int * bool = (1, true)
```
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.

```ml
# 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 [];;
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:

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

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

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

```ml
# 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) \rightarrow \text{eval env } f \ast \text{ eval env } g
# | Quot(f, g) \rightarrow \text{eval env } f \div \text{ eval env } g;

val eval : (string * float) list \rightarrow \text{expression} \rightarrow float = \text{<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 \rightarrow \text{string} \rightarrow \text{expression} = \text{<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. \(2x+1\)).

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

Chapter 1. The core language

# | 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 [12]), 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 ocamlllex and ocamlyacc is given in chapter [12]. Here, we will use stream parsers. The syntactic support for stream parsers is provided by the Camlp4 preprocessor, which can be loaded into the interactive toplevel via the #load directives below.

# #load "dynlink.cma";;
# #load "camlp4o.cma";;
Camlp4 Parsing version 3.12.1
# 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.

```ocaml
# 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 the Camlp4 documentation for more details.

```ocaml
# 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>

# let parse_expression = parser [< e = parse_expr; _ = Stream.empty >] -> e;;
val parse_expression : Genlex.token Stream.t -> expression = <fun>
```
Composing the lexer and parser, we finally obtain a function to read an expression from a character string:

```ocaml
# let read_expression s = parse_expression(lexer(Stream.of_string s));;
val read_expression : string -> expression = <fun>
# read_expression "2*(x+y)";;
- : expression = Prod (Const 2., Sum (Var "x", Var "y"))
```

A small puzzle: why do we get different results in the following two examples?

```ocaml
# read_expression "x - 1";;
- : expression = Diff (Var "x", Const 1.)
# read_expression "x-1";;
Exception: Stream.Error "".
```

Answer: the generic lexer provided by Genlex recognizes negative integer literals as one integer token. Hence, \(x-1\) is read as the token \texttt{Ident "x"} followed by the token \texttt{Int(-1)}; this sequence does not match any of the parser rules. On the other hand, the second space in \(x - 1\) causes the lexer to return the three expected tokens: \texttt{Ident "x"}, then \texttt{Kwd "}", then \texttt{Int(1)}.

### 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 \texttt{ocamlc} or \texttt{ocamlopt}. The source code must be put in a file with extension \texttt{.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:

```ocaml
(* 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 ();;
```

\texttt{Sys.argv} is an array of strings containing the command-line parameters. \texttt{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

The module system

This chapter introduces the module system of OCaml.

2.1 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:

```ocaml
# 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)
```

25
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 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)
```

### 2.2 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
#   type 'a queue (* still concrete *)
#   val empty : 'a queue
#   val insert : 'a queue -> priority -> 'a -> 'a queue
#   val remove_top : 'a queue -> 'a queue
#   val extract : 'a queue -> priority * 'a * 'a queue
# exception Queue_is_empty
# end

module type PRIOQUEUE =
sig
  type priority = int
  type 'a queue
```

Chapter 2. The module system

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;;
Error: 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;;

2.3 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 $B_1 \ldots B_n$ of $B$, yielding the corresponding structures $A_1 \ldots A_n$.

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 compare: t -> t -> comparison
# end;;
module type ORDERED_TYPE = sig type t val compare : 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.compare 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.compare 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
      type 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 compare x y = if x = y then Equal else if x < y then Less else Greater
# end;

module OrderedString :
  sig type t = string val compare : 'a -> 'a -> comparison end

# module StringSet = Set(OrderedString);;

module StringSet :
  sig
    type element = OrderedString.t
    type 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
2.4 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:

```ocaml
# 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
#       type 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
    type 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
     type 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
     type set = WrongSet(OrderedString).set
     val empty : set
     val add : element -> set -> set
     val member : element -> set -> bool
   end

# WrongStringSet.add "gee" WrongStringSet.empty;;
Error: This expression has type string but an expression was expected of 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, OCaml 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
        type set
        val empty : set
        val add : element -> set -> set
      end
Chapter 2. The module system

val member : element -> set -> bool

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

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

# NoCaseStringSet.add "FOO" AbstractStringSet.empty;;
Error: This expression has type
    AbstractStringSet.set = AbstractSet/1139(OrderedString).set
but an expression was expected of type
    NoCaseStringSet.set = AbstractSet/1171(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.

2.5 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 practi-
nal necessity to split the source into several files, called compilation units, that can be compiled separately, thus minimizing recompilation after changes.

In OCaml, 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 $A$ 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 `ocamlc -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 file $Aux.mli * ) end
  = struct (* contents of file $Aux.ml * ) end ;;
module Main: sig (* contents of file $Main.mli * ) end
  = struct (* contents of file $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 functions 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 3

Objects in Caml

(Chapter written by Jérôme Vouillon, Didier Rémy and Jacques Garrigue)

This chapter gives an overview of the object-oriented features of OCaml. Note that the relation between object, class and type in OCaml is very different from that in main stream object-oriented languages like Java or C++, so that you should not assume that similar keywords mean the same thing.

3.1 Classes and objects

The class `point` below defines 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.

```ocaml
# 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`, instance of the `point` class.

# 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.

We now invoke some methods to `p`:

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

```ocaml
# 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 \texttt{point} can also be abstracted over the initial values of the \( x \) coordinate.

```ocaml
# 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
```

Like in function definitions, the definition above can be abbreviated as:

```ocaml
# 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
```
Chapter 3. Objects in Caml

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 relative to its initial position.

```
# 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 the nearest point on a 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 \texttt{x\_init} coordinate is not on the grid.) In fact, the same effect could here be obtained by calling the definition of class \texttt{point} with the value of the \texttt{origin}.

\begin{verbatim}
# 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:

\begin{verbatim}
# let new_adjusted_point x_init = new point ((x_init / 10) * 10);
val new_adjusted_point : int -> point = <fun>
\end{verbatim}

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; an alternative is to use initializers, as described below in section 3.4.

### 3.2 Immediate objects

There is another, more direct way to create an object: create it without going through a class.

The syntax is exactly the same as for class expressions, but the result is a single object rather than a class. All the constructs described in the rest of this section also apply to immediate objects.

\begin{verbatim}
# let p =
#   object
#     val mutable x = 0
#     method get_x = x
#     method move d = x <- x + d
#   end;;
val p : < get_x : int; move : int -> unit > = <obj>

# p#get_x;;
- : int = 0
# p#move 3;;
- : unit = ()
# p#get_x;;
- : int = 3
\end{verbatim}

Unlike classes, which cannot be defined inside an expression, immediate objects can appear anywhere, using variables from their environment.

\begin{verbatim}
# let minmax x y =
#   if x < y then object method min = x method max = y end
#   else object method min = y method max = x end;;
val minmax : 'a -> 'a -> < max : 'a; min : 'a > = <fun>
\end{verbatim}

Immediate objects have two weaknesses compared to classes: their types are not abbreviated, and you cannot inherit from them. But these two weaknesses can be advantages in some situations, as we will see in sections 3.3 and 3.10.
3.3 Reference to self

A method or an initializer can 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`.)

```ocaml
# 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` is inherited, the variable `s` will be correctly bound to the object of the subclass.

A common problem with self is that, as its type may be extended in subclasses, you cannot fix it in advance. Here is a simple example.

```ocaml
# let ints = ref [];;
val ints : '_a list ref = {contents = []}
#
# class my_int =
#   object (self)
#     method n = 1
#     method register = ints := self :: !ints
#   end
;
Error: This expression has type < n : int; register : 'a; .. >
but an expression was expected of type 'b
Self type cannot escape its class
```

You can ignore the first two lines of the error message. What matters is the last one: putting self into an external reference would make it impossible to extend it afterwards. We will see in section 3.12 a workaround to this problem. Note however that, since immediate objects are not extensible, the problem does not occur with them.
3.4 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, it can access self and the instance variables.

```ocaml
# 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 5.1.

3.5 Virtual methods

It is possible to declare a method without actually defining it, using the keyword virtual. This method will be provided later 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 type abbreviations (treating virtual methods as other methods.)
# class virtual abstract_point x_init =
#    object (self)
#      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
      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
#      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_offset : int
      method get_x : int
      method move : int -> unit
    end

Instance variables can also be declared as virtual, with the same effect as with methods.

# class virtual abstract_point2 =
#    object
#      val mutable virtual x : int
#      method move d = x <- x + d
#    end;

class virtual abstract_point2 :
    object val mutable virtual x : int method move : int -> unit end

# class point2 x_init =
#    object
#      inherit abstract_point2
#      val mutable x = x_init
#      method get_offset = x - x_init
#    end;

class point2 :
    int ->
    object
      val mutable x : int
method get_offset : int
method move : int -> unit
end

3.6 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.

```ocaml
# 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;;
Error: This expression has type restricted_point
    It has no method move

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

Note that this is not the same thing as private and protected methods in Java or C++, which can be called from other objects of the same class. This is a direct consequence of the independence between types and classes in OCaml: two unrelated classes may produce objects of the same type, and there is no way at the type level to ensure that an object comes from a specific class. However a possible encoding of friend methods is given in section 3.17.

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.

```ocaml
# 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 mention a method without providing its definition. Since we didn’t add the private annotation, this makes the method public, keeping the original 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

The constraint on self’s type is requiring a public move method, and this is sufficient to override private.

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

# class point_again x =
#   object
#   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

Of course, private methods can also be virtual. Then, the keywords must appear in this order method private virtual.
3.7 Class interfaces

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

```ocaml
# 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 program documentation, class interfaces can be used to constrain the type of a class. Both concrete instance variables and concrete private methods can be hidden by a class type constraint. Public methods and virtual members, however, cannot.

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

Or, equivalently:

```ocaml
# 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.

```ocaml
# 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
```
3.8 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.

```ml
# 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_successor 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.

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

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

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

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

# let incr p = set_x p (get_successor p);;
val incr : < get_x : int; set_x : int -> 'a; .. > -> 'a = <fun>
```
### 3.9 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 a pseudo value identifier that can only be used to invoke a super-class method, as in `super#print`.

```ocaml
# 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
```

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

A private method that has been hidden in the parent class is no longer visible, and is thus not overridden. Since initializers are treated as private methods, all initializers along the class hierarchy are evaluated, in the order they are introduced.

### 3.10 Parameterized classes

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

```ocaml
# class ref x_init =
#    object
```
Chapter 3. Objects in Caml

```ocaml
# val mutable x = x_init
# method get = x
# method set y = x <- y
# end;

Error: 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 either 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:

```ocaml
# 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
```

Note that since immediate objects do not define a class type, they have no such restriction.

```ocaml
# let new_ref x_init =
# object
#  val mutable x = x_init
#  method get = x
#  method set y = x <- y
# end;

val new_ref : `a -> < get : `a; set : `a -> unit > = <fun>
```

On the other hand, 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.

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

class ['a] ref :
```

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.

```ocaml
# 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 complex example: define a circle, whose center may be any kind of point. We put an additional type constraint in method \texttt{move}, since no free variables must remain unaccounted for by the class type parameters.

```ocaml
# 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 \texttt{circle}, using a \texttt{constraint} clause in the class definition, is shown below. The type \texttt{#point} used below in the \texttt{constraint} clause is an abbreviation produced by the definition of class \texttt{point}. This abbreviation unifies with the type of any object belonging to a subclass of class \texttt{point}. It actually expands to \texttt{< get_x : int; move : int -> unit; .. >}. This leads to the following alternate definition of \texttt{circle}, which has slightly stronger constraints on its argument, as we now expect \texttt{center} to have a method \texttt{get_x}.
Chapter 3. Objects in Caml

# 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 that 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 between [ 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

3.11 Polymorphic methods

While parameterized classes may be polymorphic in their contents, they are not enough to allow polymorphism of method use.

A classical example is defining an iterator.

# List.fold_left;;
- : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>
At first look, we seem to have a polymorphic iterator, however this does not work in practice.

```
# let l = new intlist [1; 2; 3];;
val l : '_a intlist = <obj>
# l#fold (fun x y -> x+y) 0;;
- : int = 6
```

Our iterator works, as shows its first use for summation. However, since objects themselves are not polymorphic (only their constructors are), using the `fold` method fixes its type for this individual object. Our next attempt to use it as a string iterator fails.

```
The problem here is that quantification was wrongly located: this is not the class we want to be polymorphic, but the `fold` method. This can be achieved by giving an explicitly polymorphic type in the method definition.

# let l = new intlist [1; 2; 3];;
val l : intlist = <obj>
# l#fold (fun x y -> x+y) 0;;
- : int = 6
```

As you can see in the class type shown by the compiler, while polymorphic method types must be fully explicit in class definitions (appearing immediately after the method name), quantified type
variables can be left implicit in class descriptions. Why require types to be explicit? The problem
is that \((\text{int} \to \text{int} \to \text{int}) \to \text{int} \to \text{int}\) would also be a valid type for \texttt{fold}, and it happens
to be incompatible with the polymorphic type we gave (automatic instantiation only works for
toplevel types variables, not for inner quantifiers, where it becomes an undecidable problem.) So
the compiler cannot choose between those two types, and must be helped.

However, the type can be completely omitted in the class definition if it is already known,
through inheritance or type constraints on self. Here is an example of method overriding.

```plaintext
# class intlist_rev l =
# object
#   inherit intlist l
#   method fold f accu = List.fold_left f accu (List.rev l)
# end;;
```

The following idiom separates description and definition.

```plaintext
# class type ['a] iterator =
# object method fold : ('b -> 'a -> 'b) -> 'b -> 'b end;;
# class intlist l =
# object (self : int #iterator)
#   method empty = (l = [])
#   method fold f accu = List.fold_left f accu l
# end;;
```

Note here the \((self : int #iterator)\) idiom, which ensures that this object implements the
interface \texttt{iterator}.

Polymorphic methods are called in exactly the same way as normal methods, but you should
be aware of some limitations of type inference. Namely, a polymorphic method can only be called
if its type is known at the call site. Otherwise, the method will be assumed to be monomorphic,
and given an incompatible type.

```plaintext
# let sum lst = lst#fold (fun x y -> x+y) 0;;
val sum : < fold : (int -> int -> int) -> int -> 'a; .. > -> 'a = <fun>
```

The workaround is easy: you should put a type constraint on the parameter.

```plaintext
# let sum (lst : _ #iterator) = lst#fold (fun x y -> x+y) 0;;
val sum : int #iterator -> int = <fun>
```

Of course the constraint may also be an explicit method type. Only occurrences of quantified
variables are required.

```plaintext
# let sum lst =
#   (lst : < fold : 'a. ('a -> _ -> 'a) -> 'a -> 'a; .. >)#fold (+) 0;;
val sum : < fold : 'a. ('a -> int -> 'a) -> 'a -> 'a; .. > -> int = <fun>
```
Another use of polymorphic methods is to allow some form of implicit subtyping in method arguments. We have already seen in section 3.8 how some functions may be polymorphic in the class of their argument. This can be extended to methods.

```ocaml
# class type point0 = object method get_x : int end;;

class type point0 = object method get_x : int end

# class distance_point x =
#   object
#     inherit point x
#     method distance : 'a. (#point0 as 'a) -> int =
#       fun other -> abs (other#get_x - x)
#   end;;

class distance_point :
  int ->
  object
    val mutable x : int
    method distance : #point0 -> int
    method get_offset : int
    method get_x : int
    method move : int -> unit
  end

# let p = new distance_point 3 in
# (p#distance (new point 8), p#distance (new colored_point 1 "blue"));;
- : int * int = (5, 2)
```

Note here the special syntax (#point0 as 'a) we have to use to quantify the extensible part of #point0. As for the variable binder, it can be omitted in class specifications. If you want polymorphism inside object field it must be quantified independently.

```ocaml
# class multi_poly =
#   object
#     method m1 : 'a. (< n1 : 'b. 'b -> 'b; .. > as 'a) -> _ =
#       fun o -> o#n1 true, o#n1 "hello"
#     method m2 : 'a 'b. (< n2 : 'b -> bool; .. > as 'a) -> 'b -> _ =
#       fun o x -> o#n2 x
#   end;;

class multi_poly :
  object
    method m1 : < n1 : 'a. 'a -> 'a; .. > -> bool * string
    method m2 : < n2 : 'b -> bool; .. > -> 'b -> bool
  end
```

In method m1, o must be an object with at least a method n1, itself polymorphic. In method m2, the argument of n2 and x must have the same type, which is quantified at the same level as 'a.

### 3.12 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
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:

```ocaml
# 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.

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

Indeed, narrowing coercions without runtime checks would be unsafe. Runtime type checks might raise exceptions, and they would require the presence of type information at runtime, which is not the case in the OCaml system. For these reasons, 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 often be omitted. For instance, one can define:

```ocaml
# let to_point cp = (cp :> point);;
val to_point : #point -> 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. Consider, for example, the following class:

```ocaml
# class c0 = object method m = {< >} method n = 0 end;;
class c0 : object ('a) method m : 'a method n : int end
```

The object type `c0` is an abbreviation for `<m : 'a; n : int> as 'a`. Consider now the type declaration:

```ocaml
# class type c1 = object method m : c1 end;;
class type c1 = object method m : c1 end
```
The object type \( c_1 \) is an abbreviation for the type \(< m : 'a>\) as \('a\). The coercion from an object of type \( c_0 \) to an object of type \( c_1 \) is correct:

```ocaml
# fun (x : c0) -> (x : c0 :> c1);;
- : c0 -> c1 = <fun>
```

However, the domain of the coercion cannot always be omitted. In that case, the solution is to use the explicit form. Sometimes, a change in the class-type definition can also solve the problem

```ocaml
# class type c2 = object ('a) method m : 'a end;;
class type c2 = object ('a) method m : 'a end
# fun (x : c0) -> (x :> c2);;
- : c0 -> c2 = <fun>
```

While class types \( c_1 \) and \( c_2 \) are different, both object types \( c_1 \) and \( c_2 \) expand to the same object type (same method names and types). Yet, when the domain of a coercion is left implicit and its co-domain is an abbreviation of a known class type, then the class type, rather than the object type, is used to derive the coercion function. This allows to leave the domain implicit in most cases when coercing form a subclass to its superclass. The type of a coercion can always be seen as below:

```ocaml
# let to_c1 x = (x :> c1);;
val to_c1 : < m : c1; .. > -> c1 = <fun>
# let to_c2 x = (x :> c2);;
val to_c2 : c0 -> c2 = <fun>
```

Note the difference between the two coercions: in the second case, the type \(#c2 = < m : 'a; .. >\) as \('a\) is polymorphically recursive (according to the explicit recursion in the class type of \( c_2 \)); hence the success of applying this coercion to an object of class \( c_0 \). On the other hand, in the first case, \( c_1 \) was only expanded and unrolled twice to obtain \(< m : < m : c1; .. >; .. >\) (remember \#c1 = \(< m : c1; .. >\) without introducing recursion. You may also note that the type of \( \text{to}_c2 \) is \#c_2 \( \rightarrow c_2 \) while the type of \( \text{to}_c1 \) is more general than \#c_1 \( \rightarrow c_1 \). This is not always true, since there are class types for which some instances of \#c are not subtypes of c, as explained in section 3.16. Yet, for parameterless classes the coercion \((_ : c)\) is always more general than \((_ : \#c :> c)\).

A common problem may occur when one tries to define a coercion to a class \( c \) while defining 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 \((_ : c)\) or \((_ : \#c :> c)\) is taken to be the identity function, as in

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

As a consequence, if the coercion is applied to \( \text{self} \), as in the following example, the type of \( \text{self} \) is unified with the closed type \( c \) (a closed object type is an object type without ellipsis). This would constrain the type of \( \text{self} \) be closed and is thus rejected. Indeed, the type of \( \text{self} \) 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.
Chapter 3. Objects in Caml

# class c = object method m = 1 end
# and d = object (self)
#   inherit c
#   method n = 2
#   method as_c = (self => c)
# end;;

Error: This expression cannot be coerced to type c = m : int ; it has type
< as_c : c ; m : int ; n : int ; .. >
but is here used with type c
Self type cannot be unified with a closed object type

However, the most common instance of this problem, coercing self to its current class, is detected
as a special case by the type checker, and properly typed.

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

This allows the following idiom, keeping a list of all objects belonging to a class or its subclasses:

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

# class c (m : int) =
# object (self)
#   method m = m
#   initializer all_c := (self => c) :: !all_c
# end;;
class c : int -> object method m : int end

This idiom can in turn be used to retrieve an object whose type has been weakened:

# let rec lookup_obj obj = function [] -> raise Not_found
#   | obj' :: l ->
#     if (obj => < >) = (obj' => < >) then obj' else lookup_obj obj l ;;
val lookup_obj : < .. > -> ( < .. > as 'a) list -> 'a = <fun>

# let lookup_c obj = lookup_obj obj !all_c;;
val lookup_c : < .. > -> < m : int > = <fun>

The type < m : int > we see here is just the expansion of c, due to the use of a reference; we have
succeeded in getting back an object of type c.

The previous coercion problem can often be avoided by first defining the abbreviation, using a
class type:

# class type c' = object method m : int end;;
class type c' = object method m : int end

# class c : c' = object method m = 1 end
# and d = object (self)
#   inherit c
#   method n = 2
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 \( c' \).

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

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

One could think of defining the type abbreviation directly:

```
# type c' = <m : int>;;
```

However, the abbreviation \#c' cannot be defined directly in a similar way. It can only be defined by a class or a class-type definition. This is because \# sharp abbreviations carry an implicit anonymous variable .. that cannot be explicitly named. The closer you get to it is:

```
# type 'a c'_class = 'a constraint 'a = < m : int; .. >;;
```

with an extra type variable capturing the open object type.

### 3.13 Functional objects

It is possible to write a version of class \texttt{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:

```ocaml
# class bad_functional_point y =
#   object
#   val x = y
#   method get_x = x
#   method move d = new bad_functional_point (x+d)
#   end;
class bad_functional_point :
  int ->
  object
    val x : int
    method get_x : int
    method move : int -> bad_functional_point
  end
```

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 5.2.1.

### 3.14 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:

```ocaml
# 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.

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

# let q = Oo.copy p;;
val q : point = <obj>
```
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 `=` and `< >`. Two objects are equal if and only if they are physically equal. In particular, an object and its copy are not equal.

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

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

Other generic comparisons such as `<`, `<=,...`) can also be used on objects. The relation `<` defines an unspecified but strict ordering on objects. The ordering relationship between two objects is fixed once for all after the two objects have been created and it is not affected by mutation of fields.

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

```ocaml
# 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 for saving and restoring the state of objects.

```ocaml
# 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. The backup facility can be added to any class using multiple inheritance.
Chapter 3. Objects in Caml

# 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 backup 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
3.15 Recursive classes

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

```ocaml
# 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
and widget : window -> object val window : window method window : window end
```

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

3.16 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.

```ocaml
# 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 OCaml. 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
#   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.

Similarly, the type money2 below 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 5.2.1 and 5.2.3.

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

3.17 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;

Another example of friend functions may be found in section \[5.2.3\]. 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 4

Labels and variants

(Chapter written by Jacques Garrigue)

This chapter gives an overview of the new features in OCaml 3: labels, and polymorphic variants.

4.1 Labels

If you have a look at modules ending in Labels in the standard library, you will see that function types have annotations you did not have in the functions you defined yourself.

# ListLabels.map;;
- : f:('a -> 'b) -> 'a list -> 'b list = <fun>

# StringLabels.sub;;
- : string -> pos:int -> len:int -> string = <fun>

Such annotations of the form \texttt{name:} are called labels. They are meant to document the code, allow more checking, and give more flexibility to function application. You can give such names to arguments in your programs, by prefixing them with a tilde \texttt{~}.

# let f ~x ~y = x - y;;
val f : x:int -> y:int -> int = <fun>

# let x = 3 and y = 2 in f ~x ~y;;
- : int = 1

When you want to use distinct names for the variable and the label appearing in the type, you can use a naming label of the form \texttt{~name:}. This also applies when the argument is not a variable.

# let f ~x:x1 ~y:y1 = x1 - y1;;
val f : x:int -> y:int -> int = <fun>

# f ~x:3 ~y:2;;
- : int = 1
Labels obey the same rules as other identifiers in Caml, that is you cannot use a reserved keyword (like in or to) as label.

Formal parameters and arguments are matched according to their respective labels, the absence of label being interpreted as the empty label. This allows commuting arguments in applications. One can also partially apply a function on any argument, creating a new function of the remaining parameters.

```ocaml
# let f ~x ~y = x - y;;
val f : x:int -> y:int -> int = <fun>
# f ~y:2 ~x:3;;
- : int = 1
# ListLabels.fold_left;;
- : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a = <fun>
# ListLabels.fold_left [1;2;3] ~init:0 ~f: (+);;
- : int = 6
# ListLabels.fold_left ~init:0;~
- : f:(int -> 'a -> int) -> 'a list -> int = <fun>
```

If in a function several arguments bear the same label (or no label), they will not commute among themselves, and order matters. But they can still commute with other arguments.

```ocaml
# let hline ~x:x1 ~x:x2 ~y = (x1, x2, y);;
val hline : x:'a -> x:'b -> y:'c -> 'a * 'b * 'c = <fun>
# hline ~x:3 ~y:2 ~x:5;;
- : int * int * int = (3, 5, 2)
```

As an exception to the above parameter matching rules, if an application is total, labels may be omitted. In practice, most applications are total, so that labels can be omitted in applications.

```ocaml
# f 3 2;;
- : int = 1
# ListLabels.map succ [1;2;3];;
- : int list = [2; 3; 4]
```

But beware that functions like `ListLabels.fold_left` whose result type is a type variable will never be considered as totally applied.

```ocaml
# ListLabels.fold_left (+) 0 [1;2;3];;
Error: This expression has type int -> int -> int
  but an expression was expected of type 'a list
```

When a function is passed as an argument to an higher-order function, labels must match in both types. Neither adding nor removing labels are allowed.

---

1This correspond to the commuting label mode of Objective Caml 3.00 through 3.02, with some additional flexibility on total applications. The so-called classic mode (`-nolabels` options) is now deprecated for normal use.
Chapter 4. Labels and variants

# let h g = g ~x:3 ~y:2;;
val h : (x:int -> y:int -> 'a) -> 'a = <fun>

# h f;;
- : int = 1

# h (+);;
Error: This expression has type int -> int -> int
but an expression was expected of type x:int -> y:int -> 'a

Note that when you don’t need an argument, you can still use a wildcard pattern, but you must prefix it with the label.

# h (fun ~x:_ ~y -> y+1);;
- : int = 3

4.1.1 Optional arguments

An interesting feature of labeled arguments is that they can be made optional. For optional parameters, the question mark ? replaces the tilde ~ of non-optional ones, and the label is also prefixed by ? in the function type. Default values may be given for such optional parameters.

# let bump ?(step = 1) x = x + step;;
val bump : ?step:int -> int -> int = <fun>

# bump 2;;
- : int = 3

# bump ~step:3 2;;
- : int = 5

A function taking some optional arguments must also take at least one non-labeled argument. This is because the criterion for deciding whether an optional has been omitted is the application on a non-labeled argument appearing after this optional argument in the function type.

# let test ?(x = 0) ?(y = 0) () ?(z = 0) () = (x, y, z);;

# test ();;
- : ?z:int -> unit -> int * int * int = <fun>

# test ~x:2 () ~z:3 ();;
- : int * int * int = (2, 0, 3)

Optional parameters may also commute with non-optional or unlabelled ones, as long as they are applied simultaneously. By nature, optional arguments do not commute with unlabeled arguments applied independently.
# test ~y:2 ~x:3 () ();
- : int * int * int = (3, 2, 0)

# test () () ~z:1 ~y:2 ~x:3;;
- : int * int * int = (3, 2, 1)

# (test () ()) ~z:1;;

Error: This expression is not a function; it cannot be applied

Here (test () ()) is already (0,0,0) and cannot be further applied.

Optional arguments are actually implemented as option types. If you do not give a default value, you have access to their internal representation, type 'a option = None | Some of 'a. You can then provide different behaviors when an argument is present or not.

# let bump ?step x =
#  match step with
#   | None -> x * 2
#   | Some y -> x + y
#;;
val bump : ?step:int -> int -> int = <fun>

It may also be useful to relay an optional argument from a function call to another. This can be done by prefixing the applied argument with ? . This question mark disables the wrapping of optional argument in an option type.

# let test2 ?x ?y () = test ?x ?y () ;;;
val test2 : ?x:int -> ?y:int -> unit -> int * int * int = <fun>

# test2 ?x:None;;
- : ?y:int -> unit -> int * int * int = <fun>

4.1.2 Labels and type inference

While they provide an increased comfort for writing function applications, labels and optional arguments have the pitfall that they cannot be inferred as completely as the rest of the language. You can see it in the following two examples.

# let h' g = g ~y:2 ~x:3;;
val h' : (y:int -> x:int -> 'a) -> 'a = <fun>

# h' f;;
Error: This expression has type x:int -> y:int -> int
       but an expression was expected of type y:int -> x:int -> 'a

# let bump_it bump x =
#  bump ~step:2 x;;
val bump_it : (step:int -> 'a -> 'b) -> 'a -> 'b = <fun>

# bump_it bump 1;;
Error: This expression has type ?step:int -> int -> int
       but an expression was expected of type step:int -> 'a -> 'b
The first case is simple: $g$ is passed $\sim y$ and then $\sim x$, but $f$ expects $\sim x$ and then $\sim y$. This is correctly handled if we know the type of $g$ to be $x:int \to y:int \to int$ in advance, but otherwise this causes the above type clash. The simplest workaround is to apply formal parameters in a standard order.

The second example is more subtle: while we intended the argument $\text{bump}$ to be of type $\text{?step:int} \to \text{int} \to \text{int}$, it is inferred as $\text{step:int} \to \text{int} \to \text{'a}$. These two types being incompatible (internally normal and optional arguments are different), a type error occurs when applying $\text{bump}_\text{it}$ to the real $\text{bump}$.

We will not try here to explain in detail how type inference works. One must just understand that there is not enough information in the above program to deduce the correct type of $g$ or $\text{bump}$. That is, there is no way to know whether an argument is optional or not, or which is the correct order, by looking only at how a function is applied. The strategy used by the compiler is to assume that there are no optional arguments, and that applications are done in the right order.

The right way to solve this problem for optional parameters is to add a type annotation to the argument $\text{bump}$.

```ocaml
# let bump_it (bump : ?step:int -> int -> int) x =
#   bump ~step:2 x;;
val bump_it : (?step:int -> int -> int) -> int -> int = <fun>

# bump_it bump 1;;
- : int = 3
```

In practice, such problems appear mostly when using objects whose methods have optional arguments, so that writing the type of object arguments is often a good idea.

Normally the compiler generates a type error if you attempt to pass to a function a parameter whose type is different from the expected one. However, in the specific case where the expected type is a non-labeled function type, and the argument is a function expecting optional parameters, the compiler will attempt to transform the argument to have it match the expected type, by passing None for all optional parameters.

```ocaml
# let twice f (x : int) = f(f x);;
val twice : (int -> int) -> int -> int = <fun>

# twice bump 2;;
- : int = 8
```

This transformation is coherent with the intended semantics, including side-effects. That is, if the application of optional parameters shall produce side-effects, these are delayed until the received function is really applied to an argument.

### 4.1.3 Suggestions for labeling

Like for names, choosing labels for functions is not an easy task. A good labeling is a labeling which

- makes programs more readable,
- is easy to remember,
• when possible, allows useful partial applications.

We explain here the rules we applied when labeling OCaml libraries.

To speak in an “object-oriented” way, one can consider that each function has a main argument, its object, and other arguments related with its action, the parameters. To permit the combination of functions through functionals in commuting label mode, the object will not be labeled. Its role is clear by the function itself. The parameters are labeled with names reminding either of their nature or role. Best labels combine in their meaning nature and role. When this is not possible the role is to prefer, since the nature will often be given by the type itself. Obscure abbreviations should be avoided.

```
ListLabels.map : f:('a -> 'b) -> 'a list -> 'b list
```

When there are several objects of same nature and role, they are all left unlabeled.

```
ListLabels.iter2 : f:('a -> 'b -> 'c) -> 'a list -> 'b list -> unit
```

When there is no preferable object, all arguments are labeled.

```
StringLabels.blit :
  src:string -> src_pos:int -> dst:string -> dst_pos:int -> len:int -> unit
```

However, when there is only one argument, it is often left unlabeled.

```
StringLabels.create : int -> string
```

This principle also applies to functions of several arguments whose return type is a type variable, as long as the role of each argument is not ambiguous. Labeling such functions may lead to awkward error messages when one attempts to omit labels in an application, as we have seen with `ListLabels.fold_left`.

Here are some of the label names you will find throughout the libraries.

<table>
<thead>
<tr>
<th>Label</th>
<th>Meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>f</code></td>
<td>a function to be applied</td>
</tr>
<tr>
<td><code>pos</code></td>
<td>a position in a string or array</td>
</tr>
<tr>
<td><code>len</code></td>
<td>a length</td>
</tr>
<tr>
<td><code>buf</code></td>
<td>a string used as buffer</td>
</tr>
<tr>
<td><code>src</code></td>
<td>the source of an operation</td>
</tr>
<tr>
<td><code>dst</code></td>
<td>the destination of an operation</td>
</tr>
<tr>
<td><code>init</code></td>
<td>the initial value for an iterator</td>
</tr>
<tr>
<td><code>cmp</code></td>
<td>a comparison function, e.g. <code>Pervasives.compare</code></td>
</tr>
<tr>
<td><code>mode</code></td>
<td>an operation mode or a flag list</td>
</tr>
</tbody>
</table>

All these are only suggestions, but one shall keep in mind that the choice of labels is essential for readability. Bizarre choices will make the program harder to maintain.

In the ideal, the right function name with right labels shall be enough to understand the function’s meaning. Since one can get this information with OCamlBrowser or the `ocaml` toplevel, the documentation is only used when a more detailed specification is needed.
4.2 Polymorphic variants

Variants as presented in section 1.4 are a powerful tool to build data structures and algorithms. However, they sometimes lack flexibility when used in modular programming. This is due to the fact every constructor reserves a name to be used with a unique type. One cannot use the same name in another type, or consider a value of some type to belong to some other type with more constructors.

With polymorphic variants, this original assumption is removed. That is, a variant tag does not belong to any type in particular, the type system will just check that it is an admissible value according to its use. You need not define a type before using a variant tag. A variant type will be inferred independently for each of its uses.

Basic use

In programs, polymorphic variants work like usual ones. You just have to prefix their names with a backquote character ‘.

```
# ['On; 'Off];;
- : [> 'Off | 'On ] list = ['On; 'Off]

# 'Number 1;;
- : [> 'Number of int ] = 'Number 1

# let f = function 'On -> 1 | 'Off -> 0 | 'Number n -> n;;
val f : [< 'Number of int | 'Off | 'On ] -> int = <fun>

# List.map f ['On; 'Off];;
- : int list = [1; 0]
```

[>'Off|'On] list means that to match this list, you should at least be able to match 'Off and 'On, without argument. [<'On|'Off|'Number of int] means that f may be applied to 'Off, 'On (both without argument), or 'Number n where n is an integer. The > and < inside the variant type shows that they may still be refined, either by defining more tags or allowing less. As such they contain an implicit type variable. Both variant types appearing only once in the type, the implicit type variables they constrain are not shown.

The above variant types were polymorphic, allowing further refinement. When writing type annotations, one will most often describe fixed variant types, that is types that can be no longer refined. This is also the case for type abbreviations. Such types do not contain < or >, but just an enumeration of the tags and their associated types, just like in a normal datatype definition.

```
# type 'a vlist = ['Nil | 'Cons of 'a * 'a vlist];;
type 'a vlist = [ 'Cons of 'a * 'a vlist | 'Nil ]

# let rec map f : 'a vlist -> 'b vlist = function
# | 'Nil -> 'Nil
# | 'Cons(a, l) -> 'Cons(f a, map f l)
# ;;
val map : ('a -> 'b) -> 'a vlist -> 'b vlist = <fun>
```
Advanced use

Type-checking polymorphic variants is a subtle thing, and some expressions may result in more complex type information.

```ocaml
# let f = function 'A -> 'C | 'B -> 'D | x -> x;;
val f : ('a as 'a) -> 'a = <fun>
# f 'E;;
- : ('a as 'a) = 'E
```

Here we are seeing two phenomena. First, since this matching is open (the last case catches any tag), we obtain the type `('a as 'a)` rather than `('a)` in a closed matching. Then, since `x` is returned as is, input and return types are identical. The notation `as 'a` denotes such type sharing. If we apply `f` to yet another tag `'E`, it gets added to the list.

```ocaml
# let f1 = function 'A x -> x = 1 | 'B -> true | 'C -> false
# let f2 = function 'A x -> x = "a" | 'B -> true ;;
val f1 : ('a of int | 'B | 'C) -> bool = <fun>
val f2 : ('a of string | 'B) -> bool = <fun>
# let f x = f1 x && f2 x;;
val f : ('a of string & int | 'B) -> bool = <fun>
```

Here `f1` and `f2` both accept the variant tags `'A` and `'B`, but the argument of `'A` is `int` for `f1` and `string` for `f2`. In `f`’s type `('C)`, only accepted by `f1`, disappears, but both argument types appear for `('A)` as `int & string`. This means that if we pass the variant tag `'A` to `f`, its argument should be both `int` and `string`. Since there is no such value, `f` cannot be applied to `('A)`, and `('B)` is the only accepted input.

Even if a value has a fixed variant type, one can still give it a larger type through coercions. Coercions are normally written with both the source type and the destination type, but in simple cases the source type may be omitted.

```ocaml
# type 'a wlist = ['Nil | 'Cons of 'a * 'a wlist | 'Snoc of 'a wlist * 'a];;
type 'a wlist = ['Nil | 'Cons of 'a * 'a wlist | 'Snoc of 'a wlist * 'a]
# let wlist_of_vlist l = (l : 'a vlist :> 'a wlist);;
val wlist_of_vlist : 'a vlist -> 'a wlist = <fun>
# let open_vlist l = (l : 'a vlist :> ['a vlist]);;
val open_vlist : 'a vlist -> ['a vlist] = <fun>
# fun x -> (x : ['A | 'B | 'C]);;
- : ['A | 'B | 'C] = <fun>
```

You may also selectively coerce values through pattern matching.
Chapter 4. Labels and variants

When an or-pattern composed of variant tags is wrapped inside an alias-pattern, the alias is given a type containing only the tags enumerated in the or-pattern. This allows for many useful idioms, like incremental definition of functions.

```ocaml
# let num x = 'Num x
# let eval1 eval ('Num x) = x
# let rec eval x = eval1 eval x ;;
val num : 'a -> [> 'Num of 'a ] = <fun>
val eval1 : 'a -> [< 'Num of 'b ] -> 'b = <fun>
val eval : [< 'Num of 'a ] -> 'a = <fun>

# let plus x y = 'Plus(x,y)
# let eval2 eval = function
# | 'Plus(x,y) -> eval x + eval y
# | 'Num _ as x -> eval1 eval x
# let rec eval x = eval2 eval x ;;
val plus : 'a -> 'b -> [> 'Plus of 'a * 'b ] = <fun>
val eval2 : ('a -> int) -> [< 'Num of int | 'Plus of 'a * 'a ] -> int = <fun>
val eval : ([< 'Num of int | 'Plus of 'a * 'a ] as 'a) -> int = <fun>
```

To make this even more comfortable, you may use type definitions as abbreviations for or-patterns. That is, if you have defined `type myvariant = ['Tag1 int | 'Tag2 bool]`, then the pattern `#myvariant` is equivalent to writing `'Tag1(_ : int) | 'Tag2(_ : bool)`.

Such abbreviations may be used alone,

```ocaml
# let f = function
# | #myvariant -> "myvariant"
# | 'Tag3 -> "Tag3";;
val f : [< 'Tag1 of int | 'Tag2 of bool | 'Tag3 ] -> string = <fun>
```

or combined with with aliases.

```ocaml
# let g1 = function 'Tag1 _ -> "Tag1" | 'Tag2 _ -> "Tag2";;
val g1 : [< 'Tag1 of 'a | 'Tag2 of 'b ] -> string = <fun>
```

```ocaml
# let g = function
# | #myvariant as x -> g1 x
# | 'Tag3 -> "Tag3";;
val g : [< 'Tag1 of int | 'Tag2 of bool | 'Tag3 ] -> string = <fun>
```

4.2.1 Weaknesses of polymorphic variants

After seeing the power of polymorphic variants, one may wonder why they were added to core language variants, rather than replacing them.

The answer is twofold. One first aspect is that while being pretty efficient, the lack of static type information allows for less optimizations, and makes polymorphic variants slightly heavier than core language ones. However noticeable differences would only appear on huge data structures.

More important is the fact that polymorphic variants, while being type-safe, result in a weaker type discipline. That is, core language variants do actually much more than ensuring type-safety,
they also check that you use only declared constructors, that all constructors present in a data-
structure are compatible, and they enforce typing constraints to their parameters.

For this reason, you must be more careful about making types explicit when you use polymorphic
variants. When you write a library, this is easy since you can describe exact types in interfaces,
but for simple programs you are probably better off with core language variants.

Beware also that some idioms make trivial errors very hard to find. For instance, the following
code is probably wrong but the compiler has no way to see it.

```ocaml
# type abc = [ 'A | 'B | 'C ] ;;
type abc = [ 'A | 'B | 'C ]

# let f = function
# | 'As -> "A"
# | #abc -> "other" ;;
val f : [< 'A | 'As | 'B | 'C ] -> string = <fun>

# let f : abc -> string = f ;;
val f : abc -> string = <fun>

You can avoid such risks by annotating the definition itself.

# let f : abc -> string = function
# | 'As -> "A"
# | #abc -> "other" ;;
Warning 11: this match case is unused.
val f : abc -> string = <fun>
```
Chapter 5

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.

5.1 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 3)

# let euro = new Euro.c;;
val euro : float -> Euro.c = <fun>

# let zero = euro 0.;;
val zero : Euro.c = <obj>

# let neg x = x#times (-1.);;
val neg : < times : float -> 'a; .. > -> 'a = <fun>

# 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;;
class account :
  object
val mutable balance : Euro.c
  method balance : Euro.c
  method deposit : Euro.c -> unit
  method withdraw : Euro.c -> Euro.c
end

# let c = new account in c # deposit (euro 100.); c # withdraw (euro 50.);
- : Euro.c = <obj>

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;

class account_with_interests :
  object
    val mutable balance : Euro.c
    method balance : Euro.c
    method deposit : Euro.c -> unit
    method private interest : unit
    method withdraw : Euro.c -> Euro.c
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;

class safe_account :
  object
    val mutable balance : Euro.c
    method balance : Euro.c
    method deposit : Euro.c -> unit
    method withdraw : Euro.c -> Euro.c
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
;
class safe_account :
  object
    val mutable balance : Euro.c
    method balance : Euro.c
    method deposit : Euro.c \rightarrow unit
    method withdraw : Euro.c \rightarrow Euro.c
  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;;
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
;
class account_with_history :
  object
    val mutable balance : Euro.c
    val mutable history : Euro.c operation list
    method balance : Euro.c
    method deposit : Euro.c \rightarrow unit
    method history : Euro.c operation list
    method private trace : Euro.c operation \rightarrow unit
    method withdraw : Euro.c \rightarrow Euro.c
  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
;
class account_with_deposit :
  Euro.c \rightarrow
  object
    val mutable balance : Euro.c
val mutable history : Euro.c operation list
method balance : Euro.c
method deposit : Euro.c -> unit
method history : Euro.c operation list
method private trace : Euro.c operation -> unit
method withdraw : Euro.c -> Euro.c
end

A better alternative is:

```
# class account_with_deposit x =
#   object (self)
#     inherit account_with_history
#     initializer self#deposit x
#   end;;
class account_with_deposit :
  Euro.c ->
  object
    val mutable balance : Euro.c
    val mutable history : Euro.c operation list
    method balance : Euro.c
    method deposit : Euro.c -> unit
    method history : Euro.c operation list
    method private trace : Euro.c operation -> unit
    method withdraw : Euro.c -> Euro.c
  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;

- : Euro.c operation list = [Deposit <obj>; Retrieval <obj>]
```

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

```
# let close c = c#withdraw (c#balance);;
val close : < balance : 'a; withdraw : 'a -> 'b; .. > -> 'b = <fun>
```

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.

```ocaml
# 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 function `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.

```ocaml
# 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.

```ocaml
# module Internet_account (M : MONEY) =
#   struct
#     type m = M.c
#     module A = Account(M)
#     #
#     class bank =
```
Chapter 5. Advanced examples with classes and modules

# 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;;

5.2 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 \texttt{money} above is such an example. We show here how to do it for strings.

5.2.1 Strings

A naive definition of strings as objects could be:

# class ostring s =
#   object
#     method get n = String.get s n
#     method set n c = String.set s n c
#     method print = print_string s
#     method copy = new ostring (String.copy s)
#   end;
class ostring :
  string ->
  object
method copy : ostring
method get : int -> char
method print : unit
method set : int -> char -> unit
end

However, the method copy returns an object of the class ostring, and not an object 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;
class sub_string :
  string ->
  object
    method copy : ostring
    method get : int -> char
    method print : unit
    method set : int -> char -> unit
    method sub : int -> int -> sub_string
  end

As seen in section 3.16, 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 repr n
#     method set n c = String.set repr n c
#     method print = print_string repr
#     method copy = {< repr = String.copy repr >}
#     method sub start len = {< repr = String.sub s start len >}
#   end;
class better_string :
  string ->
  object (‘a)
    val repr : string
    method copy : ‘a
    method get : int -> char
    method print : unit
    method set : int -> char -> unit
    method sub : int -> int -> ‘a
  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:

```ocaml
# class ostring s =
#   object (self : 'mytype)
#     val repr = s
#     method repr = repr
#     method get n = String.get repr n
#     method set n c = String.set repr 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;
class ostring :
  string ->
  object ('a)
    val repr : string
    method concat : 'a -> 'a
    method copy : 'a
    method get : int -> char
    method print : unit
    method repr : string
    method set : int -> char -> unit
    method sub : int -> int -> 'a
  end

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

```ocaml
# class cstring n = ostring (String.create n);;
class cstring : int -> ostring
```

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 3.17.

**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:

```ocaml
# exception Empty;;
exception Empty
# class ['a] stack =
#   object
#     val mutable l = ([] : 'a list)
#     method push x = l <- x :: l
```
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`. A naive approach is to make `'b` an extra parameter of class `stack`:

```ocaml
# class ['a, 'b] stack2 =
# object
#   inherit ['a] stack
#   method fold f (x : 'b) = List.fold_left f x l
# end;;
class ['a, 'b] stack2 :
  object
    val mutable l : 'a list
    method clear : unit
    method length : int
    method pop : 'a
    method push : 'a -> unit
  end
```

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

```ocaml
# let s = new stack2;;
val s : ('_a, '_b) stack2 = <obj>
```

```ocaml
# s#fold (+) 0;;
- : int = 0
```

A better solution is to use polymorphic methods, which were introduced in OCaml version 3.05. Polymorphic methods makes it possible to treat the type variable `'b` in the type of `fold` as universally quantified, giving `fold` the polymorphic type `Forall 'b. ('b -> 'a -> 'b) -> 'b -> 'b`. An explicit type declaration on the method `fold` is required, since the type checker cannot infer the polymorphic type by itself.
Chapter 5. Advanced examples with classes and modules

# class ['a] stack3 =
#   object
#   inherit ['a] stack
#   method fold : 'b. ('b -> 'a -> 'b) -> 'b -> 'b
#       = fun f x -> List.fold_left f x l
#   end;
class ['a] stack3 :
  object
  val mutable l : 'a list
  method clear : unit
  method fold : ('b -> 'a -> 'b) -> 'b -> 'b
  method length : int
  method pop : 'a
  method push : 'a -> unit
end

5.2.2 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;
class type ['a, 'b] hash_table =
  object method add : 'a -> 'b -> unit method find : 'a -> 'b end

A simple implementation, which is quite reasonable for small hashtables 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;
class ['a, 'b] small_hashtbl : ['a, 'b] hash_table

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;
class ['a, 'b] hashtbl : int -> ['a, 'b] hash_table
5.2.3 Sets

Implementing sets leads to another difficulty. Indeed, the method \texttt{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 3.17. Indeed, this is the same mechanism used in the module \texttt{Set} in the absence of objects.

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

```ocaml
# 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 add x = List.exists ((=) x) repr
#             method union (s : ’b) = {< repr = merge [x] repr >}
#             method iter (f : ‘a -> unit) = List.iter f repr
```
5.3 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.

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

```ocaml
# class ['observer, 'event] subject =
#   object
#     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;;
class ['a, 'b] subject :
  object ('c)
    constraint 'a = < notify : 'c -> 'b -> unit; .. >
    val mutable observers : 'a list
    method add_observer : 'a -> unit
    method notify_observers : 'b -> unit
  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.

```ocaml
# type event = Raise | Resize | Move;;
type event = Raise | Resize | Move
# let string_of_event = function
#   Raise -> "Raise" | Resize -> "Resize" | Move -> "Move";;
val string_of_event : event -> string = <fun>
```
# let count = ref 0;;
val count : int ref = {contents = 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 ['a] window_subject :
  object ('b)
    constraint 'a = < notify : 'b -> event -> unit; .. >
    val mutable observers : 'a list
    val mutable position : int
    method add_observer : 'a -> unit
    method draw : unit
    method identity : int
    method move : int -> unit
    method notify_observers : event -> unit
  end

# class ['subject] window_observer =
#   object
#     inherit ['subject, event] observer
#     method notify s e = s#draw
#   end;;
class ['a] window_observer :
  object
    constraint 'a = < draw : unit; .. >
    method notify : 'a -> event -> unit
  end

Unsurprisingly the type of window is recursive.

# let window = new window_subject;;
val window : < notify : 'a -> event -> unit; _.. > window_subject as 'a = <obj>

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

# let window_observer = new window_observer;;
val window_observer : < draw : unit; _.. > window_observer = <obj>

# window#add_observer window_observer;;
- : unit = ()

# window#move 1;;
{Position = 1}
- : unit = ()
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.

```ocaml
# 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 ['a] richer_window_observer :
  object
    constraint 'a = < draw : unit; raise : unit; .. >
    val mutable observers : 'a list
    val mutable position : int
    val mutable size : int
    val mutable top : bool
    method add_observer : 'a -> unit
    method draw : unit
    method identity : int
    method move : int -> unit
    method notify_observers : event -> unit
    method raise : unit
    method resize : int -> unit
  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;

class ['a] richer_window_observer :
  object
    constraint 'a = < draw : unit; raise : unit; .. >
    method notify : 'a -> event -> unit
  end

We can also create a different kind of observer:

```ocaml
# 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;

class ['a] trace_observer :
  object
constraint 'a = < identity : int; .. >
method notify : 'a -> event -> unit
end

and attached several observers to the same object:

# let window = new richer_window_subject;;
val window : < notify : 'a -> event -> unit; _.. > richer_window_subject as 'a = <obj>

# window#add_observer (new richer_window_observer);;
- : unit = ()

# window#add_observer (new trace_observer);;
- : unit = ()

# window#move 1; window#resize 2;;
<Window 1 <== Move>
<Window 1 <== Raise>
{Position = 1; Size = 1}
{Position = 1; Size = 1}
<Window 1 <== Resize>
<Window 1 <== Raise>
{Position = 1; Size = 3}
{Position = 1; Size = 3}
- : unit = ()
Part II

The OCaml language
Chapter 6

The OCaml language

Foreword

This document is intended as a reference manual for the OCaml 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 (likethat). Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components. Curly bracket with a trailing plus sign {...}^+ denote one or several repetitions of the enclosed components. Parentheses (...) denote grouping.

6.1 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

\[
\text{ident} ::= (\text{letter} | \_ ) \{ \text{letter} | 0 \ldots 9 | \_ | ' \}
\]

\[
\text{letter} ::= A \ldots Z | a \ldots z
\]

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. All characters in an identifier are meaningful. The current implementation accepts identifiers up to 16000000 characters in length.

Integer literals

\[
\text{integer-literal} ::= [-] (0 \ldots 9) \{0 \ldots 9 | \_\}
\]

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:

<table>
<thead>
<tr>
<th>Prefix</th>
<th>Radix</th>
</tr>
</thead>
<tbody>
<tr>
<td>0x, 0X</td>
<td>hexadecimal (radix 16)</td>
</tr>
<tr>
<td>0o, 0O</td>
<td>octal (radix 8)</td>
</tr>
<tr>
<td>0b, 0B</td>
<td>binary (radix 2)</td>
</tr>
</tbody>
</table>

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

For convenience and readability, underscore characters (\_) are accepted (and ignored) within integer literals.

Floating-point literals

\[
\text{float-literal} ::= [-] (0 \ldots 9) \{0 \ldots 9 | \_\} \{ . \{0 \ldots 9 | \_\}\} [(e | E) [+ | -] (0 \ldots 9) \{0 \ldots 9 | \_\}]
\]

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.

For convenience and readability, underscore characters (\_) are accepted (and ignored) within floating-point literals.
Character literals

char-literal ::= ' regular-char ' |
                  ' escape-sequence ' 

escape-sequence ::= \ ( \ | " | n | t | b | r ) |
                      \ ( 0...9 ) ( 0...9 ) ( 0...9 ) |
                      \x ( 0...9 | A...F | a...f ) ( 0...9 | A...F | a...f )

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:

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Character denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>\</td>
<td>backslash ()</td>
</tr>
<tr>
<td>&quot;</td>
<td>double quote (&quot;)</td>
</tr>
<tr>
<td>'</td>
<td>single quote (?)</td>
</tr>
<tr>
<td>n</td>
<td>linefeed (LF)</td>
</tr>
<tr>
<td>r</td>
<td>carriage return (CR)</td>
</tr>
<tr>
<td>t</td>
<td>horizontal tabulation (TAB)</td>
</tr>
<tr>
<td>b</td>
<td>backspace (BS)</td>
</tr>
<tr>
<td>space</td>
<td>space (SPC)</td>
</tr>
<tr>
<td>ddd</td>
<td>the character with ASCII code ddd in decimal</td>
</tr>
<tr>
<td>xhh</td>
<td>the character with ASCII code hh in hexadecimal</td>
</tr>
</tbody>
</table>

String literals

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

string-character ::= regular-char-str |
                        escape-sequence

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 given above for character literals.

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 practically no restrictions on the length of string literals.

Naming labels

To avoid ambiguities, naming labels in expressions cannot just be defined syntactically as the sequence of the three tokens ~, ident and :, and have to be defined at the lexical level.

label-name ::= (a...z | _ ) {letter | 0...9 | _ | '}

label ::= ~ label-name :

optlabel ::= ? label-name :
Naming labels come in two flavours: \textit{label} for normal arguments and \textit{optlabel} for optional ones. They are simply distinguished by their first character, either \textasciitilde{} or \texttt{?}. Despite \textit{label} and \textit{optlabel} being lexical entities in expressions, their expansions \textasciitilde{} \textit{label-name} : and \texttt{? label-name} : will be used in grammars, for the sake of readability. Note also that inside type expressions, this expansion can be taken literally, \textit{i.e.} there are really 3 tokens, with optional spaces between them.

\textbf{Prefix and infix symbols}

\begin{verbatim}
infix-symbol ::= (= | < | > | @ | ^ | & | + | * | / | $ | %) \{operator-char\}
prefix-symbol ::= (! | ? | ~) \{operator-char\}
operator-char ::= ! | $ | % | & | * | + | - | . | / | : | < | > | ? | @ | ^ | | |
\end{verbatim}

Sequences of “operator characters”, such as \texttt{<>} or \texttt{!!}, are read as a single token from the \textit{infix-symbol} or \textit{prefix-symbol} class. These symbols are parsed as prefix and infix operators inside expressions, but otherwise behave much as identifiers.

\textbf{Keywords}

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

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

The following character sequences are also keywords:

\begin{verbatim}
!= # & && ' ( ) * + , -
-. -> . . . : :: ::= := ::= >; ; ; ; <
<- = > >]} >]} ? ?? [ [ [< [>] [|] ] _ ` { {< | | |] } ~
\end{verbatim}

Note that the following identifiers are keywords of the Camlp4 extensions and should be avoided for compatibility reasons:

\begin{verbatim}
parser << <: >> $ $$ $:
\end{verbatim}
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.

Line number directives

\[
\text{linenum-directive ::= } \# \{0 \ldots 9\}^+ \\
| \# \{0 \ldots 9\}^+ \, \{\text{string-character}\} \\
\]

Preprocessors that generate Caml source code can insert line number directives in their output so that error messages produced by the compiler contain line numbers and file names referring to the source file before preprocessing, instead of after preprocessing. A line number directive is composed of a # (sharp sign), followed by a positive integer (the source line number), optionally followed by a character string (the source file name). Line number directives are treated as blank characters during lexical analysis.

6.2 Values

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

6.2.1 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 \(2^{24} - 5\) characters (16777211 characters); on 64-bit platforms, the limit is \(2^{57} - 9\).
6.2.2 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).

6.2.3 Records

Record values are labeled tuples of values. The record value written \(\{field_1 = v_1; \ldots; field_n = v_n\}\) associates the value \(v_i\) to the record field \(field_i\), for \(i = 1 \ldots n\). The current implementation supports records with up to \(2^{22} - 1\) fields (4194303 fields).

6.2.4 Arrays

Arrays are finite, variable-sized sequences of values of the same type. The current implementation supports arrays containing up to \(2^{22} - 1\) elements (4194303 elements) unless the elements are floating-point numbers (2097151 elements in this case); on 64-bit platforms, the limit is \(2^{54} - 1\) for all arrays.

6.2.5 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 \(\text{constr}\); the latter case is written \(\text{constr}(v)\), where \(v\) is said to be the argument of the non-constant constructor \(\text{constr}\).

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

<table>
<thead>
<tr>
<th>Constant</th>
<th>Constructor</th>
</tr>
</thead>
<tbody>
<tr>
<td>false</td>
<td>the boolean false</td>
</tr>
<tr>
<td>true</td>
<td>the boolean true</td>
</tr>
<tr>
<td>()</td>
<td>the “unit” value</td>
</tr>
<tr>
<td>[]</td>
<td>the empty list</td>
</tr>
</tbody>
</table>

The current implementation limits each variant type to have at most 246 non-constant constructors.

6.2.6 Polymorphic variants

Polymorphic variants are an alternate form of variant values, not belonging explicitly to a predefined variant type, and following specific typing rules. They can be either constant, written ‘tag-name’, or non-constant, written ‘tag-name\((v)\).

6.2.7 Functions

Functional values are mappings from values to values.

6.2.8 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.
6.3 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 and exception constructors (class `constr-name`),
- labels (`label-name`),
- variant tags (`tag-name`),
- type constructors (`typeconstr-name`),
- record fields (`field-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 eleven 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-op
infix-op ::= infix-symbol
            | * | = | or | & | :=
            | mod | land | lor | lxor | lsl | lsr | asr
constr-name ::= capitalized-ident
label-name ::= lowercase-ident
tag-name ::= capitalized-ident
typeconstr-name ::= lowercase-ident
field-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. The capitalization rules are summarized in the table below.

<table>
<thead>
<tr>
<th>Name space</th>
<th>Case of first letter</th>
</tr>
</thead>
<tbody>
<tr>
<td>Values</td>
<td>lowercase</td>
</tr>
<tr>
<td>Constructors</td>
<td>uppercase</td>
</tr>
<tr>
<td>Labels</td>
<td>lowercase</td>
</tr>
<tr>
<td>Variant tags</td>
<td>uppercase</td>
</tr>
<tr>
<td>Exceptions</td>
<td>uppercase</td>
</tr>
<tr>
<td>Type constructors</td>
<td>lowercase</td>
</tr>
<tr>
<td>Record fields</td>
<td>lowercase</td>
</tr>
<tr>
<td>Classes</td>
<td>lowercase</td>
</tr>
<tr>
<td>Instance variables</td>
<td>lowercase</td>
</tr>
<tr>
<td>Methods</td>
<td>lowercase</td>
</tr>
<tr>
<td>Modules</td>
<td>uppercase</td>
</tr>
<tr>
<td>Module types</td>
<td>any</td>
</tr>
</tbody>
</table>

Note on variant tags: the current implementation accepts lowercase variant tags in addition to uppercase variant tags, but we suggest you avoid lowercase variant tags for portability and compatibility with future OCaml versions.
Referring to named objects

\[
\begin{align*}
\text{value-path} & ::= \text{value-name} \\
& \quad | \text{module-path . value-name} \\
\text{constr} & ::= \text{constr-name} \\
& \quad | \text{module-path . constr-name} \\
\text{typeconstr} & ::= \text{typeconstr-name} \\
& \quad | \text{extended-module-path . typeconstr-name} \\
\text{field} & ::= \text{field-name} \\
& \quad | \text{module-path . field-name} \\
\text{module-path} & ::= \text{module-name} \\
& \quad | \text{module-path . module-name} \\
\text{extended-module-path} & ::= \text{module-name} \\
& \quad | \text{extended-module-path . module-name} \\
& \quad | \text{extended-module-path ( extended-module-path )} \\
\text{modtype-path} & ::= \text{modtype-name} \\
& \quad | \text{extended-module-path . modtype-name} \\
\text{class-path} & ::= \text{class-name} \\
& \quad | \text{module-path . class-name}
\end{align*}
\]

A named object can be referred to either by its name (following the usual static scoping rules for names) or by an access path \text{prefix . name}, where \text{prefix} designates a module and \text{name} is the name of an object defined in that module. The first component of the path, \text{prefix}, is either a simple module name or an access path \text{name}_1 . \text{name}_2 . . . , in case the defining module is itself nested inside other modules. For referring to type constructors or module types, the \text{prefix} can also contain simple functor applications (as in the syntactic class \text{extended-module-path} above), in case the defining module is the result of a functor application.

Label names, tag names, method names and instance variable names need not be qualified: the former three are global labels, while the latter are local to a class.
## 6.4 Type expressions

\[ \text{typexpr} ::= \ ' \text{ident} \]

\[ - \]

\[ ( \text{typexpr} ) \]

\[ [? \label-name : ] \text{typexpr} \rightarrow \text{typexpr} \]

\[ \text{typexpr} \{ \ast \text{typexpr}\}^+ \]

\[ \text{typeconstr} \]

\[ \text{typexpr} \text{typeconstr} \]

\[ ( \text{typexpr} \{ , \text{typexpr} \} ) \text{typeconstr} \]

\[ \text{typexpr as } ' \text{ident} \]

\[ \langle [..] \rangle \]

\[ \langle \text{method-type \{ ; \text{method-type\}} [;..] \rangle \]

\[ \# \text{class-path} \]

\[ \text{typexpr} \# \text{class-path} \]

\[ ( \text{typexpr} \{ , \text{typexpr}\} ) \# \text{class-path} \]

\[ \text{poly-typexpr} ::= \text{typexpr} \]

\[ \{ ' \text{ident}\}^+. \text{typexpr} \]

\[ \text{method-type} ::= \text{method-name} : \text{poly-typexpr} \]

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

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Type constructor application</td>
<td>–</td>
</tr>
<tr>
<td>*</td>
<td>–</td>
</tr>
<tr>
<td>( \rightarrow )</td>
<td>right</td>
</tr>
<tr>
<td>( \text{as} )</td>
<td>–</td>
</tr>
</tbody>
</table>

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

### Type variables

The type expression \( ' \text{ident} \) stands for the type variable named \text{ident}. The type expression \( _- \) stands for an anonymous type variable. 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. In general the scope of a named type variable is the whole enclosing definition; and they can only be generalized when leaving this scope. Anonymous variables have no such restriction. In the following cases, the scope of named type variables is restricted to the type expression where they appear: 1) for universal (explicitly polymorphic) type variables; 2) for type variables that only appear in public method specifications (as those variables will be made universal, as described in section 6.9.1); 3) for variables used as aliases, when the type they are aliased to would be invalid in the scope of the enclosing definition (\textit{i.e.} when it contains free universal type variables, or locally defined types.)
Parenthesized types

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

Function types

The type expression typexpr₁ → typexpr₂ denotes the type of functions mapping arguments of type typexpr₁ to results of type typexpr₂.

label-name : typexpr₁ → typexpr₂ denotes the same function type, but the argument is labeled label.

? label-name : typexpr₁ → typexpr₂ denotes the type of functions mapping an optional labeled argument of type typexpr₁ to results of type typexpr₂. That is, the physical type of the function will be typexpr₁ option → typexpr₂.

Tuple types

The type expression typexpr₁ * ... * typexprₙ denotes the type of tuples whose elements belong to types typexpr₁, ..., typexprₙ 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 (typexpr₁,...,typexprₙ) typeconstr, where typeconstr is a type constructor with n parameters, denotes the application of the n-ary type constructor typeconstr to the types typexpr₁ through typexprₙ.

Aliased and 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 other types. In general the scope of an alias is the same as for a named type variable, and covers the whole enclosing definition. If the type variable ident actually occurs in typexpr, a recursive type is created. Recursive types for which there exists a recursive path that does not contain an object or polymorphic variant type constructor are rejected, except when the -rectypes mode is selected.

If ' ident denotes an explicit polymorphic variable, and typexpr denotes either an object or polymorphic variant type, the row variable of typexpr is captured by ' ident, and quantified upon.

Polymorphic variant types

polymorphic-variant-type ::= [ [ ] tag-spec { 1 tag-spec } ]
                   | [ ] tag-spec { 1 tag-spec } ]
                   | [ ] tag-spec-full { 1 tag-spec-full} [ > { ' tag-name}+ ] ]

  tag-spec ::= ' tag-name [of typexpr] 
                  | typexpr
tag-spec-full ::= 'tag-name [of typexpr] {& typexpr} |
                  typexpr

Polymorphic variant types describe the values a polymorphic variant may take.

The first case is an exact variant type: all possible tags are known, with their associated types,
and they can all be present. Its structure is fully known.

The second case is an open variant type, describing a polymorphic variant value: it gives the
list of all tags the value could take, with their associated types. This type is still compatible with a
variant type containing more tags. A special case is the unknown type, which does not define any
tag, and is compatible with any variant type.

The third case is a closed variant type. It gives information about all the possible tags and their
associated types, and which tags are known to potentially appear in values. The above exact variant
type is just an abbreviation for a closed variant type where all possible tags are also potentially
present.

In all three cases, tags may be either specified directly in the 'tag-name [...] form, or indirectly
through a type expression. In this last case, the type expression must expand to an exact variant
type, whose tag specifications are inserted in its place.

Full specification of variant tags are only used for non-exact closed types. They can be under-
stood as a conjunctive type for the argument: it is intended to have all the types enumerated in
the specification.

Such conjunctive constraints may be unsatisfiable. In such a case the corresponding tag may
not be used in a value of this type. This does not mean that the whole type is not valid: one can
still use other available tags.

Object types

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

Each method may have an explicit polymorphic type: { 'ident }^+. typexpr. Explicit poly-
morphic variables have a local scope, and an explicit polymorphic type can only be unified to an
equivalent one, with polymorphic variables at the same positions.

The type < method-type { ; method-type } ; . . > is the type of an object with methods and
their associated types are described by method-type_1, . . . , method-type_n, and possibly some other
methods represented by the ellipsis. This ellipsis actually is a special kind of type variable (also
called row variable in the literature) that stands for any number of extra method types.

#-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.
Use of # types to abbreviate polymorphic variant types is deprecated. If \( t \) is an exact variant type then \#t translates to \([< t] \), and \#t[‘tag\_1...‘tag\_k] translates to \([< t> ‘tag\_1...‘tag\_k] \).

**Variant and record types**

There are no type expressions describing (defined) variant types nor record types, since those are always named, i.e. defined before use and referred to by name. Type definitions are described in section \[6.8.1\].

### 6.5 Constants

\[
\text{constant ::= integer-literal} \\
\text{        | float-literal} \\
\text{        | char-literal} \\
\text{        | string-literal} \\
\text{        | constr} \\
\text{        | false} \\
\text{        | true} \\
\text{        | []} \\
\text{        | ()} \\
\text{        | ‘ tag-name}
\]

The syntactic class of constants comprises literals from the four base types (integers, floating-point numbers, characters, character strings), and constant constructors from both normal and polymorphic variants, as well as the special constants false, true, [], and (), which behave like constant constructors.

### 6.6 Patterns

\[
\text{pattern ::= value-name} \\
\text{ | _} \\
\text{ | constant} \\
\text{ | pattern as value-name} \\
\text{ | ( pattern )} \\
\text{ | ( pattern : typexpr )} \\
\text{ | pattern | pattern} \\
\text{ | constr pattern} \\
\text{ | ‘ tag-name pattern} \\
\text{ | # typeconstr-name} \\
\text{ | pattern {, pattern}} \\
\text{ | { field = pattern {; field = pattern} } } \\
\text{ | [ pattern {; pattern} ] } \\
\text{ | pattern :: pattern} \\
\text{ | [1 pattern {; pattern} |]} \\
\text{ | lazy pattern}
\]
The table below shows the relative precedences and associativity of operators and non-closed pattern constructions. The constructions with higher precedences come first.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>Constructor application</td>
<td>–</td>
</tr>
<tr>
<td>::</td>
<td>right</td>
</tr>
<tr>
<td>,</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td>left</td>
</tr>
<tr>
<td>as</td>
<td>–</td>
</tr>
</tbody>
</table>

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 `typexpr`.

**“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 bind exactly the same identifiers to values having the same types. Matching is performed from left to right. More precisely, in case some value `v` matches `pattern_1 | pattern_2`, the bindings performed are those of `pattern_1` when `v` matches `pattern_1`. Otherwise, value `v` matches `pattern_2` whose bindings are performed.
Chapter 6. The OCaml language

Variant patterns

The pattern `constr pattern_1` matches all variants whose constructor is equal to `constr`, 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`.

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 :: []`.

Polymorphic variant patterns

The pattern `tag-name pattern_1` matches all polymorphic variants whose tag is equal to `tag-name`, and whose argument matches `pattern_1`.

Variant abbreviation patterns

If the type `[(‘a,’b,...)] typeconstr = [‘tag_1 typexpr_1 | ... | ‘tag_n typexpr_n ]` is defined, then the pattern `#typeconstr` is a shorthand for the or-pattern `(‘tag_1 (_ : typexpr_1 ) | ... | ‘tag_n (_ : typexpr_n ))`. It matches all values of type `#typeconstr`.

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,...,v_n)` such that `pattern_i` matches `v_i` for `i = 1,...,n`.

Record patterns

The pattern `{ field_1 = pattern_1 ; ... ; field_n = pattern_n }` matches records that define at least the fields `field_1` through `field_n`, and such that the value associated to `field_i` matches the pattern `pattern_i`, for `i = 1,...,n`. The record value can define more fields than `field_1` ... `field_n`; the values associated to these extra fields are not taken into account for matching.

Array patterns

The pattern `[ | pattern_1 ; ... ; pattern_n | ]` matches arrays of length `n` such that the `i`-th array element matches the pattern `pattern_i`, for `i = 1,...,n`. 
6.7 Expressions

\[ \text{expr} ::= \text{value-path} \]
\[ \text{constant} \]
\[ ( \text{expr} ) \]
\[ \text{begin expr end} \]
\[ ( \text{expr} : \text{typexpr} ) \]
\[ \text{expr , expr \{ , expr\}} \]
\[ \text{constr expr} \]
\[ \text{'} tag-name expr \]
\[ \text{expr :: expr} \]
\[ [ \text{expr \{ ; expr\}} ] \]
\[ [\text{expr \{ ; expr\} |] \]
\[ \text{\{ field = expr \{ ; field = expr\}} \}
\[ \text{\{ expr with field = expr \{ ; field = expr\}} \}
\[ \text{expr \{argument\} }^+ \]
\[ \text{prefix-symbol expr} \]
\[ \text{expr infix-op expr} \]
\[ \text{expr . field} \]
\[ \text{expr . field \text{x<- expr} \]
\[ \text{expr .( expr )} \]
\[ \text{expr .( expr ) '\text{x<- expr} \]
\[ \text{expr .[]} \]
\[ \text{expr .[]} '\text{x<- expr} \]
\[ \text{if expr then expr [else expr]} \]
\[ \text{while expr do expr done} \]
\[ \text{for ident = expr (to | downto) expr do expr done} \]
\[ \text{expr ; expr} \]
\[ \text{match expr with pattern-matching} \]
\[ \text{function pattern-matching} \]
\[ \text{fun multiple-matching} \]
\[ \text{try expr with pattern-matching} \]
\[ \text{let [rec] let-binding \{and let-binding\} in expr} \]
\[ \text{new class-path} \]
\[ \text{object class-body end} \]
\[ \text{expr \# method-name} \]
\[ \text{inst-var-name} \]
\[ \text{inst-var-name \text{x<- expr} \]
\[ ( \text{expr : typexprpr}) \]
\[ ( \text{expr : typexpr :> typexpr} ) \]
\[ \{\text{< inst-var-name = expr \{ ; inst-var-name = expr\} >}\}
\[ \text{assert expr} \]
\[ \text{lazy expr} \]
argument ::= expr | ~ label-name | ~ label-name : expr | ? label-name | ? label-name : expr

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

multiple-matching ::= { (parameter)* [when expr] -> expr

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

parameter ::= pattern | ~ label-name | ~ (label-name [: typexpr]) | ~ label-name : pattern | ? label-name | ? (label-name [: typexpr] [= expr]) | ? label-name : pattern | ? label-name : (pattern [: typexpr] [= expr])

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 *”.

<table>
<thead>
<tr>
<th>Construction or operator</th>
<th>Associativity</th>
</tr>
</thead>
<tbody>
<tr>
<td>prefix-symbol</td>
<td>-</td>
</tr>
<tr>
<td>. . ( . [ function application, constructor application, assert, lazy</td>
<td>-</td>
</tr>
<tr>
<td>- - . (prefix)</td>
<td>left</td>
</tr>
<tr>
<td>**... lsl lsr asr</td>
<td>right</td>
</tr>
<tr>
<td>*... / ... % ... mod land lor lxor</td>
<td>left</td>
</tr>
<tr>
<td>+ ... - ...</td>
<td>left</td>
</tr>
<tr>
<td>; :</td>
<td>right</td>
</tr>
<tr>
<td>@ ... ^ ...</td>
<td>right</td>
</tr>
<tr>
<td>= ... &lt; ... &gt; ...</td>
<td>... &amp; ... $ ...</td>
</tr>
<tr>
<td>&amp; &amp; &amp;</td>
<td>right</td>
</tr>
<tr>
<td>or</td>
<td></td>
</tr>
<tr>
<td>,</td>
<td>-</td>
</tr>
<tr>
<td>&lt;= :=</td>
<td>right</td>
</tr>
<tr>
<td>if</td>
<td>-</td>
</tr>
<tr>
<td>;</td>
<td>right</td>
</tr>
<tr>
<td>let match fun function try</td>
<td>-</td>
</tr>
</tbody>
</table>
6.7.1 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. The two 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 : typexpr ). This constraint forces the type of expr to be compatible with typexpr.

Parenthesized expressions can also contain coercions ( expr [ : typexpr ] :: typexpr ) (see subsection 6.7.6 below).

Function application

Function application is denoted by juxtaposition of (possibly labeled) expressions. The expression expr argument\_1 ... argument\_n evaluates the expression expr and those appearing in argument\_1 to argument\_n. The expression expr must evaluate to a functional value f, which is then applied to the values of argument\_1,..., argument\_n.

The order in which the expressions expr, argument\_1,..., argument\_n are evaluated is not specified.

Arguments and parameters are matched according to their respective labels. Argument order is irrelevant, except among arguments with the same label, or no label.

If a parameter is specified as optional (label prefixed by ?) in the type of expr, the corresponding argument will be automatically wrapped with the constructor Some, except if the argument itself is also prefixed by ?, in which case it is passed as is. If a non-labeled argument is passed, and its corresponding parameter is preceded by one or several optional parameters, then these parameters are defaulted, i.e. the value None will be passed for them. All other missing parameters (without corresponding argument), both optional and non-optional, will be kept, and the result of the function will still be a function of these missing parameters to the body of f.

As a special case, if the function has a known arity, all the arguments are unlabeled, and their number matches the number of non-optional parameters, then labels are ignored and non-optional parameters are matched in their definition order. Optional arguments are defaulted.

In all cases but exact match of order and labels, without optional parameters, the function type should be known at the application point. This can be ensured by adding a type constraint. Principality of the derivation can be checked in the -principal mode.
Function definition

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

\[
\text{function} \quad \text{pattern}_1 \rightarrow \text{expr}_1 \\
| \quad \ldots \\
| \quad \text{pattern}_n \rightarrow \text{expr}_n
\]

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 \(\text{pattern}_1\) to \(\text{pattern}_n\). If one of these matchings succeeds, that is, if the value \(v\) matches the pattern \(\text{pattern}_i\) for some \(i\), then the expression \(\text{expr}_i\) associated to the selected pattern is evaluated, and its value becomes the value of the function application. The evaluation of \(\text{expr}_i\) 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`:

\[
\text{fun} \quad \text{parameter}_1 \ldots \text{parameter}_n \rightarrow \text{expr}
\]

This expression is equivalent to:

\[
\text{fun} \quad \text{parameter}_1 \rightarrow \ldots \text{fun} \quad \text{parameter}_n \rightarrow \text{expr}
\]

The parameter patterns `~var` and `~(var [: typexpr ] )` are shorthands for respectively `~var:var` and `~var:(var [: typexpr] )`, and similarly for their optional counterparts.

Functions of the form `fun ?lab: ( pattern = expr_0 ) \rightarrow \text{expr}` are equivalent to

\[
\text{fun} \quad ?\text{lab}: \quad \text{ident} \rightarrow \text{let} \quad \text{pattern} = \text{match} \quad \text{ident} \quad \text{with} \quad \text{Some} \quad \text{ident} \rightarrow \text{ident} \quad | \quad \text{None} \rightarrow \text{expr}_0 \quad \text{in} \quad \text{expr}
\]

where `ident` is a fresh variable. When `expr_0` will be evaluated is left unspecified.

After these two transformations, expressions are of the form

\[
\text{fun} \quad \text{[label}_1\text{]} \quad \text{pattern}_1 \rightarrow \ldots \text{fun} \quad \text{[label}_n\text{]} \quad \text{pattern}_n \rightarrow \text{expr}
\]

If we ignore labels, which will only be meaningful at function application, this is equivalent to

\[
\text{function} \quad \text{pattern}_1 \rightarrow \ldots \text{function} \quad \text{pattern}_n \rightarrow \text{expr}
\]

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 \(\text{pattern}_1 \ldots \text{pattern}_n\). If the matching succeeds, the function returns the value of \(\text{expr}\) 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
  | ...
  | pattern_n [when cond_n] -> expr_n
```

