The OCaml system
release 4.04

Documentation and user’s manual

Xavier Leroy,
Damien Doligez, Alain Frisch, Jacques Garrigue, Didier Rémy and Jérôme Vouillon

November 4, 2016

Copyright © 2016 Institut National de Recherche en Informatique et en Automatique
## Contents

1 **An introduction to OCaml** 11  
   1.1 Basics 13  
   1.2 Data types 14  
   1.3 Functions as values 15  
   1.4 Records and variants 16  
   1.5 Imperative features 18  
   1.6 Exceptions 20  
   1.7 Symbolic processing of expressions 21  
   1.8 Pretty-printing 22  
   1.9 Standalone OCaml programs 23  

2 **The module system** 25  
   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 OCaml** 33  
   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>84</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>104</td>
</tr>
<tr>
<td>6.7</td>
<td>Expressions</td>
<td>108</td>
</tr>
<tr>
<td>6.8</td>
<td>Type and exception definitions</td>
<td>121</td>
</tr>
<tr>
<td>6.9</td>
<td>Classes</td>
<td>123</td>
</tr>
<tr>
<td>6.10</td>
<td>Module types (module specifications)</td>
<td>130</td>
</tr>
<tr>
<td>6.11</td>
<td>Module expressions (module implementations)</td>
<td>134</td>
</tr>
<tr>
<td>6.12</td>
<td>Compilation units</td>
<td>137</td>
</tr>
<tr>
<td>7</td>
<td>Language extensions</td>
<td>139</td>
</tr>
<tr>
<td>7.1</td>
<td>Integer literals for types int32, int64 and nativeint</td>
<td>139</td>
</tr>
<tr>
<td>7.2</td>
<td>Recursive definitions of values</td>
<td>139</td>
</tr>
<tr>
<td>7.3</td>
<td>Lazy patterns</td>
<td>141</td>
</tr>
<tr>
<td>7.4</td>
<td>Recursive modules</td>
<td>141</td>
</tr>
<tr>
<td>7.5</td>
<td>Private types</td>
<td>142</td>
</tr>
<tr>
<td>7.6</td>
<td>Local opens</td>
<td>145</td>
</tr>
<tr>
<td>7.7</td>
<td>Record and object notations</td>
<td>146</td>
</tr>
<tr>
<td>7.8</td>
<td>Explicit polymorphic type annotations</td>
<td>147</td>
</tr>
<tr>
<td>7.9</td>
<td>Locally abstract types</td>
<td>147</td>
</tr>
<tr>
<td>7.10</td>
<td>First-class modules</td>
<td>149</td>
</tr>
<tr>
<td>7.11</td>
<td>Recovering the type of a module</td>
<td>151</td>
</tr>
<tr>
<td>7.12</td>
<td>Substituting inside a signature</td>
<td>151</td>
</tr>
<tr>
<td>7.13</td>
<td>Type-level module aliases</td>
<td>152</td>
</tr>
<tr>
<td>7.14</td>
<td>Explicit overriding in class definitions</td>
<td>154</td>
</tr>
<tr>
<td>7.15</td>
<td>Overriding in open statements</td>
<td>154</td>
</tr>
<tr>
<td>7.16</td>
<td>Generalized algebraic datatypes</td>
<td>155</td>
</tr>
</tbody>
</table>
12 Lexer and parser generators (ocamllex, ocamlyacc) 227
  12.1 Overview of ocamlllex ........................................ 227
  12.2 Syntax of lexer definitions ................................... 228
  12.3 Overview of ocamlyacc ......................................... 233
  12.4 Syntax of grammar definitions ................................ 233
  12.5 Options ......................................................... 236
  12.6 A complete example ............................................ 237
  12.7 Common errors ................................................ 238
  12.8 Module Depend: Module dependencies ........................ 239

13 The browser/editor (ocamlbrowser) 241

14 The documentation generator (ocamldoc) 243
  14.1 Usage .......................................................... 243
  14.2 Syntax of documentation comments ............................ 250
  14.3 Custom generators ............................................. 260
  14.4 Adding command line options .................................. 263

15 The debugger (ocamldebug) 265
  15.1 Compiling for debugging ........................................ 265
  15.2 Invocation ...................................................... 265
  15.3 Commands ....................................................... 266
  15.4 Executing a program ............................................ 267
  15.5 Breakpoints ..................................................... 270
  15.6 The call stack ................................................ 270
  15.7 Examining variable values ..................................... 271
  15.8 Controlling the debugger ....................................... 272
  15.9 Miscellaneous commands ....................................... 275
  15.10 Running the debugger under Emacs ......................... 275

16 Profiling (ocamlprof) 277
  16.1 Compiling for profiling ........................................ 277
  16.2 Profiling an execution ........................................ 278
  16.3 Printing profiling information ............................... 278
  16.4 Time profiling ................................................ 279

17 The ocamlbuild compilation manager 281

18 Interfacing C with OCaml 283
  18.1 Overview and compilation information ....................... 283
  18.2 The value type ................................................ 290
  18.3 Representation of OCaml data types ......................... 291
  18.4 Operations on values .......................................... 294
  18.5 Living in harmony with the garbage collector ............. 297
  18.6 A complete example .......................................... 301
  18.7 Advanced topic: callbacks from C to OCaml ............... 305
18.8 Advanced example with callbacks .................................................. 309
18.9 Advanced topic: custom blocks ..................................................... 311
18.10 Advanced topic: cheaper C call ................................................... 315
18.11 Advanced topic: multithreading .................................................. 317
18.12 Building mixed C/OCaml libraries: ocamlmklib ................................ 319

19 Optimisation with Flambda ............................................................ 323
19.1 Overview ..................................................................................... 323
19.2 Command-line flags ...................................................................... 323
19.3 Inlining ......................................................................................... 326
19.4 Specialisation ................................................................................. 331
19.5 Default settings of parameters ..................................................... 334
19.6 Manual control of inlining and specialisation .................................. 335
19.7 Simplification ................................................................................. 336
19.8 Other code motion transformations .............................................. 337
19.9 Unboxing transformations .............................................................. 338
19.10 Removal of unused code and values ............................................. 342
19.11 Other code transformations .......................................................... 342
19.12 Treatment of effects ..................................................................... 343
19.13 Compilation of statically-allocated modules ................................ 344
19.14 Inhibition of optimisation ............................................................. 344
19.15 Use of unsafe operations ............................................................... 344
19.16 Glossary ....................................................................................... 345
19.17 Module Spacetime: Profiling of a program’s space behaviour over time 346

IV The OCaml library ............................................................................ 349

20 The core library ................................................................................. 351
20.1 Built-in types and predefined exceptions ...................................... 351
20.2 Module Pervasives: The initially opened module. ......................... 354

21 The standard library .......................................................................... 377
21.1 Module Arg: Parsing of command line arguments ....................... 379
21.2 Module Array: Array operations .................................................. 382
21.3 Module Buffer: Extensible buffers ............................................... 386
21.4 Module Bytes: Byte sequence operations ..................................... 389
21.5 Module Callback: Registering OCaml values with the C runtime. 395
21.6 Module Char: Character operations ............................................. 396
21.7 Module Complex: Complex numbers ........................................... 397
21.8 Module Digest: MD5 message digest .......................................... 398
21.9 Module Ephemeron: Ephemerons and weak hash table ............... 400
21.10 Module Filename: Operations on file names ................................ 408
21.11 Module Format: Pretty printing ................................................... 410
21.12 Module Gc: Memory management control and statistics; finalised values 425
21.13 Module Genlex: A generic lexical analyzer .................................. 431
21.14 Module Hashbl: Hash tables and hash functions .......................... 432
21.15 Module Int32: 32-bit integers .............................................. 439
21.16 Module Int64: 64-bit integers .............................................. 442
21.17 Module Lazy: Deferred computations .................................... 445
21.18 Module Lexing: The run-time library for lexers generated by ocamllex .... 447
21.19 Module List: List operations .................................................. 449
21.20 Module Map: Association tables over ordered types .................. 454
21.21 Module Marshal: Marshaling of data structures ....................... 459
21.22 Module MoreLabels: Extra labeled libraries ......................... 461
21.23 Module Nativeint: Processor-native integers ........................... 466
21.24 Module Obj: Operations on objects ........................................ 469
21.25 Module Parsing: The run-time library for parsers generated by ocamlyacc .... 470
21.26 Module Printexc: Facilities for printing exceptions and inspecting current call stack ................................................................. 471
21.27 Module Printf: Formatted output functions ............................... 476
21.28 Module Queue: First-in-first-out queues .................................. 479
21.29 Module Random: Pseudo-random number generators (PRNG) .......... 480
21.30 Module Scanf: Formatted input functions .................................. 483
21.31 Module Set: Sets over ordered types ...................................... 492
21.32 Module Sort: Sorting and merging lists .................................... 496
21.33 Module Spacetime: Profiling of a program’s space behaviour over time .... 497
21.34 Module Stack: Last-in-first-out stacks ..................................... 499
21.35 Module StdLabels: Standard labeled libraries ............................ 500
21.36 Module Stream: Streams and parsers ...................................... 500
21.37 Module String: String operations ......................................... 502
21.38 Module Sys: System interface ............................................... 506
21.39 Module Uchar: Unicode characters ........................................ 513
21.40 Module Weak: Arrays of weak pointers and hash sets of weak pointers .... 514

22 The compiler front-end ................................................................. 519
22.1 Module Ast_mapper: The interface of a -ppx rewriter .................... 519
22.2 Module Asttypes: Auxiliary AST types used by parsetree and typedtree .... 523
22.3 Module Location: Source code locations (ranges of positions), used in parsetree ................................................................. 524
22.4 Module Longident: Long identifiers, used in parsetree .................... 526
22.5 Module Parse: Entry points in the parser .................................... 526
22.6 Module Parsetree: Abstract syntax tree produced by parsing .......... 527
22.7 Module Pprintast ................................................................. 535

23 The unix library: Unix system calls ............................................. 537
23.1 Module Unix: Interface to the Unix system .................................. 537
23.2 Module UnixLabels: labelized version of the interface ................... 572
The num library: arbitrary-precision rational arithmetic

24.1 Module Num: Operation on arbitrary-precision numbers

24.2 Module Big_int: Operations on arbitrary-precision integers

24.3 Module Arith_status: Flags that control rational arithmetic

The str library: regular expressions and string processing

25.1 Module Str: Regular expressions and high-level string processing

The threads library

26.1 Module Thread: Lightweight threads for Posix 1003.1c and Win32

26.2 Module Mutex: Locks for mutual exclusion

26.3 Module Condition: Condition variables to synchronize between threads

26.4 Module Event: First-class synchronous communication

26.5 Module ThreadUnix: Thread-compatible system calls

The graphics library

27.1 Module Graphics: Machine-independent graphics primitives

The dynlink library: dynamic loading and linking of object files

28.1 Module Dynlink: Dynamic loading of object files

The bigarray library

29.1 Module Bigarray: Large, multi-dimensional, numerical arrays

29.2 Big arrays in the OCaml-C interface

Appendix

Index to the library

Index of keywords
### Foreword

This manual documents the release 4.04 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, Seven).

### License

The OCaml system is copyright © 1996–2016 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 © 2016 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 2 deals with the module system, chapter 3 with the object-oriented features, chapter 4 with extensions to the core language (labeled arguments and polymorphic variants), and chapter 5 gives some advanced examples.

1.1 Basics

For this overview of OCaml, 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 OCaml phrases terminated by ;; in response to the # prompt, and the system compiles them on the fly, executes them, and prints the outcome of evaluation. Phrases are either simple expressions, or let definitions of identifiers (either values or functions).

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

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

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

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

The OCaml 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.
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, OCaml offers the usual basic data types: booleans, characters, and immutable 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 OCaml data structures, lists do not need to be explicitly allocated and deallocated from memory: all memory management is entirely automatic in OCaml. Similarly, there is no explicit handling of pointers: the OCaml compiler silently introduces pointers where necessary.

As with most OCaml 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)
```
Chapter 1. The core language

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

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

The type inferred for sort, 'a list -> 'a list, means that sort can actually apply to lists of any type, and returns a list of the same type. The type 'a is a type variable, and stands for any given type. The reason why sort can apply to lists of any type is that the comparisons (=, <=, etc.) are polymorphic in OCaml: 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 OCaml to modify in-place a list once it is built: we say that lists are immutable data structures. Most OCaml 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

OCaml 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 OCaml library provides a `List.map` functional that applies a given function to each element of a list, and returns the list of the results:

```ocaml
# 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.

```ocaml
# 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.

```ocaml
# type ratio = {num: int; denom: int};;
type ratio = { num : int; denom : int; }
# let add_ratio r1 r2 =
#   {num = r1.num * r2.denom + r2.num * r1.denom;
#    denom = r1.denom * r2.denom};;

val add_ratio : ratio -> ratio -> ratio = <fun>
# add_ratio {num=1; denom=3} {num=2; denom=5};;
- : ratio = {num = 11; denom = 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):

```ocaml
# type number = Int of int | Float of float | Error;;
type number = Int of int | Float of float | Error
# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```

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:

```ocaml
# type sign = Positive | Negative;;
type sign = Positive | Negative
# let sign_int n = if n >= 0 then Positive else Negative;;
val sign_int : int -> sign = <fun>
```
To define arithmetic operations for the `number` type, we use pattern-matching on the two numbers involved:

```ocaml
# 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:

```ocaml
# 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):

```ocaml
# 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, OCaml 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.make` function, then filled up later by assignments. For instance, the function below sums two vectors (represented as float arrays) componentwise.

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

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

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

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

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

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

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

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

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

```
# let insertion_sort a =  
#   for i = 1 to Array.length a - 1 do  
#     let val_i = a.(i) in  
#     let j = ref i in  
#     while !j > 0 && val_i < a.(!j - 1) do
```
Chapter 1. The core language

19

#    a.(!j) <- a.(!j - 1);
#    j := !j - 1
# done;
#    a.(!j) <- val_i
# done;;
val insertion_sort : 'a array -> unit = <fun>

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

# let current_rand = ref 0;;
val current_rand : int ref = {contents = 0}
# let random () =
#    current_rand := !current_rand * 25713 + 1345;
#    !current_rand;;
val random : unit -> int = <fun>

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

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

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.

# type idref = { mutable id: 'a. 'a -> 'a };;
type idref = { mutable id : 'a. 'a -> 'a; }
# 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

OCaml 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.

```ocaml
# 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:

```ocaml
# 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:

```ocaml
# 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 OCaml
for symbolic processing: formal manipulations of arithmetic expressions containing variables. The
following variant type describes the expressions we shall manipulate:

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

type expression =
  Const of float
  | Var of string
  | Sum of expression * expression
  | Diff of expression * expression
  | Prod of expression * expression
  | Quot of expression * expression

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

# exception Unbound_variable of string;;
exception Unbound_variable of string

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

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

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

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

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

1.8 Pretty-printing

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

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

# let print_expr exp =
#   (* Local function definitions *)
#   let open_paren prec op_prec =
#     if prec > op_prec then print_string "(" in
#   let close_paren prec op_prec =
#     if prec > op_prec then print_string ")" in
#   let rec print prec exp =
#     (* prec is the current precedence *)
#     match exp with
#       Const c -> print_float c
Chapter 1. The core language

23

# | 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 = ()

1.9 Standalone OCaml programs

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

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

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

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

More complex standalone OCaml programs are typically composed of multiple source files, and can link with precompiled libraries. Chapters 8 and 11 explain how to use the batch compilers ocamlc and ocamlopt. Recompilation of multi-file OCaml projects can be automated using third-party build systems, such as the ocamlbuild compilation manager.
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)
```
else Node(rprio, relt, left, remove_top right)

let extract = function
  | Empty -> raise Queue_is_empty
  | Node(prio, elt, _, _) as queue -> (prio, elt, remove_top queue)
end

module PrioQueue :
  sig
    type priority = int
    type 'a queue = Empty | Node of priority * 'a * 'a queue * 'a queue
    val empty : 'a queue
    val insert : 'a queue -> priority -> 'a -> 'a queue
    exception Queue_is_empty
    val remove_top : 'a queue -> 'a queue
    val extract : 'a queue -> priority * 'a * 'a queue
  end

Outside the structure, its components can be referred to using the “dot notation”, that is, identifiers qualified by a structure name. For instance, PrioQueue.insert is the function insert defined inside the structure PrioQueue and PrioQueue.queue 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 (* still concrete *)
#   type 'a queue (* now abstract *)
#   val empty : 'a queue
#   val insert : 'a queue -> int -> 'a -> 'a queue
#   val extract : 'a queue -> int * 'a * 'a queue
#   exception Queue_is_empty
# end
module type PRIOQUEUE =
  sig
    type priority = int
    type 'a queue
    val empty : 'a queue
Chapter 2. The module system

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

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
end
Chapter 2. The module system

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:

```ocaml
# 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 enriching a signature with extra type equalities:

# module AbstractSet2 =
#   (Set : functor(Elt: ORDERED_TYPE) -> (SET with type element = Elt.t));;
module AbstractSet2 :
  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

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

```ocaml
module AbstractSet2(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.

```ocaml
# module NoCaseString =
#   struct
#     type t = string
#     let compare s1 s2 =
#       OrderedString.compare (String.lowercase_ascii s1) (String.lowercase_ascii 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(OrderedString).set
but an expression was expected of type
  NoCaseStringSet.set = AbstractSet(NoCaseString).set
```

Note 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), they 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-
cal 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.

These two files together define a structure named \( A \) as if the following definition was entered at top-level:

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

The files that define 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 `ocamlc` command. For instance, the following commands compile and link a program composed of two compilation units `Aux` and `Main`:

```bash
$ 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:

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

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

The order in which the `.cmo` files are given to `ocamlc` 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`.

Note that only top-level structures can be mapped to separately-compiled files, but neither functors nor module types. However, all module-class objects can appear as components of a structure, so the solution is to put the functor or module type inside a structure, which can then be mapped to a file.
Chapter 3

Objects in OCaml

(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 mainstream 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.

```ocaml
# 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:

```ocaml
# 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.

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

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

# new point#get_x;;
- : int = 1
# new point#get_x;;
- : int = 2
```

The class `point` can also be abstracted over the initial values of the \( x \) coordinate.

```
# 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:

```
# class point x_init =
# object
#   val mutable x = x_init
#   method get_x = x
#   method move d = x <- x + d
# end;;
class point :
  int ->
  object val mutable x : int method get_x : int method move : int -> unit end
```
An instance of the class `point` is now a function that expects an initial parameter to create a point object:

```ocaml
# new point;;
- : int -> point = <fun>
# let p = new point 7;;
val p : point = <obj>
```

The parameter \(x_{\text{init}}\) 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.

```ocaml
# 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:

```ocaml
# 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 alternate 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 \texttt{3.4}.

\section{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 \texttt{3.3} and \texttt{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 \texttt{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 \texttt{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 through inheritance. 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.

# 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.

# 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:

# class restricted_point' x = (restricted_point x : 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 \( \text{color} \).

```ocaml
# 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 \( \text{color} \). However, the function \( \text{get}_x \) below is a generic function applying method \( \text{get}_x \) 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.

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

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

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

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

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

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

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

A private method that has been hidden in the parent class is no 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 oref x_init =
#    object
```
Chapter 3. Objects in 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 oref :
  '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:

# class oref (x_init:int) =
#  object
#    val mutable x = x_init
#    method get = x
#    method set y = x <- y
#  end;;
class oref :
  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.

# let new_oref x_init =
#  object
#    val mutable x = x_init
#    method get = x
#    method set y = x <- y
#  end;;
val new_oref : '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 listed between [ and ]. The type parameters must also be bound somewhere in the class body by a type constraint.

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

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 move, since no free variables must remain unaccounted for by the class type parameters.

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

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.

```ocaml
# 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.

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: it 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.

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 \text{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.

```ocaml
# 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.

```ocaml
# class type ['a] iterator =
# object method fold : ('b -> 'a -> '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 \((\text{self} : \text{int} \ #\text{iterator})\) idiom, which ensures that this object implements the interface \text{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.

```ocaml
# let sum lst = lst#fold (fun x y -> x+y) 0;;
val sum : < fold : (int -> int -> int) -> int -> .. > -> 'a = <fun>
# sum l;;
Error: This expression has type \text{intlist}
  but an expression was expected of type
  < fold : (int -> int -> int) -> int -> 'a; .. >
Types for method fold are incompatible
```

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

```ocaml
# let sum (lst : _ #iterator) = lst#fold (fun x y -> x+y) 0;;
val sum : \text{int} #iterator -> int = <fun>
```

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

```ocaml
# 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.

```ml
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.

```ml
# 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 : 'b. 'b -> 'b; .. > -> 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>
```

```ocaml
# 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:

\[
\begin{align*}
# \text{fun (x:c0) -> (x : c0) c1);} ;
- : c0 -> c1 = \langle\text{fun}\rangle
\end{align*}
\]

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:

\[
\begin{align*}
# \text{class type } c2 = \text{object ('a) method m : 'a end);} ;
\text{class type } c2 = \text{object ('a) method m : 'a end}
\end{align*}
\]

\[
\begin{align*}
# \text{fun (x:c0) -> (x : c2);} ;
- : c0 -> c2 = \langle\text{fun}\rangle
\end{align*}
\]

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 leaving 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:

\[
\begin{align*}
# \text{let to_c1 x = (x : c1);} ;
\text{val to_c1 : } <m : c1; .. > -> c1 = \langle\text{fun}\rangle
\end{align*}
\]

\[
\begin{align*}
# \text{let to_c2 x = (x : c2);} ;
\text{val to_c2 : } c2 -> c2 = \langle\text{fun}\rangle
\end{align*}
\]

Note the difference between these two coercions: in the case of \(\text{to_c2}\), 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 \#c2 -> c2 while the type of \(\text{to_c1}\) is more general than \#c1 -> c1. 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

\[
\begin{align*}
# \text{function x -> (x : 'a);} ;
- : 'a -> 'a = \langle\text{fun}\rangle
\end{align*}
\]

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 OCaml

# 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 escape its class

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 forces 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 a \#-abbreviation carries 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 `point` without assignments on the instance variables. The override 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 `bad_functional_point`, the method `move` will keep returning an object of the parent class. On the contrary, in a subclass of `functional_point`, 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 a new object that has the same methods and instance variables as its argument. The instance variables are 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 is 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>

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

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

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

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

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

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:

# 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.

# 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 restores : ‘a
    method save : unit
  end
The above definition will only backup one level. The backup facility can be added to any class by using multiple inheritance.

```ocaml
# class [\'a\] backup_ref x = object inherit [\'a\] oref 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]
```

We can define a variant of backup that retains all copies. (We also add a method `clear` to manually erase all copies.)

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

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

3.15 Recursive classes

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

# class window =
# object
#   val mutable top_widget = (None : widget option)
#   method top_widget = top_widget
# end
# and widget (w : window) =
# object
#   val window = w
#   method window = window
# end;
class window :
  object
    val mutable top_widget : widget option
    method top_widget : widget option
  end
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 writing recursive types.

# class virtual comparable =
# object (_ : 'a)
#   method virtual leq : 'a -> bool
# end;;
class virtual comparable : object ('a) method virtual leq : 'a -> bool end
We then define a subclass `money` of `comparable`. The class `money` simply wraps floats as comparable objects. We will extend it below with more operations. We have to use a type constraint on the class parameter `x` because the primitive `<=` is a polymorphic function in OCaml. The `inherit` clause ensures that the type of objects of this class is an instance of `comparable`.

```ocaml
# 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 `money` 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 a call to 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`.

```ocaml
# 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 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`.

```ocaml
# 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 \texttt{money} or \texttt{money2}.

\begin{verbatim}
# (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
\end{verbatim}

More examples of binary methods can be found in sections \texttt{5.2.1} and \texttt{5.2.3}.

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

The class \texttt{money} could naturally carry another binary method. Here is a direct definition:

\begin{verbatim}
# 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 : 'a
    method leq : 'a -> bool
    method plus : 'a -> 'a
    method print : unit
    method times : float -> 'a
    method value : float
  end
\end{verbatim}

3.17 Friends

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

\begin{verbatim}
# class safe_money x =
# object (self : 'a)
#  val repr = x
#  method print = print_float repr
\end{verbatim}
Here, the representation of the object is known only to a particular object. To make it available to other objects of the same class, we are forced to make it available to the whole world. However we can easily restrict the visibility of the representation using the module system.

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

Another example of friend functions may be found in section 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 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 \texttt{Labels} in the standard library, you will see that function
types have annotations you did not have in the functions you defined yourself.

\begin{verbatim}
# ListLabels.map;;
- : f: ('a -> 'b) -> 'a list -> 'b list = <fun>

# StringLabels.sub;;
- : string -> pos:int -> len:int -> string = <fun>
\end{verbatim}

Such annotations of the form \texttt{name:} are called \textit{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{~}.

\begin{verbatim}
# 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
\end{verbatim}

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.

\begin{verbatim}
# let f ~x:x1 ~y:y1 = x1 - y1;;
val f : x:int -> y:int -> int = <fun>

# f ~x:3 ~y:2;;
- : int = 1
\end{verbatim}
Labels obey the same rules as other identifiers in OCaml, 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 several arguments of a function 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 (omitting all optional arguments), labels may be omitted. In practice, many applications are total, so that labels can often be omitted.

```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 a 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.
# 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-optional argument. The criterion for deciding whether an optional argument has been omitted is the non-labeled application of an argument appearing after this optional argument in the function type. Note that if that argument is labeled, you will only be able to eliminate optional arguments through the special case for total applications.

# 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 unlabeled 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 has type int * int * int
This 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
Chapter 4. Labels and variants

The first case is simple: \( g \) is passed \(~y\) and then \(~x\), but \( f \) expects \(~x\) and then \(~y\). This is correctly handled if we know the type of \( g \) to be \( x : \text{int} \to y : \text{int} \to \text{int} \) 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 \( \text{step} : \text{int} \to \text{int} \to \text{int} \), it is inferred as \( \text{step} : \text{int} \to \text{int} \to 'a \). 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 \( \text{bump} \).

\[
\text{let} \ \text{bump}_{\text{it}} \ (\text{bump} : \ ?\text{step} : \text{int} \to \text{int} \to \text{int}) \ \text{x} = \\
\text{bump} \ ?\text{step} : 2 \ \text{x}; \\
\text{val} \ \text{bump}_{\text{it}} : \ (?\text{step} : \text{int} \to \text{int} \to \text{int}) \to \text{int} \to \text{int} = <\text{fun}> \\
\text{let} \ \text{bump}_{\text{it}} \ \text{bump} \ 1;; \\
- : \text{int} = 3
\]

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 \( \text{None} \) for all optional parameters.

\[
\text{let} \ \text{twice} \ f \ (\text{x} : \text{int}) = f(f \ \text{x}); \\
\text{val} \ \text{twice} : \ (\text{int} \to \text{int}) \to \text{int} \to \text{int} = <\text{fun}> \\
\text{let} \ \text{twice} \ \text{bump} \ 2;; \\
- : \text{int} = 8
\]

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

\begin{itemize}
  \item makes programs more readable,
  \item is easy to remember,
\end{itemize}
• 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 from the function itself. The parameters are labeled with names reminding of their nature or their role. The best labels combine nature and role. When this is not possible the role is to be preferred, 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.

BytesLabels.blit :

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

BytesLabels.create : int -> bytes

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

All these are only suggestions, but 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 should 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 that every constructor is assigned to an unique type when defined and used. Even if the same name appears in the definition of multiple types, the constructor itself belongs to only one type. Therefore, one cannot decide that a given constructor belongs to multiple types, 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 `.`.