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 pattern1 = expr1 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 parameter_1 ... parameter_m -> expr
```
in a let expression, one may instead write

```
let ident parameter_1 ... parameter_m = expr
```

Recursive definitions of names are introduced by let rec:

```
let rec pattern1 = expr1 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, as explained in section 7.3.
6.7.2 Control structures

Sequence

The expression \( \text{expr}_1 ; \text{expr}_2 \) evaluates \( \text{expr}_1 \) first, then \( \text{expr}_2 \), and returns the value of \( \text{expr}_2 \).

Conditional

The expression \( \text{if } \text{expr}_1 \text{ then } \text{expr}_2 \text{ else } \text{expr}_3 \) evaluates to the value of \( \text{expr}_2 \) if \( \text{expr}_1 \) evaluates to the boolean \( \text{true} \), and to the value of \( \text{expr}_3 \) if \( \text{expr}_1 \) evaluates to the boolean \( \text{false} \).

The \text{else } \text{expr}_3 \) part can be omitted, in which case it defaults to \text{else } ().

Case expression

The expression

\[
\text{match } \text{expr} \\
\text{with } \text{pattern}_1 -> \text{expr}_1 \\
| ... \\
| \text{pattern}_n -> \text{expr}_n
\]

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

Boolean operators

The expression \( \text{expr}_1 && \text{expr}_2 \) evaluates to \( \text{true} \) if both \( \text{expr}_1 \) and \( \text{expr}_2 \) evaluate to \( \text{true} \); otherwise, it evaluates to \( \text{false} \). The first component, \( \text{expr}_1 \), is evaluated first. The second component, \( \text{expr}_2 \), is not evaluated if the first component evaluates to \( \text{false} \). Hence, the expression \( \text{expr}_1 && \text{expr}_2 \) behaves exactly as

\[ \text{if } \text{expr}_1 \text{ then } \text{expr}_2 \text{ else } \text{false}. \]

The expression \( \text{expr}_1 || \text{expr}_2 \) evaluates to \( \text{true} \) if one of \( \text{expr}_1 \) and \( \text{expr}_2 \) evaluates to \( \text{true} \); otherwise, it evaluates to \( \text{false} \). The first component, \( \text{expr}_1 \), is evaluated first. The second component, \( \text{expr}_2 \), is not evaluated if the first component evaluates to \( \text{true} \). Hence, the expression \( \text{expr}_1 || \text{expr}_2 \) behaves exactly as

\[ \text{if } \text{expr}_1 \text{ then } \text{true} \text{ else } \text{expr}_2. \]

The boolean operator \& is synonymous for \&\&. The boolean operator \text{or} is synonymous for \|.
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 pattern_1 -> expr_1
| ...
| pattern_n -> expr_n
```

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_1` succeeds, the associated expression `expr_1` is evaluated, and its value becomes the value of the whole `try` expression. The evaluation of `expr_1` 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.

6.7.3 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 `constr expr` evaluates to the variant value whose constructor is `constr`, 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`. 
Polymorphic variants

The expression `tag-name expr` evaluates to the polymorphic variant value whose tag is `tag-name`, and whose argument is the value of `expr`.

Records

The expression `{ field_1 = expr_1 ; ... ; field_n = expr_n }` evaluates to the record value `{field_1 = v_1; ... ; field_n = v_n}` where \( v_i \) is the value of `expr_i` for \( i = 1, ..., n \). The fields `field_1` to `field_n` must all belong to the same record types; all fields 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 with field_1 = expr_1 ; ... ; field_n = expr_n }` builds a fresh record with fields `field_1`...`field_n` equal to `expr_1`...`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 `field_1`...`field_n`, which are initialized to `expr_1`...`expr_n`.

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

The expression `expr_1 . field <- expr_2` evaluates `expr_1` to a record value, which is then modified in-place by replacing the value associated to `field` in this record by the value of `expr_2`. This operation is permitted only if `field` has been declared `mutable` in the definition of the record type. The whole expression `expr_1 . field <- 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 `()`.
6.7.4 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 `expr₁ infix-symbol expr₂` is interpreted as the application `(infix-symbol) expr₁ expr₂`.

The table below lists the symbols defined in the initial environment and their initial meaning. (See the description of the core library module `Pervasives` in chapter 19 for more details). Their meaning may be changed at any time using `let (infix-op) name₁ name₂ = ...`.

<table>
<thead>
<tr>
<th>Operator</th>
<th>Initial meaning</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>+</code></td>
<td>Integer addition.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>(infix) Integer subtraction.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>(prefix) Integer negation.</td>
</tr>
<tr>
<td><code>*</code></td>
<td>Integer multiplication.</td>
</tr>
<tr>
<td><code>/</code></td>
<td>Integer division. Raise <code>Division_by_zero</code> if second argument is zero.</td>
</tr>
<tr>
<td><code>mod</code></td>
<td>Integer modulus. Raise <code>Division_by_zero</code> if second argument is zero.</td>
</tr>
<tr>
<td><code>land</code></td>
<td>Bitwise logical “and” on integers.</td>
</tr>
<tr>
<td><code>lor</code></td>
<td>Bitwise logical “or” on integers.</td>
</tr>
<tr>
<td><code>lxor</code></td>
<td>Bitwise logical “exclusive or” on integers.</td>
</tr>
<tr>
<td><code>lsl</code></td>
<td>Bitwise logical shift left on integers.</td>
</tr>
<tr>
<td><code>lsr</code></td>
<td>Bitwise logical shift right on integers.</td>
</tr>
<tr>
<td><code>asr</code></td>
<td>Bitwise arithmetic shift right on integers.</td>
</tr>
<tr>
<td><code>+.</code></td>
<td>Floating-point addition.</td>
</tr>
<tr>
<td><code>-.</code></td>
<td>(infix) Floating-point subtraction.</td>
</tr>
<tr>
<td><code> -.</code></td>
<td>(prefix) Floating-point negation.</td>
</tr>
<tr>
<td><code>*.</code></td>
<td>Floating-point multiplication.</td>
</tr>
<tr>
<td><code>/.</code></td>
<td>Floating-point division.</td>
</tr>
<tr>
<td><code>**</code></td>
<td>Floating-point exponentiation.</td>
</tr>
<tr>
<td><code>@</code></td>
<td>List concatenation.</td>
</tr>
<tr>
<td><code>-</code></td>
<td>String concatenation.</td>
</tr>
<tr>
<td><code>!</code></td>
<td>Dereferencing (return the current contents of a reference).</td>
</tr>
<tr>
<td><code>:</code></td>
<td>Reference assignment (update the reference given as first argument with the value of the second argument).</td>
</tr>
<tr>
<td><code>=</code></td>
<td>Structural equality test.</td>
</tr>
<tr>
<td><code>&lt;&gt;</code></td>
<td>Structural inequality test.</td>
</tr>
<tr>
<td><code>==</code></td>
<td>Physical equality test.</td>
</tr>
<tr>
<td><code>!=</code></td>
<td>Physical inequality test.</td>
</tr>
<tr>
<td><code>&lt;</code></td>
<td>Test “less than”.</td>
</tr>
<tr>
<td><code>&lt;=</code></td>
<td>Test “less than or equal”.</td>
</tr>
<tr>
<td><code>&gt;</code></td>
<td>Test “greater than”.</td>
</tr>
<tr>
<td><code>&gt;=</code></td>
<td>Test “greater than or equal”.</td>
</tr>
</tbody>
</table>
6.7.5 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.

Immediate object creation

Creating directly an object through the `object class-body end` construct is operationally equivalent to defining locally a `class class-name = object class-body end`—see sections 6.9.2 and following for the syntax of `class-body`—and immediately creating a single object from it by `new class-name`.

The typing of immediate objects is slightly different from explicitly defining a class in two respects. First, the inferred object type may contain free type variables. Second, since the class body of an immediate object will never be extended, its self type can be unified with a closed object type.

Message sending

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

If `method-name` is a polymorphic method, its type should be known at the invocation site. This is true for instance if `expr` is the name of a fresh object (`let ident = new class-path...`) or if there is a type constraint. Principality of the derivation can be checked in the `-principal` mode.

Accessing and modifying instance variables

The instance variables of a class are visible only in the body of the methods defined in the same class or a class that inherits from the class defining the instance variables. The expression `inst-var-name` evaluates to the value of the given instance variable. The expression `inst-var-name <- expr` assigns the value of `expr` to the instance variable `inst-var-name`, which must be mutable. The whole expression `inst-var-name <- expr` evaluates to `()`. Object duplication

An object can be duplicated using the library function `Oo.copy` (see section 20.21). 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.

6.7.6 Coercions

Expressions whose type contains object or polymorphic variant types can be explicitly coerced (weakened) to a supertype. The expression `( expr :> typexpr )` coerces the expression `expr` to type `typexpr`. The expression `( expr : typexpr1 :> typexpr2 )` coerces the expression `expr` from type `typexpr1` to type `typexpr2`. 


The former operator will sometimes fail to coerce an expression \( \text{expr} \) from a type \( t_1 \) to a type \( t_2 \) even if type \( t_1 \) is a subtype of type \( t_2 \): in the current implementation it only expands two levels of type abbreviations containing objects and/or polymorphic variants, keeping only recursion when it is explicit in the class type (for objects). As an exception to the above algorithm, if both the inferred type of \( \text{expr} \) and \( \text{typexpr} \) are ground (i.e. do not contain type variables), the former operator behaves as the latter one, taking the inferred type of \( \text{expr} \) as \( \text{typexpr}_1 \). In case of failure with the former operator, the latter one should be used.

It is only possible to coerce an expression \( \text{expr} \) from type \( \text{typexpr}_1 \) to type \( \text{typexpr}_2 \), if the type of \( \text{expr} \) is an instance of \( \text{typexpr}_1 \) (like for a type annotation), and \( \text{typexpr}_1 \) is a subtype of \( \text{typexpr}_2 \). The type of the coerced expression is an instance of \( \text{typexpr}_2 \). If the types contain variables, they may be instanciated by the subtyping algorithm, but this is only done after determining whether \( \text{typexpr}_1 \) is a potential subtype of \( \text{typexpr}_2 \). This means that typing may fail during this latter unification step, even if some instance of \( \text{typexpr}_1 \) is a subtype of some instance of \( \text{typexpr}_2 \). In the following paragraphs we describe the subtyping relation used.

**Object types**

A fixed object type admits as subtype any object type including all its methods. The types of the methods shall be subtypes of those in the supertype. Namely,

\[
< \text{met}_1 : \text{typ}_1 ; \ldots ; \text{met}_n : \text{typ}_n >
\]

is a supertype of

\[
< \text{met}_1 : \text{typ}_1' ; \ldots ; \text{met}_n : \text{typ}_n' ; \text{met}_{n+1} : \text{typ}_{n+1}' ; \ldots ; \text{met}_{n+m} : \text{typ}_{n+m}'[; \ldots] >
\]

which may contain an ellipsis \( \ldots \), if every \( \text{typ}_i \) is a supertype of \( \text{typ}_i' \).

A monomorphic method type can be a supertype of a polymorphic method type. Namely, if \( \text{typ} \) is an instance of \( \text{typ}' \), then \( 'a_1\ldots'a_n.\text{typ}' \) is a subtype of \( \text{typ} \).

Inside a class definition, newly defined types are not available for subtyping, as the type abbreviations are not yet completely defined. There is an exception for coercing \( \text{self} \) to the (exact) type of its class: this is allowed if the type of \( \text{self} \) does not appear in a contravariant position in the class type, i.e. if there are no binary methods.

**Polymorphic variant types**

A polymorphic variant type \( \text{typ} \) is subtype of another polymorphic variant type \( \text{typ}' \) if the upper bound of \( \text{typ} \) (i.e. the maximum set of constructors that may appear in an instance of \( \text{typ} \)) is included in the lower bound of \( \text{typ}' \), and the types of arguments for the constructors of \( \text{typ} \) are subtypes of those in \( \text{typ}' \). Namely,

\[
[[<] 'C_1 \text{ of } \text{typ}_1 | \ldots | 'C_n \text{ of } \text{typ}_n ]
\]

which may be a shrinkable type, is a subtype of

\[
[[>] 'C_1 \text{ of } \text{typ}_1' | \ldots | 'C_n \text{ of } \text{typ}_n'| 'C_{n+1} \text{ of } \text{typ}_{n+1}' | \ldots | 'C_{n+m} \text{ of } \text{typ}_{n+m}' ]
\]

which may be an extensible type, if every \( \text{typ}_i \) is a subtype of \( \text{typ}_i' \).
Variance

Other types do not introduce new subtyping, but they may propagate the subtyping of their arguments. For instance, \( \text{typ}_1 \ast \text{typ}_2 \) is a subtype of \( \text{typ}'_1 \ast \text{typ}'_2 \) when \( \text{typ}_1 \) and \( \text{typ}_2 \) are respectively subtypes of \( \text{typ}'_1 \) and \( \text{typ}'_2 \). For function types, the relation is more subtle: \( \text{typ}_1 \to \text{typ}_2 \) is a subtype of \( \text{typ}'_1 \to \text{typ}'_2 \) if \( \text{typ}_1 \) is a supertype of \( \text{typ}'_1 \) and \( \text{typ}_2 \) is a subtype of \( \text{typ}'_2 \). For this reason, function types are covariant in their second argument (like tuples), but contravariant in their first argument. Mutable types, like \text{array} or \text{ref} are neither covariant nor contravariant, they are nonvariant, that is they do not propagate subtyping.

For user defined types, the variance is automatically inferred: a parameter is covariant if it has only covariant occurrences, contravariant if it has only contravariant occurrences, variance-free if it has no occurrences, and nonvariant otherwise. A variance-free parameter may change freely through subtyping, it does not have to be a subtype or a supertype. For abstract and private types, the variance must be given explicitly, otherwise the default is nonvariant. This is also the case for constrained arguments in type definitions.

6.8 Type and exception definitions

6.8.1 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 fields associated with the definition.

type-definition ::= type typedef \{ and typedef \}
typedef ::= [type-params] typeconstr-name [type-information]
type-information ::= [type-equation] [type-representation] \{ type-constraint \}
type-equation ::= = typexpr
type-representation ::= = constr-decl \{ | constr-decl \}
 | = \{ field-decl ; field-decl \}
type-params ::= type-param
 | ( type-param \{ , type-param \} )
type-param ::= \' ident
 | + \' ident
 | - \' ident
constr-decl ::= constr-name
 | constr-name of typexpr \{ * typexpr \}
field-decl ::= field-name : poly-typexpr
 | mutable field-name : poly-typexpr
type-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 (`ident_1, ..., ident_n`), for type constructors with several
parameters. Each type parameter may be prefixed by a variance constraint `+` (resp. `-`) indicating
that the parameter is covariant (resp. contravariant). These type parameters can appear in the type
expressions of the right-hand side of the definition, restricted eventually by a variance constraint;
* i.e. a covariant parameter may only appear on the right side of a functional arrow (more precisely,
  follow the left branch of an even number of arrows), and a contravariant parameter only the left side
  (left branch of an odd number of arrows). If the type has either a representation or an equation,
  and the parameter is free (i.e. not bound via a type constraint to a constructed type), its variance
  constraint is checked but subtyping etc. will use the inferred variance of the parameter, which may
  be better; otherwise (i.e. for abstract types or non-free parameters), the variance must be given
  explicitly, and the parameter is invariant if no variance was given.

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 fields (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 `constr-decl_1, ..., constr-decl_n` describe the constructors associated to this variant type.
The constructor declaration `constr-name of typexpr_1, ..., typexpr_n` declares the name `constr-name`
as a non-constant constructor, whose arguments have types `typexpr_1, ..., typexpr_n`. The constructor
declaration `constr-name` declares the name `constr-name` as a constant constructor. Constructor
names must be capitalized.

The type representation `= { field-decl { ; field-decl } }` describes a record type. The field
declarations `field-decl_1, ..., field-decl_n` describe the fields associated to this record type. The field
declaration `field-name : poly-typexpr` declares `field-name` as a field whose argument has type
`poly-typexpr`. The field declaration `mutable field-name : poly-typexpr` behaves similarly; in addi-
tion, it allows physical modification over the argument to this field. Immutable fields are covariant,
but mutable fields are neither covariant nor contravariant. Both mutable and immutable field
may have an explicitly polymorphic type. The polymorphism of the contents is statically checked
whenever a record value is created or modified. Extracted values may have their types instanciated.

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 con-
structor, 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 fields, 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 fields 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 fields, in the same order, with the same arguments.

The type variables appearing as type parameters can optionally be prefixed by + or - to indicate that the type constructor is covariant or contravariant with respect to this parameter. This variance information is used to decide subtyping relations when checking the validity of :> coercions (see section [6.7.6]).

For instance, `type +'a t` declares `t` as an abstract type that is covariant in its parameter; this means that if the type `τ` is a subtype of the type `σ`, then `τ t` is a subtype of `σ t`. Similarly, `type -'a t` declares that the abstract type `t` is contravariant in its parameter: if `τ` is subtype of `σ`, then `σ t` is subtype of `τ t`. If no + or - variance annotation is given, the type constructor is assumed invariant in the corresponding parameter. For instance, the abstract type declaration `type 'a t` means that `τ t` is neither a subtype nor a supertype of `σ t` if `τ` is subtype of `σ`.

The variance indicated by the + and - annotations on parameters are required only for abstract types. For abbreviations, variant types or record types, the variance properties of the type constructor are inferred from its definition, and the variance annotations are only checked for conformance with the definition.

The construct `constraint 'ident = typexpr` allows to specify type parameters. Any actual type argument corresponding to the type parameter `ident` has 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.

### 6.8.2 Exception definitions

```plaintext
exception-definition ::= exception constr-name [of typexpr { * typexpr}]
| exception constr-name = constr
```

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.

The form `exception constr-name [of typexpr { * typexpr}]` generates a new exception, distinct from all other exceptions in the system. The form `exception constr-name = constr` gives an alternate name to an existing exception.
6.9 Classes

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

6.9.1 Class types

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

\[
\text{class-type} ::= \text{class-body-type} \\
\quad | \left[\left[ \text{label-name} : \text{typexpr} \rightarrow \text{class-type} \right]\right]
\]

\[
\text{class-body-type} ::= \text{object} \left[\left( \text{typexpr} \right)\right] \{ \text{class-field-spec} \} \text{ end} \\
\quad | \text{class-path} \\
\quad | \left[ \left\{ \text{typexpr}, \ldots, \text{typexpr} \right\} \right] \text{class-path}
\]

\[
\text{class-field-spec} ::= \text{inherit} \text{ class-type} \\
\quad | \text{val} \left[\left[ \text{mutable} \right| \text{virtual} \right] \text{inst-var-name} : \text{typexpr} \\
\quad | \text{method} \left[\left[ \text{private} \right| \right] \text{method-name} : \text{poly-typexpr} \\
\quad | \text{method} \left[\left[ \text{private} \right| \text{virtual} \right] \text{method-name} : \text{poly-typexpr} \\
\quad | \text{constraint} \text{typexpr} = \text{typexpr}
\]

Simple class expressions

The expression class-path is equivalent to the class type bound to the name class-path. Similarly, the expression \([\text{typexpr}_1, \ldots, \text{typexpr}_n]\) 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\(_1\), \ldots, typexpr\(_n\).

Class function type

The class type expression typexpr \(\rightarrow\) 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 \([\left( \text{typexpr} \right)]\) \{class-field-spec\} end is the type of a class body. It specifies its instance variables and methods. In this type, typexpr is matched against the self type, therefore providing a binding for the 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, which makes it possible 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] [virtual] 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. The flag `virtual` indicates that this instance variable is not initialized. It can be initialized later through inheritance.

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

Method specification

The specification of a method is written `method [private] method-name : poly-typexpr`, where `method-name` is the name of the method and `poly-typexpr` its expected type, possibly polymorphic. The flag `private` indicates that the method cannot be accessed from outside the object.

The polymorphism may be left implicit in public method specifications: any type variable which is not bound to a class parameter and does not appear elsewhere inside the class specification will be assumed to be universal, and made polymorphic in the resulting method type. Writing an explicit polymorphic type will disable this behaviour.

Several specifications for the same method must have compatible types. Any non-private specification of a method forces it to be public.

Virtual method specification

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

6.9.2 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.
Simple class expressions

The expression `class-path` evaluates to the class bound to the name `class-path`. Similarly, the expression `[ typexpr 
1, \ldots, 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, \ldots, 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 (possibly labeled) expressions. It denotes the class whose constructor is the original constructor applied to the given arguments. The arguments are evaluated like for expression application, but the constructor itself will only be evaluated when objects are created. In particular, side-effects caused by the application of the constructor will only occur at object creation time.

Class function

The expression `fun [ label-name : ] pattern \rightarrow 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.
Conversion from functions with default values to functions with patterns only works identically for class functions as for normal functions.

The expression

\[
\text{fun parameter}_1 \ldots \text{parameter}_n \rightarrow \text{class-expr}
\]

is a short form for

\[
\text{fun parameter}_1 \rightarrow \ldots \text{fun parameter}_n \rightarrow \text{expr}
\]

Local definitions

The \texttt{let} and \texttt{let rec} constructs bind value names locally, as for the core language expressions.

If a local definition occurs at the very beginning of a class definition, it will be evaluated when the class is created (just as if the definition was outside of the class). Otherwise, it will be evaluated when the object constructor is called.

Class body

\[
\text{class-body} ::= [\text{pattern} [\vdash \text{typexpr} ] ] \{ \text{class-field} \}
\]

The expression \texttt{object class-body end} denotes a class body. This is the prototype for an object: it lists the instance variables and methods of an object 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 \((\text{pattern} [\vdash \text{typexpr}])\) is matched against self, therefore providing 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 \texttt{inherit class-expr} allows to reuse methods and instance variables from other classes. The class expression \texttt{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 \texttt{as value-name} to the inheritance construct above. \texttt{value-name} is not a true variable and can only be used to select a method, i.e. in an expression \texttt{value-name # method-name}. This gives access to the method \texttt{method-name} as it was defined in the parent class even if it is redefined in the current class. The scope of an ancestor binding is limited to the current class. The ancestor method may be called from a subclass but only indirectly.

Instance variable definition

The definition \texttt{val [mutable] inst-var-name = expr} adds an instance variable \texttt{inst-var-name} whose initial value is the value of expression \texttt{expr}. The flag \texttt{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.
Since version 3.10, redefinitions of a visible instance variable with the same name do not create a new variable, but are merged, using the last value for initialization. They must have identical types and mutability. However, if an instance variable is hidden by omitting it from an interface, it will be kept distinct from other instance variables with the same name.

**Virtual instance variable definition**

Variable specification is written `val [mutable] virtual inst-var-name : typexpr`. It specifies whether the variable is modifiable, and gives its type.

Virtual instance variables were added in version 3.10.

**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 same object, defined in this class or one of its subclasses). 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. A method may have both public and private definitions, but as soon as there is a public one, all subsequent definitions will be made public.

Methods may have an explicitly polymorphic type, allowing them to be used polymorphically in programs (even for the same object). The explicit declaration may be done in one of three ways:

1. by giving an explicit polymorphic type in the method definition, immediately after the method name, i.e. `method [private] method-name : \{'ident\}+ . typexpr = expr;`
2. by a forward declaration of the explicit polymorphic type through a virtual method definition;
3. by importing such a declaration through inheritance and/or constraining the type of `self`.

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

\[
expr ::= \ldots \\
| \text{inst-var-name} \leftarrow expr \\
| \{< \text{inst-var-name} = expr \{; \text{inst-var-name} = expr\}>\}
\]

The expression `inst-var-name \leftarrow 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 `\{< \text{inst-var-name} = expr \{; \text{inst-var-name} = expr\}>\}` evaluates to a copy of the current object in which the values of instance variables `inst-var-name_1, \ldots, inst-var-name_n` have been replaced by the values of the corresponding expressions `expr_1, \ldots, expr_n`.

**Virtual method definition**

Method specification is written `method [private] virtual method-name : poly-typexpr`. It specifies whether the method is public or private, and gives its type. If the method is intended to be
polymorphic, the type should be explicit.

**Constraints on type parameters**

The construct `constraint typexpr₁ = typexpr₂` 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.

### 6.9.3 Class definitions

```
class-definition ::= class class-binding {and class-binding}
class-binding ::= [virtual] [[ type-parameters ]] class-name {parameter} [: 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 6.4).

**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.

### 6.9.4 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.

6.9.5 Class type definitions

\[
\text{classtype-definition} ::= \text{class type classtype-def \{and classtype-def\}}
\]

\[
\text{classtype-def} ::= [\text{virtual}] [[\text{type-parameters}]] \text{class-name} = \text{class-body-type}
\]

A class type definition \texttt{class class-name = class-body-type} defines an abbreviation \texttt{class-name}
for the class body type \texttt{class-body-type}. As for class definitions, two type abbreviations \texttt{class-name}
and \texttt{# 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 \texttt{virtual}.

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

6.10 Module types (module specifications)

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

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

\[
\text{mod-constraint} ::= \text{type [type-parameters] typeconstr = typexpr} \\
| \text{module module-path = extended-module-path}
\]

\[
\text{specification} ::= \text{val value-name : typexpr} \\
| \text{external value-name : typexpr = external-declaration} \\
| \text{type-definition} \\
| \text{exception constr-decl} \\
| \text{class-specification} \\
| \text{classtype-definition} \\
| \text{module module-name : module-type} \\
| \text{module module-name \{ ( module-name : module-type ) \} : module-type} \\
| \text{module type modtype-name} \\
| \text{module type modtype-name = module-type} \\
| \text{open module-path} \\
| \text{include module-type}
\]
6.10.1 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`.

6.10.2 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 18).

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 `= typexpr` and an optional type representation `= constr-decl ... or = { field-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 fields 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 = typexpr, no representation.**

The type name must be implemented by a type compatible with typexpr. All users of the structure know that the type name is compatible with typexpr.

**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 fields specified. All users of the structure have access to the constructors or fields, 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 6.9.4.

**Class type specifications**

A specification of one or several classe types in a signature is written `class type classtype-def {and classtype-def}` and consists of a sequence of mutually recursive definitions of class type names.

Class type specifications are described more precisely in section 6.9.5.

**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 ( name1 : module-type1 ) ... ( name_n : module-type_n ) : module-type
```

instead of

```
module module-name : functor ( name1 : module-type1 ) -> ... -> 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 module-type in a signature performs textual inclusion of the components of the signature denoted by module-type. It behaves as if the components of the included signature were copied at the location of the include. The module-type argument must refer to a module type that is a signature, not a functor type.

6.10.3 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).

6.10.4 The with operator

Assuming module-type denotes a signature, the expression module-type with mod-constraint { and mod-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 = typexpr adds the type equation = typexpr 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) ...

Constraints are added left to right. After each constraint has been applied, the resulting signature must be a subtype of the signature before the constraint was applied. Thus, the with operator can only add information on the type components of a signature, but never remove information.

6.11 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.

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

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

6.11.1 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.
Chapter 6. The OCaml language

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

6.11.2 Structures

Structures \texttt{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 9) and with Caml Light, an optional \texttt{;;} is allowed after each definition in a structure. The \texttt{;;} has no semantic meaning. Also for compatibility, \texttt{expr ;;} is allowed as a component of a structure, meaning \texttt{let _ = expr}, i.e. evaluate \texttt{expr} for its side-effects. In this case, the \texttt{;;} of the previous component is not optional.

Value definitions

A value definition \texttt{let [rec] let-binding \{and let-binding\}} bind value names in the same way as a \texttt{let...in...} expression (see section 6.7.1). 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 \texttt{external value-name : typexpr = external-declaration} implements \texttt{value-name} as the external function specified in \texttt{external-declaration} (see chapter 18).

Type definitions

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

Exception definitions

Exceptions are defined with the syntax \texttt{exception constr-decl} or \texttt{exception constr-name = constr}.

Class definitions

A definition of one or several classes is written \texttt{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 6.9.3.

Class type definitions

A definition of one or several classes is written \texttt{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 6.9.5.
Module definitions

The basic form for defining a module component is \texttt{module \texttt{name} = \texttt{expr}}, which evaluates \texttt{expr} and binds the result to the name \texttt{name}.

One can write

\texttt{module \texttt{name} : \texttt{type} = \texttt{expr}}

instead of

\texttt{module \texttt{name} = ( \texttt{expr} : \texttt{type} )}.

Another derived form is

\texttt{module \texttt{name} ( \texttt{name}_1 : \texttt{type}_1 ) \ldots ( \texttt{name}_n : \texttt{type}_n ) = \texttt{expr}}

which is equivalent to

\texttt{module \texttt{name} = functor ( \texttt{name}_1 : \texttt{type}_1 ) -> ... -> \texttt{expr}}

Module type definitions

A definition for a module type is written \texttt{module \texttt{type} \texttt{name} = \texttt{type}}. It binds the name \texttt{name} to the module type denoted by the expression \texttt{type}.

Opening a module path

The expression \texttt{open \texttt{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 \texttt{path} to be referred to by their simple names \texttt{name} instead of \texttt{path.name}. The scope of the \texttt{open} stops at the end of the structure expression.

Including the components of another structure

The expression \texttt{include \texttt{expr}} in a structure re-exports in the current structure all definitions of the structure denoted by \texttt{expr}. For instance, if the identifier \texttt{S} is bound to the module

\texttt{struct type \texttt{t} = int let \texttt{x} = 2 end}

the module expression

\texttt{struct include \texttt{S} let \texttt{y} = (\texttt{x} + 1 : \texttt{t}) end}

is equivalent to the module expression

\texttt{struct type \texttt{t} = int let \texttt{x} = 2 let \texttt{y} = (\texttt{x} + 1 : \texttt{t}) end}

The difference between \texttt{open} and \texttt{include} is that \texttt{open} simply provides short names for the components of the opened structure, without defining any components of the current structure, while \texttt{include} also adds definitions for the components of the included structure.
6.11.3 Functors

Functor definition

The expression \texttt{functor ( module-name : module-type ) -> module-expr} evaluates to a functor that takes as argument modules of the type \texttt{module-type}, binds \texttt{module-name} to these modules, evaluates \texttt{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 \texttt{module-expr\_1 ( module-expr\_2 )} evaluates \texttt{module-expr\_1} to a functor and \texttt{module-expr\_2} to a module, and applies the former to the latter. The type of \texttt{module-expr\_2} must match the type expected for the arguments of the functor \texttt{module-expr\_1}.

6.12 Compilation units

\begin{align*}
\text{unit-interface} & := \{ \text{specification } [ ; ; ] \} \\
\text{unit-implementation} & := \{ \text{definition } [ ; ; ] \}
\end{align*}

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 \texttt{sig...end} signature expression. The implementation contains a sequence of definitions, just as the inside of a \texttt{struct...end} module expression. A compilation unit also has a name \texttt{unit-name}, derived from the names of the files containing the interface and the implementation (see chapter 8 for more details). A compilation unit behaves roughly as the module definition

\begin{verbatim}
module unit-name : sig unit-interface end = struct unit-implementation end
\end{verbatim}

A compilation unit can refer to other compilation units by their names, as if they were regular modules. For instance, if \texttt{U} is a compilation unit that defines a type \texttt{t}, other compilation units can refer to that type under the name \texttt{U.t}; they can also refer to \texttt{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

\begin{verbatim}
name\_1 : sig specification\_1 end...name\_n : sig specification\_n end
\end{verbatim}

where \texttt{name\_1...name\_n} are the names of the other compilation units available in the search path (see chapter 8 for more details) and \texttt{specification\_1...specification\_n} are their respective interfaces.
Chapter 7

Language extensions

This chapter describes language extensions and convenience features that are implemented in OCaml, but not described in the OCaml reference manual.

7.1 Integer literals for types int32, int64 and nativeint

(Introduced in Objective Caml 3.07)

\[
\begin{align*}
\text{int32-literal} & ::= \text{integer-literal} 1 \\
\text{int64-literal} & ::= \text{integer-literal} L \\
\text{nativeint-literal} & ::= \text{integer-literal} n
\end{align*}
\]

An integer literal can be followed by one of the letters 1, L or n to indicate that this integer has type int32, int64 or nativeint respectively, instead of the default type int for integer literals. The library modules Int32\[20.13\], Int64\[20.14\] and Nativeint\[20.20\] provide operations on these integer types.

7.2 Streams and stream parsers

(Removed in Objective Caml 3.03)

The syntax for streams and stream parsers is no longer part of the OCaml language, but available through a Camlp4 syntax extension. See the Camlp4 reference manual for more information. Support for basic operations on streams is still available through the Stream\[20.32\] module of the standard library. OCaml programs that use the stream parser syntax should be compiled with the -pp camlp4o option to ocamlc and ocamlopt. For interactive use, run ocaml and issue the #load "camlp4o.cma";; command.

7.3 Recursive definitions of values

(Introduced in Objective Caml 1.00)
As mentioned in section 6.7.1 the let rec binding construct, in addition to the definition of recursive functions, also supports a certain class of recursive definitions of non-functional values, such as

\[
\text{let rec } \text{name}_1 = 1 :: \text{name}_2 \text{ and } \text{name}_2 = 2 :: \text{name}_1 \text{ in expr}
\]

which binds \text{name}_1 to the cyclic list \(1::2::1::2::\ldots\), and \text{name}_2 to the cyclic list \(2::1::2::1::\ldots\). 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.

More precisely, consider the expression:

\[
\text{let rec } \text{name}_1 = \text{expr}_1 \text{ and } \ldots \text{ and } \text{name}_n = \text{expr}_n \text{ in expr}
\]

It will be accepted if each one of \(\text{expr}_1\ldots\text{expr}_n\) is statically constructive with respect to \(\text{name}_1\ldots\text{name}_n\), is not immediately linked to any of \(\text{name}_1\ldots\text{name}_n\), and is not an array constructor whose arguments have abstract type.

An expression \(e\) is said to be statically constructive with respect to the variables \(\text{name}_1\ldots\text{name}_n\) if at least one of the following conditions is true:

- \(e\) has no free occurrence of any of \(\text{name}_1\ldots\text{name}_n\)
- \(e\) is a variable
- \(e\) has the form \text{fun...} \rightarrow \ldots
- \(e\) has the form \text{function...} \rightarrow \ldots
- \(e\) has the form \text{lazy (...)}
- \(e\) has one of the following forms, where each one of \(\text{expr}_1\ldots\text{expr}_m\) is statically constructive with respect to \(\text{name}_1\ldots\text{name}_n\), and \(\text{expr}_0\) is statically constructive with respect to \(\text{name}_1\ldots\text{name}_n\),\(\text{xname}_1\ldots\text{xname}_m\):
  - \text{let [rec] } \text{xname}_1 = \text{expr}_1 \text{ and } \ldots \text{ and } \text{xname}_m = \text{expr}_m \text{ in } \text{expr}_0
  - \text{let module... in expr}_1
  - \text{constr ( expr}_1 , \ldots , \text{expr}_m \)\)
  - \(\{'\text{ tag-name ( expr}_1 , \ldots , \text{expr}_m\)\)
  - [\text{expr}_1 ; \ldots ; \text{expr}_m \]
  - { \text{field}_1 = \text{expr}_1 ; \ldots ; \text{field}_m = \text{expr}_m \}
  - \{ \text{expr}_1 \text{ with field}_2 = \text{expr}_2 ; \ldots ; \text{field}_m = \text{expr}_m \} \text{ where expr}_1 \text{ is not immediately linked to name}_1\ldots\text{name}_n\)
  - ( \text{expr}_1 , \ldots , \text{expr}_m \)
  - \text{expr}_1 ; \ldots ; \text{expr}_m

An expression \(e\) is said to be immediately linked to the variable \(\text{name}\) in the following cases:

- \(e\) is \text{name}
• $e$ has the form $expr_1 ; \ldots ; expr_m$ where $expr_m$ is immediately linked to $name$

• $e$ has the form $let \{ rec \} \ xname_1 = expr_1 \ and \ldots \ and \ xname_m = expr_m \ in \ expr_0$ where $expr_0$ is immediately linked to $name$ or to one of the $xname_i$ such that $expr_i$ is immediately linked to $name$.

7.4 Range patterns
(Introduced in Objective Caml 1.00)

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

'c' | 'c_1' | 'c_2' | \ldots | 'c_n' | '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.

7.5 Assertion checking
(Introduced in Objective Caml 1.06)

OCaml 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 source file name and the location of expr as arguments. Assertion checking can be turned off with the -noassert compiler option.

As a special case, assert false is reduced to raise (Assert_failure ...), which is polymorphic (and is not turned off by the -noassert option).

7.6 Lazy evaluation
(Introduced in Objective Caml 1.06 for expressions; in Objective Caml 3.11 for patterns)

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. Applications of Lazy.force may be implicit through pattern matching.

The pattern lazy pattern matches values $v$ of type Lazy.t, provided pattern matches the result of forcing $v$ with Lazy.force. A successful match of a pattern containing lazy sub-patterns forces the corresponding parts of the value being matched, even those that imply no test such as lazy value-name or lazy _.

Matching a value with a pattern-matching where some patterns contain lazy sub-patterns may imply forcing parts of the value, even when the pattern selected in the end has no lazy sub-pattern.

For more information, see the description of module Lazy in the standard library (see section 20.15).
7.7 Local modules

(Introduced in Objective Caml 2.00)

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:

```ocaml
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)
```

7.8 Recursive modules

(Introduced in Objective Caml 3.07)

Recursive module definitions, introduced by the `module rec` ...`and`... construction, generalize regular module definitions `module module-name = module-expr` and module specifications `module module-name : module-type` by allowing the defining `module-expr` and the `module-type` to refer recursively to the module identifiers being defined. A typical example of a recursive module definition is:

```ocaml
module rec A : sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end
= struct
  type t = Leaf of string | Node of ASet.t
  let compare t1 t2 =
    match (t1, t2) with
      (Leaf s1, Leaf s2) -> Pervasives.compare s1 s2
      | (Leaf _, Node _) -> 1
      | (Node _, Leaf _) -> -1
      | (Node n1, Node n2) -> ASet.compare n1 n2
  end
  and ASet : Set.S with type elt = A.t
  = Set.Make(A)
```
It can be given the following specification:

```ocaml
module rec A : sig
  type t = Leaf of string | Node of ASet.t
  val compare: t -> t -> int
end

and ASet : Set.S with type elt = A.t
```

This is an experimental extension of OCaml: the class of recursive definitions accepted, as well as its dynamic semantics are not final and subject to change in future releases.

Currently, the compiler requires that all dependency cycles between the recursively-defined module identifiers go through at least one “safe” module. A module is “safe” if all value definitions that it contains have function types $\text{typexpr}_1 \rightarrow \text{typexpr}_2$. Evaluation of a recursive module definition proceeds by building initial values for the safe modules involved, binding all (functional) values to `fun _ -> raise Undefined_recursive_module`. The defining module expressions are then evaluated, and the initial values for the safe modules are replaced by the values thus computed. If a function component of a safe module is applied during this computation (which corresponds to an ill-founded recursive definition), the `Undefined_recursive_module` exception is raised.

## 7.9 Private types

Private type declarations in module signatures, of the form `type t = private ...`, enable libraries to reveal some, but not all aspects of the implementation of a type to clients of the library. In this respect, they strike a middle ground between abstract type declarations, where no information is revealed on the type implementation, and data type definitions and type abbreviations, where all aspects of the type implementation are publicized. Private type declarations come in three flavors: for variant and record types (section 7.9.1), for type abbreviations (section 7.9.2), and for row types (section 7.9.3).

### 7.9.1 Private variant and record types

(Introduced in Objective Caml 3.07)

```
type-representation ::= ...
  | =private constr-decl { | constr-decl}
  | =private { field-decl { ; field-decl} }
```

Values of a variant or record type declared `private` can be de-structured normally in pattern-matching or via the `expr . field` notation for record accesses. However, values of these types cannot be constructed directly by constructor application or record construction. Moreover, assignment on a mutable field of a private record type is not allowed.

The typical use of private types is in the export signature of a module, to ensure that construction of values of the private type always go through the functions provided by the module, while still allowing pattern-matching outside the defining module. For example:

```ocaml
module M : sig
  type t = private A | B of int
```
val a : t
val b : int -> t
end
= struct
  type t = A | B of int
  let a = A
  let b n = assert (n > 0); B n
end

Here, the private declaration ensures that in any value of type M.t, the argument to the B constructor is always a positive integer.

With respect to the variance of their parameters, private types are handled like abstract types. That is, if a private type has parameters, their variance is the one explicitly given by prefixing the parameter by a ‘+’ or a ‘-‘, it is invariant otherwise.

7.9.2 Private type abbreviations
(Introduced in Objective Caml 3.11)

    type-equation ::= ...
            | = private typexpr

Unlike a regular type abbreviation, a private type abbreviation declares a type that is distinct from its implementation type typexpr. However, coercions from the type to typexpr are permitted. Moreover, the compiler “knows” the implementation type and can take advantage of this knowledge to perform type-directed optimizations. For ambiguity reasons, typexpr cannot be an object or polymorphic variant type, but a similar behaviour can be obtained through private row types.

The following example uses a private type abbreviation to define a module of nonnegative integers:

module N : sig
  type t = private int
  val of_int : int -> t
  val to_int : t -> int
end
= struct
  type t = int
  let of_int n = assert (n >= 0); n
  let to_int n = n
end

The type N.t is incompatible with int, ensuring that nonnegative integers and regular integers are not confused. However, if x has type N.t, the coercion (x :> int) is legal and returns the underlying integer, just like N.to_int x. Deep coercions are also supported: if l has type N.t list, the coercion (l :> int list) returns the list of underlying integers, like List.map N.to_int l but without copying the list l.

Note that the coercion (expr :> typexpr) is actually an abbreviated form, and will only work in presence of private abbreviations if both the type of expr and typexpr contain no type variables.
If this is not the case, you must use the full form \((\text{expr} : \text{typ}_e :> \text{typexpr})\) where \(\text{typ}_e\) is the expected type of \(\text{expr}\). Concretely, this would be \((x : \text{N.t} :> \text{int})\) and \((l : \text{N.t list} :> \text{int list})\) for the above examples.

### 7.9.3 Private row types

(Introduced in Objective Caml 3.09)

\[
\begin{align*}
\text{type-equation} & ::= \ldots \\
& \mid =\text{private typexpr}
\end{align*}
\]

Private row types are type abbreviations where part of the structure of the type is left abstract. Concretely \(\text{typexpr}\) in the above should denote either an object type or a polymorphic variant type, with some possibility of refinement left. If the private declaration is used in an interface, the corresponding implementation may either provide a ground instance, or a refined private type.

```ocaml
module M : sig
  type c = private < x : int; .. >
  val o : c
end =
  struct
    class c = object
      method x = 3
      method y = 2
    end
    let o = new c
  end
```

This declaration does more than hiding the \(y\) method, it also makes the type \(c\) incompatible with any other closed object type, meaning that only \(o\) will be of type \(c\). In that respect it behaves similarly to private record types. But private row types are more flexible with respect to incremental refinement. This feature can be used in combination with functors.

```ocaml
module F(X : sig
  type c = private < x : int; .. >
end) =
  struct
    let get_x (o : X.c) = o#x
  end
module G(X : sig
  type c = private < x : int; y : int; .. >
end) =
  struct
    include F(X)
    let get_y (o : X.c) = o#y
  end
```

Polymorphic variant types can be refined in two ways, either to allow the addition of new constructors, or to allow the disparition of declared constructors. The second case corresponds to private variant types (one cannot create a value of the private type), while the first case requires default cases in pattern-matching to handle addition.

```ocaml
type t = ['A of int | 'B of bool ]
type u = private ['< t > 'A ]
type v = private ['> t ]
```

With type \(u\), it is possible to create values of the form \(('A n)\), but not \(('B b)\). With type \(v\), construction is not restricted but pattern-matching must have a default case.

Like for abstract and private types, the variance of type parameters is not inferred, and must be given explicitly.
7.10 Local opens

(Introduced in OCaml 3.12)

\[
\text{expr ::= \ldots} \\
\quad | \ \text{let open module-path in expr} \\
\quad | \ \text{module-path . ( expr )}
\]

The expressions \text{let open module-path in expr} and module-path. ( expr ) are strictly equivalent. They locally open the module referred to by the module path module-path in the scope of the expression expr.

Restricting opening to the scope of a single expression instead of a whole structure allows one to benefit from shorter syntax to refer to components of the opened module, without polluting the global scope. Also, this can make the code easier to read (the open statement is closer to where it is used) and to refactor (because the code fragment is more self-contained).

7.11 Record notations

(Introduced in OCaml 3.12)

\[
\text{pattern ::= \ldots} \\
\quad | \ \{ \text{field \[= pattern\]} \{ ; \text{field \[= pattern\]}\} \{ ; \_\} \}
\]

\[
\text{expr ::= \ldots} \\
\quad | \ \{ \text{field \[= expr\]} \{ ; \text{field \[= expr\]}\} \}
\quad | \ \{ \text{expr with field \[= expr\]} \{ ; \text{field \[= expr\]}\} \}
\]

In a record pattern or a record construction expression, a single identifier \text{id} stands for \text{id = id}, and a qualified identifier \text{path . id} stands for \text{path . id = id}. For example, assuming the record type

\[
\text{type point} = \{ \text{x: float; y: float} \}
\]

has been declared, the following expressions are equivalent:

\[
\text{let x = 1 and y = 2 in \{ x = x; y = y \}} \\
\text{let x = 1 and y = 2 in \{ x; y \}} \\
\text{let x = 1 and y = 2 in \{ x = x; y \}}
\]

Likewise, the following functions are equivalent:

\[
\text{fun \{x = x; y = y\} -> x + y} \\
\text{fun \{x; y\} -> x + y}
\]
Optionally, a record pattern can be terminated by ; _ to convey the fact that not all fields of the record type are listed in the record pattern and that it is intentional. By default, the compiler ignores the ; _ annotation. If the R warning is turned on, however, the compiler will warn if a record pattern fails to list all fields of the corresponding record type and is not terminated by ; _. Continuing the point example above,

```ocaml
fun {x} -> x + 1
```

will warn if the R warning is on, while

```ocaml
fun {x; _} -> x + 1
```

will not warn. This warning can help spot program points where record patterns may need to be modified after new fields were added to a record type.

### 7.12 Explicit polymorphic type annotations

(Introduced in OCaml 3.12)

\[
\text{let-binding} ::= \ldots \\
| \text{value-name} : \text{poly-typexpr} = \text{expr}
\]

Polymorphic type annotations in \texttt{let}-definitions behave in a way similar to polymorphic methods: they explicitly require the defined value to be polymorphic, and allow one to use this polymorphism in recursive occurrences (when using \texttt{let rec}). Note however that this is just an usual polymorphic type, unifiable with any instance of itself.

There two possible applications of this feature. One is polymorphic recursion:

```ocaml
type 'a t = Leaf of 'a | Node of ('a * 'a) t
let rec depth : 'a. 'a t -> 'b = function
  | Leaf _ -> 1
  | Node x -> 1 + depth x
```

Note that \texttt{'b} is not explicitly polymorphic here, and it will actually be unified with \texttt{int}.

The other application is to ensure that some definition is sufficiently polymorphic.

```ocaml
# let id : 'a. 'a -> 'a = fun x -> x+1 ;;
Error: This definition has type int -> int which is less general than 'a. 'a -> 'a
```

### 7.13 Explicit naming of type variables

(Introduced in OCaml 3.12)

\[
\text{parameter} ::= \ldots \\
| ( \text{type typeconstr-name} )
\]
The expression \texttt{fun (type typeconstr-name) \rightarrow expr} introduces a type constructor named \texttt{typeconstr-name} which is considered abstract in the scope of the sub-expression, but then replaced by a fresh type variable. Note that contrary to what the syntax could suggest, the expression \texttt{fun (type typeconstr-name) \rightarrow expr} itself does not suspend the evaluation of \texttt{expr} as a regular abstraction would. The syntax has been chosen to fit nicely in the context of function declarations, where it is generally used. It is possible to freely mix regular function parameters with pseudo type parameters, as in:

\begin{verbatim}
let f = fun (type t) (foo : t list) \rightarrow ...
\end{verbatim}

and even use the alternative syntax for declaring functions:

\begin{verbatim}
let f (type t) (foo : t list) = ...
\end{verbatim}

This construction is useful because the type constructor it introduces can be used in places where a type variable is not allowed. For instance, one can use it to define an exception in a local module within a polymorphic function.

\begin{verbatim}
let f (type t) () =
let module M = struct exception E of t end in
(fun x \rightarrow M.E x), (function M.E x \rightarrow Some x | _ \rightarrow None)
\end{verbatim}

Here is another example:

\begin{verbatim}
let sort_uniq (type s) (cmp : s \rightarrow s \rightarrow int) =
let module S = Set.Make(struct type t = s let compare = cmp end) in
fun l \rightarrow
  S.elements (List.fold_right S.add l S.empty)
\end{verbatim}

This construction by itself does not make polymorphic the type variable it introduces, but it can be combined with explicit polymorphic annotations where needed. It is also extremely useful for first-class modules.

### 7.14 First-class modules

(Introduced in OCaml 3.12)

\begin{verbatim}
tyexpr ::= ...
  | (module package-type )

module-expr ::= ...
  | (val expr : package-type )

expr ::= ...
  | (module module-expr : package-type )

package-type ::= modtype-path
  | modtype-path with package-type-constraint {and package-type-constraint}

package-type-constraint ::= type typeconstr-name = typexpr
\end{verbatim}
Modules are typically thought as static components. This extension makes it possible to pack a module as a first-class value, which can later be dynamically unpacked into a module.

The expression \((\text{module } \text{module-expr} : \text{package-type})\) converts the module (structure or functor) denoted by module expression \(\text{module-expr}\) to a value of the core language that encapsulates this module. The type of this core language value is \((\text{module } \text{package-type})\).

Conversely, the module expression \((\text{val } \text{expr} : \text{package-type})\) evaluates the core language expression \(\text{expr}\) to a value, which must have type \(\text{module } \text{package-type}\), and extracts the module that was encapsulated in this value.

The \text{package-type} syntactic class appearing in the \((\text{module } \text{module-expr} : \text{package-type})\) expressions, the \((\text{val } \text{expr} : \text{package-type})\) module expressions and the \((\text{module } \text{package-type})\) type expression represents a subset of module types. This subset consists of named module types with optional constraints of a limited form: only non-parametrized types can be specified. For type-checking purposes, package types are compared by path equality on the module type name component, and normal type equality for constraints.

The module expression \((\text{val } \text{expr} : \text{package-type})\) cannot be used in the body of a functor, because this can cause unsoundness in conjunction with applicative functors. It can be used anywhere in the context of a local module binding \(\text{let } M = (\text{val } \text{expr}_1 : \text{package-type}) \text{ in } \text{expr}_2\), however.

**Basic example** A typical use of first-class modules is to select at run-time among several implementations of a signatures. Each implementation is a structure that we can encapsulate as a first-class module, then store in a data structure such as a hash table:

```ocaml
module type DEVICE = sig ... end
let devices : (string, module DEVICE) Hashtbl.t = Hashtbl.create 17

module SVG = struct ... end
let _ = Hashtbl.add devices "SVG" (module SVG : DEVICE)

module PDF = struct ... end
let _ = Hashtbl.add devices "PDF" (module PDF: DEVICE)
```

We can then select one implementation based on command-line arguments, for instance:

```ocaml
module Device =
  (val (try Hashtbl.find devices (parse_cmdline()))
    with Not_found -> eprintf "Unknown device \%s\n"; exit 2)
  : DEVICE)
```

Alternatively, the selection can be performed within a function:

```ocaml
let draw_using_device device_name picture =
  let module Device =
    (val (Hashtbl.find_devices device_name) : DEVICE)
  in
  Device.draw picture
```
Advanced examples

With first-class modules, it is possible to parametrize some code over the implementation of a module without using a functor.

```
let sort (type s) set l =
  let module Set = (val set : Set.S with type elt = s) in
  Set.elements (List.fold_right Set.add l Set.empty)
```

The type inferred for this function is `module Set.S with type elt = 'a) -> 'a list -> 'a list`. To use this function, one can wrap the `Set.Make` functor:

```
let make_set (type s) cmp =
  let module S = Set.Make(struct
    type t = s
    let compare = cmp
  end) in
  (module S : Set.S with type elt = s)
```

This function has type `('a -> 'a -> int) -> (module Set.S with type elt = 'a)`. Another advanced use of first-class module is to encode existential types. In particular, they can be used to simulate generalized algebraic data types (GADT). To demonstrate this, we first define a type of witnesses for type equalities:

```
module TypEq : sig
  type ('a, 'b) t
  val apply: ('a, 'b) t -> 'a -> 'b
  val refl: ('a, 'a) t
  val sym: ('a, 'b) t -> ('b, 'a) t
end = struct
  type ('a, 'b) t = ('a -> 'b) * ('b -> 'a)
  let refl = (fun x -> x), (fun x -> x)
  let apply (f, g) x = f x
  let sym (f, g) = (g, f)
end
```

We can then define a paramatrized algebraic data type whose constructors provide some information about the type parameter:

```
module rec Typ : sig
  module type PAIR = sig
    type t and t1 and t2
    val eq: (t, t1 * t2) TypEq.t
    val t1: t1 Typ.typ
    val t2: t2 Typ.typ
  end

  type 'a typ =
```
Chapter 7. Language extensions

| Int of ('a, int) TypEq.t |
| String of ('a, string) TypEq.t |
| Pair of (module PAIR with type t = 'a) |

end = Typ

Values of type 'a typ are supposed to be runtime representations for the type 'a. The constructors Int and String are easy: they directly give a witness of type equality between the parameter 'a and the ground types int (resp. string). The constructor Pair is more complex. One wants to give a witness of type equality between 'a and a type of the form t1 * t2 together with the representations for t1 and t2. However, these two types are unknown. The code above shows how to use first-class modules to simulate existentials.