```ocaml
# ['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 types show that they may still be refined, either by defining more tags or by allowing less. As such, they contain an implicit type variable. Because each of the variant types appears only once in the whole type, their implicit type variables 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 cannot be 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.

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

Here we are seeing two phenomena. First, since this matching is open (the last case catches any tag), we obtain the type (>`A | `B) rather than [<`A | `B] 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 = [Cons of 'a * 'a wlist | Nil | 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] -> [`A | `B | `C] = <fun>
```

You may also selectively coerce values through pattern matching.

```ocaml
# let split_cases = function
  # | Nil | Cons _ as x -> `A x
  # | Snoc _ as x -> `B x
  # ;;
val split_cases : [<Cons of 'a | Nil | Snoc of 'b] ->
  [>`A of [>`Cons of 'a | Nil] | `B of [>`Snoc of 'b] ] = <fun>
```
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>
```

```ocaml
# 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 of int | 'Tag2 of 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.

```ocaml
# let f : abc -> string = function
#  | `As -> "A"
#  | abc -> "other" ;;
Error: This pattern matches values of type [? `As ]
  but a pattern was expected which matches values of type abc
  The second variant type does not allow tag(s) `As
```
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 known 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)

```ocaml
# 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
```

73
We now refine this definition with a method to compute interest.

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

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:

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

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

``` OCaml 
# class safe_account =
#   object
#   inherit account as unsafe
#   method deposit x =
Chapter 5. Advanced examples with classes and modules

```ocaml
# 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 -> unit
    method withdraw : Euro.c -> Euro.c
  end

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

To keep track 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.

```ocaml
# 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 -> unit
    method history : Euro.c operation list
    method private trace : Euro.c operation -> unit
    method withdraw : Euro.c operation -> unit
  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.

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

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.
Chapter 5. Advanced examples with classes and modules

```ocaml
# 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
#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 to provide the client’s view as a functor Client so that client accounts can still be built 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 =
#       object
#         inherit A.bank
#         method mail s = print_string s
#       end
#     class type client_view =
#       object
#         method deposit : m -> unit
```
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 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:

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

However, the method escaped returns an object of the class ostring, and not an object of the current class. Hence, if the class is further extended, the method escaped will only return an object of the parent class.

```ocaml
# class sub_string s =
#   object
```
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.

```plaintext
# class better_string s =
# object
#   val repr = s
#   method get n = String.get repr n
#   method print = print_string repr
#   method escaped = \(<\ repr = String.escaped repr >\)
#   method sub start len = \(<\ repr = String.sub s start len >\)
# end;

class better_string :
  string ->
  object
    val repr : string
    method escaped : 'a
    method get : int -> char
    method print : unit
    method sub : int -> int -> 'a
  end
```

As shown in the inferred type, the methods \( \text{escaped} \) and \( \text{sub} \) now return objects of the same type as the one of the class.

Another difficulty is the implementation of the method \( \text{concat} \). 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 \( \text{repr} \) returning \( s \) must be defined. Here is the correct definition of strings:

```plaintext
# class ostring s =
# object (self : 'mytype)
#   val repr = s
#   method repr = repr
#   method get n = String.get repr n
#   method print = print_string repr
#   method escaped = \(<\ repr = String.escaped repr >\)
#   method sub start len = \(<\ repr = String.sub s start len >\)
#   method concat (t : 'mytype) = \(<\ repr = repr \^ t#repr >\)
# end;
```
Chapter 5. Advanced examples with classes and modules

class ostring :
    string ->
    object ('a)
    val repr : string
    method concat : 'a -> 'a
    method escaped : 'a
    method get : int -> char
    method print : unit
    method repr : string
    method sub : int -> int -> 'a
end

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

# class cstring n = ostring (String.make 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:

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

However, writing a method for iterating over a stack is more problematic. A method fold would have type ('b -> 'a -> 'b) -> 'b -> 'b. Here 'a is the parameter of the stack. The parameter 'b is not related to the class 'a stack but to the argument that will be passed to the method fold. A naive approach is to make 'b an extra parameter of class stack:
# class ['a, 'b] stack2 =
# object
#   inherit ['a] stack
#   method fold f (x : 'b) = List.fold_left f x l
# end;
class ['a, 'b] stack2 :
  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

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

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

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

# s;;
- : (int, int) stack2 = <obj>

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.

# 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.

```ocaml
# 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 hash tables is to use an association list:

```ocaml
# 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 table... whose elements are small hash tables!

```ocaml
# 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 `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 Set in the absence of objects.

In the object-oriented version of sets, we only need to add an additional method `tag` to return the representation of a set. Since sets are parametric in the type of elements, the method `tag` has a parametric type `'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 `tag` of the same type will share the same representation.
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.
Chapter 5. Advanced examples with classes and modules

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

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

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

The difficulty usually lies 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#notifyObservers Move
```
As can be expected, 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 = ()
```

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

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

# 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 attach 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 OCaml is assumed.

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

Notations

The syntax of the language is given in BNF-like notation. Terminal symbols are set in typewriter font (like this). Non-terminal symbols are set in italic font (like that). Square brackets [...] denote optional components. Curly brackets {...} denotes zero, one or several repetitions of the enclosed components. Curly brackets 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, 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

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

\[\text{capitalized-ident} ::= (A \ldots Z) \{\text{letter} \mid 0 \ldots 9 \mid \_ \mid '\}\]

\[\text{lowercase-ident} ::= (a \ldots z \mid \_) \{\text{letter} \mid 0 \ldots 9 \mid \_ \mid '\}\]

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

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 some characters from the ISO 8859-1 set (characters 192–214 and 216–222 as uppercase letters; characters 223–246 and 248–255 as lowercase letters). This feature is deprecated and should be avoided for future compatibility.

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

In many places, OCaml makes a distinction between capitalized identifiers and identifiers that begin with a lowercase letter. The underscore character is considered a lowercase letter for this purpose.

Integer literals

\[\text{integer-literal} ::= [-] (0 \ldots 9) \{0 \ldots 9 \mid \_\} \mid [-] (0x \mid 0X) (0 \ldots 9 \mid A \ldots F \mid a \ldots f) \{0 \ldots 9 \mid A \ldots F \mid a \ldots f \mid \_\} \mid [-] (0o \mid 0O) \{0 \ldots 7 \mid 0 \ldots 1 \mid \_\} \mid [-] (0b \mid 0B) \{0 \ldots 1 \mid \_\}\]

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

\[\text{float-literal} ::= [-] (0 \ldots 9) \{0 \ldots 9 \mid \_\} \{0 \ldots 9 \mid \_\} \{[e \mid E] \{+ \mid -\} (0 \ldots 9) \{0 \ldots 9 \mid \_\}\] \mid [-] (0x \mid 0X) (0 \ldots 9 \mid A \ldots F \mid a \ldots f) \{0 \ldots 9 \mid A \ldots F \mid a \ldots f \mid \_\} \mid [. \{0 \ldots 9 \mid A \ldots F \mid a \ldots f \mid \_\} \{[p \mid P] \{+ \mid -\} (0 \ldots 9) \{0 \ldots 9 \mid \_\}\]

Floating-point decimal literals consist in an integer part, a fractional part and an exponent part. The integer part is a sequence of one or more digits, optionally preceded by a minus sign.
The fractional 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. It is interpreted as a power of 10. The fractional 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.

Floating-point hexadecimal literals are denoted with the 0x or 0X prefix. The syntax is similar to that of floating-point decimal literals, with the following differences. The integer part and the fractional part use hexadecimal digits. The exponent part starts with the character p or P. It is written in decimal and interpreted as a power of 2.

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

Character literals

```plaintext
char-literal ::= ' regular-char '
             | ' escape-sequence '
escape-sequence ::= \ ( " | ' | n | t | b | r | space )
                   | \ ( 0...9 ) ( 0...9 ) ( 0...9 )
                   | \x ( 0...9 | A...F | a...f ) ( 0...9 | A...F | a...f )
                   | \o ( 0...3 ) ( 0...7 ) ( 0...7 )
```

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>
<tr>
<td>oooo</td>
<td>the character with ASCII code ooo in octal</td>
</tr>
</tbody>
</table>

String literals

```plaintext
string-literal ::= " {string-character} "
string-character ::= regular-string-char
                   | escape-sequence
                   | \ newline {space | tab}
```
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 spaces-or-tabs (a backslash at the end of a line followed by any number of spaces and horizontal tabulations 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.

\[
\begin{align*}
\text{label-name} & \ ::= \ \text{lowercase-ident} \\
\text{label} & \ ::= \ ~ \text{label-name} : \\
\text{optlabel} & \ ::= \ ? \text{label-name} : \\
\end{align*}
\]

Naming labels come in two flavours: label for normal arguments and optlabel for optional ones. They are simply distinguished by their first character, either ~ or ?.

Despite label and optlabel being lexical entities in expressions, their expansions ~ label-name : and ? label-name : will be used in grammars, for the sake of readability. Note also that inside type expressions, this expansion can be taken literally, i.e. there are really 3 tokens, with optional blanks between them.

**Prefix and infix symbols**

\[
\begin{align*}
\text{infix-symbol} & \ ::= \ (=|<|>|@|\^|\_|&|+|-|\*|/|\$|\%)\{\text{operator-char}\} \\
& \quad | \ #\{\text{operator-char}\}^+ \\
\text{prefix-symbol} & \ ::= \ !\{\text{operator-char}\} \\
& \quad | \ (?|\~)\{\text{operator-char}\}^+ \\
\text{operator-char} & \ ::= \ !|\$|\%|\&|\*|+|-|\_|/|\:|\<|\>|\?|\@|\~|\_|\_\_ \\
\end{align*}
\]

Sequences of “operator characters”, such as <= or !!, are read as a single token from the infix-symbol or prefix-symbol class. These symbols are parsed as prefix and infix operators inside expressions, but otherwise behave like normal identifiers.

**Keywords**

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

<table>
<thead>
<tr>
<th>and</th>
<th>as</th>
<th>assert</th>
<th>asr</th>
<th>begin</th>
<th>class</th>
</tr>
</thead>
<tbody>
<tr>
<td>constraint</td>
<td>do</td>
<td>done</td>
<td>downto</td>
<td>else</td>
<td>end</td>
</tr>
<tr>
<td>exception</td>
<td>external</td>
<td>false</td>
<td>for</td>
<td>fun</td>
<td>function</td>
</tr>
<tr>
<td>functor</td>
<td>if</td>
<td>in</td>
<td>include</td>
<td>inherit</td>
<td>initializer</td>
</tr>
<tr>
<td>land</td>
<td>lazy</td>
<td>let</td>
<td>lor</td>
<td>lsl</td>
<td>lsr</td>
</tr>
</tbody>
</table>
Chapter 6. The OCaml language

lxor match method mod module mutable
new nonrec object of open or
private rec sig struct then to
true try type val virtual when
while with

The following character sequences are also keywords:

!= # & && ' ( ) * + , -
-. -> . .. :: := => ; ;; <
<- = > >> ? [ [< ] ] [ ] |

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

parser value $ $$ $: <: << >> ??

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

\[
\text{linenum-directive ::= } \# \{0\ldots9\}^+ \\
\text{linenum-directive ::= } \# \{0\ldots9\}^+ " \{\text{string-character} \} "
\]

Preprocessors that generate OCaml 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 blanks 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 $2^{53} - 1$ for all arrays.

6.2.5 Variant values
Variant values are either a constant constructor, or a non-constant constructor applied to a number of values. The former case is written constr; the latter case is written constr (v$_1$, \ldots , v$_n$), where the v$_i$ are said to be the arguments of the non-constant constructor constr. The parentheses may be omitted if there is only one argument.

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 and $2^{30} - 1$ 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`, defined in section 6.1),
- polymorphic 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

\[
\text{value-name} ::= \text{lowercase-ident} \\
| (\text{operator-name})
\]

\[
\text{operator-name} ::= \text{prefix-symbol} | \text{infix-op}
\]

\[
\text{infix-op} ::= \text{infix-symbol} \\
| \ast | + | - | \cdot | = | != | < | > | \text{or} | || | \& | && | := \\
| \text{mod} | \text{land} | \text{lor} | \text{lxor} | \text{lsl} | \text{lsr} | \text{asr}
\]

\[
\text{constr-name} ::= \text{capitalized-ident}
\]

\[
\text{tag-name} ::= \text{capitalized-ident}
\]

\[
\text{typeconstr-name} ::= \text{lowercase-ident}
\]

\[
\text{field-name} ::= \text{lowercase-ident}
\]

\[
\text{module-name} ::= \text{capitalized-ident}
\]

\[
\text{modtype-name} ::= \text{ident}
\]

\[
\text{class-name} ::= \text{lowercase-ident}
\]

\[
\text{inst-var-name} ::= \text{lowercase-ident}
\]

\[
\text{method-name} ::= \text{lowercase-ident}
\]

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

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

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, module types, or class 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

\[
\text{typexpr} ::= \ 'ident \\
                 \ - \\
                 \ (\ \text{typexpr}) \\
                 \ [?\ ]\ \text{label-name}\ :\ \text{typexpr} \rightarrow \ \text{typexpr} \\
                 \ \text{typexpr}\ \{\ *\ \text{typexpr}\}\}^+ \\
                 \ \text{typeconstr} \\
                 \ \text{typexpr}\ \text{typeconstr} \\
                 \ (\ \text{typexpr}\ \{,\ \text{typexpr}\}\ )\ \text{typeconstr} \\
                 \ \text{typexpr}\ \text{as}\ \ 'ident \\
                 \ \text{polymorphic-variant-type} \\
                 \ <\ [\ldots]\ > \\
                 \ <\ \text{method-type}\ \{;\ \text{method-type}\}\ [;\ |\ ;\ldots]\ > \\
                 \ #\ \text{class-path} \\
                 \ \text{typexpr}\ #\ \text{class-path} \\
                 \ (\ \text{typexpr}\ \{,\ \text{typexpr}\}\ )\ #\ \text{class-path}
\]

\[
\text{poly-typexpr} ::= \ \text{typexpr} \\
                     \ {\ 'ident}\}^+\ .\ \text{typexpr}
\]

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

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>*</td>
<td>−</td>
</tr>
<tr>
<td>−→</td>
<td>right</td>
</tr>
<tr>
<td>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 ‘ident stands for the type variable named ident. The type expression _ stands for either an anonymous type variable or anonymous type parameters. 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 top-level phrase where it appears, and it 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 (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 \(\rightarrow\) typexpr₂ denotes the type of functions mapping arguments of type typexpr₁ to results of type typexpr₂.

- label-name : typexpr₁ \(\rightarrow\) typexpr₂ denotes the same function type, but the argument is labeled label.
- ? label-name : typexpr₁ \(\rightarrow\) 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 \(\rightarrow\) 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.

In the type expression _ typeconstr, the anonymous type expression _ stands in for anonymous type parameters and is equivalent to (_,...,_) with as many repetitions of _ as the arity of typeconstr.

**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

\[
\text{polymorphic-variant-type ::= [ tag-spec-first \{ | tag-spec \} ]}
| [\{ tag-spec \} \{ | tag-spec \} ]
| [\{ tag-spec-full \{ | tag-spec-full \} ]\{ tag-name \} ]^+ ]
\]

\[
tag-spec-first ::= \` tag-name [of typexpr]
| [typexpr] | tag-spec
\]

\[
tag-spec ::= \` tag-name [of typexpr]
| typexpr
\]

\[
tag-spec-full ::= \` tag-name [of [k] typexpr \& typexpr ]
| typexpr
\]

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 exact variant type (first case) 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 [of typexpr] form, or indirectly through a type expression, which must expand to an exact variant type, whose tag specifications are inserted in its place.

Full specifications of variant tags are only used for non-exact closed types. They can be understood 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. Conjunctive constraints are mainly intended as output from the type checker. When they are used in source programs, unsolvable constraints may cause early failures.

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 polymorphic variables have a local scope, and an explicit polymorphic type can only be unified to an equivalent one, where only the order and names of polymorphic variables may change.

The type <{method-type ;} .. > is the type of an object whose method names and 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 (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

```
class const ::= integer-literal |
float-literal |
char-literal |
string-literal |
constr |
false |
true |
( ) |
begin end |
[] |
[ | ] |
` 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, and `begin end`, which is equivalent to `( )`.  

6.6 Patterns

\[
\text{pattern} ::= \text{value-name} \\
| - \text{constant} \\
| \text{pattern as value-name} \\
| ( \text{pattern} ) \\
| ( \text{pattern} : \text{typexpr} ) \\
| \text{pattern} \mid \text{pattern} \\
| \text{constr pattern} \\
| `\text{tag-name pattern} \\
| # \text{typeconstr} \\
| \text{pattern}\{, \text{pattern}\}^+ \\
| \{ \text{field} [: \text{typexpr}] = \text{pattern} ; \text{field} [: \text{typexpr}] = \text{pattern} \} [;] \\
| [ \text{pattern} ; \text{pattern} ] [;] \\
| \text{pattern} :: \text{pattern} \\
| [\mid \text{pattern} ; \text{pattern} ] [;] [;] \\
| \text{char.literal} .. \text{char.literal}
\]

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>..</td>
<td>–</td>
</tr>
<tr>
<td>lazy (see section 7.3)</td>
<td>–</td>
</tr>
<tr>
<td>Constructor application,</td>
<td>right</td>
</tr>
<tr>
<td>Tag application</td>
<td></td>
</tr>
<tr>
<td>::</td>
<td>right</td>
</tr>
<tr>
<td>,</td>
<td>–</td>
</tr>
<tr>
<td></td>
<td></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 be bound several times by 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.
**Chapter 6. The OCaml language**

**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 `value-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` if it matches `pattern_1` or `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.

**Variant patterns**
The pattern `constr ( pattern_1 , . . . , pattern_n )` matches all variants whose constructor is equal to `constr`, and whose arguments match `pattern_1` . . . `pattern_n`. It is a type error if `n` is not the number of arguments expected by the constructor.

The pattern `constr _` matches all variants whose constructor is `constr`.

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`.

**Polymorphic variant abbreviation patterns**
If the type `[ ('a, 'b, . . .) typeconstr = [ ` tag-name_1 typexpr_1 | . . . | ` tag-name_n typexpr_n ] ]` is defined, then the pattern `# typeconstr` is a shorthand for the following or-pattern: `( ` tag-name_1 ( _ : typexpr_1 ) | . . . | ` tag-name_n ( _ : typexpr_n ) )`. It matches all values of type `[< typeconstr ]`. 

**Tuple patterns**

The pattern $pattern_1, \ldots, pattern_n$ matches $n$-tuples whose components match the patterns $pattern_1$ through $pattern_n$. That is, the pattern matches the tuple values $(v_1, \ldots, v_n)$ such that $pattern_i$ matches $v_i$ for $i = 1, \ldots, n$.

**Record patterns**

The pattern \{ field\_1 = pattern\_1; \ldots; 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 $i = 1, \ldots, n$. The record value can define more fields than field\_1 \ldots field\_n; the values associated to these extra fields are not taken into account for matching. Optional type constraints can be added field by field with \{ field\_1 : typexpr\_1 = pattern\_1; \ldots; field\_n : typexpr\_n = pattern\_n \} to force the type of field\_k to be compatible with typexpr\_k.

**Array patterns**

The pattern \[| pattern\_1; \ldots; pattern\_n |\] matches arrays of length $n$ such that the $i$-th array element matches the pattern pattern\_i, for $i = 1, \ldots, n$.

**Range patterns**

The pattern ‘$c$ ’..‘$d$ ’ is a shorthand for the pattern

‘$c$ ’ | ‘$c_1$ ’ | ‘$c_2$ ’ | \ldots | ‘$c_n$ ’ | ‘$d$ ’

where $c_1$, $c_2$, \ldots, $c_n$ are the characters that occur between $c$ and $d$ in the ASCII character set. For instance, the pattern ‘0’..‘9’ matches all characters that are digits.
6.7 Expressions

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

pattern-matching ::= [1] pattern [when expr] -> expr { | pattern [when expr] -> expr}

let-binding ::= pattern = expr
| value-name {parameter} [: typexpr] [:> 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>. . ( .[ .{ (see section 7.17)</td>
<td>-</td>
</tr>
<tr>
<td>#...</td>
<td>-</td>
</tr>
<tr>
<td>function application, constructor application, tag application, assert, lazy</td>
<td>left</td>
</tr>
<tr>
<td>- - . (prefix)</td>
<td>-</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>right</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

An expression consisting in a constant evaluates to this constant.

Value paths

An expression consisting in an access path evaluates 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 argument1...argumentn evaluates the expression expr and those appearing in argument1 to argumentn. The expression expr must evaluate to a functional value f, which is then applied to the values of argument1,...,argumentn.

The order in which the expressions expr, argument1,...,argumentn 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`:

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

This expression evaluates to a functional value with one argument. When this function is applied to a value \( v \), this value is matched against each pattern \( \text{pattern}_i \) 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`:

```
fun parameter₁...parameterₙ -> expr
```

This expression is equivalent to:

```
fun parameter₁ -> ...fun parameterₙ -> expr
```

An optional type constraint `typexpr` can be added before `->` to enforce the type of the result to be compatible with the constraint `typexpr`:

```
fun parameter₁...parameterₙ : typexpr -> expr
```

is equivalent to

```
fun parameter₁ -> ...fun parameterₙ -> (expr : typexpr)
```

Beware of the small syntactic difference between a type constraint on the last parameter

```
fun parameter₁... (parameterₙ : typexpr) -> expr
```

and one on the result

```
fun parameter₁...parameterₙ : typexpr -> expr
```

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

A function of the form `fun ? lab : ( pattern = expr₀ ) -> expr` is equivalent to

```
fun ? lab : ident -> let pattern = match ident with Some ident -> ident | None -> expr₀ in expr
```

where `ident` is a fresh variable, except that it is unspecified when `expr₀` is evaluated.

After these two transformations, expressions are of the form

```
fun [label₁] pattern₁ -> ...fun [labelᵦ] patternₙ -> expr
```
If we ignore labels, which will only be meaningful at function application, this is equivalent to

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

That is, the \texttt{fun} expression above evaluates to a curried function with \(n\) arguments: after applying this function \(n\) times to the values \(v_1 \ldots v_n\), 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 \texttt{expr} in an environment enriched by the bindings performed during the matchings. If the matching fails, the exception \texttt{Match_failure} is raised.

**Guards in pattern-matchings**

The cases of a pattern matching (in the \texttt{function}, \texttt{match} and \texttt{try} constructs) can include guard expressions, which are arbitrary boolean expressions that must evaluate to \texttt{true} for the match case to be selected. Guards occur just before the \(\rightarrow\) token and are introduced by the \texttt{when} keyword:

\[
\text{function} \quad \text{pattern}_1 \begin{cases} \text{when} \text{cond}_1 & \rightarrow \text{expr}_1 \\ \ldots \\ \text{when} \text{cond}_n & \rightarrow \text{expr}_n \end{cases}
\]

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

**Local definitions**

The \texttt{let} and \texttt{let rec} constructs bind value names locally. The construct

\[
\text{let } \text{pattern}_1 = \text{expr}_1 \land \ldots \land \text{pattern}_n = \text{expr}_n \text{ in } \text{expr}
\]

evaluates \(\text{expr}_1 \ldots \text{expr}_n\) in some unspecified order and matches their values against the patterns \(\text{pattern}_1 \ldots \text{pattern}_n\). If the matchings succeed, \texttt{expr} is evaluated in the environment enriched by the bindings performed during matching, and the value of \texttt{expr} is returned as the value of the whole \texttt{let} expression. If one of the matchings fails, the exception \texttt{Match_failure} is raised.

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

\[
\text{let ident = fun parameter}_1 \ldots \text{parameter}_m \rightarrow \text{expr}
\]

in a \texttt{let} expression, one may instead write

\[
\text{let ident parameter}_1 \ldots \text{parameter}_m = \text{expr}
\]

Recursive definitions of names are introduced by \texttt{let rec}:

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

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

$$\text{let rec } \text{name}_1 = \text{fun ... and ... and } \text{name}_n = \text{fun ... in expr}$$

This defines $\text{name}_1$ ... $\text{name}_n$ as mutually recursive functions local to $\text{expr}$.

The behavior of other forms of \texttt{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 \ref{sec:recursive-values}.

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

Case expression

The expression

$$\text{match } \text{expr} \text{ with } \text{pattern}_1 \rightarrow \text{expr}_1$$
$$\phantom{\text{match } \text{expr}} | \ldots$$
$$\phantom{\text{match } \text{expr}} | \text{pattern}_n \rightarrow \text{expr}_n$$

matches the value of $\text{expr}$ against the patterns $\text{pattern}_1$ to $\text{pattern}_n$. If the matching against $\text{pattern}_n$ succeeds, the associated expression $\text{expr}_i$ is evaluated, and its value becomes the value of the whole \texttt{match} expression. The evaluation of $\text{expr}_i$ 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 \texttt{match} expression is selected. If none of the patterns match the value of $\text{expr}$, the exception \texttt{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
if \( expr_1 \) then \( expr_2 \) else false.

The expression \( expr_1 \mid\mid expr_2 \) evaluates to true if one of the expressions \( expr_1 \) and \( expr_2 \) evaluates to true; otherwise, it evaluates to false. The first component, \( expr_1 \), is evaluated first. The second component, \( expr_2 \), is not evaluated if the first component evaluates to true. Hence, the expression \( expr_1 \mid\mid expr_2 \) behaves exactly as

\[
\text{if } expr_1 \text{ then true else } expr_2.
\]

The boolean operators \& and or are deprecated synonyms for (respectively) \&\& and ||.

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 \( n, n + 1, \ldots, p - 1, p \). 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 \( n, n - 1, \ldots, p + 1, p \). 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

\[
\text{try} \quad expr \\
\text{with} \quad pattern_1 \rightarrow expr_1 \\
\quad \ldots \\
\quad pattern_n \rightarrow expr_n
\]

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 \( pattern_i \) succeeds, the associated expression \( expr_i \) is evaluated, and its value becomes the value of the whole try expression. The evaluation of \( expr_i \) takes place in an environment enriched by the bindings performed during matching. If several patterns match the value of expr, the one that occurs first in the try expression is selected. If none of the patterns matches the value of expr, the exception value is raised again, thereby transparently “passing through” the try construct.

6.7.3 Operations on data structures

Products

The expression \( expr_1, \ldots, expr_n \) evaluates to the \( n \)-tuple of the values of expressions \( expr_1 \) to \( expr_n \). The evaluation order of the subexpressions is not specified.
Variants

The expression `constr expr` evaluates to the unary variant value whose constructor is `constr`, and whose argument is the value of `expr`. Similarly, the expression `constr (expr_1, ..., expr_n)` evaluates to the n-ary variant value whose constructor is `constr` and whose arguments are the values of `expr_1, ..., expr_n`.

The expression `constr (expr_1, ..., expr_n)` evaluates to the variant value whose constructor is `constr`, and whose arguments are the values of `expr_1` to `expr_n`.

For lists, some syntactic sugar is provided. The expression `expr_1 :: expr_2` stands for the constructor `(: :)` applied to the arguments `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 type; each field of 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. Optional type constraints can be added after each field `{field_1 : typexpr_1 = expr_1 ; ... ; field_n : typexpr_n = expr_n}` to force the type of `field_k` to be compatible with `typexpr_k`.

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`. As previously, it is possible to add an optional type constraint on each field being updated with `{expr with field_1 : typexpr_1 = expr_1 ; ... ; field_n : typexpr_n = 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 `expr1.(expr2)` returns the value of element number `expr2` in the array denoted by `expr1`. 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 `expr1.(expr2) <- expr3` modifies in-place the array denoted by `expr1`, replacing element number `expr2` by the value of `expr3`. The exception `Invalid_argument` is raised if the access is out of bounds. The value of the whole expression is `()`. 

### Strings

The expression `expr1.[expr2]` returns the value of character number `expr2` in the string denoted by `expr1`. 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 `expr1.[expr2] <- expr3` modifies in-place the string denoted by `expr1`, replacing character number `expr2` by the value of `expr3`. The exception `Invalid_argument` is raised if the access is out of bounds. The value of the whole expression is `()`. 

**Note:** this possibility is offered only for backward compatibility with older versions of OCaml and will be removed in a future version. New code should use byte sequences and the `Bytes.set` function.

### 6.7.4 Operators

Symbols from the class `infix-symbol`, as well as the keywords `*`, `+`, `-`, `=`, `!=`, `<`, `>`, `or`, `||`, `&`, `&&`, `:=`, `mod`, `land`, `lor`, `lxor`, `ls1`, `lsr`, and `asr` can appear in infix position (between two expressions). Symbols from the class `prefix-symbol`, as well as the keywords `-` and `-.` 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 20 for more details). Their meaning may be changed at any time using `let (infix-op) name1 name2 =...`

**Note:** the operators `&&`, `||`, and `~-` are handled specially and it is not advisable to change their meaning.

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

### 6.7.5 Objects

**Object creation**

When `class-path` evaluates to a class body, `new class-path` evaluates to a new 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.

Method invocation

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 `O0.copy` (see section 21.24). 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 `expr` from a type `typ₁` to a type `typ₂` even if type `typ₁` is a subtype of type `typ₂`: 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 `expr` and `typ` are ground (i.e. do not contain type variables), the former operator behaves as the latter one, taking the inferred type of `expr` as `typ₁`. In case of failure with the former operator, the latter one should be used.
It is only possible to coerce an expression \( expr \) from type \( \text{typ}_1 \) to type \( \text{typ}_2 \), if the type of \( expr \) is an instance of \( \text{typ}_1 \) (like for a type annotation), and \( \text{typ}_1 \) is a subtype of \( \text{typ}_2 \). The type of the coerced expression is an instance of \( \text{typ}_2 \). If the types contain variables, they may be instantiated by the subtyping algorithm, but this is only done after determining whether \( \text{typ}_1 \) is a potential subtype of \( \text{typ}_2 \). This means that typing may fail during this latter unification step, even if some instance of \( \text{typ}_1 \) is a subtype of some instance of \( \text{typ}_2 \). In the following paragraphs we describe the subtyping relation used.

**Object types**

A fixed object type admits as subtype any object type that includes 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 .. if every \( \text{typ}_i \) is a supertype of the corresponding \( \text{typ}'_i \).

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 a 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,

\[
[ [\times] \ ` C_1 \text{ of } \text{typ}_1 \ ` \ldots \ ` C_n \text{ of } \text{typ}_n ]
\]

which may be a shrinkable type, is a subtype of

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

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: \( \text{typ}_1 \rightarrow \text{typ}_2 \) is a subtype of \( \text{typ}'_1 \rightarrow \text{typ}'_2 \) 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 array or 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 (see section 6.8.1), otherwise the default is nonvariant. This is also the case for constrained arguments in type definitions.

6.7.7 Other

Assertion checking

OCaml supports the assert construct to check debugging assertions. The expression assert expr evaluates the expression expr and returns () if expr evaluates to true. If it evaluates to false 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. In this case, expr is not evaluated at all.

As a special case, assert false is reduced to raise (Assert_failure ...), which gives it a polymorphic type. This means that it can be used in place of any expression (for example as a branch of any pattern-matching). It also means that the assert false “assertions” cannot be turned off by the -noassert option.

Lazy expressions

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 the function 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 (see 7.3).

Local modules

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

```
let remove_duplicates comparison_fun string_list =
  let module StringSet =
    Set.Make(struct type t = string
              let compare = comparison_fun end) in
  StringSet.elements (List.fold_right StringSet.add string_list StringSet.empty)
```
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 [nonrec] 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 ]}
                         | = record-decl
type-params ::= type-param
               | ( type-param {, type-param} )
type-param ::= [variance] ' ident
variance ::= +
           | -
record-decl ::= { field-decl {; field-decl} [;] }
constr-decl ::= ( constr-name | [] | [:]) [of constr-args]
constr-args ::= typexpr {* typexpr}
field-decl ::= [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.

In the right-hand side of type definitions, references to one of the type constructor name being defined are considered as recursive, unless `type` is followed by `nonrec`. The `nonrec` keyword was introduced in OCaml 4.02.2.

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, optionally restricted 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 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 less restrictive; otherwise (i.e. for abstract types or non-free parameters), the variance must be given explicitly, and the parameter is invariant if no variance is given.

The optional type equation \( = \) typexpr makes the defined type equivalent to the type expression typexpr: 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-decl1, ..., constr-decln describe the constructors associated to this variant type. The constructor declaration constr-name \( \) of typexpr1 \( \ast \) ... \( \ast \) typexprn declares the name constr-name as a non-constant constructor, whose arguments have types typexpr1 \( \ast \) ... typexprn. 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-decl1, ..., field-decln 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 addition, it allows physical modification of this field. Immutable fields are covariant, mutable fields are non-variant. Both mutable and immutable fields may have a explicitly polymorphic types. The polymorphism of the contents is statically checked whenever a record value is created or modified. Extracted values may have their types instantiated.

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

**Abstract type: no equation, no representation.**

When appearing in a module signature, this definition specifies nothing on the type constructor, besides its number of parameters: its representation is hidden and it is assumed incompatible with any other type.

**Type abbreviation: an equation, no representation.**

This defines the type constructor as an abbreviation for the type expression on the right of the \( = \) sign.

**New variant type or record type: no equation, a representation.**

This generates a new type constructor and defines associated constructors or 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 \( \tau \) is a subtype of the type \( \sigma \), then \( \tau \ t \) is a subtype of \( \sigma \ t \). Similarly, type -'a t declares that the abstract type t is contravariant in its parameter: if \( \tau \) is a subtype of \( \sigma \), then \( \sigma \ t \) is a subtype of \( \tau \ t \). If no + or - variance annotation is given, the type constructor is assumed non-variant in the corresponding parameter. For instance, the abstract type declaration type 'a t means that \( \tau \ t \) is neither a subtype nor a supertype of \( \sigma \ t \) if \( \tau \) is subtype of \( \sigma \).

The variance indicated by the + and - annotations on parameters is enforced only for abstract and private types, or when there are type constraints. Otherwise, for abbreviations, variant and record types without type constraints, 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 the specification of 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

\[
\text{exception-definition ::= \ exception constr-decl} \\
| \ \ exception constr-name = constr
\]

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

The form exception constr-decl 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.
**Simple class expressions**

The expression classtype-path is equivalent to the class type bound to the name classtype-path. Similarly, the expression \([\text{typexpr}_1, \ldots, \text{typexpr}_n]\) classtype-path is equivalent to the parametric class type bound to the name classtype-path, in which type parameters have been instantiated 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 \([(\text{typexpr} )]\) \{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 name 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 body type, and these definitions meet the type requirements given in the class body type. Furthermore, all methods either virtual or public present in the class body must also be present in the class body 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-body-type provides for inclusion of methods and instance variables from other class types. The instance variable and method types from class-body-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.

If several specifications are present for the same method, they must have compatible types. Any non-private specification of a method forces it to be public.

Virtual method specification

A 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 typexpr₁ = typexpr₂` forces the two type expressions to be equal. This is typically used to specify type parameters: in this way, they can be bound to specific type expressions.

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.

```
class-expr ::= class-path
| [ typexpr {, typexpr} ] class-path
| ( class-expr )
| ( class-expr : class-type )
| class-expr {argument}+
| fun {parameter}+ -> class-expr
| let [rec] let-binding {and let-binding} in class-expr
| object class-body end
```
class-field ::= inherit class-expr [as lowercase-ident]  
| val [mutable] inst-var-name [: typexpr] = expr  
| val [mutable] virtual inst-var-name : typexpr  
| val [mutable] virtual inst-var-name : typexpr  
| method [private] method-name {parameter} [: typexpr] = expr  
| method [private] method-name : poly-typexpr = expr  
| method [private] virtual method-name : poly-typexpr  
| method virtual private method-name : poly-typexpr  
| constraint typexpr = typexpr  
| initializer expr

Simple class expressions

The expression class-path evaluates to the class bound to the name class-path. Similarly, the expression [ typexpr_1, ..., typexpr_n ] class-path evaluates to the parametric class bound to the name class-path, in which type parameters have been instantiated respectively to typexpr_1, ..., typexpr_n.

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

The expression ( class-expr : class-type ) checks that class-type matches 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 first expression applied to the given arguments. The arguments are evaluated as 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 -> 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

fun parameter_1...parameter_n -> class-expr

is a short form for

fun parameter_1 -> ... fun parameter_n -> expr
Local definitions

The `let` and `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

\[
\text{class-body} ::= \left\{ \begin{array}{l}
\text{pattern} \mid \text{typexpr} \\
\end{array} \right\} \{\text{class-field}\}
\]

The expression \text{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 \( \text{pattern} \mid \text{typexpr} \) 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.

Since OCaml 4.01, it is an error if the same method or instance variable name is defined several times in the same class body.

Inheritance

The inheritance construct \text{inherit class-expr} allows reusing methods and instance variables from other classes. The class expression \text{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 method of the same name.

An ancestor can be bound by appending \text{as lowercase-ident} to the inheritance construct. \text{lowercase-ident} is not a true variable and can only be used to select a method, i.e. in an expression \text{lowercase-ident} \# \text{method-name}. This gives access to the method \text{method-name} as it was defined in the parent class even if it is redefined in the current class. The scope of this 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 \text{val [mutable] inst-var-name = expr} adds an instance variable \text{inst-var-name} whose initial value is the value of expression \text{expr}. The flag \text{mutable} allows physical modification of this variable by methods.

An instance variable can only be used in the methods and initializers that follow its definition.

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

A 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

A 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:

\[
\begin{align*}
expr & := \ldots \\
& \mid \text{inst-var-name} <- \text{expr} \\
& \mid \{ \text{inst-var-name} = \text{expr} ; \text{inst-var-name} = \text{expr} \} [;] >
\end{align*}
\]

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

The expression `{< inst-var-name_1 = expr_1 ; \ldots ; inst-var-name_n = expr_n >}` 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

A 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 must be explicitly polymorphic.

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: in that way they can be bound to specific type expressions.
Chapter 6. The OCaml language

Initializers

A class initializer initializer expr specifies an expression that will be evaluated whenever an object is created from the class, once all its 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 specifications

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

classtype-definition ::= class type classtype-def {and classtype-def}

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

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

### 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.

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

mod-constraint ::= type [type-params] typeconstr type-equation {type-constraint}
                 | module module-path = extended-module-path
```

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

An optional ;; is allowed after each specification in a signature. It serves as a syntactic separator with 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 { ... 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 ) ... ( namen : module-typen ) : 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.

If the type module-type_2 does not depend on type components of module-name, the module type expression can be simplified with the alternative short syntax module-type_1 -> module-type_2. 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 \{ typeconstr = 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) ...
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 scopes 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), optional \texttt{;;} are allowed after and before each definition in a structure. These \texttt{;;} have no semantic meanings. Similarly, an \texttt{expr} preceded by \texttt{;;} is allowed as a component of a structure. It is equivalent to \texttt{let _ = expr}, i.e. \texttt{expr} is evaluated for its side-effects but is not bound to any identifier. If \texttt{expr} is the first component of a structure, the preceding \texttt{;;} can be omitted.

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 module-name = module-expr}, which evaluates \texttt{module-expr} and binds the result to the name \texttt{module-name}.

One can write
module module-name : module-type = module-expr

instead of

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

Another derived form is

module module-name ( name_1 : module-type_1 ) ... ( name_n : module-type_n ) = module-expr

which is equivalent to

module module-name = functor ( name_1 : module-type_1 ) -> ... -> module-expr

Module type definitions

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

Opening a module path

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

Including the components of another structure

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

\[
\text{struct type } t = \text{int let } x = 2 \text{ end}
\]

the module expression

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

is equivalent to the module expression

\[
\text{struct type } t = S.t \text{ let } x = S.x \text{ let y = (x + 1 : t) end}
\]

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

6.11.3 Functors

Functor definition

The expression functor ( module-name : module-type ) -> module-expr evaluates to a functor that takes as argument modules of the type module-type_1, binds module-name to these modules, evaluates module-expr in the extended environment, and returns the resulting modules as results. No restrictions are placed on the type of the functor argument; in particular, a functor may take another functor as argument ("higher-order" functor).
Functor application

The expression \texttt{module-expr} \_1 \ ( \texttt{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{module-items} ]
\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}...\texttt{end} signature expression. The implementation contains a sequence of definitions and expressions, just as the inside of a \texttt{struct}...\texttt{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 \ldots name \_n : sig specification \_n end
\end{verbatim}

where \texttt{name} \_1 \ldots \texttt{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 \ldots \texttt{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)

constant ::= ...
   | int32-literal
   | int64-literal
   | nativeint-literal

int32-literal ::= integer-literal 1
int64-literal ::= integer-literal L
nativeint-literal ::= integer-literal n

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[21.15], Int64[21.16] and Nativeint[21.23] provide operations on these integer types.

7.2 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

let rec name1 = 1 :: name2 and name2 = 2 :: name1 in expr

which binds name1 to the cyclic list 1::2::1::2::..., and name2 to the cyclic list 2::1::2::1::... Informally, the class of accepted definitions consists of those definitions where the defined names occur only inside function bodies or as argument to a data constructor.
More precisely, consider the expression:

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

It will be accepted if each one of expr\(_1\)\ldots expr\(_n\) is statically constructive with respect to name\(_1\)\ldots name\(_n\), is not immediately linked to any of name\(_1\)\ldots 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 name\(_1\)\ldots name\(_n\) if at least one of the following conditions is true:

- \(e\) has no free occurrence of any of name\(_1\)\ldots name\(_n\)
- \(e\) is a variable
- \(e\) has the form \(\text{fun} \ldots \rightarrow \ldots\)
- \(e\) has the form \(\text{function} \ldots \rightarrow \ldots\)
- \(e\) has the form \(\text{lazy (} \ldots \))
- \(e\) has one of the following forms, where each one of expr\(_1\)\ldots expr\(_m\) is statically constructive with respect to name\(_1\)\ldots name\(_n\), and expr\(_0\) is statically constructive with respect to name\(_1\)\ldots name\(_n\), xname\(_1\)\ldots xname\(_m\):
  - \(\text{let } [\text{rec}] \ \text{xname}_1 = \text{expr}_1 \text{ and } \ldots \text{ and xname}_m = \text{expr}_m \text{ in expr}_0\)
  - \(\text{let module } \ldots \text{ in expr}_1\)
  - \(\text{constr ( expr}_1 , \ldots , \text{expr}_m )\)
  - \(\text{` tag-name ( expr}_1 , \ldots , \text{expr}_m )\)
  - \(\text{[ expr}_1 ; \ldots ; \text{expr}_m ]\)
  - \(\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}
    \text{ 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 name in the following cases:

- \(e\) is name
- \(e\) has the form \(\text{expr}_1 ; \ldots ; \text{expr}_m\) where \(\text{expr}_m\) is immediately linked to name
- \(e\) has the form \(\text{let } [\text{rec}] \ \text{xname}_1 = \text{expr}_1 \text{ and } \ldots \text{ and xname}_m = \text{expr}_m \text{ in expr}_0\) where expr\(_0\) is immediately linked to name or to one of the xname\(_i\) such that expr\(_i\) is immediately linked to name.
7.3 Lazy patterns

(Introduced in Objective Caml 3.11)

\[ \text{pattern ::= ...} \]

\[ | \text{lazy pattern} \]

The pattern `lazy pattern` matches a value \( v \) 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 (section 21.17).

7.4 Recursive modules

(Introduced in Objective Caml 3.07)

\[ \text{definition ::= ...} \]

\[ | \text{module rec module-name : module-type = module-expr} \]

\[ \{\text{and module-name : module-type = module-expr}\} \]

\[ \text{specification ::= ...} \]

\[ | \text{module rec module-name : module-type} \{\text{and module-name : module-type}\} \]

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
```

```ocaml
= 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:

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

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 typexpr1 -> typexpr2. Evaluation of a recursive module definition proceeds by building initial values for the safe modules involved, binding all (functional) values to fun _ -> raiseUndefined_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.

Note that, in the specification case, the module-types must be parenthesized if they use the with mod-constraint construct.

7.5 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 7.5.1), for type abbreviations (section 7.5.2), and for row types (section 7.5.3).

7.5.1 Private variant and record types

(Introduced in Objective Caml 3.07)

    type-representation ::= ...
        | = private [1] constr-decl {1 constr-decl}
        | = private record-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 `-'o, it is invariant otherwise.

### 7.5.2 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:

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

Note that the coercion \( (\text{expr} :> \text{typexpr}) \) is actually an abbreviated form, and will only work in presence of private abbreviations if neither the type of \( \text{expr} \) nor \( \text{typexpr} \) contain any type variables. If they do, you must use the full form \( (\text{expr} : \text{typexpr}_1 :> \text{typexpr}_2) \) where \( \text{typexpr}_1 \) is the expected type of \( \text{expr} \). Concretely, this would be \( (x : N.t :> \text{int}) \) and \( (\mathbf{l} : N.t \ \text{list} :> \text{int} \ \text{list}) \) for the above examples.

### 7.5.3 Private row types

(Introduced in Objective Caml 3.09)

\[
\text{type-equation} \ ::= \ ...
\mid = \text{private} \ \text{typexpr}
\]

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
```

A polymorphic variant type \([t]\), for example

\[
t = [ \text{`A of int} \mid \text{`B of bool} ]
\]
can be refined in two ways. A definition \( u \) may add new field to \( t \), and the declaration

\[
\text{type } u = \text{private } [> t]
\]

will keep those new fields abstract. Construction of values of type \( u \) is possible using the known variants of \( t \), but any pattern-matching will require a default case to handle the potential extra fields. Dually, a declaration \( u \) may restrict the fields of \( t \) through abstraction: the declaration

\[
\text{type } v = \text{private } [< t > \text{`A}]
\]

corresponds to private variant types. One cannot create a value of the private type \( v \), except using the constructors that are explicitly listed as present, \((\text{`A } n)\) in this example; yet, when pattern-matching on a \( v \), one should assume that any of the constructors of \( t \) could be present.

Similarly to abstract types, the variance of type parameters is not inferred, and must be given explicitly.

### 7.6 Local opens

(Introduced in OCaml 3.12, extended to patterns in 4.03)

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

\[
\text{pattern ::= ...}
| \text{module-path . ( pattern )}
\]

The expressions \( \text{let open module-path in expr} \) and \( \text{module-path . ( expr )} \) are strictly equivalent. On the pattern side, only the pattern \( \text{module-path . ( pattern )} \) is available. These constructions locally open the module referred to by the module path \( \text{module-path} \) in the respective scope of the expression \( \text{expr} \) or pattern \( \text{pattern} \).

Restricting opening to the scope of a single expression or pattern 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).

**Local opens for delimited expressions or patterns**  
(Introduced in OCaml 4.02)

\[
\text{expr ::= ...}
| \text{module-path . [ expr ]}
| \text{module-path . [ | expr | ]}
| \text{module-path . { expr }}
| \text{module-path . {< expr >}}
\]

\[
\text{pattern ::= ...}
| \text{module-path . [ pattern ]}
| \text{module-path . [ | pattern | ]}
| \text{module-path . { pattern }}
\]
When the body of a local open expression or pattern is delimited by [], [ | | ], or { }, the parentheses can be omitted. For expression, parentheses can also be omitted for {< >}. For example, module-path . [ expr ] is equivalent to module-path . ([ expr ]), and module-path . [ | pattern | ] is equivalent to module-path . ([ | pattern | ]).

7.7 Record and object notations

(Introduced in OCaml 3.12, object copy notation added in Ocaml 4.03)

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

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

In a record pattern, a record construction expression or an object copy expression, a single identifier id stands for id = id, and a qualified identifier module-path . id stands for module-path . id = id. For example, assuming the record type

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

has been declared, the following expressions are equivalent:

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

On the object side, all following methods are equivalent:

\[
\text{object}
\text{\quad} \text{val } x=0. \text{ val } y=0. \text{ val } z=0.
\text{\quad} \text{method } f_0 x y = \{ < x; y > \}
\text{\quad} \text{method } f_1 x y = \{ < x = x; y > \}
\text{\quad} \text{method } f_2 x y = \{ < x=x ; y = y > \}
\text{\quad} \text{end}
\]

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 warning 9 is turned on, the compiler will warn when a record pattern fails to list all fields of the corresponding record type and is not terminated by ; _. Continuing the point example above,
fun {x} -> x .+ 1.
will warn if warning 9 is on, while
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 are added to a record type.

7.8 Explicit polymorphic type annotations
(Introduced in OCaml 3.12)

let-binding ::= ...
    | value-name : poly-typexpr = expr

Polymorphic type annotations in 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 let rec). Note however that this is a normal polymorphic type, unifiable with any instance of itself.

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

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 'b is not explicitly polymorphic here, and it will actually be unified with int.

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

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

7.9 Locally abstract types
(Introduced in OCaml 3.12, short syntax added in 4.03)

parameter ::= ...
    | (type {typeconstr-name}+)

The expression fun ( type typeconstr-name ) -> expr introduces a type constructor named 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 fun ( type typeconstr-name ) -> expr itself does not suspend the evaluation of 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:
let \( f = \text{fun (type } t \text{) (foo : } t \text{ list) -> ...} \)

and even use the alternative syntax for declaring functions:

\[
\text{let } f \ (\text{type } t) \ (\text{foo : } t \text{ list}) = ...
\]

If several locally abstract types need to be introduced, it is possible to use the syntax \( \text{fun ( type } \text{typeconstr-name}_1 \ldots \text{typeconstr-name}_n \text{ ) -> expr} \) as syntactic sugar for \( \text{fun ( type } \text{typeconstr-name}_1 \text{ ) -> } \ldots \text{ -> fun ( type } \text{typeconstr-name}_n \text{ ) -> expr} \). For instance,

\[
\text{let } f = \text{fun (type } t \text{ u v) -> fun (foo : } (t \text{ * } u \text{ * } v) \text{ list) -> ...}
\]

\[
\text{let } f' \ (\text{type } t \text{ u v) (foo : } (t \text{ * } u \text{ * } v) \text{ list}) = ...
\]

This construction is useful because the type constructors 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.

\[
\text{let } f \ (\text{type } t) () =
\text{let module } M = \text{struct } \text{exception } E \text{ of } t \text{ end } \text{in}
\text{(fun } x \text{ -> } M.E \text{ x}, \text{function } M.E \text{ x} \text{ -> } \text{Some } x \mid _{\_} \text{ -> None})
\]

Here is another example:

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

It is also extremely useful for first-class modules (see section \[7.10\]) and generalized algebraic datatypes (GADTs: see section \[7.16\]).

**Polymorphic syntax** (Introduced in OCaml 4.00)

\[
\text{let-binding ::= }...
\]

\[
\text{| value-name : type } \{ \text{typeconstr-name}\}^+. \text{typexpr = expr}
\]

\[
\text{class-field ::= }...
\]

\[
\text{| method [private] method-name : type } \{ \text{typeconstr-name}\}^+. \text{typexpr = expr}
\]

\[
\text{| method! [private] method-name : type } \{ \text{typeconstr-name}\}^+. \text{typexpr = expr}
\]

The \( \text{(type } \text{typeconstr-name} \) syntax construction by itself does not make polymorphic the type variable it introduces, but it can be combined with explicit polymorphic annotations where needed. The above rule is provided as syntactic sugar to make this easier:

\[
\text{let rec } f : \text{type } t1 \text{ t2. } t1 \ast t2 \text{ list } \rightarrow t1 = ...
\]

is automatically expanded into

\[
\text{let rec } f : 't1 't2. 't1 \ast 't2 \text{ list } \rightarrow 't1 =
\text{fun (type } t1) \ (\text{type } t2) \rightarrow (...) : t1 \ast t2 \text{ list } \rightarrow t1)
\]

This syntax can be very useful when defining recursive functions involving GADTs, see the section \[7.16\] for a more detailed explanation.