Here is how to construct values of type 'a typ:

let int = Typ.Int TypEq.refl

let str = Typ.String TypEq.refl

let pair (type s1) (type s2) t1 t2 =
    let module P = struct
        type t = s1 * s2
        type t1 = s1
        type t2 = s2
        let eq = TypEq.refl
        let t1 = t1
        let t2 = t2
    end in
    let pair = (module P : Typ.PAIR with type t = s1 * s2) in
    Typ.Pair pair

And finally, here is an example of a polymorphic function that takes the runtime representation of some type 'a and a value of the same type, then pretty-print the value as a string:

open Typ

let rec to_string: 'a. 'a Typ.typ -> 'a -> string =
    fun (type s) t x ->
        match t with
        | Int eq -> string_of_int (TypEq.apply eq x)
        | String eq -> Printf.sprintf "%S" (TypEq.apply eq x)
        | Pair p ->
            let module P = (val p : PAIR with type t = s) in
            let (x1, x2) = TypEq.apply P.eq x in
            Printf.sprintf "(%s,%s)" (to_string P.t1 x1) (to_string P.t2 x2)

Note that this function uses an explicit polymorphic annotation to obtain polymorphic recursion.
7.15 Recovering the type of a module

(Introduced in OCaml 3.12)

\[\text{module-type ::= } \ldots \]
\[| \text{module type of } \text{module-expr}\]

The construction `module type of` \text{module-expr} expands to the module type (signature or functor type) inferred for the module expression \text{module-expr}.

A typical use, in conjunction with the signature-level \text{include} construct, is to extend the signature of an existing structure, as in the following example.

\[
\begin{align*}
\text{module type MYHASH = sig} \\
\quad \text{include module type of Hashtbl} \\
\quad \text{val replace: ('a, 'b) t -> 'a -> 'b -> unit} \\
\end{align*}
\]

The signature \text{MYHASH}, then, contains all the fields of the signature of module \text{Hashtbl}, plus the new field \text{replace}. An implementation of this signature can be obtained easily, using the \text{include} construct at the structure level this time:

\[
\begin{align*}
\text{module MyHash : MYHASH = struct} \\
\quad \text{include Hashtbl} \\
\quad \text{let replace t k v = remove t k; add t k v} \\
\end{align*}
\]

7.16 Substituting inside a signature

(Introduced in OCaml 3.12)

\[\text{mod-constraint ::= } \ldots \]
\[| \text{type} [\text{type-parameters}] \text{typeconstr-name := [type-parameters] typeconstr} \]
\[| \text{module} \text{module-name := extended-module-path}\]

“Destructive” substitution (\text{with} \ldots :=) behaves essentially like normal signature constraints (\text{with} \ldots =), but it additionally removes the redefined type or module from the signature. There are a number of restrictions: one can only remove types and modules at the outermost level (not inside submodules), and the definition must be either another type constructor (with identical type parameters), or a module path.

A natural application of destructive substitution is merging two signatures sharing a type name.

\[
\begin{align*}
\text{module type Printable = sig} \\
\quad \text{type t} \\
\quad \text{val print : Format.formatter -> t -> unit} \\
\end{align*}
\]
module type Comparable = sig
  type t
  val compare : t -> t -> int
end

module type PrintableComparable = sig
  include Printable
  include Comparable with type t := t
end

One can also use this to completely remove a field:

# module type S = Comparable with type t := int;;
module type S = sig val compare : int -> int -> int end

or to rename one:

# module type S = sig
#  type u
#  include Comparable with type t := u
# end;;
module type S = sig type u val compare : u -> u -> int end

Note that you can also remove manifest types, by substituting with the same type.

# module type ComparableInt = Comparable with type t = int ;;
module type ComparableInt = sig type t = int val compare : t -> t -> int end
# module type CompareInt = ComparableInt with type t := int ;;
module type CompareInt = sig val compare : int -> int -> int end

7.17 Explicit overriding in class definitions
(Introduced in OCaml 3.12)

class-field ::= ...
  | inherit! class-expr [as value-name]
  | val! [mutable] inst-var-name [: typexpr] = expr
  | method! [private] method-name {parameter} [: typexpr] = expr
  | method! [private] method-name : poly-typexpr = expr

The keywords inherit!, val! and method! have the same semantics as inherit, val and method, but they additionally require the definition they introduce to be an overriding. Namely, method! requires method-name to be already defined in this class, val! requires inst-var-name to be already defined in this class, and inherit! requires class-expr to override some definitions. If no such overriding occurs, an error is signaled.

As a side-effect, these 3 keywords avoid the warnings “method override” and “instance variable override”. As of OCaml 3.12, the warning “method override” has to be enabled manually for backwards compatibility reasons.
Part III

The OCaml tools
Chapter 8

Batch compilation (ocamlc)

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

8.1 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 and processes them sequentially:

- 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 OCaml 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 (`.obj` under Windows). 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` (`.obj` or `.lib` under Windows) 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).

• Arguments ending in `.so` (`.dll` under Windows) are assumed to be C shared libraries (DLLs). During linking, they are searched for external C functions referenced from the Caml code, and their names are written in the generated bytecode executable. The run-time system `ocamldr` then loads them dynamically at program start-up time.

The output of the linking phase is a file containing compiled bytecode that can be executed by the OCaml bytecode interpreter: the command named `ocamldr`. 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 10 for more details.)

On most 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.

### 8.2 Options

The following command-line options are recognized by `ocamlc`. The options `-pack`, `-a`, `-c` and `-output-obj` are mutually exclusive.

- `-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 must be set with the `-o` option.

  If `-custom`, `-cclib` or `-ccopt` options are passed on the command line, these options are stored in the resulting `.cma` library. Then, linking with this library automatically adds back the `-custom`, `-cclib` and `-ccopt` options as if they had been provided on the command line, unless the `-noautolink` option is given.
-annot
Dump detailed information about the compilation (types, bindings, tail-calls, etc). The information for file src.ml is put into file src.annot. In case of a type error, dump all the information inferred by the type-checker before the error. The src.annot file can be used with the emacs commands given in emacs/caml-types.el to display types and other annotations interactively.

-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 when linking in “custom runtime” mode (see the -custom option) 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, for instance, -ccopt -L dir causes the C linker to search for C libraries in directory dir. (See the -custom option.)

-config
Print the version number of ocamlc and a detailed summary of its configuration, then exit.

-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 static linking of Caml code with user-defined C functions, as described in chapter [18].

Unix:

Never use the strip command on executables produced by ocamlc -custom, this would remove the bytecode part of the executable.

-dllib -l libname
Arrange for the C shared library dlllibname.so (dlllibname.dll under Windows) to be loaded dynamically by the run-time system ocamlrun at program start-up time.

-dllpath dir
Adds the directory dir to the run-time search path for shared C libraries. At link-time, shared libraries are searched in the standard search path (the one corresponding to the -I option). The -dllpath option simply stores dir in the produced executable file, where ocamlrun can find it and use it as described in section [10.3].
-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 16), and to produce stack backtraces when the program terminates on an uncaught exception (see section 10.2).

-help-warnings
Show description for all available warning numbers.

-i Cause the compiler to print all defined names (with their inferred types or their definitions) when compiling an implementation (.ml file). No compiled files (.cmo and .cmi files) are produced. 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), compiled object code files (.cmo), libraries (.cma), and C libraries specified with -cclib -lxxx. 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.
If the given directory starts with +, it is taken relative to the standard library directory. For instance, -I +labltk adds the subdirectory labltk of the standard library to the search path.

-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.

-intf-suffix string
Recognize file names ending with string as interface files (instead of the default .mli).

-labels
Labels are not ignored in types, labels may be used in applications, and labelled parameters can be given in any order. This is the default.

-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 (option -a), setting the -linkall option 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 18.1.6 for more information.
-noassert
Do not compile assertion checks. Note that the special form `assert false` is always compiled because it is typed specially. This flag has no effect when linking already-compiled files.

-noautolink
When linking .cma libraries, ignore `-custom`, `-cclib` and `-ccopt` options potentially contained in the libraries (if these options were given when building the libraries). This can be useful if a library contains incorrect specifications of C libraries or C options; in this case, during linking, set `-noautolink` and pass the correct C libraries and options on the command line.

-nolabels
Ignore non-optional labels in types. Labels cannot be used in applications, and parameter order becomes strict.

-o exec-file
Specify the name of the output file produced by the compiler. The default output name is `a.out` under Unix and `camlprog.exe` under Windows. If the `-a` option is given, specify the name of the library produced. If the `-pack` option is given, specify the name of the packed object file produced. If the `-output-obj` option is given, specify the name of the output file produced. If the `-c` option is given, specify the name of the object file produced for the next source file that appears on the command line.

-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 18 section 18.7.5 The name of the output object file is `camlprogs.o` by default; it can be set with the `-o` option. This option can also be used to produce a C source file (.c extension) or a compiled shared/dynamic library (.so extension, .dll under Windows).

-pack
Build a bytecode object file (.cmo file) and its associated compiled interface (.cmi) that combines the object files given on the command line, making them appear as sub-modules of the output .cmo file. The name of the output .cmo file must be given with the `-o` option. For instance,

```
ocamlc -pack -o p.cmo a.cmo b.cmo c.cmo
```

generates compiled files `p.cmo` and `p.cmi` describing a compilation unit having three sub-modules `A`, `B` and `C`, corresponding to the contents of the object files `a.cmo`, `b.cmo` and `c.cmo`. These contents can be referenced as `P.A`, `P.B` and `P.C` in the remainder of the program.

-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.
-principal
Check information path during type-checking, to make sure that all types are derived in a principal way. When using labelled arguments and/or polymorphic methods, this flag is required to ensure future versions of the compiler will be able to infer types correctly, even if internal algorithms change. All programs accepted in -principal mode are also accepted in the default mode with equivalent types, but different binary signatures, and this may slow down type checking; yet it is a good idea to use it once before publishing source code.

-rectypes
Allow arbitrary recursive types during type-checking. By default, only recursive types where the recursion goes through an object type are supported. Note that once you have created an interface using this flag, you must use it again for all dependencies.

-runtime-variant suffix
Add the suffix string to the name of the runtime library used by the program. Currently, only one such suffix is supported: d, and only if the OCaml compiler was configured with option -with-debug-runtime. This suffix gives the debug version of the runtime, which is useful for debugging pointer problems in low-level code such as C stubs.

-thread
Compile or link multithreaded programs, in combination with the system threads library described in chapter 24.

-unsafe
Turn bound checking off for 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 18.1.6 for more information.

-v Print the version number of the compiler and the location of the standard library directory, then exit.

-verbose
Print all external commands before they are executed, in particular invocations of the C compiler and linker in -custom mode. Useful to debug C library problems.

-vnum or -version
Print the version number of the compiler in short form (e.g. 3.11.0), then exit.

-vmthread
Compile or link multithreaded programs, in combination with the VM-level threads library described in chapter 24.

-w warning-list
Enable, disable, or mark as errors the warnings specified by the argument warning-list. Each
warning can be enabled or disabled, and each warning can be marked or unmarked. If a warning is disabled, it isn’t displayed and doesn’t affect compilation in any way (even if it is marked). If a warning is enabled, it is displayed normally by the compiler whenever the source code triggers it. If it is enabled and marked, the compiler will stop with an error after displaying that warning if the source code triggers it.

The warning-list argument is a sequence of warning specifiers, with no separators between them. A warning specifier is one of the following:

+num
    Enable warning number num.
-num
    Disable warning number num.
@num
    Enable and mark warning number num.
+num1..num2
    Enable warnings in the given range.
-num1..num2
    Disable warnings in the given range.
@num1..num2
    Enable and mark warnings in the given range.
+letter
    Enable the set of warnings corresponding to letter. The letter may be uppercase or lowercase.
-letter
    Disable the set of warnings corresponding to letter. The letter may be uppercase or lowercase.
@letter
    Enable and mark the set of warnings corresponding to letter. The letter may be uppercase or lowercase.

uppercase-letter
    Enable the set of warnings corresponding to uppercase-letter.
lowercase-letter
    Disable the set of warnings corresponding to lowercase-letter.

Warning numbers which are out of the range of warnings that are currently defined are ignored. The warning numbers are as follows.

1    Suspicious-looking start-of-comment mark.
2    Suspicious-looking end-of-comment mark.
3    Deprecated syntax.
4    Fragile pattern matching: matching that will remain complete even if additional constructors are added to one of the variant types matched.
Partially applied function: expression whose result has function type and is ignored.
Label omitted in function application.
Some methods are overridden in the class where they are defined.
Partial match: missing cases in pattern-matching.
Missing fields in a record pattern.
Expression on the left-hand side of a sequence that doesn’t have type "unit" (and that is not a function, see warning number 5).
Redundant case in a pattern matching (unused match case).
Redundant sub-pattern in a pattern-matching.
Override of an instance variable.
Illegal backslash escape in a string constant.
Private method made public implicitly.
Unerasable optional argument.
Undeclared virtual method.
Non-principal type.
Type without principality.
Unused function argument.
Non-returning statement.
Camlp4 warning.
Useless record "with" clause.
Bad module name: the source file name is not a valid OCaml module name.
Pattern-matching with all clauses guarded. Exhaustiveness cannot be checked
Suspicious unused variable: unused variable that is bound with "let" or "as", and doesn’t start with an underscore ("_”) character.
Innocuous unused variable: unused variable that is not bound with "let" nor "as", and doesn’t start with an underscore ("_”) character.
Wildcard pattern given as argument to a constant constructor.
Unescaped end-of-line in a string constant (non-portable code).
Two labels or constructors of the same name are defined in two mutually recursive types.

The letters stand for the following sets of warnings. Any letter not mentioned here corresponds to the empty set.

A all warnings
C 1, 2
D 3
E 4
F 5
The default setting is `-w +a-4-6-7-9-27..29`. Note that warnings 5 and 10 are not always triggered, depending on the internals of the type checker.

**-warn-error** `warning-list`

Mark as errors the warnings specified in the argument `warning-list`. The compiler will stop with an error when one of these warnings is emitted. The `warning-list` has the same meaning as for the `-w` option: a `+` sign (or an uppercase letter) turns the corresponding warnings into errors, a `-` sign (or a lowercase letter) turns them back into warnings, and a `@` sign both enables and marks the corresponding warnings.

Note: it is not recommended to use warning sets (i.e. letters) as arguments to `-warn-error` in production code, because this can break your build when future versions of OCaml add some new warnings.

The default setting is `-warn-error -a` (none of the warnings is treated as an error).

**-where**

Print the location of the standard library, then exit.

**-file**

Process `file` as a file name, even if it starts with a dash (`-`) character.

**-help** or **--help**

Display a short usage summary and exit.

### 8.3 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 named Mod.cmi or mod.cmi 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.

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. 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 $\text{foo}$ cannot be used with type $\text{foo}$”, and it really seems that the two types $\text{foo}$ 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:

```ocaml
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 $\text{foo}$ cannot be used with type $\text{foo}$”.

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 unsoundsness (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 for all variables occuring in contravariant or non-variant branches of the type. For instance, if the type of a non-value is ’a list the variable is generalizable (list is a covariant type constructor), but not in ’a list -> ’a list (the left branch of -> is contravariant) or ’a ref (ref is non-variant).

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

  ```ocaml
  let sort_int_list = Sort.list (<)
  (* inferred type ’a list -> ’a list, with ’a not generalized *)
  ```

  write

  ```ocaml
  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

  ```ocaml
  let map_length = List.map Array.length
  (* inferred type ’a array list -> int list, with ’a not generalized *)
  ```

  write

  ```ocaml
  let map_length = fun (list : ’a array list) -> List.map Array.length list
  ```
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

```ml
mod1.ml: let f x = ... Mod2.g ...
mod2.ml: let g y = ... Mod1.f ...
```

...define...

```ml
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 :

```ml
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
```

The external function f is not available
This error appears when trying to link code that calls external functions written in C. As explained in chapter 18, such code must be linked with C libraries that implement the required fC function. If the C libraries in question are not shared libraries (DLLs), the code must be linked in “custom runtime” mode. Fix: add the required C libraries to the command line, and possibly the -custom option.
Chapter 9

The toplevel system (ocaml)

This chapter describes the toplevel system for OCaml, that permits interactive use of the OCaml 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:

\[
\text{toplevel-input} ::= \{ \text{toplevel-phrase} \} ;;
\]

\[
\text{toplevel-phrase} ::= \text{definition} |
\text{expr} |
# \text{ident} \text{directive-argument}
\]

\[
\text{directive-argument} ::= \epsilon |
\text{string-literal} |
\text{integer-literal} |
\text{value-path}
\]

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 6.11), or a value expression (section 6.7). 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 9.2.

Unix:

The toplevel system is started by the command ocaml, as follows:

\[
\text{ocaml} \text{ options} \text{ objects} \quad \# \text{interactive mode}
\]

\[
\text{ocaml} \text{ options} \text{ objects} \text{ scriptfile} \quad \# \text{script mode}
\]
options are described below. objects are filenames ending in .cmo or .cma; they are loaded into the interpreter immediately after options are set. scriptfile is any file name not ending in .cmo or .cma.

If no scriptfile 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 9.2).

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 OCaml phrases and executed as per the #use directive described in section 9.2. The evaluation outcode for each phrase are not displayed. If the current directory does not contain an .ocamlinit file, but the user’s home directory (environment variable HOME) does, the latter is read and executed as described below.

The toplevel system does not perform line editing, but it can easily be used in conjunction with an external line editor such as ledit, ocaml2 or rlwrap (see the Caml Hump http://caml.inria.fr/humps/index_framed_caml.html). Another option is to use ocaml under Gnu Emacs, which gives the full editing power of Emacs (command run-caml from library inf-caml).

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 INTR signal to the ocaml process). The toplevel then immediately returns to the # prompt.

If scriptfile 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 OCaml phrases and executed, as per the #use directive (section 9.2). 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. Sys.argv is transformed, ignoring all OCaml parameters, and starting with the script file name in Sys.argv.(0).

In script mode, the first line of the script is ignored if it starts with #!. Thus, it should be 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 OCaml, and Unix kernels usually do not handle nested #! scripts. A better solution is to put the following as the first line of the script:

    #!/usr/local/bin/ocamlrun /usr/local/bin/ocaml

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. This interface provides a text window in which commands can be entered and edited, and the toplevel responses are printed.

9.1 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.

If the given directory starts with +, it is taken relative to the standard library directory. For instance, -I +labltk adds the subdirectory labltk of the standard library to the search path.

Directories can also be added to the list once the toplevel is running with the #directory directive (section 9.2).

-init file
Load the given file instead of the default initialization file. The default file is .ocamlinit in the current directory if it exists, otherwise .ocamlinit in the user’s home directory.

-labels
Labels are not ignored in types, labels may be used in applications, and labelled parameters can be given in any order. This is the default.

-noassert
Do not compile assertion checks. Note that the special form assert false is always compiled because it is typed specially.

-nolabels
Ignore non-optional labels in types. Labels cannot be used in applications, and parameter order becomes strict.

-noprompt
Do not display any prompt when waiting for input.

-nostdlib
Do not include the standard library directory in the list of directories searched for source and compiled files.

-principal
Check information path during type-checking, to make sure that all types are derived in a principal way. When using labelled arguments and/or polymorphic methods, this flag is required to ensure future versions of the compiler will be able to infer types correctly, even if internal algorithms change. All programs accepted in -principal mode are also accepted in the default mode with equivalent types, but different binary signatures, and this may slow down type checking; yet it is a good idea to use it once before publishing source code.

-rectypes
Allow arbitrary recursive types during type-checking. By default, only recursive types where the recursion goes through an object type are supported.

-unsafe
See the corresponding option for ocamlc, chapter 8. Turn bound checking off on array and
string accesses (the \texttt{v.(i)} and \texttt{s.[i]} constructs). Programs compiled with \texttt{-unsafe} are therefore slightly faster, but unsafe: anything can happen if the program accesses an array or string outside of its bounds.

\textbf{-version}

Print version string and exit.

\textbf{-vnum}

Print short version number and exit.

\textbf{-w warning-list}

Enable or disable warnings according to the argument \textit{warning-list}. See section \ref{section:8.2} for the syntax of the argument.

\textbf{-warn-error warning-list}

Treat as errors the warnings enabled by the argument \textit{warning-list}. See section \ref{section:8.2} for the syntax of the argument.

\textbf{-help or --help}

Display a short usage summary and exit.

\textbf{Unix:}

The following environment variables are also consulted:

\textbf{LC_CTYPE}

If set to \texttt{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 (\texttt{"ddd}).

\textbf{TERM}

When printing error messages, the toplevel system attempts to underline visually the location of the error. It consults the \texttt{TERM} variable to determine the type of output terminal and look up its capabilities in the terminal database.

\textbf{HOME}

Directory where the \texttt{.ocamllinit} file is searched.

\section{9.2 Toplevel directives}

The following directives control the toplevel behavior, load files in memory, and trace program execution.

\textbf{Note:} all directives start with a \# (sharp) symbol. This \# must be typed before the directive, and must not be confused with the \# prompt displayed by the interactive loop. For instance, typing \texttt{#quit;;} will exit the toplevel loop, but typing \texttt{quit;;} will result in an “unbound value \texttt{quit}” error.

\texttt{#quit;;}

Exit the toplevel loop and terminate the \texttt{ocaml} command.
#labels bool;;
Ignore labels in function types if argument is false, or switch back to default behaviour
(commuting style) if argument is true.

#warnings "warning-list";;
Enable or disable warnings according to the argument.

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 values
whose types match the argument type of the function. That is, the toplevel loop will call
printer-name when it has such a value to print.

The printing function printer-name should have type Format.formatter -> t -> unit, where
the type for the values to be printed, and should output its textual representation for
the value of type t on the given formatter, using the functions provided by the Format library. For
backward compatibility, printer-name can also have type t -> unit and should then output
on the standard formatter, but this usage is deprecated.

#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).

### 9.3 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 6.3).

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 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.

### 9.4 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 t₁, but is used with type t₂**

See section 8.4.

**Reference to undefined global mod**

You have neglected to load in memory an implementation for a module with #load. See section 9.3 above.
9.5 Building custom toplevel systems: ocamlmktop

The ocamlmktop command builds OCaml 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 OCaml toplevel. The typical use is:

    ocamlmktop -o mytoplevel foo.cmo bar.cmo gee.cmo

This creates the bytecode file mytoplevel, containing the OCaml 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.

9.6 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 §.

-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 §.

-custom
  Link in “custom runtime” mode. See the corresponding option for ocamlc, in chapter §.

-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 10

The runtime system (ocamlrun)

The ocamlrun command executes bytecode files produced by the linking phase of the ocamlc command.

10.1 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 arg1 ... argn

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 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 arg1 to argn.

As mentioned in chapter 8, 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 arg1 ... argn

works exactly as

    ocamlrun caml.out arg1 ... argn

Notice that it is not possible to pass options to ocamlrun when invoking caml.out directly.

Windows:

Under several versions of Windows, bytecode executable files are self-executable only if their name ends in .exe. It is recommended to always give .exe names to bytecode executables, e.g. compile with ocamlc -o myprog.exe ... rather than ocamlc -o myprog ....
10.2 Options

The following command-line options are recognized by ocamlrun.

-b  When the program aborts due to an uncaught exception, print a detailed “back trace” of the execution, showing where the exception was raised and which function calls were outstanding at this point. The back trace is printed only if the bytecode executable contains debugging information, i.e. was compiled and linked with the -g option to ocamlc set. This is equivalent to setting the b flag in the OCAMLRUNPARAM environment variable (see below).

-I dir  Search the directory dir for dynamically-loaded libraries, in addition to the standard search path (see section 10.3).

-p  Print the names of the primitives known to this version of ocamlrun and exit.

-v  Direct the memory manager to print some progress messages on standard error. This is equivalent to setting v=63 in the OCAMLRUNPARAM environment variable (see below).

-version  Print version string and exit.

-vnum  Print short version number and exit.

The following environment variables are also consulted:

CAML_LD_LIBRARY_PATH
     Additional directories to search for dynamically-loaded libraries (see section 10.3).

OCAMLLIB
     The directory containing the OCaml standard library. (If OCAMLLIB is not set, CAMLLIB will be used instead.) Used to locate the ld.conf configuration file for dynamic loading (see section 10.3). If not set, default to the library directory specified when compiling OCaml.

OCAMLRUNPARAM
     Set the runtime system options and garbage collection parameters. (If OCAMLRUNPARAM is not set, CAMLRUNPARAM will be used instead.) This variable must be a sequence of parameter specifications. A parameter specification is an option letter followed by an = sign, a decimal number (or an hexadecimal number prefixed by 0x), and an optional multiplier. There are nine options, six of which correspond to the fields of the control record documented in section 20.10.

b  (backtrace) Trigger the printing of a stack backtrace when an uncaught exception aborts the program. This option takes no argument.

p  (parser trace) Turn on debugging support for ocamlyacc-generated parsers. When this option is on, the pushdown automaton that executes the parsers prints a trace of its actions. This option takes no argument.
s (minor_heap_size) Size of the minor heap. (in words)
i (major_heap_increment) Default size increment for the major heap. (in words)
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 (= 0x001)
Start of major GC cycle.

2 (= 0x002)
Minor collection and major GC slice.

4 (= 0x004)
Growing and shrinking of the heap.

8 (= 0x008)
Resizing of stacks and memory manager tables.

16 (= 0x010)
Heap compaction.

32 (= 0x020)
Change of GC parameters.

64 (= 0x040)
Computation of major GC slice size.

128 (= 0x080)
Calling of finalisation functions

256 (= 0x100)
Startup messages (loading the bytecode executable file, resolving shared libraries).

I (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 OCAMLRUNPARAM=’b,s=256k,v=0x015’
```

tells a subsequent ocamlrun to print backtraces for uncaught exceptions, set its initial minor heap size to 1 megabyte and print a message at the start of each major GC cycle, when the heap size changes, and when compaction is triggered.

CAMLRUNPARAM
If OCAMLRUNPARAM is not found in the environment, then CAMLRUNPARAM will be used instead. If CAMLRUNPARAM is not found, then the default values will be used.

PATH
List of directories searched to find the bytecode executable file.
### 10.3 Dynamic loading of shared libraries

On platforms that support dynamic loading, `ocamlruncan link dynamically with C shared libraries (DLLs) providing additional C primitives beyond those provided by the standard runtime system. The names for these libraries are provided at link time as described in section [8.1.4](#), and recorded in the bytecode executable file; `ocamlruncan then, locates these libraries and resolves references to their primitives when the bytecode executable program starts.

The `ocamlruncan command searches shared libraries in the following directories, in the order indicated:

1. Directories specified on the `ocamlruncan command line with the `-I` option.
2. Directories specified in the `CAML_LD_LIBRARY_PATH` environment variable.
3. Directories specified at link-time via the `-dllpath` option to `ocamlc`. (These directories are recorded in the bytecode executable file.)
4. Directories specified in the file `ld.conf`. This file resides in the OCaml standard library directory, and lists directory names (one per line) to be searched. Typically, it contains only one line naming the `stublibs` subdirectory of the OCaml standard library directory. Users can add there the names of other directories containing frequently-used shared libraries; however, for consistency of installation, we recommend that shared libraries are installed directly in the system `stublibs` directory, rather than adding lines to the `ld.conf` file.
5. Default directories searched by the system dynamic loader. Under Unix, these generally include `/lib` and `/usr/lib`, plus the directories listed in the file `/etc/ld.so.conf` and the environment variable `LD_LIBRARY_PATH`. Under Windows, these include the Windows system directories, plus the directories listed in the `PATH` environment variable.

### 10.4 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 `ocamlruncbytecode interpreter on itself. The second possibility indicates that OCaml has not been properly installed on your system.

**Cannot exec ocamlrunc**

(When launching a self-executable bytecode file.) The `ocamlruncould not be found in the executable path. Check that OCaml has been properly installed on your system.

**Cannot find the bytecode file**

The file that `ocamlruncis trying to execute (e.g. the file given as first non-option argument to `ocamlrunc) either does not exist, or is not a valid executable bytecode file.

**Truncated bytecode file**

The file that `ocamlruncis 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, along with its string and integer arguments (arguments of more complex types are not correctly printed). To locate the context of the uncaught exception, compile the program with the -g option and either run it again under the ocamldebug debugger (see chapter 16), or run it with ocamlrun -b or with the OCAMLRUNPARAM environment variable set to b=1.

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, or with the OCAMLRUNPARAM environment variable set to v=63. 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 11

Native-code compilation (ocamlopt)

This chapter describes the OCaml 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 ocamlopt with bytecode object files produced by ocamlc: 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.

11.1 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, and processes them sequentially:

- 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 or .obj 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 8).
• 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 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.cm file must come before all .cm 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/.lib) a set of object files (.cmx and .o/.obj 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/.obj object file. This object file is linked with the program.

• Arguments ending in .o, .a or .so (.obj, .lib and .dll under Windows) 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 or Windows executable file. It does not need ocamlrun to run.

11.2 Options

The following command-line options are recognized by ocamlopt. The options -pack, -a, -shared, -c and -output-obj are mutually exclusive.

-a Build a library (.cmxa and .a/.lib files) with the object files (.cmx and .o/.obj files) given on the command line, instead of linking them into an executable file. The name of the library must be set with the -o option.

If -cclib or -ccopt options are passed on the command line, these options are stored in the resulting .cmxa library. Then, linking with this library automatically adds back the -cclib and -ccopt options as if they had been provided on the command line, unless the -noautolink option is given.

-annot Dump detailed information about the compilation (types, bindings, tail-calls, etc). The information for file src.ml is put into file src.annot. In case of a type error, dump all the information inferred by the type-checker before the error. The src.annot file can be used with the emacs commands given in emacs/caml-types.el to display types and other annotations interactively.

-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 -llibname
Pass the -llibname 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 -Ldir 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.

-config
Print the version number of ocamlopt and a detailed summary of its configuration, then exit.

-for-pack module-path
Generate an object file (.cmx and .o/.obj files) that can later be included as a sub-module (with the given access path) of a compilation unit constructed with -pack. For instance, ocamlopt -for-pack P -c A.ml will generate a.cmx and a.o files that can later be used with ocamlopt -pack -o P.cmx a.cmx.

-g
Add debugging information while compiling and linking. This option is required in order to produce stack backtraces when the program terminates on an uncaught exception (see section 10.2).

-i
Cause the compiler to print all defined names (with their inferred types or their definitions) when compiling an implementation (.ml file). No compiled files (.cmo and .cmi files) are produced. 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), compiled object code files (.cmx), and libraries (.cmxa). 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.

If the given directory starts with +, it is taken relative to the standard library directory. For instance, -I +labltk adds the subdirectory labltk of the standard library to the search path.

-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.

```
-intf filename
Compile the file `filename` as an interface file, even if its extension is not `.mli`.

-intf-suffix string
Recognize file names ending with `string` as interface files (instead of the default `.mli`).

-labels
Labels are not ignored in types, labels may be used in applications, and labelled parameters can be given in any order. This is the default.

-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.

-noassert
Do not compile assertion checks. Note that the special form `assert false` is always compiled because it is typed specially. This flag has no effect when linking already-compiled files.

-noautolink
When linking `.cmxa` libraries, ignore `-cclib` and `-ccopt` options potentially contained in the libraries (if these options were given when building the libraries). This can be useful if a library contains incorrect specifications of C libraries or C options; in this case, during linking, set `-noautolink` and pass the correct C libraries and options on the command line.

-nodynlink
Allow the compiler to use some optimizations that are valid only for code that is never dynlinked.

-nolabels
Ignore non-optional labels in types. Labels cannot be used in applications, and parameter order becomes strict.

-o exec-file
Specify the name of the output file produced by the linker. The default output name is `a.out` under Unix and `camlprog.exe` under Windows. If the `-a` option is given, specify the name of the library produced. If the `-pack` option is given, specify the name of the packed object file produced. If the `-output-obj` option is given, specify the name of the output file produced. If the `-shared` option is given, specify the name of plugin 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 18 section 18.7.5.
The name of the output object file is `camlprog.o` by default; it can be set with the `-o` option.
This option can also be used to produce a compiled shared/dynamic library (.so extension, .dll under Windows).

- `-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 17 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 pro-
files.

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.

- `-pack` Build an object file (.cmx and .o/.obj files) and its associated compiled interface (.cmi)
  that combines the .cmx object files given on the command line, making them appear as sub-
  modules of the output .cmx file. The name of the output .cmx file must be given with the `-o`
  option. For instance,

  ```
  ocamlopt -pack -o P.cmx A.cmx B.cmx C.cmx
  ```

  generates compiled files `P.cmx`, `P.o` and `P.cmi` describing a compilation unit having three
  sub-modules A, B and C, corresponding to the contents of the object files `A.cmx`, `B.cmx` and
  `C.cmx`. These contents can be referenced as `P.A`, `P.B` and `P.C` in the remainder of the program.
The .cmx object files being combined must have been compiled with the appropriate
-`for-pack` option. In the example above, `A.cmx`, `B.cmx` and `C.cmx` must have been compiled
with `ocamlopt -for-pack P`

Multiple levels of packing can be achieved by combining `–pack` with `–for-pack`. Consider
the following example:

  ```
  ocamlopt -for-pack P.Q -c A.ml
  ocamlopt -pack -o Q.cmx -for-pack P A.cmx
  ocamlopt -for-pack P -c B.ml
  ocamlopt -pack -o P.cmx Q.cmx B.cmx
  ```

The resulting `P.cmx` object file has sub-modules `P.Q`, `P.Q.A` and `P.B`. 
-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.

-principal
  Check information path during type-checking, to make sure that all types are derived in a principal way. All programs accepted in -principal mode are also accepted in default mode with equivalent types, but different binary signatures.

-rectypes
  Allow arbitrary recursive types during type-checking. By default, only recursive types where the recursion goes through an object type are supported. Note that once you have created an interface using this flag, you must use it again for all dependencies.

-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.

-shared
  Build a plugin (usually .cmxs) that can be dynamically loaded with the Dynlink module. The name of the plugin must be set with the -o option. A plugin can include a number of Caml modules and libraries, and extra native objects (.o, .obj, .a, .lib files). Building native plugins is only supported for some operating system. Under some systems (currently, only Linux AMD 64), all the Caml code linked in a plugin must have been compiled without the -nodynlink flag. Some constraints might also apply to the way the extra native objects have been compiled (under Linux AMD 64, they must contain only position-independent code).

-thread
  Compile or link multithreaded programs, in combination with the system threads library described in chapter 24.

-unsafe
  Turn bound checking off for 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. Additionally, turn off the check for zero divisor in integer division and modulus operations. With -unsafe, an integer division (or modulus) by zero can halt the program or continue with an unspecified result instead of raising a Division_by_zero exception.

-v
  Print the version number of the compiler and the location of the standard library directory, then exit.

-verbose
  Print all external commands before they are executed, in particular invocations of the assembler, C compiler, and linker.

-vnum or -version
  Print the version number of the compiler in short form (e.g. 3.11.0), then exit.
-w warning-list
   Enable, disable, or mark as errors the warnings specified by the argument warning-list. Each
   warning can be enabled or disabled, and each warning can be marked or unmarked. If a
   warning is disabled, it isn’t displayed and doesn’t affect compilation in any way (even if it
   is marked). If a warning is enabled, it is displayed normally by the compiler whenever the
   source code triggers it. If it is enabled and marked, the compiler will stop with an error after
   displaying that warning if the source code triggers it.

   The warning-list argument is a sequence of warning specifiers, with no separators between
   them. A warning specifier is one of the following:

   +num
      Enable warning number num.

   -num
      Disable warning number num.

   @num
      Enable and mark warning number num.

   +num1..num2
      Enable warnings in the given range.

   -num1..num2
      Disable warnings in the given range.

   @num1..num2
      Enable and mark warnings in the given range.

   +letter
      Enable the set of warnings corresponding to letter. The letter may be uppercase or
      lowercase.

   -letter
      Disable the set of warnings corresponding to letter. The letter may be uppercase or
      lowercase.

   @letter
      Enable and mark the set of warnings corresponding to letter. The letter may be upper-
      case or lowercase.

   uppercase-letter
      Enable the set of warnings corresponding to uppercase-letter.

   lowercase-letter
      Disable the set of warnings corresponding to lowercase-letter.

   Warning numbers which are out of the range of warnings that are currently defined are
   ignored. The warning numbers are as follows.

   1  Suspicious-looking start-of-comment mark.

   2  Suspicious-looking end-of-comment mark.

   3  Deprecated syntax.
Fragile pattern matching: matching that will remain complete even if additional constructors are added to one of the variant types matched.

Partially applied function: expression whose result has function type and is ignored.

Label omitted in function application.

Some methods are overridden in the class where they are defined.

Partial match: missing cases in pattern-matching.

Missing fields in a record pattern.

Expression on the left-hand side of a sequence that doesn’t have type "unit" (and that is not a function, see warning number 5).

Redundant case in a pattern matching (unused match case).

Redundant sub-pattern in a pattern-matching.

Override of an instance variable.

Illegal backslash escape in a string constant.

Private method made public implicitly.

Unerasable optional argument.

Undeclared virtual method.

Non-principal type.

Type without principality.

Unused function argument.

Non-returning statement.

Camlp4 warning.

Useless record "with" clause.

Bad module name: the source file name is not a valid OCaml module name.

Pattern-matching with all clauses guarded. Exhaustiveness cannot be checked

Suspicious unused variable: unused variable that is bound with "let" or "as", and doesn’t start with an underscore ("_") character.

Innocuous unused variable: unused variable that is not bound with "let" nor "as", and doesn’t start with an underscore ("_") character.

Wildcard pattern given as argument to a constant constructor.

Unescaped end-of-line in a string constant (non-portable code).

Two labels or constructors of the same name are defined in two mutually recursive types.

The letters stand for the following sets of warnings. Any letter not mentioned here corresponds to the empty set.

A all warnings
C 1, 2
D 3
The default setting is `-w +a-4-6-7-9-27..29`. Note that warnings 5 and 10 are not always triggered, depending on the internals of the type checker.

`-warn-error warning-list`
Mark as errors the warnings specified in the argument `warning-list`. The compiler will stop with an error when one of these warnings is emitted. The `warning-list` has the same meaning as for the `-w` option: a `+` sign (or an uppercase letter) turns the corresponding warnings into errors, a `-` sign (or a lowercase letter) turns them back into warnings, and a `@` sign both enables and marks the corresponding warnings.

Note: it is not recommended to use warning sets (i.e. letters) as arguments to `--warn-error` in production code, because this can break your build when future versions of OCaml add some new warnings.

The default setting is `--warn-error -a` (none of the warnings is treated as an error).

`-where`
Print the location of the standard library, then exit.

`-file`
Process `file` as a file name, even if it starts with a dash (`-`) character.

`-help` or `--help`
Display a short usage summary and exit.

**Options for the IA32 architecture**
The IA32 code generator (Intel Pentium, AMD Athlon) supports the following additional option:

`-ffast-math`
Use the IA32 instructions to compute trigonometric and exponential functions, instead of calling the corresponding library routines. The functions affected are: `atan`, `atan2`, `cos`, `log`, `log10`, `sin`, `sqrt` and `tan`. The resulting code runs faster, but the range of supported arguments and the precision of the result can be reduced. In particular, trigonometric operations `cos`, `sin`, `tan` have their range reduced to `[−2^{64}, 2^{64}]`. 
Options for the AMD64 architecture  The AMD64 code generator (64-bit versions of Intel Pentium and AMD Athlon) supports the following additional options:

- **-fPIC**
  Generate position-independent machine code. This is the default.

- **-fno-PIC**
  Generate position-dependent machine code.

Options for the Sparc architecture  The Sparc code generator supports the following additional options:

- **-march=v8**
  Generate SPARC version 8 code.

- **-march=v9**
  Generate SPARC version 9 code.

The default is to generate code for SPARC version 7, which runs on all SPARC processors.

11.3 Common errors

The error messages are almost identical to those of ocamlc. See section 8.4.

11.4 Running executables produced by ocamlopt

Executables generated by ocamlopt are native, stand-alone executable files that can be invoked directly. They do not depend on the ocamlrun bytecode runtime system nor on dynamically-loaded C/Caml stub libraries.

During execution of an ocamlopt-generated executable, the following environment variables are also consulted:

**OCAMLRUNPARAM**
Same usage as in ocamlrun (see section 10.2), except that option 1 is ignored (the operating system’s stack size limit is used instead).

**CAMLRUNPARAM**
If OCAMLRUNPARAM is not found in the environment, then CAMLRUNPARAM will be used instead. If CAMLRUNPARAM is not found, then the default values will be used.

11.5 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.

- 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.
Stack overflow, typically caused by excessively deep recursion, is handled in one of the following ways, depending on the platform used:

- By raising a `Stack_overflow` exception, like the bytecode compiler does. (IA32/Linux, AMD64/Linux, PowerPC/MacOSX, MS Windows 32-bit ports).
- By aborting the program on a “segmentation fault” signal. (All other Unix systems.)
- By terminating the program silently. (MS Windows 64 bits).

On IA32 processors only (Intel Pentium, AMD Athlon, etc, in 32-bit mode), some intermediate results in floating-point computations are kept in extended precision rather than being rounded to double precision like the bytecode compiler always does. Floating-point results can therefore differ between bytecode and native code; in general, the results obtained with native code are “more exact” (less affected by rounding errors and loss of precision).

On the Alpha processor only, floating-point operations involving infinite or denormalized numbers can abort the program on a “floating-point exception” signal.
Chapter 12

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).

12.1 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 20.)

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.)
12.1.1 Options

The following command-line options are recognized by ocamllex.

-ml   Output code that does not use OCaml’s built-in automata interpreter. Instead, the automaton is encoded by Caml functions. This option mainly is useful for debugging ocamllex, using it for production lexers is not recommended.

-o output-file
     Specify the name of the output file produced by ocamllex. The default is the input file name with its extension replaced by .ml.

-q   Quiet mode. ocamllex normally outputs informational messages to standard output. They are suppressed if option -q is used.

-v or -version
     Print version string and exit.

-vnum
     Print short version number and exit.

-help or --help
     Display a short usage summary and exit.

12.2 Syntax of lexer definitions

The format of lexer definitions is as follows:

```caml
{header}
let ident = regexp ...
rule entrypoint [arg1 ... argn] =
    parse regexp { action }
    | ...
    | regexp { action }
and entrypoint [arg1 ... argn] =
    parse ...
and ...
{trailer}
```

Comments are delimited by (* and *), as in Caml. The parse keyword, can be replaced by the shortest keyword, with the semantic consequences explained below.

12.2.1 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.
12.2.2 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 regular expressions that follow this declaration, the identifier ident can be used as shorthand for regexp.

12.2.3 Entry points

The names of the entry points must be valid identifiers for Caml values (starting with a lowercase letter). Similarly, the arguments arg₁... argₙ must be valid identifiers for Caml. Each entry point becomes a Caml function that takes n + 1 arguments, the extra implicit last argument being of type Lexing.lexbuf. Characters are read from the Lexing.lexbuf argument and matched against the regular expressions provided in the rule, until a prefix of the input matches one of the rule. The corresponding action is then evaluated and returned as the result of the function.

If several regular expressions match a prefix of the input, the “longest match” rule applies: the regular expression that matches the longest prefix of the input is selected. In case of tie, the regular expression that occurs earlier in the rule is selected.

However, if lexer rules are introduced with the shortest keyword in place of the parse keyword, then the “shortest match” rule applies: the shortest prefix of the input is selected. In case of tie, the regular expression that occurs earlier in the rule is still selected. This feature is not intended for use in ordinary lexical analyzers, it may facilitate the use of ocamllex as a simple text processing tool.

12.2.4 Regular expressions

The regular expressions are in the style of lex, with a more Caml-like syntax.

regexp ::= ...

' regular-char | escape-sequence '

A character constant, with the same syntax as OCaml 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, an 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-character} "

A string constant, with the same syntax as OCaml 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 'c₁' - 'c₂' (all characters between c₁ and c₂, 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.

regexp1 # regexp2
   (Difference of character sets). Regular expressions regexp1 and regexp2 must be character sets
   defined with [ ... ] (or a a single character expression or underscore _). Match the difference
   of the two specified character sets.

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.

regexp1 | regexp2
   (Alternative.) Match any string that matches either regexp1 or regexp2

regexp1 regexp2
   (Concatenation.) Match the concatenation of two strings, the first matching regexp1, the
   second matching regexp2.

( regexp )
   Match the same strings as regexp.

ident
   Reference the regular expression bound to ident by an earlier let ident = regexp definition.

regexp as ident
   Bind the substring matched by regexp to identifier ident.

Concerning the precedences of operators, # has the highest precedence, followed by *, + and ?,
then concatenation, then | (alternation), then as.

12.2.5 Actions

The actions are arbitrary Caml expressions. They are evaluated in a context where the identifiers
defined by using the as construct are bound to subparts of the matched string. Additionally,
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
\textsuperscript{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 (i.e. the
offset of the first character of the matched string). The first character read from the input
text has offset 0.

Lexing.lexeme_end lexbuf
Return the absolute position in the input text of the end of the matched string (i.e. the offset
of the first character after the matched string). The first character read from the input text
has offset 0.

entrypoint [exp1... expn] lexbuf
(Where entrypoint is the name of another entry point in the same lexer definition.) Recursively
call the lexer on the given entry point. Notice that lexbuf is the last argument. Useful for
lexing nested comments, for example.

12.2.6 Variables in regular expressions

The as construct is similar to “groups” as provided by numerous regular expression packages. The
type of these variables can be string, char, string option or char option.

We first consider the case of linear patterns, that is the case when all as bound variables are
distinct. In regexp as ident, the type of ident normally is string (or string option) except when
regexp is a character constant, an underscore, a string constant of length one, a character set
specification, or an alternation of those. Then, the type of ident is char (or char option). Option
types are introduced when overall rule matching does not imply matching of the bound sub-pattern.
This is in particular the case of ( regexp as ident ) ? and of regexp1 \| ( regexp2 as ident ).

There is no linearity restriction over as bound variables. When a variable is bound more than
once, the previous rules are to be extended as follows:

- A variable is a char variable when all its occurrences bind char occurrences in the previous
  sense.

- A variable is an option variable when the overall expression can be matched without binding
  this variable.

For instance, in ('a' as x) \| ('a' (_) as x) the variable x is of type char, whereas in
("ab" as x) \| ('a' (_) as x) ? the variable x is of type string option.

In some cases, a successful match may not yield a unique set of bindings. For instance the
matching of aba by the regular expression (('a'|"ab") as x) (("ba"|'a') as y) may result
in binding either x to "ab" and y to "a", or x to "a" and y to "ba". The automata produced
ocamllex on such ambiguous regular expressions will select one of the possible resulting sets of
bindings. The selected set of bindings is purposely left unspecified.

12.2.7 Reserved identifiers

All identifiers starting with __ocaml_lex are reserved for use by ocamllex; do not use any such
identifier in your programs.
12.3 Overview of ocamllyacc

The ocamllyacc 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

```
ocamllyacc 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 ocamllyacc.

12.4 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.

12.4.1 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.

12.4.2 Declarations

Declarations are given one per line. They all start with a % sign.

```
%token constr ... constr
```

Declare the given symbols constr ... constr as tokens (terminal symbols). These symbols are added as constant constructors for the token concrete type.
Chapter 12. Lexer and parser generators (ocamllex, ocamlyacc)

%token < typexpr > constr ... constr
Declare the given symbols constr ... constr 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 typexpr 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 typexpr 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 < typexpr > 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 OCaml compiler (unless the -s option is in effect). The typexpr 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.

The precedence declarations are used in the following way to resolve reduce/reduce and shift/reduce conflicts:

- Tokens and rules have precedences. By default, the precedence of a rule is the precedence of its rightmost terminal. You can override this default by using the %prec directive in the rule.
- A reduce/reduce conflict is resolved in favor of the first rule (in the order given by the source file), and ocamlyacc outputs a warning.
• A shift/reduce conflict is resolved by comparing the precedence of the rule to be reduced with the precedence of the token to be shifted. If the precedence of the rule is higher, then the rule will be reduced; if the precedence of the token is higher, then the token will be shifted.

• A shift/reduce conflict between a rule and a token with the same precedence will be resolved using the associativity: if the token is left-associative, then the parser will reduce; if the token is right-associative, then the parser will shift. If the token is non-associative, then the parser will declare a syntax error.

• When a shift/reduce conflict cannot be resolved using the above method, then ocamlyacc will output a warning and the parser will always shift.

12.4.3 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.

Nonterminal symbols are like regular Caml symbols, except that they cannot end with ' (single quote).

12.4.4 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 raises 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.

12.5 Options

The ocamlyacc command recognizes the following options:

- **-b prefix**
  Name the output files prefix.ml, prefix.mli, prefix.output, instead of the default naming convention.

- **-q**
  This option has no effect.

- **-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.

- **-version**
  Print version string and exit.

- **-vnum**
  Print short version number and exit.

- **-**
  Read the grammar specification from standard input. The default output file names are stdin.ml and stdin.mli.

-- **file**
  Process file as the grammar specification, even if its name starts with a dash (-) character. This option must be the last on the command line.

At run-time, the ocamlyacc-generated parser can be debugged by setting the p option in the OCAMLRUNPARAM environment variable (see section [10.2](#)). This causes the pushdown automaton executing the parser to print a trace of its action (tokens shifted, rules reduced, etc). The trace mentions rule numbers and state numbers that can be interpreted by looking at the file grammar.output generated by ocamlyacc -v.

12.6 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:

```plaintext
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS /* lowest precedence */
```
Here is the definition for the corresponding lexer:

```ocaml
(* File lexer.mll *)
{
  open Parser  (* The type token is defined in parser.mli *)
  exception Eof
}

rule token = parse
  [' ' '	'] { token lexbuf }  (* skip blanks *)
| ['n'] { EOL }
| ['0-'9]+ as lxm { INT(int_of_string lxm) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIV }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { raise Eof }
```

Here is the main program, that combines the parser with the lexer:

```ocaml
(* 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:

    ocamlex 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

12.7 Common errors

ocamlex: transition table overflow, automaton is too big

The deterministic automata generated by ocamlex 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' '_'] * as id
  { IDENT id}

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' '_'] * as id
  { try
      Hashtbl.find keyword_table id
    with Not_found ->
      IDENT id }

ocamllex: Position memory overflow, too many bindings

The deterministic automata generated by ocamllex maintains a table of positions inside the scanned lexer buffer. The size of this table is limited to at most 255 cells. This error should not show up in normal situations.
Chapter 13

Dependency generator (ocamldep)

The ocamldep command scans a set of OCaml 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.

13.1 Options

The following command-line options are 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 from 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.

-ml-synonym .ext
Consider the given extension (with leading dot) to be a synonym for .ml.

-mli-synonym .ext
Consider the given extension (with leading dot) to be a synonym for .mli.

-modules
Output raw dependencies of the form
filename: Module1 Module2 ... ModuleN

where Module1, ..., ModuleN are the names of the compilation units referenced within the file filename, but these names are not resolved to source file names. Such raw dependencies cannot be used by make, but can be post-processed by other tools such as Omake.

-native
Generate dependencies for a pure native-code program (no bytecode version). When an implementation file (.ml file) has no explicit interface file (.mli file), ocamldep generates dependencies on the bytecode compiled file (.cmo file) to reflect interface changes. This can cause unnecessary bytecode recomputations for programs that are compiled to native-code only. The flag -native causes dependencies on native compiled files (.cmx) to be generated instead of on .cmo files. (This flag makes no difference if all source files have explicit .mli interface files.)

-pp command
Cause ocamldep to call the given command as a preprocessor for each source file.

-slash
Under Windows, use a forward slash (/) as the path separator instead of the usual backward slash (\). Under Unix, this option does nothing.

-version
Print version string and exit.

-vnum
Print short version number and exit.

-help or --help
Display a short usage summary and exit.

13.2 A typical Makefile

Here is a template Makefile for a OCaml 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 14

The browser/editor (ocamlbrowser)

This chapter describes OCamlBrowser, a source and compiled interface browser, written using LablTk. This is a useful companion to the programmer.

Its functions are:

- navigation through OCaml’s modules (using compiled interfaces).
- source editing, type-checking, and browsing.
- integrated OCaml shell, running as a subprocess.

14.1 Invocation

The browser is started by the command `ocamlbrowser`, as follows:

```
    ocamlbrowser options
```

The following command-line options are recognized by `ocamlbrowser`.

- `-I directory` Add the given directory to the list of directories searched for source and compiled files. By default, only the standard library directory is searched. The standard library can also be changed by setting the `OCAMLLIB` environment variable.

- `-nolabels` Ignore non-optional labels in types. Labels cannot be used in applications, and parameter order becomes strict.

- `-oldui` Old multi-window interface. The default is now more like Smalltalk’s class browser.

- `-rectypes` Allow arbitrary recursive types during type-checking. By default, only recursive types where the recursion goes through an object type are supported.

- `-version` Print version string and exit.
-vnum
  Print short version number and exit.

-w warning-list
  Enable or disable warnings according to the argument warning-list.

Most options can also be modified inside the application by the Modules - Path editor and Compiler - Preferences commands. They are inherited when you start a toplevel shell.

14.2 Viewer

This is the first window you get when you start OCamlBrowser. It displays a search window, and the list of modules in the load path. At the top a row of menus.

- File - Open and File - Editor give access to the editor.

- File - Shell creates an OCaml subprocess in a shell.

- View - Show all defs displays the signature of the currently selected module.

- View - Search entry shows/hides the search entry just below the menu bar.

- Modules - Path editor changes the load path. Modules - Reset cache rescans the load path and resets the module cache. Do it if you recompile some interface, or get confused about what is in the cache.

- Modules - Search symbol allows to search a symbol either by its name, like the bottom line of the viewer, or, more interestingly, by its type. Exact type searches for a type with exactly the same information as the pattern (variables match only variables). Included type allows to give only partial information: the actual type may take more arguments and return more results, and variables in the pattern match anything. In both cases, argument and tuple order is irrelevant\(^1\), and unlabeled arguments in the pattern match anything. In both cases, argument and tuple order is irrelevant\(^1\), and unlabeled arguments in the pattern match anything.

- The Search entry just below the menu bar allows one to search for an identifier in all modules (wildcards “?” and “*” allowed). If you choose the type option, the search is done by type inclusion (cf. Search Symbol - Included type).

- The Close all button is there to dismiss the windows created by the Detach button. By double-clicking on it you will quit the browser.

14.3 Module browsing

You select a module in the leftmost box by either clicking on it or pressing return when it is selected. Fast access is available in all boxes pressing the first few letter of the desired name. Double-clicking / double-return displays the whole signature for the module.

\(^1\)To avoid combinatorial explosion of the search space, optional arguments in the actual type are ignored in the actual if (1) there are too many of them, and (2) they do not appear explicitly in the pattern.
Defined identifiers inside the module are displayed in a box to the right of the previous one. If you click on one, this will either display its contents in another box (if this is a sub-module) or display the signature for this identifier below.

Signatures are clickable. Double clicking with the left mouse button on an identifier in a signature brings you to its signature. A single click on the right button pops up a menu displaying the type declaration for the selected identifier. Its title, when selectable, also brings you to its signature.

At the bottom, a series of buttons, depending on the context.

- **Detach** copies the currently displayed signature in a new window, to keep it.
- **Impl** and **Intf** bring you to the implementation or interface of the currently displayed signature, if it is available.

Control-S lets you search a string in the signature.

### 14.4 File editor

You can edit files with it, if you’re not yet used to emacs. Otherwise you can use it as a browser, making occasional corrections.

The **Edit** menu contains commands for jump (C-g), search (C-s), and sending the current phrase (or selection if some text is selected) to a sub-shell (M-x). For this last option, you may choose the shell via a dialog.

Essential functions are in the **Compiler** menu.

- **Preferences** opens a dialog to set internals of the editor and type-checker.
- **Lex** adds colors according to lexical categories.
- **Typecheck** verifies typing, and memorizes to let one see an expression’s type by double-clicking on it. This is also valid for interfaces. If an error occurs, the part of the interface preceding the error is computed.
  
  After typechecking, pressing the right button pops up a menu giving the type of the pointed expression, and eventually allowing to follow some links.

- **Clear errors** dismisses type-checker error messages and warnings.
- **Signature** shows the signature of the current file (after type checking).

### 14.5 Shell

When you create a shell, a dialog is presented to you, letting you choose which command you want to run, and the title of the shell (to choose it in the Editor).

The executed subshell is given the current load path.

- **File** use a source file or load a bytecode file. You may also import the browser’s path into the subprocess.
• **History** M-p and M-n browse up and down.

• **Signal** C-c interrupts, and you can also kill the subprocess.
Chapter 15

The documentation generator (ocamldoc)

This chapter describes OCamldoc, a tool that generates documentation from special comments embedded in source files. The comments used by OCamldoc are of the form (**...**) and follow the format described in section 15.2.

OCamldoc can produce documentation in various formats: HTML, \LaTeX, TeXinfo, Unix man pages, and dot dependency graphs. Moreover, users can add their own custom generators, as explained in section 15.3.

In this chapter, we use the word element to refer to any of the following parts of an OCaml source file: a type declaration, a value, a module, an exception, a module type, a type constructor, a record field, a class, a class type, a class method, a class value or a class inheritance clause.

15.1 Usage

15.1.1 Invocation

OCamldoc is invoked via the command ocamldoc, as follows:

```
ocamldoc options sourcefiles
```

Options for choosing the output format

The following options determine the format for the generated documentation.

- `html`
  Generate documentation in HTML default format. The generated HTML pages are stored in the current directory, or in the directory specified with the `-d` option. You can customize the style of the generated pages by editing the generated `style.css` file, or by providing your own style sheet using option `-css-style`. The file `style.css` is not generated if it already exists.

- `latex`
  Generate documentation in \LaTeX default format. The generated \LaTeX document is saved in.
file `ocamldoc.out`, or in the file specified with the `-o` option. The document uses the style file `ocamldoc.sty`. This file is generated when using the `-latex` option, if it does not already exist. You can change this file to customize the style of your \LaTeX{} documentation.

-texi
Generate documentation in TeXinfo default format. The generated \LaTeX{} document is saved in file `ocamldoc.out`, or in the file specified with the `-o` option.

-man
Generate documentation as a set of Unix `man` pages. The generated pages are stored in the current directory, or in the directory specified with the `-d` option.

-dot
Generate a dependency graph for the toplevel modules, in a format suitable for displaying and processing by `dot`. The `dot` tool is available from [http://www.research.att.com/sw/tools/graphviz/](http://www.research.att.com/sw/tools/graphviz/). The textual representation of the graph is written to the file `ocamldoc.out`, or to the file specified with the `-o` option. Use `dot ocamldoc.out` to display it.

-g file.cm[o,a]
Dynamically load the given file, which defines a custom documentation generator. See section 15.4.1 This option is supported by the `ocamldoc` command, but not by its native-code version `ocamldoc.opt`. If the given file is a simple one and does not exist in the current directory, then `ocamldoc` looks for it in the custom generators default directory, and in the directories specified with optional `-i` options.

-customdir
Display the custom generators default directory.

-i directory
Add the given directory to the path where to look for custom generators.

General options

-d dir
Generate files in directory `dir`, rather than the current directory.

-dump file
Dump collected information into `file`. This information can be read with the `-load` option in a subsequent invocation of `ocamldoc`.

-hide modules
Hide the given complete module names in the generated documentation. `modules` is a list of complete module names are separated by ’,’ without blanks. For instance: Pervasives,M2.M3.

-inv-merge-ml-mli
Reverse the precedence of implementations and interfaces when merging. All elements in
implementation files are kept, and the \texttt{-m} option indicates which parts of the comments in interface files are merged with the comments in implementation files.

\textbf{-keep-code}

Always keep the source code for values, methods and instance variables, when available. The source code is always kept when a \texttt{.ml} file is given, but is by default discarded when a \texttt{.mli} is given. This option allows to always keep the source code.

\textbf{-load \textit{file}}

Load information from \textit{file}, which has been produced by \texttt{ocamldoc -dump}. Several \texttt{-load} options can be given.

\textbf{-m \textit{flags}}

Specify merge options between interfaces and implementations. (see section\ref{sec:merge-flags} for details). \textit{flags} can be one or several of the following characters:

\begin{itemize}
  \item \texttt{d} merge description
  \item \texttt{a} merge \texttt{@author}
  \item \texttt{v} merge \texttt{@version}
  \item \texttt{l} merge \texttt{@see}
  \item \texttt{s} merge \texttt{@since}
  \item \texttt{b} merge \texttt{@before}
  \item \texttt{o} merge \texttt{@deprecated}
  \item \texttt{p} merge \texttt{@param}
  \item \texttt{e} merge \texttt{@raise}
  \item \texttt{r} merge \texttt{@return}
  \item \texttt{A} merge everything
\end{itemize}

\textbf{-no-custom-tags}

Do not allow custom \texttt{@-tags} (see section\ref{sec:custom-tags}).

\textbf{-no-stop}

Keep elements placed after/between the (**/**) special comment(s) (see section\ref{sec:stop}).

\textbf{-o \textit{file}}

Output the generated documentation to \textit{file} instead of \texttt{ocamldoc.out}. This option is meaningful only in conjunction with the \texttt{-latex}, \texttt{-texi}, or \texttt{-dot} options.

\textbf{-pp \textit{command}}

Pipe sources through preprocessor \textit{command}.

\textbf{-impl \textit{filename}}

Process the file \textit{filename} as an implementation file, even if its extension is not \texttt{.ml}.

\textbf{-intf \textit{filename}}

Process the file \textit{filename} as an interface file, even if its extension is not \texttt{.mli}.
-text filename
    Process the file filename as a text file, even if its extension is not .txt.

-sort
    Sort the list of top-level modules before generating the documentation.

-stars
    Remove blank characters until the first asterisk ('*') in each line of comments.

-t title
    Use title as the title for the generated documentation.

-intro file
    Use content of file as ocamldoc text to use as introduction (HTML, \LaTeX and TeXinfo only).
    For HTML, the file is used to create the whole index.html file.

-v
    Verbose mode. Display progress information.

-version
    Print version string and exit.

-vnum
    Print short version number and exit.

-warn-error
    Treat Ocamldoc warnings as errors.

-hide-warnings
    Do not print OCamldoc warnings.

-help or --help
    Display a short usage summary and exit.

Type-checking options

OCamldoc calls the OCaml type-checker to obtain type information. The following options impact
the type-checking phase. They have the same meaning as for the ocamlc and ocamlopt commands.

-I directory
    Add directory to the list of directories search for compiled interface files (.cmi files).

-nolabels
    Ignore non-optional labels in types.

-rectypes
    Allow arbitrary recursive types. (See the -rectypes option to ocamlc.)
Chapter 15. The documentation generator (ocamldoc)

Options for generating HTML pages

The following options apply in conjunction with the \texttt{-html} option:

\texttt{-all-params}
Display the complete list of parameters for functions and methods.

\texttt{-css-style \textit{filename}}
Use \textit{filename} as the Cascading Style Sheet file.

\texttt{-colorize-code}
Colorize the OCaml code enclosed in \texttt{[ ]} and \texttt{\{[ ]\}}, using colors to emphasize keywords, etc. If the code fragments are not syntactically correct, no color is added.

\texttt{-index-only}
Generate only index files.

\texttt{-short-functors}
Use a short form to display functors: \texttt{module M : functor (A:Module) -> functor (B:Module2) -> sig .. end} is displayed as \texttt{module M (A:Module) (B:Module2) : sig .. end}.

Options for generating \LaTeX{} files

The following options apply in conjunction with the \texttt{-latex} option:

\texttt{-latex-value-prefix \textit{prefix}}
Give a prefix to use for the labels of the values in the generated \LaTeX{} document. The default prefix is the empty string. You can also use the options \texttt{-latex-type-prefix}, \texttt{-latex-exception-prefix}, \texttt{-latex-module-prefix}, \texttt{-latex-module-type-prefix}, \texttt{-latex-class-prefix}, \texttt{-latex-class-type-prefix}, \texttt{-latex-attribute-prefix} and \texttt{-latex-method-prefix}.

These options are useful when you have, for example, a type and a value with the same name. If you do not specify prefixes, \LaTeX{} will complain about multiply defined labels.

\texttt{-latextitle \textit{n},style}
Associate style number \textit{n} to the given \LaTeX{} sectioning command \textit{style}, e.g. \texttt{section} or \texttt{subsection}. (\LaTeX{} only.) This is useful when including the generated document in another \LaTeX{} document, at a given sectioning level. The default association is 1 for \texttt{section}, 2 for \texttt{subsection}, 3 for \texttt{subsubsection}, 4 for \texttt{paragraph} and 5 for \texttt{subparagraph}.

\texttt{-noheader}
Suppress header in generated documentation.

\texttt{-notoc}
Do not generate a table of contents.

\texttt{-notrailer}
Suppress trailer in generated documentation.
-sepfiles
  Generate one .tex file per toplevel module, instead of the global ocamldoc.out file.

Options for generating TeXinfo files

The following options apply in conjunction with the -texi option:

-esc8
  Escape accented characters in Info files.

-info-entry
  Specify Info directory entry.

-info-section
  Specify section of Info directory.

-noheader
  Suppress header in generated documentation.

-noindex
  Do not build index for Info files.

-notrailler
  Suppress trailer in generated documentation.

Options for generating dot graphs

The following options apply in conjunction with the -dot option:

-dot-colors colors
  Specify the colors to use in the generated dot code. When generating module dependencies, ocamldoc uses different colors for modules, depending on the directories in which they reside. When generating types dependencies, ocamldoc uses different colors for types, depending on the modules in which they are defined. colors is a list of color names separated by ’,’ as in Red,Blue,Green. The available colors are the ones supported by the dot tool.

-dot-include-all
  Include all modules in the dot output, not only modules given on the command line or loaded with the -load option.

-dot-reduce
  Perform a transitive reduction of the dependency graph before outputting the dot code. This can be useful if there are a lot of transitive dependencies that clutter the graph.

-dot-types
  Output dot code describing the type dependency graph instead of the module dependency graph.
Options for generating man files

The following options apply in conjunction with the -man option:

-man-mini
Generate man pages only for modules, module types, classes and class types, instead of pages for all elements.

-man-suffix suffix
Set the suffix used for generated man filenames. Default is ’3o’, as in List.3o.

-man-section section
Set the section number used for generated man filenames. Default is ’3’.

15.1.2 Merging of module information

Information on a module can be extracted either from the .mli or .ml file, or both, depending on the files given on the command line. When both .mli and .ml files are given for the same module, information extracted from these files is merged according to the following rules:

- Only elements (values, types, classes, ...) declared in the .mli file are kept. In other terms, definitions from the .ml file that are not exported in the .mli file are not documented.

- Descriptions of elements and descriptions in @-tags are handled as follows. If a description for the same element or in the same @-tag of the same element is present in both files, then the description of the .ml file is concatenated to the one in the .mli file, if the corresponding -m flag is given on the command line. If a description is present in the .ml file and not in the .mli file, the .ml description is kept. In either case, all the information given in the .mli file is kept.

15.1.3 Coding rules

The following rules must be respected in order to avoid name clashes resulting in cross-reference errors:

- In a module, there must not be two modules, two module types or a module and a module type with the same name. In the default HTML generator, modules ab and AB will be printed to the same file on case insensitive file systems.

- In a module, there must not be two classes, two class types or a class and a class type with the same name.

- In a module, there must not be two values, two types, or two exceptions with the same name.

- Values defined in tuple, as in let (x,y,z) = (1,2,3) are not kept by OCamlDoc.

- Avoid the following construction:
open Foo (* which has a module Bar with a value x *)
module Foo =
  struct
    module Bar =
      struct
        let x = 1
      end
    end
  end
let dummy = Bar.x

In this case, OCamlDoc will associate Bar.x to the x of module Foo defined just above, instead of to the Bar.x defined in the opened module Foo.

15.2 Syntax of documentation comments

Comments containing documentation material are called special comments and are written between (** and *). Special comments must start exactly with (**. Comments beginning with ( and more than two * are ignored.

15.2.1 Placement of documentation comments

OCamlDoc can associate comments to some elements of the language encountered in the source files. The association is made according to the locations of comments with respect to the language elements. The locations of comments in .mli and .ml files are different.

Comments in .mli files

A special comment is associated to an element if it is placed before or after the element. A special comment before an element is associated to this element if:

- There is no blank line or another special comment between the special comment and the element. However, a regular comment can occur between the special comment and the element.
- The special comment is not already associated to the previous element.
- The special comment is not the first one of a toplevel module.

A special comment after an element is associated to this element if there is no blank line or comment between the special comment and the element.

There are two exceptions: for type constructors and record fields in type definitions, the associated comment can only be placed after the constructor or field definition, without blank lines or other comments between them. The special comment for a type constructor with another type constructor following must be placed before the ‘—’ character separating the two constructors.

The following sample interface file foo.mli illustrates the placement rules for comments in .mli files.
(** The first special comment of the file is the comment associated with the whole module. *)

(** Special comments can be placed between elements and are kept by the OCamlDoc tool, but are not associated to any element. @-tags in these comments are ignored. *)

(*******************************************************************)
(** Comments like the one above, with more than two asterisks, are ignored. *)

(** The comment for function f. *)
val f : int -> int -> int
(** The continuation of the comment for function f. *)

(** Comment for exception My_exception, even with a simple comment between the special comment and the exception. *)
(* Hello, I’m a simple comment :) *)
exception My_exception of (int -> int) * int

(** Comment for type weather *)
type weather =
| Rain of int (** The comment for constructor Rain *)
| Sun (** The comment for constructor Sun *)

(** Comment for type weather2 *)
type weather2 =
| Rain of int (** The comment for constructor Rain *)
| Sun (** The comment for constructor Sun *)
(** I can continue the comment for type weather2 here because there is already a comment associated to the last constructor. *)

(** The comment for type my_record *)
type my_record = {
  val foo : int ; (** Comment for field foo *)
  val bar : string ; (** Comment for field bar *)
}
(** Continuation of comment for type my_record *)

(** Comment for foo *)
val foo : string
(** This comment is associated to foo and not to bar. *)
val bar : string
(** This comment is associated to bar. *)
(** The comment for class my_class *)
class my_class :
  object
    (** A comment to describe inheritance from cl *)
    inherit cl

    (** The comment for attribute tutu *)
    val mutable tutu : string

    (** The comment for attribute toto. *)
    val toto : int

    (** This comment is not attached to titi since there is a blank line before titi, but is kept as a comment in the class. *)
    val titi : string

    (** Comment for method toto *)
    method toto : string

    (** Comment for method m *)
    method m : float -> int
end

(** The comment for the class type my_class_type *)
class type my_class_type =
  object
    (** The comment for variable x. *)
    val mutable x : int

    (** The commend for method m. *)
    method m : int -> int
end

(** The comment for module Foo *)
module Foo =
  struct
    (** The comment for x *)
    val x : int

    (** A special comment that is kept but not associated to any element *)
  end
Chapter 15. The documentation generator (ocamldoc)

(** The comment for module type my_module_type. *)
module type my_module_type =
  sig
    (** The comment for value x. *)
    val x : int

    (** The comment for module M. *)
    module M =
      struct
        (** The comment for value y. *)
        val y : int

        (* ... *)
      end

  end

end

Comments in .ml files

A special comment is associated to an element if it is placed before the element and there is no blank line between the comment and the element. Meanwhile, there can be a simple comment between the special comment and the element. There are two exceptions, for type constructors and record fields in type definitions, whose associated comment must be placed after the constructor or field definition, without blank line between them. The special comment for a type constructor with another type constructor following must be placed before the ‘—’ character separating the two constructors.

The following example of file toto.ml shows where to place comments in a .ml file.

(** The first special comment of the file is the comment associated to the whole module. *)

(** The comment for function f. *)
let f x y = x + y

(** This comment is not attached to any element since there is another special comment just before the next element. *)

(** Comment for exception My_exception, even with a simple comment between the special comment and the exception. *)
(* A simple comment. *)
exception My_exception of (int -> int) * int

(** Comment for type weather. *)
type weather =
  | Rain of int (** The comment for constructor Rain. *)
type my_record = {
  val foo : int ; (** Comment for field foo *)
  val bar : string ; (** Comment for field bar *)
}

class my_class =
  object
    (** A comment to describe inheritance from cl *)
    inherit cl

    (** The comment for the instance variable tutu *)
    val mutable tutu = "tutu"
    (** The comment for toto *)
    val toto = 1
    val titi = "titi"
    (** Comment for method toto *)
    method toto = tutu ^ "!"
    (** Comment for method m *)
    method m (f : float) = 1
  end

class type my_class_type =
  object
    (** The comment for the instance variable x. *)
    val mutable x : int
    (** The commend for method m. *)
    method m : int -> int
  end

module Foo =
  struct
    (** The comment for x *)
    val x : int
    (** A special comment in the class, but not associated to any element. *)
  end

module type my_module_type =
  sig
Chapter 15. The documentation generator (ocamldoc)

Chapter 15. The documentation generator (ocamldoc)

(* Comment for value x. *)
val x : int
(* ... *)
end

15.2.2 The Stop special comment

The special comment (**/**) tells OCamldoc to discard elements placed after this comment, up to the end of the current class, class type, module or module type, or up to the next stop comment. For instance:

class type foo =
  object
    (** comment for method m *)
    method m : string
    (**/**)
    (** This method won’t appear in the documentation *)
    method bar : int
  end

(** This value appears in the documentation, since the Stop special comment in the class does not affect the parent module of the class.*)
val foo : string

(**/**)
(** The value bar does not appear in the documentation.*)
val bar : string
(**/**)

(** The type t appears since in the documentation since the previous stop comment toggled off the "no documentation mode". *)
type t = string

The -no-stop option to ocamldoc causes the Stop special comments to be ignored.

15.2.3 Syntax of documentation comments

The inside of documentation comments (**...**) consists of free-form text with optional formatting annotations, followed by optional tags giving more specific information about parameters, version, authors, ... The tags are distinguished by a leading @ character. Thus, a documentation comment has the following shape:

(** The comment begins with a description, which is text formatted according to the rules described in the next section.
The description continues until the first non-escaped '@' character.
@author Mr Smith
@param x description for parameter x
*)

Some elements support only a subset of all @-tags. Tags that are not relevant to the documented element are simply ignored. For instance, all tags are ignored when documenting type constructors, record fields, and class inheritance clauses. Similarly, a @param tag on a class instance variable is ignored.

At last, (**) is the empty documentation comment.

15.2.4 Text formatting

Here is the BNF grammar for the simple markup language used to format text descriptions.
text ::= (text_element)*

text_element ::= 
| {[0-9]+ text} | format text as a section header; the integer following \{ indicates the sectioning level.
| {[0-9]+: label text} | same, but also associate the name label to the current point. This point can be referenced by its fully-qualified label in a \{! command, just like any other element.
| {b text} | set text in bold.
| {i text} | set text in italic.
| {e text} | emphasize text.
| {C text} | center text.
| {L text} | left align text.
| {R text} | right align text.
| {ul list} | build a list.
| {ol list} | build an enumerated list.
| {\string}text | put a link to the given address (given as a string) on the given text.
| {\string} | set the given \string in source code style.
| {\string} | set the given \string in preformatted source code style.
| {\% string \%} | take the given \string as raw \LaTeX code.
| {!string} | insert a reference to the element named \string. \string must be a fully qualified element name, for example \Foo.Bar.t. The kind of the referenced element can be forced (useful when various elements have the same qualified name) with the following syntax: \{!kind: \Foo.Bar.t\} where kind can be module, modtype, class, classtype, val, type, exception, attribute, method or section.
| {!modules: string string ...} | insert an index table for the given module names. Used in HTML only.
| {!indexlist} | insert a table of links to the various indexes (types, values, modules, ...). Used in HTML only.
| ^text | set text in superscript.
| _text | set text in subscript.
| escaped_string | typeset the given string as is; special characters (', ', '[', '] and '@') must be escaped by a `\`
| blank_line | force a new line.

list ::= 
| (\{- text\})+ 
| (\{li text\})+

A shortcut syntax exists for lists and enumerated lists:

(** Here is a \{b list\)
- item 1
- item 2
- item 3

The list is ended by the blank line.*)

is equivalent to:

(** Here is a {b list}
{ul {- item 1}
{- item 2}
{- item 3}}
The list is ended by the blank line.*)

The same shortcut is available for enumerated lists, using ’+’ instead of ’-’. Note that only one list can be defined by this shortcut in nested lists.

In the description of a value, type, exception, module, module type, class or class type, the first sentence is sometimes used in indexes, or when just a part of the description is needed. The first sentence is composed of the first characters of the description, until

- the first dot followed by a blank, or
- the first blank line

outside of the following text formatting: {ul list}, {ol list}, [string], {{string}}, {v string v}, {%^ string%}, {^string}, {_text}. {_% text}.

15.2.5 Documentation tags (@-tags)

Predefined tags

The following table gives the list of predefined @-tags, with their syntax and meaning.
**Chapter 15. The documentation generator (ocamldoc)**

### Custom Tags

You can use custom tags in the documentation comments, but they will have no effect if the generator used does not handle them. To use a custom tag, for example `foo`, just put `@foo` with some text in your comment, as in:

```plaintext
(** My comment to show you a custom tag.
@foo this is the text argument to the [foo] custom tag.
*)
```

To handle custom tags, you need to define a custom generator, as explained in section [15.3.2](#custom-generators).

### 15.3 Custom Generators

OCamldoc operates in two steps:

1. analysis of the source files;
2. generation of documentation, through a documentation generator, which is an object of class `Odoc_args.class_generator`.

Users can provide their own documentation generator to be used during step 2 instead of the default generators. All the information retrieved during the analysis step is available through the `Odoc_info` module, which gives access to all the types and functions representing the elements found in the given modules, with their associated description.

<table>
<thead>
<tr>
<th>Tag</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>@author string</td>
<td>The author of the element. One author by @author tag. There may be several @author tags for the same element.</td>
</tr>
<tr>
<td>@deprecated text</td>
<td>The text should describe when the element was deprecated, what to use as a replacement, and possibly the reason for deprecation.</td>
</tr>
<tr>
<td>@param id text</td>
<td>Associate the given description (text) to the given parameter name id. This tag is used for functions, methods, classes and functors.</td>
</tr>
<tr>
<td>@raise Exc text</td>
<td>Explain that the element may raise the exception Exc.</td>
</tr>
<tr>
<td>@return text</td>
<td>Describe the return value and its possible values. This tag is used for functions and methods.</td>
</tr>
<tr>
<td>@see &lt;url&gt; text</td>
<td>Add a reference to the URL between '&lt;' and '&gt;' with the given text as comment.</td>
</tr>
<tr>
<td>@see 'filename' text</td>
<td>Add a reference to the given file name (written between single quotes), with the given text as comment.</td>
</tr>
<tr>
<td>@see &quot;document name&quot; text</td>
<td>Add a reference to the given document name (written between double quotes), with the given text as comment.</td>
</tr>
<tr>
<td>@since string</td>
<td>Indicate when the element was introduced.</td>
</tr>
<tr>
<td>@before v text</td>
<td>Associate the given description (text) to the given version v in order to document compatibility issues.</td>
</tr>
<tr>
<td>@version string</td>
<td>The version number for the element.</td>
</tr>
</tbody>
</table>
The files you can use to define custom generators are installed in the `ocamldoc` sub-directory of the OCaml standard library.

### 15.3.1 The generator class

A generator class is a class of type `Odoc_args.doc_generator`. It has only one method

```ocaml
method generator : Odoc_info.Module.t_module list -> unit
```

This method will be called with the list of analysed and possibly merged `Odoc_info.t_module` structures. Of course the class can have other methods, but the object of this class must be coerced to `Odoc_args.doc_generator` before being passed to the function `Odoc_args.set_doc_generator : Odoc_args.doc_generator -> unit` which installs the new documentation generator.

The following example shows how to define and install a new documentation generator. See the `odoc_fhtml` generator (in the Ocamldoc Hump) for a complete example.

```ocaml
class my_doc_gen =
  object
    (* ... *)

    method generate module_list =
      (* ... *)
      ()
      (* ... *)
  end

let my_generator = new my_doc_gen
let _ = Odoc_args.set_doc_generator (my_generator :> Odoc_args.doc_generator)
```

Note: The new class can inherit from `Odoc_html.html`, `Odoc_latex.latex`, `Odoc_man.man`, `Odoc_texi.texi` or `Odoc_dot.dot`, and redefine only some methods to benefit from the existing methods.

### 15.3.2 Handling custom tags

Making a custom generator handle custom tags (see [15.2.5](#)) is very simple.

**For HTML**

Here is how to develop a HTML generator handling your custom tags.

The class `Odoc_html.html` inherits from the class `Odoc_html.info`, containing a field `tag_functions` which is a list pairs composed of a custom tag (e.g. `'foo'`) and a function taking a `text` and returning HTML code (of type `string`). To handle a new tag `bar`, create a HTML generator class from the existing one and complete the `tag_functions` field:

```ocaml
class my_gen =
  object(self)
```
Chapter 15. The documentation generator (ocamldoc)

inherit Odoc_html.html

(** Return HTML code for the given text of a bar tag. *)
method html_of_bar t = (* your code here *)

initializer
  tag_functions <- ("bar", self#html_of_bar) :: tag_functions
end

Another method of the class Odoc_html.info will look for the function associated to a custom tag and apply it to the text given to the tag. If no function is associated to a custom tag, then the method prints a warning message on stderr.

For other generators

As for the HTML custom generator, you can define a new \LaTeX (resp. man) generator by inheriting from the class Odoc_latex.latex (resp. Odoc_man.man) and adding your own tag handler to the field tag_functions.

15.4 Adding command line options

The command line analysis is performed after loading the module containing the documentation generator, thus allowing command line options to be added to the list of existing ones. Adding an option can be done with the function

Odoc_args.add_option : string * Arg.spec * string -> unit

Note: Existing command line options can be redefined using this function.

15.4.1 Compilation and usage

Defining a custom generator class in one file

Let custom.ml be the file defining a new generator class. Compilation of custom.ml can be performed by the following command:

ocamlc -I +ocamldoc -c custom.ml

The file custom.cmo is created and can be used this way:

ocamldoc -g custom.cmo other-options source-files

It is important not to give the -html or any other option selecting a built in generator to ocamldoc, which would result in using this generator instead of the one you just loaded.

Defining a custom generator class in several files

It is possible to define a generator class in several modules, which are defined in several files file1.ml[i], file2.ml[i], ..., fileN.ml[i]. A .cma library file must be created, including all these files.

The following commands create the custom.cma file from files file1.ml[i], ..., fileN.ml[i] :

ocamlc -I +ocamldoc -c file1.ml[i]
ocamlc -I +ocamldoc -c file2.ml[i]

...
... 
```
ocamlc -I +ocamldoc -c fileN.ml
ocamlc -o custom.cma -a file1.cmo file2.cmo ... fileN.cmo
```
Then, the following command uses `custom.cma` as custom generator:
```
ocamldoc -g custom.cma other-options source-files
```
Again, it is important not to give the `-html` or any other option selecting a built in generator to `ocamldoc`, which would result in using this generator instead of the one you just loaded.
Chapter 16

The debugger (ocamldump)

This chapter describes the OCaml source-level replay debugger ocamldump.

Unix:
The debugger is available on Unix systems that provide BSD sockets.

Windows:
The debugger is available under the Cygwin port of OCaml, but not under the native Win32 ports.

16.1 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.

16.2 Invocation

16.2.1 Starting the debugger

The OCaml debugger is invoked by running the program ocamldump with the name of the bytecode executable file as first argument:

    ocamldump [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:

- c count
  Set the maximum number of simultaneously live checkpoints to count.

231
-cd  dir
   Run the debugger program from the working directory dir, instead of the current directory.
   (See also the cd command.)

-emacs
   Tell the debugger it is executed under Emacs. (See section 16.10 for information on how to
   run the debugger under Emacs.)

-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 com-
   mand set socket (section 16.8.6) for the format of socket.

-version
   Print version string and exit.

-vnum
   Print short version number and exit.

-help or --help
   Display a short usage summary and exit.

16.2.2 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.

16.3 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.
16.3.1 Getting help

The OCaml 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.

16.3.2 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.

16.4 Executing a program

16.4.1 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 bowtie) are located in the source code:

• Following a function application:

(f arg)bowtie

• On entrance to a function:
fun x y z -> bowtie ...

- On each case of a pattern-matching definition (function, match...with construct, try...with construct):

function pat1 -> bowtie expr1
| ... | patN -> bowtie exprN

- Between subexpressions of a sequence:

expr1; bowtie expr2; bowtie ...; bowtie exprN

- In the two branches of a conditional expression:

if cond then bowtie expr1 else bowtie expr2

- At the beginning of each iteration of a loop:

while cond do bowtie body done
for i = a to b do bowtie 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.

16.4.2 Starting the debugged program

The debugger starts executing the debugged program only when needed. This allows setting breakpoints 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 16.5).

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).
16.4.3 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.

16.4.4 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.
16.4.5 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.

16.5 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 16.10).

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.

16.6 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.

16.7 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 OCaml expressions, as described by the following grammar:

\[
\text{simple-expr ::= lowercase-ident} \\
\quad \mid \{\text{capitalized-ident .}\} \text{lowercase-ident} \\
\quad \mid * \\
\quad \mid $ \text{integer} \\
\quad \mid \text{simple-expr . lowercase-ident} \\
\quad \mid \text{simple-expr . ( integer )} \\
\quad \mid \text{simple-expr . [ integer ]} \\
\quad \mid ! \text{simple-expr} \\
\quad \mid ( \text{simple-expr} )
\]

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.

### 16.8 Controlling the debugger

#### 16.8.1 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.

#### 16.8.2 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 executes the OCaml runtime `ocamlrun` 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 [16.8.6]).

16.8.3 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 directorynames for modulename
   Same as `directory directorynames`, but the given directories will be searched only when looking for the source file of a module that has been packed into `modulename`.

directory
   Reset the search path. This requires confirmation.

16.8.4 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`.

16.8.5 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.
16.8.6 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 through the `CAML_DEBUG_SOCKET` environment variable.

16.8.7 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).
16.8.8 User-defined printers

Just as in the toplevel system (section 9.2), 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 OCaml 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.

16.9 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`.

16.10 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 17

Profiling (ocamlprof)

This chapter describes how the execution of OCaml programs can be profiled, by recording how many times functions are called, branches of conditionals are taken, ...

17.1 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 8). 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.

Note If a module (.ml file) doesn’t have a corresponding interface (.mli file), then compiling it with ocamlcp will produce object files (.cmi and .cmo) that are not compatible with the ones produced by ocamlc, which may lead to problems (if the .cmi or .cmo is still around) when switching between profiling and non-profiling compilations. To avoid this problem, you should always have a .mli file for each .ml file.

Note To make sure your programs can be compiled in profiling mode, avoid using any identifier that begins with __ocaml_prof.

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 each function body
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 -p film` 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.

**Note:** Due to the implementation of streams and stream patterns as syntactic sugar, it is hard to predict what parts of stream expressions and patterns will be profiled by a given flag. To profile a program with streams, we recommend using `ocamlcp -p a`.

### 17.2 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.

If the environment variable `OCAMLPROF_DUMP` is set when the program exits, its value is used as the file name instead of `ocamlprof.dump`.

The 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.

### 17.3 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 to be read.

- `-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 `(* sn *)`.

- `-impl filename`
  Process the file `filename` as an implementation file, even if its extension is not `.ml`.

- `-intf filename`
  Process the file `filename` as an interface file, even if its extension is not `.mli`. 
-version
  Print version string and exit.

-vnum
  Print short version number and exit.

-help or --help
  Display a short usage summary and exit.

17.4 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:

```
ocamllopt -o myprog -p other-options files
./myprog
gprof myprog
```

Caml function names in the output of gprof have the following format:

\[
\text{Module-name}_\text{function-name}_\text{unique-number}
\]

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 18

Interfacing C with OCaml

This chapter describes how user-defined primitives, written in C, can be linked with Caml code and called from Caml functions.

18.1 Overview and compilation information

18.1.1 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 as a function with 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,

\[
\begin{align*}
\text{type int_endo} & = \text{int} \rightarrow \text{int} \\
\text{external f : int_endo} & \rightarrow \text{int} = "f" \\
\text{external g : (int} & \rightarrow \text{int}) \rightarrow (\text{int} \rightarrow \text{int}) = "f"
\end{align*}
\]

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.

### 18.1.2 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, \ldots), 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:

\[
\text{CAMLprim value input(value channel, value buffer, value offset, value length)} \\
\{ \\
\text{...} \\
\}
\]

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:

\[
\text{CAMLprim value add_nat_native(value nat1, value ofs1, value len1,} \\
\text{value nat2, value ofs2, value len2,} \\
\text{value carry_in)} \\
\{ \\
\text{...} \\
\}
\]

\[
\text{CAMLprim value add_nat_bytecode(value} * \text{argv, int argn)} \\
\{ \\
\text{return add_nat_native(argv[0], argv[1], argv[2], argv[3],} \\
\text{argv[4], argv[5], argv[6]);} \\
\}
\]
Chapter 18. Interfacing C with OCaml

The names of the two C functions must be given in the primitive declaration, as follows:

```plaintext
type =
bytecode-C-function-name native-code-C-function-name
```

For instance, in the case of `add_nat`, the declaration is:

```plaintext
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:

```plaintext
CAMLprim 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 `CAMLprim` macro expands to the required compiler directives to ensure that the function following it is exported and accessible from Caml.) The hard work is performed by the function `getblock`, which is declared as:

```plaintext
long getblock(struct channel * channel, char * p, long n)
{
    ...
}
```

To write C code that operates on OCaml values, the following include files are provided:

<table>
<thead>
<tr>
<th>Include file</th>
<th>Provides</th>
</tr>
</thead>
<tbody>
<tr>
<td>caml/mlvalues.h</td>
<td>definition of the <code>value</code> type, and conversion macros</td>
</tr>
<tr>
<td>caml/alloc.h</td>
<td>allocation functions (to create structured Caml objects)</td>
</tr>
<tr>
<td>caml/memory.h</td>
<td>miscellaneous memory-related functions and macros (for GC interface, in-place modification of structures, etc).</td>
</tr>
<tr>
<td>caml/fail.h</td>
<td>functions for raising exceptions (see section 18.4.5)</td>
</tr>
<tr>
<td>caml/callback.h</td>
<td>callback from C to Caml (see section 18.7)</td>
</tr>
<tr>
<td>caml/custom.h</td>
<td>operations on custom blocks (see section 18.9).</td>
</tr>
<tr>
<td>caml/intext.h</td>
<td>operations for writing user-defined serialization and deserialization functions for custom blocks (see section 18.9).</td>
</tr>
<tr>
<td>caml/threads.h</td>
<td>operations for interfacing in the presence of multiple threads (see section 18.10).</td>
</tr>
</tbody>
</table>
These files reside in the `caml/` subdirectory of the OCaml standard library directory (usually `/usr/local/lib/ocaml`).

### 18.1.3 Statically linking C code with Caml code

The OCaml 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. (Unless dynamic loading of C libraries is supported – see section 18.1.4 below.)

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 and Windows, a library named `libname.a` (respectively, `lib`) residing in one of the standard library directories can also be specified as `cclib -lname`.

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, .a, .so or .dll files) that implement the required primitives.
Starting with Objective Caml 3.00, it is possible to record the `-custom` option as well as the names of C libraries in a Caml library file `.cma` or `.cmxa`. For instance, consider a Caml library `mylib.cma`, built from the Caml object files `a.cmo` and `b.cmo`, which reference C code in `libmylib.a`. If the library is built as follows:

```
ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

users of the library can simply link with `mylib.cma`:

```
ocamlc -o myprog mylib.cma ...
```

and the system will automatically add the `-custom` and `-cclib -lmylib` options, achieving the same effect as

```
ocamlc -o myprog -custom a.cmo b.cmo ... -cclib -lmylib
```

The alternative, of course, is to build the library without extra options:

```
ocamlc -a -o mylib.cma a.cmo b.cmo
```

and then ask users to provide the `-custom` and `-cclib -lmylib` options themselves at link-time:

```
ocamlc -o myprog -custom mylib.cma ... -cclib -lmylib
```

The former alternative is more convenient for the final users of the library, however.

### 18.1.4 Dynamically linking C code with Caml code

Starting with Objective Caml 3.03, an alternative to static linking of C code using the `-custom` code is provided. In this mode, the Caml linker generates a pure bytecode executable (no embedded custom runtime system) that simply records the names of dynamically-loaded libraries containing the C code. The standard Caml runtime system `ocamlrun` then loads dynamically these libraries, and resolves references to the required primitives, before executing the bytecode.

This facility is currently supported and known to work well under Linux, MacOS X, and Windows. It is supported, but not fully tested yet, under FreeBSD, Tru64, Solaris and Irix. It is not supported yet under other Unixes.

To dynamically link C code with Caml code, the C code must first be compiled into a shared library (under Unix) or DLL (under Windows). This involves 1- compiling the C files with appropriate C compiler flags for producing position-independent code (when required by the operating system), and 2- building a shared library from the resulting object files. The resulting shared library or DLL file must be installed in a place where `ocamlrun` can find it later at program start-up time (see section 10.3). Finally (step 3), execute the `ocamlc` command with

- the names of the desired Caml object files (`.cmo` and `.cma` files);

- the names of the C shared libraries (`.so` or `.dll` files) that implement the required primitives.

Under Unix and Windows, a library named `dllname.so` (respectively, `dllname.dll`) residing in one of the standard library directories can also be specified as `-dllib -lname.`
Do not set the \texttt{-custom} flag, otherwise you’re back to static linking as described in section \ref{sec:static-linking}. The \texttt{ocamlmklib} tool (see section \ref{sec:ocamlmklib}) automates steps 2 and 3.

As in the case of static linking, it is possible (and recommended) to record the names of C libraries in a Caml \texttt{.cma} library archive. Consider again a Caml library \texttt{mylib.cma}, built from the Caml object files \texttt{a.cmo} and \texttt{b.cmo}, which reference C code in \texttt{dllmylib.so}. If the library is built as follows:

\begin{verbatim}
ocamlc -a -o mylib.cma a.cmo b.cmo -dllib -lmylib
\end{verbatim}

users of the library can simply link with \texttt{mylib.cma}:

\begin{verbatim}
ocamlc -o myprog mylib.cma ...
\end{verbatim}

and the system will automatically add the \texttt{-dllib -lmylib} option, achieving the same effect as

\begin{verbatim}
ocamlc -o myprog a.cmo b.cmo ... -dllib -lmylib
\end{verbatim}

Using this mechanism, users of the library \texttt{mylib.cma} do not need to known that it references C code, nor whether this C code must be statically linked (using \texttt{-custom}) or dynamically linked.

\section{Choosing between static linking and dynamic linking}

After having described two different ways of linking C code with Caml code, we now review the pros and cons of each, to help developers of mixed Caml/C libraries decide.

The main advantage of dynamic linking is that it preserves the platform-independence of bytecode executables. That is, the bytecode executable contains no machine code, and can therefore be compiled on platform $A$ and executed on other platforms $B$, $C$, ..., as long as the required shared libraries are available on all these platforms. In contrast, executables generated by \texttt{ocamlc -custom} run only on the platform on which they were created, because they embark a custom-tailored runtime system specific to that platform. In addition, dynamic linking results in smaller executables.

Another advantage of dynamic linking is that the final users of the library do not need to have a C compiler, C linker, and C runtime libraries installed on their machines. This is no big deal under Unix and Cygwin, but many Windows users are reluctant to install Microsoft Visual C just to be able to do \texttt{ocamlc -custom}.

There are two drawbacks to dynamic linking. The first is that the resulting executable is not stand-alone: it requires the shared libraries, as well as \texttt{ocamlrun}, to be installed on the machine executing the code. If you wish to distribute a stand-alone executable, it is better to link it statically, using \texttt{ocamlc -custom -ccopt -static} or \texttt{ocamlopt -ccopt -static}. Dynamic linking also raises the “DLL hell” problem: some care must be taken to ensure that the right versions of the shared libraries are found at start-up time.

The second drawback of dynamic linking is that it complicates the construction of the library. The C compiler and linker flags to compile to position-independent code and build a shared library vary wildly between different Unix systems. Also, dynamic linking is not supported on all Unix systems, requiring a fall-back case to static linking in the Makefile for the library. The \texttt{ocamlmklib} command (see section \ref{sec:ocamlmklib}) tries to hide some of these system dependencies.

In conclusion: dynamic linking is highly recommended under the native Windows port, because there are no portability problems and it is much more convenient for the end users. Under Unix,
dynamic linking should be considered for mature, frequently used libraries because it enhances platform-independence of bytecode executables. For new or rarely-used libraries, static linking is much simpler to set up in a portable way.

### 18.1.6 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
```

To generate a bytecode executable that runs on this runtime system, do:

```
ocamlc -use-runtime /home/me/ocamlunixrun -o myprog unix.cma threads.cma
```

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 are required) and also when building the bytecode executable (so that the bytecode from `unix.cma` and `threads.cma` is actually linked in).

### 18.2 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 `caml_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).

#### 18.2.1 Integer values

Integer values encode 31-bit signed integers (63-bit on 64-bit architectures). They are unboxed (unallocated).
18.2.2 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:

<table>
<thead>
<tr>
<th>Tag</th>
<th>Contents of the block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to No_scan_tag − 1</td>
<td>A structured block (an array of Caml objects). Each field is a value.</td>
</tr>
<tr>
<td>Closure_tag</td>
<td>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.</td>
</tr>
<tr>
<td>String_tag</td>
<td>A character string.</td>
</tr>
<tr>
<td>Double_tag</td>
<td>A double-precision floating-point number.</td>
</tr>
<tr>
<td>Double_array_tag</td>
<td>An array or record of double-precision floating-point numbers.</td>
</tr>
<tr>
<td>Abstract_tag</td>
<td>A block representing an abstract datatype.</td>
</tr>
<tr>
<td>Custom_tag</td>
<td>A block representing an abstract datatype with user-defined finalization, comparison, hashing, serialization and deserialization functions attached.</td>
</tr>
</tbody>
</table>

18.2.3 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.

Caution: if a pointer returned by malloc is cast to the type value and returned to Caml, explicit deallocation of the pointer using free is potentially dangerous, because the pointer may still be accessible from the Caml world. Worse, the memory space deallocated by free can later be reallocated as part of the Caml heap; the pointer, formerly pointing outside the Caml heap, now points inside the Caml heap, and this can confuse the garbage collector. To avoid these problems, it is preferable to wrap the pointer in a Caml block with tag Abstract_tag or Custom_tag.

18.3 Representation of Caml data types

This section describes how Caml data types are encoded in the value type.
18.3.1 Atomic types

<table>
<thead>
<tr>
<th>Caml type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Unboxed integer values.</td>
</tr>
<tr>
<td>char</td>
<td>Unboxed integer values (ASCII code).</td>
</tr>
<tr>
<td>float</td>
<td>Blocks with tag Double_tag.</td>
</tr>
<tr>
<td>string</td>
<td>Blocks with tag String_tag.</td>
</tr>
<tr>
<td>int32</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
<tr>
<td>int64</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
<tr>
<td>nativeint</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
</tbody>
</table>

18.3.2 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.)

18.3.3 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 caml_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.

18.3.4 Concrete data 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. A non-constant constructors declared with $n$ arguments is represented by a block of size $n$, tagged with the constructor number; the $n$ fields contain its arguments. Example:

<table>
<thead>
<tr>
<th>Constructed term</th>
<th>Representation</th>
</tr>
</thead>
<tbody>
<tr>
<td>()</td>
<td>Val_int(0)</td>
</tr>
<tr>
<td>false</td>
<td>Val_int(0)</td>
</tr>
<tr>
<td>true</td>
<td>Val_int(1)</td>
</tr>
<tr>
<td>[]</td>
<td>Val_int(0).</td>
</tr>
<tr>
<td>h::t</td>
<td>Block with size = 2 and tag = 0; first field contains h, second field t.</td>
</tr>
</tbody>
</table>
As a convenience, `caml/mlvalues.h` defines the macros `Val_unit`, `Val_false` and `Val_true` to refer to `()`, `false` and `true`.

The following artificial example illustrates the assignment of integers and block tags to constructors:

```haskell
type t =
  | A (* First constant constructor -> integer "Val_int(0)" *)
  | B of string (* First non-constant constructor -> block with tag 0 *)
  | C (* Second constant constructor -> integer "Val_int(1)" *)
  | D of bool (* Second non-constant constructor -> block with tag 1 *)
  | E of t * t (* Third non-constant constructor -> block with tag 2 *)
```

### 18.3.5 Objects

Objects are represented as blocks with tag `Object_tag`. 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 second field contains a unique object ID, used for comparisons. The remaining fields of the object contain the values of the instance variables of the object. It is unsafe to access directly instance variables, as the type system provides no guaranteee about the instance variables contained by an object.

One may extract a public method from an object using the C function `caml_get_public_method` (declared in `<caml/mlvalues.h>`). Since public method tags are hashed in the same way as variant tags, and methods are functions taking self as first argument, if you want to do the method call `foo#bar` from the C side, you should call:

```c
callback(caml_get_public_method(foo, hash_variant("bar")), foo);
```

### 18.3.6 Polymorphic variants

Like constructed terms, polymorphic variant values are represented either as integers (for polymorphic variants without arguments), or as blocks (for polymorphic variants with an argument). Unlike constructed terms, variant constructors are not numbered starting from 0, but identified by a hash value (a Caml integer), as computed by the C function `hash_variant` (declared in `<caml/mlvalues.h>`): the hash value for a variant constructor named, say, `VConstr` is `hash_variant("VConstr")`.

The variant value `VConstr` is represented by `hash_variant("VConstr")`. The variant value `VConstr(v)` is represented by a block of size 2 and tag 0, with field number 0 containing `hash_variant("VConstr")` and field number 1 containing `v`.

Unlike constructed values, polymorphic variant values taking several arguments are not flattened. That is, `VConstr(v, v')` is represented by a block of size 3 containing the representation of the pair `(v, v')`, rather than a block of size 3 containing `v` and `v'` in fields 1 and 2.
18.4 Operations on values

18.4.1 Kind tests
- \texttt{Is\_long(v)} is true if value \( v \) is an immediate integer, false otherwise.
- \texttt{Is\_block(v)} is true if value \( v \) is a pointer to a block, and false if it is an immediate integer.

18.4.2 Operations on integers
- \texttt{Val\_long(l)} returns the value encoding the \texttt{long int} \( l \).
- \texttt{Long\_val(v)} returns the \texttt{long int} encoded in value \( v \).
- \texttt{Val\_int(i)} returns the value encoding the \texttt{int} \( i \).
- \texttt{Int\_val(v)} returns the \texttt{int} encoded in value \( v \).
- \texttt{Val\_bool(x)} returns the Caml boolean representing the truth value of the C integer \( x \).
- \texttt{Bool\_val(v)} returns 0 if \( v \) is the Caml boolean \texttt{false}, 1 if \( v \) is \texttt{true}.
- \texttt{Val\_true, Val\_false} represent the Caml booleans \texttt{true} and \texttt{false}.

18.4.3 Accessing blocks
- \texttt{Wosize\_val(v)} returns the size of the block \( v \), in words, excluding the header.
- \texttt{Tag\_val(v)} returns the tag of the block \( v \).
- \texttt{Field(v, n)} returns the value contained in the \( n \text{th} \) field of the structured block \( v \). Fields are numbered from 0 to \texttt{Wosize\_val(v)} − 1.
- \texttt{Store\_field(b, n, v)} stores the value \( v \) in the field number \( n \) of value \( b \), which must be a structured block.
- \texttt{Code\_val(v)} returns the code part of the closure \( v \).
- \texttt{caml\_string\_length(v)} returns the length (number of characters) of the string \( v \).
- \texttt{Byte(v, n)} returns the \( n \text{th} \) character of the string \( v \), with type \texttt{char}. Characters are numbered from 0 to \texttt{string\_length(v)} − 1.
- \texttt{Byte\_u(v, n)} returns the \( n \text{th} \) character of the string \( v \), with type \texttt{unsigned char}. Characters are numbered from 0 to \texttt{string\_length(v)} − 1.
- \texttt{String\_val(v)} returns a pointer to the first byte of the string \( v \), with type \texttt{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.
- \texttt{Double\_val(v)} returns the floating-point number contained in value \( v \), with type \texttt{double}.
• Double_field(v, n) returns the \(n\)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 \(n\)th element of the array of floating-point numbers \(v\).

• Data_custom_val(v) returns a pointer to the data part of the custom block \(v\). This pointer has type \texttt{void *} and must be cast to the type of the data contained in the custom block.

• Int32_val(v) returns the 32-bit integer contained in the \texttt{int32} \(v\).

• Int64_val(v) returns the 64-bit integer contained in the \texttt{int64} \(v\).

• Nativeint_val(v) returns the long integer contained in the \texttt{nativeint} \(v\).

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).

18.4.4 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.

• caml_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.

• caml_alloc_tuple\((n)\) returns a fresh block of size \(n\) words, with tag 0.

• caml_alloc_string\((n)\) returns a string value of length \(n\) characters. The string initially contains garbage.

• caml_copy_string\((s)\) returns a string value containing a copy of the null-terminated C string \(s\) (a char *).

• caml_copy_double\((d)\) returns a floating-point value initialized with the double \(d\).

• caml_copy_int32\((i)\), copy_int64\((i)\) and caml_copy_nativeint\((i)\) return a value of Caml type int32, int64 and nativeint, respectively, initialized with the integer \(i\).

• caml_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.)

• caml_copy_string_array\((p)\) allocates an array of strings, copied from the pointer to a string array \(p\) (a char **). \(p\) must be NULL-terminated.
Low-level interface

The following functions are slightly more efficient than `caml_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.

- `caml_alloc_small(n, t)` returns a fresh small block of size \( n \leq \text{Max}_{\text{young wosize}} \) words, with tag \( t \). If this block is a structured block (i.e. if \( t < \text{No}_{\text{scan tag}} \)), 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.

- `caml_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 `caml_alloc_small` instead of `caml_alloc_shr`. If this block is a structured block (i.e. if \( t < \text{No}_{\text{scan tag}} \)), then the fields of the block (initially containing garbage) must be initialized with legal values (using the `caml_initialize` function described below) before the next allocation.

18.4.5 Raising exceptions

Two functions are provided to raise two standard exceptions:

- `caml_failwith(s)`, where \( s \) is a null-terminated C string (with type `char *`), raises exception `Failure` with argument \( s \).

- `caml_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 18.7.3. Once the exception identifier is recovered in C, the following functions actually raise the exception:

- `caml_raise_constant(id)` raises the exception \( id \) with no argument;

- `caml_raise_with_arg(id, v)` raises the exception \( id \) with the Caml value \( v \) as argument;

- `caml_raise_with_args(id, n, v)` raises the exception \( id \) with the Caml values \( v[0], \ldots, v[n-1] \) as arguments;

- `caml_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.
18.5 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.

18.5.1 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, CAMLreturn0, or CAMLreturnT.

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 (CAMLxparam1 to CAMLxparam5).

The macros CAMLreturn, CAMLreturn0, and CAMLreturnT are used to replace the C keyword return. Every occurrence of return x must be replaced by CAMLreturn (x) if x has type value, or CAMLreturnT (t, x) (where t is the type of x); every occurrence of return without argument must be replaced by CAMLreturn0. If your C function is a procedure (i.e. if it returns void), you must insert CAMLreturn0 at the end (to replace C’s implicit return).

Note: some C compilers give bogus warnings about unused variables caml__dummy_xxx at each use of CAMLparam and CAMLlocal. You should ignore them.

Example:

```c
void foo (value v1, value v2, value v3)
{
    CAMLparam3 (v1, v2, v3);
    ...
    CAMLreturn0;
}
```

Note: if your function is a primitive with more than 5 arguments for use with the byte-code runtime, its arguments are not values 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 values are declared with CAMLlocalN. These macros must be used at the beginning of the function, not in a nested block.

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.

Example:
value bar (value v1, value v2, value v3)
{
    CAMLparam3 (v1, v2, v3);
    CAMLlocal1 (result);
    result = caml_alloc (3, 0);
    ...
    CAMLreturn (result);
}

Rule 3 Assignments to the fields of structured blocks must be done with the Store_field macro (for normal blocks) or Store_double_field macro (for arrays and records of floating-point numbers). Other assignments must not use Store_field nor Store_double_field.

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 = caml_alloc (3, 0);
    Store_field (result, 0, v1);
    Store_field (result, 1, v2);
    Store_field (result, 2, v3);
    CAMLreturn (result);
}

Warning: The first argument of Store_field and Store_double_field must be a variable declared by CAMLparam* or a parameter declared by CAMLlocal* to ensure that a garbage collection triggered by the evaluation of the other arguments will not invalidate the first argument after it is computed.

Rule 4 Global variables containing values must be registered with the garbage collector using the caml_register_global_root function.

Registration of a global variable v is achieved by calling caml_register_global_root(&v) just before or just after a valid value is stored in v for the first time. You must not call any of the OCaml runtime functions or macros between registering and storing the value.

A registered global variable v can be un-registered by calling caml_remove_global_root(&v).

If the contents of the global variable v are not modified after registration, better performance can be achieved by calling caml_register_generational_global_root(&v) to register v, and caml_remove_generational_global_root(&v) to un-register it. The garbage collector takes advantage of the guarantee that v is not modified to scan it less often. This improves performance if many global variables need to be registered.
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.

18.5.2 Low-level interface

We now give the GC rules corresponding to the low-level allocation functions `caml_alloc_small` and `caml_alloc_shr`. You can ignore those rules if you stick to the simplified allocation function `caml_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 `caml_alloc_small`, filling is performed by direct assignment to the fields of the block:

\[
\text{Field}(v, n) = v_n;
\]

If the block has been allocated with `caml_alloc_shr`, filling is performed through the `caml_initialize` function:

\[
\text{caml_initialize}(&\text{Field}(v, n), v_n);
\]

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_unit`), 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

\[
\text{Field}(v, n) = w;
\]

is safe only if \(v\) is a block newly allocated by `caml_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 `caml_alloc_shr`, use `caml_initialize` to assign a value to a field for the first time:

\[
\text{caml_initialize}(&\text{Field}(v, n), w);
\]

Otherwise, you are updating a field that previously contained a well-formed value; then, call the `caml_modify` function:

\[
\text{caml_modify}(&\text{Field}(v, n), w);
\]

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 = caml_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 = caml_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 caml_alloc_small, and filled by direct assignments on their fields.

value alloc_list_int(int i1, int i2)
{
    CAMLparam0 ();
    CAMLLocal2 (result, r);

    r = caml_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 = caml_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 caml_modify.

value alloc_list_int(int i1, int i2)
{
    CAMLparam0 ();
    CAMLLocal2 (tail, r);

    r = caml_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 = caml_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 [] */
caml_modify(&Field(r, 1), tail); /* cdr of the result = tail */
CAMLreturn (r);
}

It would be incorrect to perform Field(r, 1) = tail directly, because the allocation of tail has taken place since r was allocated.

### 18.6 A complete example

This section outlines how the functions from the Unix curses library can be made available to OCaml programs. First of all, here is the interface curses.mli that declares the curses primitives and data types:

```ocaml
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 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_stubs.c, looks like this:

```c
#include <curses.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>
#include <caml/custom.h>

/* Encapsulation of opaque window handles (of type WINDOW *)
   as Caml custom blocks. */

static struct custom_operations curses_window_ops = {
    "fr.inria.caml.curses_windows",
    custom_finalize_default,
    custom_compare_default,
    custom_hash_default,
```

```c
```
custom_serialize_default,
custom_deserialize_default
};

/* Accessing the WINDOW * part of a Caml custom block */
#define Window_val(v) (*((WINDOW **) Data_custom_val(v)))

/* Allocating a Caml custom block to hold the given WINDOW * */
static value alloc_window(WINDOW * w)
{
    value v = alloc_custom(&curses_window_ops, sizeof(WINDOW *), 0, 1);
    Window_val(v) = w;
    return v;
}

value caml_curses_initscr(value unit)
{
    CAMLparam1 (unit);
    CAMLreturn (alloc_window(initscr()));
}

value caml_curses_endwin(value unit)
{
    CAMLparam1 (unit);
    endwin();
    CAMLreturn (Val_unit);
}

value caml_curses_refresh(value unit)
{
    CAMLparam1 (unit);
    refresh();
    CAMLreturn (Val_unit);
}

value caml_curses_wrefresh(value win)
{
    CAMLparam1 (win);
    wrefresh(Window_val(win));
    CAMLreturn (Val_unit);
}

value caml_curses_newwin(value nlines, value ncols, value x0, value y0)
{
    CAMLparam4 (nlines, ncols, x0, y0);
CAMLreturn (alloc_window(newwin(Int_val(nlines), Int_val(ncols),
                          Int_val(x0), Int_val(y0))));
}

value caml_curses_addch(value c)
{
  CAMLparam1 (c);
  addch(Int_val(c));     /* Characters are encoded like integers */
  CAMLreturn (Val_unit);
}

value caml_curses_mvwaddch(value win, value x, value y, value c)
{
  CAMLparam4 (win, x, y, c);
  mvwaddch(Window_val(win), Int_val(x), Int_val(y), Int_val(c));
  CAMLreturn (Val_unit);
}

value caml_curses_addstr(value s)
{
  CAMLparam1 (s);
  addstr(String_val(s));
  CAMLreturn (Val_unit);
}

value caml_curses_mvwaddstr(value win, value x, value y, value s)
{
  CAMLparam4 (win, x, y, s);
  mvwaddstr(Window_val(win), Int_val(x), Int_val(y), String_val(s));
  CAMLreturn (Val_unit);
}

/* This goes on for pages. */

The file curses_stubs.c can be compiled with:

    cc -c -I`ocamlc -where` 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();
Unix.sleep 5;
endwin()

To compile and link this program, run:

    ocamlc -custom -o test unix.cma test.ml curses_stubs.o -cclib -lcurses

(On some machines, you may need to put -cclib -ltermcap or -cclib -lcurses -cclib
-ltermcap instead of -cclib -lcurses.)

18.7 Advanced topic: 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.

18.7.1 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:

- **caml_callback**(f, a) applies the functional value f to the value a and return the value re-
turned by f.

- **caml_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.

- **caml_callback3**(f, a, b, c) applies the functional value f (a curried Caml function with three
arguments) to a, b and c.

- **caml_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
caml_callback_exn, caml_callback2_exn, caml_callback3_exn, caml_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 caml_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).
18.7.2 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, OCaml 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:

```ocaml
let f x = print_string "f is applied to "; print_int x; 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:

```c
void call_caml_f(int arg)
{
    caml_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:

```c
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");
    }
    caml_callback(*closure_f, Val_int(arg));
}
```

18.7.3 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:

```ocaml
exception Error of string
let _ = Callback.register_exception "test exception" (Error "any string")
```
The C code can then recover the exception identifier using \texttt{caml\_named\_value} and pass it as first argument to the functions \texttt{raise\_constant}, \texttt{raise\_with\_arg}, and \texttt{raise\_with\_string} (described in section 18.4.5) to actually raise the exception. For example, here is a C function that raises the \texttt{Error} exception with the given argument:

\begin{verbatim}
void raise_error(char * msg)
{
    caml_raise_with_string(*caml\_named\_value("test exception"), msg);
}
\end{verbatim}

18.7.4 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 \texttt{main} function, which will override the default \texttt{main} function provided by the Caml runtime system. Execution will start in the user-defined \texttt{main} function just like for a regular C program.

- At some point, the C code must call \texttt{caml\_main(argv)} to initialize the Caml code. The \texttt{argv} argument is a C array of strings (type \texttt{char **}), terminated with a NULL pointer, which represents the command-line arguments, as passed as second argument to \texttt{main}. The Caml array \texttt{Sys.argv} will be initialized from this parameter. For the bytecode compiler, \texttt{argv[0]} and \texttt{argv[1]} are also consulted to find the file containing the bytecode.

- The call to \texttt{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 \texttt{Callback\_register}. Once the Caml initialization code is complete, control returns to the C code that called \texttt{caml\_main}.

- The C code can then invoke Caml functions using the callback mechanism (see section 18.7.1).

18.7.5 Embedding the Caml code in the C code

The bytecode compiler in custom runtime mode (\texttt{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 \texttt{ocamlc}. Second, the Caml runtime library must be able to find the name of the executable file from the command-line arguments. When using \texttt{caml\_main(argv)} as in section 18.7.4, this means that \texttt{argv[0]} or \texttt{argv[1]} must contain the executable file name.

An alternative is to embed the bytecode in the C code. The \texttt{-output-obj} option to \texttt{ocamlc} is provided for this purpose. It causes the \texttt{ocamlc} compiler to output a C object file (.o file, .obj under Windows) containing the bytecode for the Caml part of the program, as well as a \texttt{caml\_startup} function. The C object file produced by \texttt{ocamlc -output-obj} can then be linked with C code using the standard C compiler, or stored in a C library.

The \texttt{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 \texttt{caml\_main}, it takes one \texttt{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 `-output-obj` option can also be used to obtain the C source file. More interestingly, the same option can also produce directly a shared library (.so file, .dll under Windows) that contains the Caml code, the Caml runtime system and any other static C code given to `ocamlc` (.o, .a, respectively, .obj, .lib). This use of `-output-obj` is very similar to a normal linking step, but instead of producing a main program that automatically runs the Caml code, it produces a shared library that can run the Caml code on demand. The three possible behaviors of `-output-obj` are selected according to the extension of the resulting file (given with `-o`).

The native-code compiler `ocamlopt` also supports the `-output-obj` option, causing it to output a C object file or a shared library 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 OCaml 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:

```bash
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:

```bash
ocamlopt -output-obj -o camlcode.o unix.cmxa other.cmxa and .cmxa files
cc -o myprog C objects and libraries \ 
camlcode.o -L/usr/local/lib/ocaml -lunix -lasmrun
```

The shared libraries produced by `ocamlc -output-obj` or by `ocamlopt -output-obj` already contains the OCaml runtime library as well as all the needed C libraries.

**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 OCaml, 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:

- Alpha under Digital Unix / Tru64 Unix with `gcc`: object files produced by `ocamlc -output-obj` must be linked with the `gcc` options `-Wl,-T,12000000 -Wl,-D,14000000`. This is not necessary for object files produced by `ocamlopt -output-obj`.

- Windows NT: the object file produced by OCaml have been compiled with the `/MD` flag, and therefore all other object files linked with it should also be compiled with `/MD`. 
18.8 Advanced example with callbacks

This section illustrates the callback facilities described in section [18.7]. 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(caml_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(caml_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 functions 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 -lcurses

(On some machines, you may need to put -ltermcap or -lcurses -ltermcap instead of -lcurses.)

18.9 Advanced topic: custom blocks

Blocks with tag Custom_tag contain both arbitrary user data and a pointer to a C struct, with type struct custom_operations, that associates user-provided finalization, comparison, hashing, serialization and deserialization functions to this block.

18.9.1 The struct custom_operations

The struct custom_operations is defined in <caml/custom.h> and contains the following fields:

- char *identifier
  A zero-terminated character string serving as an identifier for serialization and deserialization operations.
• `void (*finalize)(value v)`
  The `finalize` field contains a pointer to a C function that is called when the block becomes unreachable and is about to be reclaimed. The block is passed as first argument to the function. The `finalize` field can also be `custom_finalize_default` to indicate that no finalization function is associated with the block.

• `int (*compare)(value v1, value v2)`
  The `compare` field contains a pointer to a C function that is called whenever two custom blocks are compared using Caml’s generic comparison operators (`=, <>, <=, =>, <, >` and `compare`). The C function should return 0 if the data contained in the two blocks are structurally equal, a negative integer if the data from the first block is less than the data from the second block, and a positive integer if the data from the first block is greater than the data from the second block.

  The `compare` field can be set to `custom_compare_default`; this default comparison function simply raises `Failure`.

• `long (*hash)(value v)`
  The `hash` field contains a pointer to a C function that is called whenever Caml’s generic hash operator (see module `Hashtbl`) is applied to a custom block. The C function can return an arbitrary long integer representing the hash value of the data contained in the given custom block. The hash value must be compatible with the `compare` function, in the sense that two structurally equal data (that is, two custom blocks for which `compare` returns 0) must have the same hash value.

  The `hash` field can be set to `custom_hash_default`, in which case the custom block is ignored during hash computation.

• `void (*serialize)(value v, unsigned long * wsize_32, unsigned long * wsize_64)`
  The `serialize` field contains a pointer to a C function that is called whenever the custom block needs to be serialized (marshaled) using the Caml functions `output_value` or `Marshal.to_...`. For a custom block, those functions first write the identifier of the block (as given by the `identifier` field) to the output stream, then call the user-provided `serialize` function. That function is responsible for writing the data contained in the custom block, using the `serialize_...` functions defined in `<caml/intext.h>` and listed below. The user-provided `serialize` function must then store in its `wsize_32` and `wsize_64` parameters the sizes in bytes of the data part of the custom block on a 32-bit architecture and on a 64-bit architecture, respectively.

  The `serialize` field can be set to `custom_serialize_default`, in which case the `Failure` exception is raised when attempting to serialize the custom block.

• `unsigned long (*deserialize)(void * dst)`
  The `deserialize` field contains a pointer to a C function that is called whenever a custom block with identifier `identifier` needs to be deserialized (un-marshaled) using the Caml functions `input_value` or `Marshal.from_...`. This user-provided function is responsible for reading back the data written by the `serialize` operation, using the `deserialize_...` functions defined in `<caml/intext.h>` and listed below. It must then rebuild the data part of the custom block and store it at the pointer given as the `dst` argument. Finally, it returns
the size in bytes of the data part of the custom block. This size must be identical to the
\texttt{wsize\_32} result of the \texttt{serialize} operation if the architecture is 32 bits, or \texttt{wsize\_64} if the
architecture is 64 bits.

The \texttt{deserialize} field can be set to \texttt{custom\_deserialize\_default} to indicate that deseri-
alization is not supported. In this case, do not register the \texttt{struct custom\_operations} with
the deserializer using \texttt{register\_custom\_operations} (see below).

Note: the \texttt{finalize}, \texttt{compare}, \texttt{hash}, \texttt{serialize} and \texttt{deserialize} functions attached to custom
block descriptors must never trigger a garbage collection. Within these functions, do not call any
of the Caml allocation functions, and do not perform a callback into Caml code. Do not use
\texttt{CAMLparam} to register the parameters to these functions, and do not use \texttt{CAMLreturn} to return the
result.

18.9.2 Allocating custom blocks

Custom blocks must be allocated via the \texttt{caml\_alloc\_custom} function. \texttt{caml\_alloc\_custom(ops, size, used, max)} returns a fresh custom block, with room for \texttt{size} bytes of user data, and whose as-
sociated operations are given by \texttt{ops} (a pointer to a \texttt{struct custom\_operations}, usually statically
allocated as a C global variable).

The two parameters \texttt{used} and \texttt{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 incre-
mental 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. How-
ever, 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 \texttt{used} and \texttt{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 \texttt{used}, and the maximum amount that you want to see in floating garbage as
parameter \texttt{max}. The units are arbitrary: the GC cares only about the ratio \texttt{used}/\texttt{max}.

For instance, if you are allocating a finalized block holding an X Windows bitmap of \texttt{w} by
\texttt{h} pixels, and you’d rather not have more than 1 mega-pixels of unreclaimed bitmaps, specify
\texttt{used} = \texttt{w} * \texttt{h} and \texttt{max} = 1000000.

Another way to describe the effect of the \texttt{used} and \texttt{max} parameters is in terms of full GC
cycles. If you allocate many custom blocks with \texttt{used}/\texttt{max} = 1/\texttt{N}, the GC will then do one
full cycle (examining every object in the heap and calling finalization functions on those that are
unreachable) every \texttt{N} allocations. For instance, if \texttt{used} = 1 and \texttt{max} = 1000, the GC will do one
full cycle at least every 1000 allocations of custom blocks.

If your finalized blocks contain no pointers to out-of-heap resources, or if the previous discussion
made little sense to you, just take \texttt{used} = 0 and \texttt{max} = 1. But if you later find that the finalization
functions are not called “often enough”, consider increasing the \texttt{used}/\texttt{max} ratio.
18.9.3 Accessing custom blocks

The data part of a custom block \( v \) can be accessed via the pointer `Data_custom_val(v)`. This pointer has type `void *` and should be cast to the actual type of the data stored in the custom block.

The contents of custom blocks are not scanned by the garbage collector, and must therefore not contain any pointer inside the Caml heap. In other terms, never store a Caml `value` in a custom block, and do not use `Field`, `Store_field` nor `caml_modify` to access the data part of a custom block. Conversely, any C data structure (not containing heap pointers) can be stored in a custom block.

18.9.4 Writing custom serialization and deserialization functions

The following functions, defined in `<caml/intext.h>`, are provided to write and read back the contents of custom blocks in a portable way. Those functions handle endianness conversions when e.g. data is written on a little-endian machine and read back on a big-endian machine.

<table>
<thead>
<tr>
<th>Function</th>
<th>Action</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>caml_serialize_int_1</code></td>
<td>Write a 1-byte integer</td>
</tr>
<tr>
<td><code>caml_serialize_int_2</code></td>
<td>Write a 2-byte integer</td>
</tr>
<tr>
<td><code>caml_serialize_int_4</code></td>
<td>Write a 4-byte integer</td>
</tr>
<tr>
<td><code>caml_serialize_int_8</code></td>
<td>Write a 8-byte integer</td>
</tr>
<tr>
<td><code>caml_serialize_float_4</code></td>
<td>Write a 4-byte float</td>
</tr>
<tr>
<td><code>caml_serialize_float_8</code></td>
<td>Write a 8-byte float</td>
</tr>
<tr>
<td><code>caml_serialize_block_1</code></td>
<td>Write an array of 1-byte quantities</td>
</tr>
<tr>
<td><code>caml_serialize_block_2</code></td>
<td>Write an array of 2-byte quantities</td>
</tr>
<tr>
<td><code>caml_serialize_block_4</code></td>
<td>Write an array of 4-byte quantities</td>
</tr>
<tr>
<td><code>caml_serialize_block_8</code></td>
<td>Write an array of 8-byte quantities</td>
</tr>
<tr>
<td><code>caml_deserialize_uint_1</code></td>
<td>Read an unsigned 1-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_sint_1</code></td>
<td>Read a signed 1-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_uint_2</code></td>
<td>Read an unsigned 2-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_sint_2</code></td>
<td>Read a signed 2-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_uint_4</code></td>
<td>Read an unsigned 4-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_sint_4</code></td>
<td>Read a signed 4-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_uint_8</code></td>
<td>Read an unsigned 8-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_sint_8</code></td>
<td>Read a signed 8-byte integer</td>
</tr>
<tr>
<td><code>caml_deserialize_float_4</code></td>
<td>Read a 4-byte float</td>
</tr>
<tr>
<td><code>caml_deserialize_float_8</code></td>
<td>Read an 8-byte float</td>
</tr>
<tr>
<td><code>caml_deserialize_block_1</code></td>
<td>Read an array of 1-byte quantities</td>
</tr>
<tr>
<td><code>caml_deserialize_block_2</code></td>
<td>Read an array of 2-byte quantities</td>
</tr>
<tr>
<td><code>caml_deserialize_block_4</code></td>
<td>Read an array of 4-byte quantities</td>
</tr>
<tr>
<td><code>caml_deserialize_block_8</code></td>
<td>Read an array of 8-byte quantities</td>
</tr>
<tr>
<td><code>caml_deserialize_error</code></td>
<td>Signal an error during deserialization; <code>input_value</code> or <code>Marshal.from...</code> raise a <code>Failure</code> exception after cleaning up their internal data structures</td>
</tr>
</tbody>
</table>
Serialization functions are attached to the custom blocks to which they apply. Obviously, deserialization functions cannot be attached this way, since the custom block does not exist yet when deserialization begins! Thus, the `struct custom_operations` that contain deserialization functions must be registered with the deserializer in advance, using the `register_custom_operations` function declared in `<caml/custom.h>`. Deserialization proceeds by reading the identifier off the input stream, allocating a custom block of the size specified in the input stream, searching the registered `struct custom_operation` blocks for one with the same identifier, and calling its `deserialize` function to fill the data part of the custom block.

18.9.5 Choosing identifiers

Identifiers in `struct custom_operations` must be chosen carefully, since they must identify uniquely the data structure for serialization and deserialization operations. In particular, consider including a version number in the identifier; this way, the format of the data can be changed later, yet backward-compatible deserialization functions can be provided.

Identifiers starting with `_` (an underscore character) are reserved for the OCaml run-time system; do not use them for your custom data. We recommend to use a URL (http://mymachine.mydomain.com/mylibrary/version-number) or a Java-style package name (com.mydomain.mymachine.mylibrary.version-number) as identifiers, to minimize the risk of identifier collision.

18.9.6 Finalized blocks

Custom blocks generalize the finalized blocks that were present in OCaml prior to version 3.00. For backward compatibility, the format of custom blocks is compatible with that of finalized blocks, and the `alloc_final` function is still available to allocate a custom block with a given finalization function, but default comparison, hashing and serialization functions. `caml_alloc_final(n, f, used, max)` returns a fresh custom block of size $n$ words, with finalization function $f$. The first word is reserved for storing the custom operations; the other $n-1$ words are available for your data. The two parameters `used` and `max` are used to control the speed of garbage collection, as described for `caml_alloc_custom`.

18.10 Advanced topic: multithreading

Using multiple threads (shared-memory concurrency) in a mixed Caml/C application requires special precautions, which are described in this section.

18.10.1 Registering threads created from C

Callbacks from C to Caml are possible only if the calling thread is known to the Caml run-time system. Threads created from Caml (through the `Thread.create` function of the system threads library) are automatically known to the run-time system. If the application creates additional threads from C and wishes to callback into Caml code from these threads, it must first register them with the run-time system. The following functions are declared in the include file `<caml/threads.h>`.
- `caml_c_thread_register()` registers the calling thread with the Caml run-time system. Returns 1 on success, 0 on error. Registering an already-register thread does nothing and returns 0.

- `caml_c_thread_unregister()` must be called before the thread terminates, to unregister it from the Caml run-time system. Returns 1 on success, 0 on error. If the calling thread was not previously registered, does nothing and returns 0.

### 18.10.2 Parallel execution of long-running C code

The Caml run-time system is not reentrant: at any time, at most one thread can be executing Caml code or C code that uses the Caml run-time system. Technically, this is enforced by a “master lock” that any thread must hold while executing such code.

When Caml calls the C code implementing a primitive, the master lock is held, therefore the C code has full access to the facilities of the run-time system. However, no other thread can execute Caml code concurrently with the C code of the primitive.

If a C primitive runs for a long time or performs potentially blocking input-output operations, it can explicitly release the master lock, enabling other Caml threads to run concurrently with its operations. The C code must re-acquire the master lock before returning to Caml. This is achieved with the following functions, declared in the include file `<caml/threads.h>`.

- `caml_release_runtime_system()` The calling thread releases the master lock and other Caml resources, enabling other threads to run Caml code in parallel with the execution of the calling thread.

- `caml_acquire_runtime_system()` The calling thread re-acquires the master lock and other Caml resources. It may block until no other thread uses the Caml run-time system.

After `caml_release_runtime_system()` was called and until `caml_acquire_runtime_system()` is called, the C code must not access any Caml data, nor call any function of the run-time system, nor call back into Caml code. Consequently, arguments provided by Caml to the C primitive must be copied into C data structures before calling `caml_release_runtime_system()`, and results to be returned to Caml must be encoded as Caml values after `caml_acquire_runtime_system()` returns.

Example: the following C primitive invokes `gethostbyname` to find the IP address of a host name. The `gethostbyname` function can block for a long time, so we choose to release the Caml run-time system while it is running.