The same feature is provided for method definitions. The \text{method!} form combines this extension with the “explicit overriding” extension described in section \[7.14\].
Chapter 7. Language extensions

7.10 First-class modules

(Introduced in OCaml 3.12; pattern syntax and package type inference introduced in 4.00; structural comparison of package types introduced in 4.02.)

\[
typexpr ::= ... \\
| (module package-type )
\]

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

\[
expr ::= ...
| (module module-expr [ package-type ])
\]

\[
pattern ::= ...
| (module module-name [ package-type ])
\]

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

\[
package-constraint ::= type typeconstr = typexpr
\]

Modules are typically thought of 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 (module module-expr : package-type) converts the module (structure or functor) denoted by module expression module-expr to a value of the core language that encapsulates this module. The type of this core language value is (module package-type). The package-type annotation can be omitted if it can be inferred from the context.

Conversely, the module expression (val expr : package-type) evaluates the core language expression expr to a value, which must have type module package-type, and extracts the module that was encapsulated in this value. Again package-type can be omitted if the type of expr is known.

The pattern (module module-name : package-type) matches a package with type package-type and binds it to module-name. It is not allowed in toplevel let bindings. Again package-type can be omitted if it can be inferred from the enclosing pattern.

The package-type syntactic class appearing in the (module package-type) type expression and in the annotated forms 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 (and starting from OCaml 4.02), package types are compared using the structural comparison of module types.

In general, the module expression (val expr : package-type) cannot be used in the body of a functor, because this could cause unsoundness in conjunction with applicative functors. Since OCaml 4.02, this is relaxed in two ways: if package-type does not contain nominal type declarations (i.e. types that are created with a proper identity), then this expression can be used anywhere, and even if it contains such types it can be used inside the body of a generative functor, described in section 7.23. It can also be used anywhere in the context of a local module binding let module module-name = (val expr₁ : package-type) in expr₂.
**Basic example**  A typical use of first-class modules is to select at run-time among several implementations of a signature. 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:

```plaintext
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:

```plaintext
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:

```plaintext
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.

```plaintext
let sort (type s) (module Set : Set.S with type elt = s) l = 
  Set.elements (List.fold_right Set.add l Set.empty)
val sort : (module Set.S with type elt = 'a) -> 'a list -> 'a list
```

To use this function, one can wrap the Set.Make functor:

```plaintext
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)
val make_set : ('a -> 'a -> int) -> (module Set.S with type elt = 'a)
Chapter 7. Language extensions

7.11 Recovering the type of a module

(Introduced in OCaml 3.12)

\[
\text{module-type ::= ...}
\]

\[
\mid \text{module type of module-expr}
\]

The construction \text{module type of module-expr} expands to the module type (signature or functor type) inferred for the module expression \text{module-expr}. To make this module type reusable in many situations, it is intentionally not strengthened: abstract types and datatypes are not explicitly related with the types of the original module. For the same reason, module aliases in the inferred type are expanded.

A typical use, in conjunction with the signature-level \text{include} construct, is to extend the signature of an existing structure. In that case, one wants to keep the types equal to types in the original module. This can done using the following idiom.

\[
\text{module type MYHASH = sig}
\]

\[
\text{include module type of struct include Hashtbl end}
\]

\[
\text{val replace: ('a, 'b) t -> 'a -> 'b -> unit}
\]

\[
\text{end}
\]

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

\[
\text{module MyHash : MYHASH = struct}
\]

\[
\text{include Hashtbl}
\]

\[
\text{let replace t k v = remove t k; add t k v}
\]

\[
\text{end}
\]

Another application where the absence of strengthening comes handy, is to provide an alternative implementation for an existing module.

\[
\text{module MySet : module type of Set = struct}
\]

\[
\text{...}
\]

\[
\text{end}
\]

This idiom guarantees that \text{Myset} is compatible with \text{Set}, but allows it to represent sets internally in a different way.

7.12 Substituting inside a signature

(Introduced in OCaml 3.12)

\[
\text{mod-constraint ::= ...}
\]

\[
\mid \text{type [type-params] typeconstr-name := typexpr}
\]

\[
\mid \text{module module-name := extended-module-path}
\]
“Destructive” substitution (with... :=...) behaves essentially like normal signature constraints (with... =...), 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 in the case of with type the definition must be another type constructor with the same type parameters.

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

```ocaml
class Printable = class
  type t
  val print : Format.formatter -> t -> unit
end;
class Comparable = class
  type t
  val compare : t -> t -> int
end;
class PrintableComparable = class
  include Printable
  include Comparable with type t := t
end;
```

One can also use this to completely remove a field:

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

or to rename one:

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

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

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

### 7.13 Type-level module aliases

(Introduced in OCaml 4.02)

```ocaml
specification ::= ...
  module module-name = module-path
```
Chapter 7. Language extensions

The above specification, inside a signature, only matches a module definition equal to `module-path`. Conversely, a type-level module alias can be matched by itself, or by any supertype of the type of the module it references.

There are several restrictions on `module-path`:

1. it should be of the form $M_0.M_1...M_n$ (i.e. without functor applications);
2. inside the body of a functor, $M_0$ should not be one of the functor parameters;
3. inside a recursive module definition, $M_0$ should not be one of the recursively defined modules.

Such specifications are also inferred. Namely, when $P$ is a path satisfying the above constraints, `# module N = P` has type `module N = P`.

Type-level module aliases are used when checking module path equalities. That is, in a context where module name $N$ is known to be an alias for $P$, not only these two module paths check as equal, but $F(N)$ and $F(P)$ are also recognized as equal. In the default compilation mode, this is the only difference with the previous approach of module aliases having just the same module type as the module they reference.

When the compiler flag `-no-alias-deps` is enabled, type-level module aliases are also exploited to avoid introducing dependencies between compilation units. Namely, a module alias referring to a module inside another compilation unit does not introduce a link-time dependency on that compilation unit, as long as it is not dereferenced; it still introduces a compile-time dependency if the interface needs to be read, i.e. if the module is a submodule of the compilation unit, or if some type components are referred to. Additionally, accessing a module alias introduces a link-time dependency on the compilation unit containing the module referenced by the alias, rather than the compilation unit containing the alias. Note that these differences in link-time behavior may be incompatible with the previous behavior, as some compilation units might not be extracted from libraries, and their side-effects ignored.

These weakened dependencies make possible to use module aliases in place of the `-pack` mechanism. Suppose that you have a library `Mylib` composed of modules `A` and `B`. Using `-pack`, one would issue the command line

```
ocamlc -pack a.cmo b.cmo -o mylib.cmo
```

and as a result obtain a `Mylib` compilation unit, containing physically `A` and `B` as submodules, and with no dependencies on their respective compilation units. Here is a concrete example of a possible alternative approach:

1. Rename the files containing `A` and `B` to `Mylib_A` and `Mylib_B`.
2. Create a packing interface `Mylib.ml`, containing the following lines.

```ml
module A = Mylib_A
module B = Mylib_B
```
3. Compile `Mylib.ml` using `-no-alias-deps`, and the other files using `-no-alias-deps` and `-open Mylib` (the last one is equivalent to adding the line `open! Mylib` at the top of each file).

```ocaml
ocamlc -c -no-alias-deps Mylib.ml
ocamlc -c -no-alias-deps -open Mylib Mylib_*.mli Mylib_*.ml
```

4. Finally, create a library containing all the compilation units, and export all the compiled interfaces.

```ocaml
ocamlc -a Mylib*.cmo -o Mylib.cma
```

This approach lets you access `A` and `B` directly inside the library, and as `Mylib.A` and `Mylib.B` from outside. It also has the advantage that `Mylib` is no longer monolithic: if you use `Mylib.A`, only `Mylib_A` will be linked in, not `Mylib_B`.

### 7.14 Explicit overriding in class definitions

(Introduced in OCaml 3.12)

```
class-field ::= ...
  | inherit! class-expr [as lowercase-ident]
  | 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 7 (method override) and 13 (instance variable override). Note that warning 7 is disabled by default.

### 7.15 Overriding in open statements

(Introduced in OCaml 4.01)

```
definition ::= ...
  | open! module-path

specification ::= ...
  | open! module-path

eexpr ::= ...
  | let open! module-path in expr
```
Chapter 7. Language extensions

Since OCaml 4.01, open statements shadowing an existing identifier (which is later used) trigger the warning 44. Adding a ! character after the open keyword indicates that such a shadowing is intentional and should not trigger the warning.

7.16 Generalized algebraic datatypes

(Introduced in OCaml 4.00)

\[
\begin{align*}
\text{constr-decl} & ::= \ldots \\
 & | \text{constr-name} : [\text{constr-args} \rightarrow] \text{typexpr} \\
\text{type-param} & ::= \ldots \\
 & | [\text{variance}] \_ 
\end{align*}
\]

Generalized algebraic datatypes, or GADTs, extend usual sum types in two ways: constraints on type parameters may change depending on the value constructor, and some type variables may be existentially quantified. Adding constraints is done by giving an explicit return type (the rightmost typexpr in the above syntax), where type parameters are instantiated. This return type must use the same type constructor as the type being defined, and have the same number of parameters. Variables are made existential when they appear inside a constructor’s argument, but not in its return type.

Since the use of a return type often eliminates the need to name type parameters in the left-hand side of a type definition, one can replace them with anonymous types \_ in that case.

The constraints associated to each constructor can be recovered through pattern-matching. Namely, if the type of the scrutinee of a pattern-matching contains a locally abstract type, this type can be refined according to the constructor used. These extra constraints are only valid inside the corresponding branch of the pattern-matching. If a constructor has some existential variables, fresh locally abstract types are generated, and they must not escape the scope of this branch.

Recursive functions

Here is a concrete example:

```
let rec eval : type a. a term \rightarrow a = function
    | Int n \rightarrow n (* a = int *)
    | Add \rightarrow (fun x y \rightarrow x+y) (* a = int \rightarrow int \rightarrow int *)
    | App(f,x) \rightarrow (eval f) (eval x)
        (* eval called at types (b->a) and b for fresh b *)

let two = eval (App (App (Add, Int 1), Int 1))
val two : int = 2
```
It is important to remark that the function `eval` is using the polymorphic syntax for locally abstract types. When defining a recursive function that manipulates a GADT, explicit polymorphic recursion should generally be used. For instance, the following definition fails with a type error:

```ocaml
let rec eval (type a) : a term -> a = function
  | Int n     -> n
  | Add       -> (fun x y -> x+y)
  | App(f,x)  -> (eval f) (eval x)

(*
  Error: This expression has type (\App_\b \to a) term but an expression was expected of type 'a
  The type constructor \App_\b would escape its scope
*)
```

In absence of an explicit polymorphic annotation, a monomorphic type is inferred for the recursive function. If a recursive call occurs inside the function definition at a type that involves an existential GADT type variable, this variable flows to the type of the recursive function, and thus escapes its scope. In the above example, this happens in the branch `App(f,x)` when `eval` is called with `f` as an argument. In this branch, the type of `f` is `(\App_\b \to a)`. The prefix `$` in `\App_\b` denotes an existential type named by the compiler (see 7.16). Since the type of `eval` is `'a term -> 'a`, the call `eval f` makes the existential type `$\App_\b` flow to the type variable `'a` and escape its scope. This triggers the above error.

**Type inference**

Type inference for GADTs is notoriously hard. This is due to the fact some types may become ambiguous when escaping from a branch. For instance, in the `Int` case above, `n` could have either type `int` or `a`, and they are not equivalent outside of that branch. As a first approximation, type inference will always work if a pattern-matching is annotated with types containing no free type variables (both on the scrutinee and the return type). This is the case in the above example, thanks to the type annotation containing only locally abstract types.

In practice, type inference is a bit more clever than that: type annotations do not need to be immediately on the pattern-matching, and the types do not have to be always closed. As a result, it is usually enough to only annotate functions, as in the example above. Type annotations are propagated in two ways: for the scrutinee, they follow the flow of type inference, in a way similar to polymorphic methods; for the return type, they follow the structure of the program, they are split on functions, propagated to all branches of a pattern matching, and go through tuples, records, and sum types. Moreover, the notion of ambiguity used is stronger: a type is only seen as ambiguous if it was mixed with incompatible types (equated by constraints), without type annotations between them. For instance, the following program types correctly.

```ocaml
let rec sum : type a. a term -> _ = fun x ->
  let y =
    match x with
    | Int n -> n
    | Add   -> 0
    | App(f,x) -> sum f + sum x
```
in y + 1

val sum : 'a term -> int = <fun>

Here the return type int is never mixed with a, so it is seen as non-ambiguous, and can be inferred. When using such partial type annotations we strongly suggest specifying the -principal mode, to check that inference is principal.

The exhaustiveness check is aware of GADT constraints, and can automatically infer that some cases cannot happen. For instance, the following pattern matching is correctly seen as exhaustive (the Add case cannot happen).

let get_int : int term -> int = function
  | Int n   -> n
  | App(_,_) -> 0

Refutation cases and redundancy  (Introduced in OCaml 4.03)

Usually, the exhaustiveness check only tries to check whether the cases omitted from the pattern matching are typable or not. However, you can force it to try harder by adding refutation cases:

\[
\text{matching-case} ::= \text{pattern [when expr]} \rightarrow \text{expr} \\
\quad | \text{pattern} \rightarrow .
\]

In presence of a refutation case, the exhaustiveness check will first compute the intersection of the pattern with the complement of the cases preceding it. It then checks whether the resulting patterns can really match any concrete values by trying to type-check them. Wild cards in the generated patterns are handled in a special way: if their type is a variant type with only GADT constructors, then the pattern is split into the different constructors, in order to check whether any of them is possible (this splitting is not done for arguments of these constructors, to avoid non-termination.) We also split tuples and variant types with only one case, since they may contain GADTs inside.

For instance, the following code is deemed exhaustive:

```
type _ t =
  | Int : int t
  | Bool : bool t

let deep : (char t * int) option -> char = function
  | None       -> 'c'
  | _          -> .
```

Namely, the inferred remaining case is Some _, which is split into Some (Int, _) and Some (Bool, _), which are both untypable. Note that the refutation case could be omitted here, because it is automatically added when there is only one case in the pattern matching.

Another addition is that the redundancy check is now aware of GADTs: a case will be detected as redundant if it could be replaced by a refutation case using the same pattern.
Advanced examples  The term type we have defined above is an indexed type, where a type parameter reflects a property of the value contents. Another use of GADTs is singleton types, where a GADT value represents exactly one type. This value can be used as runtime representation for this type, and a function receiving it can have a polytypic behavior.

Here is an example of a polymorphic function that takes the runtime representation of some type \( t \) and a value of the same type, then pretty-prints the value as a string:

```ml
type _ typ =
| Int : int typ
| String : string typ
| Pair : 'a typ * 'b typ -> ('a * 'b) typ

let rec to_string : type t. t typ -> t -> string =
fun t x ->
match t with
| Int -> string_of_int x
| String -> Printf.sprintf "%S" x
| Pair(t1,t2) ->
  let (x1, x2) = x in
  Printf.sprintf "(%s,%s)" (to_string t1 x1) (to_string t2 x2)
```

Another frequent application of GADTs is equality witnesses.

```ml
type (_,_) eq = Eq : ('a,'a) eq

let cast : type a b. (a,b) eq -> a -> b = fun Eq x -> x
```

Here type eq has only one constructor, and by matching on it one adds a local constraint allowing the conversion between \( a \) and \( b \). By building such equality witnesses, one can make equal types which are syntactically different.

Here is an example using both singleton types and equality witnesses to implement dynamic types.

```ml
let rec eq_type : type a b. a typ -> b typ -> (a,b) eq option =
fun a b ->
match a, b with
| Int, Int -> Some Eq
| String, String -> Some Eq
| Pair(a1,a2), Pair(b1,b2) ->
  begin match eq_type_a1_b1, eq_type_a2_b2 with
   | Some Eq, Some Eq -> Some Eq
   | _ → None
  end
| _ → None

type dyn = Dyn : 'a typ * 'a -> dyn
```
let get_dyn : type a. a typ -> dyn -> a option =
  fun a (Dyn(b,x)) ->
  match eq_type a b with
  | None -> None
  | Some Eq -> Some x

Existential type names in error messages  (Updated in OCaml 4.03.0)

The typing of pattern matching in presence of GADT can generate many existential types. When necessary, error messages refer to these existential types using compiler-generated names. Currently, the compiler generates these names according to the following nomenclature:

- First, types whose name starts with a $ are existentials.
- $Constr_.'a denotes an existential type introduced for the type variable 'a of the GADT constructor Constr:

```ocaml
# type any = Any : 'name -> any
# let escape (Any x) = x;;
Error: This expression has type $Any_.'name
but an expression was expected of type 'a
The type constructor $Any_.'name would escape its scope
```

- $Constr denotes an existential type introduced for an anonymous type variable in the GADT constructor Constr:

```ocaml
# type any = Any : _ -> any
# let escape (Any x) = x;;
Error: This expression has type $Any but an expression was expected of type 
'a
The type constructor $Any would escape its scope
```

- $'a if the existential variable was unified with the type variable 'a during typing:

```ocaml
# type ('arg,'result,'aux) fn =
# | Fun: ('a ->'b) -> ('a,'b,unit) fn
# | Mem1: ('a ->'b) * 'a * 'b -> ('a, 'b, 'a * 'b) fn
# let apply: ('arg,'result, _) fn -> 'arg -> 'result = fun f x ->
# match f with
# | Fun f -> f x
# | Mem1 (f,y,fy) -> if x = y then fy else f x;;
Error: This pattern matches values of type 
($'arg, $'result, $'arg * $'result) fn
but a pattern was expected which matches values of type 
($'arg, $'result, unit) fn
Type $'arg * $'result is not compatible with type unit
```

- $n (n a number) is an internally generated existential which could not be named using one of the previous schemes.

As shown by the last item, the current behavior is imperfect and may be improved in future versions.
Equations on non-local abstract types  (Introduced in OCaml 4.04)

GADT pattern-matching may also add type equations to non-local abstract types. The behaviour is the same as with local abstract types. Reusing the above eq type, one can write:

```ocaml
module M : sig type t val x : t val e : (x,int) eq end = struct
  type t = int
  let x = 33
  let e = Eq
end

let x : int = let Eq = M.e in M.x
```

Of course, not all abstract types can be refined, as this would contradict the exhaustiveness check. Namely, builtin types (those defined by the compiler itself, such as int or array), and abstract types defined by the local module, are non-instantiable, and as such cause a type error rather than introduce an equation.

7.17 Syntax for Bigarray access

(Introduced in Objective Caml 3.00)

```
expr ::= ... 
  | expr .{ expr { , expr} }
  | expr .{ expr { , expr} } <- expr
```

This extension provides syntactic sugar for getting and setting elements in the arrays provided by the Bigarray library.

The short expressions are translated into calls to functions of the Bigarray module as described in the following table.

<table>
<thead>
<tr>
<th>expression</th>
<th>translation</th>
</tr>
</thead>
<tbody>
<tr>
<td>expr₀ .{ expr₁ }</td>
<td>Bigarray.Array1.get expr₀ expr₁</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ } &lt;- expr</td>
<td>Bigarray.Array1.set expr₀ expr₁ expr</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , expr₂ }</td>
<td>Bigarray.Array2.get expr₀ expr₁ expr₂</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , expr₂ } &lt;- expr</td>
<td>Bigarray.Array2.set expr₀ expr₁ expr₂ expr</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , expr₂ , expr₃ }</td>
<td>Bigarray.Array3.get expr₀ expr₁ expr₂ expr₃</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , expr₂ , expr₃ } &lt;- expr</td>
<td>Bigarray.Array3.set expr₀ expr₁ expr₂ expr₃ expr</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , ..., exprₙ }</td>
<td>Bigarray.Genarray.get expr₀ [ expr₁ , ..., exprₙ ]</td>
</tr>
<tr>
<td>expr₀ .{ expr₁ , ..., exprₙ } &lt;- expr</td>
<td>Bigarray.Genarray.set expr₀ [ expr₁ , ..., exprₙ ] expr</td>
</tr>
</tbody>
</table>

The last two entries are valid for any \( n > 3 \).

7.18 Attributes

(Introduced in OCaml 4.02, infix notations for constructs other than expressions added in 4.03)
Attributes are “decorations” of the syntax tree which are mostly ignored by the type-checker but can be used by external tools. An attribute is made of an identifier and a payload, which can be a structure, a type expression (prefixed with `:`), a signature (prefixed with `:`) or a pattern (prefixed with `?`) optionally followed by a `when` clause:

\[
\begin{align*}
\text{attr-id} & \ ::= \ \text{lowercase-ident} \\
& \quad | \ \text{capitalized-ident} \\
& \quad | \ attr-id . attr-id \\
\text{attr-payload} & \ ::= \ [\text{module-items}] \\
& \quad | \ : \ \text{typexpr} \\
& \quad | \ : [\text{specification}] \\
& \quad | \ ? \ \text{pattern} [\text{when expr}]
\end{align*}
\]

The first form of attributes is attached with a postfix notation on “algebraic” categories:

\[
\begin{align*}
\text{attribute} & \ ::= \ [@ \text{attr-id attr-payload }] \\
\text{expr} & \ ::= \ ... \ \\
& \quad | \ \text{expr attribute} \\
\text{typexpr} & \ ::= \ ... \ \\
& \quad | \ \text{typexpr attribute} \\
\text{pattern} & \ ::= \ ... \\
& \quad | \ \text{pattern attribute} \\
\text{module-expr} & \ ::= \ ... \ \\
& \quad | \ \text{module-expr attribute} \\
\text{module-type} & \ ::= \ ... \ \\
& \quad | \ \text{module-type attribute} \\
\text{class-expr} & \ ::= \ ... \ \\
& \quad | \ \text{class-expr attribute} \\
\text{class-type} & \ ::= \ ... \ \\
& \quad | \ \text{class-type attribute}
\end{align*}
\]

This form of attributes can also be inserted after the `tag-name` in polymorphic variant type expressions (`tag-spec-first`, `tag-spec`, `tag-spec-full`) or after the `method-name` in `method-type`. The same syntactic form is also used to attach attributes to labels and constructors in type declarations:

\[
\begin{align*}
\text{field-decl} & \ ::= \ [\text{mutable}] \ \text{field-name} : \ \text{poly-typexpr} \ \{\text{attribute}\} \\
\text{constr-decl} & \ ::= \ (\text{constr-name} | () [\text{of constr-args}] \ \{\text{attribute}\}
\end{align*}
\]
Note: when a label declaration is followed by a semi-colon, attributes can also be put after the semi-colon (in which case they are merged to those specified before).

The second form of attributes are attached to “blocks” such as type declarations, class fields, etc:
item-attribute ::= [@@ attr-id attr-payload ]
typedef ::= ...
   | typedef item-attribute
exception-definition ::= exception constr-decl
   | exception constr-name = constr
module-items ::= [;;] (definition | expr {item-attribute}) {[;;] definition | ; ; expr {item-attribute}} [;;]
class-binding ::= ...
   | class-binding item-attribute
class-spec ::= ...
   | class-spec item-attribute
class-type-def ::= ...
   | classtype-def item-attribute
definition ::= let [rec] let-binding {and let-binding}
   | external value-name : typexpr = external-declaration {item-attribute}
   | type-definition
   | exception-definition {item-attribute}
   | class-definition
   | classtype-definition
   | module module-name { ( module-name : module-type )} [: module-type]
   = module-expr {item-attribute}
   | module type modtype-name = module-type {item-attribute}
   | open module-path {item-attribute}
   | include module-expr {item-attribute}
   | module rec module-name : module-type = module-expr
   | {and module-name : module-type = module-expr
   | {item-attribute}}

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

class-field-spec ::= ...
   | class-field-spec item-attribute
class-field ::= ...
   | class-field item-attribute
A third form of attributes appears as stand-alone structure or signature items in the module or class sub-languages. They are not attached to any specific node in the syntax tree:

\[
\text{floating-attribute} ::= \begin{array}{l}
\text{@attr-id attr-payload}
\end{array}
\]

\[
\text{definition} ::= ...
| \text{floating-attribute}
\]

\[
\text{specification} ::= ...
| \text{floating-attribute}
\]

\[
\text{class-field-spec} ::= ...
| \text{floating-attribute}
\]

\[
\text{class-field} ::= ...
| \text{floating-attribute}
\]

(Note: contrary to what the grammar above describes, \text{item-attributes} cannot be attached to these floating attributes in \text{class-field-spec} and \text{class-field}.)

It is also possible to specify attributes using an infix syntax. For instance:

\[
\begin{align*}
\text{let[@foo] x = 2 in x + 1} & \quad \text{=== (let x = 2 [@foo] in x + 1)} \\
\text{begin[@foo][@bar x] ... end} & \quad \text{=== (begin ... end)[@foo][@bar x]} \\
\text{module[@foo] M = ...} & \quad \text{=== module M = ... [@foo]} \\
\text{type[@foo] t = T} & \quad \text{=== type t = T [@foo]} \\
\text{method[@foo] m = ...} & \quad \text{=== method m = ... [@foo]}
\end{align*}
\]

For \text{let}, the attributes are applied to each bindings:

\[
\begin{align*}
\text{let[@foo] x = 2 and y = 3 in x + y} & \quad \text{=== (let x = 2 [@foo] and y = 3 in x + y)} \\
\text{let[@foo] x = 2} & \quad \text{=== (let x = 2 [@foo] and y = 3 [@bar] in x + y)}
\end{align*}
\]

7.18.1 Built-in attributes

Some attributes are understood by the type-checker:

- “ocaml.warning” or “warning”, with a string literal payload. This can be used as floating attributes in a signature/structure/object/object type. The string is parsed and has the same effect as the \text{-w} command-line option, in the scope between the attribute and the end of the current signature/structure/object/object type. The attribute can also be used on an expression, in which case its scope is limited to that expression. Note that it is not well-defined which scope is used for a specific warning. This is implementation dependant and can change between versions. For instance, warnings triggered by the “ppwarning” attribute (see below) are issued using the global warning configuration.

- “ocaml.warnerror” or “warnerror”, with a string literal payload. Same as “ocaml.warning”, for the \text{-warn-error} command-line option.
Chapter 7. Language extensions

- “ocaml.deprecated” or “deprecated”. Can be applied to most kind of items in signatures or structures. When the element is later referenced, a warning (3) is triggered. If the payload of the attribute is a string literal, the warning message includes this text. It is also possible to use this “ocaml.deprecated” as a floating attribute on top of an “.mli” file (i.e. before any other non-attribute item) or on top of an “.ml” file without a corresponding interface; this marks the unit itself as being deprecated.

- “ocaml.deprecated_mutable” or “deprecated_mutable”. Can be applied to a mutable record label. If the label is later used to modify the field (with “expr.1 |- expr”), a warning (3) will be triggered. If the payload of the attribute is a string literal, the warning message includes this text.

- “ocaml.ppwarning” or “ppwarning”, in any context, with a string literal payload. The text is reported as warning (22) by the compiler (currently, the warning location is the location of the string payload). This is mostly useful for preprocessors which need to communicate warnings to the user. This could also be used to mark explicitly some code location for further inspection.

- “ocaml.warn_on_literal_pattern” or “warn_on_literal_pattern” annotate constructors in type definition. A warning (52) is then emitted when this constructor is pattern matched with a constant literal as argument. This attribute denotes constructors whose argument is purely informative and may change in the future. Therefore, pattern matching on this argument with a constant literal is unreliable. For instance, all built-in exception constructors are marked as “warn_on_literal_pattern”. Note that, due to an implementation limitation, this warning (52) is only triggered for single argument constructor.

- “ocaml.tailcall” or “tailcall” can be applied to function application in order to check that the call is tailcall optimized. If it is not the case, a warning (51) is emitted.

- “ocaml.inline” or “inline” take either “never”, “always” or nothing as payload on a function or functor definition. If no payload is provided, the default value is “always”. This payload controls when applications of the annotated functions should be inlined.

- “ocaml.inlined” or “inlined” can be applied to any function or functor application to check that the call is inlined by the compiler. If the call is not inlined, a warning (55) is emitted.

- “ocaml.noalloc”, “ocaml.unboxed” and “ocaml.untagged” or “noalloc”, “unboxed” and “untagged” can be used on external definitions to obtain finer control over the C-to-OCaml interface. See [18.10] for more details.

- “ocaml.immediate” or “immediate” applied on an abstract type mark the type as having a non-pointer implementation (e.g. “int”, “bool”, “char” or enumerated types). Mutation of these immediate types does not activate the garbage collector’s write barrier, which can significantly boost performance in programs relying heavily on mutable state.

- “ocaml.unboxed” or “unboxed” can be used on a type definition if the type is a single-field record or a concrete type with a single constructor that has a single argument. It tells the compiler to optimize the representation of the type by removing the block that represents the record
or the constructor (i.e. a value of this type is physically equal to its argument). In the case of GADTs, an additional restriction applies: the argument must not be an existential variable, represented by an existential type variable, or an abstract type constructor applied to an existential type variable.

- `ocaml.boxed` or `boxed` can be used on type definitions to mean the opposite of `ocaml.unboxed`: keep the unoptimized representation of the type. When there is no annotation, the default is currently `boxed` but it may change in the future.