```c
CAMLprim stub_gethostbyname(value vname)
{
  CAMLparam1 (vname);
  CAMLlocal1 (vres);
  struct hostent * h;

  /* Copy the string argument to a C string, allocated outside the Caml heap. */
  name = stat_alloc(caml_string_length(vname) + 1);
```
Callbacks from C to Caml must be performed while holding the master lock to the Caml run-time system. This is naturally the case if the callback is performed by a C primitive that did not release the run-time system. If the C primitive released the run-time system previously, or the callback is performed from other C code that was not invoked from Caml (e.g. an event loop in a GUI application), the run-time system must be acquired before the callback and released after:

```c
strcpy(name, String_val(vname));
/* Release the Caml run-time system */
caml_release_runtime_system();
/* Resolve the name */
h = gethostbyname(name);
/* Re-acquire the Caml run-time system */
caml_acquire_runtime_system();
/* Encode the relevant fields of h as the Caml value vres */
... /* Omitted */
/* Return to Caml */
CAMLreturn (vres);
```

Note: the `acquire` and `release` functions described above were introduced in OCaml 3.12. Older code uses the following historical names, declared in `<caml/signals.h>`:

- `caml_enter_blocking_section` as an alias for `caml_release_runtime_system`
- `caml_leave_blocking_section` as an alias for `caml_acquire_runtime_system`

Intuition: a “blocking section” is a piece of C code that does not use the Caml run-time system, typically a blocking input/output operation.

### 18.11 Building mixed C/Caml libraries: ocamlmklib

The `ocamlmklib` command facilitates the construction of libraries containing both Caml code and C code, and usable both in static linking and dynamic linking modes. This command is available under Windows since Objective Caml 3.11 and under other operating systems since Objective Caml 3.03.

The `ocamlmklib` command takes three kinds of arguments:

- Caml source files and object files (.cmo, .cmx, .ml) comprising the Caml part of the library;
- C object files (.o, .a, respectively, .obj, .lib) comprising the C part of the library;
Chapter 18. Interfacing C with OCaml

- Support libraries for the C part (-l).

It generates the following outputs:

- A Caml bytecode library .cma incorporating the .cmo and .ml Caml files given as arguments, and automatically referencing the C library generated with the C object files.

- A Caml native-code library .cmxa incorporating the .cmx and .ml Caml files given as arguments, and automatically referencing the C library generated with the C object files.

- If dynamic linking is supported on the target platform, a .so (respectively, .dll) shared library built from the C object files given as arguments, and automatically referencing the support libraries.

- A C static library .a (respectively, .lib) built from the C object files.

In addition, the following options are recognized:

-cclib, -ccopt, -I, -linkall
  These options are passed as is to ocamlc or ocamlopt. See the documentation of these commands.

-rpath, -R, -Wl,-rpath, -Wl,-R
  These options are passed as is to the C compiler. Refer to the documentation of the C compiler.

-custom
  Force the construction of a statically linked library only, even if dynamic linking is supported.

-failsafe
  Fall back to building a statically linked library if a problem occurs while building the shared library (e.g. some of the support libraries are not available as shared libraries).

-Ldir
  Add dir to the search path for support libraries (-l).

-ocamlc cmd
  Use cmd instead of ocamlc to call the bytecode compiler.

-ocamlopt cmd
  Use cmd instead of ocamlopt to call the native-code compiler.

-o output
  Set the name of the generated Caml library. ocamlmklib will generate output.cma and/or output.cmxa. If not specified, defaults to a.

-oc outputc
  Set the name of the generated C library. ocamlmklib will generate liboutputc.so (if shared libraries are supported) and liboutputc.a. If not specified, defaults to the output name given with -o.
Example  Consider a Caml interface to the standard libz C library for reading and writing compressed files. Assume this library resides in /usr/local/zlib. This interface is composed of a Caml part zip.cmo/zip.cmx and a C part zipstubs.o containing the stub code around the libz entry points. The following command builds the Caml libraries zip.cma and zip.cmxa, as well as the companion C libraries dllzip.so and libzip.a:

```
ocamlmklib -o zip zip.cmo zip.cmx zipstubs.o -lz -L/usr/local/zlib
```

If shared libraries are supported, this performs the following commands:

```
ocamlc -a -o zip.cma zip.cmo -dllib -lz -lzip \  -cclib -lz -cclib -lz -ccopt -L/usr/local/zlib
ocamlopt -a -o zip.cmxa zip.cmx -cclib -lz -lzip \  -cclib -lz -cclib -lz -ccopt -L/usr/local/zlib
gcc -shared -o dllzip.so zipstubs.o -lz -L/usr/local/zlib
ar rc libzip.a zipstubs.o
```

If shared libraries are not supported, the following commands are performed instead:

```
ocamlc -a -custom -o zip.cma zip.cmo -cclib -lz -lzip \  -cclib -lz -ccopt -L/usr/local/zlib
ocamlopt -a -o zip.cmxa zip.cmx -cclib -lz -lzip \  -cclib -lz -ccopt -L/usr/local/zlib
ar rc libzip.a zipstubs.o
```

Instead of building simultaneously the bytecode library, the native-code library and the C libraries, `ocamlmklib` can be called three times to build each separately. Thus,

```
ocamlmklib -o zip zip.cmo -lz -L/usr/local/zlib
```

builds the bytecode library zip.cma, and

```
ocamlmklib -o zip zip.cmx -lz -L/usr/local/zlib
```

builds the native-code library zip.cmxa, and

```
ocamlmklib -o zip zipstubs.o -lz -L/usr/local/zlib
```

builds the C libraries dllzip.so and libzip.a. Notice that the support libraries (-lz) and the corresponding options (-L/usr/local/zlib) must be given on all three invocations of `ocamlmklib`, because they are needed at different times depending on whether shared libraries are supported.
Part IV

The OCaml library
Chapter 19

The core library

This chapter describes the OCaml core library, which is composed of declarations for built-in types and exceptions, plus the module Pervasives that provides basic operations on these built-in types. The Pervasives module is special in two ways:

- It is automatically linked with the user’s object code files by the ocamlc command (chapter 8).
- 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 of the built-in types and the components of module Pervasives are printed one by one in typewriter font, followed by a short comment. All library modules and the components they provide are indexed at the end of this report.

19.1 Built-in types and predefined exceptions

The following built-in types and predefined exceptions are always defined in the compilation environment, but are not part of any module. As a consequence, they can only be referred by their short names.

Built-in 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 = false | true
   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 of type 'a.

type int32
   The type of signed 32-bit integers. See the Int32[20.13] module.

type int64
   The type of signed 64-bit integers. See the Int64[20.14] module.

type nativeint
   The type of signed, platform-native integers (32 bits on 32-bit processors, 64 bits on 64-bit processors). See the Nativeint[20.20] module.

type ('a, 'b, 'c, 'd) format4
   The type of format strings. 'a is the type of the parameters of the format, 'd is the result type for the printf-style function, 'b is the type of the first argument given to \%a and \%t printing functions (see module Printf[20.24]), and 'c is the result type of these functions.

type 'a lazy_t
   This type is used to implement the Lazy[20.15] module. It should not be used directly.
Chapter 19. The core library

Predefined exceptions

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 match keyword in the source code (file name, line number, column number).

exception Assert_failure of (string * int * int)
Exception raised when an assertion fails. The arguments are the location of the assert keyword in the source code (file name, line number, column number).

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. (Not fully implemented by the native-code compiler; see section 11.5)

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 integer division and remainder operations when their second argument is zero.

exception Sys_blocked_io
A special case of Sys_error raised when no I/O is possible on a non-blocking I/O channel.

exception Undefined_recursive_module of (string * int * int)
Exception raised when an ill-founded recursive module definition is evaluated. (See section 7.8) The arguments are the location of the definition in the source code (file name, line number, column number).
19.2 Module Pervasives: The initially opened module.

This module provides the basic operations over the built-in types (numbers, booleans, strings, exceptions, references, lists, arrays, input-output channels, ...).

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.

Exceptions

val raise : exn -> 'a
Raise the given exception value

val invalid_arg : string -> 'a
Raise exception Invalid_argument with the given string.

val failwith : string -> 'a
Raise exception Failure with the given string.

exception Exit
The Exit exception is not raised by any library function. It is provided for use in your programs.

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 Pervasives.(=) [19.2].

val (<): 'a -> 'a -> bool
See Pervasives.(>=) [19.2].

val (>): 'a -> 'a -> bool
See Pervasives.(>=) [19.2].

val (<=): 'a -> 'a -> bool
See Pervasives.(>=) [19.2].

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.

```plaintext
val compare : 'a -> 'a -> int

compare x y returns 0 if x is equal to y, a negative integer if x is less than y, and a positive integer if x is greater than y. The ordering implemented by compare is compatible with the comparison predicates =, < and > defined above, with one difference on the treatment of the float value Pervasives.nan\[19.2\]. Namely, the comparison predicates treat nan as different from any other float value, including itself; while compare treats nan as equal to itself and less than any other float value. This treatment of nan ensures that compare defines a total ordering relation.

compare applied to functional values may raise Invalid_argument. compare applied to cyclic structures may not terminate.

The compare function can be used as the comparison function required by the Set.Make\[20.28\] and Map.Make\[20.18\] functors, as well as the List.sort\[20.17\] and Array.sort\[20.2\] functions.

val min : 'a -> 'a -> 'a

Return the smaller of the two arguments. The result is unspecified if one of the arguments contains the float value nan.

val max : 'a -> 'a -> 'a

Return the greater of the two arguments. The result is unspecified if one of the arguments contains the float value nan.

val (==) : 'a -> 'a -> bool

e1 == e2 tests for physical equality of e1 and e2. On mutable types such as references, arrays, strings, records with mutable fields and objects with mutable instance variables, e1 == e2 is true if and only if physical modification of e1 also affects e2. On non-mutable types, the behavior of (==) is implementation-dependent; however, it is guaranteed that e1 == e2 implies compare e1 e2 = 0.

val (!=) : 'a -> 'a -> bool

Negation of Pervasives.(==)\[19.2\].

**Boolean operations**

val not : bool -> bool

The boolean negation.

val (&&) : bool -> bool -> bool

The boolean conjunction.
The boolean “and”. Evaluation is sequential, left-to-right: in \( e_1 \land e_2 \), \( e_1 \) is evaluated first, and if it returns \text{false}, \( e_2 \) is not evaluated at all.

\[
\begin{align*}
\text{val (\&)} : \text{bool} & \rightarrow \text{bool} \\
\text{val (\&\&)} : \text{bool} & \rightarrow \text{bool} \\
\text{val (\|\|)} : \text{bool} & \rightarrow \text{bool}
\end{align*}
\]

\text{val (\&)} \text{ should be used instead.}

\text{val (\&\&)} \text{ should be used instead.}

\text{val (\|\|)} \text{ should be used instead.}

**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.

\[
\begin{align*}
\text{val (~-)} : \text{int} & \rightarrow \text{int} \\
\text{val (~+)} : \text{int} & \rightarrow \text{int} \\
\text{val succ} : \text{int} & \rightarrow \text{int} \\
\text{val pred} : \text{int} & \rightarrow \text{int} \\
\text{val (+)} : \text{int} & \rightarrow \text{int} \\
\text{val (-)} : \text{int} & \rightarrow \text{int} \\
\text{val (\* )} : \text{int} & \rightarrow \text{int} \\
\text{val (/)} : \text{int} & \rightarrow \text{int}
\end{align*}
\]

\text{Unary negation. You can also write } - e \text{ instead of } ~- e.

\text{Unary addition. You can also write } + e \text{ instead of } \sim+ e.

\text{Since: 3.12.0}

\text{val succ} \text{ x is } x + 1.

\text{val pred} \text{ x is } x - 1.

\text{val succ} \text{ x is } x + 1.

\text{val pred} \text{ x is } x - 1.

\text{val (+)} \text{ Integer addition.}

\text{val (-)} \text{ Integer subtraction.}

\text{val (\* )} \text{ Integer multiplication.}

\text{val (/)} \text{ Integer division. Raise } \text{Division_by_zero} \text{ if the second argument is 0. Integer division rounds the real quotient of its arguments towards zero. More precisely, if } x \geq 0 \text{ and } y > 0, x / y \text{ is the greatest integer less than or equal to the real quotient of } x \text{ by } y. \text{ Moreover, } (- x) / y = x / (- y) = - (x / y).
val (mod) : int -> int -> int
Integer remainder. If y is not zero, the result of x mod y satisfies the following properties: x = (x / y) * y + x mod y and abs(x mod y) <= abs(y) - 1. If y = 0, x mod y raises Division_by_zero. Note that x mod y is negative only if x < 0. Raise Division_by_zero if y is zero.

val abs : int -> int
Return the absolute value of the argument. Note that this may be negative if the argument is min_int.

val max_int : int
The greatest representable integer.

val min_int : int
The smallest representable integer.

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. The result is unspecified if m < 0 or m >= bitsize, where bitsize is 32 on a 32-bit platform and 64 on a 64-bit platform.

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. The result is unspecified if m < 0 or m >= bitsize.

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. The result is unspecified if m < 0 or m >= bitsize.
Floating-point arithmetic

Caml's floating-point numbers follow the IEEE 754 standard, using double precision (64 bits) numbers. Floating-point operations never raise an exception on overflow, underflow, division by zero, etc. Instead, special IEEE numbers are returned as appropriate, such as infinity for 1.0 /. 0.0, neg_infinity for -1.0 /. 0.0, and nan (“not a number”) for 0.0 /. 0.0. These special numbers then propagate through floating-point computations as expected: for instance, 1.0 /. infinity is 0.0, and any arithmetic operation with nan as argument returns nan as result.

val (~-.) : float -> float
  Unary negation. You can also write -. e instead of ~-. e.

val (~+. ) : float -> float
  Unary addition. You can also write +. e instead of ~+. e.

  Since: 3.12.0

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
  Exponential.

val log : float -> float
  Natural logarithm.

val log10 : float -> float
  Base 10 logarithm.

val expm1 : float -> float
expm1 \( x \) computes \( \exp x - 1.0 \), giving numerically-accurate results even if \( x \) is close to 0.0.

Since: 3.12.0

val [\text{log1p}] : float -> float

log1p \( x \) computes \( \log(1.0 + x) \) (natural logarithm), giving numerically-accurate results even if \( x \) is close to 0.0.

Since: 3.12.0

val [\text{cos}] : float -> float

Cosine. Argument is in radians.

val [\text{sin}] : float -> float

Sine. Argument is in radians.

val [\text{tan}] : float -> float

Tangent. Argument is in radians.

val [\text{acos}] : float -> float

Arc cosine. The argument must fall within the range \([-1.0, 1.0]\). Result is in radians and is between 0.0 and \( \pi \).

val [\text{asin}] : float -> float

Arc sine. The argument must fall within the range \([-1.0, 1.0]\). Result is in radians and is between \(-\pi/2\) and \(\pi/2\).

val [\text{atan}] : float -> float

Arc tangent. Result is in radians and is between \(-\pi/2\) and \(\pi/2\).

val [\text{atan2}] : float -> float -> float

atan2 \( y \) \( x \) returns the arc tangent of \( y / x \). The signs of \( x \) and \( y \) are used to determine the quadrant of the result. Result is in radians and is between \(-\pi\) and \(\pi\).

val [\text{cosh}] : float -> float

Hyperbolic cosine. Argument is in radians.

val [\text{sinh}] : float -> float

Hyperbolic sine. Argument is in radians.

val [\text{tanh}] : float -> float

Hyperbolic tangent. Argument is in radians.

val [\text{ceil}] : float -> float
Round above to an integer value. `ceil f` returns the least integer value greater than or equal to `f`. The result is returned as a float.

```ml
val floor : float -> float
```

Round below to an integer value. `floor f` returns the greatest integer value less than or equal to `f`. The result is returned as a float.

```ml
val abs_float : float -> float
```

`abs_float f` returns the absolute value of `f`.

```ml
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.

```ml
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`.

```ml
val ldexp : float -> int -> float
```

`ldexp x n` returns `x *. 2 ** n`.

```ml
val modf : float -> float * float
```

`modf f` returns the pair of the fractional and integral part of `f`.

```ml
val float : int -> float
```

Same as `Pervasives.float_of_int`.

```ml
val float_of_int : int -> float
```

Convert an integer to floating-point.

```ml
val truncate : float -> int
```

Same as `Pervasives.int_of_float`.

```ml
val int_of_float : float -> int
```

Truncate the given floating-point number to an integer. The result is unspecified if the argument is `nan` or falls outside the range of representable integers.

```ml
val infinity : float
```

Positive infinity.

```ml
val neg_infinity : float
```

Negative infinity.

```ml
val nan : float
```
A special floating-point value denoting the result of an undefined operation such as \(0.0 / 0.0\). Stands for “not a number”. Any floating-point operation with \texttt{nan} as argument returns \texttt{nan} as result. As for floating-point comparisons, \(=, <, <=, >, >=\) return \texttt{false} and \texttt{<>} returns \texttt{true} if one or both of their arguments is \texttt{nan}.

\begin{verbatim}
val max_float : float
  The largest positive finite value of type \texttt{float}.

val min_float : float
  The smallest positive, non-zero, non-denormalized value of type \texttt{float}.

val epsilon_float : float
  The difference between 1.0 and the smallest exactly representable floating-point number greater than 1.0.

type fpclass =
  | FP_normal
    Normal number, none of the below
  | FP_subnormal
    Number very close to 0.0, has reduced precision
  | FP_zero
    Number is 0.0 or -0.0
  | FP_infinite
    Number is positive or negative infinity
  | FP_nan
    Not a number: result of an undefined operation

val classify_float : float -> fpclass
  Return the class of the given floating-point number: normal, subnormal, zero, infinite, or not a number.
\end{verbatim}

\textbf{String operations}

More string operations are provided in module \texttt{String}\[20.33].

\begin{verbatim}
val (\^) : string -> string -> string
  String concatenation.
\end{verbatim}
Character operations

More character operations are provided in module Char.[20.5].

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 the latter
  may generate a compiler warning; writing ignore(f x) instead avoids the warning.

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 (if it begins with 0x or 0X), octal (if it begins with 0o or 0O), or binary (if it
  begins with 0b or 0B). Raise Failure "int_of_string" if the given string is not a valid
  representation of an integer, or if the integer represented exceeds the range of integers
  representable in type int.

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. Raise Failure "float_of_string" 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\[20.17\].

val (@) : 'a list -> 'a list -> 'a list
    List concatenation.

Input/output

type in_channel
    The type of input channel.

type out_channel
    The type of output channel.

val stdin : in_channel
    The standard input for the process.

val stdout : out_channel
    The standard output for the process.

val stderr : out_channel
    The 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 and flush 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.
Chapter 19. The core library

General output functions

type open_flag =
  | Open_rdonly
    open for reading.
  | Open_wronly
    open for writing.
  | Open_append
    open for appending: always write at end of file.
  | Open_creat
    create the file if it does not exist.
  | Open_trunc
    empty the file if it already exists.
  | Open_excl
    fail if Open_creat and 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.

Opening modes for Pervasives.open_out_gen\[19.2\] and Pervasives.open_in_gen\[19.2\].

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 Pervasives.open_out\[19.2\], 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 Pervasives.open_out\[19.2\].

val open_out_gen : open_flag list -> int -> string -> out_channel
  open_out_gen mode perm filename opens the named file for writing, as described above. The extra argument mode specify the opening mode. The extra argument perm specifies the file permissions, in case the file must be created. Pervasives.open_out\[19.2\] and Pervasives.open_out_bin\[19.2\] 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 flush_all : unit -> unit
Flush all open output channels; ignore errors.

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 oc buf pos len writes len characters from string buf, starting at offset pos, to the given output channel oc. Raise Invalid_argument "output" if pos and len do not designate a valid substring of buf.

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 (4 bytes, big-endian) on the given output channel. The given integer is taken modulo $2^{32}$. The only reliable way to read it back is through the Pervasives.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 Pervasives.input_value. See the description of module Marshal for more information. Pervasives.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. Does not work on channels opened with the Open_append flag (returns unspecified results).

val out_channel_length : out_channel -> int
Return the size (number of characters) of the regular file on which the given channel is opened. If the channel is opened on a file that is not a regular file, the result is meaningless.

```ml
val close_out : out_channel -> unit
Close the given channel, flushing all buffered write operations. Output functions raise a Sys_error exception when they are applied to a closed output channel, except close_out and flush, which do nothing when applied to an already closed channel. Note that close_out may raise Sys_error if the operating system signals an error when flushing or closing.
```

```ml
val close_out_noerr : out_channel -> unit
Same as close_out, but ignore all errors.
```

```ml
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**

```ml
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.
```

```ml
val open_in_bin : string -> in_channel
Same as Pervasives.open_in [19.2], 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 Pervasives.open_in [19.2].
```

```ml
val open_in_gen : open_flag list -> int -> string -> in_channel
open_in_gen mode perm filename opens the named file for reading, as described above. The extra arguments mode and perm specify the opening mode and file permissions. Pervasives.open_in [19.2] and Pervasives.open_in_bin [19.2] are special cases of this function.
```

```ml
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.
```

```ml
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 ic buf pos len reads up to len characters from the given channel ic, storing them in string buf, starting at character number pos. 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 not all requested len characters were read, either because no more characters were available at that time, or because the implementation found it convenient to do a partial read; input must be called again to read the remaining characters, if desired. (See also `Pervasives.really_input` for reading exactly len characters.) Exception `Invalid_argument "input"` is raised if pos and len do not designate a valid substring of buf.
```

```
val really_input : in_channel -> string -> int -> int -> unit
  really_input ic buf pos len reads len characters from channel ic, storing them in string buf, starting at character number pos. Raise `End_of_file` if the end of file is reached before len characters have been read. Raise `Invalid_argument "really_input"` if pos and len do not designate a valid substring of buf.
```

```
val input_byte : in_channel -> int
  Same as `Pervasives.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 (4 bytes, big-endian) from the given input channel. See `Pervasives.output_binary_int` for more information, in particular concerning the lack of type safety.
```

```
val input_value : in_channel -> 'a
  Read the representation of a structured value, as produced by `Pervasives.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
```

Chapter 19. The core library

Return the size (number of characters) of the regular file on which the given channel is opened. If the channel is opened on a file that is not a regular file, the result is meaningless. The returned size does not take into account the end-of-line translations that can be performed when reading from a channel opened in text mode.

val close_in : in_channel -> unit

Close the given channel. Input functions raise a Sys.error exception when they are applied to a closed input channel, except close_in, which does nothing when applied to an already closed channel. Note that close_in may raise Sys.error if the operating system signals an error.

val close_in_noerr : in_channel -> unit

Same as close_in, but ignore all errors.

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.

Operations on large files

module LargeFile :

sig
  val seek_out : Pervasives.out_channel -> int64 -> unit
  val pos_out : Pervasives.out_channel -> int64
  val out_channel_length : Pervasives.out_channel -> int64
  val seek_in : Pervasives.in_channel -> int64 -> unit
  val pos_in : Pervasives.in_channel -> int64
  val in_channel_length : Pervasives.in_channel -> int64
end

Operations on large files. This sub-module provides 64-bit variants of the channel functions that manipulate file positions and file sizes. By representing positions and sizes by 64-bit integers (type int64) instead of regular integers (type int), these alternate functions allow operating on files whose sizes are greater than max_int.
References

type 'a ref = {
  mutable contents : 'a ;
}\n
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.

Operations on format strings

Format strings are used to read and print data using formatted input functions in module Scanf and formatted output in modules Printf and Format.

type ('a, 'b, 'c, 'd) format4 = ('a, 'b, 'c, 'c, 'c, 'd) format6
Format strings have a general and highly polymorphic type ('a, 'b, 'c, 'd, 'e, 'f) format6. Type format6 is built in. The two simplified types, format and format4 below are included for backward compatibility with earlier releases of Objective Caml. 'a is the type of the parameters of the format, 'b is the type of the first argument given to %a and %t printing functions, 'c is the type of the argument transmitted to the first argument of "kprintf"-style functions, 'd is the result type for the "scanf"-style functions, 'e is the type of the receiver function for the "scanf"-style functions, 'f is the result type for the "printf"-style function.

val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string
Converts a format string into a string.

val format_of_string : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'c, 'c, 'd, 'e, 'f) format6
format_of_string s returns a format string read from the string literal s.

val (^^) :
('a, 'b, 'c, 'd, 'e, 'f) format6 ->
('f, 'b, 'c, 'e, 'g, 'h) format6 ->
('a, 'b, 'c, 'd, 'g, 'h) format6

f1 ^^ f2 concatenates formats f1 and f2. The result is a format that accepts arguments from f1, then arguments from f2.

Program termination

val exit : int -> 'a

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. All open output channels are flushed with flush_all. An implicit exit 0 is performed each time a program terminates normally. An implicit exit 2 is performed if the program terminates early 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 Pervasives.exit[19.2], or terminates, either normally or 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 20

The standard library

This chapter describes the functions provided by the OCaml 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 `Pervasives` 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:**

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Char</td>
<td>315</td>
<td>character operations</td>
</tr>
<tr>
<td>String</td>
<td>390</td>
<td>string operations</td>
</tr>
<tr>
<td>Array</td>
<td>309</td>
<td>array operations</td>
</tr>
<tr>
<td>List</td>
<td>350</td>
<td>list operations</td>
</tr>
<tr>
<td>StdLabels</td>
<td>385</td>
<td>labelized versions of the above 3 modules</td>
</tr>
<tr>
<td>Sort</td>
<td>384</td>
<td>sorting and merging lists</td>
</tr>
<tr>
<td>Hashtbl</td>
<td>338</td>
<td>hash tables and hash functions</td>
</tr>
<tr>
<td>Random</td>
<td>370</td>
<td>pseudo-random number generator</td>
</tr>
<tr>
<td>Set</td>
<td>381</td>
<td>sets over ordered types</td>
</tr>
<tr>
<td>Map</td>
<td>355</td>
<td>association tables over ordered types</td>
</tr>
<tr>
<td>Oo</td>
<td>363</td>
<td>useful functions on objects</td>
</tr>
<tr>
<td>Stack</td>
<td>384</td>
<td>last-in first-out stacks</td>
</tr>
<tr>
<td>Queue</td>
<td>369</td>
<td>first-in first-out queues</td>
</tr>
<tr>
<td>Buffer</td>
<td>313</td>
<td>string buffers that grow on demand</td>
</tr>
<tr>
<td>Lazy</td>
<td>347</td>
<td>delayed evaluation</td>
</tr>
<tr>
<td>Weak</td>
<td>397</td>
<td>references that don’t prevent objects from being garbage-collected</td>
</tr>
</tbody>
</table>

**Arithmetic:**

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Complex</td>
<td>316</td>
<td>Complex numbers</td>
</tr>
<tr>
<td>Int32</td>
<td>341</td>
<td>operations on 32-bit integers</td>
</tr>
<tr>
<td>Int64</td>
<td>344</td>
<td>operations on 64-bit integers</td>
</tr>
<tr>
<td>Nativeint</td>
<td>360</td>
<td>operations on platform-native integers</td>
</tr>
</tbody>
</table>

**Input/output:**

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Format</td>
<td>320</td>
<td>pretty printing with automatic indentation and line breaking</td>
</tr>
<tr>
<td>Marshal</td>
<td>358</td>
<td>marshaling of data structures</td>
</tr>
<tr>
<td>Printf</td>
<td>366</td>
<td>formatting printing functions</td>
</tr>
<tr>
<td>Scanf</td>
<td>372</td>
<td>formatted input functions</td>
</tr>
<tr>
<td>Digest</td>
<td>317</td>
<td>MD5 message digest</td>
</tr>
</tbody>
</table>

**Parsing:**

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Genlex</td>
<td>337</td>
<td>a generic lexer over streams</td>
</tr>
<tr>
<td>Lexing</td>
<td>348</td>
<td>the run-time library for lexers generated by ocamllex</td>
</tr>
<tr>
<td>Parsing</td>
<td>363</td>
<td>the run-time library for parsers generated by ocamlyacc</td>
</tr>
<tr>
<td>Stream</td>
<td>358</td>
<td>basic functions over streams</td>
</tr>
</tbody>
</table>
System interface:

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg</td>
<td>307</td>
<td>parsing of command line arguments</td>
</tr>
<tr>
<td>Callback</td>
<td>315</td>
<td>registering Caml functions to be called from C</td>
</tr>
<tr>
<td>Filename</td>
<td>318</td>
<td>operations on file names</td>
</tr>
<tr>
<td>Gc</td>
<td>332</td>
<td>memory management control and statistics</td>
</tr>
<tr>
<td>Printexc</td>
<td>365</td>
<td>a catch-all exception handler</td>
</tr>
<tr>
<td>Sys</td>
<td>393</td>
<td>system interface</td>
</tr>
</tbody>
</table>

20.1 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, Bool, Set, Clear, String, Set_string, Int, Set_int, Float, Set_float, Tuple, Symbol, and Rest. Unit, Set and Clear keywords take no argument. A Rest keyword takes the remaining of the command line as arguments. Every other keyword takes the following word on the command line as argument. Arguments not preceded by a keyword are called anonymous arguments.

Examples (cmd is assumed to be the command name):

- \texttt{cmd -flag} (a unit option)
- \texttt{cmd -int 1} (an int option with argument 1)
- \texttt{cmd -string foobar} (a string option with argument "foobar")
- \texttt{cmd -float 12.34} (a float option with argument 12.34)
- \texttt{cmd a b c} (three anonymous arguments: "a", "b", and "c")
- \texttt{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 |
| Bool of (bool -> unit) |
| Call the function with a bool argument |
| Set of bool Pervasives.ref |
| Set the reference to true |
| Clear of bool Pervasives.ref |
| Set the reference to false |
| String of (string -> unit) |
| Call the function with a string argument |
```
| Set_string of string Pervasives.ref  
  Set the reference to the string argument |
| Int of (int -> unit)  
  Call the function with an int argument |
| Set_int of int Pervasives.ref  
  Set the reference to the int argument |
| Float of (float -> unit)  
  Call the function with a float argument |
| Set_float of float Pervasives.ref  
  Set the reference to the float argument |
| Tuple of spec list  
  Take several arguments according to the spec list |
| Symbol of string list * (string -> unit)  
  Take one of the symbols as argument and call the function with the symbol |
| 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.

type key = string

type doc = string

type usage_msg = string

type anon_fun = string -> unit

val parse : (key * spec * doc) list -> anon_fun -> usage_msg -> unit
  Arg.parse speclist anon_fun 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. anon_fun is called on anonymous arguments. The functions in spec and anon_fun 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 to standard error 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 anon_fun, doc) in speclist.

By default, parse recognizes two unit options, -help and --help, which will print to standard output usage_msg and the list of options, and exit the program. You can override this behaviour by specifying your own -help and --help options in speclist.
val parse_argv :
?current:int Pervasives.ref ->
string array ->
(key * spec * doc) list -> anon_fun -> usage_msg -> unit

Arg.parse_argv ~current args speclist anon_fun usage_msg parses the array args as
if it were the command line. It uses and updates the value of ~current (if given), or
Arg.current. You must set it before calling parse_argv. The initial value of current is
the index of the program name (argument 0) in the array. If an error occurs,
Arg.parse_argv raises Arg.Bad with the error message as argument. If option -help or
--help is given, Arg.parse_argv raises Arg.Help with the help message as argument.

exception Help of string
Raised by Arg.parse_argv when the user asks for help.

exception Bad of string
Functions in spec or anon_fun can raise Arg.Bad with an error message to reject invalid
arguments. Arg.Bad is also raised by Arg.parse_argv in case of an error.

val usage : (key * spec * doc) list -> usage_msg -> unit
Arg.usage speclist usage_msg prints to standard error an error message that includes
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 usage_string : (key * spec * doc) list -> usage_msg -> string
Returns the message that would have been printed by Arg.usage, if provided with the
same parameters.

val align : (key * spec * doc) list -> (key * spec * doc) list
Align the documentation strings by inserting spaces at the first space, according to the
length of the keyword. Use a space as the first character in a doc string if you want to align
the whole string. The doc strings corresponding to Symbol arguments are aligned on the
next line.

val current : int Pervasives.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. Arg.parse uses the initial value
of Arg.current as the index of argument 0 (the program name) and starts parsing
arguments at the next element.

20.2 Module Array : Array operations.

val length : 'a array -> int
Return the length (number of elements) of the given array.

```ml
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. You can also write a.(n) instead of Array.get a n.

Raise Invalid_argument "index out of bounds" if n is outside the range 0 to (Array.length a - 1).
```

```ml
val set : 'a array -> int -> 'a -> unit

Array.set a n x modifies array a in place, replacing element number n with x. You can also write a.(n) <- x instead of Array.set a n x.

Raise Invalid_argument "index out of bounds" if n is outside the range 0 to Array.length a - 1.
```

```ml
val make : 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.

Raise Invalid_argument if n < 0 or n > Sys.max_array_length. If the value of x is a floating-point number, then the maximum size is only Sys.max_array_length / 2.
```

```ml
val create : int -> 'a -> 'a array

Deprecated. Array.create is an alias for Array.make.
```

```ml
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.

Raise Invalid_argument if n < 0 or n > Sys.max_array_length. If the return type of f is float, then the maximum size is only Sys.max_array_length / 2.
```

```ml
val make_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).

Raise Invalid_argument if dimx or dimy is negative or greater than Sys.max_array_length. If the value of e is a floating-point number, then the maximum size is only Sys.max_array_length / 2.
```

```ml
val create_matrix : int -> int -> 'a -> 'a array array
```
Deprecated. \texttt{Array.create_matrix} is an alias for \texttt{Array.make_matrix}\cite{20.2}.

\begin{verbatim}
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 \texttt{Array.append}, but concatenates 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 \texttt{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 \texttt{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 \texttt{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: [\mid f a.(0); f a.(1); ...; f a.(Array.length a - 1) \mid].
\end{verbatim}
val iteri : (int -> 'a -> unit) -> 'a array -> unit
   Same as Array.iter, but the function is applied to the index of the element as first argument, and the element itself as second argument.

val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array
   Same as Array.map, 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 : ('a -> 'b -> 'b) -> 'a array -> 'b -> 'b
   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.

Sorting

val sort : ('a -> 'a -> int) -> 'a array -> unit
   Sort an array in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see below for a complete specification). For example, Pervasives.compare is a suitable comparison function, provided there are no floating-point NaN values in the data. After calling Array.sort, the array is sorted in place in increasing order. Array.sort is guaranteed to run in constant heap space and (at most) logarithmic stack space.

   Specification of the comparison function: Let a be the array and cmp the comparison function. The following must be true for all x, y, z in a :

   • cmp x y > 0 if and only if cmp y x < 0
   • if cmp x y ≥ 0 and cmp y z ≥ 0 then cmp x z ≥ 0

   When Array.sort returns, a contains the same elements as before, reordered in such a way that for all i and j valid indices of a :

   • cmp a.(i) a.(j) ≥ 0 if and only if i ≥ j

val stable_sort : ('a -> 'a -> int) -> 'a array -> unit
   Same as Array.sort, but the sorting algorithm is stable (i.e. elements that compare equal are kept in their original order) and not guaranteed to run in constant heap space.

   The current implementation uses Merge Sort. It uses n/2 words of heap space, where n is the length of the array. It is usually faster than the current implementation of Array.sort.
Chapter 20. The standard library

val fast_sort : ('a -> 'a -> int) -> 'a array -> unit
    Same as Array.sort\[20.2\] or Array.stable_sort\[20.2\], whichever is faster on typical input.

20.3 Module Buffer : Extensible string buffers.

This module implements string buffers that automatically expand as necessary. It provides accumu-
lative concatenation of strings in quasi-linear time (instead of quadratic time when strings are
collapsed 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. If n is not between 1 and
    Sys.max_string_length\[20.34\], it will be clipped to that interval.

val contents : t -> string
    Return a copy of the current contents of the buffer. The buffer itself is unchanged.

val sub : t -> int -> int -> string
    Buffer.sub b off len returns (a copy of) the substring of the current contents of the
    buffer b starting at offset off of length len bytes. May raise Invalid_argument if out of
    bounds request. The buffer itself is unaffected.

val blit : t -> int -> string -> int -> int -> unit
    Buffer.blit src srcoff dst dstoff len copies len characters from the current contents
    of the buffer src, starting at offset srcoff to string dst, starting at character dstoff.
    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.
    Since: 3.11.2

val nth : t -> int -> char
    get the n-th character of the buffer. Raise Invalid_argument if index out of bounds

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 Buffer.create n. For
    long-lived buffers that may have grown a lot, reset allows faster reclamation 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_substitute : t -> (string -> string) -> string -> unit
    add_substitute b f s appends the string pattern s at the end of the buffer b with
    substitution. The substitution process looks for variables into the pattern and substitutes
    each variable name by its value, as obtained by applying the mapping f to the variable
    name. Inside the string pattern, a variable name immediately follows a non-escaped $ character
    and is one of the following:
    • a non empty sequence of alphanumeric or _ characters,
    • an arbitrary sequence of characters enclosed by a pair of matching parentheses or curly
      brackets. An escaped $ character is a $ that immediately follows a backslash character;
      it then stands for a plain $. Raise Not_found if the closing character of a parenthesized
      variable cannot be found.

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 -> Pervasives.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 : Pervasives.out_channel -> t -> unit
    output_buffer oc b writes the current contents of buffer b on the output channel oc.
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.

```ocaml
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 passing as first argument to raise_constant or raise_with_arg.
```

Module Char : Character operations.

```ocaml
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
Convert the given character to its equivalent lowercase character.

val uppercase : char -> char
Convert the given character to its equivalent uppercase character.

type t = char
An alias for the type of characters.

val compare : t -> t -> int
The comparison function for characters, with the same specification as Pervasives.compare. Along with the type t, this function compare allows the module Char to be passed as argument to the functors Set.Make and Map.Make.
```
20.6 Module Complex : Complex numbers.

This module provides arithmetic operations on complex numbers. Complex numbers are represented by their real and imaginary parts (cartesian representation). Each part is represented by a double-precision floating-point number (type float).

```haskell
type t = {
    re : float ;
    im : float ;
}

The type of complex numbers. re is the real part and im the imaginary part.

val zero : t
    The complex number 0.

val one : t
    The complex number 1.

val i : t
    The complex number i.

val neg : t -> t
    Unary negation.

val conj : t -> t
    Conjugate: given the complex x + i.y, returns x - i.y.

val add : t -> t -> t
    Addition

val sub : t -> t -> t
    Subtraction

val mul : t -> t -> t
    Multiplication

val inv : t -> t
    Multiplicative inverse (1/z).

val div : t -> t -> t
    Division

val sqrt : t -> t
```
Square root. The result \( x + i.y \) is such that \( x > 0 \) or \( x = 0 \) and \( y \geq 0 \). This function has a discontinuity along the negative real axis.

val norm2 : t -> float
Norm squared: given \( x + i.y \), returns \( x^2 + y^2 \).

val norm : t -> float
Norm: given \( x + i.y \), returns \( \sqrt{x^2 + y^2} \).

val arg : t -> float
Argument. The argument of a complex number is the angle in the complex plane between the positive real axis and a line passing through zero and the number. This angle ranges from \(-\pi\) to \(\pi\). This function has a discontinuity along the negative real axis.

val polar : float -> float -> t
\( \text{polar norm arg} \) returns the complex having norm \( \text{norm} \) and argument \( \text{arg} \).

val exp : t -> t
Exponentiation. \( \text{exp z} \) returns \( e \) to the \( z \) power.

val log : t -> t
Natural logarithm (in base \( e \)).

val pow : t -> t -> t
Power function. \( \text{pow z1 z2} \) returns \( z1 \) to the \( z2 \) power.

20.7 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
\( \text{Digest.substring s ofs len} \) returns the digest of the substring of \( s \) starting at character number \( ofs \) and containing \( len \) characters.

val channel : Pervasives.in_channel -> int -> t
If \texttt{len} is nonnegative, \texttt{Digest.channel ic len} reads \texttt{len} characters from channel \texttt{ic} and returns their digest, or raises \texttt{End_of_file} if end-of-file is reached before \texttt{len} characters are read. If \texttt{len} is negative, \texttt{Digest.channel ic len} reads all characters from \texttt{ic} until end-of-file is reached and return their digest.

\textbf{val file} : \texttt{string -> t}

Return the digest of the file whose name is given.

\textbf{val output} : \texttt{Pervasives.out_channel -> t -> unit}

Write a digest on the given output channel.

\textbf{val input} : \texttt{Pervasives.in_channel -> t}

Read a digest from the given input channel.

\textbf{val to_hex} : \texttt{t -> string}

Return the printable hexadecimal representation of the given digest.

\subsection{20.8 Module Filename : Operations on file names.}

\textbf{val current_dir_name} : \texttt{string}

The conventional name for the current directory (e.g. . in Unix).

\textbf{val parent_dir_name} : \texttt{string}

The conventional name for the parent of the current directory (e.g. .. in Unix).

\textbf{val dir_sep} : \texttt{string}

The directory separator (e.g. / in Unix).

\textit{Since: 3.11.2}

\textbf{val concat} : \texttt{string -> string -> string}

\texttt{concat dir file} returns a file name that designates file \texttt{file} in directory \texttt{dir}.

\textbf{val is_relative} : \texttt{string -> bool}

Return \texttt{true} if the file name is relative to the current directory, \texttt{false} if it is absolute (i.e. in Unix, starts with /).

\textbf{val is_implicit} : \texttt{string -> bool}

Return \texttt{true} if the file name is relative and does not start with an explicit reference to the current directory (./ or ../ in Unix), \texttt{false} if it starts with an explicit reference to the root directory or the current directory.

\textbf{val check_suffix} : \texttt{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 and not including a directory separator, .xyz for instance.
Raise Invalid_argument if the given name does not contain an extension.

val basename : 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[20.34]), references to basename name (which is a relative file name) designate the same file as name before the call to Sys.chdir[20.34]. The result is not specified if the argument is not a valid file name (for example, under Unix if there is a NUL character in the string).

val dirname : string -> string
See Filename.basename[20.8].

val temp_file : ?temp_dir:string -> string -> string -> string

temp_file prefix suffix returns the name of a fresh 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. The optional argument temp_dir indicates the temporary directory to use, defaulting to Filename.temp_dir_name[20.8]. The temporary file is created empty, with permissions 0o600 (readable and writable only by the file owner). The file is guaranteed to be different from any other file that existed when temp_file was called. Raise Sys_error if the file could not be created.

Before 3.11.2 no ?temp_dir optional argument

val open_temp_file : ?mode:Pervasives.open_flag list -> ?temp_dir:string -> string -> string -> string * Pervasives.out_channel

Same as Filename.temp_file[20.8], but returns both the name of a fresh temporary file, and an output channel opened (atomically) on this file. This function is more secure than temp_file: there is no risk that the temporary file will be modified (e.g. replaced by a symbolic link) before the program opens it. The optional argument mode is a list of additional flags to control the opening of the file. It can contain one or several of Open_append, Open_binary, and Open_text. The default is [Open_text] (open in text mode). Raise Sys_error if the file could not be opened.

Before 3.11.2 no ?temp_dir optional argument
val temp_dir_name : string

The name of the temporary directory: Under Unix, the value of the TMPDIR environment variable, or "/tmp" if the variable is not set. Under Windows, the value of the TEMP environment variable, or "." if the variable is not set.

Since: 3.09.1

val quote : string -> string

Return a quoted version of a file name, suitable for use as one argument in a command line, escaping all meta-characters. Warning: under Windows, the output is only suitable for use with programs that follow the standard Windows quoting conventions.

20.9 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.

For a gentle introduction to the basics of pretty-printing using Format, read http://caml.inria.fr/resources/doc/guides/format.html. 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 Format.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 0; print_string "x ="; print_space (); print_int 1; close_box (); print_newline () that prints x = 1 within a pretty-printing box, can be abbreviated as printf "@[x@ %i@]@." "x =" 1, or even shorter printf "@[x =@ %i@]@." 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, flush the pretty-printer to display all the remaining material, e.g. evaluate print_newline ()

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 \texttt{print_newline} function) after each phrase. Each phrase is therefore executed in the initial state of the pretty-printer.

Warning: the material output by the following functions is delayed in the pretty-printer queue in order to compute the proper line breaking. Hence, you should not mix calls to the printing functions of the basic I/O system with calls to the functions of this module: this could result in some strange output seemingly unrelated with the evaluation order of printing commands.

\textbf{Boxes}

\begin{verbatim}
val open_box : int -> unit
    opens a new pretty-printing box with offset \texttt{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, \texttt{d} is added to the current indentation.

val close_box : unit -> unit
    Closes the most recently opened pretty-printing box.
\end{verbatim}

\textbf{Formatting functions}

\begin{verbatim}
val print_string : string -> unit
    \texttt{print_string str} prints \texttt{str} in the current box.

val print_as : int -> string -> unit
    \texttt{print_as len str} prints \texttt{str} as if it were of length \texttt{len}.

val print_int : int -> unit
    Prints an integer in the current box.

val print_float : float -> unit
    Prints a floating point number in the current box.

val print_char : char -> unit
    Prints a character in the current box.

val print_bool : bool -> unit
    Prints a boolean in the current box.
\end{verbatim}
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

Inserts 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 the current 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

Flushes 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

Forces a newline in the current box. Not the normal way of pretty-printing, you should prefer break hints.

val print_if_newline : unit -> unit

Executes 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. If d is too large, the right margin is set to the maximum admissible value (which is greater than 10^10).

val get_margin : unit -> int

Returns 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. If d is too large, the limit is set to the maximum admissible value (which is greater than \(10^{10}\)).

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 smaller than 2.

val get_max_boxes : unit -> int
  Returns the maximum number of boxes allowed before ellipsis.

val over_max_boxes : unit -> bool
  Tests if the maximum number of boxes allowed have already been opened.

Advanced formatting

val open_hbbox : unit -> unit
  open_hbbox () 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_vbbox : int -> unit
  open_vbbox 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_hbbox : int -> unit
  open_hbbox 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_hbbox : int -> unit
  open_hbbox 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
Opens a tabulation box.

val close_tbox : unit -> unit
Closes 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
Sets 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.

Semantics Tags

type tag = string
Semantics tags (or simply tags) are used to decorate printed entities for user's defined purposes, e.g. setting font and giving size indications for a display device, or marking delimitation of semantics entities (e.g. HTML or TeX elements or terminal escape sequences).

By default, those tags do not influence line breaking calculation: the tag “markers” are not considered as part of the printing material that drives line breaking (in other words, the length of those strings is considered as zero for line breaking).

Thus, tag handling is in some sense transparent to pretty-printing and does not interfere with usual pretty-printing. Hence, a single pretty printing routine can output both simple “verbatim” material or richer decorated output depending on the treatment of tags. By default, tags are not active, hence the output is not decorated with tag information. Once set_tags is set to true, the pretty printer engine honours tags and decorates the output accordingly.
When a tag has been opened (or closed), it is both and successively “printed” and “marked”. Printing a tag means calling a formatter specific function with the name of the tag as argument: that “tag printing” function can then print any regular material to the formatter (so that this material is enqueued as usual in the formatter queue for further line-breaking computation). Marking a tag means to output an arbitrary string (the “tag marker”), directly into the output device of the formatter. Hence, the formatter specific “tag marking” function must return the tag marker string associated to its tag argument. Being flushed directly into the output device of the formatter, tag marker strings are not considered as part of the printing material that drives line breaking (in other words, the length of the strings corresponding to tag markers is considered as zero for line breaking). In addition, advanced users may take advantage of the specificity of tag markers to be precisely output when the pretty printer has already decided where to break the lines, and precisely when the queue is flushed into the output device.

In the spirit of HTML tags, the default tag marking functions output tags enclosed in "<" and ">": hence, the opening marker of tag \texttt{t} is "\texttt{<t>}" and the closing marker "\texttt{</t>}".

Default tag printing functions just do nothing.

Tag marking and tag printing functions are user definable and can be set by calling \texttt{set_formatter_tag_functions}.

\begin{verbatim}
val open_tag : tag -> unit
  open_tag \texttt{t} opens the tag named \texttt{t}; the print_open_tag function of the formatter is called with \texttt{t} as argument; the tag marker \texttt{mark_open_tag \texttt{t}} will be flushed into the output device of the formatter.

val close_tag : unit -> unit
  close_tag () closes the most recently opened tag \texttt{t}. In addition, the print_close_tag function of the formatter is called with \texttt{t} as argument. The marker \texttt{mark_close_tag \texttt{t}} will be flushed into the output device of the formatter.

val set_tags : bool -> unit
  set_tags \texttt{b} turns on or off the treatment of tags (default is off).

val set_print_tags : bool -> unit
val set_mark_tags : bool -> unit
  set_print_tags \texttt{b} turns on or off the printing of tags, while set_mark_tags \texttt{b} turns on or off the output of tag markers.

val get_print_tags : unit -> bool
val get_mark_tags : unit -> bool
  Return the current status of tags printing and tags marking.
\end{verbatim}

### Redirecting the standard formatter output

\begin{verbatim}
val set_formatter_out_channel : Pervasives.out_channel -> unit
  Redirect the pretty-printer output to the given channel. (All the output functions of the standard formatter are set to the default output functions printing to the given channel.)
\end{verbatim}
val set_formatter_output_functions : 
  (string -> int -> int -> unit) -> (unit -> unit) -> unit

  set_formatter_output_functions out flush redirects the relevant pretty-printer output 
  functions to the functions out and flush.

  The out function performs the pretty-printer string 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 (via 
  conversion %!, pretty-printing indications @? or @., or using low level function 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 standard formatter pretty printing

The Format module is versatile enough to let you completely redefine the meaning of pretty printing: 
you may provide your own functions to define how to handle indentation, line breaking, and even 
printing of all the characters that have to be printed!

val set_all_formatter_output_functions :
  out:(string -> int -> int -> unit) -> 
  flush:(unit -> unit) -> 
  newline:(unit -> unit) -> 
  spaces:(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 space characters) 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. Useful to record the current setting and restore it afterwards.
Changing the meaning of printing semantics tags

type formatter_tag_functions = {
  mark_open_tag : tag -> string ;
  mark_close_tag : tag -> string ;
  print_open_tag : tag -> unit ;
  print_close_tag : tag -> unit ;
}

The tag handling functions specific to a formatter: mark versions are the “tag marking” functions that associate a string marker to a tag in order for the pretty-printing engine to flush those markers as 0 length tokens in the output device of the formatter. print versions are the “tag printing” functions that can perform regular printing when a tag is closed or opened.

val set_formatter_tag_functions : formatter_tag_functions -> unit

set_formatter_tag_functions tag_funs changes the meaning of opening and closing tags to use the functions in tag_funs.

When opening a tag name t, the string t is passed to the opening tag marking function (the mark_open_tag field of the record tag_funs), that must return the opening tag marker for that name. When the next call to close_tag () happens, the tag name t is sent back to the closing tag marking function (the mark_close_tag field of record tag_funs), that must return a closing tag marker for that name.

The print_ field of the record contains the functions that are called at tag opening and tag closing time, to output regular material in the pretty-printer queue.

val get_formatter_tag_functions : unit -> formatter_tag_functions

Return the current tag functions of the pretty-printer.

Multiple formatted output

type formatter

Abstract data corresponding to a pretty-printer (also called a formatter) and all its machinery.

Defining new pretty-printers permits unrelated output of material in parallel on several output channels. All the parameters of a pretty-printer are local to this 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 a Pervasives.out_channel output channel oc, a new formatter writing to that channel is simply 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 : Pervasives.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.

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. str_formatter is defined as formatter_of_buffer stdbuf.

val flush_str_formatter : unit -> string
  Returns the material printed with str_formatter, flushes the formatter and resets the corresponding buffer.

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. For instance, a formatter to the Pervasives.out_channel oc is returned by make_formatter (Pervasives.output oc) (fun () -> Pervasives.flush oc).

Basic functions to use with formatters

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_open_tag : formatter -> string -> unit
val pp_close_tag : 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_set_tags : formatter -> bool -> unit
val pp_set_print_tags : formatter -> bool -> unit
val pp_get_print_tags : formatter -> unit -> bool
val pp_set_mark_tags : formatter -> bool -> unit
val pp_get_mark_tags : formatter -> unit -> bool
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 -> Pervasives.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 →
  out:(string -> int -> int -> unit) →
  flush:(unit -> unit) →
  newline:(unit -> unit) →
  spaces:(int -> unit) → unit

val pp_get_all_formatter_output_functions :
  formatter →
  unit →
  (string -> int -> int -> unit) * (unit -> unit) * (unit -> unit) *
  (int -> unit)

val pp_set_formatter_tag_functions :
  formatter -> formatter_tag_functions -> unit

val pp_get_formatter_tag_functions :
  formatter -> unit -> formatter_tag_functions

  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.

printf like functions for pretty-printing.

val fprintf : formatter -> ('a, formatter, unit) Pervasives.format -> 'a
  fprintf ff fmt arg1 ... argN formats the arguments arg1 to argN according to the format
  string fmt, and outputs the resulting string on the formatter ff.

  The format fmt 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 specific
to the Format module.

  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 as obtained with open_hovbox
    2. For more details about boxes, see the various box opening functions open_*box.

  • @]: 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 ()
Chapter 20. The standard library

• @;: output a good break as with print_break. The \texttt{nspaces} and \texttt{offset} parameters of the break may be optionally specified with the following syntax: the \texttt{<} character, followed by an integer \texttt{nspaces} value, then an integer \texttt{offset}, and a closing \texttt{>} character. If no parameters are provided, the good break defaults to a space.

• @?: flush the pretty printer as with \texttt{print_flush ()}. This is equivalent to the conversion \texttt{\%}.!

• @.: flush the pretty printer and output a new line, as with \texttt{print_newline ()}.

• @<n>: print the following item as if it were of length \texttt{n}. Hence, \texttt{printf "@<0>\%s" arg} is equivalent to \texttt{print_as 0 arg}. If \texttt{@<n>} is not followed by a conversion specification, then the following character of the format is printed as if it were of length \texttt{n}.

• @}: open a tag. The name of the tag may be optionally specified with the following syntax: the \texttt{<} character, followed by an optional string specification, and the closing \texttt{>} character. The string specification is any character string that does not contain the closing character ‘\texttt{>}’.

• @}: close the most recently opened tag.

• @@: print a plain @ character.

Example: \texttt{printf "@[@%s@ %d@]@." "x =" 1} is equivalent to \texttt{open_box () \textnormal{; print_string \\
"x =";} \textnormal{; print_space () \textnormal{; print_int 1 \textnormal{; close_box () \textnormal{; print_newline ()}.}} It prints \texttt{x = 1} within a pretty-printing box.

val \texttt{printf : ('a, \textnormal{formatter, unit)} Pervasives.format \rightarrow 'a}}

Same as \texttt{fprintf} above, but output on \texttt{std_formatter}.

val \texttt{eprintf : ('a, \textnormal{formatter, unit)} Pervasives.format \rightarrow 'a}}

Same as \texttt{fprintf} above, but output on \texttt{err_formatter}.

val \texttt{sprintf : ('a, unit, string) Pervasives.format \rightarrow 'a}}

Same as \texttt{printf} above, but instead of printing on a formatter, returns a string containing the result of formatting the arguments. Note that the pretty-printer queue is flushed at the end of each call to \texttt{sprintf}.

In case of multiple and related calls to \texttt{sprintf} to output material on a single string, you should consider using \texttt{fprintf} with the predefined formatter \texttt{str_formatter} and call \texttt{flush_str_formatter ()} to get the final result.

Alternatively, you can use \texttt{Format.fprintf} with a formatter writing to a buffer of your own: flushing the formatter and the buffer at the end of pretty-printing returns the desired string.

val \texttt{ifprintf : \textnormal{formatter} \rightarrow ('a, \textnormal{formatter, unit)} Pervasives.format \rightarrow 'a}}

Same as \texttt{fprintf} above, but does not print anything. Useful to ignore some material when conditionally printing.

Since: 3.10.0
Formatted output functions with continuations.

val kfprintf : (formatter -> 'a) ->
  formatter -> ('b, formatter, unit, 'a) Pervasives.format4 -> 'b

  Same as fprintf above, but instead of returning immediately, passes the formatter to its
  first argument at the end of printing.

val ikfprintf : (formatter -> 'a) ->
  formatter -> ('b, formatter, unit, 'a) Pervasives.format4 -> 'b

  Same as kfprintf above, but does not print anything. Useful to ignore some material when
  conditionally printing.

  Since: 3.12.0

val ksprintf : (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b

  Same as sprintf above, but instead of returning the string, passes it to the first argument.

Deprecated

val bprintf : Buffer.t -> ('a, formatter, unit) Pervasives.format -> 'a

  A deprecated and error prone function. Do not use it.

  If you need to print to some buffer b, you must first define a formatter writing to b, using
  let to_b = formatter_of_buffer b; then use regular calls to Format.fprintf on
  formatter to_b.

val kprintf : (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b

  A deprecated synonym for ksprintf.

20.10 Module Gc : Memory management control and statistics;
finalised values.

type stat = {
  minor_words : float ;

  Number of words allocated in the minor heap since the program was started. This
  number is accurate in byte-code programs, but only an approximation in programs
  compiled to native code.

  promoted_words : float ;
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.

\texttt{major\_words : float ;}
Number of words allocated in the major heap, including the promoted words, since the program was started.

\texttt{minor\_collections : int ;}
Number of minor collections since the program was started.

\texttt{major\_collections : int ;}
Number of major collection cycles completed since the program was started.

\texttt{heap\_words : int ;}
Total size of the major heap, in words.

\texttt{heap\_chunks : int ;}
Number of contiguous pieces of memory that make up the major heap.

\texttt{live\_words : int ;}
Number of words of live data in the major heap, including the header words.

\texttt{live\_blocks : int ;}
Number of live blocks in the major heap.

\texttt{free\_words : int ;}
Number of words in the free list.

\texttt{free\_blocks : int ;}
Number of blocks in the free list.

\texttt{largest\_free : int ;}
Size (in words) of the largest block in the free list.

\texttt{fragments : int ;}
Number of wasted words due to fragmentation. These are 1-words free blocks placed between two live blocks. They are not available for allocation.

\texttt{compactions : int ;}
Number of heap compactions since the program was started.

\texttt{top\_heap\_words : int ;}
Maximum size reached by the major heap, in words.

\texttt{stack\_size : int ;}
Current size of the stack, in words.

The memory management counters are returned in a \texttt{stat} record.

The total amount of memory allocated by the program since it was started is (in words) \texttt{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 ;
    The size (in words) of the minor heap. Changing this parameter will trigger a minor collection. Default: 32k.
    mutable major_heap_increment : int ;
    The minimum number of words to add to the major heap when increasing it. Default: 124k.
    mutable space_overhead : int ;
    The major GC speed is computed from this parameter. This is the memory that will be "wasted" because the GC does not immediately collect unreachable blocks. It is expressed as a percentage of the memory used for live data. The GC will work more (use more CPU time and collect blocks more eagerly) if space_overhead is smaller. Default: 80.
    mutable verbose : int ;
    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:
    • 0x001 Start of major GC cycle.
    • 0x002 Minor collection and major GC slice.
    • 0x004 Growing and shrinking of the heap.
    • 0x008 Resizing of stacks and memory manager tables.
    • 0x010 Heap compaction.
    • 0x020 Change of GC parameters.
    • 0x040 Computation of major GC slice size.
    • 0x080 Calling of finalisation functions.
    • 0x100 Bytecode executable search at start-up.
    • 0x200 Computation of compaction triggering condition. Default: 0.
    mutable max_overhead : int ;
    Heap compaction is triggered when the estimated amount of "wasted" 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: 500.
    mutable stack_limit : int ;
    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.
    mutable allocation_policy : int ;
The policy used for allocating in the heap. Possible values are 0 and 1. 0 is the next-fit policy, which is quite fast but can result in fragmentation. 1 is the first-fit policy, which can be slower in some cases but can be better for programs with fragmentation problems. Default: 0.

}

The GC parameters are given as a control record. Note that these parameters can also be initialised by setting the OCAMLRUNPARAM environment variable. See the documentation of ocamlrun.

val stat : unit -> stat
Return the current values of the memory management counters in a stat record. This function examines every heap block to get the statistics.

val quick_stat : unit -> stat
Same as stat except that live_words, live_blocks, free_words, free_blocks, largest_free, and fragments are set to 0. This function is much faster than stat because it does not need to go through the heap.

val counters : unit -> float * float * float
Return (minor_words, promoted_words, major_words). This function is as fast at quick_stat.

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:
Gc.set { (Gc.get()) with Gc.verbose = 0x00d }

val minor : unit -> unit
Trigger a minor collection.

val major_slice : int -> int
Do a minor collection and a slice of major collection. The argument is the size of the slice, 0 to use the automatically-computed slice size. In all cases, the result is the computed slice size.

val major : unit -> unit
Do a minor collection and finish the current major collection cycle.

val full_major : unit -> unit
Do a minor collection, finish the current major collection cycle, and perform a complete new cycle. This will collect all currently unreachable blocks.
val compact : unit -> unit
Perform a full major collection and compact the heap. Note that heap compaction is a lengthy operation.

val print_stat : Pervasives.out_channel -> unit
Print the current values of the memory management counters (in human-readable form) into the channel argument.

val allocated_bytes : unit -> float
Return the total number of bytes allocated since the program was started. It is returned as a float to avoid overflow problems with int on 32-bit machines.

val finalise : ('a -> unit) -> 'a -> unit
finalise f v registers f as a finalisation function for v. v must be heap-allocated. f will be called with v as argument at some point between the first time v becomes unreachable and the time v is collected by the GC. Several functions can be registered for the same value, or even several instances of the same function. Each instance will be called once (or never, if the program terminates before v becomes unreachable).

The GC will call the finalisation functions in the order of deallocation. When several values become unreachable at the same time (i.e. during the same GC cycle), the finalisation functions will be called in the reverse order of the corresponding calls to finalise. If finalise is called in the same order as the values are allocated, that means each value is finalised before the values it depends upon. Of course, this becomes false if additional dependencies are introduced by assignments.

Anything reachable from the closure of finalisation functions is considered reachable, so the following code will not work as expected:

- let v = ... in Gc.finalise (fun x -> ...) v

Instead you should write:

- let f = fun x -> ... ;; let v = ... in Gc.finalise f v

The f function can use all features of O'Caml, including assignments that make the value reachable again. It can also loop forever (in this case, the other finalisation functions will not be called during the execution of f, unless it calls finalise_release). It can call finalise on v or other values to register other functions or even itself. It can raise an exception; in this case the exception will interrupt whatever the program was doing when the function was called.

finalise will raise Invalid_argument if v is not heap-allocated. Some examples of values that are not heap-allocated are integers, constant constructors, booleans, the empty array, the empty list, the unit value. The exact list of what is heap-allocated or not is implementation-dependent. Some constant values can be heap-allocated but never deallocated during the lifetime of the program, for example a list of integer constants; this is also implementation-dependent. You should also be aware that compiler optimisations may
duplicate some immutable values, for example floating-point numbers when stored into arrays, so they can be finalised and collected while another copy is still in use by the program.

The results of calling `String.make`, `String.create`, `Array.make`, and `Pervasives.ref` are guaranteed to be heap-allocated and non-constant except when the length argument is 0.

```ocaml
val finalise_release : unit -> unit
```

A finalisation function may call `finalise_release` to tell the GC that it can launch the next finalisation function without waiting for the current one to return.

```ocaml
type alarm
```

An alarm is a piece of data that calls a user function at the end of each major GC cycle. The following functions are provided to create and delete alarms.

```ocaml
val create_alarm : (unit -> unit) -> alarm
```

`create_alarm f` will arrange for `f` to be called at the end of each major GC cycle, starting with the current cycle or the next one. A value of type `alarm` is returned that you can use to call `delete_alarm`.

```ocaml
val delete_alarm : alarm -> unit
```

`delete_alarm a` will stop the calls to the function associated to `a`. Calling `delete_alarm a` again has no effect.

### 20.11 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.

Example: a lexer suitable for a desk calculator is obtained by

```ocaml
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:

```ocaml
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
| ...
```
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.

20.12 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. For best results, n should be on the order of the expected number of elements that will be in the table. The table grows as needed, so n is just an initial guess.

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 copy : ('a, 'b) t -> ('a, 'b) t
Return a copy of the given hashtable.

```ocaml
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 replace : ('a, 'b) t -> 'a -> 'b -> unit
  Hashtbl.replace tbl x y replaces the current binding of x in tbl by a binding of x to y. If x is unbound in tbl, a binding of x to y is added to tbl. This is functionally equivalent to Hashtbl.remove tbl x followed by Hashtbl.add tbl x y.

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. Each binding is presented exactly once to f. The order in which the bindings are passed to f is unspecified. However, if the table contains several bindings for the same key, they are passed to f in reverse order of introduction, that is, the most recent binding is passed first.

val fold : ('a -> 'b -> 'c -> 'c) -> ('a, 'b) t -> 'c -> 'c
  Hashtbl.fold f tbl init computes (f kN dN ... (f k1 d1 init)...), where k1 ... kN are the keys of all bindings in tbl, and d1 ... dN are the associated values. Each binding is presented exactly once to f. The order in which the bindings are passed to f is unspecified. However, if the table contains several bindings for the same key, they are passed to f in reverse order of introduction, that is, the most recent binding is passed first.

val length : ('a, 'b) t -> int
  Hashtbl.length tbl returns the number of bindings in tbl. Multiple bindings are counted multiply, so Hashtbl.length gives the number of times Hashtbl.iter calls its first argument.
```
Functorial interface

module type HashedType =
  sig
    type t
    The type of the hashtable keys.

    val equal : t -> t -> bool
    The equality predicate used to compare keys.

    val hash : t -> int
    A hashing function on keys. It must be such that if two keys are equal according to equal, then they 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, ((fun x y -> compare x y = 0), Hashtbl.hash) for comparing objects by structure and handling Pervasives.nan correctly, and (==), Hashtbl.hash for comparing objects by addresses (e.g. for cyclic keys).

  end

The input signature of the functor Hashtbl.Make.

module type S =
  sig
    type key
    type 'a t
    val create : int -> 'a t
    val clear : 'a t -> unit
    val copy : 'a t -> 'a t
    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 replace : 'a t -> key -> 'a -> unit
    val mem : 'a t -> key -> bool
    val iter : (key -> 'a -> unit) -> 'a t -> unit
    val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
    val length : 'a t -> int
  end
module Make :
  functor (H : HashedType) -> S with type key = H.t

Functor building an implementation of the hashtable structure. 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
Hash = Hashtbl.hash x associates a positive integer to any value of any type. It is guaranteed that if x = y or Pervasives.compare x y = 0, then hash x = hash y. Moreover, hash always terminates, even on cyclic structures.

val hash_param : int -> int -> 'a -> int
Hash = 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.

20.13 Module Int32 : 32-bit integers.

This module provides operations on the type int32 of signed 32-bit integers. Unlike the built-in int type, the type int32 is guaranteed to be exactly 32-bit wide on all platforms. All arithmetic operations over int32 are taken modulo 2^32.

Performance notice: values of type int32 occupy more memory space than values of type int, and arithmetic operations on int32 are generally slower than those on int. Use int32 only when the application requires exact 32-bit arithmetic.

val zero : int32
  The 32-bit integer 0.

val one : int32
  The 32-bit integer 1.
val minus_one : int32
  The 32-bit integer -1.

val neg : int32 -> int32
  Unary negation.

val add : int32 -> int32 -> int32
  Addition.

val sub : int32 -> int32 -> int32
  Subtraction.

val mul : int32 -> int32 -> int32
  Multiplication.

val div : int32 -> int32 -> int32
  Integer division. Raise Division_by_zero if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for Pervasives.(/)[19.2].

val rem : int32 -> int32 -> int32
  Integer remainder. If y is not zero, the result of Int32.rem x y satisfies the following property: x = Int32.add (Int32.mul (Int32.div x y) y) (Int32.rem x y). If y = 0, Int32.rem x y raises Division_by_zero.

val succ : int32 -> int32
  Successor. Int32.succ x is Int32.add x Int32.one.

val pred : int32 -> int32
  Predecessor. Int32.pred x is Int32.sub x Int32.one.

val abs : int32 -> int32
  Return the absolute value of its argument.

val max_int : int32
  The greatest representable 32-bit integer, $2^{31} - 1$.

val min_int : int32
  The smallest representable 32-bit integer, $-2^{31}$.

val logand : int32 -> int32 -> int32
  Bitwise logical and.

val logor : int32 -> int32 -> int32
Chapter 20. The standard library

Bitwise logical or.

val logxor : int32 -> int32 -> int32
  Bitwise logical exclusive or.

val lognot : int32 -> int32
  Bitwise logical negation

val shift_left : int32 -> int -> int32
  Int32.shift_left x y shifts x to the left by y bits. The result is unspecified if y < 0 or y >= 32.

val shift_right : int32 -> int -> int32
  Int32.shift_right x y shifts x to the right by y bits. This is an arithmetic shift: the sign bit of x is replicated and inserted in the vacated bits. The result is unspecified if y < 0 or y >= 32.

val shift_right_logical : int32 -> int -> int32
  Int32.shift_right_logical x y shifts x to the right by y bits. This is a logical shift: zeroes are inserted in the vacated bits regardless of the sign of x. The result is unspecified if y < 0 or y >= 32.

val of_int : int -> int32
  Convert the given integer (type int) to a 32-bit integer (type int32).

val to_int : int32 -> int
  Convert the given 32-bit integer (type int32) to an integer (type int). On 32-bit platforms, the 32-bit integer is taken modulo 2^{31}, i.e. the high-order bit is lost during the conversion. On 64-bit platforms, the conversion is exact.

val of_float : float -> int32
  Convert the given floating-point number to a 32-bit integer, discarding the fractional part (truncate towards 0). The result of the conversion is undefined if, after truncation, the number is outside the range [Int32.min_int, Int32.max_int].

val to_float : int32 -> float
  Convert the given 32-bit integer to a floating-point number.

val of_string : string -> int32
  Convert the given string to a 32-bit 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, or if the integer represented exceeds the range of integers representable in type int32.

val to_string : int32 -> string
Return the string representation of its argument, in signed decimal.

val bits_of_float : float -> int32

Return the internal representation of the given float according to the IEEE 754 floating-point “single format” bit layout. Bit 31 of the result represents the sign of the float; bits 30 to 23 represent the (biased) exponent; bits 22 to 0 represent the mantissa.

val float_of_bits : int32 -> float

Return the floating-point number whose internal representation, according to the IEEE 754 floating-point “single format” bit layout, is the given int32.

type t = int32

An alias for the type of 32-bit integers.

val compare : t -> t -> int

The comparison function for 32-bit integers, with the same specification as Pervasives.compare[19.2]. Along with the type t, this function compare allows the module Int32 to be passed as argument to the functors Set.Make[20.28] and Map.Make[20.18].

20.14 Module Int64 : 64-bit integers.

This module provides operations on the type int64 of signed 64-bit integers. Unlike the built-in int type, the type int64 is guaranteed to be exactly 64-bit wide on all platforms. All arithmetic operations over int64 are taken modulo $2^{64}$.

Performance notice: values of type int64 occupy more memory space than values of type int, and arithmetic operations on int64 are generally slower than those on int. Use int64 only when the application requires exact 64-bit arithmetic.

val zero : int64

The 64-bit integer 0.

val one : int64

The 64-bit integer 1.

val minus_one : int64

The 64-bit integer -1.

val neg : int64 -> int64

Unary negation.

val add : int64 -> int64 -> int64

Addition.

val sub : int64 -> int64 -> int64
Subtraction.

val mul : int64 -> int64 -> int64

Multiplication.

val div : int64 -> int64 -> int64

Integer division. Raise Division_by_zero if the second argument is zero. This division rounds the real quotient of its arguments towards zero, as specified for Pervasives./(19.2).

val rem : int64 -> int64 -> int64

Integer remainder. If \( y \) is not zero, the result of Int64.rem \( x \) \( y \) satisfies the following property: \( x = \text{Int64.add} (\text{Int64.mul} (\text{Int64.div} x y) y) \text{(Int64.rem} x y) \). If \( y = 0 \), Int64.rem \( x \) \( y \) raises Division_by_zero.

val succ : int64 -> int64

Successor. Int64.succ \( x \) is Int64.add \( x \) Int64.one.

val pred : int64 -> int64

Predecessor. Int64.pred \( x \) is Int64.sub \( x \) Int64.one.

val abs : int64 -> int64

Return the absolute value of its argument.

val max_int : int64

The greatest representable 64-bit integer, \( 2^{63} - 1 \).

val min_int : int64

The smallest representable 64-bit integer, \(-2^{63}\).

val logand : int64 -> int64 -> int64

Bitwise logical and.

val logor : int64 -> int64 -> int64

Bitwise logical or.

val logxor : int64 -> int64 -> int64

Bitwise logical exclusive or.

val lognot : int64 -> int64

Bitwise logical negation

val shift_left : int64 -> int -> int64

Int64.shift_left \( x \) \( y \) shifts \( x \) to the left by \( y \) bits. The result is unspecified if \( y < 0 \) or \( y \geq 64 \).
val shift_right : int64 -> int -> int64
  Int64.shift_right x y shifts x to the right by y bits. This is an arithmetic shift: the sign
  bit of x is replicated and inserted in the vacated bits. The result is unspecified if y < 0 or y
  >= 64.

val shift_right_logical : int64 -> int -> int64
  Int64.shift_right_logical x y shifts x to the right by y bits. This is a logical shift:
  zeroes are inserted in the vacated bits regardless of the sign of x. The result is unspecified if
  y < 0 or y >= 64.

val of_int : int -> int64
  Convert the given integer (type int) to a 64-bit integer (type int64).

val to_int : int64 -> int
  Convert the given 64-bit integer (type int64) to an integer (type int). On 64-bit platforms,
  the 64-bit integer is taken modulo 2^{63}, i.e. the high-order bit is lost during the conversion.
  On 32-bit platforms, the 64-bit integer is taken modulo 2^{31}, i.e. the top 33 bits are lost
  during the conversion.

val of_float : float -> int64
  Convert the given floating-point number to a 64-bit integer, discarding the fractional part
  (truncate towards 0). The result of the conversion is undefined if, after truncation, the
  number is outside the range \([\text{Int64.min_int} \to \text{Int64.max_int}]\).

val to_float : int64 -> float
  Convert the given 64-bit integer to a floating-point number.

val of_int32 : int32 -> int64
  Convert the given 32-bit integer (type int32) to a 64-bit integer (type int64).

val to_int32 : int64 -> int32
  Convert the given 64-bit integer (type int64) to a 32-bit integer (type int32). The 64-bit
  integer is taken modulo 2^{32}, i.e. the top 32 bits are lost during the conversion.

val of_nativeint : nativeint -> int64
  Convert the given native integer (type nativeint) to a 64-bit integer (type int64).

val to_nativeint : int64 -> nativeint
  Convert the given 64-bit integer (type int64) to a native integer. On 32-bit platforms, the
  64-bit integer is taken modulo 2^{32}. On 64-bit platforms, the conversion is exact.

val of_string : string -> int64
  Convert the given string to a 64-bit 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, or
  if the integer represented exceeds the range of integers representable in type int64.
val to_string : int64 -> string

Return the string representation of its argument, in decimal.

val bits_of_float : float -> int64

Return the internal representation of the given float according to the IEEE 754 floating-point “double format” bit layout. Bit 63 of the result represents the sign of the float; bits 62 to 52 represent the (biased) exponent; bits 51 to 0 represent the mantissa.

val float_of_bits : int64 -> float

Return the floating-point number whose internal representation, according to the IEEE 754 floating-point “double format” bit layout, is the given int64.

type t = int64

An alias for the type of 64-bit integers.

val compare : t -> t -> int

The comparison function for 64-bit integers, with the same specification as Pervasives.compare[19.2]. Along with the type t, this function compare allows the module Int64 to be passed as argument to the functors Set.Make[20.28] and Map.Make[20.18].

20.15 Module Lazy : Deferred computations.

type 'a t = 'a lazy_t

A value of type 'a Lazy.t is a deferred computation, called a suspension, that has a result of type 'a. The special expression syntax lazy (expr) makes a suspension of the computation of expr, without computing expr itself yet. ”Forcing” the suspension will then compute expr and return its result.

Note: lazy_t is the built-in type constructor used by the compiler for the lazy keyword. You should not use it directly. Always use Lazy.t instead.

Note: Lazy.force is not thread-safe. If you use this module in a multi-threaded program, you will need to add some locks.

Note: if the program is compiled with the -rectypes option, ill-founded recursive definitions of the form let rec x = lazy x or let rec x = lazy(lazy(...(lazy x))) are accepted by the type-checker and lead, when forced, to ill-formed values that trigger infinite loops in the garbage collector and other parts of the run-time system. Without the -rectypes option, such ill-founded recursive definitions are rejected by the type-checker.

exception Undefined

val force : 'a t -> 'a

force x forces the suspension x and returns its result. If x has already been forced, Lazy.force x returns the same value again without recomputing it. If it raised an exception, the same exception is raised again. Raise Undefined if the forcing of x tries to force x itself recursively.

val force_val : 'a t -> 'a
force_val x forces the suspension x and returns its result. If x has already been forced, force_val x returns the same value again without recomputing it. Raise Undefined if the forcing of x tries to force x itself recursively. If the computation of x raises an exception, it is unspecified whether force_val x raises the same exception or Undefined.

val lazy_from_fun : (unit -> 'a) -> 'a t
  lazy_from_fun f is the same as lazy (f ()) but slightly more efficient.

val lazy_from_val : 'a -> 'a t
  lazy_from_val v returns an already-forced suspension of v This is for special purposes only and should not be confused with lazy (v).

val lazy_is_val : 'a t -> bool
  lazy_is_val x returns true if x has already been forced and did not raise an exception.

20.16 Module Lexing: The run-time library for lexers generated by ocamllex.

Positions

type position = {
  pos_fname : string ;
  pos_lnum : int ;
  pos_bol : int ;
  pos_cnum : int ;
}

A value of type position describes a point in a source file. pos_fname is the file name; pos_lnum is the line number; pos_bol is the offset of the beginning of the line (number of characters between the beginning of the file and the beginning of the line); pos_cnum is the offset of the position (number of characters between the beginning of the file and the position).

See the documentation of type lexbuf for information about how the lexing engine will manage positions.

val dummy_pos : position
  A value of type position, guaranteed to be different from any valid position.

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 ;
mutable lex_mem : int array ;
mutable lex_start_p : position ;
mutable lex_curr_p : position ;
}

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.

At each token, the lexing engine will copy lex_curr_p to lex_start_p, then change the pos_cnum field of lex_curr_p by updating it with the number of characters read since the start of the lexbuf. The other fields are left unchanged by the lexing engine. In order to keep them accurate, they must be initialised before the first use of the lexbuf, and updated by the relevant lexer actions (i.e. at each end of line – see also new_line).

val from_channel : Pervasives.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 offset in the input stream of the first character of the matched string. The first character of the stream has offset 0.

val lexeme_end : lexbuf -> int

Lexing.lexeme_end lexbuf returns the offset in the input stream of the character following the last character of the matched string. The first character of the stream has offset 0.

val lexeme_start_p : lexbuf -> position

Like lexeme_start, but return a complete position instead of an offset.

val lexeme_end_p : lexbuf -> position

Like lexeme_end, but return a complete position instead of an offset.

val new_line : lexbuf -> unit

Update the lex_curr_p field of the lexbuf to reflect the start of a new line. You can call this function in the semantic action of the rule that matches the end-of-line character.

Since: 3.11.0

Miscellaneous functions

val flush_input : lexbuf -> unit

Discard the contents of the buffer and reset the current position to 0. The next use of the lexbuf will trigger a refill.

20.17 Module List: List operations.

Some functions are flagged as not tail-recursive. A tail-recursive function uses constant stack space, while a non-tail-recursive function uses stack space proportional to the length of its list argument, which can be a problem with very long lists. When the function takes several list arguments, an approximate formula giving stack usage (in some unspecified constant unit) is shown in parentheses.

The above considerations can usually be ignored if your lists are not longer than about 10000 elements.

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.
Chapter 20. The standard library

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. Raise Invalid_argument "List.nth" if n is negative.

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 @. Not tail-recursive (length of the first argument). The @ operator is not tail-recursive either.

val rev_append : 'a list -> 'a list -> 'a list
    List.rev_append l1 12 reverses l1 and concatenates it to 12. This is equivalent to List.rev[20.17] l1 @ 12, but rev_append is tail-recursive and more efficient.

val concat : 'a list list -> 'a list
    Concatenate a list of lists. The elements of the argument are all concatenated together (in the same order) to give the result. Not tail-recursive (length of the argument + length of the longest sub-list).

val flatten : 'a list list -> 'a list
    Same as concat. Not tail-recursive (length of the argument + length of the longest sub-list).

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. Not tail-recursive.

val rev_map : ('a -> 'b) -> 'a list -> 'b list
    List.rev_map f l gives the same result as List.rev[20.17] (List.map[20.17] f l), but is tail-recursive and more efficient.

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) ...)). Not tail-recursive.
Iterators on two lists

```ocaml
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.
```

```ocaml
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. Not tail-recursive.
```

```ocaml
val rev_map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list

List.rev_map2 f l1 l2 gives the same result as List.rev
(List.map2 f l1 l2), but is tail-recursive and more efficient.
```

```ocaml
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
b2 c2) ...) bn cn. Raise Invalid_argument if the two lists have different lengths.
```

```ocaml
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. Not
tail-recursive.
```

List scanning

```ocaml
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).
```

```ocaml
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).
```

```ocaml
val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool

Same as List.for_all, but for a two-argument predicate. Raise Invalid_argument if the two
lists have different lengths.
```

```ocaml
val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool

Same as List.exists, but for a two-argument predicate. Raise Invalid_argument if the two
lists have different lengths.
```

```ocaml
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 List.mem\textsuperscript{20.17}, 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
   filter p l returns all the elements of the list l that satisfy the predicate p. The order of the elements in the input list is preserved.

val find_all : ('a -> bool) -> 'a list -> 'a list
   find_all is another name for List.filter\textsuperscript{20.17}.

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. The order of the elements in the input list is preserved.

**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 List.assoc\textsuperscript{20.17}, but uses physical equality instead of structural equality to compare keys.

val mem_assoc : 'a -> ('a * 'b) list -> bool
   Same as List.assoc\textsuperscript{20.17}, 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 List.mem_assoc\textsuperscript{20.17}, 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. Not tail-recursive.

val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
Same as List.remove_assoc, but uses physical equality instead of structural equality to compare keys. Not tail-recursive.

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]). Not tail-recursive.

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. Not tail-recursive.

Sorting

val sort : ('a -> 'a -> int) -> 'a list -> 'a list
Sort a list in increasing order according to a comparison function. The comparison function must return 0 if its arguments compare as equal, a positive integer if the first is greater, and a negative integer if the first is smaller (see Array.sort for a complete specification). For example, Pervasives.compare is a suitable comparison function. The resulting list is sorted in increasing order. List.sort is guaranteed to run in constant heap space (in addition to the size of the result list) and logarithmic stack space.

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

val stable_sort : ('a -> 'a -> int) -> 'a list -> 'a list
Same as List.sort, but the sorting algorithm is guaranteed to be stable (i.e. elements that compare equal are kept in their original order).

The current implementation uses Merge Sort. It runs in constant heap space and logarithmic stack space.

val fast_sort : ('a -> 'a -> int) -> 'a list -> 'a list
Same as List.sort or List.stable_sort, whichever is faster on typical input.

val merge : ('a -> 'a -> int) -> 'a list -> 'a list
Merge two lists: Assuming that l1 and l2 are sorted according to the comparison function cmp, merge cmp l1 l2 will return a sorted list containing all the elements of l1 and l2. If several elements compare equal, the elements of l1 will be before the elements of l2. Not tail-recursive (sum of the lengths of the arguments).
20.18 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.

```ocaml
module type OrderedType =
  sig
    type t
      The type of the map keys.
    val compare : t -> t -> int
      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 Pervasives.compare.
  end

Input signature of the functor Map.Make.

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 is_empty : ’a t -> bool
      Test whether a map is empty or not.
    val mem : key -> ’a t -> bool
      mem x m returns true if m contains a binding for x, and false otherwise.
    val add : key -> ’a -> ’a t -> ’a t
  end
```
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 singleton : key -> 'a -> 'a t

singleton x y returns the one-element map that contains a binding y for x.
Since: 3.12.0

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 merge : (key -> 'a option -> 'b option -> 'c option) -> 'a t -> 'b t -> 'c t

merge f m1 m2 computes a map whose keys is a subset of keys of m1 and of m2. The presence of each such binding, and the corresponding value, is determined with the function f.
Since: 3.12.0

val compare : ('a -> 'a -> int) -> 'a t -> 'a t -> int

Total ordering between maps. The first argument is a total ordering used to compare data associated with equal keys in the two maps.

val equal : ('a -> 'a -> bool) -> 'a t -> 'a t -> bool

equal cmp m1 m2 tests whether the maps m1 and m2 are equal, that is, contain equal keys and associate them with equal data. cmp is the equality predicate used to compare the data associated with the keys.

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 bindings are passed to f in increasing order with respect to the ordering over the type of the keys.

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 (in increasing order), and d1 ... dN are the associated data.

val for_all : (key -> 'a -> bool) -> 'a t -> bool

for_all p m checks if all the bindings of the map satisfy the predicate p.
Since: 3.12.0

val exists : (key -> 'a -> bool) -> 'a t -> bool
exists p m checks if at least one binding of the map satisfy the predicate p.
Since: 3.12.0

val filter : (key -> 'a -> bool) -> 'a t -> 'a t
filter p m returns the map with all the bindings in m that satisfy predicate p.
Since: 3.12.0

val partition : (key -> 'a -> bool) -> 'a t -> 'a t * 'a t
partition p m returns a pair of maps (m1, m2), where m1 contains all the bindings of s that satisfy the predicate p, and m2 is the map with all the bindings of s that do not satisfy p.
Since: 3.12.0

val cardinal : 'a t -> int
Return the number of bindings of a map.
Since: 3.12.0

val bindings : 'a t -> (key * 'a) list
Return the list of all bindings of the given map. The returned list is sorted in increasing order with respect to the ordering Ord.compare, where Ord is the argument given to Map.Make\[20.18\].
Since: 3.12.0

val min_binding : 'a t -> key * 'a
Return the smallest binding of the given map (with respect to the Ord.compare ordering), or raise Not_found if the map is empty.
Since: 3.12.0

val max_binding : 'a t -> key * 'a
Same as Map.S.min_binding\[20.18\], but returns the largest binding of the given map.
Since: 3.12.0

val choose : 'a t -> key * 'a
Return one binding of the given map, or raise Not_found if the map is empty. Which binding is chosen is unspecified, but equal bindings will be chosen for equal maps.
Since: 3.12.0

val split : key -> 'a t -> 'a t * 'a option * 'a t
split x m returns a triple (l, data, r), where l is the map with all the bindings of m whose key is strictly less than x; r is the map with all the bindings of m whose key is strictly greater than x; data is None if m contains no binding for x, or Some v if m binds v to x.
Since: 3.12.0
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 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 bindings are passed to f in increasing order with respect to the ordering over the type of the keys.

val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t

Same as Map.S.map, but the function receives as arguments both the key and the associated value for each binding of the map.

end

Output signature of the functor Map.Make.

module Make :
  functor (Ord : OrderedType) -> S with type key = Ord.t
  Functor building an implementation of the map structure given a totally ordered type.

20.19 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.

```ocaml
val to_channel : Pervasives.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 : Pervasives.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

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.

val data_size : string -> int -> int

See Marshal.header_size.

val total_size : string -> int -> int

See Marshal.header_size.


This module provides operations on the type nativeint of signed 32-bit integers (on 32-bit platforms) or signed 64-bit integers (on 64-bit platforms). This integer type has exactly the same width as that of a long integer type in the C compiler. All arithmetic operations over nativeint are taken modulo \(2^{32}\) or \(2^{64}\) depending on the word size of the architecture.

Performance notice: values of type nativeint occupy more memory space than values of type int, and arithmetic operations on nativeint are generally slower than those on int. Use nativeint only when the application requires the extra bit of precision over the int type.

val zero : nativeint

The native integer 0.

val one : nativeint

The native integer 1.

val minus_one : nativeint

The native integer -1.
Chapter 20. The standard library

val neg : nativeint -> nativeint
  Unary negation.

val add : nativeint -> nativeint -> nativeint
  Addition.

val sub : nativeint -> nativeint -> nativeint
  Subtraction.

val mul : nativeint -> nativeint -> nativeint
  Multiplication.

val div : nativeint -> nativeint -> nativeint
  Integer division. Raise Division_by_zero if the second argument is zero. This division
  rounds the real quotient of its arguments towards zero, as specified for
  Pervasives.(/)[19.2].

val rem : nativeint -> nativeint -> nativeint
  Integer remainder. If y is not zero, the result of Nativeint.rem x y satisfies the following
  properties: Nativeint.zero <= Nativeint.rem x y < Nativeint.abs y and x =
  Nativeint.add (Nativeint.mul (Nativeint.div x y) y) (Nativeint.rem x y). If y
  = 0, Nativeint.rem x y raises Division_by_zero.

val succ : nativeint -> nativeint
  Successor. Nativeint.succ x is Nativeint.add x Nativeint.one.

val pred : nativeint -> nativeint
  Predecessor. Nativeint.pred x is Nativeint.sub x Nativeint.one.

val abs : nativeint -> nativeint
  Return the absolute value of its argument.

val size : int
  The size in bits of a native integer. This is equal to 32 on a 32-bit platform and to 64 on a
  64-bit platform.

val max_int : nativeint
  The greatest representable native integer, either $2^{31} - 1$ on a 32-bit platform, or $2^{63} - 1$ on a
  64-bit platform.

val min_int : nativeint
  The greatest representable native integer, either $-2^{31}$ on a 32-bit platform, or $-2^{63}$ on a
  64-bit platform.

val logand : nativeint -> nativeint -> nativeint
Bitwise logical and.

val logor : nativeint -> nativeint -> nativeint
  Bitwise logical or.

val logxor : nativeint -> nativeint -> nativeint
  Bitwise logical exclusive or.

val lognot : nativeint -> nativeint
  Bitwise logical negation

val shift_left : nativeint -> int -> nativeint
  Nativeint.shift_left x y shifts x to the left by y bits. The result is unspecified if y < 0
  or y >= bitsize, where bitsize is 32 on a 32-bit platform and 64 on a 64-bit platform.

val shift_right : nativeint -> int -> nativeint
  Nativeint.shift_right x y shifts x to the right by y bits. This is an arithmetic shift: the
  sign bit of x is replicated and inserted in the vacated bits. The result is unspecified if y < 0
  or y >= bitsize.

val shift_right_logical : nativeint -> int -> nativeint
  Nativeint.shift_right_logical x y shifts x to the right by y bits. This is a logical shift:
  zeroes are inserted in the vacated bits regardless of the sign of x. The result is unspecified if
  y < 0 or y >= bitsize.

val of_int : int -> nativeint
  Convert the given integer (type int) to a native integer (type nativeint).

val to_int : nativeint -> int
  Convert the given native integer (type nativeint) to an integer (type int). The high-order
  bit is lost during the conversion.

val of_float : float -> nativeint
  Convert the given floating-point number to a native integer, discarding the fractional part
  (truncate towards 0). The result of the conversion is undefined if, after truncation, the
  number is outside the range [Nativeint.min_int 20.20, Nativeint.max_int 20.20].

val to_float : nativeint -> float
  Convert the given native integer to a floating-point number.

val of_int32 : int32 -> nativeint
  Convert the given 32-bit integer (type int32) to a native integer.

val to_int32 : nativeint -> int32
Chapter 20. The standard library

Convert the given native integer to a 32-bit integer (type int32). On 64-bit platforms, the 64-bit native integer is taken modulo 2^{32}, i.e. the top 32 bits are lost. On 32-bit platforms, the conversion is exact.

val of_string : string -> nativeint
Convert the given string to a native 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, or if the integer represented exceeds the range of integers representable in type nativeint.

val to_string : nativeint -> string
Return the string representation of its argument, in decimal.

type t = nativeint
An alias for the type of native integers.

val compare : t -> t -> int
The comparison function for native integers, with the same specification as
Pervasives.compare\[19.2\]. Along with the type t, this function compare allows the module Nativeint to be passed as argument to the functors Set.Make\[20.28\] and Map.Make\[20.18\].

20.21 Module 0o : Operations on objects

val copy : (< .. > as 'a) -> 'a
0o.copy o returns a copy of object o, that is a fresh object with the same methods and instance variables as o

val id : < .. > -> int
Return an integer identifying this object, unique for the current execution of the program.

20.22 Module Parsing : The run-time library for parsers generated by ocamlyacc.

val symbol_start : unit -> int
symbol_start and Parsing.symbol_end\[20.22\] are to be called in the action part of a grammar rule only. They return the offset of the string that matches the left-hand side of the rule: symbol_start() returns the offset of the first character; symbol_end() returns the offset after the last character. The first character in a file is at offset 0.

val symbol_end : unit -> int
See Parsing.symbol_start[20.22].

val rhs_start : int -> int
  Same as Parsing.symbol_start[20.22] and Parsing.symbol_end[20.22], but return the offset of the string matching the nth item on the right-hand side of the rule, where n is the integer parameter to rhs_start and rhs_end. n is 1 for the leftmost item.

val rhs_end : int -> int
  See Parsing.rhs_start[20.22].

val symbol_start_pos : unit -> Lexing.position
  Same as symbol_start, but return a position instead of an offset.

val symbol_end_pos : unit -> Lexing.position
  Same as symbol_end, but return a position instead of an offset.

val rhs_start_pos : int -> Lexing.position
  Same as rhs_start, but return a position instead of an offset.

val rhs_end_pos : int -> Lexing.position
  Same as rhs_end, but return a position instead of an offset.

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.

val set_trace : bool -> bool
  Control debugging support for ocamlyacc-generated parsers. After Parsing.set_trace true, the pushdown automaton that executes the parsers prints a trace of its actions (reading a token, shifting a state, reducing by a rule) on standard output. Parsing.set_trace false turns this debugging trace off. The boolean returned is the previous state of the trace flag.

Since: 3.11.0
Module Printexc : Facilities for printing exceptions.

val to_string : exn -> string
  Printexc.to_string e returns a string representation of the exception e.

val print : ('a -> 'b) -> 'a -> 'b
  Printexc.print 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 exception is raised again. The typical use is to catch and report exceptions that escape a function application.

val catch : ('a -> 'b) -> 'a -> 'b
  Printexc.catch fn x is similar to Printexc.print, but aborts the program with exit code 2 after printing the uncaught exception. This function is deprecated: the runtime system is now able to print uncaught exceptions as precisely as Printexc.catch does. Moreover, calling Printexc.catch makes it harder to track the location of the exception using the debugger or the stack backtrace facility. So, do not use Printexc.catch in new code.

val print_backtrace : Pervasives.out_channel -> unit
  Printexc.print_backtrace oc prints an exception backtrace on the output channel oc. The backtrace lists the program locations where the most-recently raised exception was raised and where it was propagated through function calls.
  
  Since: 3.11.0

val get_backtrace : unit -> string
  Printexc.get_backtrace () returns a string containing the same exception backtrace that Printexc.print_backtrace would print.
  
  Since: 3.11.0

val record_backtrace : bool -> unit
  Printexc.record_backtrace b turns recording of exception backtraces on (if b = true) or off (if b = false). Initially, backtraces are not recorded, unless the b flag is given to the program through the OCAMLRUNPARAM variable.
  
  Since: 3.11.0

val backtrace_status : unit -> bool
  Printexc.backtrace_status() returns true if exception backtraces are currently recorded, false if not.
  
  Since: 3.11.0

val register_printer : (exn -> string option) -> unit
Printexc.register_printer fn registers fn as an exception printer. The printer should return None or raise an exception if it does not know how to convert the passed exception, and Some s with s the resulting string if it can convert the passed exception. Exceptions raised by the printer are ignored.

When converting an exception into a string, the printers will be invoked in the reverse order of their registrations, until a printer returns a Some s value (if no such printer exists, the runtime will use a generic printer).

Since: 3.11.2

20.24 Module Printf: Formatted output functions.

val fprintf :
  Pervasives.out_channel ->
  ('a, Pervasives.out_channel, unit) Pervasives.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 arguments.

Conversion specifications have the following form:
% [flags] [width] [.precision] type

In short, a conversion specification consists in the % character, followed by optional modifiers and a type which is made of one or two characters. The types and their meanings are:

- d, i, n, l, L, or N: 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.
- o: convert an integer argument to unsigned octal.
- s: insert a string argument.
- S: insert a string argument in Caml syntax (double quotes, escapes).
- c: insert a character argument.
- C: insert a character argument in Caml syntax (single quotes, escapes).
- f: convert a floating-point argument to decimal notation, in the style dddd.ddd.
- F: convert a floating-point argument to Caml syntax (dddd. or dddd.ddd or d.ddd e+-dd).
- 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
• b: convert a boolean argument (for backward compatibility; do not use in new
  programs).
• ld, li, lu, lx, lo: convert an int32 argument to the format specified by the
  second letter (decimal, hexadecimal, etc).
• nd, ni, nu, nx, nX, no: convert a nativeint argument to the format specified by the
  second letter.
• Ld, Li, Lu, Lx, LX, Lo: convert an int64 argument to the format specified by the
  second letter.
• a: user-defined printer. Takes two arguments and applies 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 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.
• { fmt %}: convert a format string argument. The argument must have the same type
  as the internal format string fmt.
• ( fmt %): format string substitution. Takes a format string argument and substitutes
  it to the internal format string fmt to print following arguments. The argument must
  have the same type as the internal format string fmt.
• !: take no argument and flush the output.
• %: take no argument and output one % character.
• ,: the no-op delimiter for conversion specifications.

The optional flags are:
• -: left-justify the output (default is right justification).
• 0: for numerical conversions, pad with zeroes instead of spaces.
• +: for numerical conversions, prefix number with a + sign if positive.
• space: for numerical conversions, prefix number with a space if positive.
• #: request an alternate formatting style for numbers.

The optional width is an integer indicating the minimal width of the result. For instance,
%6d prints an integer, prefixing it with spaces to fill at least 6 characters.

The optional precision is a dot . followed by an integer indicating how many digits follow
the decimal point in the %f, %e, and %E conversions. For instance, %.4f prints a float with
4 fractional digits.

The integer in a width or precision can also be specified as *, in which case an extra
integer argument is taken to specify the corresponding width or precision. This integer
argument precedes immediately the argument to print. For instance, %.4f prints a float
with as many fractional digits as the value of the argument given before the float.
val printf : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
  Same as Printf.fprintf, but output on stdout.

val eprintf : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
  Same as Printf.fprintf, but output on stderr.

val ifprintf : 'a -> ('b, 'a, unit) Pervasives.format -> 'b
  Same as Printf.fprintf, but does not print anything. Useful to ignore some
  material when conditionally printing.
  Since: 3.10.0

val sprintf : ('a, unit, string) Pervasives.format -> 'a
  Same as Printf.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) Pervasives.format -> 'a
  Same as Printf.fprintf, but instead of printing on an output channel, append the
  formatted arguments to the given extensible buffer (see module Buffer).

Formatted output functions with continuations.

val kfprintf :
  (Pervasives.out_channel -> 'a) ->
  Pervasives.out_channel ->
  ('b, Pervasives.out_channel, unit, 'a) Pervasives.format4 -> 'b
  Same as fprintf, but instead of returning immediately, passes the out channel to its first
  argument at the end of printing.
  Since: 3.09.0

val ksprintf :
  (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b
  Same as sprintf above, but instead of returning the string, passes it to the first argument.
  Since: 3.09.0

val kbprintf :
  (Buffer.t -> 'a) ->
  Buffer.t -> ('b, Buffer.t, unit, 'a) Pervasives.format4 -> 'b
  Same as bprintf, but instead of returning immediately, passes the buffer to its first
  argument at the end of printing.
  Since: 3.10.0

  Deprecated

val kprintf :
  (string -> 'a) -> ('b, unit, string, 'a) Pervasives.format4 -> 'b
  A deprecated synonym for ksprintf.
20.25 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 Queue.take or Queue.peek 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 push : 'a -> 'a t -> unit
   push is a synonym for add.

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 pop : 'a t -> 'a
   pop is a synonym for take.

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 top : 'a t -> 'a
   top is a synonym for peek.

val clear : 'a t -> unit
   Discard all elements from a queue.

val copy : 'a t -> 'a t
   Return a copy of the given queue.

val is_empty : 'a t -> bool
   Return true if the given queue is empty, false otherwise.

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.

val fold : ('a -> 'b -> 'a) -> 'a -> 'b t -> 'a

fold f accu q is equivalent to List.fold_left f accu l, where l is the list of q’s elements. The queue remains unchanged.

val transfer : 'a t -> 'a t -> unit

transfer q1 q2 adds all of q1’s elements at the end of the queue q2, then clears q1. It is equivalent to the sequence iter (fun x -> add x q2) q1; clear q1, but runs in constant time.


Basic functions

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 Random.init but takes more data as seed.

val self_init : unit -> unit

Initialize the generator with a more-or-less random seed chosen in a system-dependent way.

val bits : unit -> int

Return 30 random bits in a nonnegative integer.

Before 3.12.0 used a different algorithm (affects all the following functions)

val int : int -> int

Random.int bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0 and less than $2^{30}$.

val int32 : Int32.t -> Int32.t

Random.int32 bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0.

val nativeint : Nativeint.t -> Nativeint.t
Random.nativeint bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0.

val int64 : Int64.t -> Int64.t

Random.int64 bound returns a random integer between 0 (inclusive) and bound (exclusive). bound must be greater than 0.

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 or zero. If bound is 0, the result is 0.

val bool : unit -> bool

Random.bool () returns true or false with probability 0.5 each.

Advanced functions

The functions from module State manipulate the current state of the random generator explicitly. This allows using one or several deterministic PRNGs, even in a multi-threaded program, without interference from other parts of the program.

module State :
  sig

    type t

    The type of PRNG states.

    val make : int array -> t

    Create a new state and initialize it with the given seed.

    val make_self_init : unit -> t

    Create a new state and initialize it with a system-dependent low-entropy seed.

    val copy : t -> t

    Return a copy of the given state.

    val bits : t -> int
    val int : t -> int -> int
    val int32 : t -> Int32.t -> Int32.t
    val nativeint : t -> Nativeint.t -> Nativeint.t
    val int64 : t -> Int64.t -> Int64.t
    val float : t -> float -> float
    val bool : t -> bool
These functions are the same as the basic functions, except that they use (and update) the given PRNG state instead of the default one.

val get_state : unit -> State.t
    Return the current state of the generator used by the basic functions.

val set_state : State.t -> unit
    Set the state of the generator used by the basic functions.

20.27 Module Scanf : Formatted input functions.

Introduction

Functional input with format strings

The module Scanf provides formatted input functions or scanners.

The formatted input functions can read from any kind of input, including strings, files, or anything that can return characters. The more general source of characters is named a formatted input channel (or scanning buffer) and has type Scanf.Scanning.in_channel[20.27]. The more general formatted input function reads from any scanning buffer and is named bscanf.

Generally speaking, the formatted input functions have 3 arguments:

- the first argument is a source of characters for the input,
- the second argument is a format string that specifies the values to read,
- the third argument is a receiver function that is applied to the values read.

Hence, a typical call to the formatted input function Scanf.bscanf[20.27] is bscanf ic fmt f, where:

- ic is a source of characters (typically a formatted input channel with type Scanf.Scanning.in_channel[20.27]),
- fmt is a format string (the same format strings as those used to print material with module Printf[20.24] or Format[20.9]),
- f is a function that has as many arguments as the number of values to read in the input.
A simple example

As suggested above, the expression `bscanf ic "%d" f` reads a decimal integer `n` from the source of characters `ic` and returns `f n`.

For instance,

- if we use `stdin` as the source of characters (`Scanf.Scanning.stdin` is the predefined formatted input channel that reads from standard input),

- if we define the receiver `f` as `let f x = x + 1`,

  then `bscanf Scanning.stdin "%d" f` reads an integer `n` from the standard input and returns `f n` (that is `n + 1`). Thus, if we evaluate `bscanf stdin "%d" f`, and then enter `41` at the keyboard, we get `42` as the final result.

Formatted input as a functional feature

The Caml scanning facility is reminiscent of the corresponding C feature. However, it is also largely different, simpler, and yet more powerful: the formatted input functions are higher-order functionals and the parameter passing mechanism is just the regular function application not the variable assignment based mechanism which is typical for formatted input in imperative languages; the Caml format strings also feature useful additions to easily define complex tokens; as expected within a functional programming language, the formatted input functions also support polymorphism, in particular arbitrary interaction with polymorphic user-defined scanners. Furthermore, the Caml formatted input facility is fully type-checked at compile time.

```ocaml
module Scanning :
  sig
    type in_channel
    type scanbuf = in_channel

    The type of scanning buffers. A scanning buffer is the source from which a formatted
    input function gets characters. The scanning buffer holds the current state of the scan,
    plus a function to get the next char from the input, and a token buffer to store the
    string matched so far.

    Note: a scanning action may often require to examine one character in advance; when
    this “lookahead” character does not belong to the token read, it is stored back in the
    scanning buffer and becomes the next character yet to be read.

    val stdin : in_channel

    The standard input notion for the Scanf module. Scanning.stdin is the formatted
    input channel attached to Pervasives.stdin.

    Note: in the interactive system, when input is read from `stdin`, the newline character
    that triggers the evaluation is incorporated in the input; thus, the scanning
    specifications must properly skip this additional newline character (for instance, simply
    add a `\n` as the last character of the format string).

    Since: 3.12.0
```
val open_in : string -> in_channel

Scanning.open_in fname returns a formatted input channel for bufferized reading in text mode of file fname.
Note: open_in returns a formatted input channel that efficiently reads characters in large chunks; in contrast, from_channel below returns formatted input channels that must read one character at a time, leading to a much slower scanning rate.
Since: 3.12.0

val open_in_bin : string -> in_channel

Scanning.open_in_bin fname returns a formatted input channel for bufferized reading in binary mode of file fname.
Since: 3.12.0

val close_in : in_channel -> unit

Closes the Pervasives.input_channel associated with the given Scanning.in_channel formatted input channel.
Since: 3.12.0

val from_file : string -> in_channel

An alias for open_in above.

val from_file_bin : string -> in_channel

An alias for open_in_bin above.

val from_string : string -> in_channel

Scanning.from_string s returns a formatted input channel which reads from the given string. Reading starts from the first character in the string. The end-of-input condition is set when the end of the string is reached.

val from_function : (unit -> char) -> in_channel

Scanning.from_function f returns a formatted input channel with the given function as its reading method.
When scanning needs one more character, the given function is called.
When the function has no more character to provide, it must signal an end-of-input condition by raising the exception End_of_file.

val from_channel : Pervasives.in_channel -> in_channel

Scanning.from_channel ic returns a formatted input channel which reads from the regular input channel ic argument, starting at the current reading position.

val end_of_input : in_channel -> bool
**Chapter 20. The standard library**

`Scanning.end_of_input ic` tests the end-of-input condition of the given formatted input channel.

```ml
val beginning_of_input : in_channel -> bool
```

`Scanning.beginning_of_input ic` tests the beginning of input condition of the given formatted input channel.

```ml
val name_of_input : in_channel -> string
```

`Scanning.name_of_input ic` returns the name of the character source for the formatted input channel `ic`.

Since: 3.09.0

```ml
val stdib : in_channel
```

A deprecated alias for `Scanning.stdin`, the scanning buffer reading from `Pervasives.stdin`.

```
end
```

**Formatted input channel**

**Type of formatted input functions**

```ml
type ('a, 'b, 'c, 'd) scanner = ('a, Scanning.in_channel, 'b, 'c, 'a -> 'd, 'd) format6 -> 'c
```

The type of formatted input scanners: `('a, 'b, 'c, 'd) scanner` is the type of a formatted input function that reads from some formatted input channel according to some format string; more precisely, if `scan` is some formatted input function, then `scan ic fmt f` applies `f` to the arguments specified by the format string `fmt`, when `scan` has read those arguments from the formatted input channel `ic`.

For instance, the `scanf` function below has type `('a, 'b, 'c, 'd) scanner`, since it is a formatted input function that reads from `Scanning.stdin`: `scanf fmt f` applies `f` to the arguments specified by `fmt`, reading those arguments from `Pervasives.stdin` as expected.

If the format `fmt` has some `%r` indications, the corresponding input functions must be provided before the receiver `f` argument. For instance, if `read_elem` is an input function for values of type `t`, then `bscanf ic "%r;" read_elem f` reads a value `v` of type `t` followed by a `';'` character, and returns `f v`.

Since: 3.10.0

```ml
exception Scan_failure of string
```

The exception that formatted input functions raise when the input cannot be read according to the given format.
The general formatted input function

```plaintext
val bscanf : Scanning.in_channel -> ('a, 'b, 'c, 'd) scanner

bscanf ic fmt r1 ... rN f reads arguments for the function f, from the formatted input channel ic, according to the format string fmt, and applies f to these values. The result of this call to f is returned as the result of the entire bscanf call. For instance, if f is the function fun s i -> i + 1, then Scanf.sscanf "x= 1" "%s = %i" f returns 2.

Arguments r1 to rN are user-defined input functions that read the argument corresponding to a %r conversion.
```

Format string description

The format is a character string which contains three types of objects:

- plain characters, which are simply matched with the characters of the input (with a special case for space and line feed, see [20.27]),
- conversion specifications, each of which causes reading and conversion of one argument for the function f (see [20.27]),
- scanning indications to specify boundaries of tokens (see scanning [20.27]).

The space character in format strings

As mentioned above, a plain character in the format string is just matched with the next character of the input; however, two characters are special exceptions to this rule: the space character (’ ’ or ASCII code 32) and the line feed character (’\n’ or ASCII code 10). A space does not match a single space character, but any amount of “whitespace” in the input. More precisely, a space inside the format string matches any number of tab, space, line feed and carriage return characters. Similarly, a line feed character in the format string matches either a single line feed or a carriage return followed by a line feed.

Matching any amount of whitespace, a space in the format string also matches no amount of whitespace at all; hence, the call bscanf ib "Price = \%d $" (fun p -> p) succeeds and returns 1 when reading an input with various whitespace in it, such as Price = 1 $, Price = 1 $, or even Price=1$.

Conversion specifications in format strings

Conversion specifications consist in the % character, followed by an optional flag, an optional field width, and followed by one or two conversion characters. The conversion characters and their meanings are:

- d: reads an optionally signed decimal integer.
- i: reads an optionally signed integer (usual input conventions for decimal (0-9+), hexadecimal (0x[0-9a-f]+ and 0X[0-9A-F]+), octal (0o[0-7]+), and binary (0b[0-1]+) notations are understood).
Chapter 20. The standard library

- **u**: reads an unsigned decimal integer.
- **x** or **X**: reads an unsigned hexadecimal integer ([0-9a-f]+ or [0-9A-F]+).
- **o**: reads an unsigned octal integer ([0-7]+).
- **s**: reads a string argument that spreads as much as possible, until the following bounding condition holds:
  - a whitespace has been found (see [20.27]),
  - a scanning indication (see scanning [20.27]) has been encountered,
  - the end-of-input has been reached.
  Hence, this conversion always succeeds: it returns an empty string, if the bounding condition holds when the scan begins.
- **S**: reads a delimited string argument (delimiters and special escaped characters follow the lexical conventions of Caml).
- **c**: reads a single character. To test the current input character without reading it, specify a null field width, i.e. use specification `%0c`. Raise **Invalid_argument**, if the field width specification is greater than 1.
- **C**: reads a single delimited character (delimiters and special escaped characters follow the lexical conventions of Caml).
- **f**, **e**, **E**, **g**, **G**: reads an optionally signed floating-point number in decimal notation, in the style `dddd.ddd e/E+-dd`.
- **F**: reads a floating point number according to the lexical conventions of Caml (hence the decimal point is mandatory if the exponent part is not mentioned).
- **B**: reads a boolean argument (**true** or **false**).
- **b**: reads a boolean argument (for backward compatibility; do not use in new programs).
- **ld**, **li**, **lu**, **lx**, **lo**: reads an **int32** argument to the format specified by the second letter for regular integers.
- **nd**, **ni**, **nu**, **nx**, **no**: reads a **nativeint** argument to the format specified by the second letter for regular integers.
- **Ld**, **Li**, **Lu**, **Lx**, **Lo**: reads an **int64** argument to the format specified by the second letter for regular integers.
- `[ range ]`: reads characters that matches one of the characters mentioned in the range of characters **range** (or not mentioned in it, if the range starts with `^`). Reads a **string** that can be empty, if the next input character does not match the range. The set of characters from **c1** to **c2** (inclusively) is denoted by **c1-c2**. Hence, `%[0-9]` returns a string representing a decimal number or an empty string if no decimal digit is found; similarly,
%r: user-defined reader. Takes the next ri formatted input function and applies it to the scanning buffer ib to read the next argument. The input function ri must therefore have type Scanning.in_channel -> 'a and the argument read has type 'a.

• { fmt %}: reads a format string argument. The format string read must have the same type as the format string specification fmt. For instance, "%{ %i %}" reads any format string that can read a value of type int; hence, if s is the string "fmt:\"number is %u\"", then Scanf.sscanf s "fmt: %(%i%)" succeeds and returns the format string "number is %u".

• \( ( fmt %) \): scanning format substitution. Reads a format string and then goes on scanning with the format string read, instead of using fmt. The format string read must have the same type as the format string specification fmt that it replaces. For instance, "%( %i %)" reads any format string that can read a value of type int. Returns the format string read, and the value read using the format string read. Hence, if f is the function fun i -> fmt, i) evaluates to "8( %4d", 1234). If the special flag _ is used, the conversion discards the format string read and only returns the value read with the format string read. Hence, if s is the string "\"%4d\"1234.00", then Scanf.sscanf s "%(%i%)" is simply equivalent to Scanf.sscanf "1234.00" "%4d".

• l: returns the number of lines read so far.

• n: returns the number of characters read so far.

• N or L: returns the number of tokens read so far.

• !: matches the end of input condition.

• %: matches one % character in the input.

• ,: the no-op delimiter for conversion specifications.

Following the % character that introduces a conversion, there may be the special flag _: the conversion that follows occurs as usual, but the resulting value is discarded. For instance, if f is the function fun i -> i + 1, and s is the string "$x = 1", then Scanf.sscanf s "%_%s = %i" f returns 2.

The field width is composed of an optional integer literal indicating the maximal width of the token to read. For instance, %6d reads an integer, having at most 6 decimal digits; %4f reads a float with at most 4 characters; and %8[\000-\0255] returns the next 8 characters (or all the characters still available, if fewer than 8 characters are available in the input).

Notes:

• as mentioned above, a %s conversion always succeeds, even if there is nothing to read in the input: in this case, it simply returns "".
• in addition to the relevant digits, ‘.’ characters may appear inside numbers (this is reminiscent to the usual Caml lexical conventions). If stricter scanning is desired, use the range conversion facility instead of the number conversions.

• the scanf facility is not intended for heavy duty lexical analysis and parsing. If it appears not expressive enough for your needs, several alternative exists: regular expressions (module Str), stream parsers, ocamllex-generated lexers, ocamlyacc-generated parsers.

Scanning indications in format strings
Scanning indications appear just after the string conversions %s and %[ range ] to delimit the end of the token. A scanning indication is introduced by a @ character, followed by some constant character c. It means that the string token should end just before the next matching c (which is skipped). If no c character is encountered, the string token spreads as much as possible. For instance, "%s@\t" reads a string up to the next tab character or to the end of input. If a scanning indication @c does not follow a string conversion, it is treated as a plain c character.

Note:

• the scanning indications introduce slight differences in the syntax of Scanf format strings, compared to those used for the Printf module. However, the scanning indications are similar to those used in the Format module; hence, when producing formatted text to be scanned by !Scanf.bscanf, it is wise to use printing functions from the Format module (or, if you need to use functions from Printf, banish or carefully double check the format strings that contain ’@’ characters).

Exceptions during scanning
Scanners may raise the following exceptions when the input cannot be read according to the format string:

• Raise Scanf.Scan_failure if the input does not match the format.

• Raise Failure if a conversion to a number is not possible.

• Raise End_of_file if the end of input is encountered while some more characters are needed to read the current conversion specification.

• Raise Invalid_argument if the format string is invalid.

Note:

• as a consequence, scanning a %s conversion never raises exception End_of_file: if the end of input is reached the conversion succeeds and simply returns the characters read so far, or "" if none were ever read.
Specialised formatted input functions

val fscanf : Pervasives.in_channel -> ('a, 'b, 'c, 'd) scanner

Same as Scanf.bscanf[20.27], but reads from the given regular input channel.

Warning: since all formatted input functions operate from a formatted input channel, be aware that each fscanf invocation will operate with a formatted input channel reading from the given channel. This extra level of bufferization can lead to a strange scanning behaviour if you use low level primitives on the channel (reading characters, seeking the reading position, and so on).

As a consequence, never mix direct low level reading and high level scanning from the same regular input channel.

val sscanf : string -> ('a, 'b, 'c, 'd) scanner

Same as Scanf.bscanf[20.27], but reads from the given string.

val scanf : ('a, 'b, 'c, 'd) scanner

Same as Scanf.bscanf[20.27], but reads from the predefined formatted input channel Scanf.Scanning.stdin[20.27] that is connected to Pervasives.stdin.

val kscanf :
   Scanning.in_channel ->
   (Scanning.in_channel -> exn -> 'a) -> ('b, 'c, 'd, 'a) scanner

Same as Scanf.bscanf[20.27], but takes an additional function argument ef that is called in case of error: if the scanning process or some conversion fails, the scanning function aborts and calls the error handling function ef with the formatted input channel and the exception that aborted the scanning process as arguments.

Reading format strings from input

val bscanf_format :
   Scanning.in_channel ->
   ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
   ('a, 'b, 'c, 'd, 'e, 'f) format6 -> 'g) -> 'g

bscanf_format ic fmt f reads a format string token from the formatted input channel ic, according to the given format string fmt, and applies f to the resulting format string value. Raise Scan_failure if the format string value read does not have the same type as fmt.

Since: 3.09.0

val sscanf_format :
   string ->
   ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
   ('a, 'b, 'c, 'd, 'e, 'f) format6 -> 'g) -> 'g
Same as \texttt{Scanf.bscanf\_format}\textsuperscript{20.27}, but reads from the given string.

\textbf{Since:} 3.09.0

\texttt{val format\_from\_string : string \rightarrow} \hfill \\
\texttt{('a', 'b', 'c', 'd', 'e', 'f) format6 \rightarrow ('a', 'b', 'c', 'd', 'e', 'f) format6}

\texttt{format\_from\_string s fmt} converts a string argument to a format string, according to the given format string \texttt{fmt}. Raise \texttt{Scan\_failure} if \texttt{s}, considered as a format string, does not have the same type as \texttt{fmt}.

\textbf{Since:} 3.10.0

\section{20.28 Module \texttt{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.

\begin{verbatim}
module type OrderedType =
  sig
    type t
    The type of the set elements.
    val compare : t \rightarrow t \rightarrow int
    A total ordering function over the set elements. This is a two-argument function \texttt{f} such that \texttt{f e1 e2} is zero if the elements \texttt{e1} and \texttt{e2} are equal, \texttt{f e1 e2} is strictly negative if \texttt{e1} is smaller than \texttt{e2}, and \texttt{f e1 e2} is strictly positive if \texttt{e1} is greater than \texttt{e2}. Example: a suitable ordering function is the generic structural comparison function \texttt{Pervasives.compare}\textsuperscript{19.2}.
  end

Input signature of the functor \texttt{Set.Make}\textsuperscript{20.28}.

module type S =
  sig
    type elt
    The type of the set elements.
    type t
    The type of sets.
  end
\end{verbatim}
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

    Set union.

val inter : t -> t -> t

    Set intersection.

val diff : t -> t -> t

    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 elements of s are presented to f in increasing order with respect to the ordering over the type of the elements.

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, in increasing order.

val for_all : (elt -> bool) -> t -> bool

for_all p s checks if all elements of the set satisfy the predicate p.

val exists : (elt -> bool) -> t -> bool

exists p s checks if at least one element of the set satisfies the predicate p.

val filter : (elt -> bool) -> t -> t

filter p s returns the set of all elements in s that satisfy predicate p.

val partition : (elt -> bool) -> t -> t * t

partition p s returns a pair of sets (s1, s2), where s1 is the set of all the elements of s that satisfy the predicate p, and s2 is the set of all the elements of s that do not satisfy p.

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 [20.28].

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 Set.S.min_elt [20.28], 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.

val split : elt -> t -> t * bool * t
split x s returns a triple (l, present, r), where l is the set of elements of s that are strictly less than x; r is the set of elements of s that are strictly greater than x; present is false if s contains no element equal to x, or true if s contains an element equal to x.

end

Output signature of the functor Set.Make[20.28].

module Make :
  functor (Ord : OrderedType) -> S with type elt = Ord.t
  Functor building an implementation of the set structure given a totally ordered type.

20.29 Module Sort : Sorting and merging lists.

This module is obsolete and exists only for backward compatibility. The sorting functions in Array[20.2] and List[20.17] should be used instead. The new functions are faster and use less memory.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.

20.30 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 Stack.pop[20.30] or Stack.top[20.30] is applied to an empty stack.
Chapter 20. The standard library

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 top : 'a t -> 'a
  top s 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 copy : 'a t -> 'a t
  Return a copy of the given stack.

val is_empty : 'a t -> bool
  Return true if the given stack is empty, false otherwise.

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.

20.31 Module StdLabels: Standard labeled libraries.

This meta-module provides labelized version of the Array\[20.2\], List\[20.17\] and String\[20.33\] modules.

They only differ by their labels. Detailed interfaces can be found in arrayLabels.mli, listLabels.mli and stringLabels.mli.

module Array :
  sig
    val length : 'a array -> int
    val get : 'a array -> int -> 'a
    val set : 'a array -> int -> 'a -> unit
    val make : int -> 'a array
  end
val create : int -> 'a -> 'a array
val init : int -> f:(int -> 'a) -> 'a array
val make_matrix : dimx:int -> dimy:int -> 'a -> 'a array array
val create_matrix : dimx:int -> dimy:int -> 'a -> 'a array array
val append : 'a array -> 'a array -> 'a array
val concat : 'a array list -> 'a array
val sub : 'a array -> pos:int -> len:int -> 'a array
val copy : 'a array -> 'a array
val fill : 'a array -> pos:int -> len:int -> 'a -> unit
val blit :
  src:'a array -> src_pos:int -> dst:'a array -> dst_pos:int -> len:int -> unit
val to_list : 'a array -> 'a list
val of_list : 'a list -> 'a array
val iter : f:('a -> unit) -> 'a array -> unit
val map : f:('a -> 'b) -> 'a array -> 'b array
val iteri : f:(int -> 'a -> unit) -> 'a array -> unit
val mapi : f:(int -> 'a -> 'b) -> 'a array -> 'b array
val fold_left : f:('a -> 'b -> 'a) -> init:'a -> 'b array -> 'a
val fold_right : f:('a -> 'b -> 'b) -> 'a array -> init:'b -> 'b
val sort : cmp:('a -> 'a -> int) -> 'a array -> unit
val stable_sort : cmp:('a -> 'a -> int) -> 'a array -> unit
val fast_sort : cmp:('a -> 'a -> int) -> 'a array -> unit
val unsafe_get : 'a array -> int -> 'a
val unsafe_set : 'a array -> int -> 'a -> unit

end

module List :
sig

  val length : 'a list -> int
  val hd : 'a list -> 'a
  val tl : 'a list -> 'a list
  val nth : 'a list -> int -> 'a
  val rev : 'a list -> 'a list
  val append : 'a list -> 'a list -> 'a list
  val rev_append : 'a list -> 'a list -> 'a list
  val concat : 'a list list -> 'a list
  val flatten : 'a list list -> 'a list
  val iter : f:('a -> unit) -> 'a list -> unit

end
Chapter 20. The standard library

val map : f:('a -> 'b) -> 'a list -> 'b list
val rev_map : f:('a -> 'b) -> 'a list -> 'b list
val fold_left : f:('a -> 'b -> 'a) -> init:'a -> 'b list -> 'a
val fold_right : f:('a -> 'b -> 'b) -> 'a list -> init:'b -> 'b
val iter2 : f:('a -> 'b -> unit) -> 'a list -> 'b list -> unit
val map2 : f:('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val rev_map2 : f:('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
val fold_left2 :
  f:('a -> 'b -> 'c -> 'a) -> init:'a -> 'b list -> 'c list -> 'a
val fold_right2 :
  f:('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> init:'c -> 'c
val for_all : f:('a -> bool) -> 'a list -> bool
val exists : f:('a -> bool) -> 'a list -> bool
val for_all2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val exists2 : f:('a -> 'b -> bool) -> 'a list -> 'b list -> bool
val mem : 'a -> set:'a list -> bool
val memq : 'a -> set:'a list -> bool
val find : f:('a -> bool) -> 'a list -> 'a
val filter : f:('a -> bool) -> 'a list -> 'a list
val find_all : f:('a -> bool) -> 'a list -> 'a list
val partition : f:('a -> bool) -> 'a list -> 'a list * 'a list
val assoc : 'a -> ('a * 'b) list -> 'b
val assq : 'a -> ('a * 'b) list -> 'b
val mem_assoc : 'a -> map:('a * 'b) list -> bool
val mem_assq : 'a -> map:('a * 'b) list -> bool
val remove_assoc : 'a -> ('a * 'b) list -> ('a * 'b) list
val remove_assq : 'a -> ('a * 'b) list -> ('a * 'b) list
val split : ('a * 'b) list -> 'a list * 'b list
val combine : 'a list -> 'b list -> ('a * 'b) list
val sort : cmp:('a -> 'a -> int) -> 'a list -> 'a list
val stable_sort : cmp:('a -> 'a -> int) -> 'a list -> 'a list
val fast_sort : cmp:('a -> 'a -> int) -> 'a list -> 'a list
val merge : cmp:('a -> 'a -> int) -> 'a list -> 'a list -> 'a list

end

module String :
  sig
val length : string -> int
val get : string -> int -> char
val set : string -> int -> char -> unit
val create : int -> string
val make : int -> char -> string
val copy : string -> string
val sub : string -> pos:int -> len:int -> string
val fill : string -> pos:int -> len:int -> char -> unit
val blit :
  src:string -> src_pos:int -> dst:string -> dst_pos:int -> len:int -> unit
val concat : sep:string -> string list -> string
val iter : f:(char -> unit) -> string -> unit
val escaped : string -> string
val index : string -> char -> int
val rindex : string -> char -> int
val index_from : string -> int -> char -> int
val rindex_from : string -> int -> char -> int
val contains : string -> char -> bool
val contains_from : string -> int -> char -> bool
val rcontains_from : string -> int -> char -> bool
val uppercase : string -> string
val lowercase : string -> string
val capitalize : string -> string
val uncapitalize : string -> string

type t = string
val compare : t -> t -> int
val unsafe_get : string -> int -> char
val unsafe_set : string -> int -> char -> unit
val unsafe_blit :
  src:string -> src_pos:int -> dst:string -> dst_pos:int -> len:int -> unit
val unsafe_fill : string -> pos:int -> len:int -> char -> unit

end

20.32 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 : Pervasives.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
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.

20.33 Module String : String operations.

Given a string s of length l, we call character number in s the index of a character in s. Indexes start at 0, and we will call a character number valid in s if it falls within the range [0...l-1]. A position is the point between two characters or at the beginning or end of the string. We call a position valid in s if it falls within the range [0...l]. Note that character number n is between positions n and n+1.

Two parameters start and len are said to designate a valid substring of s if len >= 0 and start and start+len are valid positions in s.

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. You can also write s.[n] instead of String.get s n.
Raise Invalid_argument if n not a valid character number in s.

val set : string -> int -> char -> unit
String.set s n c modifies string s in place, replacing the character number n by c. You can also write s.[n] <- c instead of String.set s n c.
Raise Invalid_argument if n is not a valid character number in s.

val create : int -> string
String.create n returns a fresh string of length n. The string initially contains arbitrary characters.
Raise Invalid_argument if n < 0 or n > Sys.max_string_length[20.34].
Chapter 20. The standard library

val make : int -> char -> string

String.make n c returns a fresh string of length n, filled with the character c.

Raise Invalid_argument if n < 0 or n > Sys.max_string_length 20.34.

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 substring of s that starts at position start and has length len.

Raise Invalid_argument if start and len do not designate a valid substring of s.

val fill : string -> int -> int -> char -> unit

String.fill s start len c modifies string s in place, replacing len characters by c, starting at start.

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 intervals 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 concatenates the list of strings sl, inserting the separator string sep between each.

val iter : (char -> unit) -> string -> unit

String.iter f s applies function f in turn to all the characters of s. It is equivalent to f s.[0]; f s.[1]; ...; f s.[String.length s - 1]; ().

val escaped : string -> string

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Objective Caml. If there is no special character in the argument, return the original string itself, not a copy.

val index : string -> char -> int

String.index s c returns the character number of the first 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 character number of the last occurrence of character c in string s.
Raise Not_found if c does not occur in s.

val index_from : string -> int -> char -> int
String.index_from s i c returns the character number of the first occurrence of character c in string s after position i. String.index s c is equivalent to String.index_from s 0 c.
Raise Invalid_argument if i is not a valid position in s. Raise Not_found if c does not occur in s after position i.

val rindex_from : string -> int -> char -> int
String.rindex_from s i c returns the character number of the last occurrence of character c in string s before position i+1. String.rindex s c is equivalent to String.rindex_from s (String.length s - 1) c.
Raise Invalid_argument if i+1 is not a valid position in s. Raise Not_found if c does not occur in s before position i+1.

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 s after position start. String.contains s c is equivalent to String.contains_from s 0 c.
Raise Invalid_argument if start is not a valid position in s.

val rcontains_from : string -> int -> char -> bool
String.rcontains_from s stop c tests if character c appears in s before position stop+1. Raise Invalid_argument if stop < 0 or stop+1 is not a valid position in 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 character set to uppercase.

val uncapitalize : string -> string
Chapter 20. The standard library

Return a copy of the argument, with the first character set to lowercase.

type t = string
An alias for the type of strings.

val compare : t -> t -> int
The comparison function for strings, with the same specification as
Pervasives.compare[19.2]. Along with the type t, this function compare allows the module
String to be passed as argument to the functors Set.Make[20.28] and Map.Make[20.18].

20.34 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 executable_name : string
The name of the file containing the executable currently running.

val file_exists : string -> bool
Test if a file with the given name exists.

val is_directory : string -> bool
Returns true if the given name refers to a directory, false if it refers to another kind of
file. Raise Sys_error if no file exists with the given name.

Since: 3.10.0

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. If there
is already another file under the new name, rename may replace it, or raise an exception,
depending on your operating system.

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 readdir : string -> string array
    Return the names of all files present in the given directory. Names denoting the current
directory and the parent directory ("." and ".." in Unix) are not returned. Each string in
the result is a file name rather than a complete path. There is no guarantee that the name
strings in the resulting array will appear in any specific order; they are not, in particular,
guaranteed to appear in alphabetical order.

val interactive : bool Pervasives.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" (for all Unix versions, including Linux and Mac OS X),
    - "Win32" (for MS-Windows, OCaml compiled with MSVC++ or Mingw),
    - "Cygwin" (for MS-Windows, OCaml compiled with Cygwin).

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 a normal array. The maximum length of a float array is
    max_array_length/2 on 32-bit machines and max_array_length on 64-bit machines.

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.