```ocaml
module X = struct
  [@warning "+9"] (* locally enable warning 9 in this structure *)
  ...
end
[@deprecated "Please use module 'Y' instead."]

let x = begin[@warning "+9"] ... end in ....

type t = A | B
[@deprecated "Please use type 's' instead."]

let f x =
  assert (x >= 0) [@ppwarning "TODO: remove this later"];

let rec no_op = function
  | [] -> ()
  | _ :: q -> (no_op[@tailcall]) q;;

let f x = x [@inline]
let () = (f[@inlined]) ()

type fragile =
  | Int of int [@warn_on_literal_pattern]
  | String of string [@warn_on_literal_pattern]

let f = function
  | Int 0 | String "constant" -> () (* trigger warning 52 *)
  | _ -> ()

module Immediate: sig
  type t [@immediate]
  val x: t ref
end = struct
  type t = A | B
  let x = ref 0
end
```
7.19 Extension nodes

(Introduced in OCaml 4.02, infix notations for constructs other than expressions added in 4.03)

Extension nodes are generic placeholders in the syntax tree. They are rejected by the type-checker and are intended to be “expanded” by external tools such as `ppx` rewriters.

Extension nodes share the same notion of identifier and payload as attributes 7.18.

The first form of extension node is used for “algebraic” categories:

\[
\begin{align*}
\text{extension } &::= \[ \%\ attr-id\ attr-payload \] \\
\text{expr } &::= \ldots \\
&\quad | \text{extension} \\
\text{typexpr } &::= \ldots \\
&\quad | \text{extension} \\
\text{pattern } &::= \ldots \\
&\quad | \text{extension} \\
\text{module-expr } &::= \ldots \\
&\quad | \text{extension} \\
\text{module-type } &::= \ldots \\
&\quad | \text{extension} \\
\text{class-expr } &::= \ldots \\
&\quad | \text{extension} \\
\text{class-type } &::= \ldots \\
&\quad | \text{extension}
\end{align*}
\]

A second form of extension node can be used in structures and signatures, both in the module and object languages: 

item-extension ::= [% attr-id attr-payload ]
definition ::= ...
    | item-extension
specification ::= ...
    | item-extension
class-field-spec ::= ...
    | item-extension
class-field ::= ...
    | item-extension

An infix form is available for extension nodes when the payload is of the same kind (expression with expression, pattern with pattern ...).
Examples:

let%foo x = 2 in x + 1 === [%foo let x = 2 in x + 1]
beg%foo ... end === [%foo begin ... end]
module%foo M = .. === [%foo module M = ... ]
val%foo x : t === [%foo: val x : t]

When this form is used together with the infix syntax for attributes, the attributes are considered to apply to the payload:

fun%foo[@bar] x -> x + 1 === [%foo (fun x -> x + 1)[@foo ] ];

7.19.1 Built-in extension nodes
(Introduced in OCaml 4.03)
Some extension nodes are understood by the compiler itself:

• “ocaml.extension_constructor” or “extension_constructor” take as payload a constructor from an extensible variant type (see 7.22) and return its extension constructor slot.

type t = ..
type t += X of int | Y of string
let x = [%extension_constructor X]
let y = [%extension_constructor Y]

# x <> y;;
- : bool = true
7.20 Quoted strings

(Introduced in OCaml 4.02)

Quoted strings provide a different lexical syntax to write string literals in OCaml code. This can be used to embed pieces of foreign syntax fragments in OCaml code, to be interpreted by a -ppx filter or just a library.

```
string-literal ::= ...
               | { quoted-string-id | ....... | quoted-string-id }
```

```
quoted-string-id ::= {a...z | _}
```

The opening delimiter has the form \{id\} where id is a (possibly empty) sequence of lowercase letters and underscores. The corresponding closing delimiter is \{|id\} (with the same identifier). Unlike regular OCaml string literals, quoted strings do not interpret any character in a special way.

Example:

```
String.length {"|} (* returns 2 *)
String.length {foo"|foo} (* returns 2 *)
```

7.21 Exception cases in pattern matching

(Introduced in OCaml 4.02)

A new form of exception patterns is allowed, only as a toplevel pattern under a `match...with` pattern-matching (other occurrences are rejected by the type-checker).

```
pattern ::= ...
          | exception pattern
```

Cases with such a toplevel pattern are called “exception cases”, as opposed to regular “value cases”. Exception cases are applied when the evaluation of the matched expression raises an exception. The exception value is then matched against all the exception cases and re-raised if none of them accept the exception (as for a `try...with` block). Since the bodies of all exception and value cases is outside the scope of the exception handler, they are all considered to be in tail-position: if the `match...with` block itself is in tail position in the current function, any function call in tail position in one of the case bodies results in an actual tail call.

It is an error if all cases are exception cases in a given pattern matching.

7.22 Extensible variant types

(Introduced in OCaml 4.02)
Extensible variant types are variant types which can be extended with new variant constructors. Extensible variant types are defined using \ldots\ New variant constructors are added using $\mathbb{+}=\$.  

\begin{verbatim}
type attr = ..

type attr $\mathbb{+}=\$ Str of string

| Int of int
| Float of float
\end{verbatim}

Pattern matching on an extensible variant type requires a default case to handle unknown variant constructors:

\begin{verbatim}
let to_string = function
    | Str s -> s
    | Int i -> string_of_int i
    | Float f -> string_of_float f
    | _ -> "?"
\end{verbatim}

A preexisting example of an extensible variant type is the built-in \texttt{exn} type used for exceptions. Indeed, exception constructors can be declared using the type extension syntax:

\begin{verbatim}
type exn $\mathbb{+}=\$ Exc of int
\end{verbatim}

Extensible variant constructors can be rebound to a different name. This allows exporting variants from another module.

\begin{verbatim}
type Expr.attr $\mathbb{+}=\$ Str = Expr.Str
\end{verbatim}

Extensible variant constructors can be declared \texttt{private}. As with regular variants, this prevents them from being constructed directly by constructor application while still allowing them to be destructured in pattern-matching.
Chapter 7. Language extensions

7.23 Generative functors

(Introduced in OCaml 4.02)

A generative functor takes a unit () argument. In order to use it, one must necessarily apply it to this unit argument, ensuring that all type components in the result of the functor behave in a generative way, i.e. they are different from types obtained by other applications of the same functor. This is equivalent to taking an argument of signature sig end, and always applying to struct end, but not to some defined module (in the latter case, applying twice to the same module would return identical types).

As a side-effect of this generativity, one is allowed to unpack first-class modules in the body of generative functors.

7.24 Extension-only syntax

(Introduced in OCaml 4.02.2, extended in 4.03)

Some syntactic constructions are accepted during parsing and rejected during type checking. These syntactic constructions can therefore not be used directly in vanilla OCaml. However, -ppx rewriters and other external tools can exploit this parser leniency to extend the language with these new syntactic constructions by rewriting them to vanilla constructions.

7.24.1 Extension operators

(Introduced in OCaml 4.02.2)

Operator names starting with a # character and containing more than one # character are reserved for extensions.
7.24.2 Extension literals

(Introduced in OCaml 4.03)

```
float-literal ::= ...
    | [-] (0...9) {0...9 | _} [. {0...9 | _}] [(e | E) [+ | -] (0...9) {0...9 | _}] [g...z | G...Z]
    | [-] (0x | 0X) (0...9 | A...F | a...f) {0...9 | A...F | a...f | _}
    | [. {0...9 | A...F | a...f | _}] [(p | P) [+ | -] (0...9) {0...9 | _}] [g...z | G...Z]
```

```
int-literal ::= ...
    | [-] (0...9) {0...9 | _} [g...z | G...Z]
    | [-] (0x | 0X) (0...9 | A...F | a...f) {0...9 | A...F | a...f | _} [g...z | G...Z]
    | [-] (0o | 0O) (0...7) {0...7 | _} [g...z | G...Z]
    | [-] (0b | 0B) (0...1) {0...1 | _} [g...z | G...Z]
```

Int and float literals followed by an one-letter identifier in the range [g.. z | G.. Z] are extension-only literals.

7.25 Inline records

(Introduced in OCaml 4.03)

```
constr-args ::= ...
    | record-decl
```

The arguments of a sum-type constructors can now be defined using the same syntax as records. Mutable and polymorphic fields are allowed. GADT syntax is supported. Attributes can be specified on individual fields.

Syntactically, building or matching constructors with such an inline record argument is similar to working with a unary constructor whose unique argument is a declared record type. A pattern can bind the inline record as a pseudo-value, but the record cannot escape the scope of the binding and can only be used with the dot-notation to extract or modify fields or to build new constructor values.

```
type t =
    | Point of {width: int; mutable x: float; mutable y: float}
    | ...

let v = Point {width = 10; x = 0.; y = 0.}

let scale l = function
    | Point p -> Point {p with x = l * p.x; y = l * p.y}
    | ....

let print = function
```
| Point {x; y; _} -> Printf.printf "%.f/%.f" x y
| ....

let reset = function
| Point p -> p.x <- 0.; p.y <- 0.
| ...

let invalid = function
| Point p -> p (* INVALID *)
| ...

### 7.26 Local exceptions

(Introduced in OCaml 4.04)

It is possible to define local exceptions in expressions:

\[
expr ::= ... \\
\quad \mid \text{let exception constr-decl in expr}
\]

The syntactic scope of the exception constructor is the inner expression, but nothing prevents exception values created with this constructor from escaping this scope. Two executions of the definition above result in two incompatible exception constructors (as for any exception definition).

### 7.27 Documentation comments

(Introduced in OCaml 4.03)

Comments which start with ** are treated specially by the compiler. They are automatically converted during parsing into attributes (see 7.18) to allow tools to process them as documentation.

Such comments can take three forms: floating comments, item comments and label comments. Any comment starting with ** which does not match one of these forms will cause the compiler to emit warning 50.

Comments which start with ** are also used by the ocamldoc documentation generator (see 14). The three comment forms recognised by the compiler are a subset of the forms accepted by ocamldoc (see 14.2).

#### 7.27.1 Floating comments

Comments surrounded by blank lines that appear within structures, signatures, classes or class types are converted into floating-attributes. For example:

```
type t = T

(** Now some definitions for [t] *)
```

```
let mkT = T
```
will be converted to:

```ocaml
type t = T
[@@ocaml.text " Now some definitions for [t] "]
let mkT = T
```

### 7.27.2 Item comments

Comments which appear *immediately before* or *immediately after* a structure item, signature item, class item or class type item are converted into *item-attributes*. Immediately before or immediately after means that there must be no blank lines, ; ; , or other documentation comments between them. For example:

```ocaml
type t = T
(** A description of [t] *)
```

or

```ocaml
(** A description of [t] *)
type t = T
```

will be converted to:

```ocaml
type t = T
[@@ocaml.doc " A description of [t] "]
```

Note that, if a comment appears immediately next to multiple items, as in:

```ocaml
type t = T
(** An ambiguous comment *)
type s = S
```

then it will be attached to both items:

```ocaml
type t = T
[@@ocaml.doc " An ambiguous comment "]
type s = S
[@@ocaml.doc " An ambiguous comment "]
```

and the compiler will emit warning 50.
7.27.3 Label comments

Comments which appear *immediately after* a labelled argument, record field, variant constructor, object method or polymorphic variant constructor are converted into attributes. Immediately after means that there must be no blank lines or other documentation comments between them. For example:

```ocaml
type t1 = lbl:int (** Labelled argument *) -> unit

type t2 = {
    fld: int; (** Record field *)
    fld2: float;
}

type t3 =
| Cstr of string (** Variant constructor *)
| Cstr2 of string

type t4 = < meth: int * int; (** Object method *) >

type t5 = [
    `PCstr (** Polymorphic variant constructor *)
]
```

will be converted to:

```ocaml
type t1 = lbl:(int [@ocaml.doc " Labelled argument "]) -> unit

type t2 = {
    fld: int [@ocaml.doc " Record field "];
    fld2: float;
}

type t3 =
| Cstr of string [@ocaml.doc " Variant constructor "]
| Cstr2 of string

type t4 = < meth : int * int [@ocaml.doc " Object method "] >

type t5 = [
    `PCstr [@ocaml.doc " Polymorphic variant constructor "]
]
```

Note that label comments take precedence over item comments, so:

```ocaml
type t = T of string
(** Attaches to T not t *)
```
will be converted to:

type t = T of string [@ocaml.doc " Attaches to T not t "]

whilst:

type t = T of string
(** Attaches to T not t *)
(** Attaches to t *)

will be converted to:

type t = T of string [@ocaml.doc " Attaches to T not t "]
[@@ocaml.doc " Attaches to t "]

In the absence of meaningful comment on the last constructor of a type, an empty comment (***) can be used instead:


type t = T of string
(***)
(** Attaches to t *)

will be converted directly to


type t = T of string
[@@ocaml.doc " Attaches to t "]
Part III

The OCaml tools
Chapter 8

Batch compilation (ocamlc)

This chapter describes the OCaml batch compiler ocamlc, which compiles OCaml 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 ocamlrun.

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, after all options have been processed:

- 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 OCaml code, and their names are written in the generated bytecode executable. The run-time system `ocamlrun` 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 `ocamlrun`. If `a.out` is the name of the file produced by the linking phase, the command

```
ocamlrun a.out arg_1 arg_2 ... arg_n
```

executes the compiled code contained in `a.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:

```
./a.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.
-absname
   Force error messages to show absolute paths for file names.

-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.

-bin-annot
   Dump detailed information about the compilation (types, bindings, tail-calls, etc) in binary format. The information for file src.ml is put into file src.cmt. In case of a type error, dump all the information inferred by the type-checker before the error. The *.cmt files produced by -bin-annot contain more information and are much more compact than the files produced by -annot.

-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.)

-color mode
   Enable or disable colors in compiler messages (especially warnings and errors). The following modes are supported:
      
auto
   use heuristics to enable colors only if the output supports them (an ANSI-compatible tty terminal);

always
   enable colors unconditionally;

never
   disable color output.

The default setting is ’auto’, and the current heuristic checks that the TERM environment variable exists and is not empty or dumb, and that isatty(stderr) holds.
-compat-32
Check that the generated bytecode executable can run on 32-bit platforms and signal an error if it cannot. This is useful when compiling bytecode on a 64-bit machine.

-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 OCaml 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 dllibname.so (dllibname.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].

-for-pack module-path
Generate an object file (.cmo) that can later be included as a sub-module (with the given path) of a compilation unit constructed with -pack. For instance, ocamlc -for-pack P -c A.ml will generate a.cmo that can later be used with ocamlc -pack -o P.cmo a.cmo. Note: you can still pack a module that was compiled without -for-pack but in this case exceptions will be printed with the wrong names.

-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 [15]), and 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.
Chapter 8. Batch compilation (ocamlc)

-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. See also option -nostdlib.

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.

-no-alias-deps
Do not record dependencies for module aliases. See section 7.13 for more information.

-no-app-funct
Deactivates the applicative behaviour of functors. With this option, each functor application generates new types in its result and applying the same functor twice to the same argument yields two incompatible structures.

-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.

-nostdlib
Do not include the standard library directory in the list of directories searched for compiled interface files (.cki), compiled object code files (.cmo), libraries (.cma), and C libraries specified with -cclib -lxxx. See also option -I.

-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.

-open Module
Opens the given module before processing the interface or implementation files. If several -open options are given, they are processed in order, just as if the statements open! Module1;; ... open! ModuleN;; were added at the top of each file.

-output-obj
Cause the linker to produce a C object file instead of a bytecode executable file. This is useful to wrap OCaml code as a C library, callable from any C program. See chapter 18 section 18.7.5 The name of the output object file must 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 (.cki) that combines the object files given on the command line, making them appear as sub-modules of the output .cma 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.cki 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.
Chapter 8. Batch compilation (ocamlc)

- **plugin plugin**
  Dynamically load the code of the given plugin (a .cmo, .cma or .cmxs file) in the compiler. *plugin* must exist in the same kind of code as the compiler (ocamlc.byte must load bytecode plugins, while ocamlc.opt must load native code plugins), and extension adaptation is done automatically for .cma files (to .cmxs files if the compiler is compiled in native code).

- **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.

- **ppx command**
  After parsing, pipe the abstract syntax tree through the preprocessor *command*. The module Ast_mapper, described in chapter [22.1](#), implements the external interface of a preprocessor.

- **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.

- **safe-string**
  Enforce the separation between types *string* and *bytes*, thereby making strings read-only. This will become the default in a future version of OCaml.

- **short-paths**
  When a type is visible under several module-paths, use the shortest one when printing the type’s name in inferred interfaces and error and warning messages. Identifier names starting with an underscore _ or containing double underscores __ incur a penalty of +10 when computing their length.

- **strict-sequence**
  Force the left-hand part of each sequence to have type unit.

- **strict-formats**
  Reject invalid formats that were accepted in legacy format implementations. You should use
this flag to detect and fix such invalid formats, as they will be rejected by future OCaml versions.

-thread
Compile or link multithreaded programs, in combination with the system threads library
described in chapter \[18\].

-unboxed-types
When a type is unboxable (i.e. a record with a single argument or a concrete datatype
with a single constructor of one argument) it will be unboxed unless annotated with
[@@ocaml.boxed].

-no-unboxed-types
When a type is unboxable it will be boxed unless annotated with [@@ocaml.unboxed]. This
is the default.

-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.

-unsafe-string
Identify the types string and bytes, thereby making strings writable. For reasons of back-
ward compatibility, this is the default setting for the moment, but this will change in a future
version of OCaml.

-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.

-vmthread
Compile or link multithreaded programs, in combination with the VM-level threads library
described in chapter \[18\].

-version or -vnum
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 fatal the warnings specified by the argument warning-list. Each
warning can be enabled or disabled, and each warning can be fatal or non-fatal. If a warning
is disabled, it isn’t displayed and doesn’t affect compilation in any way (even if it is fatal).
If a warning is enabled, it is displayed normally by the compiler whenever the source code triggers it. If it is enabled and fatal, the compiler will also stop with an error after displaying 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 as fatal warning number num.
+num1..num2
    Enable warnings in the given range.
-num1..num2
    Disable warnings in the given range.
@num1..num2
    Enable and mark as fatal 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 as fatal 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 and letters which are out of the range of warnings that are currently defined are ignored. The warnings are as follows.

1 Suspicious-looking start-of-comment mark.
2 Suspicious-looking end-of-comment mark.
3 Deprecated feature.
4 Fragile pattern matching: matching that will remain complete even if additional constructors are added to one of the variant types matched.
5 Partially applied function: expression whose result has function type and is ignored.
Label omitted in function application.
Method overridden.
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.
Instance variable overridden.
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.
Preprocessor warning.
Useless record with clause.
Bad module name: the source file name is not a valid OCaml module name.
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.
A module is linked twice in the same executable.
Unused value declaration.
Unused open statement.
Unused type declaration.
Unused for-loop index.
Unused ancestor variable.
Unused constructor.
Unused extension constructor.
Unused rec flag.
Chapter 8. Batch compilation (ocamlc)

40 Constructor or label name used out of scope.
41 Ambiguous constructor or label name.
42 Disambiguated constructor or label name (compatibility warning).
43 Nonoptional label applied as optional.
44 Open statement shadows an already defined identifier.
45 Open statement shadows an already defined label or constructor.
46 Error in environment variable.
47 Illegal attribute payload.
48 Implicit elimination of optional arguments.
49 Absent cmi file when looking up module alias.
50 Unexpected documentation comment.
51 Warning on non-tail calls if @tailcall present.
52 (see 8.5.1)
   Fragile constant pattern.
53 Attribute cannot appear in this context
54 Attribute used more than once on an expression
55 Inlining impossible
56 Unreachable case in a pattern-matching (based on type information).
57 (see 8.5.2)
   Ambiguous or-pattern variables under guard
58 Missing cmx file
59 Assignment to non-mutable value
60 Unused module declaration
A all warnings
C warnings 1, 2.
D Alias for warning 3.
E Alias for warning 4.
F Alias for warning 5.
K warnings 32, 33, 34, 35, 36, 37, 38, 39.
L Alias for warning 6.
M Alias for warning 7.
P Alias for warning 8.
R Alias for warning 9.
S Alias for warning 10.
U warnings 11, 12.
V Alias for warning 13.
warnings 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 30.

Alias for warning 26.

Alias for warning 27.

Some warnings are described in more detail in section 8.5.

The default setting is `\texttt{-w +a-4-6-7-9-27-29-32..39-41..42-44-45-48-50}`. It is displayed by `\texttt{ocamlc -help}`. Note that warnings 5 and 10 are not always triggered, depending on the internals of the type checker.

\texttt{--warn-error} \texttt{\textit{warning-list}}

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

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

The default setting is \texttt{--warn-error \texttt{-a+31}} (only warning 31 is fatal).

\texttt{--warn-help}

Show the description of all available warning numbers.

\texttt{--where}

Print the location of the standard library, then exit.

\texttt{--file}

Process \texttt{file} as a file name, even if it starts with a dash (-) character.

\texttt{--help} or \texttt{----help}

Display a short usage summary and exit.

\textbf{Contextual control of command-line options}

The compiler command line can be modified “from the outside” with the following mechanisms. These are experimental and subject to change. They should be used only for experimental and development work, not in released packages.

\texttt{OCAMLPARAM} (environment variable)

Arguments that will be inserted before or after the arguments from the command line.

\texttt{ocaml_compiler_internal_params} (file in the stdlib directory)

A mapping of file names to lists of arguments that will be added to the command line (and \texttt{OCAMLPARAM}) arguments.

\texttt{OCAML_FLEXLINK} (environment variable)

Alternative executable to use on native Windows for \texttt{flexlink} instead of the configured value. Primarily used for bootstrapping.
Chapter 8. Batch compilation (ocamlc)

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.*; 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 foo cannot be used with type foo”, and it really seems that the two types foo are compatible. This is not always true. Two type constructors can have the same name, but actually represent different types. This can happen if a type constructor is redefined. Example:

```
module type foo = A | B
let f = function A -> 0 | B -> 1
module type foo = C | D
f C
```

This result in the error message “expression C of type foo cannot be used with type foo”.

The type of this expression, \( t \), contains type variables that cannot be generalized

Type variables (‘a, ‘b, ...) in a type \( t \) can be in either of two states: generalized (which means that the type \( t \) is valid for all possible instantiations of the variables) and not generalized (which means that the type \( t \) is valid only for one instantiation of the variables). In a \textbf{let} binding \textbf{let} \textit{name} = \textit{expr}, the type-checker normally generalizes as many type variables as possible in the type of \textit{expr}. However, this leads to unsoundness (a well-typed program can crash) in conjunction with polymorphic mutable data structures. To avoid this, generalization is performed at \textbf{let} bindings only if the bound expression \textit{expr} belongs to the class of “syntactic values”, which includes constants, identifiers, functions, tuples of syntactic values, etc. In all other cases (for instance, \textit{expr} is a function application), a polymorphic mutable could have been created and generalization is therefore turned off for all variables occurring 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 \textit{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 \textit{name}. For instance, instead of writing
let sort_int_list = Sort.list (<)
(* inferred type 'a list -> 'a list, with 'a not generalized *)

write

let sort_int_list = (Sort.list (<) : int list -> int list);;

• If you really need name to have a polymorphic type, turn its defining expression into a function by adding an extra parameter. For instance, instead of writing

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

write

let map_length lv = List.map Array.length lv

Reference to undefined global mod
This error appears when trying to link an incomplete or incorrectly ordered set of files. Either you have forgotten to provide an implementation for the compilation unit named mod on the command line (typically, the file named mod.cmo, or a library containing that file). Fix: add the missing .ml or .cmo file to the command line. Or, you have provided an implementation for the module named mod, but it comes too late on the command line: the implementation of mod must come before all bytecode object files that reference mod. Fix: change the order of .ml and .cmo files on the command line.

Of course, you will always encounter this error if you have mutually recursive functions across modules. That is, function Mod1.f calls function Mod2.g, and function Mod2.g calls function Mod1.f. In this case, no matter what permutations you perform on the command line, the program will be rejected at link-time. Fixes:

• Put f and g in the same module.
• Parameterize one function by the other. That is, instead of having

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

define

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

and link mod1.cmo before mod2.cmo.
• Use a reference to hold one of the two functions, as in :

mod1.ml: let forward_g =
    ref((fun x -> failwith "forward_g") : <type>)
    let f x = ... !forward_g ...
mod2.ml: let g y = ... Mod1.f ...
    let _ = Mod1.forward_g := g
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 \( f \) C 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 \(-\text{custom}\) option.

8.5 Warning reference

This section describes and explains in detail some warnings:

8.5.1 Warning 52: fragile constant pattern

Some constructors, such as the exception constructors Failure and Invalid_argument, take as parameter a string value holding a text message intended for the user.

These text messages are usually not stable over time: call sites building these constructors may refine the message in a future version to make it more explicit, etc. Therefore, it is dangerous to match over the precise value of the message. For example, until OCaml 4.02, Array.iter2 would raise the exception

Invalid_argument "arrays must have the same length"

Since 4.03 it raises the more helpful message

Invalid_argument "Array.iter2: arrays must have the same length"

but this means that any code of the form

try ...
with Invalid_argument "arrays must have the same length" -> ...

is now broken and may suffer from uncaught exceptions.

Warning 52 is there to prevent users from writing such fragile code in the first place. It does not occur on every matching on a literal string, but only in the case in which library authors expressed their intent to possibly change the constructor parameter value in the future, by using the attribute \texttt{ocaml.warn\_on\_literal\_pattern} (see the manual section on builtin attributes in 7.18.1):

type t =
  | Foo of string [@ocaml.warn_on_literal_pattern]
  | Bar of string

let no_warning = function
  | Bar "specific value" -> 0
  | _ -> 1

let warning = function
  | Foo "specific value" -> 0
  | _ -> 1
Chapter 8. Batch compilation (ocamlc) 195

> | Foo "specific value" -> 0
> --------------------------
> Warning 52: Code should not depend on the actual values of this constructor’s arguments. They are only for information and may change in future versions. (See manual section 8.5)

In particular, all built-in exceptions with a string argument have this attribute set: Invalid_argument, Failure, Sys_error will all raise this warning if you match for a specific string argument.

If your code raises this warning, you should not change the way you test for the specific string to avoid the warning (for example using a string equality inside the right-hand-side instead of a literal pattern), as your code would remain fragile. You should instead enlarge the scope of the pattern by matching on all possible values.

```ocaml
let warning = function
  | Foo _ -> 0
  | _    -> 1
```

This may require some care: if the scrutinee may return several different cases of the same pattern, or raise distinct instances of the same exception, you may need to modify your code to separate those several cases.

For example,

```ocaml
try (int_of_string count_str, bool_of_string choice_str) with
  | Failure "int_of_string" -> (0, true)
  | Failure "bool_of_string" -> (-1, false)

  should be rewritten into more atomic tests. For example, using the exception patterns documented in Section 7.21, one can write:

match int_of_string count_str with
  | exception (Failure _) -> (0, true)
  | count ->
    begin match bool_of_string choice_str with
      | exception (Failure _) -> (-1, false)
      | choice    -> (count, choice)
    end
```

The only case where that transformation is not possible is if a given function call may raises distinct exceptions with the same constructor but different string values. In this case, you will have to check for specific string values. This is dangerous API design and it should be discouraged: it’s better to define more precise exception constructors than store useful information in strings.

8.5.2 Warning 57: Ambiguous or-pattern variables under guard

The semantics of or-patterns in OCaml is specified with a left-to-right bias: a value $v$ matches the pattern $p \lor q$ if it matches $p$ or $q$, but if it matches both, the environment captured by the match is the environment captured by $p$, never the one captured by $q$. 
While this property is generally intuitive, there is at least one specific case where a different semantics might be expected. Consider a pattern followed by a when-guard: \( p \text{ when } g \rightarrow e \), for example:

\[
| ((\text{Const } x, \_ ) | (\_, \text{Const } x)) \text{ when is\_neutral } x \rightarrow \text{branch}
\]

The semantics is clear: match the scrutinee against the pattern, if it matches, test the guard, and if the guard passes, take the branch. In particular, consider the input \((\text{Const } a, \text{Const } b)\), where \(a\) fails the test \text{is\_neutral}\( a\), while \(b\) passes the test \text{is\_neutral}\( b\). With the left-to-right semantics, the clause above is not taken by its input: matching \((\text{Const } a, \text{Const } b)\) against the or-pattern succeeds in the left branch, it returns the environment \(x \rightarrow a\), and then the guard \text{is\_neutral}\( a\) is tested and fails, the branch is not taken.

However, another semantics may be considered more natural here: any pair that has one side passing the test will take the branch. With this semantics the previous code fragment would be equivalent to:

\[
| (\text{Const } x, \_) \text{ when is\_neutral } x \rightarrow \text{branch}
\]

\[
| (\_, \text{Const } x) \text{ when is\_neutral } x \rightarrow \text{branch}
\]

This is not the semantics adopted by OCaml.

Warning 57 is dedicated to these confusing cases where the specified left-to-right semantics is not equivalent to a non-deterministic semantics (any branch can be taken) relatively to a specific guard. More precisely, it warns when guard uses “ambiguous” variables, that are bound to different parts of the scrutinees by different sides of a or-pattern.
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 OCaml phrases from the input, then typechecks, compile and evaluate them, then prints the inferred type and result value, if any. The system prints a # (sharp) prompt before reading each phrase.

Input to the toplevel can span several lines. It is terminated by ;; (a double-semicolon). The toplevel input consists in one or several toplevel phrases, with the following syntax:

\[
toplevel-input ::= \{definition\}^+ ;;
| expr ;;
| # ident [directive-argument] ;;
\]

\[
directive-argument ::= string-literal
| integer-literal
| value-path
| true | false
\]

A phrase can consist of a definition, like 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 value expression (section 6.7). It is simply evaluated without performing any bindings, and its value 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:

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

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

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](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.

**-absname**

Force error messages to show absolute paths for file names.
-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.

-no-app-funct
Deactivates the applicative behaviour of functors. With this option, each functor application generates new types in its result and applying the same functor twice to the same argument yields two incompatible structures.

-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.

-nopromptcont
Do not display the secondary prompt when waiting for continuation lines in multi-line inputs. This should be used e.g. when running ocaml in an emacs window.

-nostdlib
Do not include the standard library directory in the list of directories searched for source and compiled files.

-ppx command
After parsing, pipe the abstract syntax tree through the preprocessor command. The module Ast_mapper, described in chapter 22.1 implements the external interface of a preprocessor.
-principal
Check information paths 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.

-rectype
Allow arbitrary recursive types during type-checking. By default, only recursive types where the recursion goes through an object type are supported.

-safe-string
Enforce the separation between types string and bytes, thereby making strings read-only. This will become the default in a future version of OCaml.

-short-paths
When a type is visible under several module-paths, use the shortest one when printing the type’s name in inferred interfaces and error and warning messages. Identifier names starting with an underscore _ or containing double underscores __ incur a penalty of +10 when computing their length.

-stdin
Read the standard input as a script file rather than starting an interactive session.

-strict-sequence
Force the left-hand part of each sequence to have type unit.

-strict-formats
Reject invalid formats that were accepted in legacy format implementations. You should use this flag to detect and fix such invalid formats, as they will be rejected by future OCaml versions.

-unsafe
See the corresponding option for ocamlc, chapter 8. Turn bound checking off on array and string accesses (the v.(i) and s.[i] constructs). Programs compiled with -unsafe are therefore slightly faster, but unsafe: anything can happen if the program accesses an array or string outside of its bounds.

-unsafe-string
Identify the types string and bytes, thereby making strings writable. For reasons of backward compatibility, this is the default setting for the moment, but this will change in a future version of OCaml.

-version
Print version string and exit.

-vnum
Print short version number and exit.
Chapter 9. The toplevel system (ocaml)

-no-version
Do not print the version banner at startup.

-w warning-list
Enable or disable warnings according to the argument warning-list. See section 8.2 for the syntax of the argument.

-warn-error warning-list
Mark as fatal the warnings enabled by the argument warning-list. See section 8.2 for the syntax of the argument.

-warn-help
Show the description of all available warning numbers.

-file
Use file as a script file name, even when it starts with a hyphen (-).

-help or --help
Display a short usage summary and exit.

Unix:
The following environment variables are also consulted:

LC_CTYPE
If set to iso_8859_1, accented characters (from the ISO Latin-1 character set) in string and character literals are printed as is; otherwise, they are printed as decimal escape sequences (\ddd).

TERM
When printing error messages, the toplevel system attempts to underline visually the location of the error. It consults the TERM variable to determine the type of output terminal and look up its capabilities in the terminal database.

HOME
Directory where the .ocamlinit file is searched.

9.2 Toplevel directives

The following directives control the toplevel behavior, load files in memory, and trace program execution.

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 #quit;; will exit the toplevel loop, but typing quit;; will result in an “unbound value quit” error.

General

#help;;
Prints a list of all available directives, with corresponding argument type if appropriate.
```
#quit;;
Exit the toplevel loop and terminate the ocaml command.

Loading codes

#cd "dir-name";;
Change the current working directory.

#directory "dir-name";;
Add the given directory to the list of directories searched for source and compiled files.

#remove_directory "dir-name";;
Remove the given directory from the list of directories searched for source and compiled files. Do nothing if the list does not contain the given directory.

#load "file-name";;
Load in memory a bytecode object file (.cmo file) or library file (.cma file) produced by the batch compiler ocamlc.

#load_rec "file-name";;
Load in memory a bytecode object file (.cmo file) or library file (.cma file) produced by the batch compiler ocamlc. When loading an object file that depends on other modules which have not been loaded yet, the .cmo files for these modules are searched and loaded as well, recursively. The loading order is not specified.

#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.

#mod_use "file-name";;
Similar to #use but also wrap the code into a top-level module of the same name as capitalized file name without extensions, following semantics of the compiler.

Environment queries

#show_class class-path;;
#show_class_type class-path;;
#show_exception ident;;
#show_module module-path;;
#show_module_type modtype-path;;
#show_type typeconstr;;
#show_val value-path;;
Print the signature of the corresponding component.

#show ident;;
Print the signatures of components with name ident in all the above categories.

Pretty-printing
#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 t is 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.

#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).

#remove_printer printer-name;;
Remove the named function from the table of toplevel printers.

Tracing

#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.

Compiler options

#labels bool;;
Ignore labels in function types if argument is false, or switch back to default behaviour (commuting style) if argument is true.

#ppx "file-name";;
After parsing, pipe the abstract syntax tree through the preprocessor command.

#principal bool;;
If the argument is true, check information paths during type-checking, to make sure that all types are derived in a principal way. If the argument is false, do not check information paths.
rectypes;;
Allow arbitrary recursive types during type-checking. Note: once enabled, this option
cannot be disabled because that would lead to unsoundness of the type system.

warn_error "warning-list";;
Treat as errors the warnings enabled by the argument and as normal warnings the
warnings disabled by the argument.

warnings "warning-list";;
Enable or disable warnings according to the argument.

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 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 $t_1$, but is used with type $t_2$
See section 8.4.
Reference to undefined global \textit{mod}

You have neglected to load in memory an implementation for a module with \texttt{#load}. See section \ref{section:9.3} above.

\section{Building custom toplevel systems: \texttt{ocamlmktop}}

The \texttt{ocamlmktop} command builds OCaml toplevels that contain user code preloaded at start-up.

The \texttt{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:

\begin{verbatim}
ocamlmktop -o mytoplevel foo.cmo bar.cmo gee.cmo
\end{verbatim}

This creates the bytecode file \texttt{mytoplevel}, containing the OCaml toplevel system, plus the code from the three .cmo files. This toplevel is directly executable and is started by:

\begin{verbatim}
./mytoplevel
\end{verbatim}

This enters a regular toplevel loop, except that the code from \texttt{foo.cmo}, \texttt{bar.cmo} and \texttt{gee.cmo} is already loaded in memory, just as if you had typed:

\begin{verbatim}
#load "foo.cmo";;
#load "bar.cmo";;
#load "gee.cmo";;
\end{verbatim}

on entrance to the toplevel. The modules \texttt{Foo}, \texttt{Bar} and \texttt{Gee} are not opened, though; you still have to do

\begin{verbatim}
open Foo;;
\end{verbatim}

yourself, if this is what you wish.

\section{Options}

The following command-line options are recognized by \texttt{ocamlmktop}.

\begin{itemize}
  \item \texttt{-cclib} \textit{libname}
    \begin{verbatim}
    Pass the \texttt{-l}libname option to the C linker when linking in “custom runtime” mode. See the corresponding option for \texttt{ocamlc}, in chapter \ref{section:8.5}.
    \end{verbatim}
  \item \texttt{-ccopt} \textit{option}
    \begin{verbatim}
    Pass the given option to the C compiler and linker, when linking in “custom runtime” mode. See the corresponding option for \texttt{ocamlc}, in chapter \ref{section:8.5}.
    \end{verbatim}
  \item \texttt{-custom}
    \begin{verbatim}
    Link in “custom runtime” mode. See the corresponding option for \texttt{ocamlc}, in chapter \ref{section:8.5}.
    \end{verbatim}
  \item \texttt{-I} \textit{directory}
    \begin{verbatim}
    Add the given directory to the list of directories searched for compiled object code files (.cmo and .cma).
    \end{verbatim}
  \item \texttt{-o} \textit{exec-file}
    \begin{verbatim}
    Specify the name of the toplevel file produced by the linker. The default is \texttt{a.out}.
    \end{verbatim}
\end{itemize}
Chapter 10

The runtime system (ocamlrune)

The `ocamlrune` command executes bytecode files produced by the linking phase of the `ocamlc` command.

10.1 Overview

The `ocamlrune` 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 `ocamlrune` is:

```
ocamlrun options bytecode-executable arg1 ... arg_n
```

The first non-option argument is taken to be the name of the file containing the executable bytecode. (That file is searched in the executable path as well as in the current directory.) The remaining arguments are passed to the OCaml 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 `arg_n`.

As mentioned in chapter 8, the bytecode executable files produced by the `ocamlc` command are self-executable, and manage to launch the `ocamlrune` command on themselves automatically. That is, assuming `a.out` is a bytecode executable file,

```
a.out arg1 ... arg_n
```

works exactly as

```
ocamlrune a.out arg1 ... arg_n
```

Notice that it is not possible to pass options to `ocamlrune` when invoking `a.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` ....

207
10.2 Options

The following command-line options are recognized by ocamlrun.

-\texttt{-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 \texttt{-g} option to \texttt{ocamlc} set. This is equivalent to setting the \texttt{b} flag in the \texttt{OCAMLRUNPARAM} environment variable (see below).

-\texttt{-I dir} Search the directory \texttt{dir} for dynamically-loaded libraries, in addition to the standard search path (see section 10.3).

-\texttt{-p} Print the names of the primitives known to this version of \texttt{ocamlrun} and exit.

-\texttt{-v} Direct the memory manager to print some progress messages on standard error. This is equivalent to setting \texttt{v=63} in the \texttt{OCAMLRUNPARAM} environment variable (see below).

-\texttt{-version} Print version string and exit.

-\texttt{-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 \texttt{OCAMLLIB} is not set, \texttt{CAMLLIB} will be used instead.) Used to locate the \texttt{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 \texttt{OCAMLRUNPARAM} is not set, \texttt{CAMLRUNPARAM} will be used instead.) This variable must be a sequence of parameter specifications separated by commas. 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. The options are documented below; the last six correspond to the fields of the control record documented in section 21.12.

-\texttt{b} (backtrace) Trigger the printing of a stack backtrace when an uncaught exception aborts the program. This option takes no argument.

-\texttt{p} (parser trace) Turn on debugging support for \texttt{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.
R  (randomize) Turn on randomization of all hash tables by default (see section 21.14).
   This option takes no argument.

h  The initial size of the major heap (in words).

a  (allocation_policy) The policy used for allocating in the OCaml heap. Possible values
    are 0 for the next-fit policy, and 1 for the first-fit policy. Next-fit is usually faster, but
    first-fit is better for avoiding fragmentation and the associated heap compactions.

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.

l  (stack_limit) The limit (in words) of the stack size.

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 finalization functions
 256 (= 0x100)
     Startup messages (loading the bytecode executable file, resolving shared libraries).
 512 (= 0x200)
     Computation of compaction-triggering condition.

The multiplier is k, M, or G, for multiplication by $2^{10}$, $2^{20}$, and $2^{30}$ respectively.
If the option letter is not recognized, the whole parameter is ignored; if the equal sign or the
number is missing, the value is taken as 1; if the multiplier is not recognized, it is ignored.
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 also 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, `ocamlrun` can 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 [18.1.4](#), and recorded in the bytecode executable file; `ocamlrun`, then, locates these libraries and resolves references to their primitives when the bytecode executable program starts.

The `ocamlrun` command searches shared libraries in the following directories, in the order indicated:

1. Directories specified on the `ocamlrun` 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 `ocamlrun` bytecode interpreter on itself. The second possibility indicates that OCaml has not been properly installed on your system.
Cannot exec ocamlrun
(When launching a self-executable bytecode file.) The ocamlrun could 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 ocamlrun is trying to execute (e.g. the file given as first non-option argument to ocamlrun) either does not exist, or is not a valid executable bytecode file.

Truncated bytecode file
The file that ocamlrun is trying to execute is not a valid executable bytecode file. Probably it has been truncated or mangled since created. Erase and rebuild it.

Uncaught exception
The program being executed contains a “stray” exception. That is, it raises an exception at some point, and this exception is never caught. This causes immediate termination of the program. The name of the exception is printed, along with its string, byte sequence, 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 [15]), 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, string or byte sequence, 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, string or byte sequence.
Chapter 11

Native-code compilation (ocamlopt)

This chapter describes the OCaml high-performance native-code compiler ocamlopt, which compiles OCaml 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, after all options have been processed:

- 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 OCaml).

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 OCaml standard library, to produce a native-code executable program. The order in which .cmx and .ml arguments are presented on the command line is relevant: compilation units are initialized in that order at run-time, and it is a link-time error to use a component of a unit before having initialized it. Hence, a given \( x \).cmx file must come before all .cmx files that refer to the unit \( x \).

• Arguments ending in .cmxa are taken to be libraries of object code. Such a library packs in two files (\( \text{lib}.\text{cmxa} \) and \( \text{lib}.a/.\text{lib} \) a set of object files (\( .\text{cmx} \) and \( .o/.\text{obj} \) 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.

-absname
Force error messages to show absolute paths for file names.

-anoot
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.
-bin-annot
Dump detailed information about the compilation (types, bindings, tail-calls, etc) in binary format. The information for file src.ml is put into file src.cmt. In case of a type error, dump all the information inferred by the type-checker before the error. The *.cmt files produced by -bin-annot contain more information and are much more compact than the files produced by -annot.

-c
Compile only. Suppress the linking phase of the compilation. Source code files are turned into compiled files, but no executable file is produced. This option is useful to compile modules separately.

-cc ccomp
Use ccomp as the C linker called to build the final executable and as the C compiler for compiling .c source files.

-cclib -l libname
Pass the -l libname option to the linker. This causes the given C library to be linked with the program.

-ccopt option
Pass the given option to the C compiler and linker. For instance, -ccopt -L dir causes the C linker to search for C libraries in directory dir.

-compact
Optimize the produced code for space rather than for time. This results in slightly smaller but slightly slower programs. The default is to optimize for speed.

-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. See also option -nostdlib.

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.

-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.

-no-app-funct
Deactivates the applicative behaviour of functors. With this option, each functor application generates new types in its result and applying the same functor twice to the same argument yields two incompatible structures.

-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.
Chapter 11. Native-code compilation (ocamlopt)

-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.

-nostdlib
Do not automatically add the standard library directory to the list of directories searched for compiled interface files (.cmi), compiled object code files (.cfx), and libraries (.cmxa). See also option -I.

-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 the plugin file produced.

-output-obj
Cause the linker to produce a C object file instead of an executable file. This is useful to wrap OCaml code as a C library, callable from any C program. See chapter 18 section 18.7.5. The name of the output object file must 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 16 for more information on profiling.) The -p option must be given both at compile-time and at link-time. Linking object files not compiled with -p is possible, but results in less precise profiling.

Unix:

See the Unix manual page for gprof(1) for more information about the profiles.

Full support for gprof is only available for certain platforms (currently: Intel x86 32 and 64 bits under Linux, BSD and MacOS X). 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.

-**plugin** plugin

Dynamically load the code of the given plugin (a .cmo, .cma or .cmxs file) in the compiler. plugin must exist in the same kind of code as the compiler (ocamlopt.byte must load bytecode plugins, while ocamlopt.opt must load native code plugins), and extension adaptation is done automatically for .cma files (to .cmxs files if the compiler is compiled in native code).

-**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.

-**ppx** command

After parsing, pipe the abstract syntax tree through the preprocessor command. The module Ast_mapper, described in chapter 22.1 implements the external interface of a preprocessor.

-**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.
-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.

-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 OCaml 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 OCaml 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).

-safe-string
Enforce the separation between types string and bytes, thereby making strings read-only. This will become the default in a future version of OCaml.

-short-paths
When a type is visible under several module-paths, use the shortest one when printing the type’s name in inferred interfaces and error and warning messages. Identifier names starting with an underscore _ or containing double underscores __ incur a penalty of +10 when computing their length.

-strict-sequence
Force the left-hand part of each sequence to have type unit.

-strict-formats
Reject invalid formats that were accepted in legacy format implementations. You should use this flag to detect and fix such invalid formats, as they will be rejected by future OCaml versions.

-thread
Compile or link multithreaded programs, in combination with the system threads library described in chapter 26.

-unboxed-types
When a type is unboxable (i.e. a record with a single argument or a concrete datatype with a single constructor of one argument) it will be unboxed unless annotated with [@@ocaml.boxed].

-no-unboxed-types
When a type is unboxable it will be boxed unless annotated with [@@ocaml.unboxed]. This is the default.
-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.

-unsafe-string
  Identify the types string and bytes, thereby making strings writable. For reasons of backward compatibility, this is the default setting for the moment, but this will change in a future version of OCaml.

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

-verble
  Print all external commands before they are executed, in particular invocations of the assembler, C compiler, and linker.

-version or -vnum
  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 fatal the warnings specified by the argument warning-list. Each warning can be enabled or disabled, and each warning can be fatal or non-fatal. If a warning is disabled, it isn’t displayed and doesn’t affect compilation in any way (even if it is fatal). If a warning is enabled, it is displayed normally by the compiler whenever the source code triggers it. If it is enabled and fatal, the compiler will also stop with an error after displaying 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 as fatal warning number num.

+num1..num2
  Enable warnings in the given range.

-num1..num2
  Disable warnings in the given range.

@num1..num2
  Enable and mark as fatal warnings in the given range.
Chapter 11. Native-code compilation (ocamlopt)

+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 as fatal 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 and letters which are out of the range of warnings that are currently defined are ignored. The warning are as follows.

1 Suspicious-looking start-of-comment mark.
2 Suspicious-looking end-of-comment mark.
3 Deprecated feature.
4 Fragile pattern matching: matching that will remain complete even if additional constructors are added to one of the variant types matched.
5 Partially applied function: expression whose result has function type and is ignored.
6 Label omitted in function application.
7 Method overridden.
8 Partial match: missing cases in pattern-matching.
9 Missing fields in a record pattern.
10 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).
11 Redundant case in a pattern matching (unused match case).
12 Redundant sub-pattern in a pattern-matching.
13 Instance variable overridden.
14 Illegal backslash escape in a string constant.
15 Private method made public implicitly.
16 Unerasable optional argument.
17 Undeclared virtual method.
18 Non-principal type.
19 Type without principality.
20 Unused function argument.
21 Non-returning statement.
22 Preprocessor warning.
23 Useless record with clause.
24 Bad module name: the source file name is not a valid OCaml module name.
26 Suspicious unused variable: unused variable that is bound with let or as, and doesn’t start with an underscore (_ ) character.
27 Innocuous unused variable: unused variable that is not bound with let nor as, and doesn’t start with an underscore (_ ) character.
28 Wildcard pattern given as argument to a constant constructor.
29 Unescaped end-of-line in a string constant (non-portable code).
30 Two labels or constructors of the same name are defined in two mutually recursive types.
31 A module is linked twice in the same executable.
32 Unused value declaration.
33 Unused open statement.
34 Unused type declaration.
35 Unused for-loop index.
36 Unused ancestor variable.
37 Unused constructor.
38 Unused extension constructor.
39 Unused rec flag.
40 Constructor or label name used out of scope.
41 Ambiguous constructor or label name.
42 Disambiguated constructor or label name (compatibility warning).
43 Nonoptional label applied as optional.
44 Open statement shadows an already defined identifier.
45 Open statement shadows an already defined label or constructor.
46 Error in environment variable.
47 Illegal attribute payload.
48 Implicit elimination of optional arguments.
49 Absent cmi file when looking up module alias.
50 Unexpected documentation comment.
51 Warning on non-tail calls if @tailcall present.
52 (see 8.5.1)
   Fragile constant pattern.
53 Attribute cannot appear in this context
54 Attribute used more than once on an expression
55  Inlining impossible
56  Unreachable case in a pattern-matching (based on type information).
57  (see [8.5.2])
58  Ambiguous or-pattern variables under guard
59  Missing cmx file
60  Assignment to non-mutable value

A  all warnings
C  warnings 1, 2.
D  Alias for warning 3.
E  Alias for warning 4.
F  Alias for warning 5.
K  warnings 32, 33, 34, 35, 36, 37, 38, 39.
L  Alias for warning 6.
M  Alias for warning 7.
P  Alias for warning 8.
R  Alias for warning 9.
S  Alias for warning 10.
U  warnings 11, 12.
V  Alias for warning 13.
X  warnings 14, 15, 16, 17, 18, 19, 20, 21, 22, 23, 24, 30.
Y  Alias for warning 26.
Z  Alias for warning 27.

The default setting is `-w +a-4-6-7-9-27-29-32...39-41...42-44-45-48-50`. It is displayed by `ocamlopt -help`. Note that warnings 5 and 10 are not always triggered, depending on the internals of the type checker.

-`warn-error` `warning-list`
Mark as fatal 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) marks the corresponding warnings as fatal, a `-` sign (or a lowercase letter) turns them back into non-fatal warnings, and a `@` sign both enables and marks as fatal 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+31` (only warning 31 is fatal).
-warn-help
   Show the description of all available warning numbers.

-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.

Contextual control of command-line options
   The compiler command line can be modified “from the outside” with the following mechanisms. These are experimental and subject to change. They should be used only for experimental and development work, not in released packages.
OCAMLPARAM (environment variable)
Arguments that will be inserted before or after the arguments from the command line.

ocaml_compiler_internal_params (file in the stdlib directory)
A mapping of file names to lists of arguments that will be added to the command line (and OCAMLPARAM) arguments.

OCAML_FLEXLINK (environment variable)
Alternative executable to use on native Windows for flexlink instead of the configured value.
Primarily used for bootstrapping.

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/OCaml 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 l 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 and AMD x86 processors 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 slightly between bytecode and native code.
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 OCaml 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 21.)

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 OCaml 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:

{ header }
let ident = regexp ...
[refill { refill-handler }]
rule entrypoint [arg1... argn] =
    parse regexp { action }
    | ...
    | regexp { action }
and entrypoint [arg1... argn] =
    parse ...
and ...
{ trailer }

Comments are delimited by (* and *), as in OCaml. The parse keyword, can be replaced by
the shortest keyword, with the semantic consequences explained below.

Refill handlers are a recent (optional) feature introduced in 4.02, documented below in subsection 12.2.7.
12.2.1 Header and trailer

The *header* and *trailer* sections are arbitrary OCaml 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 OCaml values (starting with a lowercase letter). Similarly, the arguments `arg1 ... argn` must be valid identifiers for OCaml. Each entry point becomes an OCaml 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 OCaml-like syntax.

\[
\text{regexp} ::= \ldots
\]

- `' 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
(diff) 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 the empty string, or a string matching regexp.

regexp1 | regexp2
(alternative) Match any string that matches 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 OCaml 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 [exp\_1\ldots exp\_n] lexbuf
(Where \textit{entrypoint} is the name of another entry point in the same lexer definition.) Recursively call the lexer on the given entry point. Notice that \texttt{lexbuf} is the last argument. Useful for lexing nested comments, for example.

### 12.2.6 Variables in regular expressions

The \texttt{as} construct is similar to "\textit{groups}" as provided by numerous regular expression packages. The type of these variables can be \texttt{string}, \texttt{char}, \texttt{string option} or \texttt{char option}.

We first consider the case of linear patterns, that is the case when all \texttt{as} bound variables are distinct. In \texttt{regexp as ident}, the type of \texttt{ident} normally is \texttt{string} (or \texttt{string option}) except when \texttt{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 \texttt{ident} is \texttt{char} (or \texttt{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 \(( \texttt{regexp as ident} ) ? \) and of \texttt{regexp\_1 | ( regexp\_2 as ident )}.

There is no linearity restriction over \texttt{as} bound variables. When a variable is bound more than once, the previous rules are to be extended as follows:

- A variable is a \texttt{char} variable when all its occurrences bind \texttt{char} occurrences in the previous sense.
- A variable is an \texttt{option} variable when the overall expression can be matched without binding this variable.

For instance, in \(( 'a' as x) | ( 'a' ( _ as x) ) \) the variable \texttt{x} is of type \texttt{char}, whereas in \(("ab" as x) | ( 'a' ( _ as x) ? ) \) the variable \texttt{x} is of type \texttt{string option}.

In some cases, a successful match may not yield a unique set of bindings. For instance the matching of \texttt{aba} by the regular expression \((('a'|"ab") as x) ("ba"|'a') as y) \) may result in binding either \texttt{x} to "ab" and \texttt{y} to "a", or \texttt{x} to "a" and \texttt{y} to "ba". The automata produced \texttt{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 Refill handlers

By default, when ocamllex reaches the end of its lexing buffer, it will silently call the refill_buff function of lexbuf structure and continue lexing. It is sometimes useful to be able to take control of refilling action; typically, if you use a library for asynchronous computation, you may want to wrap the refilling action in a delaying function to avoid blocking synchronous operations.

Since OCaml 4.02, it is possible to specify a refill-handler, a function that will be called when refill happens. It is passed the continuation of the lexing, on which it has total control. The OCaml expression used as refill action should have a type that is an instance of

\[(\text{Lexing.lexbuf -> 'a}) \rightarrow \text{Lexing.lexbuf -> 'a}\]

where the first argument is the continuation which captures the processing ocamllex would usually perform (refilling the buffer, then calling the lexing function again), and the result type that instantiates ['a] should unify with the result type of all lexing rules.

As an example, consider the following lexer that is parametrized over an arbitrary monad:

```plaintext
rule token = parse
| [' ' 't']
  { token lexbuf }
| '\n'
  { M.return EOL }
| ['0'-'9']* as i
  { M.return (INT (int_of_string i)) }
| '+'
  { M.return PLUS }
```
12.2.8 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 `ocamlyacc`

The `ocamlyacc` command produces a parser from a context-free grammar specification with attached semantic actions, in the style of `yacc`. Assuming the input file is `grammar.mly`, executing

```
ocamlyacc options grammar.mly
```

produces OCaml code for a parser in the file `grammar.ml`, and its interface in file `grammar.mli`. The generated module defines one parsing function per entry point in the grammar. These functions have the same names as the entry points. Parsing functions take as arguments a lexical analyzer (a function from lexer buffers to tokens) and a lexer buffer, and return the semantic attribute of the corresponding entry point. Lexical analyzer functions are usually generated from a lexer specification by the `ocamllex` program. Lexer buffers are an abstract data type implemented in the standard library module `Lexing`. Tokens are values from the concrete type `token`, defined in the interface file `grammar.mli` produced by `ocamlyacc`.

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 OCaml) in the “header” and “trailer” sections.
12.4.1 Header and trailer

The header and the trailer sections are OCaml 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
Declarer the given symbols constr ... constr as tokens (terminal symbols). These symbols are added as constant constructors for the token concrete type.

%%token < typexpr > constr ... constr
Declarer 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 OCaml 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
Declarer 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 OCaml 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:

```plaintext
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 OCaml 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 OCaml 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:

```ocaml
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS /* lowest precedence */
%left TIMES DIV /* medium precedence */
%nonassoc UMINUS /* highest precedence */
%start main /* the entry point */
%type <int> main

main:
    expr EOL { $1 }
;
expr:
    INT { $1 }
| LPAREN expr RPAREN { $2 }
| expr PLUS expr { $1 + $3 }
| expr MINUS expr { $1 - $3 }
| expr TIMES expr { $1 * $3 }
| expr DIV expr { $1 / $3 }
| MINUS expr %prec UMINUS { - $2 }

Here is the definition for the corresponding lexer:

(* File lexer.mll *)
{"Open Parser (* The type token is defined in parser.mli *)
exception Eof
}

rule token = parse
    [' ' 't''] { token lexbuf } (* skip blanks *)
| ['n'] { EOL }
| ['0'-'9']+ as lxm { INT(int_of_string lxm) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIV }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { raise Eof }

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:

```ocaml
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS /* lowest precedence */
%left TIMES DIV /* medium precedence */
%nonassoc UMINUS /* highest precedence */
%start main /* the entry point */
%type <int> main

main:
    expr EOL { $1 }
;
expr:
    INT { $1 }
| LPAREN expr RPAREN { $2 }
| expr PLUS expr { $1 + $3 }
| expr MINUS expr { $1 - $3 }
| expr TIMES expr { $1 * $3 }
| expr DIV expr { $1 / $3 }
| MINUS expr %prec UMINUS { - $2 }

Here is the definition for the corresponding lexer:

(* File lexer.mll *)
{"Open Parser (* The type token is defined in parser.mli *)
exception Eof
}

rule token = parse
    [' ' 't''] { token lexbuf } (* skip blanks *)
| ['n'] { EOL }
| ['0'-'9']+ 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:

(* File calc.ml *)
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
    while true do
      let result = Parser.main Lexer.token lexbuf in
      print_int result; print_newline(); flush stdout
    done
  with Lexer.Eof ->
  exit 0

To compile everything, execute:

    ocamllex lexer.mll # generates lexer.ml
    ocamlyacc parser.mly # generates parser.ml and parser.mli
    ocamlc -c parser.mli
    ocamlc -c lexer.ml
    ocamlc -c parser.ml
    ocamlc -c calc.ml
    ocamlc -o calc lexer.cmo parser.cmo calc.cmo

12.7 Common errors

ocamllex: transition table overflow, automaton is too big

The deterministic automata generated by ocamllex are limited to at most 32767 transitions. The message above indicates that your lexer definition is too complex and overflows this limit. This is commonly caused by lexer definitions that have separate rules for each of the alphabetic keywords of the language, as in the following example.

rule token = parse
  "keyword1" { KWD1 }
| "keyword2" { KWD2 }
| ...
| "keyword100" { KWD100 }
| [ 'A'-'Z' 'a'-'z' ] [ 'A'-'Z' 'a'-'z' '0'-'9' '_'] * 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)
Chapter 12. Lexer and parser generators (ocamllex, ocamlyacc)

```ocaml
[ "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 maintain 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.

12.8 Module Depend : Module dependencies.

```ocaml
module StringSet : 
  Set.S with type elt = string
module StringMap :
  Map.S with type key = string

type map_tree =
  | Node of StringSet.t * bound_map

type bound_map = map_tree StringMap.t
val make_leaf : string -> map_tree
val make_node : bound_map -> map_tree
val weaken_map : StringSet.t -> map_tree -> map_tree
val free_structure_names : StringSet.t Pervasives.ref
val open_module : bound_map -> Longident.t -> bound_map
val add_use_file : bound_map -> Parsetree.toplevel_phrase list -> unit
val add_signature : bound_map -> Parsetree.signature -> unit
val add_implementation : bound_map -> Parsetree.structure -> unit
val add_implementation_binding :
  bound_map -> Parsetree.structure -> bound_map
val add_signature_binding : bound_map -> Parsetree.signature -> bound_map
```
Chapter 13

The browser/editor (ocamlbrowser)

Since OCaml version 4.02, the OCamlBrowser tool and the Labltk library are distributed separately from the OCaml compiler. The project is now hosted at https://forge.ocamlcore.org/projects/labltk/.
Chapter 14

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 14.2.

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 14.3.

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.

14.1 Usage

14.1.1 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 or if -css-style is used.

- 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 \url{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[0,a,xs]**
  
  Dynamically load the given file, which defines a custom documentation generator. See section \ref{sec:custom}. This option is supported by the `ocamldoc` command (to load `.cmo` and `.cma` files) and by its native-code version `ocamldoc.opt` (to load `.cmxs` files). 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 separated by `\`, without blanks. For instance: \texttt{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 `-m` option indicates which parts of the comments in interface files are merged with the comments in implementation files.
-keep-code
Always keep the source code for values, methods and instance variables, when available. The source code is always kept when a .ml file is given, but is by default discarded when a .mli is given. This option keeps the source code in all cases.

-load file
Load information from file, which has been produced by ocamldoc -dump. Several -load options can be given.

-m flags
Specify merge options between interfaces and implementations. (see section 14.1.2 for details). flags can be one or several of the following characters:

  d   merge description
  a   merge @author
  v   merge @version
  l   merge @see
  s   merge @since
  b   merge @before
  o   merge @deprecated
  p   merge @param
  e   merge @raise
  r   merge @return
  A   merge everything

-no-custom-tags
Do not allow custom @-tags (see section 14.2.5).

-no-stop
Keep elements placed after/between the (**/**) special comment(s) (see section 14.2).

-o file
Output the generated documentation to file instead of ocamldoc.out. This option is meaningful only in conjunction with the -latex, -texi, or -dot options.

-pp command
Pipe sources through preprocessor command.

-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.

-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 ocamlmodoc 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.)

Options for generating HTML pages

The following options apply in conjunction with the -html option:

-all-params
  Display the complete list of parameters for functions and methods.
-charset charset
  Add information about character encoding being charset (default is iso-8859-1).

-colorize-code
  Colorize the OCaml code enclosed in [ ] and {{ }}, using colors to emphasize keywords, etc. If the code fragments are not syntactically correct, no color is added.

-css-style filename
  Use filename as the Cascading Style Sheet file.

-index-only
  Generate only index files.

-short-functors
  Use a short form to display functors:

  module M : functor (A:Module) -> functor (B:Module2) -> sig .. end

  is displayed as:

  module M (A:Module) (B:Module2) : sig .. end

Options for generating \LaTeX{} files

The following options apply in conjunction with the -latex option:

-latex-value-prefix 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 -latex-type-prefix, -latex-exception-prefix, -latex-module-prefix, -latex-module-type-prefix, -latex-class-prefix, -latex-class-type-prefix, -latex-attribute-prefix and -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.

-latextitle n,style
  Associate style number n to the given \LaTeX{} sectioning command style, e.g. section or 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 section, 2 for subsection, 3 for subsubsection, 4 for paragraph and 5 for subparagraph.

-noheader
  Suppress header in generated documentation.

-notoc
  Do not generate a a table of contents.

-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.

-notrailer
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'.

### 14.1.2 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.

### 14.1.3 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 OCamlc.

- 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.

14.2 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.

14.2.1 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 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 constructor with another 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
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
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 14. 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

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 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 constructor with another 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 *)
| Sun (** The comment for constructor Sun *)
(** The comment for type my_record *)
type my_record = {
    val foo : int ; (** Comment for field foo *)
    val bar : string ; (** Comment for field bar *)
}

(** The comment for class my_class *)
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

(** The comment for class type my_class_type *)
class type my_class_type =
    object
      (** The comment for the instance variable x. *)
      val mutable x : int
      (** The command 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 in the class, but not associated to any element. *)
    end

(** The comment for module type my_module_type. *)
module type my_module_type =
    sig
      (* Comment for value x. *)
val x : int
(* ... *)
end

14.2.2 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 *)
    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 ocamlloc causes the Stop special comments to be ignored.

14.2.3 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

```ocaml
val x : int
(* ... *)
end
```
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.

14.2.4 Text formatting

Here is the BNF grammar for the simple markup language used to format text descriptions.
Chapter 14. The documentation generator (ocamldoc)

\[text ::= \{text-element\}^+\]

\text-element ::= \n
| \{\{0...9\}\text\} | format text as a section header; the integer following \{ indicates the sectioning level.
| \{\{0...9\}\text: label\} | 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 string) on the given text.
| [\text\} | set the given text in source code style.
| {{\text\} | set the given text in preformatted source code style.
| \{v\text\} | set the given text in verbatim style.
| \%\{\text\} | target-specific content (\LaTeX\ code by default, see details in 14.2.4.4)
| \{!\text\} | insert a cross-reference to an element (see section 14.2.4.2 for the syntax of cross-references).
| \{!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.

### 14.2.4.1 List formatting

\[list ::= \n
| \{\{\text\}\}^+\]

| \{\{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.

14.2.4.2 Cross-reference formatting

Cross-references are fully qualified element names, as in the example \{!Foo.Bar.t\}. This is an ambiguous reference as it may designate a type name, a value name, a class name, etc. It is possible to make explicit the intended syntactic class, using \{!type:Foo.Bar.t\} to designate a type, and \{!val:Foo.Bar.t\} a value of the same name.

The list of possible syntactic class is as follows:

<table>
<thead>
<tr>
<th>tag</th>
<th>syntactic class</th>
</tr>
</thead>
<tbody>
<tr>
<td>module:</td>
<td>module</td>
</tr>
<tr>
<td>modtype:</td>
<td>module type</td>
</tr>
<tr>
<td>class:</td>
<td>class</td>
</tr>
<tr>
<td>classtype:</td>
<td>class type</td>
</tr>
<tr>
<td>val:</td>
<td>value</td>
</tr>
<tr>
<td>type:</td>
<td>type</td>
</tr>
<tr>
<td>exception:</td>
<td>exception</td>
</tr>
<tr>
<td>attribute:</td>
<td>attribute</td>
</tr>
<tr>
<td>method:</td>
<td>class method</td>
</tr>
<tr>
<td>section:</td>
<td>ocamldoc section</td>
</tr>
<tr>
<td>const:</td>
<td>variant constructor</td>
</tr>
<tr>
<td>recfield:</td>
<td>record field</td>
</tr>
</tbody>
</table>

In the case of variant constructors or record field, the constructor or field name should be preceded by the name of the correspond type – to avoid the ambiguity of several types having the same constructor names. For example, the constructor Node of the type tree will be referenced as \{!tree.Node\} or \{!const:tree.Node\}, or possibly \{!Mod1.Mod2.tree.Node\} from outside the module.
14.2.4.3 First sentence

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 }.

14.2.4.4 Target-specific formatting

The content inside %{foo: ... %} is target-specific and will only be interpreted by the backend foo, and ignored by the others. The backends of the distribution are latex, html, texi and man. If no target is specified (syntax {% ... %}), latex is chosen by default. Custom generators may support their own target prefix.

14.2.4.5 Recognized HTML tags

The HTML tags <b>..</b>, <code>..</code>, <i>..</i>, <ul>..</ul>, <ol>..</ol>, <li>..</li>, <center>..</center> and <h[0-9]>..</h[0-9]> can be used instead of, respectively, {b␣..}, {[␣..]}, {i␣..}, {ul␣..}, {ol␣..}, {li␣..}, {C␣..} and {[0-9]␣..}.

14.2.5 Documentation tags (@-tags)

Predefined tags

The following table gives the list of predefined @-tags, with their syntax and meaning.
<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 per @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 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 version text</td>
<td>Associate the given description (text) to the given version in order to document compatibility issues.</td>
</tr>
<tr>
<td>@version string</td>
<td>The version number for the element.</td>
</tr>
</tbody>
</table>

**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:

```markdown
(** 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 14.3.2.

### 14.3 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.

The files you can use to define custom generators are installed in the ocamldoc sub-directory of the OCaml standard library.
14.3.1 The generator modules

The type of a generator module depends on the kind of generated documentation. Here is the list of generator module types, with the name of the generator class in the module:

- for HTML: `Odoc_html.Html_generator` (class `html`),
- for \LaTeX: `Odoc_latex.Latex_generator` (class `latex`),
- for TeXinfo: `Odoc_texi.Texi_generator` (class `texi`),
- for man pages: `Odoc_man.Man_generator` (class `man`),
- for graphviz (dot): `Odoc_dot.Dot_generator` (class `dot`),
- for other kinds: `Odoc_gen.Base` (class `generator`).

That is, to define a new generator, one must implement a module with the expected signature, and with the given generator class, providing the `generate` method as entry point to make the generator generates documentation for a given list of modules:

```ocaml
method generate : 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.

It is recommended to inherit from the current generator of the same kind as the one you want to define. Doing so, it is possible to load various custom generators to combine improvements brought by each one.

This is done using first class modules (see chapter 7.10).

The easiest way to define a custom generator is the following example, here extending the current HTML generator. We don’t have to know if this is the original HTML generator defined in ocamldoc or if it has been extended already by a previously loaded custom generator:

```ocaml
module Generator (G : Odoc_html.Html_generator) =
    struct
        class html =
            object(self)
                inherit G.html as html
                (* ... *)

                method generate module_list =
                    (* ... *)
                    ()
                    (* ... *)
            end
    end
end;;

let _ = Odoc_args.extend_html_generator (module Generator : Odoc_gen.Html_functor);
```
To know which methods to override and/or which methods are available, have a look at the different base implementations, depending on the kind of generator you are extending:

- for HTML: `odoc_html.ml`
- for LATEX: `odoc_latex.ml`
- for TeXinfo: `odoc_texi.ml`
- for man pages: `odoc_man.ml`
- for graphviz (dot): `odoc_dot.ml`

### 14.3.2 Handling custom tags

Making a custom generator handle custom tags (see [14.2.5](#)) is very simple.

**For HTML**

Here is how to develop a HTML generator handling your custom tags.

The class `Odoc_html.Generator.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`, extend the current HTML generator and complete the `tag_functions` field:

```ocaml
module Generator (G : Odoc_html.Html_generator) =
  struct
    class html =
      object(self)
        inherit G.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
      end
  end
let _ = Odoc_args.extend_html_generator (module Generator : Odoc_gen.Html_functor);
```

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**

You can act the same way for other kinds of generators.
14.4 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

\[
\text{Odoc args.add_option : string * Arg.spec * string -> unit}
\]

Note: Existing command line options can be redefined using this function.

14.4.1 Compilation and usage

Defining a custom generator class in one file

Let \texttt{custom.ml} be the file defining a new generator class. Compilation of \texttt{custom.ml} can be performed by the following command:

\[
\text{ocamlc -I +ocamldoc -c custom.ml}
\]

The file \texttt{custom.cmo} is created and can be used this way:

\[
\text{ocamldoc -g custom.cmo other-options source-files}
\]

Options selecting a built-in generator to \texttt{ocamldoc}, such as \texttt{-html}, have no effect if a custom generator of the same kind is provided using \texttt{-g}. If the kinds do not match, the selected built-in generator is used and the custom one is ignored.

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 \texttt{file1.ml[i], file2.ml[i], ..., file\_n.ml[i]}. A \texttt{.cma} library file must be created, including all these files.

The following commands create the \texttt{custom.cma} file from files \texttt{file1.ml[i], ..., file\_n.ml[i]}:

\[
\begin{align*}
\text{ocamlc -I +ocamldoc -c file1.ml[i]} \\
\text{ocamlc -I +ocamldoc -c file2.ml[i]} \\
\vdots \\
\text{ocamlc -I +ocamldoc -c file\_n.ml[i]} \\
\text{ocamlc -o custom.cma -a file1.cmo file2.cmo ... file\_n.cmo}
\end{align*}
\]

Then, the following command uses \texttt{custom.cma} as custom generator:

\[
\text{ocamldoc -g custom.cma other-options source-files}
\]
Chapter 15

The debugger (ocamldebug)

This chapter describes the OCaml source-level replay debugger ocamldebug.

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.

15.1 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.

15.2 Invocation

15.2.1 Starting the debugger

The OCaml debugger is invoked by running the program ocamldebug with the name of the bytecode executable file as first argument:

    ocamldebug [options] program [arguments]

The arguments following program are optional, and are passed as command-line arguments to the program being debugged. (See also the set arguments command.) The following command-line options are recognized:

- c count
    Set the maximum number of simultaneously live checkpoints to count.
-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 15.10 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 command set socket (section 15.8.6) 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.

15.2.2 Initialization file

On start-up, the debugger will read commands from an initialization file before giving control to the user. The default file is .ocamldebug in the current directory if it exists, otherwise .ocamldebug in the user’s home directory.

15.2.3 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.

15.3 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.

### 15.3.1 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.

### 15.3.2 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.

### 15.4 Executing a program

#### 15.4.1 Events

Events are "interesting" locations in the source code, corresponding to the beginning or end of evaluation of "interesting" sub-expressions. Events are the unit of single-stepping (stepping goes to the next or previous event encountered in the program execution). Also, breakpoints can only be set at events. Thus, events play the role of line numbers in debuggers for conventional languages.

During program execution, a counter is incremented at each event encountered. The value of this counter is referred as the `current time`. Thanks to reverse execution, it is possible to jump back and forth to any time of the execution.

Here is where the debugger events (written `⊿◁`) are located in the source code:
• Following a function application:

(f arg)▷

• On entrance to a function:

fun x y z -> ▷ ...

• On each case of a pattern-matching definition (function, match...with construct, try...with construct):

function pat1 -> ▷ expr1
| ... ▷
| patN -> ▷ exprN

• Between subexpressions of a sequence:

expr1; ▷ expr2; ▷ ...; ▷ exprN

• In the two branches of a conditional expression:

if cond then ▷ expr1 else ▷ expr2

• At the beginning of each iteration of a loop:

while cond do ▷ body done
for i = a to b do ▷ body done

Exceptions: A function application followed by a function return is replaced by the compiler by a jump (tail-call optimization). In this case, no event is put after the function application.

15.4.2 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.

goto 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 15.5).

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).
15.4.3 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. If count is 0, run until the program terminates or a breakpoint is hit.

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.

15.4.4 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.
15.4.5 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.

15.5 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 [15.10]).

**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.

15.6 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
Chapter 15. The debugger (ocamldebug)

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.

15.7 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:

\[
\text{simple-expr} ::= \text{lowercase-ident}
| \{\text{capitalized-ident .}\} \text{lowercase-ident}
| *
| $ \text{integer}
| \text{simple-expr . lowercase-ident}
| \text{simple-expr . ( integer )}
| \text{simple-expr . [ integer ]}
| ! \text{simple-expr}
| ( \text{simple-expr} )
\]

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.

\textbf{print variables}

Print the values of the given variables. \texttt{print} can be abbreviated as \texttt{p}.

\textbf{display variables}

Same as \texttt{print}, but limit the depth of printing to 1. Useful to browse large data structures without printing them in full. \texttt{display} can be abbreviated as \texttt{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 \texttt{p }\$integer or \texttt{d }\$integer. Named values are valid only as long as the program is stopped. They are forgotten as soon as the program resumes execution.

\texttt{set print_depth }d

Limit the printing of values to a maximal depth of \texttt{d}.

\texttt{set print_length }l

Limit the printing of values to at most \texttt{l} nodes printed.

15.8 Controlling the debugger

15.8.1 Setting the program name and arguments

\texttt{set program file}

Set the program name to \texttt{file}.

\texttt{set arguments arguments}

Give \texttt{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 \texttt{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.

15.8.2 How programs are loaded

The \texttt{loadingmode} variable controls how the program is executed.

\texttt{set loadingmode direct}

The program is run directly by the debugger. This is the default mode.
Chapter 15. The debugger (ocamldebug)

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 15.8.6).

15.8.3 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.

15.8.4 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 ocamldebug to directory.

pwd Print the working directory for ocamldebug.

15.8.5 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.
15.8.6 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.

```scheme
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.

15.8.7 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.

```scheme
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.
```

```scheme
set bigstep count
Set the number of events between two checkpoints for long displacements.
```

```scheme
set smallstep count
Set the number of events between two checkpoints for small displacements.
```

The following commands display information on checkpoints and events:

```scheme
info checkpoints
Print a list of checkpoints.
```

```scheme
info events [module]
Print the list of events in the given module (the current module, by default).
```
15.8.8 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.

15.9 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`.

15.10 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 OCaml support.

The OCaml 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 OCaml 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 16

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, ...