```ml
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. If the signal number is invalid (or not available on your system), an `Invalid_argument` exception is raised.

```ml
val set_signal : int -> signal_behavior -> unit
```

Same as `Sys.signal` but return value is ignored.

**Signal numbers for the standard POSIX signals.**

```ml
val sigabrt : int
val sigalrm : int
val sigfpe : int
val sighup : int
val sigill : int
val sigint : int
val sigkill : int
val sigpipe : int
val sigquit : int
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 sintstp : int
  Interactive stop

val sigttin : int
  Terminal read from background process

val sigttou : int
  Terminal write from background process

val sigvtaerm : int
  Timeout in virtual time

val sigprof : int
  Profiling interrupt

exception Break
  Exception raised on interactive interrupt if Sys.catch_break 20.34 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.

val ocaml_version : string
  ocaml_version is the version of Objective Caml. It is a string of the form "major.minor[.patchlevel][+additional-info]", where major, minor, and patchlevel are integers, and additional-info is an arbitrary string. The [.patchlevel] and [+additional-info] parts may be absent.
Chapter 20. The standard library

20.35 Module Weak: Arrays of weak pointers and hash tables of weak pointers.

Low-level functions

type 'a t

The type of arrays of weak pointers (weak arrays). A weak pointer is a value that the
garbage collector may erase whenever the value is not used any more (through normal
pointers) by the program. Note that finalisation functions are run after the weak pointers
are erased.

A weak pointer is said to be full if it points to a value, empty if the value was erased by the
GC.

Notes:

• Integers are not allocated and cannot be stored in weak arrays.

• Weak arrays cannot be marshaled using Pervasives.output_value[19.2] nor the

val create : int -> 'a t

Weak.create n returns a new weak array of length n. All the pointers are initially empty.
Raise Invalid_argument if n is negative or greater than Sys.max_array_length[20.34]-1.

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[20.35] 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 value) if
it is full. Raise Invalid_argument "Weak.get" if n is not in the range 0 to
Weak.length[20.35] a - 1.

val get_copy : 'a t -> int -> 'a option

Weak.get_copy ar n returns None if the n-th cell of ar is empty, Some x (where x is a
(shallow) copy of the value) if it is full. In addition to pitfalls with mutable values, the
interesting difference with get is that get_copy does not prevent the incremental GC from
erasing the value in its current cycle (get may delay the erasure to the next GC cycle).
Raise Invalid_argument "Weak.get" if n is not in the range 0 to Weak.length[20.35] a -
1.
val check : 'a t -> int -> bool

Weak.check ar n returns true if the nth 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 ar.

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.

Weak hash tables

A weak hash table is a hashed set of values. Each value may magically disappear from the set when it is not used by the rest of the program any more. This is normally used to share data structures without inducing memory leaks. Weak hash tables are defined on values from a Hashtbl.HashedType module; the equal relation and hash function are taken from that module. We will say that v is an instance of x if equal x v is true.

The equal relation must be able to work on a shallow copy of the values and give the same result as with the values themselves.

module type S =

sig

  type data

  The type of the elements stored in the table.

  type t

  The type of tables that contain elements of type data. Note that weak hash tables cannot be marshaled using Pervasives.output_value or the functions of the Marshal module.

  val create : int -> t

  create n creates a new empty weak hash table, of initial size n. The table will grow as needed.

  val clear : t -> unit

  Remove all elements from the table.

  val merge : t -> data -> data
merge t x returns an instance of x found in t if any, or else adds x to t and return x.

val add : t -> data -> unit
add t x adds x to t. If there is already an instance of x in t, it is unspecified which one will be returned by subsequent calls to find and merge.

val remove : t -> data -> unit
remove t x removes from t one instance of x. Does nothing if there is no instance of x in t.

val find : t -> data -> data
find t x returns an instance of x found in t. Raise Not_found if there is no such element.

val find_all : t -> data -> data list
find_all t x returns a list of all the instances of x found in t.

val mem : t -> data -> bool
mem t x returns true if there is at least one instance of x in t, false otherwise.

val iter : (data -> unit) -> t -> unit
iter f t calls f on each element of t, in some unspecified order. It is not specified what happens if f tries to change t itself.

val fold : (data -> 'a -> 'a) -> t -> 'a -> 'a
fold f t init computes (f d1 (... (f dN init))) where d1 ... dN are the elements of t in some unspecified order. It is not specified what happens if f tries to change t itself.

val count : t -> int
Count the number of elements in the table. count t gives the same result as fold (fun _ n -> n+1) t 0 but does not delay the deallocation of the dead elements.

val stats : t -> int * int * int * int * int * int
Return statistics on the table. The numbers are, in order: table length, number of entries, sum of bucket lengths, smallest bucket length, median bucket length, biggest bucket length.

The output signature of the functor Weak.Make

module Make :
functor (H : Hashtbl.HashedType) -> S with type data = H.t
Functor building an implementation of the weak hash table structure.
Chapter 21

The unix library: Unix system calls

The unix library makes many Unix system calls and system-related library functions available to OCaml 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 as follows:

```
ocamlc other options unix.cma other files
ocamlopt other options unix.cmxa other files
```

For interactive use of the unix library, do:

```
ocamlmktop -o mytop unix.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type 

```ocaml
#load "unix.cma";;
```

Windows:

A fairly complete emulation of the Unix system calls is provided in the Windows version of OCaml. The end of this chapter gives more information on the functions that are not supported under Windows.

21.1 Module Unix: Interface to the Unix system

Error report

```
type error =
 | E2BIG
     Argument list too long
 | EACCES
     Permission denied
```
| EAGAIN   | Resource temporarily unavailable; try again
| EBADF    | Bad file descriptor
| E_BUSY   | Resource unavailable
| E_CHILD  | No child process
| E_DEADLK | Resource deadlock would occur
| E_DOM    | Domain error for math functions, etc.
| E_EXIST  | File exists
| E_FAULT  | Bad address
| E_BIG    | File too large
| E_INTR   | Function interrupted by signal
| EINVAL   | Invalid argument
| E_IO     | Hardware I/O error
| E_ISDIR  | Is a directory
| E_FILE   | Too many open files by the process
| E_LINK   | Too many links
| E_NAME_LONG | Filename too long
| E_FILE   | 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               |
| 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           |
| EPROTO        | 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        |
| NOBUFS                | 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|
| EOVERFLOW             | File size or position not representable |
| EUNKNOWNERR of int   | Unknown error                   |

The type of error codes. Errors defined in the POSIX standard and additional errors from UNIX98 and BSD. All other errors are mapped to EUNKNOWNERR.
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
    The process terminated normally by exit; the argument is the return code.
  | WSIGNALED of int
    The process was killed by a signal; the argument is the signal number.
  | WSTOPPED of int
    The process was stopped by a signal; the argument is the signal number.

The termination status of a process. See module Sys for the definitions of the
standard signal numbers. Note that they are not the numbers used by the OS.

type wait_flag =
  | WNOHANG
    do not block if no child has died yet, but immediately return with a pid equal to 0.
  | WUNTRACED

report also the children that receive stop signals.

Flags for Unix.waitpid[21.1].

val execv : string -> string array -> 'a
  execv prog args execute the program in file prog, with the arguments args, and the
current process environment. These execv* functions never return; on success, the current
program is replaced by the new one; on failure, a Unix.Unix_error[21.1] exception is raised.

val execve : string -> string array -> string array -> 'a
  Same as Unix.execv[21.1], except that the third argument provides the environment to the
program executed.

val execvp : string -> string array -> 'a
  Same as Unix.execv[21.1], except that the program is searched in the path.

val execvpe : string -> string array -> string array -> 'a
  Same as Unix.execve[21.1], 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 Unix.wait[21.1], but waits for the child 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
    File descriptor for standard input.

val stdout : file_descr
    File descriptor for standard output.

val stderr : file_descr
    File descriptor for 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
    | O_NOCTTY
      Don’t make this dev a controlling tty
    | O_DSYNC
      Writes complete as ‘Synchronised I/O data integrity completion’
    | O_SYNC
      Writes complete as ‘Synchronised I/O file integrity completion’
    | O_RSYNC
Chapter 21. The unix library: Unix system calls

Reads complete as writes (depending on O_SYNC/O_DSYNC)

The flags to Unix.openfile

```ocaml
type file_perm = int

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.

```ocaml
val close : file_descr -> unit
```

Close a file descriptor.

```ocaml
val read : file_descr -> string -> int -> int -> int

val write : file_descr -> string -> int -> int -> int

val single_write : file_descr -> string -> int -> int -> int
```

write repeats the writing operation until all characters have been written or an error occurs.

```ocaml
val in_channel_of_descr : file_descr -> Pervasives.in_channel

val out_channel_of_descr : file_descr -> Pervasives.out_channel

val descr_of_in_channel : Pervasives.in_channel -> file_descr

val descr_of_out_channel : Pervasives.out_channel -> file_descr
```

Interfacing with the standard input/output library
Seeking and truncating

type seek_command =
  | SEEK_SET
    indicates positions relative to the beginning of the file
  | SEEK_CUR
    indicates positions relative to the current position
  | SEEK_END
    indicates positions relative to the end of the file

Positioning modes for Unix.lseek[21.1].

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 status

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

\texttt{st\_ino} : int ;
   Inode number

\texttt{st\_kind} : \text{file\_kind} ;
   Kind of the file

\texttt{st\_perm} : \text{file\_perm} ;
   Access rights

\texttt{st\_nlink} : int ;
   Number of links

\texttt{st\_uid} : int ;
   User id of the owner

\texttt{st\_gid} : int ;
   Group ID of the file's group

\texttt{st\_rdev} : int ;
   Device minor number

\texttt{st\_size} : int ;
   Size in bytes

\texttt{st\_atime} : float ;
   Last access time

\texttt{st\_mtime} : float ;
   Last modification time

\texttt{st\_ctime} : float ;
   Last status change time

\}

The information returned by the \texttt{Unix.stat} \ref{21.1} calls.

\textbf{val \texttt{stat}} : string \rightarrow \text{stats}
   Return the information for the named file.

\textbf{val \texttt{lstat}} : string \rightarrow \text{stats}
   Same as \texttt{Unix.stat} \ref{21.1}, but in case the file is a symbolic link, return the information for the link itself.

\textbf{val \texttt{fstat}} : file\_descr \rightarrow \text{stats}
   Return the information for the file associated with the given descriptor.

\textbf{val \texttt{isatty}} : file\_descr \rightarrow \text{bool}
   Return true if the given file descriptor refers to a terminal or console window, false otherwise.
module LargeFile :
  sig
    val lseek : Unix.file_descr -> int64 -> Unix.seek_command -> int64
    val truncate : string -> int64 -> unit
    val ftruncate : Unix.file_descr -> int64 -> unit
  type stats = {
    st_dev : int ;
    Device number
    st_ino : int ;
    Inode number
    st_kind : Unix.file_kind ;
    Kind of the file
    st_perm : Unix.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 file’s group
    st_rdev : int ;
    Device minor number
    st_size : int64 ;
    Size in bytes
    st_atime : float ;
    Last access time
    st_mtime : float ;
    Last modification time
    st_ctime : float ;
    Last status change time
  
File operations on large files
Chapter 21. The unix library: Unix system calls

{ }
val stat : string -> stats
val lstat : string -> stats
val fstat : Unix.file_descr -> stats
end

File operations on large files. This sub-module provides 64-bit variants of the functions
Unix.lseek\[21.1\] (for positioning a file descriptor), Unix.truncate\[21.1\] and
Unix.ftruncate\[21.1\] (for changing the size of a file), and Unix.stat\[21.1\],
Unix.lstat\[21.1\] and Unix.fstat\[21.1\] (for obtaining information on files). These alternate
functions represent positions and sizes by 64-bit integers (type int64) instead of regular
integers (type int), thus allowing operating on files whose sizes are greater than max_int.

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 source.

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 Unix.access\[21.1\] 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.

\[
\text{val chown : string -> int -> int -> unit}
\]
Change the owner uid and owner gid of the named file.

\[
\text{val fchown : file_descr -> int -> int -> unit}
\]
Change the owner uid and owner gid of an opened file.

\[
\text{val umask : int -> int}
\]
Set the process’s file mode creation mask, and return the previous mask.

\[
\text{val access : string -> access_permission list -> unit}
\]
Check that the process has the given permissions over the named file. Raise Unix_error otherwise.

**Operations on file descriptors**

\[
\text{val dup : file_descr -> file_descr}
\]
Return a new file descriptor referencing the same file as the given descriptor.

\[
\text{val dup2 : file_descr -> file_descr -> unit}
\]
dup2 fd1 fd2 duplicates fd1 to fd2, closing fd2 if already opened.

\[
\text{val set_nonblock : file_descr -> unit}
\]
Set 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.

\[
\text{val clear_nonblock : file_descr -> unit}
\]
Clear the “non-blocking” flag on the given descriptor. See Unix.set_nonblock[21.1].

\[
\text{val set_close_on_exec : file_descr -> unit}
\]
Set 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.

\[
\text{val clear_close_on_exec : file_descr -> unit}
\]
Clear the “close-on-exec” flag on the given descriptor. See Unix.set_close_on_exec[21.1].
Chapter 21. The unix library: Unix system calls

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.
   Raises 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.

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
Unix.create_process, except that the extra argument env specifies the environment
passed to the program.

val open_process_in : string -> Pervasives.in_channel

High-level pipe and process management. This function runs the given command in parallel
with the program. The standard output of the command is redirected to a pipe, which can
be read via the returned input channel. The command is interpreted by the shell /bin/sh
(cf. system).

val open_process_out : string -> Pervasives.out_channel

Same as Unix.open_process_in, but redirect the standard input of the command to a
pipe. Data written to the returned output channel is sent to the standard input of the
command. Warning: writes on output channels are buffered, hence be careful to call
Pervasives.flush at the right times to ensure correct synchronization.

val open_process : string -> Pervasives.in_channel * Pervasives.out_channel

Same as Unix.open_process_out, but redirects both the standard input and standard
output of the command to pipes connected to the two returned channels. The input channel
is connected to the output of the command, and the output channel to the input of the
command.

val open_process_full :
  string ->
  string array ->
Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel
Similar to Unix.open_process\[21.1\], but the second argument specifies the environment passed to the command. The result is a triple of channels connected respectively to the standard output, standard input, and standard error of the command.

```ocaml
val close_process_in : Pervasives.in_channel -> process_status
  Close channels opened by Unix.open_process_in\[21.1\], wait for the associated command to terminate, and return its termination status.

val close_process_out : Pervasives.out_channel -> process_status
  Close channels opened by Unix.open_process_out\[21.1\], wait for the associated command to terminate, and return its termination status.

val close_process : Pervasives.in_channel * Pervasives.out_channel -> process_status
  Close channels opened by Unix.open_process\[21.1\], wait for the associated command to terminate, and return its termination status.

val close_process_full : Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel -> process_status
  Close channels opened by Unix.open_process_full\[21.1\], wait for the associated command to terminate, and return its termination status.
```

**Symbolic links**

```ocaml
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**

```ocaml
val select : file_descr list -> 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
  | F_LOCK
  | F_TLOCK
  | F_TEST
  | F_RLOCK
  | F_TRLOCK

Unlock a region
Lock a region for writing, and block if already locked
Lock a region for writing, or fail if already locked
Test a region for other process locks
Lock a region for reading, and block if already locked
Lock a region for reading, or fail if already locked

Commands for Unix.lockf[21.1].

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 Unix.lseek[21.1]), 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 prevents any other process from acquiring a read or write lock on the region. A read lock prevents any other process from acquiring a write lock on the region, but lets other processes acquire read locks on it.

The F_LOCK and F_TLOCK commands attempts to put a write lock on the specified region. The F_RLOCK and F_TRLOCK commands attempts to put a read lock on the specified region. If one or several locks put by another process prevent the current process from acquiring the lock, F_LOCK and F_RLOCK block until these locks are removed, while F_TLOCK and F_TRLOCK fail immediately with an exception. The F_ULOCK removes whatever locks the current process has on the specified region. Finally, the F_TEST command tests whether a write lock can be acquired on the specified region, without actually putting a lock. It returns immediately if successful, or fails otherwise.

Signals

Note: installation of signal handlers is performed via the functions Sys.signal[20.34] and Sys.set_signal[20.34].

val kill : int -> int -> unit

kill pid sig sends signal number sig to the process with id pid.
Chapter 21. The unix library: Unix system calls

```ml

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 sigs 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

```ml

```ml

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 sigs 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.

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..60
  tm_min : int ;
  Minutes 0..59
```
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 Unix.time[21.1], but with resolution better than 1 second.

val gmtime : float -> tm
    Convert a time in seconds, as returned by Unix.time[21.1], into a date and a time. Assumes
UTC (Coordinated Universal Time), also known as GMT.

val localtime : float -> tm
    Convert a time in seconds, as returned by Unix.time[21.1], 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 Unix.time[21.1]. The tm_isdst, tm_wday and tm_yday fields of tm are ignored. 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, and the other fields normalized (so that, e.g., 40
October is changed into 9 November). The tm argument is interpreted in the local time zone.

val alarm : int -> int
    Schedule a SIGALRM signal 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. A time of 0.0 is interpreted as the current time.

type interval_timer =
   | 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.

The three kinds of interval timers.

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.

val setgroups : int array -> unit
    setgroups groups sets the supplementary group IDs for the calling process. Appropriate privileges are required.

val initgroups : string -> int -> unit
    initgroups user group initializes the group access list by reading the group database /etc/group and using all groups of which user is a member. The additional group group is also added to the list.

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
  Conversion from the printable representation of an Internet address to its internal representation. The argument string consists of 4 numbers separated by periods (XXX.YYY.ZZZ.TTT) for IPv4 addresses, and up to 8 numbers separated by colons for IPv6 addresses. Raise Failure when given a string that does not match these formats.

val string_of_inet_addr : inet_addr -> string
  Return the printable representation of the given Internet address. See Unix.inet_addr_of_string[21.1] for a description of the printable representation.

val inet_addr_any : inet_addr
  A special IPv4 address, for use only with bind, representing all the Internet addresses that the host machine possesses.

val inet_addr_loopback : inet_addr
  A special IPv4 address representing the host machine (127.0.0.1).
val inet6_addr_any : inet_addr
A special IPv6 address, for use only with `bind`, representing all the Internet addresses that the host machine possesses.

val inet6_addr_loopback : inet_addr
A special IPv6 address representing the host machine (`::1`).

### Sockets

```
type socket_domain =
  | PF_UNIX
    Unix domain
  | PF_INET
    Internet domain (IPv4)
  | PF_INET6
    Internet domain (IPv6)

  The type of socket domains. Not all platforms support IPv6 sockets (type `PF_INET6`).
```

```
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 domain_of_sockaddr : sockaddr -> socket_domain

Return the socket domain adequate for the given socket address.

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

Shut down 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 Unix.recv\[21.1\], Unix.recvfrom\[21.1\], Unix.send\[21.1\] and
  Unix.sendto\[21.1\].

val recv : file_descr -> string -> int -> int -> msg_flag list -> int
  Receive data from a connected socket.

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
  Send data over a connected socket.

val sendto :
  file_descr ->
  string -> int -> int -> msg_flag list -> sockaddr -> int
  Send data over an unconnected socket.

Socket options

type socket_bool_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
  | SO_ACCEPTCONN
    Report whether socket listening is enabled
| TCP_NODELAY
  | Control the Nagle algorithm for TCP sockets
| IPV6_ONLY
  | Forbid binding an IPv6 socket to an IPv4 address

The socket options that can be consulted with \texttt{Unix.getsockopt} and modified with \texttt{Unix.setsockopt}. These options have a boolean \texttt{(true/false)} value.

```haskell
type socket_int_option =
  | SO_SNDBUF
    | Size of send buffer
  | SO_RCVBUF
    | Size of received buffer
  | SO_ERROR
    | Deprecated. Use \texttt{Unix.getsockopt_error} instead.
  | SO_TYPE
    | Report the socket type
  | SO_RCVLOWAT
    | Minimum number of bytes to process for input operations
  | SO_SNDLOWAT
    | Minimum number of bytes to process for output operations
```

The socket options that can be consulted with \texttt{Unix.getsockopt_int} and modified with \texttt{Unix.setsockopt_int}. These options have an integer value.

```haskell
type socket_optint_option =
  | SO_LINGER
    | Whether to linger on closed connections that have data present, and for how long (in seconds)
```

The socket options that can be consulted with \texttt{Unix.getsockopt_optint} and modified with \texttt{Unix.setsockopt_optint}. These options have a value of type \texttt{int option}, with \texttt{None} meaning “disabled”.

```haskell
type socket_float_option =
  | SO_RCVTIMEO
    | Timeout for input operations
  | SO_SNDTIMEO
    | Timeout for output operations
```

The socket options that can be consulted with \texttt{Unix.getsockopt_float} and modified with \texttt{Unix.setsockopt_float}. These options have a floating-point value representing a time in seconds. The value 0 means infinite timeout.
val getsockopt : file_descr -> socket_bool_option -> bool
  Return the current status of a boolean-valued option in the given socket.
val setsockopt : file_descr -> socket_bool_option -> bool -> unit
  Set or clear a boolean-valued option in the given socket.
val getsockopt_int : file_descr -> socket_int_option -> int
  Same as Unix.getsockopt\[21.1\] for an integer-valued socket option.
val setsockopt_int : file_descr -> socket_int_option -> int -> unit
  Same as Unix.setsockopt\[21.1\] for an integer-valued socket option.
val getsockopt_optint : file_descr -> socket_optint_option -> int option
  Same as Unix.getsockopt\[21.1\] for a socket option whose value is an int option.
val setsockopt_optint : file_descr -> socket_optint_option -> int option -> unit
  Same as Unix.setsockopt\[21.1\] for a socket option whose value is an int option.
val getsockopt_float : file_descr -> socket_float_option -> float
  Same as Unix.getsockopt\[21.1\] for a socket option whose value is a floating-point number.
val setsockopt_float : file_descr -> socket_float_option -> float -> unit
  Same as Unix.setsockopt\[21.1\] for a socket option whose value is a floating-point number.
val getsockopt_error : file_descr -> error option
  Return the error condition associated with the given socket, and clear it.

High-level network connection functions

val open_connection : sockaddr -> Pervasives.in_channel * Pervasives.out_channel
  Connect to a server at the given address. Return a pair of buffered channels connected to
  the server. Remember to call Pervasives.flush\[19.2\] on the output channel at the right
times to ensure correct synchronization.
val shutdown_connection : Pervasives.in_channel -> unit
  “Shut down” a connection established with Unix.open_connection\[21.1\]; that is, transmit
  an end-of-file condition to the server reading on the other side of the connection.
val establish_server : (Pervasives.in_channel -> Pervasives.out_channel -> unit) ->
                     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 Unix.establish_server\[21.1\] 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 getprotobynumber : int -> protocol_entry
  Find an entry in protocols with the given protocol number, 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.
type addr_info = {
  ai_family : socket_domain ;
  Socket domain
  ai_socktype : socket_type ;
  Socket type
  ai_protocol : int ;
  Socket protocol number
  ai_addr : sockaddr ;
  Address
  ai_canonname : string ;
  Canonical host name
}

Address information returned by Unix.getaddrinfo

val getaddrinfo : string -> string -> getaddrinfo_option list -> addr_info list

getaddrinfo host service opts returns a list of Unix.addr_info records describing
socket parameters and addresses suitable for communicating with the given host and
service. The empty list is returned if the host or service names are unknown, or the
constraints expressed in opts cannot be satisfied.

host is either a host name or the string representation of an IP address. host can be given
as the empty string; in this case, the “any” address or the “loopback” address are used,
depending whether opts contains AI_PASSIVE. service is either a service name or the
string representation of a port number. service can be given as the empty string; in this case, the port field of the returned addresses is set to 0. opts is a possibly empty list of options that allows the caller to force a particular socket domain (e.g. IPv6 only or IPv4 only) or a particular socket type (e.g. TCP only or UDP only).

```ocaml
type name_info = {
  ni_hostname : string ;
  Name or IP address of host
  ni_service : string ;
  Name of service or port number
}
```

Host and service information returned by Unix.getnameinfo[21.1].

```ocaml
type getnameinfo_option =
  | NI_NOFQDN
  Do not qualify local host names
  | NI_NUMERICHOST
  Always return host as IP address
  | NI_NAMEREQD
  Fail if host name cannot be determined
  | NI_NUMERICSERV
  Always return service as port number
  | NI_DGRAM
  Consider the service as UDP-based instead of the default TCP
```

Options to Unix.getnameinfo[21.1].

```ocaml
val getnameinfo : sockaddr -> getnameinfo_option list -> name_info
getnameinfo addr opts returns the host name and service name corresponding to the socket address addr. opts is a possibly empty list of options that governs how these names are obtained. Raise Not_found if an error occurs.
```

**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.

```ocaml
type terminal_io = {
  mutable c_ignbrk : bool ;
  Ignore the break condition.
  mutable c_brkint : bool ;
  ...}
```


Signal interrupt on break condition.

```plaintext
mutable c_ignpar : bool ;
    Ignore characters with parity errors.
```

```plaintext
mutable c_parmrk : bool ;
    Mark parity errors.
```

```plaintext
mutable c_inpck : bool ;
    Enable parity check on input.
```

```plaintext
mutable c_istrip : bool ;
    Strip 8th bit on input characters.
```

```plaintext
mutable c_inlcr : bool ;
    Map NL to CR on input.
```

```plaintext
mutable c_igncr : bool ;
    Ignore CR on input.
```

```plaintext
mutable c_icrnl : bool ;
    Map CR to NL on input.
```

```plaintext
mutable c_ixon : bool ;
    Recognize XON/XOFF characters on input.
```

```plaintext
mutable c_ixoff : bool ;
    Emit XON/XOFF chars to control input flow.
```

```plaintext
mutable c_opost : bool ;
    Enable output processing.
```

```plaintext
mutable c_obaud : int ;
    Output baud rate (0 means close connection).
```

```plaintext
mutable c_ibaud : int ;
    Input baud rate.
```

```plaintext
mutable c_csize : int ;
    Number of bits per character (5-8).
```

```plaintext
mutable c_cstopb : int ;
    Number of stop bits (1-2).
```

```plaintext
mutable c_cread : bool ;
    Reception is enabled.
```

```plaintext
mutable c_parenb : bool ;
    Enable parity generation and detection.
```

```plaintext
mutable c_parodd : bool ;
```
Specify odd parity instead of even.

```c
mutable c_hupcl : bool ;
Hang up on last close.
```

```c
mutable c_clocal : bool ;
Ignore modem status lines.
```

```c
mutable c_isig : bool ;
Generate signal on INTR, QUIT, SUSP.
```

```c
mutable c_icanon : bool ;
Enable canonical processing (line buffering and editing)
```

```c
mutable c_noflsh : bool ;
Disable flush after INTR, QUIT, SUSP.
```

```c
mutable c_echo : bool ;
Echo input characters.
```

```c
mutable c_echoe : bool ;
Echo ERASE (to erase previous character).
```

```c
mutable c_echok : bool ;
Echo KILL (to erase the current line).
```

```c
mutable c_echonl : bool ;
Echo NL even if c_echo is not set.
```

```c
mutable c_vintr : char ;
Interrupt character (usually ctrl-C).
```

```c
mutable c_vquit : char ;
Quit character (usually ctrl-\).
```

```c
mutable c_verase : char ;
Erase character (usually DEL or ctrl-H).
```

```c
mutable c_vkill : char ;
Kill line character (usually ctrl-U).
```

```c
mutable c_veof : char ;
End-of-file character (usually ctrl-D).
```

```c
mutable c_veol : char ;
Alternate end-of-line char. (usually none).
```

```c
mutable c_vmin : int ;
Minimum number of characters to read before the read request is satisfied.
```

```c
mutable c_vtime : int ;
```
Maximum read wait (in 0.1s units).

```haskell
mutable c_vstart : char ;
    Start character (usually ctrl-Q).
mutable c_vstop : char ;
    Stop character (usually ctrl-S).
```

```haskell
val tcgetattr : file_descr -> terminal_io
    Return the status of the terminal referred to by the given file descriptor.
```

```haskell
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.
```

```haskell
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).
```

```haskell
val tcdrain : file_descr -> unit
    Waits until all output written on the given file descriptor has been transmitted.
```

```haskell
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.
```

```haskell
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: TCOFF suspends output, TCOON restarts output, TCIFF 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.

21.2 Module UnixLabels: labelized version of the interface

This module is identical to Unix[21.1], and only differs by the addition of labels. You may see these labels directly by looking at unixLabels.mli, or by using the ocamlbrowser tool.
Windows:
The Cygwin port of OCaml fully implements all functions from the Unix module. The native Win32 ports implement a subset of them. Below is a list of the functions that are not implemented, or only partially implemented, by the Win32 ports. Functions not mentioned are fully implemented and behave as described previously in this chapter.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>not implemented, use <code>create_process</code> or threads</td>
</tr>
<tr>
<td>wait</td>
<td>not implemented, use <code>waitpid</code></td>
</tr>
<tr>
<td>waitpid</td>
<td>can only wait for a given PID, not any child process</td>
</tr>
<tr>
<td>getppid</td>
<td>not implemented (meaningless under Windows)</td>
</tr>
<tr>
<td>nice</td>
<td>not implemented</td>
</tr>
<tr>
<td>truncate, ftruncate</td>
<td>not implemented (no links under Windows)</td>
</tr>
<tr>
<td>link, symlink, readlink</td>
<td>execute permission <code>X_OK</code> cannot be tested, it just tests for read permission instead</td>
</tr>
<tr>
<td>access</td>
<td>not implemented</td>
</tr>
<tr>
<td>fchmod</td>
<td>not implemented</td>
</tr>
<tr>
<td>chown, fchown</td>
<td>not implemented (make no sense on a DOS file system)</td>
</tr>
<tr>
<td>umask</td>
<td>not implemented</td>
</tr>
<tr>
<td>set_nonblock, clear_nonblock</td>
<td>implemented as dummy functions; use threads instead of non-blocking I/O</td>
</tr>
<tr>
<td>rewinddir</td>
<td>not implemented; re-open the directory instead</td>
</tr>
<tr>
<td>mkfifo</td>
<td>not implemented</td>
</tr>
<tr>
<td>kill, pause</td>
<td>not implemented (no inter-process signals in Windows)</td>
</tr>
<tr>
<td>alarm, times</td>
<td>not implemented</td>
</tr>
<tr>
<td>getitimer, setitimer</td>
<td>not implemented</td>
</tr>
<tr>
<td>getuid, getgid</td>
<td>always return 1</td>
</tr>
<tr>
<td>getgid, getegid, getgroups</td>
<td>not implemented</td>
</tr>
<tr>
<td>setuid, setgid</td>
<td>not implemented</td>
</tr>
<tr>
<td>getpwnam, getpwuid</td>
<td>always raise <code>Not_found</code></td>
</tr>
<tr>
<td>getgrnam, getgrgid</td>
<td>always raise <code>Not_found</code></td>
</tr>
<tr>
<td>type <code>socket_domain</code></td>
<td>the domains <code>PF_UNIX</code> and <code>PF_INET6</code> are not supported; <code>PF_INET</code> is fully supported</td>
</tr>
<tr>
<td>establish_server</td>
<td>not implemented; use threads</td>
</tr>
<tr>
<td>terminal functions (<code>tc*</code>)</td>
<td>not implemented</td>
</tr>
</tbody>
</table>
Chapter 22

The num library: arbitrary-precision rational arithmetic

The num library implements integer arithmetic and rational arithmetic in arbitrary precision.


Programs that use the num library must be linked as follows:

\begin{verbatim}
  ocamlc other options nums.cma other files
  ocamlopt other options nums.cmxa other files
\end{verbatim}

For interactive use of the nums library, do:

\begin{verbatim}
  ocamlmktop -o mytop nums.cma
  ./mytop
\end{verbatim}

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type \texttt{#load "nums.cma";;}.

22.1 Module Num : Operation on arbitrary-precision numbers.

Numbers (type num) are arbitrary-precision rational numbers, plus the special elements \(1/0\) (infinity) and \(0/0\) (undefined).

\begin{verbatim}
type num =
  | Int of int
  | Big_int of Big_int.big_int
  | Ratio of Ratio.ratio
\end{verbatim}

The type of numbers.
Arithmetic operations

val (+/ ) : num -> num -> num
  Same as Num.add_num[22.1].

val add_num : num -> num -> num
  Addition

val minus_num : num -> num
  Unary negation.

val (−/) : num -> num -> num
  Same as Num.sub_num[22.1].

val sub_num : num -> num -> num
  Subtraction

val ( */ ) : num -> num -> num
  Same as Num.mult_num[22.1].

val mult_num : num -> num -> num
  Multiplication

val square_num : num -> num
  Squaring

val (//) : num -> num -> num
  Same as Num.div_num[22.1].

val div_num : num -> num -> num
  Division

val quo_num : num -> num -> num
  Euclidean division: quotient.

val mod_num : num -> num -> num
  Euclidean division: remainder.

val ( **/ ) : num -> num -> num
  Same as Num.power_num[22.1].

val power_num : num -> num -> num
  Exponentiation

val abs_num : num -> num
Chapter 22. The num library: arbitrary-precision rational arithmetic

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 Pervasives.ref -> unit
incr r is r:=!r+1, where r is a reference to a number.

val decr_num : num Pervasives.ref -> unit
decr r is r:=!r-1, where r is a reference to a number.

val is_integer_num : num -> bool
Test if a number is an integer

The four following functions approximate a number by an integer :

val integer_num : num -> num
integer_num n returns the integer closest to n. In case of ties, rounds towards zero.

val floor_num : num -> num
floor_num n returns the largest integer smaller or equal to n.

val round_num : num -> num
round_num n returns the integer closest to n. In case of ties, rounds off zero.

val ceiling_num : num -> num
ceiling_num n returns the smallest integer bigger or equal to n.

val sign_num : num -> int
Return -1, 0 or 1 according to the sign of the argument.

Comparisons between numbers

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
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
    Return the greater of the two arguments.
val min_num : num -> num -> num
    Return the smaller of the two arguments.

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
    See Num.approx_num_exp\textsuperscript{[22.1]}.
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. Num.approx_num_fix\textsuperscript{[22.1]} uses decimal notation; the first argument is the number of digits after the decimal point. approx_num_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.nat
val num_of_nat : Nat.nat -> num
val num_of_big_int : Big_int.big_int -> num
val big_int_of_num : num -> Big_int.big_int
val ratio_of_num : num -> Ratio.ratio
val num_of_ratio : Ratio.ratio -> num
val float_of_num : num -> float
Module Big_int: Operations on arbitrary-precision integers.

Big integers (type big_int) are signed integers of arbitrary size.

```ocaml
type big_int
  The type of big integers.

val zero_big_int : big_int
  The big integer 0.

val unit_big_int : big_int
  The big integer 1.

Arithmetic operations

val minus_big_int : big_int -> big_int
  Unary negation.

val abs_big_int : big_int -> big_int
  Absolute value.

val add_big_int : big_int -> big_int -> big_int
  Addition.

val succ_big_int : big_int -> big_int
  Successor (add 1).

val add_int_big_int : int -> big_int -> big_int
  Addition of a small integer to a big integer.

val sub_big_int : big_int -> big_int -> big_int
  Subtraction.

val pred_big_int : big_int -> big_int
  Predecessor (subtract 1).

val mult_big_int : big_int -> big_int -> big_int
  Multiplication of two big integers.

val mult_int_big_int : int -> big_int -> big_int
  Multiplication of a big integer by a small integer.

val square_big_int : big_int -> big_int
```

Return the square of the given big integer

val sqrt_big_int : big_int -> big_int

sqrt_big_int a returns the integer square root of a, that is, the largest big integer r such that \( r \times r \leq a \). Raise Invalid_argument if a is negative.

val quomod_big_int : big_int -> big_int -> big_int * big_int

Euclidean division of two big integers. The first part of the result is the quotient, the second part is the remainder. Writing \( (q,r) = quomod_big_int a b \), we have \( a = q \times b + r \) and \( 0 \leq r < |b| \). Raise Division_by_zero if the divisor is zero.

val div_big_int : big_int -> big_int -> big_int

Euclidean quotient of two big integers. This is the first result q of quomod_big_int (see above).

val mod_big_int : big_int -> big_int -> big_int

Euclidean modulus of two big integers. This is the second result r of quomod_big_int (see above).

val gcd_big_int : big_int -> big_int -> big_int

Greatest common divisor of two big integers.

val power_int_positive_int : int -> int -> big_int
val power_big_int_positive_int : big_int -> int -> big_int
val power_int_positive_big_int : int -> big_int -> big_int
val power_big_int_positive_big_int : big_int -> big_int -> big_int

Exponentiation functions. Return the big integer representing the first argument a raised to the power b (the second argument). Depending on the function, a and b can be either small integers or big integers. Raise Invalid_argument if b is negative.

Comparisons and tests

val sign_big_int : big_int -> int

Return 0 if the given big integer is zero, 1 if it is positive, and -1 if it is negative.

val compare_big_int : big_int -> big_int -> int

compare_big_int a b returns 0 if a and b are equal, 1 if a is greater than b, and -1 if a is smaller than b.

val eq_big_int : big_int -> big_int -> bool
val le_big_int : big_int -> big_int -> bool
val ge_big_int : big_int -> big_int -> bool
val lt_big_int : big_int -> big_int -> bool
val gt_big_int : big_int -> big_int -> bool
Usual boolean comparisons between two big integers.

val max_big_int : big_int -> big_int -> big_int
  Return the greater of its two arguments.

val min_big_int : big_int -> big_int -> big_int
  Return the smaller of its two arguments.

val num_digits_big_int : big_int -> int
  Return the number of machine words used to store the given big integer.

Conversions to and from strings

val string_of_big_int : big_int -> string
  Return the string representation of the given big integer, in decimal (base 10).

val big_int_of_string : string -> big_int
  Convert a string to a big integer, in decimal. The string consists of an optional - or + sign, followed by one or several decimal digits.

Conversions to and from other numerical types

val big_int_of_int : int -> big_int
  Convert a small integer to a big integer.

val is_int_big_int : big_int -> bool
  Test whether the given big integer is small enough to be representable as a small integer (type int) without loss of precision. On a 32-bit platform, is_int_big_int a returns true if and only if a is between \(2^{30}\) and \(2^{30}-1\). On a 64-bit platform, is_int_big_int a returns true if and only if a is between \(-2^{62}\) and \(2^{62}-1\).

val int_of_big_int : big_int -> int
  Convert a big integer to a small integer (type int). Raises Failure "int_of_big_int" if the big integer is not representable as a small integer.

val big_int_of_int32 : int32 -> big_int
  Convert a 32-bit integer to a big integer.

val big_int_of_nativeint : nativeint -> big_int
  Convert a native integer to a big integer.

val big_int_of_int64 : int64 -> big_int
  Convert a 64-bit integer to a big integer.
val int32_of_big_int : big_int -> int32
  Convert a big integer to a 32-bit integer. Raises Failure if the big integer is outside the range \([-2^{31}, 2^{31}-1]\).

val nativeint_of_big_int : big_int -> nativeint
  Convert a big integer to a native integer. Raises Failure if the big integer is outside the range \([\text{Nativeint.min_int}, \text{Nativeint.max_int}]\).

val int64_of_big_int : big_int -> int64
  Convert a big integer to a 64-bit integer. Raises Failure if the big integer is outside the range \([-2^{63}, 2^{63}-1]\).

val float_of_big_int : big_int -> float
  Returns a floating-point number approximating the given big integer.

Bit-oriented operations

val and_big_int : big_int -> big_int -> big_int
  Bitwise logical “and”. The arguments must be positive or zero.

val or_big_int : big_int -> big_int -> big_int
  Bitwise logical “or”. The arguments must be positive or zero.

val xor_big_int : big_int -> big_int -> big_int
  Bitwise logical “exclusive or”. The arguments must be positive or zero.

val shift_left_big_int : big_int -> int -> big_int
  shift_left_big_int b n returns b shifted left by n bits. Equivalent to multiplication by 2^n.

val shift_right_big_int : big_int -> int -> big_int
  shift_right_big_int b n returns b shifted right by n bits. Equivalent to division by 2^n with the result being rounded towards minus infinity.

val shift_right_towards_zero_big_int : big_int -> int -> big_int
  shift_right_towards_zero_big_int b n returns b shifted right by n bits. The shift is performed on the absolute value of b, and the result has the same sign as b. Equivalent to division by 2^n with the result being rounded towards zero.

val extract_big_int : big_int -> int -> int -> big_int
  extract_big_int bi ofs n returns a nonnegative number corresponding to bits ofs to ofs + n - 1 of the binary representation of bi. If bi is negative, a two’s complement representation is used.
22.3 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
See *Arith_status.set_error_when_null_denominator*[22.3].

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
See *Arith_status.set_normalize_ratio*[22.3].

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
See *Arith_status.set_normalize_ratio_when_printing*[22.3].

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
See *Arith_status.set_approx_printing*[22.3].

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
See *Arith_status.set_floating_precision*[22.3].

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 23

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 as follows:

```ocaml
ocamlc other options str.cma other files
ocamlopt other options str.cmxa other files
```

For interactive use of the str library, do:

```ocaml
ocamlmktop -o mytop str.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type 

```
#load "str.cma";;
```

23.1 Module Str : Regular expressions and high-level string processing

Regular expressions

```ocaml
type regexp

The type of compiled regular expressions.
```

```ocaml
val regexp : string -> regexp

Compile a regular expression. The following constructs are recognized:
```

- .  Matches any character except newline.
- * (postfix) Matches the preceding expression zero, one or several times
• + (postfix) Matches the preceding expression one or several times
• ? (postfix) Matches the preceding 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. To include a ] character in a set, make it the first character of the set. To include a − character in a set, make it the first or the last character of the set.
• ^ Matches at beginning of line (either at the beginning of the matched string, or just after a newline character).
• $ Matches at end of line (either at the end of the matched string, or just before a newline character).
• \| (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, and so on up to \9).
• \b Matches word boundaries.
• \ Quotes special characters. The special characters are $^.*+?[].

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
  Str.regexp_string s returns a regular expression that matches exactly s and nothing else.

val regexp_string_case_fold : string -> regexp
  Str.regexp_string_case_fold is similar to Str.regexp_string[23.1], 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 a substring of s that starts at position start matches 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 searches 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
search_backward r s last searches the string s for a substring matching the regular expression r. The search first considers substrings that start at position last and proceeds towards the beginning of string. Return the position of the first character of the matched substring; raise Not_found if no substring matches.

val string_partial_match : regexp -> string -> int -> bool
Similar to Str.string_match[23.1], but also returns true if 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 Str.string_match[23.1], Str.search_forward[23.1] or Str.search_backward[23.1]. 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
match_beginning() returns the position of the first character of the substring that was matched by Str.string_match[23.1], Str.search_forward[23.1] or Str.search_backward[23.1].

val match_end : unit -> int
match_end() returns the position of the character following the last character of the substring that was matched by string_match, search_forward or search_backward.

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 Str.string_match[23.1], Str.search_forward[23.1] or Str.search_backward[23.1]. The user must make sure that the parameter s is the same string that was passed to the matching or searching function. matched_group n s raises Not_found if the n-th group of the regular expression was not matched. This can happen with groups inside alternatives \|, options ? or repetitions *. For instance, the empty string will match \( (a\)*, but matched_group 1 "" will raise Not_found because the first group itself was not matched.

val group_beginning : 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.

Raises
- Not_found if the n-th group of the regular expression was not matched.
- Invalid_argument if there are fewer than n groups in the regular expression.

val group_end : int -> int
group_end n returns the position of the character following the last character of substring that was matched by the nth group of the regular expression.

Raises

- Not_found if the nth group of the regular expression was not matched.
- Invalid_argument if there are fewer than n groups in the regular expression.

Replacement

val global_replace : regexp -> string -> string -> string

global_replace regexp templ s returns a string identical to s, except that all substrings of s that match regexp have been replaced by templ. The replacement template templ 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 Str.global_replace[23.1], 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 Str.global_substitute[23.1], 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 Str.split[23.1], but splits into at most n substrings, where n is the extra integer parameter.

val split_delim : regexp -> string -> string list
  Same as Str.split[23.1] 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"].

val bounded_split_delim : regexp -> string -> int -> string list
  Same as Str.bounded_split[23.1], 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.

type split_result =
  | Text of string
  | Delim of string

val full_split : regexp -> string -> split_result list
  Same as Str.split_delim[23.1], 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 "]"].

val bounded_full_split : regexp -> string -> int -> split_result list
  Same as Str.bounded_split_delim[23.1], 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.

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 Str.string_before[23.1].

val last_chars : string -> int -> string
  last_chars s n returns the last n characters of s.
Chapter 24

The threads library

The threads library allows concurrent programming in OCaml. 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.

Two implementations of the threads library are available, depending on the capabilities of the operating system:

- System threads. This implementation builds on the OS-provided threads facilities: POSIX 1003.1c threads for Unix, and Win32 threads for Windows. When available, system threads support both bytecode and native-code programs.
- VM-level threads. This implementation performs time-sharing and context switching at the level of the OCaml virtual machine (bytecode interpreter). It is available on Unix systems, and supports only bytecode programs. It cannot be used with native-code programs.

Programs that use system threads must be linked as follows:

\[
\text{ocamlc -thread other options unix.cma threads.cma other files} \\
\text{ocamlopt -thread other options unix.cmxa threads.cmxa other files}
\]

Compilation units that use the threads library must also be compiled with the -thread option (see chapter 8).

Programs that use VM-level threads must be compiled with the -vmthread option to ocamlc (see chapter 8), and be linked as follows:

\[
\text{ocamlc -vmthread other options threads.cma other files}
\]

Compilation units that use threads library must also be compiled with the -vmthread option (see chapter 8).
24.1 Module Thread: Lightweight threads for Posix 1003.1c and Win32.

```ocaml
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.

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
  See Thread.wait_write 24.1.
```
Chapter 24. The threads library

val wait_write : Unix.file_descr -> unit
   This function does nothing in this implementation.

val wait_timed_read : Unix.file_descr -> float -> bool
   See Thread.wait_timed_read[24.1].

val wait_timed_write : Unix.file_descr -> float -> bool
   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. 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.

   These functions return immediately true in the Win32 implementation.

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 thread until input/output becomes possible on the given
   Unix file descriptors. The arguments and results have the same meaning as for
   Unix.select. This function is not implemented yet under Win32.

val wait_pid : int -> int * Unix.process_status
   wait_pid p suspends the execution of the calling thread until the process specified by the
   process identifier p terminates. Returns the pid of the child caught and its termination
   status, as per Unix.wait. This function is not implemented under MacOS.

val yield : unit -> unit
   Re-schedule the calling thread without suspending it. This function can be used to give
   scheduling hints, telling the scheduler that now is a good time to switch to other threads.

Management of signals

Signal handling follows the POSIX thread model: signals generated by a thread are delivered to
that thread; signals generated externally are delivered to one of the threads that does not block it.
Each thread possesses a set of blocked signals, which can be modified using Thread.sigmask[24.1].
This set is inherited at thread creation time. Per-thread signal masks are supported only by the
system thread library under Unix, but not under Win32, nor by the VM thread library.

val sigmask : Unix.sigprocmask_command -> int list -> int list
   sigmask cmd sigs changes the set of blocked signals for the calling thread. 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. sigmask returns the set of previously
   blocked signals for the thread.
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. The signals sigs are expected to be blocked before calling wait_signal.

24.2 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 \( D \)):

```ocaml
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 Mutex.lock\(^{24.2}\), but does not suspend the calling thread if the mutex is already locked: just return \( \text{false} \) immediately in that case. If the mutex is unlocked, lock it and return \( \text{true} \).

val unlock : t -> unit

Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.

24.3 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):

```ocaml
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 satisfied *) then Condition.signal c;
Mutex.unlock m
```

**type t**  
The type of condition variables.

**val create : unit \(\rightarrow\) t**  
Return a new condition variable.

**val wait : t \(\rightarrow\) Mutex.t \(\rightarrow\) unit**  
\(\text{wait } c \ m\) 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 \(\text{wait}\) returns.

**val signal : t \(\rightarrow\) unit**  
\(\text{signal } c\) restarts one of the processes waiting on the condition variable \(c\).

**val broadcast : t \(\rightarrow\) unit**  
\(\text{broadcast } c\) restarts all processes waiting on the condition variable \(c\).

### 24.4 Module Event: First-class synchronous communication.

This module implements synchronous inter-thread 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 \(\rightarrow\) ’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 wrap_abort : 'a event -> (unit -> unit) -> 'a event

wrap_abort ev fn returns the event that performs the same communications as ev, but if it is not selected the function fn is called after the synchronization.

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 Event.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.
24.5 Module ThreadUnix : Thread-compatible system calls.

The functionality of this module has been merged back into the Unix
module. Threaded programs can now call the functions from module Unix
directly, and still get the correct behavior (block the calling thread, if required, but do not block all threads in the process). Thread-compatible system calls.

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
    See ThreadUnix.timed_write.

val timed_write : Unix.file_descr -> string -> int -> int -> float -> int
    Behave as ThreadUnix.read and ThreadUnix.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 rediretions

val pipe : unit -> Unix.file_descr * Unix.file_descr
val open_process_in : string -> Pervasives.in_channel
val open_process_out : string -> Pervasives.out_channel
val open_process : string -> Pervasives.in_channel * Pervasives.out_channel

Time

val sleep : int -> unit

Sockets

val socket : Unix.socket_domain -> Unix.socket_type -> int -> 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 * Unix.sockaddr
val sendto :
  Unix.file_descr ->
  string -> int -> int -> Unix.msg_flag list -> Unix.sockaddr -> int
val open_connection :
  Unix.sockaddr -> Pervasives.in_channel * Pervasives.out_channel
Chapter 25

The graphics library

The graphics library provides a set of portable drawing primitives. Drawing takes place in a separate window that is created when Graphics.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 other options graphics.cma other files
```

For interactive use of the graphics library, do:

```
ocamlmktop -o mytop graphics.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type

```
#load "graphics.cma";;
```

Here are the graphics mode specifications supported by Graphics.open_graph on the X11 implementation of this library: the argument to Graphics.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:

```
Graphics.open_graph "foo:0"
    connects to the display foo:0 and creates a window with the default geometry

Graphics.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)

Graphics.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)

Graphics.open_graph ""
    connects to the default display and creates a window with the default geometry.
```
Windows:
This library is available both for standalone compiled programs and under the toplevel application `ocamlwin.exe`. For the latter, this library must be loaded in-core by typing

```
#load "graphics.cma";;
```

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.

![Screen diagram](image)

### 25.1 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 set_window_title : string -> unit**

Set the title of the graphics window.

**val resize_window : int -> int -> unit**
Resize and erase the graphics window.

val clear_graph : unit -> unit
  Erase the graphics window.

val size_x : unit -> int
  See Graphics.size_y\[25.1\].

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 background : color
  See Graphics.foreground\[25.1\].

val foreground : color
  Default background and foreground colors (usually, either black foreground on a white background or white foreground on a black background). Graphics.clear_graph\[25.1\] fills the screen with the background color. The initial drawing color is foreground.

Some predefined colors

val black : color
val white : color
val red : color
val green : color
val blue : color
val yellow : color
val cyan : color
val magenta : color
Point and line drawing

val plot : int -> int -> unit
  Plot the given point with the current drawing color.

val plots : (int * int) array -> unit
  Plot the given points with the current drawing color.

val point_color : int -> int -> color
  Return the color of the given point in the backing store (see "Double buffering" below).

val moveto : int -> int -> unit
  Position the current point.

val rmoveto : int -> int -> unit
  rmoveto dx dy translates the current point by the given vector.

val current_x : unit -> int
  Return the abscissa of the current point.

val current_y : unit -> int
  Return the ordinate of 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 rlineto : int -> int -> unit
  Draw a line with endpoints the current point and the current point translated of the given vector, and move the current point to this point.

val curveto : int * int -> int * int -> int * int -> unit
  curveto b c d draws a cubic Bezier curve starting from the current point to point d, with control points b and c, and moves the current point to d.

val draw_rect : int -> int -> int -> int -> unit
  draw_rect x y w h draws the rectangle with lower left corner at x,y, width w and height h. The current point is unchanged. Raise Invalid_argument if w or h is negative.

val draw_poly_line : (int * int) array -> unit
draw_poly_line points draws the line that joins the points given by the array argument. The array contains the coordinates of the vertices of the polygonal line, which need not be closed. The current point is unchanged.

val draw_poly : (int * int) array -> unit
draw_poly polygon draws the given polygon. The array contains the coordinates of the vertices of the polygon. The current point is unchanged.

val draw_segments : (int * int * int * int) array -> unit
draw_segments segments draws the segments given in the array argument. Each segment is specified as a quadruple \((x_0, y_0, x_1, y_1)\) where \((x_0, y_0)\) and \((x_1, y_1)\) are the coordinates of the end points of the segment. The current point is unchanged.

val draw_arc : int -> int -> int -> int -> int -> int -> unit
draw_arc \(x\) \(y\) \(rx\) \(ry\) \(a_1\) \(a_2\) draws an elliptical arc with center \((x,y)\), horizontal radius \(rx\), vertical radius \(ry\), from angle \(a_1\) to angle \(a_2\) (in degrees). The current point is unchanged. Raise Invalid_argument if \(rx\) or \(ry\) is negative.

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. Raise Invalid_argument if \(rx\) or \(ry\) is negative.

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. Raise Invalid_argument if \(r\) is negative.

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. Raise Invalid_argument if the argument is negative.

Text drawing

val draw_char : char -> unit
See Graphics.draw_string[25.1].

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
Set the font used for drawing text. The interpretation of the argument to `set_font` is implementation-dependent.

```ocaml
val set_text_size : int -> unit
```

Set the character size used for drawing text. The interpretation of the argument to `set_text_size` is implementation-dependent.

```ocaml
val text_size : string -> int * int
```

Return the dimensions of the given text, if it were drawn with the current font and size.

### Filling

```ocaml
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. Raise Invalid_argument if w or h is negative.
```

```ocaml
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.
```

```ocaml
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 Graphics.draw_arc[25.1].
```

```ocaml
val fill_ellipse : int -> int -> int -> int -> unit
```

```
Fill an ellipse with the current color. The parameters are the same as for Graphics.draw_ellipse[25.1].
```

```ocaml
val fill_circle : int -> int -> int -> unit
```

```
Fill a circle with the current color. The parameters are the same as for Graphics.draw_circle[25.1].
```

### Images

```ocaml
type image
```

```
The abstract type for images, in internal representation. Externally, images are represented as matrices of colors.
```

```ocaml
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.
```
Chapter 25. The graphics library

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 Graphics.fill_rect.\[25.1\].

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).
Double buffering

val auto_synchronize : bool -> unit

By default, drawing takes place both on the window displayed on screen, and in a memory area (the “backing store”). The backing store image is used to re-paint the on-screen window when necessary.

To avoid flicker during animations, it is possible to turn off on-screen drawing, perform a number of drawing operations in the backing store only, then refresh the on-screen window explicitly.

auto_synchronize false turns on-screen drawing off. All subsequent drawing commands are performed on the backing store only.

auto_synchronize true refreshes the on-screen window from the backing store (as per synchronize), then turns on-screen drawing back on. All subsequent drawing commands are performed both on screen and in the backing store.

The default drawing mode corresponds to auto_synchronize true.

val synchronize : unit -> unit

Synchronize the backing store and the on-screen window, by copying the contents of the backing store onto the graphics window.

val display_mode : bool -> unit

Set display mode on or off. When turned on, drawings are done in the graphics window; when turned off, drawings do not affect the graphics window. This occurs independently of drawing into the backing store (see the function Graphics.remember_mode[25.1] below). Default display mode is on.

val remember_mode : bool -> unit

Set remember mode on or off. When turned on, drawings are done in the backing store; when turned off, the backing store is unaffected by drawings. This occurs independently of drawing onto the graphics window (see the function Graphics.display_mode[25.1] above). Default remember mode is on.
Chapter 26

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 as follows:

    ocamlc  other options  dbm.cma  other files
    ocamlopt other options  dbm.cmxa  other files

For interactive use of the dbm library, do:

    ocamlmktop -o mytop  dbm.cma
    ./mytop

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type 

    #load "dbm.cma";;

Windows:

This library is not available.

26.1 Module Dbm : Interface to the NDBM database.

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 Dbm.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

See Dbm.nextkey.

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 27

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, or of native plugins (usually .cmxs files) in a running native 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. In native code, there are also some compatibility checks on the implementations (to avoid errors with cross-module optimizations); it might be useful to hide .cmx files when building native plugins so that they remain independent of the implementation of modules in the main program.

Programs that use the dynlink library simply need to link dynlink.cma or dynlink.cmxa with their object files and other libraries.

27.1 Module Dynlink : Dynamic loading of object files.

val is_native : bool
  true if the program is native, false if the program is bytecode.

Dynamic loading of compiled files

val loadfile : string -> unit
  In bytecode: load the given bytecode object file (.cmo file) or bytecode library file (.cma file), and link it with the running program. In native code: load the given OCaml plugin file (usually .cmxs), and link it with the running program. All toplevel expressions in the loaded compilation units 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 compilation units just loaded are hidden (cannot be referenced) from other modules dynamically loaded afterwards.

```ocaml

val adapt_filename : string -> string

In bytecode, the identity function. In native code, replace the last extension with `.cmxs`.

Access control

val allow_only : string list -> unit

allow_only units restricts the compilation units that dynamically-linked units can reference: it only allows references to the units named in list `units`. References to any other compilation unit will cause a `Unavailable_unit` error during `loadfile` or `loadfile_private`.

Initially (just after calling `init`), all compilation units composing the program currently running are available for reference from dynamically-linked units. `allow_only` can be used to grant access to some of them only, e.g. to the units that compose the API for dynamically-linked code, and prevent access to all other units, e.g. private, internal modules of the running program.

val prohibit : string list -> unit

prohibit units prohibits dynamically-linked units from referencing the units named in list `units`. This can be used to prevent access to selected units, e.g. private, internal modules of the running program.

val default_available_units : unit -> unit

Reset the set of units that can be referenced from dynamically-linked code to its default value, that is, all units composing the currently running program.

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. In native code, this function does nothing; object files with external functions are always allowed to be dynamically linked.

Deprecated, low-level API for access control

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).

val add_available_units : (string * Digest.t) list -> unit
Same as `Dynlink.add_interfaces[27.1]`, 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.

```ocaml
val clear_available_units : unit -> unit
  Empty the list of compilation units accessible to dynamically-linked programs.
```

**Deprecated, initialization**

```ocaml
val init : unit -> unit
  Deprecated. Initialize the Dynlink library. This function is called automatically when needed.
```

**Error reporting**

```ocaml
type linking_error =
  | Undefined_global of string
  | Unavailable_primitive of string
  | Uninitialized_global 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
  | File_not_found of string
  | Cannot_open_dll of string
  | Inconsistent_implementation of string
exception Error of error
  Errors in dynamic linking are reported by raising the Error exception with a description of the error.
```

```ocaml
val error_message : error -> string
  Convert an error description to a printable message.
```
Chapter 28

The LablTk library: Tcl/Tk GUI interface

The labltk library provides access to the Tcl/Tk GUI from OCaml programs. This interface is generated in an automated way, and you should refer to Tcl/Tk books and man pages for detailed information on the behavior of the numerous functions. We also suggest to use ocamlbrowser to see the types of the various functions, that are the best documentation for the library itself.

Programs that use the labltk library must be linked as follows:

```plaintext
ocamlc other options -I +labltk labltk.cma other files
ocamlopt other options -I +labltk labltk.cmxa other files
```

Unix:

The labltk library is available for any system with Tcl/Tk installed, starting from Tcl 7.5/Tk 4.1 up to Tcl/Tk 8.4. Beware that some beta versions may have compatibility problems.

If the library was not compiled correctly, try to run again the configure script with the option `-tkdefs switches`, where switches is a list of C-style inclusion paths leading to the right tcl.h and tk.h, for instance `-I/usr/local/include/tcl8.4 -I/usr/local/include/tk8.4`.

A script is installed, to make easier the use of the labltk library as toplevel.

```plaintext
labltk
```

This is a toplevel including the labltk library, and the path is already set as to allow the use of the various modules. It also includes code for the Unix and Str libraries. You can use it in place of ocaml.

Windows:

The labltk library has been precompiled for use with Tcl/Tk 8.4. You must first have it installed on your system. It can be downloaded from http://www.activestate.com/products/ActiveTcl/. After installing it, you must put the dynamically loaded libraries tcl84.dll and tk84.dll (from the bin directory of the Tcl installation) in a directory included in your path.

No toplevel is available, but you can load the library from the standard toplevel with the following commands.
You can also load it directly from the command line.

C:\ocaml\bin> ocaml -I +labltk labltk.cma

The labltk library is composed of a large number of modules.

<table>
<thead>
<tr>
<th>Bell</th>
<th>Imagebitmap</th>
<th>Place</th>
</tr>
</thead>
<tbody>
<tr>
<td>Button</td>
<td>Imagephoto</td>
<td>Radiobutton</td>
</tr>
<tr>
<td>Canvas</td>
<td>Label</td>
<td>Scale</td>
</tr>
<tr>
<td>Checkbutton</td>
<td>Listbox</td>
<td>Scrollbar</td>
</tr>
<tr>
<td>Clipboard</td>
<td>Menu</td>
<td>Selection</td>
</tr>
<tr>
<td>Dialog</td>
<td>Menubutton</td>
<td>Text</td>
</tr>
<tr>
<td>Entry</td>
<td>Message</td>
<td>Tk</td>
</tr>
<tr>
<td>Focus</td>
<td>Option</td>
<td>Tkwait</td>
</tr>
<tr>
<td>Frame</td>
<td>Optionmenu</td>
<td>Toplevel</td>
</tr>
<tr>
<td>Grab</td>
<td>Pack</td>
<td>Winfo</td>
</tr>
<tr>
<td>Grid</td>
<td>Palette</td>
<td>Wm</td>
</tr>
</tbody>
</table>

Giving a detailed account of each of these module would be impractical here. We will just present some of the basic functions in the module Tk. Note that for most other modules information can be found in the Tcl man page of their name.

### 28.1 Module Tk : Basic functions and types for LablTk

#### Initialization and termination

```ocaml
val openTk : ?display:string -> ?clas:string -> unit -> Widget.toplevel Widget.widget
```

Initialize LablTk and open a toplevel window. display is described according to the X11 conventions. clas is used for the X11 resource mechanism.

```ocaml
val mainLoop : unit -> unit
```

Start the main event loop

```ocaml
val closeTk : unit -> unit
```

Quit the main loop and close all open windows.

```ocaml
val destroy : 'a Widget.widget -> unit
```

Destroy an individual widget.
Chapter 28. The LablTk library: Tcl/Tk GUI interface

Application wide commands

val update : unit -> unit
Synchronize display with internal state.

val appname_get : unit -> string
val appname_set : string -> unit
Get or set the application name.

Dimensions

type units = [ ‘Cm of float | ‘In of float | ‘Mm of float | ‘Pix of int | ‘Pt of float ]
val pixels : units -> int
Converts various on-screen units to pixels, respective to the default display. Available units
are pixels, centimeters, inches, millimeters and points

Widget layout commands

type anchor = [ ‘Center | ‘E | ‘N | ‘Ne | ‘Nw | ‘S | ‘Se | ‘Sw | ‘W ]
type fillMode = [ ‘Both | ‘None | ‘X | ‘Y ]
type side = [ ‘Bottom | ‘Left | ‘Right | ‘Top ]
val pack :
?after:'a Widget.widget ->
?anchor:anchor ->
?before:'b Widget.widget ->
?expand:bool ->
?fill:fillMode ->
?inside:'c Widget.widget ->
?ipadx:int ->
?ipady:int ->
Pack a widget inside its parent, using the standard layout engine.

val grid :
?column:int ->
?columnspan:int ->
?inside:'a Widget.widget ->
?ipadx:int ->
?ipady:int ->
?padx:int ->
?pady:int ->
Pack a widget inside its parent, using the grid layout engine.
type borderMode = [ 'Ignore | 'Inside | 'Outside ]

val place :
  ?anchor:anchor ->
  ?bordermode:borderMode ->
  ?height:int ->
  ?inside:'a Widget.widget ->
  ?relheight:float ->
  ?relwidth:float ->
  ?relx:float ->
  ?rely:float ->
  ?width:int ->
  ?x:int ->
  ?y:int ->
  'b Widget.widget -> unit

  Pack a widget inside its parent, at absolute coordinates.

val raise_window : ?above:'a Widget.widget ->
  'b Widget.widget -> unit

val lower_window : ?below:'a Widget.widget ->
  'b Widget.widget -> unit

  Raise or lower the window associated to a widget.

Event handling

type modifier = [ 'Alt
  | 'Button1
  | 'Button2
  | 'Button3
  | 'Button4
  | 'Button5
  | 'Control
  | 'Double
  | 'Lock
  | 'Meta
  | 'Mod1
  | 'Mod2
  | 'Mod3
  | 'Mod4
  | 'Mod5
  | 'Shift
  | 'Triple ]

type event = [ 'ButtonPress
  | 'ButtonPressDetail of int
  | 'ButtonRelease
  | 'ButtonReleaseDetail of int
  | 'Circulate
  | 'ColorMap
  | 'Configure
  | 'Destroy
  | 'Enter

Chapter 28. The LablTk library: Tcl/Tk GUI interface

| ‘Expose    |
| ‘FocusIn   |
| ‘FocusOut  |
| ‘Gravity   |
| ‘KeyPress  |
| ‘KeyPressDetail of string |
| ‘KeyRelease|
| ‘KeyReleaseDetail of string |
| ‘Leave     |
| ‘Map       |
| ‘Modified of modifier list * event |
| ‘Motion    |
| ‘Property  |
| ‘Reparent  |
| ‘Unmap     |
| ‘Visibility ]

An event can be either a basic X event, or modified by a key or mouse modifier.

type eventInfo = {
    mutable ev_Above : int ;
    mutable ev_ButtonNumber : int ;
    mutable ev_Count : int ;
    mutable ev_Detail : string ;
    mutable ev_Focus : bool ;
    mutable ev_Height : int ;
    mutable ev_KeyDown : int ;
    mutable ev_Mode : string ;
    mutable ev_OverrideRedirect : bool ;
    mutable ev_Place : string ;
    mutable ev_State : string ;
    mutable ev_Time : int ;
    mutable ev_Width : int ;
    mutable ev_MouseX : int ;
    mutable ev_MouseY : int ;
    mutable ev_Char : string ;
    mutable ev_BorderWidth : int ;
    mutable ev_SendEvent : bool ;
    mutable ev_KeySymString : string ;
    mutable ev_KeySymInt : int ;
    mutable ev_RootWindow : int ;
    mutable ev_SubWindow : int ;
    mutable ev_Type : int ;
    mutable ev_Widget : Widget.any Widget.widget ;
    mutable ev_RootX : int ;
    mutable ev_RootY : int ;
}
Event related information accessible in callbacks.

```haskell
type eventField = [ 'Above |
| 'BorderWidth |
| 'ButtonNumber |
| 'Char |
| 'Count |
| 'Detail |
| 'Focus |
| 'Height |
| 'KeyCode |
| 'KeySymInt |
| 'KeySymString |
| 'Mode |
| 'MouseX |
| 'MouseY |
| 'OverrideRedirect |
| 'Place |
| 'RootWindow |
| 'RootX |
| 'RootY |
| 'SendEvent |
| 'State |
| 'SubWindow |
| 'Time |
| 'Type |
| 'Widget |
| 'Width ]
```

In order to access the above event information, one has to pass a list of required event fields to
the `bind` function.

```haskell
val bind :
  events: event list ->
  ?extend: bool ->
  ?breakable: bool ->
  ?fields: eventField list ->
  ?action: (eventInfo -> unit) -> 'a Widget.widget -> unit

  Bind a succession of events on a widget to an action. If extend is true then then binding
  is added after existing ones, otherwise it replaces them. breakable should be true when
  break is to be called inside the action. action is called with the fields required set in an
  eventInfo structure. Other fields should not be accessed. If action is omitted then
  existing bindings are removed.
```

```haskell
val bind_class :
  events: event list ->
  ?extend: bool ->
  ?breakable: bool ->
```
Same thing for all widgets of a given class. If a widget is given with label `on:`, the binding will be removed as soon as it is destroyed.

```
val bind_tag :
  events:event list ->
  ?extend:bool ->
  ?breakable:bool ->
  ?fields:eventField list ->
  ?action:(eventInfo -> unit) -> ?on:'a Widget.widget -> string -> unit

  Same thing for all widgets having a given tag
```

```
val break : unit -> unit

  Used inside a bound action, do not call other actions after this one. This is only possible if this action was bound with `breakable: true`.
```
Chapter 29

The bigarray library

The bigarray library implements large, multi-dimensional, numerical arrays. These arrays are called “big arrays” to distinguish them from the standard Caml arrays described in section 20.2. The main differences between “big arrays” and standard Caml arrays are as follows:

- Big arrays are not limited in size, unlike Caml arrays (float array are limited to 2097151 elements on a 32-bit platform, other array types to 4194303 elements).

- Big arrays are multi-dimensional. Any number of dimensions between 1 and 16 is supported. In contrast, Caml arrays are mono-dimensional and require encoding multi-dimensional arrays as arrays of arrays.

- Big arrays can only contain integers and floating-point numbers, while Caml arrays can contain arbitrary Caml data types. However, big arrays provide more space-efficient storage of integer and floating-point elements, in particular because they support “small” types such as single-precision floats and 8 and 16-bit integers, in addition to the standard Caml types of double-precision floats and 32 and 64-bit integers.

- The memory layout of big arrays is entirely compatible with that of arrays in C and Fortran, allowing large arrays to be passed back and forth between Caml code and C / Fortran code with no data copying at all.

- Big arrays support interesting high-level operations that normal arrays do not provide efficiently, such as extracting sub-arrays and “slicing” a multi-dimensional array along certain dimensions, all without any copying.

Programs that use the bigarray library must be linked as follows:

```
  ocamlc other options bigarray.cma other files
  ocamlopt other options bigarray.cmxa other files
```

For interactive use of the bigarray library, do:

```
  ocamlmktop -o mytop bigarray.cma
  ./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type #load "bigarray.cma";;.
29.1 Module Bigarray: Large, multi-dimensional, numerical arrays.

This module implements multi-dimensional arrays of integers and floating-point numbers, thereafter referred to as “big arrays”. The implementation allows efficient sharing of large numerical arrays between Caml code and C or Fortran numerical libraries.

Concerning the naming conventions, users of this module are encouraged to do open Bigarray in their source, then refer to array types and operations via short dot notation, e.g. Array1.t or Array2.sub.

Big arrays support all the Caml ad-hoc polymorphic operations:

- comparisons (=, <>, <=, etc, as well as Pervasives.compare[19.2]);
- hashing (module Hash);
- and structured input-output (Pervasives.output_value[19.2] and Pervasives.input_value[19.2], as well as the functions from the Marshal[20.19] module).

Element kinds

Big arrays can contain elements of the following kinds:

- IEEE single precision (32 bits) floating-point numbers (Bigarray.float32_elt[29.1]),
- IEEE double precision (64 bits) floating-point numbers (Bigarray.float64_elt[29.1]),
- IEEE single precision (2 * 32 bits) floating-point complex numbers (Bigarray.complex32_elt[29.1]),
- IEEE double precision (2 * 64 bits) floating-point complex numbers (Bigarray.complex64_elt[29.1]),
- 8-bit integers (signed or unsigned) (Bigarray.int8_signed_elt[29.1] or Bigarray.int8_unsigned_elt[29.1]),
- 16-bit integers (signed or unsigned) (Bigarray.int16_signed_elt[29.1] or Bigarray.int16_unsigned_elt[29.1]),
- Caml integers (signed, 31 bits on 32-bit architectures, 63 bits on 64-bit architectures) (Bigarray.int_elt[29.1]),
- 32-bit signed integer (Bigarray.int32_elt[29.1]),
- 64-bit signed integers (Bigarray.int64_elt[29.1]),
- platform-native signed integers (32 bits on 32-bit architectures, 64 bits on 64-bit architectures) (Bigarray.nativeint_elt[29.1]).

Each element kind is represented at the type level by one of the abstract types defined below.

```plaintext
type float32_elt
type float64_elt
type complex32_elt```


type complex64_elt
type int8_signed_elt
type int8_unsigned_elt
type int16_signed_elt
type int16_unsigned_elt
type int elt
type int32 elt
type int64 elt
type nativeint elt

To each element kind is associated a Caml type, which is the type of Caml values that can be stored in the big array or read back from it. This type is not necessarily the same as the type of the array elements proper: for instance, a big array whose elements are of kind float32_elt contains 32-bit single precision floats, but reading or writing one of its elements from Caml uses the Caml type float, which is 64-bit double precision floats.

The abstract type ('a, 'b) kind captures this association of a Caml type 'a for values read or written in the big array, and of an element kind 'b which represents the actual contents of the big array. The following predefined values of type kind list all possible associations of Caml types with element kinds:

val float32 : (float, float32_elt) kind
  See Bigarray.char[29.1].

val float64 : (float, float64_elt) kind
  See Bigarray.char[29.1].

val complex32 : (Complex.t, complex32_elt) kind
  See Bigarray.char[29.1].

val complex64 : (Complex.t, complex64_elt) kind
  See Bigarray.char[29.1].

val int8_signed : (int, int8_signed_elt) kind
  See Bigarray.char[29.1].

val int8_unsigned : (int, int8_unsigned_elt) kind
  See Bigarray.char[29.1].

val int16_signed : (int, int16_signed_elt) kind
  See Bigarray.char[29.1].

val int16_unsigned : (int, int16_unsigned_elt) kind
See `Bigarray.char[29.1].`

val int : (int, int_elt) kind
See `Bigarray.char[29.1].`

val int32 : (int32, int32_elt) kind
See `Bigarray.char[29.1].`

val int64 : (int64, int64_elt) kind
See `Bigarray.char[29.1].`

val nativeint : (nativeint, nativeint_elt) kind
See `Bigarray.char[29.1].`

val char : (char, int8_unsigned_elt) kind
As shown by the types of the values above, big arrays of kind `float32_elt` and `float64_elt` are accessed using the Caml type `float`. Big arrays of complex kinds `complex32_elt`, `complex64_elt` are accessed with the Caml type `Complex.t[20.6]`. Big arrays of integer kinds are accessed using the smallest Caml integer type large enough to represent the array elements: `int` for 8- and 16-bit integer bigarrays, as well as Caml-integer bigarrays; `int32` for 32-bit integer bigarrays; `int64` for 64-bit integer bigarrays; and `nativeint` for platform-native integer bigarrays. Finally, big arrays of kind `int8_unsigned_elt` can also be accessed as arrays of characters instead of arrays of small integers, by using the kind value `char` instead of `int8_unsigned`.

Array layouts

type c_layout
See `Bigarray.fortran_layout[29.1].`

type fortran_layout
To facilitate interoperability with existing C and Fortran code, this library supports two different memory layouts for big arrays, one compatible with the C conventions, the other compatible with the Fortran conventions.

In the C-style layout, array indices start at 0, and multi-dimensional arrays are laid out in row-major format. That is, for a two-dimensional array, all elements of row 0 are contiguous in memory, followed by all elements of row 1, etc. In other terms, the array elements at \((x,y)\) and \((x, y+1)\) are adjacent in memory.

In the Fortran-style layout, array indices start at 1, and multi-dimensional arrays are laid out in column-major format. That is, for a two-dimensional array, all elements of column 0 are contiguous in memory, followed by all elements of column 1, etc. In other terms, the array elements at \((x,y)\) and \((x+1, y)\) are adjacent in memory.

Each layout style is identified at the type level by the abstract types `Bigarray.c_layout[29.1]` and `fortran_layout` respectively.
Chapter 29. The bigarray library

491

type 'a layout

The type 'a layout represents one of the two supported memory layouts: C-style if 'a is
Bigarray.c_layout[29.1], Fortran-style if 'a is Bigarray.fortran_layout[29.1].

Supported layouts

The abstract values c_layout and fortran_layout represent the two supported layouts at the
level of values.

val c_layout : c_layout layout
val fortran_layout : fortran_layout layout

Generic arrays (of arbitrarily many dimensions)

module Genarray :

  sig

    type ('a, 'b, 'c) t

    The type Genarray.t is the type of big arrays with variable numbers of dimensions.
    Any number of dimensions between 1 and 16 is supported.
    The three type parameters to Genarray.t identify the array element kind and layout,
    as follows:
    • the first parameter, 'a, is the Caml type for accessing array elements (float, int,
      int32, int64, nativeint);
    • the second parameter, 'b, is the actual kind of array elements (float32_elt,  
      float64_elt, int8_signed_elt, int8_unsigned_elt, etc);
    • the third parameter, 'c, identifies the array layout (c_layout or fortran_layout).
    For instance, (float, float32_elt, fortran_layout) Genarray.t is the type of
    generic big arrays containing 32-bit floats in Fortran layout; reads and writes in this
    array use the Caml type float.

  val create :

    ('a, 'b) Bigarray.kind ->
    'c Bigarray.layout -> int array -> ('a, 'b, 'c) t

    Genarray.create kind layout dimensions returns a new big array whose element
    kind is determined by the parameter kind (one of float32, float64, int8_signed,
    etc) and whose layout is determined by the parameter layout (one of c_layout or
    fortran_layout). The dimensions parameter is an array of integers that indicate the
    size of the big array in each dimension. The length of dimensions determines the
    number of dimensions of the bigarray.
    For instance, Genarray.create int32 c_layout [|4;6;8|] returns a fresh big array
    of 32-bit integers, in C layout, having three dimensions, the three dimensions being 4, 6
    and 8 respectively.
Big arrays returned by `Genarray.create` are not initialized: the initial values of array elements is unspecified. `Genarray.create` raises `Invalid_argument` if the number of dimensions is not in the range 1 to 16 inclusive, or if one of the dimensions is negative.

```ocaml
val num_dims : ('a, 'b, 'c) t -> int

Return the number of dimensions of the given big array.
```

```ocaml
val dims : ('a, 'b, 'c) t -> int array

Genarray.dims a returns all dimensions of the big array a, as an array of integers of length Genarray.num_dims a.
```

```ocaml
val nth_dim : ('a, 'b, 'c) t -> int -> int

Genarray.nth_dim a n returns the n-th dimension of the big array a. The first dimension corresponds to n = 0; the second dimension corresponds to n = 1; the last dimension, to n = Genarray.num_dims a - 1. Raise `Invalid_argument` if n is less than 0 or greater or equal than Genarray.num_dims a.
```

```ocaml
val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind

Return the kind of the given big array.
```

```ocaml
val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout

Return the layout of the given big array.
```

```ocaml
val get : ('a, 'b, 'c) t -> int array -> 'a

Read an element of a generic big array. Genarray.get a [|i1; ...; iN|] returns the element of a whose coordinates are i1 in the first dimension, i2 in the second dimension, ..., iN in the N-th dimension.

If a has C layout, the coordinates must be greater or equal than 0 and strictly less than the corresponding dimensions of a. If a has Fortran layout, the coordinates must be greater or equal than 1 and less or equal than the corresponding dimensions of a. Raise `Invalid_argument` if the array a does not have exactly N dimensions, or if the coordinates are outside the array bounds.

If N > 3, alternate syntax is provided: you can write `a.{i1, i2, ..., iN}` instead of `Genarray.get a [|i1; ...; iN|]`. (The syntax `a.{...}` with one, two or three coordinates is reserved for accessing one-, two- and three-dimensional arrays as described below.)
```

```ocaml
val set : ('a, 'b, 'c) t -> int array -> 'a -> unit

Assign an element of a generic big array. Genarray.set a [|i1; ...; iN|] v stores the value v in the element of a whose coordinates are i1 in the first dimension, i2 in the second dimension, ..., iN in the N-th dimension.
```

The array $a$ must have exactly $N$ dimensions, and all coordinates must lie inside the array bounds, as described for Genarray.get; otherwise, Invalid_argument is raised.

If $N > 3$, alternate syntax is provided: you can write $a.\{i1, i2, \ldots, iN\} \leftarrow v$ instead of Genarray.set $a \[|i1; \ldots; iN|\] v. (The syntax $a.\{\ldots\} \leftarrow v$ with one, two or three coordinates is reserved for updating one-, two- and three-dimensional arrays as described below.)

val sub_left :
    ('a, 'b, Bigarray.c_layout) t ->
    int -> int -> ('a, 'b, Bigarray.c_layout) t

Extract a sub-array of the given big array by restricting the first (left-most) dimension. Genarray.sub_left $a$ ofs len returns a big array with the same number of dimensions as $a$, and the same dimensions as $a$, except the first dimension, which corresponds to the interval $[ofs ... ofs + len - 1]$ of the first dimension of $a$. No copying of elements is involved: the sub-array and the original array share the same storage space. In other terms, the element at coordinates $[|i1; \ldots; iN|]$ of the sub-array is identical to the element at coordinates $[|i1+ofs; \ldots; iN|]$ of the original array $a$.

Genarray.sub_left applies only to big arrays in C layout. Raise Invalid_argument if ofs and len do not designate a valid sub-array of $a$, that is, if ofs < 0, or len < 0, or ofs + len > Genarray.nth_dim $a$ 0.

val sub_right :
    ('a, 'b, Bigarray.fortran_layout) t ->
    int -> int -> ('a, 'b, Bigarray.fortran_layout) t

Extract a sub-array of the given big array by restricting the last (right-most) dimension. Genarray.sub_right $a$ ofs len returns a big array with the same number of dimensions as $a$, and the same dimensions as $a$, except the last dimension, which corresponds to the interval $[ofs ... ofs + len - 1]$ of the last dimension of $a$. No copying of elements is involved: the sub-array and the original array share the same storage space. In other terms, the element at coordinates $[|i1; \ldots; iN|]$ of the sub-array is identical to the element at coordinates $[|i1+ofs; \ldots; iN|]$ of the original array $a$.

Genarray.sub_right applies only to big arrays in Fortran layout. Raise Invalid_argument if ofs and len do not designate a valid sub-array of $a$, that is, if ofs < 1, or len < 0, or ofs + len > Genarray.nth_dim $a$ (Genarray.num_dims $a$ - 1).

val slice_left :
    ('a, 'b, Bigarray.c_layout) t ->
    int array -> ('a, 'b, Bigarray.c_layout) t

Extract a sub-array of lower dimension from the given big array by fixing one or several of the first (left-most) coordinates. Genarray.slice_left $a$ $[|i1; \ldots; iM|]$ returns the “slice” of $a$ obtained by setting the first $M$ coordinates to $i1, \ldots, iM$. If
a has $N$ dimensions, the slice has dimension $N - M$, and the element at coordinates $\langle i_1; \ldots; i_M; j_1; \ldots; j_{(N-M)} \rangle$ in the slice is identical to the element at coordinates $\langle i_1; \ldots; i_M; j_1; \ldots; j_{(N-M)} \rangle$ in the original array $a$. No copying of elements is involved: the slice and the original array share the same storage space.

Genarray.slice_left applies only to big arrays in C layout. Raise Invalid_argument if $M \geq N$, or if $\langle i_1; \ldots; i_M \rangle$ is outside the bounds of $a$.

val slice_right : 
('a, 'b, Bigarray.fortran_layout) t -> int array -> ('a, 'b, Bigarray.fortran_layout) t

Extract a sub-array of lower dimension from the given big array by fixing one or several of the last (right-most) coordinates. Genarray.slice_right $a \ [\langle i_1; \ldots; i_M \rangle \]$ returns the “slice” of $a$ obtained by setting the last $M$ coordinates to $i_1, \ldots, i_M$. If $a$ has $N$ dimensions, the slice has dimension $N - M$, and the element at coordinates $\langle j_1; \ldots; j_{(N-M)} \rangle$ in the slice is identical to the element at coordinates $\langle i_1; \ldots; i_M; j_1; \ldots; j_{(N-M)} \rangle$ in the original array $a$. No copying of elements is involved: the slice and the original array share the same storage space.

Genarray.slice_right applies only to big arrays in Fortran layout. Raise Invalid_argument if $M \geq N$, or if $\langle i_1; \ldots; i_M \rangle$ is outside the bounds of $a$.

val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit

Copy all elements of a big array in another big array. Genarray.blit $src$ $dst$ copies all elements of $src$ into $dst$. Both arrays $src$ and $dst$ must have the same number of dimensions and equal dimensions. Copying a sub-array of $src$ to a sub-array of $dst$ can be achieved by applying Genarray.blit to sub-array or slices of $src$ and $dst$.

val fill : ('a, 'b, 'c) t -> 'a -> unit

Set all elements of a big array to a given value. Genarray.fill $a$ $v$ stores the value $v$ in all elements of the big array $a$. Setting only some elements of $a$ to $v$ can be achieved by applying Genarray.fill to a sub-array or a slice of $a$.

val map_file :
Unix.file_descr ->
?pos:int64 ->
('a, 'b) Bigarray.kind ->
'c Bigarray.layout -> bool -> int array -> ('a, 'b, 'c) t

Memory mapping of a file as a big array. Genarray.map_file $fd$ $kind$ $layout$ shared $dims$ returns a big array of kind $kind$, layout $layout$, and dimensions as specified in $dims$. The data contained in this big array are the contents of the file referred to by the file descriptor $fd$ (as opened previously with Unix.openfile, for example). The optional pos parameter is the byte offset in the file of the data being mapped; it defaults to 0 (map from the beginning of the file).

If shared is true, all modifications performed on the array are reflected in the file. This requires that $fd$ be opened with write permissions. If shared is false,
modifications performed on the array are done in memory only, using copy-on-write of the modified pages; the underlying file is not affected.

`Genarray.map_file` is much more efficient than reading the whole file in a big array, modifying that big array, and writing it afterwards.

To adjust automatically the dimensions of the big array to the actual size of the file, the major dimension (that is, the first dimension for an array with C layout, and the last dimension for an array with Fortran layout) can be given as `-1`. `Genarray.map_file` then determines the major dimension from the size of the file.

The file must contain an integral number of sub-arrays as determined by the non-major dimensions, otherwise `Failure` is raised.

If all dimensions of the big array are given, the file size is matched against the size of the big array. If the file is larger than the big array, only the initial portion of the file is mapped to the big array. If the file is smaller than the big array, the file is automatically grown to the size of the big array. This requires write permissions on `fd`.

One-dimensional arrays

```ocaml
module Array1 :
    sig
        type ('a, 'b, 'c) t
            The type of one-dimensional big arrays whose elements have Caml type 'a, representation kind 'b, and memory layout 'c.

        val create : ('a, 'b) Bigarray.kind ->
                        'c Bigarray.layout -> int -> ('a, 'b, 'c) t
                        Array1.create kind layout dim returns a new bigarray of one dimension, whose size is dim. kind and layout determine the array element kind and the array layout as described for Genarray.create.

        val dim : ('a, 'b, 'c) t -> int
                        Return the size (dimension) of the given one-dimensional big array.

        val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
                        Return the kind of the given big array.

        val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
                        Return the layout of the given big array.

        val get : ('a, 'b, 'c) t -> int -> 'a
```
Array1.get a x, or alternatively a.\{x\}, returns the element of a at index x. x must be greater or equal than 0 and strictly less than Array1.dim a if a has C layout. If a has Fortran layout, x must be greater or equal than 1 and less or equal than Array1.dim a. Otherwise, Invalid_argument is raised.

val set : ('a, 'b, 'c) t -> int -> 'a -> unit

Array1.set a x v, also written a.\{x\} <- v, stores the value v at index x in a. x must be inside the bounds of a as described in Bigarray.Array1.get\[29.1\]; otherwise, Invalid_argument is raised.

val sub : ('a, 'b, 'c) t ->
  int -> int -> ('a, 'b, 'c) t

Extract a sub-array of the given one-dimensional big array. See Genarray.sub_left for more details.

val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit

Copy the first big array to the second big array. See Genarray.blit for more details.

val fill : ('a, 'b, 'c) t -> 'a -> unit

Fill the given big array with the given value. See Genarray.fill for more details.

val of_array :
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> 'a array -> ('a, 'b, 'c) t

Build a one-dimensional big array initialized from the given array.

val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> bool -> int -> ('a, 'b, 'c) t

Memory mapping of a file as a one-dimensional big array. See Bigarray.Genarray.map_file\[29.1\] for more details.

val unsafe_get : ('a, 'b, 'c) t -> int -> 'a

Like Bigarray.Array1.get\[29.1\], but bounds checking is not always performed. Use with caution and only when the program logic guarantees that the access is within bounds.

val unsafe_set : ('a, 'b, 'c) t -> int -> 'a -> unit

Like Bigarray.Array1.set\[29.1\], but bounds checking is not always performed. Use with caution and only when the program logic guarantees that the access is within bounds.
end

One-dimensional arrays. The `Array1` structure provides operations similar to those of `Bigarray.Genarray`\[29.1\], but specialized to the case of one-dimensional arrays. (The `Array2` and `Array3` structures below provide operations specialized for two- and three-dimensional arrays.) Statically knowing the number of dimensions of the array allows faster operations, and more precise static type-checking.

Two-dimensional arrays

module Array2 :
  sig
    type ('a, 'b, 'c) t
      The type of two-dimensional big arrays whose elements have Caml type 'a, representation kind 'b, and memory layout 'c.

    val create :
      ('a, 'b) Bigarray.kind ->
      'c Bigarray.layout -> int -> int -> ('a, 'b, 'c) t
      Array2.create kind layout dim1 dim2 returns a new bigarray of two dimension, whose size is dim1 in the first dimension and dim2 in the second dimension. kind and layout determine the array element kind and the array layout as described for Bigarray.Genarray.create[29.1].

    val dim1 : ('a, 'b, 'c) t -> int
      Return the first dimension of the given two-dimensional big array.

    val dim2 : ('a, 'b, 'c) t -> int
      Return the second dimension of the given two-dimensional big array.

    val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
      Return the kind of the given big array.

    val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
      Return the layout of the given big array.

    val get : ('a, 'b, 'c) t -> int -> int -> 'a
      Array2.get a x y, also written a.\{x,y\}, returns the element of a at coordinates (x, y). x and y must be within the bounds of a, as described for Bigarray.Genarray.get[29.1]; otherwise, Invalid_argument is raised.

    val set : ('a, 'b, 'c) t -> int -> int -> 'a -> unit
Array2.set a x y v, or alternatively a.{x,y} <- v, stores the value v at coordinates (x, y) in a. x and y must be within the bounds of a, as described for Bigarray.Genarray.set\textsuperscript{[29.1]}; otherwise, Invalid_argument is raised.

val sub_left :
  ('a, 'b, Bigarray.c_layout) t ->
  int -> int -> ('a, 'b, Bigarray.c_layout) t

Extract a two-dimensional sub-array of the given two-dimensional big array by restricting the first dimension. See Bigarray.Genarray.sub_left\textsuperscript{[29.1]} for more details. Array2.sub_left applies only to arrays with C layout.

val sub_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> int -> ('a, 'b, Bigarray.fortran_layout) t

Extract a two-dimensional sub-array of the given two-dimensional big array by restricting the second dimension. See Bigarray.Genarray.sub_right\textsuperscript{[29.1]} for more details. Array2.sub_right applies only to arrays with Fortran layout.

val slice_left :
  ('a, 'b, Bigarray.c_layout) t ->
  int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array1.t

Extract a row (one-dimensional slice) of the given two-dimensional big array. The integer parameter is the index of the row to extract. See Bigarray.Genarray.slice_left\textsuperscript{[29.1]} for more details. Array2.slice_left applies only to arrays with C layout.

val slice_right :
  ('a, 'b, Bigarray.fortran_layout) t ->
  int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array1.t

Extract a column (one-dimensional slice) of the given two-dimensional big array. The integer parameter is the index of the column to extract. See Bigarray.Genarray.slice_right\textsuperscript{[29.1]} for more details. Array2.slice_right applies only to arrays with Fortran layout.

val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit

Copy the first big array to the second big array. See Bigarray.Genarray.blit\textsuperscript{[29.1]} for more details.

val fill : ('a, 'b, 'c) t -> 'a -> unit

Fill the given big array with the given value. See Bigarray.Genarray.fill\textsuperscript{[29.1]} for more details.
Chapter 29. The bigarray library

val of_array :  
  ('a, 'b) Bigarray.kind \rightarrow  
  'c Bigarray.layout \rightarrow  
  'a array array \rightarrow  
  ('a, 'b, 'c) t

  Build a two-dimensional big array initialized from the given array of arrays.

val map_file :  
  Unix.file_descr \rightarrow  
  ?pos:int64 \rightarrow  
  ('a, 'b) Bigarray.kind \rightarrow  
  'c Bigarray.layout \rightarrow  
  bool \rightarrow  
  int \rightarrow  
  int \rightarrow  
  ('a, 'b, 'c) t

  Memory mapping of a file as a two-dimensional big array. See
  Bigarray.Genarray.map_file\textsuperscript{[29.1]} for more details.

val unsafe_get : ('a, 'b, 'c) t \rightarrow  
  int \rightarrow  
  int \rightarrow  
  'a

  Like Bigarray.Array2.get\textsuperscript{[29.1]}, but bounds checking is not always performed.

val unsafe_set : ('a, 'b, 'c) t \rightarrow  
  int \rightarrow  
  int \rightarrow  
  'a \rightarrow  
  unit

  Like Bigarray.Array2.set\textsuperscript{[29.1]}, but bounds checking is not always performed.

end

Two-dimensional arrays. The Array2 structure provides operations similar to those of
Bigarray.Genarray\textsuperscript{[29.1]}, but specialized to the case of two-dimensional arrays.

Three-dimensional arrays

module Array3 :
  sig

  type ('a, 'b, 'c) t

  The type of three-dimensional big arrays whose elements have Caml type \texttt{'a},
  representation kind \texttt{'b}, and memory layout \texttt{'c}.

  val create :  
    ('a, 'b) Bigarray.kind \rightarrow  
    'c Bigarray.layout \rightarrow  
    int \rightarrow  
    int \rightarrow  
    int \rightarrow  
    ('a, 'b, 'c) t

    Array3.create kind layout dim1 dim2 dim3 returns a new bigarray of three
dimension, whose size is \texttt{dim1} in the first dimension, \texttt{dim2} in the second dimension, and
\texttt{dim3} in the third. \texttt{kind} and \texttt{layout} determine the array element kind and the array
layout as described for Bigarray.Genarray.create\textsuperscript{[29.1]}.

  val dim1 : ('a, 'b, 'c) t \rightarrow  
    int

    Return the first dimension of the given three-dimensional big array.
val dim2 : ('a, 'b, 'c) t -> int
    Return the second dimension of the given three-dimensional big array.

val dim3 : ('a, 'b, 'c) t -> int
    Return the third dimension of the given three-dimensional big array.

val kind : ('a, 'b, 'c) t -> ('a, 'b) Bigarray.kind
    Return the kind of the given big array.

val layout : ('a, 'b, 'c) t -> 'c Bigarray.layout
    Return the layout of the given big array.

val get : ('a, 'b, 'c) t -> int -> int -> int -> 'a
    Array3.get a x y z, also written a.\{x,y,z\}, returns the element of a at coordinates (x, y, z). x, y and z must be within the bounds of a, as described for Bigarray.Genarray.get\[29.1\]; otherwise, Invalid_argument is raised.

val set : ('a, 'b, 'c) t -> int -> int -> int -> 'a -> unit
    Array3.set a x y v, or alternatively a.\{x,y,z\} <- v, stores the value v at coordinates (x, y, z) in a. x, y and z must be within the bounds of a, as described for Bigarray.Genarray.set\[29.1\]; otherwise, Invalid_argument is raised.

val sub_left : ('a, 'b, Bigarray.c_layout) t ->
    int -> int -> ('a, 'b, Bigarray.c_layout) t
    Extract a three-dimensional sub-array of the given three-dimensional big array by restricting the first dimension. See Bigarray.Genarray.sub_left\[29.1\] for more details. Array3.sub_left applies only to arrays with C layout.

val sub_right : ('a, 'b, Bigarray.fortran_layout) t ->
    int -> int -> ('a, 'b, Bigarray.fortran_layout) t
    Extract a three-dimensional sub-array of the given three-dimensional big array by restricting the second dimension. See Bigarray.Genarray.sub_right\[29.1\] for more details. Array3.sub_right applies only to arrays with Fortran layout.

val slice_left_1 : ('a, 'b, Bigarray.c_layout) t ->
    int -> int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array1.t
    Extract a one-dimensional slice of the given three-dimensional big array by fixing the first two coordinates. The integer parameters are the coordinates of the slice to extract. See Bigarray.Genarray.slice_left\[29.1\] for more details. Array3.slice_left_1 applies only to arrays with C layout.
Chapter 29. The bigarray library

val slice_right_1 : ('a, 'b, Bigarray.fortran_layout) t ->
                  int -> int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array1.t

Extract a one-dimensional slice of the given three-dimensional big array by fixing the last two coordinates. The integer parameters are the coordinates of the slice to extract. See Bigarray.Genarray.slice_right[29.1] for more details. Array3.slice_right_1 applies only to arrays with Fortran layout.

val slice_left_2 : ('a, 'b, Bigarray.c_layout) t ->
                  int -> ('a, 'b, Bigarray.c_layout) Bigarray.Array2.t

Extract a two-dimensional slice of the given three-dimensional big array by fixing the first coordinate. The integer parameter is the first coordinate of the slice to extract. See Bigarray.Genarray.slice_left[29.1] for more details. Array3.slice_left_2 applies only to arrays with C layout.

val slice_right_2 : ('a, 'b, Bigarray.fortran_layout) t ->
                  int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array2.t

Extract a two-dimensional slice of the given three-dimensional big array by fixing the last coordinate. The integer parameter is the coordinate of the slice to extract. See Bigarray.Genarray.slice_right[29.1] for more details. Array3.slice_right_2 applies only to arrays with Fortran layout.

val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit

Copy the first big array to the second big array. See Bigarray.Genarray.blit[29.1] for more details.

val fill : ('a, 'b, 'c) t -> 'a -> unit

Fill the given big array with the given value. See Bigarray.Genarray.fill[29.1] for more details.

val of_array :
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout -> 'a array array array -> ('a, 'b, 'c) t

Build a three-dimensional big array initialized from the given array of arrays of arrays.

val map_file :
  Unix.file_descr ->
  ?pos:int64 ->
  ('a, 'b) Bigarray.kind ->
  'c Bigarray.layout ->
  bool -> int -> int -> int -> ('a, 'b, 'c) t
Memory mapping of a file as a three-dimensional big array. See Bigarray.Genarray.map_file for more details.

val unsafe_get : ('a, 'b, 'c) t -> int -> int -> int -> 'a

Like Bigarray.Array3.get, but bounds checking is not always performed.

val unsafe_set : ('a, 'b, 'c) t -> int -> int -> int -> 'a -> unit

Like Bigarray.Array3.set, but bounds checking is not always performed.

end

Three-dimensional arrays. The Array3 structure provides operations similar to those of Bigarray.Genarray, but specialized to the case of three-dimensional arrays.

Coercions between generic big arrays and fixed-dimension big arrays

val genarray_of_array1 : ('a, 'b, 'c) Array1.t -> ('a, 'b, 'c) Genarray.t

Return the generic big array corresponding to the given one-dimensional big array.

val genarray_of_array2 : ('a, 'b, 'c) Array2.t -> ('a, 'b, 'c) Genarray.t

Return the generic big array corresponding to the given two-dimensional big array.

val genarray_of_array3 : ('a, 'b, 'c) Array3.t -> ('a, 'b, 'c) Genarray.t

Return the generic big array corresponding to the given three-dimensional big array.

val array1_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array1.t

Return the one-dimensional big array corresponding to the given generic big array. Raise Invalid_argument if the generic big array does not have exactly one dimension.

val array2_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array2.t

Return the two-dimensional big array corresponding to the given generic big array. Raise Invalid_argument if the generic big array does not have exactly two dimensions.

val array3_of_genarray : ('a, 'b, 'c) Genarray.t -> ('a, 'b, 'c) Array3.t

Return the three-dimensional big array corresponding to the given generic big array. Raise Invalid_argument if the generic big array does not have exactly three dimensions.
Chapter 29. The bigarray library

Re-shaping big arrays

val reshape :
  ('a, 'b, 'c) Genarray.t ->
  int array -> ('a, 'b, 'c) Genarray.t

reshape b [|d1;...;dN|] converts the big array b to a N-dimensional array of dimensions 
d1...dN. The returned array and the original array b share their data and have the same 
layout. For instance, assuming that b is a one-dimensional array of dimension 12, reshape 
b [|3;4|] returns a two-dimensional array b' of dimensions 3 and 4. If b has C layout, the 
element (x,y) of b' corresponds to the element x * 3 + y of b. If b has Fortran layout, 
the element (x,y) of b' corresponds to the element x + (y - 1) * 4 of b. The returned 
big array must have exactly the same number of elements as the original big array b. That 
is, the product of the dimensions of b must be equal to i1 * ... * iN. Otherwise, 
Invalid_argument is raised.

val reshape_1 : ('a, 'b, 'c) Genarray.t -> int -> ('a, 'b, 'c) Array1.t

Specialized version of Bigarray.reshape[29.1] for reshaping to one-dimensional arrays.

val reshape_2 :
  ('a, 'b, 'c) Genarray.t ->
  int -> int -> ('a, 'b, 'c) Array2.t

Specialized version of Bigarray.reshape[29.1] for reshaping to two-dimensional arrays.

val reshape_3 :
  ('a, 'b, 'c) Genarray.t ->
  int -> int -> int -> ('a, 'b, 'c) Array3.t

Specialized version of Bigarray.reshape[29.1] for reshaping to three-dimensional arrays.

29.2 Big arrays in the Caml-C interface

C stub code that interface C or Fortran code with Caml code, as described in chapter 18, can 
exploit big arrays as follows.

29.2.1 Include file

The include file <caml/bigarray.h> must be included in the C stub file. It declares the functions, 
constants and macros discussed below.

29.2.2 Accessing a Caml bigarray from C or Fortran

If v is a Caml value representing a big array, the expression Data_bigarray_val(v) returns a 
pointer to the data part of the array. This pointer is of type void * and can be cast to the 
appropriate C type for the array (e.g. double [], char [] [10], etc).

Various characteristics of the Caml big array can be consulted from C as follows:
<table>
<thead>
<tr>
<th>C expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Bigarray_val(v)-&gt;num_dims</td>
<td>number of dimensions</td>
</tr>
<tr>
<td>Bigarray_val(v)-&gt;dim[i]</td>
<td>i-th dimension</td>
</tr>
<tr>
<td>Bigarray_val(v)-&gt;flags &amp; BIGARRAY_KIND_MASK</td>
<td>kind of array elements</td>
</tr>
</tbody>
</table>

The kind of array elements is one of the following constants:

<table>
<thead>
<tr>
<th>Constant</th>
<th>Element kind</th>
</tr>
</thead>
<tbody>
<tr>
<td>BIGARRAY_FLOAT32</td>
<td>32-bit single-precision floats</td>
</tr>
<tr>
<td>BIGARRAY_FLOAT64</td>
<td>64-bit double-precision floats</td>
</tr>
<tr>
<td>BIGARRAY_SINT8</td>
<td>8-bit signed integers</td>
</tr>
<tr>
<td>BIGARRAY_UINT8</td>
<td>8-bit unsigned integers</td>
</tr>
<tr>
<td>BIGARRAY_SINT16</td>
<td>16-bit signed integers</td>
</tr>
<tr>
<td>BIGARRAY_UINT16</td>
<td>16-bit unsigned integers</td>
</tr>
<tr>
<td>BIGARRAY_INT32</td>
<td>32-bit signed integers</td>
</tr>
<tr>
<td>BIGARRAY_INT64</td>
<td>64-bit signed integers</td>
</tr>
<tr>
<td>BIGARRAY_CAML_INT</td>
<td>31- or 63-bit signed integers</td>
</tr>
<tr>
<td>BIGARRAY_NATIVE_INT</td>
<td>32- or 64-bit (platform-native) integers</td>
</tr>
</tbody>
</table>

The following example shows the passing of a two-dimensional big array to a C function and a Fortran function.

```c
extern void my_c_function(double * data, int dimx, int dimy);
extern void my_fortran_function_(double * data, int * dimx, int * dimy);

value caml_stub(value bigarray)
{
    int dimx = Bigarray_val(bigarray)->dim[0];
    int dimy = Bigarray_val(bigarray)->dim[1];
    /* C passes scalar parameters by value */
    my_c_function(Data_bigarray_val(bigarray), dimx, dimy);
    /* Fortran passes all parameters by reference */
    my_fortran_function_(Data_bigarray_val(bigarray), &dimx, &dimy);
    return Val_unit;
}
```

### 29.2.3 Wrapping a C or Fortran array as a Caml big array

A pointer `p` to an already-allocated C or Fortran array can be wrapped and returned to Caml as a big array using the `alloc_bigarray` or `alloc_bigarray_dims` functions.

- **alloc_bigarray**

  ```c
  alloc_bigarray(kind | layout, numdims, p, dims)
  ```

  Return a Caml big array wrapping the data pointed to by `p`. `kind` is the kind of array elements (one of the `BIGARRAY_` kind constants above). `layout` is `BIGARRAY_C_LAYOUT` for an array with C layout and `BIGARRAY_FORTRAN_LAYOUT` for an array with Fortran layout. `numdims` is the number of dimensions in the array. `dims` is an array of `numdims` long integers, giving the sizes of the array in each dimension.
• \texttt{alloc\_bigarray\_dims(kind | layout, numdims, p, (long) dim_1, (long) dim_2, ..., (long) \texttt{dim_{numdims}})}

Same as \texttt{alloc\_bigarray}, but the sizes of the array in each dimension are listed as extra arguments in the function call, rather than being passed as an array.

The following example illustrates how statically-allocated C and Fortran arrays can be made available to Caml.

```c
extern long my_c_array[100][200];
extern float my_fortran_array_[300][400];

value caml_get_c_array(value unit)
{
    long dims[2];
    dims[0] = 100; dims[1] = 200;
    return alloc_bigarray(BIGARRAY_NATIVE_INT | BIGARRAY_C_LAYOUT,
                           2, my_c_array, dims);
}

cvalue caml_get_fortran_array(value unit)
{
    return alloc_bigarray_dims(BIGARRAY_FLOAT32 | BIGARRAY_FORTRAN_LAYOUT,
                                2, my_fortran_array_, 300L, 400L);
}
```
Part V

Appendix
Index to the library

( * ), 288
( */ ), 438
( ** ), 290
( **/ ), 438
( *. ), 290
( & ), 288
( & & ), 287
( @ ), 295
( ! ), 302
( != ), 287
( : = ), 302
( = ), 286
( == ), 287
( /= ), 439
( - ), 293
( ^ ), 303
( >= ), 286
( >= ), 286
( >= /= ), 439
( >/ ), 439
( < ), 286
( <= ), 286
( <= /= ), 439
( <= >/ ), 439
( < / ), 439
( -, 288
( /= ), 438
( . ), 290
( !! ), 288
( /= ), 438
( < > ), 286
( <+ > ), 439
( < > / ), 439
( < / ), 439
( - ), 288
( /= ), 438
( . ), 290
( ! ), 288
( // ), 438
( /. ), 286
( - - ), 288
( - + ), 288
( . + ), 290
( asr ), 289
( land ), 289
( lor ), 289
( lsl ), 289
( lsr ), 289
( lxor ), 289
( mod ), 289
( or ), 288
abs, 289 342 345 361
abs_big_int, 441
abs_float, 292
abs_num, 438
accept, 425 460
access, 414
access_permission, 413
acos, 291
adapt_filename, 476
add, 316 338 340 342 344 355 361 369 382 399 472
add_available_units, 476
add_big_int, 441
add_buffer, 314
add_channel, 314
add_char, 314
add_int_big_int, 441
add_interfaces, 476
add_num, 438
add_string, 314
add_substitute, 314
add_substring, 314
addr_info, 430
alarm, 337 420
align, 309
allocated_bytes, 336
allow_only, 476
allow_unsafe_modules, 476
always, 438
anchor, 481
and_big_int, 444
anon_fun, 308
append, 311 351 386
appname_get, 481
appname_set, 481
clear_graph, 463
clear_nonblock, 414
clear_parser, 364
close, 409, 472
close_box, 321
close_graph, 462
close_in, 301, 374
close_in_noerr, 301
close_out, 299
close_out_noerr, 299
close_process, 417
close_process_full, 417
close_process_in, 417
close_process_out, 417
close_tag, 325
close_tbox, 324
closedir, 415
closeTk, 480
code, 315
color, 463
combine, 354, 387
command, 393
compact, 336
color
compare, 287, 315, 344, 347, 355, 356, 363, 381, 382, 388, 393
compare_big_int, 442
compare_num, 440
Complex, 316
color
complex32, 489
color
complex32_elt, 488
color
complex64, 489
color
complex64_elt, 489
color
concat, 311, 318, 351, 386, 388, 391
Condition, 456
color
conj, 316
connect, 423, 460
contains, 388, 392
contains_from, 388, 392
contents, 313
color
control, 335
count, 390, 399

create, 310, 313, 338, 340, 369, 385, 386, 388, 390, 397, 398, 454, 456, 457, 491, 495, 497, 499
create_alarm, 337
create_image, 467
create_matrix, 310, 386
create_process, 416
create_process_env, 416
current, 309
create_process

current_dir_name, 318
current_point, 464
current_x, 464
current_y, 464
curveto, 464
cyan, 463
data, 398
data_size, 360
Dbm, 471
Dbm_error, 472
decr, 302
decr_num, 349
default_available_units, 476
delay, 454
delete_alarm, 337
descr_of_in_channel, 409
descr_of_out_channel, 409
destroy, 480
diff, 382
Digest, 317
dim, 495
dim1, 497, 499
dim2, 497, 500
dim3, 500
dims, 492
dir_handle, 415
dir_sep, 318
dirname, 319
display_mode, 469
div, 316, 342, 345, 361
div_big_int, 442
div_num, 338
Division_by_zero, 285
doc, 308
INDEX TO THE LIBRARY

float64_elt, 488
floor, 292
floor_num, 439
flow_action, 434
flush, 297
flush_all, 298
flush_input, 350
flush_queue, 434
flush_str_formatter, 328
fold, 339, 340, 356, 370, 383, 399
fold_left, 312, 331, 386, 387
fold_left2, 352, 387
fold_right, 312, 351, 386, 387
fold_right2, 352, 387
for_all, 352, 356, 383, 387
for_all2, 352, 387
force, 137, 347
force_newline, 322
foreground, 463
fork, 407
Format, 320
format, 302
format_from_string, 381
format_of_string, 302
format4, 284, 302
formatter, 327
formatter_of_buffer, 328
formatter_of_out_channel, 327
formatter_tag_functions, 327
fortran_layout, 490, 491
fpclass, 293
fprintf, 330, 366
frexp, 292
from, 389
from_channel, 349, 359, 374
from_file, 374
from_file_bin, 374
from_function, 349, 374
from_string, 349, 359, 374
fscanf, 380
fst, 295
fstat, 411, 413
ftruncate, 110, 112
full_init, 370
full_major, 335
full_split, 451
Gc, 332
gcd_big_int, 442
ge_big_int, 442
ge_num, 440
Genarray, 491
genarray_of_array1, 502
genarray_of_array2, 502
genarray_of_array3, 502
Genlex, 337
get, 310, 335, 383, 388, 390, 397, 492, 495, 497, 500
get_all_formatter_output_functions, 326
get_approx_printing, 445
get_backtrace, 365
get_copy, 397
get_ellipsis_text, 324
get_error_when_null_denominator, 445
get_floating_precision, 445
get_formatter_output_functions, 326
get_formatter_tag_functions, 327
get_image, 467
get_margin, 322
get_mark_tags, 325
get_max_boxes, 323
get_max_indent, 323
get_normalize_ratio, 445
generate_ratio_when_printing, 445
get_print_tags, 325
get_state, 372
getaddrinfo, 430
getaddrinfo_option, 430
getcwd, 394, 415
getegid, 422
getenv, 393, 406
geteuid, 422
getgid, 422
getgrgid, 423
getgrnam, 423
getgroups, 422
gethostbyaddr, 429
gethostbyname, 429
INDEX TO THE LIBRARY

int64_elt, 489
int64_of_big_int, 444
int8_signed, 489
int8_signed_elt, 489
int8_unsigned, 489
int8_unsigned_elt, 489
integer_num, 439
inter, 382
interactive, 394
interval_timer, 421
interval_timer_status, 421
inv, 316
invalid_arg, 286
Invalid_argument, 285
is_directory, 393
is_empty, 355, 369, 382, 385
is_implicit, 318
is_int_big_int, 443
is_integer_num, 439
is_native, 475
is_relative, 318
isatty, 411
iter, 311, 339, 340, 351, 356, 370, 382, 385
iter2, 352, 387
iteri, 312, 386
join, 454
junk, 390

kbprintf, 368
key, 308, 340, 355
key_pressed, 468
kprintf, 332, 368
kill, 418, 454
kind, 489, 492, 495, 497, 500
kprintf, 332, 368
kscanf, 380
ksprintf, 332, 368

LargeFile, 301, 412
last_chars, 451
layout, 491, 492, 495, 497, 500
Lazy, 347
Lazy (module), 137
lazy_from_fun, 348

lazy_from_val, 348
lazy_is_val, 348
lazy_t, 284
ldexp, 292
le_big_int, 442
le_num, 440
length, 309, 313, 339, 340, 350, 369, 385, 386
lexbuf, 349
lexeme, 349
lexeme_char, 350
lexeme_end, 350
lexeme_end_p, 350
lexeme_start, 350
lexeme_start_p, 350
Lexing, 348
linfo, 464
link, 413
linking_error, 477
List, 350, 386
listen, 425
link, 289
loadfile, 475
loadfile_private, 475
localtime, 420
lock, 456
lock_command, 418
lockf, 418
log, 290, 317
log10, 290
log1p, 291
logand, 342, 345, 361
lognot, 343, 345, 362
logor, 342, 345, 362
logxor, 343, 345, 362
lower_window, 482
lowercase, 315, 388, 392
lseek, 410, 412
lstat, 411, 413
lt_big_int, 442
lt_num, 439
magenta, 363
mainLoop, 480
<table>
<thead>
<tr>
<th>Term</th>
<th>Page(s)</th>
</tr>
</thead>
<tbody>
<tr>
<td>num_of_nat</td>
<td>440</td>
</tr>
<tr>
<td>num_of_ratio</td>
<td>440</td>
</tr>
<tr>
<td>num_of_string</td>
<td>440</td>
</tr>
<tr>
<td>ocaml_version</td>
<td>396</td>
</tr>
<tr>
<td>of_array</td>
<td>496</td>
</tr>
<tr>
<td>of_channel</td>
<td>389</td>
</tr>
<tr>
<td>of_float</td>
<td>343</td>
</tr>
<tr>
<td>of_int</td>
<td>343</td>
</tr>
<tr>
<td>of_int32</td>
<td>346</td>
</tr>
<tr>
<td>of_list</td>
<td>311</td>
</tr>
<tr>
<td>of_nativeint</td>
<td>346</td>
</tr>
<tr>
<td>of_string</td>
<td>343</td>
</tr>
<tr>
<td>one</td>
<td>316</td>
</tr>
<tr>
<td>Oo</td>
<td>363</td>
</tr>
<tr>
<td>open_box</td>
<td>321</td>
</tr>
<tr>
<td>open_connection</td>
<td>428</td>
</tr>
<tr>
<td>open_flag</td>
<td>297</td>
</tr>
<tr>
<td>open_graph</td>
<td>462</td>
</tr>
<tr>
<td>open_hbbox</td>
<td>323</td>
</tr>
<tr>
<td>open_hbox</td>
<td>323</td>
</tr>
<tr>
<td>open_in</td>
<td>299</td>
</tr>
<tr>
<td>open_in_bin</td>
<td>299</td>
</tr>
<tr>
<td>open_in_gen</td>
<td>299</td>
</tr>
<tr>
<td>open_out</td>
<td>297</td>
</tr>
<tr>
<td>open_out_bin</td>
<td>297</td>
</tr>
<tr>
<td>open_out_gen</td>
<td>297</td>
</tr>
<tr>
<td>open_process</td>
<td>416</td>
</tr>
<tr>
<td>open_process_full</td>
<td>416</td>
</tr>
<tr>
<td>open_process_in</td>
<td>416</td>
</tr>
<tr>
<td>open_process_out</td>
<td>416</td>
</tr>
<tr>
<td>open_tag</td>
<td>325</td>
</tr>
<tr>
<td>open_tbbox</td>
<td>323</td>
</tr>
<tr>
<td>open_temp_file</td>
<td>319</td>
</tr>
<tr>
<td>open_vbbox</td>
<td>323</td>
</tr>
<tr>
<td>opendir</td>
<td>415</td>
</tr>
<tr>
<td>openfile</td>
<td>409</td>
</tr>
<tr>
<td>openTk</td>
<td>480</td>
</tr>
<tr>
<td>option</td>
<td>284</td>
</tr>
<tr>
<td>or_big_int</td>
<td>441</td>
</tr>
<tr>
<td>OrderedType</td>
<td>355</td>
</tr>
<tr>
<td>os_type</td>
<td>394</td>
</tr>
<tr>
<td>out_channel</td>
<td>295</td>
</tr>
<tr>
<td>out_channel_length</td>
<td>298</td>
</tr>
<tr>
<td>out_channel_of_descr</td>
<td>409</td>
</tr>
<tr>
<td>Out_of_memory</td>
<td>285</td>
</tr>
<tr>
<td>output</td>
<td>298</td>
</tr>
<tr>
<td>output_binary_int</td>
<td>298</td>
</tr>
<tr>
<td>output_buffer</td>
<td>314</td>
</tr>
<tr>
<td>output_byte</td>
<td>298</td>
</tr>
<tr>
<td>output_char</td>
<td>298</td>
</tr>
<tr>
<td>output_string</td>
<td>298</td>
</tr>
<tr>
<td>output_value</td>
<td>298</td>
</tr>
<tr>
<td>over_max_boxes</td>
<td>323</td>
</tr>
<tr>
<td>pack</td>
<td>481</td>
</tr>
<tr>
<td>parent_dir_name</td>
<td>318</td>
</tr>
<tr>
<td>parse</td>
<td>308</td>
</tr>
<tr>
<td>parse_argv</td>
<td>309</td>
</tr>
<tr>
<td>Parse_error</td>
<td>364</td>
</tr>
<tr>
<td>Parsing</td>
<td>363</td>
</tr>
<tr>
<td>partition</td>
<td>353</td>
</tr>
<tr>
<td>passwd_entry</td>
<td>422</td>
</tr>
<tr>
<td>pause</td>
<td>419</td>
</tr>
<tr>
<td>peek</td>
<td>369</td>
</tr>
<tr>
<td>Pervasives</td>
<td>286</td>
</tr>
<tr>
<td>pipe</td>
<td>415</td>
</tr>
<tr>
<td>pixels</td>
<td>481</td>
</tr>
<tr>
<td>place</td>
<td>482</td>
</tr>
<tr>
<td>plot</td>
<td>464</td>
</tr>
<tr>
<td>plots</td>
<td>464</td>
</tr>
<tr>
<td>point_color</td>
<td>464</td>
</tr>
<tr>
<td>polar</td>
<td>317</td>
</tr>
<tr>
<td>poll</td>
<td>458</td>
</tr>
<tr>
<td>pop</td>
<td>369</td>
</tr>
<tr>
<td>pos_in</td>
<td>300</td>
</tr>
<tr>
<td>pos_out</td>
<td>298</td>
</tr>
<tr>
<td>position</td>
<td>348</td>
</tr>
<tr>
<td>pow</td>
<td>317</td>
</tr>
<tr>
<td>power_big_int_positive_big_int</td>
<td>442</td>
</tr>
<tr>
<td>power_big_int_positive_int</td>
<td>442</td>
</tr>
<tr>
<td>power_int_positive_big_int</td>
<td>442</td>
</tr>
<tr>
<td>power_int_positive_int</td>
<td>442</td>
</tr>
<tr>
<td>power_num</td>
<td>438</td>
</tr>
<tr>
<td>pp_close_box</td>
<td>328</td>
</tr>
<tr>
<td>pp_close_tag</td>
<td>328</td>
</tr>
<tr>
<td>pp_close_tbbox</td>
<td>329</td>
</tr>
<tr>
<td>pp_force_newline</td>
<td>329</td>
</tr>
</tbody>
</table>
INDEX TO THE LIBRARY

519

ratio_of_num, 440
rcontains_from, 388, 392
read, 409, 459
read_float, 296
read_int, 296
read_key, 468
read_line, 296
readdir, 394, 415
readlink, 417
really_input, 300
receive, 458
record_backtrace, 365
recv, 426, 460
recvfrom, 426, 460
red, 463
ref, 302
regexp, 447
regexp_case_fold, 448
regexp_string, 448
regexp_string_case_fold, 448
register, 315
register_exception, 315
register_printer, 365
rem, 342, 349, 361
remember_mode, 369
remove, 339, 340, 350, 356, 382, 393, 399, 472
remove_assoc, 353, 387
remove_assq, 354, 387
rename, 393, 413
replace, 339, 340, 472
replace_first, 450
replace_matched, 450
reset, 314
reshape, 503
reshape_1, 503
reshape_2, 503
reshape_3, 503
resize_window, 462
rev, 351, 386
rev_append, 351, 386
rev_map, 351, 387
rev_map2, 352, 387
rewinddir, 415
rgb, 463
rhs_end, 364
rhs_end_pos, 364
rhs_start, 364
rhs_start_pos, 364
rindex, 388, 392
rindex_from, 388, 392
rlineto, 464
rmdir, 415
rmoveto, 464
round_num, 439
S, 340, 355, 381, 398
Scan_failure, 375
scanbuf, 373
Scang, 372
scanf, 380
scanner, 373
Scanning, 373
search_backward, 449
search_forward, 448
seek_command, 410
seek_in, 300, 301
seek_out, 298, 301
select, 417, 455, 458, 459
self, 454
self_init, 370
send, 426, 458, 460
sendto, 426, 460
service_entry, 429
Set, 381
set, 310, 335, 385, 388, 390, 397, 492, 496, 497, 500
set_all_formatter_output_functions, 326
set_approx_printing, 445
set_binary_mode_in, 301
set_binary_mode_out, 299
set_close_on_exec, 414
set_color, 463
set_ellipsis_text, 324
set_error_when_null_denominator, 445
set_floating_precision, 445
set_font, 465
set_formatter_out_channel, 325
set_formatter_output_functions, 326
set_formatter_tag_functions, 427
set_line_width, 465
set_margin, 322
set_mark_tags, 325
set_max_boxes, 323
set_max_indent, 323
set_nonblock, 414
set_normalize_ratio, 445
set_norm_ratio_when_printing, 445
set_norm_tags, 325
set_signal, 395
set_state, 372
set_tag, 324
set_tags, 325
set_text_size, 466
set_trace, 364
set_window_title, 466
setattr_when, 434
setgid, 422
setgroups, 422
setitimer, 421
setsid, 435
setsockopt, 428
setsockopt_float, 428
setsockopt_int, 428
setsockopt_optint, 428
setuid, 422
shift_left, 343, 345, 362
shift_left_big_int, 444
shift_right, 343, 346, 362
shift_right_big_int, 444
shift_right_logical, 343, 346, 362
shift_right_towards_zero_big_int, 444
shutdown, 425
shutdown_command, 425
shutdown_connection, 428
side, 481
sigabrt, 395
sigalarm, 395
sigchld, 396
sigcont, 396
sigfpe, 395
sighup, 395
sigill, 395
sigint, 395
sigkill, 395
sigmask, 455
sign_big_int, 442
sign_num, 439
signal, 395
signal_behavior, 395
sigpending, 419
sigpipe, 395
sigprocmask, 419
sigprocmask_command, 419
sigprof, 396
sigquit, 396
sigsev, 395
sigstop, 396
sigsuspend, 419
sigterm, 396
sigtstp, 396
sigtim, 396
sigttou, 396
sigusr1, 396
sigusr2, 396
sigtarm, 396
sin, 291
single_write, 409
singleton, 356, 382
sinh, 291
size, 361
size_x, 463
size_y, 463
sleep, 421, 460
slice_left, 493, 498
slice_left_1, 500
slice_left_2, 501
slice_right, 494, 498
slice_right_1, 501
slice_right_2, 501
snd, 295
sockaddr, 424
socket, 424, 460
socket_bool_option, 427
socket_domain, 424
socket_float_option, 427
socket_int_option, 427
socket_optint_option, 427
socket_type, 424
socketpair, 425
INDEX TO THE LIBRARY

Sort, 384
sort, 312, 354, 386, 387
sound, 468
spec, 308
split, 354, 357, 383, 387, 450
split_delim, 451
split_result, 451
sprintf, 331, 368
sqrt, 290, 316
sqrt_big_int, 442
square_big_int, 441
square_num, 438
sscanf, 380
sscanf_format, 380
stable_sort, 312, 354, 386, 387
Stack, 384
Stack_overflow, 285
stat, 333, 335, 411, 413
State, 371
stats, 399, 411, 413
status, 467
std_formatter, 328
stdbuf, 328
stderr, 295, 408
stdib, 375
stdin, 295, 373, 408
StdLabels, 385
stdout, 295, 408
Str, 447
str_format, 328
Stream, 388
String, 387, 390
string, 283, 317
string_after, 451
string_before, 451
string_match, 448
string_of_big_int, 443
string_of_bool, 294
string_of_float, 294
string_of_format, 302
string_of_inet_addr, 423
string_of_int, 294
string_of_num, 440
string_partial_match, 449
sub, 311, 313, 316, 342, 344, 361, 386, 388
391, 496
sub_big_int, 441
sub_left, 493, 498, 500
sub_num, 438
sub_right, 493, 498, 500
subset, 382
substitute_first, 450
substring, 317
succ, 288, 312, 345, 361
succ_big_int, 441
succ_num, 439
symbol_end, 363
symbol_end_pos, 364
symbol_start, 363
symbol_start_pos, 364
symlink, 417
sync, 458
synchronize, 469
Sys, 393
Sys_blocked_io, 285
Sys_error, 285
system, 407, 459
369, 371, 381, 384, 388, 393, 397, 398
445, 456, 457, 471, 491, 495, 497, 499
tag, 324
take, 369
tan, 291
tanh, 291
tcdrain, 434
tcflow, 434
tcflush, 434
tcgetattr, 434
tcgetattr, 434
tcsendbreak, 434
tcsetattr, 434
temp_dir_name, 320
temp_file, 319
terminal_io, 434
text_size, 466
Thread, 454
ThreadUnix, 459
time, 394, 420
timed_read, 459
timed_write, 459
times, 421
Tk, 480
tl, 351
tm, 420
to_buffer, 359
to_channel, 359
to_float, 343 346 362
to_hex, 318
to_int, 343 346 362
to_int32, 346 362
to_list, 311 386
to_nativeint, 346
to_string, 343 347 359 363 365
token, 338
top, 369
total_size, 360
transfer, 370
transp, 466
truncate, 292 410 412
try_lock, 456

umask, 414
uncapitalize, 388 392
Undefined, 347
Undefined_recursive_module, 285
union, 382
unit, 284
unit_big_int, 441
units, 481
Unix, 401
Unix_error, 406
UnixLabels (module), 435
unlink, 413
unlock, 456
unsafe_blit, 388
unsafe_fill, 388
unsafe_get, 386 388 396 499 502
unsafe_set, 386 388 396 499 502
update, 481
uppercase, 315 388 392
usage, 309
usage_msg, 308
usage_string, 309
utimes, 421

wait, 407 457 459
wait_flag, 407
wait_next_event, 468
wait_pid, 455
wait_read, 454
wait_signal, 456
wait_timed_read, 455
wait_timed_write, 455
wait_write, 455
waitpid, 407 459
Weak, 397
white, 463
word_size, 394
wrap, 458
wrap_abort, 458
write, 409 459
xor_big_int, 444
yellow, 463
yield, 455
zero, 316 341 344 360
zero_big_int, 441
## Index of keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>and, see let, type, class</td>
<td>121, 125, 126</td>
</tr>
<tr>
<td>as</td>
<td>100, 101, 103, 104, 121, 123</td>
</tr>
<tr>
<td>assert</td>
<td>137</td>
</tr>
<tr>
<td>begin</td>
<td>106, 108</td>
</tr>
<tr>
<td>class</td>
<td>125, 126, 128, 130, 131</td>
</tr>
<tr>
<td>constraint</td>
<td>118, 121, 125</td>
</tr>
<tr>
<td>do, see while, for</td>
<td>end, 106, 108, 120, 121, 126, 127, 130, 131</td>
</tr>
<tr>
<td>done, see while, for</td>
<td>exception, 119, 126, 128, 130, 131</td>
</tr>
<tr>
<td>downto, see for</td>
<td>external, 126, 127, 130, 131</td>
</tr>
<tr>
<td>else, see if</td>
<td>false, 97</td>
</tr>
<tr>
<td>end</td>
<td>106, 112</td>
</tr>
<tr>
<td>for</td>
<td>106, 107, 109, 121, 130</td>
</tr>
<tr>
<td>fun</td>
<td>106, 107, 109</td>
</tr>
<tr>
<td>function</td>
<td>120, 129, 130, 133</td>
</tr>
<tr>
<td>functor</td>
<td>117, 126, 132, 143, 148</td>
</tr>
<tr>
<td>if</td>
<td>106, 107, 111</td>
</tr>
<tr>
<td>in, see let</td>
<td>121</td>
</tr>
<tr>
<td>include</td>
<td>126, 129, 130, 132, 148</td>
</tr>
<tr>
<td>inherit</td>
<td>120, 121, 123</td>
</tr>
<tr>
<td>initializer</td>
<td>124, 125</td>
</tr>
<tr>
<td>lazy</td>
<td>137</td>
</tr>
<tr>
<td>let</td>
<td>106, 107, 110, 121, 130, 131, 138, 142</td>
</tr>
<tr>
<td>match</td>
<td>106, 107, 111</td>
</tr>
<tr>
<td>method</td>
<td>120, 121, 124</td>
</tr>
<tr>
<td>module</td>
<td>126, 128, 130, 132, 138, 144, 148</td>
</tr>
<tr>
<td>mutable</td>
<td>118, 120, 121, 123, 124</td>
</tr>
<tr>
<td>new</td>
<td>106, 115</td>
</tr>
<tr>
<td>object</td>
<td>106, 115, 120, 121</td>
</tr>
<tr>
<td>of, see type, exception</td>
<td>126, 129, 130, 132, 142</td>
</tr>
<tr>
<td>open</td>
<td>106, 107, 111</td>
</tr>
</tbody>
</table>