16.1 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) or the ocamloptp front-end to the ocamlopt compiler (see chapter 11). When compiling modules separately, ocamlcp or ocamloptp must be used when compiling the modules (production of .cmo or .cmx 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. The same problem exists with ocamloptp.

**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 or ocamloptp, 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

277
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` or `ocamloptp` without the `-P` option defaults to `-P fm`, meaning that only function calls and pattern matching are profiled.

**Note** For compatibility with previous releases, `ocamlcp` also accepts the `-p` option, with the same arguments and behaviour as `-P`.

The `ocamlcp` and `ocamloptp` commands also accept all the options of the corresponding `ocamlc` or `ocamlopt` compiler, except the `-pp` (preprocessing) option.

### 16.2 Profiling an execution

Running an executable that has been compiled with `ocamlcp` or `ocamloptp` 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. Note that dump files produced by byte-code executables (compiled with `ocamlcp`) are compatible with the dump files produced by native executables (compiled with `ocamloptp`).

### 16.3 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 after it was compiled.

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 *)`. 
Chapter 16. Profiling (ocamlprof)

-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.

16.4 Time profiling

Profiling with ocamlprof only records execution counts, not the actual time spent within each function. There is currently no way to perform time profiling on bytecode programs generated by ocamlc.

Native-code programs generated by ocamlopt can be profiled for time and execution counts using the -p option and the standard Unix profiler gprof. Just add the -p option when compiling and linking the program:

    ocamlopt -o myprog -p other-options files
    ./myprog
    gprof myprog

OCaml function names in the output of gprof have the following format:

    Module-name_function-name_unique-number

Other functions shown are either parts of the OCaml 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 ports of ocamlopt under Linux, BSD and MacOS X 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.

Note The ocamloptp command also accepts the -p option. In that case, both kinds of profiling are performed by the program, and you can display the results with the gprof and ocamlprof commands, respectively.
Chapter 17

The ocamlbuild compilation manager

Since OCaml version 4.03, the ocamlbuild compilation manager is distributed separately from the OCaml compiler. The project is now hosted at https://github.com/ocaml/ocamlbuild/
Chapter 18

Interfacing C with OCaml

This chapter describes how user-defined primitives, written in C, can be linked with OCaml code and called from OCaml functions, and how these C functions can call back to OCaml code.

18.1 Overview and compilation information

18.1.1 Declaring primitives

\[\text{definition} ::= ...\]
\[| \text{external value-name : typexpr = external-declaration}\]

\[\text{external-declaration} ::= \text{string-literal [string-literal [string-literal]]}\]

User primitives are declared in an implementation file or \texttt{struct...end} module expression using the \texttt{external} keyword:

\texttt{external name : type = C-function-name}

This defines the value name \texttt{name} as a function with type \texttt{type} that executes by calling the given C function. For instance, here is how the \texttt{input} primitive is declared in the standard library module \texttt{Pervasives}:

\texttt{external input : in_channel -> bytes -> 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 \texttt{sig...end} signatures either as regular values

\texttt{val name : type}

thus hiding their implementation as C functions, or explicitly as “manifest” external functions

\texttt{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 OCaml function. On the other hand, it should not be used in library modules if they have side-effects at toplevel, as this direct call interferes with the linker's algorithm for removing unused modules from libraries at link-time.

The arity (number of arguments) of a primitive is automatically determined from its OCaml 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 * bytes * int * int -> int = "input2"
```

has arity 1, and the input2 C function receives one argument (which is a quadruple of OCaml values).

Type abbreviations are not expanded when determining the arity of a primitive. For instance,

```
type int_endo = int -> int
external f : int_endo -> int_endo = "f"
external g : (int -> int) -> (int -> int) = "f"
```

f has arity 1, but g has arity 2. This allows a primitive to return a functional value (as in the f example above): just remember to name the functional return type in a type abbreviation.

The language accepts external declarations with one or two flag strings in addition to the C function’s name. These flags are reserved for the implementation of the standard library.

### 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 OCaml values. It encodes objects of several base types (integers, floating-point numbers, strings, . . . ) as well as OCaml data structures. The type value and the associated conversion functions and macros are described in detail below. For instance, here is the declaration for the C function implementing the input primitive:

```
CAMLprim value input(value channel, value buffer, value offset, value length) {
...
}
```

When the primitive function is applied in an OCaml 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 OCaml 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 OCaml 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:
CAMLprim value add_nat_native(value nat1, value ofs1, value len1,
     value nat2, value ofs2, value len2,
     value carry_in)
{
    ...
}
CAMLprim value add_nat_bytecode(value * argv, int argn)
{
    return add_nat_native(argv[0], argv[1], argv[2], argv[3],
                         argv[4], argv[5], argv[6]);
}

The names of the two C functions must be given in the primitive declaration, as follows:

    external name : type =
        bytecode-C-function-name native-code-C-function-name

For instance, in the case of add_nat, the declaration is:

    external add_nat: nat -> int -> int -> nat -> int -> int -> int -> int
        = "add_nat_bytecode" "add_nat_native"

Implementing a user primitive is actually two separate tasks: on the one hand, decoding the
arguments to extract C values from the given OCaml values, and encoding the return value as an
OCaml 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 OCaml values to C values, call
the first function, and convert the returned C value to OCaml value. For instance, here is the stub
code for the input primitive:

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 is exported and accessible from OCaml.) The hard work is performed by the function
getblock, which is declared as:

long getblock(struct channel * channel, char * p, long n)
{
    ...
}
To write C code that operates on 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 value type, and conversion macros</td>
</tr>
<tr>
<td>caml/alloc.h</td>
<td>allocation functions (to create structured OCaml objects)</td>
</tr>
<tr>
<td>caml/memory.h</td>
<td>miscellaneous memory-related functions and macros (for GC interface,</td>
</tr>
<tr>
<td></td>
<td>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 OCaml (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</td>
</tr>
<tr>
<td></td>
<td>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</td>
</tr>
<tr>
<td></td>
<td>section 18.11).</td>
</tr>
</tbody>
</table>

These files reside in the caml/ subdirectory of the OCaml standard library directory, which is returned by the command `ocamlc -where` (usually `/usr/local/lib/ocaml` or `/usr/lib/ocaml`).

By default, header files in the caml/ subdirectory give only access to the public interface of the OCaml runtime. It is possible to define the macro `CAML_INTERNALS` to get access to a lower-level interface, but this lower-level interface is more likely to change and break programs that use it.

**Note:** It is recommended to define the macro `CAML_NAME_SPACE` before including these header files. If you do not define it, the header files will also define short names (without the `caml_` prefix) for most functions, which usually produce clashes with names defined by other C libraries that you might use. Including the header files without `CAML_NAME_SPACE` is only supported for backward compatibility.

### 18.1.3 Statically linking C code with OCaml 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 OCaml 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 OCaml 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 OCaml linker, that provide implementations for the user's primitives.
This builds a runtime system with the required primitives. The OCaml 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 OCaml 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 -l name`.

If you are using the native-code compiler `ocamlopt`, the `-custom` flag is not needed, as the final linking phase of `ocamlopt` always builds a standalone executable. To build a mixed OCaml/C executable, execute the `ocamlopt` command with:

- the names of the desired OCaml 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 an OCaml library file `.cma` or `.cmxa`. For instance, consider an OCaml library `mylib.cma`, built from the OCaml object files `a.cmo` and `b.cmo`, which reference C code in `libmylib.a`. If the library is built as follows:

```ocaml
ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

users of the library can simply link with `mylib.cma`:

```ocaml
ocamlc -o myprog mylib.cma ...
```

and the system will automatically add the `-custom` and `-cclib -lmylib` options, achieving the same effect as

```ocaml
ocamlc -o myprog -custom a.cmo b.cmo ... -cclib -lmylib
```

The alternative is of course to build the library without extra options:

```ocaml
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:

```ocaml
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 OCaml 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 OCaml 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 OCaml runtime system ocamlrut 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 Window.
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 OCaml 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 appro-
priate 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 li-
brary or DLL file must be installed in a place where ocamlrut 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 OCaml 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, .dll) residing in one
  of the standard library directories can also be specified as -dllib -l

Do not set the -custom flag, otherwise you’re back to static linking as described in section 18.1.3.
The ocamlmklib tool (see section 18.12) 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 an OCaml .cma library archive. Consider again an OCaml library mylib.cma, built
from the OCaml object files a.cmo and b.cmo, which reference C code in dllmylib.so. If the
library is built as follows:

```bash
ocamlc -a -o mylib.cma a.cmo b.cmo -dllib -lmylib
```

users of the library can simply link with mylib.cma:

```bash
ocamlc -o myprog mylib.cma ...
```

and the system will automatically add the -dllib -lmylib option, achieving the same effect
as

```bash
ocamlc -o myprog a.cmo b.cmo ... -dllib -lmylib
```

Using this mechanism, users of the library mylib.cma do not need to known that it references
C code, nor whether this C code must be statically linked (using -custom) or dynamically linked.

18.1.5 Choosing between static linking and dynamic linking

After having described two different ways of linking C code with OCaml code, we now review the
pros and cons of each, to help developers of mixed OCaml/C libraries decide.

The main advantage of dynamic linking is that it preserves the platform-independence of byte-
code executables. That is, the bytecode executable contains no machine code, and can therefore be
Chapter 18. Interfacing C with OCaml

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 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 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 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 ocamlc -custom -ccopt -static or 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 ocamlmklib command (see section 18.12) 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 OCaml 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 your .cmo and .cma files

The bytecode executable myprog can then be launched as usual: myprog args or /home/me/ocamlunixrun myprog args.
Notice that the bytecode libraries `unix.cma` and `threads.cma` must be given twice: when building the runtime system (so that `ocamld` 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 OCaml 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 63-bit signed integers (31-bit on 32-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 benefit 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:
Chapter 18. Interfacing C with OCaml

### 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 OCaml, explicit deallocation of the pointer using `free` is potentially dangerous, because the pointer may still be accessible from the OCaml world. Worse, the memory space deallocated by `free` can later be reallocated as part of the OCaml heap; the pointer, formerly pointing outside the OCaml heap, now points inside the OCaml heap, and this can crash the garbage collector. To avoid these problems, it is preferable to wrap the pointer in a OCaml block with tag `Abstract_tag` or `Custom_tag`.

### 18.3 Representation of OCaml data types

This section describes how OCaml data types are encoded in the `value` type.

#### 18.3.1 Atomic types

<table>
<thead>
<tr>
<th>OCaml type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>int</code></td>
<td>Unboxed integer values.</td>
</tr>
<tr>
<td><code>char</code></td>
<td>Unboxed integer values (ASCII code).</td>
</tr>
<tr>
<td><code>float</code></td>
<td>Blocks with tag <code>Double_tag</code>.</td>
</tr>
<tr>
<td><code>bytes</code></td>
<td>Blocks with tag <code>String_tag</code>.</td>
</tr>
<tr>
<td><code>string</code></td>
<td>Blocks with tag <code>String_tag</code>.</td>
</tr>
<tr>
<td><code>int32</code></td>
<td>Blocks with tag <code>Custom_tag</code>.</td>
</tr>
<tr>
<td><code>int64</code></td>
<td>Blocks with tag <code>Custom_tag</code>.</td>
</tr>
<tr>
<td><code>nativeint</code></td>
<td>Blocks with tag <code>Custom_tag</code>.</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 second label 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.)

As another optimization, unboxable record types are represented specially; unboxable record types are the immutable record types that have only one field. An unboxable type will be represented in one of two ways: boxed or unboxed. Boxed record types are represented as described above (by a block with tag 0 or `Double_array_tag`). An unboxed record type is represented directly by the value of its field (i.e. there is no block to represent the record itself).

The representation is chosen according to the following, in decreasing order of priority:

- An attribute (`[@@boxed]` or `[@@unboxed]`) on the type declaration.
- A compiler option (`-unboxed-types` or `-no-unboxed-types`).
- The default representation. In the present version of OCaml, the default is the boxed representation.

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. A constant constructor is represented by the unboxed integer equal to its constructor number. A non-constant constructor 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 example illustrates the assignment of integers and block tags to constructors:

```latex
\begin{verbatim}
  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 *)
\end{verbatim}
```

As an optimization, unboxable concrete data types are represented specially; a concrete data type is unboxable if it has exactly one constructor and this constructor has exactly one argument. Unboxable concrete data types are represented in the same ways as unboxable record types: see the description in section 18.3.2.

### 18.3.5 Objects

Objects are represented as blocks with tag `Object_tag`. The first field of the block refers to the object’s 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 guarantee 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:

```latex
  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 argument), 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 (an OCaml 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, w)` is represented by a block of size 2, whose field number 1 contains the representation of the pair `(v, w)`, rather than a block of size 3 containing `v` and `w` in fields 1 and 2.
18.4 Operations on values

18.4.1 Kind tests

- Is_long(v) is true if value v is an immediate integer, false otherwise.
- Is_block(v) is true if value v is a pointer to a block, and false if it is an immediate integer.

18.4.2 Operations on integers

- Val_long(l) returns the value encoding the long int l.
- Long_val(v) returns the long int encoded in value v.
- Val_int(i) returns the value encoding the int i.
- Int_val(v) returns the int encoded in value v.
- Val_bool(x) returns the OCaml boolean representing the truth value of the C integer x.
- Bool_val(v) returns 0 if v is the OCaml boolean false, 1 if v is true.
- Val_true, Val_false represent the OCaml booleans true and false.

18.4.3 Accessing blocks

- Wosize_val(v) returns the size of the block v, in words, excluding the header.
- Tag_val(v) returns the tag of the block v.
- Field(v, n) returns the value contained in the n\textsuperscript{th} field of the structured block v. Fields are numbered from 0 to Wosize_val(v) – 1.
- Store_field(b, n, v) stores the value v in the field number n of value b, which must be a structured block.
- Code_val(v) returns the code part of the closure v.
- caml_string_length(v) returns the length (number of bytes) of the string or byte sequence v.
- Byte(v, n) returns the n\textsuperscript{th} byte of the string or byte sequence v, with type char. Bytes are numbered from 0 to string_length(v) – 1.
- Byte_u(v, n) returns the n\textsuperscript{th} byte of the string or byte sequence v, with type unsigned char. Bytes are numbered from 0 to string_length(v) – 1.
- String_val(v) returns a pointer to the first byte of the string or byte sequence v, with type char *. This pointer is a valid C string: there is a null byte after the last byte in the string. However, OCaml strings and byte sequences can contain embedded null bytes, which will confuse the usual C functions over strings.
**Double_val(v)** returns the floating-point number contained in value \(v\), with type `double`.

**Double_field(v, n)** returns the \(n^{th}\) 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 `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 `int32 v`.

**Int64_val(v)** returns the 64-bit integer contained in the `int64 v`.

**Nativeint_val(v)** returns the long integer contained in the `nativeint v`.

**caml_field_unboxed(v)** returns the value of the field of a value \(v\) of any unboxed type (record or concrete data type).

**caml_field_boxed(v)** returns the value of the field of a value \(v\) of any boxed type (record or concrete data type).

**caml_field_unboxable(v)** calls either `caml_field_unboxed` or `caml_field_boxed` according to the default representation of unboxable types in the current version of OCaml.

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 byte sequence (or string) value of length \(n\) bytes. The sequence initially contains uninitialized bytes.

- **caml_copy_string(s)** returns a string or byte sequence 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)`, `caml_copy_int64(i)` and `caml_copy_nativeint(i)` return a value of OCaml 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 or byte sequences, copied from the pointer to a string array `p` (a `char **`). `p` must be NULL-terminated.

• `caml_alloc_float_array(n)` allocates an array of floating point numbers of size `n`. The array initially contains uninitialized values.

• `caml_alloc_unboxed(v)` returns the value (of any unboxed type) whose field is the value `v`.

• `caml_alloc_boxed(v)` allocates and returns a value (of any boxed type) whose field is the value `v`.

• `caml_alloc_unboxable(v)` calls either `caml_alloc_unboxed` or `caml_alloc_boxed` according to the default representation of unboxable types in the current version of OCaml.

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 ≤ Max_young_wosize` words, with tag `t`. If this block is a structured block (i.e. if `t < No_scan_tag`), then the fields of the block (initially containing garbage) must be initialized with legal values (using direct assignment to the fields of the block) before the next allocation.

• `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 < No_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 OCaml 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 OCaml value `v` as argument;
- `caml_raise_with_args(id, n, v)` raises the exception `id` with the OCaml values `v[0]`, ..., `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`. In particular, `CAMLlocal` and `CAMLxparam` can only be called after `CAMLparam`.

There are six `CAMLparam` macros: `CAMLparam0` to `CAMLparam5`, which take zero to five arguments respectively. If your function has no more 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:

```c
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:

```c
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);
```
Chapter 18. Interfacing C with OCaml

```c
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.

**Use with `CAMLlocalN`:** Arrays of values declared using `CAMLlocalN` must not be written to using `Store_field`. Use the normal C array syntax instead.

**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 seldom modified after registration, better performance can be achieved by calling `caml_register_generational_global_root(&v)` to register `v` (after its initialization with a valid value, but before any allocation or call to the GC functions), and `caml_remove_generational_global_root(&v)` to un-register it. In this case, you must not modify the value of `v` directly, but you must use `caml_modify_generational_global_root(&v, x)` to set it to `x`. The garbage collector takes advantage of the guarantee that `v` is not modified between calls to `caml_modify_generational_global_root` to scan it less often. This improves performance if the modifications of `v` happen less often than minor collections.

**Note:** The OCaml 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:

```
Field(v, n) = v_n;
```

If the block has been allocated with `caml_alloc_shr`, filling is performed through the `caml_initialize` function:
caml_initialize(&Field(v, n), v);

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:

\[
caml_initialize(&Field(v, n), w);
\]

Otherwise, you are updating a field that previously contained a well-formed value; then, call the `caml_modify` function:

\[
caml_modify(&Field(v, n), w);
\]

To illustrate the rules above, here is a C function that builds and returns a list containing the two integers given as parameters. First, we write it using the simplified allocation functions:

```c
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:

(* File curses.ml -- declaration of primitives and data types *)

type window (* The type "window" remains abstract *)

external initscr: unit -> window = "caml_curses_initscr"

external endwin: unit -> unit = "caml_curses_endwin"

external refresh: unit -> unit = "caml_curses_refresh"
To compile this interface:

    ocamlc -c curses.ml

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
/* File curses_stubs.c -- stub code for curses */
#include <curses.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/custom.h>

/* Encapsulation of opaque window handles (of type WINDOW *)
   as OCaml custom blocks. */
static struct custom_operations curses_window_ops = {
    "fr.inria.caml.curses_windows",
    custom_finalize_default,
    custom_compare_default,
    custom_hash_default,
    custom_serialize_default,
    custom_deserialize_default,
    custom_compare_ext_default,
    custom_compare_ext_default
};

/* Accessing the WINDOW * part of an OCaml custom block */
#define Window_val(v) (*((WINDOW **) Data_custom_val(v)))

/* Allocating an OCaml 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));
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_stubs.c

or, even simpler,

  ocamlc -c curses_stubs.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 OCaml program prog.ml that uses the curses module:

(* File prog.ml -- main program using curses *)
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 prog unix.cma curses.cmo prog.ml curses_stubs.o -cclib -lcurses

(On some machines, you may need to put -cclib -lcurses -cclib -ltermcap or -cclib -ltermcap instead of -cclib -lcurses.)
18.7 Advanced topic: callbacks from C to OCaml

So far, we have described how to call C functions from OCaml. In this section, we show how C functions can call OCaml functions, either as callbacks (OCaml calls C which calls OCaml), or with the main program written in C.

18.7.1 Applying OCaml closures from C

C functions can apply OCaml function values (closures) to OCaml values. The following functions are provided to perform the applications:

- `caml_callback(f, a)` applies the functional value `f` to the value `a` and returns the value returned by `f`.
- `caml_callback2(f, a, b)` applies the functional value `f` (which is assumed to be a curried OCaml function with two arguments) to `a` and `b`.
- `caml_callback3(f, a, b, c)` applies the functional value `f` (a curried OCaml 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 OCaml code, skipping over the C code. That is, if an OCaml function `f` calls a C function `g` that calls back an OCaml 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 OCaml 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 occurred, and `v` is the value returned by the OCaml function. If `Is_exception_result(v)` returns “true”, an exception escaped, and its value (the exception descriptor) can be recovered using `Extract_exception(v)`.

**Warning:** If the OCaml function returned with an exception, `Extract_exception` should be applied to the exception result prior to calling a function that may trigger garbage collection. Otherwise, if `v` is reachable during garbage collection, the runtime can crash since `v` does not contain a valid value.

**Example:**

```c
value call_caml_f_ex(value closure, value arg)
{
    CAMLparam2(closure, arg);
    CAMLlocal2(res, tmp);
    res = caml_callback_exn(closure, arg);
}
```
18.7.2 Obtaining or registering OCaml closures for use in C functions

There are two ways to obtain OCaml function values (closures) to be passed to the callback functions described above. One way is to pass the OCaml function as an argument to a primitive function. For example, if the OCaml code contains the declaration

```
external apply : ('a -> 'b) -> 'a -> 'b = "caml_apply"
```

the corresponding C stub can be written as follows:

```
CAMLprim value caml_apply(value vf, value vx)
{
    CAMLparam2(vf, vx);
    CAMLlocal1(vy);
    vy = caml_callback(vf, vx);
    CAMLreturn(vy);
}
```

Another possibility is to use the registration mechanism provided by OCaml. This registration mechanism enables OCaml code to register OCaml functions under some global name, and C code to retrieve the corresponding closure by this global name.

On the OCaml side, registration is performed by evaluating `Callback.register n v`. Here, `n` is the global name (an arbitrary string) and `v` the OCaml value. For instance:

```
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 OCaml 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 OCaml function `f` above:

```
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:
Chapter 18. Interfacing C with OCaml

```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 OCaml exceptions for use in C functions

The registration mechanism described above can also be used to communicate exception identifiers from OCaml to C. The OCaml 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 `caml_named_value` and pass it as first argument to the functions `raise_constant`, `raise_with_arg`, and `raise_with_string` (described in section 18.4.5) to actually raise the exception. For example, here is a C function that raises the Error exception with the given argument:

```c
void raise_error(char * msg)
{
    caml_raise_with_string(*caml_named_value("test exception"), msg);
}
```

18.7.4 Main program in C

In normal operation, a mixed OCaml/C program starts by executing the OCaml initialization code, which then may proceed to call C functions. We say that the main program is the OCaml code. In some applications, it is desirable that the C code plays the role of the main program, calling OCaml functions when needed. This can be achieved as follows:

- The C part of the program must provide a `main` function, which will override the default `main` function provided by the OCaml runtime system. Execution will start in the user-defined `main` function just like for a regular C program.

- At some point, the C code must call `caml_main(argv)` to initialize the OCaml code. The `argv` argument is a C array of strings (type `char **`), terminated with a NULL pointer, which represents the command-line arguments, as passed as second argument to `main`. The OCaml array `Sys.argv` will be initialized from this parameter. For the bytecode compiler, `argv[0]` and `argv[1]` are also consulted to find the file containing the bytecode.
• The call to `caml_main` initializes the OCaml runtime system, loads the bytecode (in the case of the bytecode compiler), and executes the initialization code of the OCaml program. Typically, this initialization code registers callback functions using `Callback.register`. Once the OCaml initialization code is complete, control returns to the C code that called `caml_main`.

• The C code can then invoke OCaml functions using the callback mechanism (see section [18.7.1]).

### 18.7.5 Embedding the OCaml code in the C code

The bytecode compiler in custom runtime mode (`ocamlc -custom`) normally appends the bytecode to the executable file containing the custom runtime. This has two consequences. First, the final linking step must be performed by `ocamlc`. Second, the OCaml runtime library must be able to find the name of the executable file from the command-line arguments. When using `caml_main(argv)` as in section [18.7.4], this means that `argv[0]` or `argv[1]` must contain the executable file name.

An alternative is to embed the bytecode in the C code. The `-output-obj` option to `ocamlc` is provided for this purpose. It causes the `ocamlc` compiler to output a C object file (.o file, .obj under Windows) containing the bytecode for the OCaml part of the program, as well as a `caml_startup` function. The C object file produced by `ocamlc -output-obj` can then be linked with C code using the standard C compiler, or stored in a C library.

The `caml_startup` function must be called from the main C program in order to initialize the OCaml runtime and execute the OCaml initialization code. Just like `caml_main`, it takes one `argv` parameter containing the command-line parameters. Unlike `caml_main`, this `argv` parameter is used only to initialize `Sys.argv`, but not for finding the name of the executable file.

The `-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) containing the bytecode for the OCaml 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 OCaml code, it produces a shared library that can run the OCaml 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 OCaml modules on the command-line, as well as the OCaml 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 OCaml libraries used. For instance, assume the OCaml 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
c -o myprog C objects and libraries \
camlcode.o -L'ocamlc -where' -lunix -lcamlrun
```

With `ocamlopt`, you should do:

```bash
ocamlopt -output-obj -o camlcode.o unix.cmxa other .cmx and .cmxa files
```
cc -o myprog C objects and libraries \ 
    camlcode.o -L'ocamlc -where' -lunix -lasmrun

**Warning:** On some ports, special options are required on the final linking phase that links together the object file produced by the `-output-obj` option and the remainder of the program. Those options are shown in the configuration file `config/Makefile` generated during compilation of OCaml, as the variables `BYTECCLINKOPTS` (for object files produced by `ocamlc -output-obj`) and `NATIVECCLINKOPTS` (for object files produced by `ocamlopt -output-obj`).

- Windows with the MSVC compiler: 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`.
- other systems: you may have to add one or more of `-lcurses`, `-lm`, `-ldl`, depending on your OS and C compiler.

**Stack backtraces.** When OCaml bytecode produced by `ocamlc -g` is embedded in a C program, no debugging information is included, and therefore it is impossible to print stack backtraces on uncaught exceptions. This is not the case when native code produced by `ocamlopt -g` is embedded in a C program: stack backtrace information is available, but the backtrace mechanism needs to be turned on programmatically. This can be achieved from the OCaml side by calling `Printexc.record_backtrace true` in the initialization of one of the OCaml modules. This can also be achieved from the C side by calling `caml_record_backtrace(Val_int(1));` in the OCaml-C glue code.

### 18.8 Advanced example with callbacks

This section illustrates the callback facilities described in section [18.7](#). We are going to package some OCaml functions in such a way that they can be linked with C code and called from C just like any C functions. The OCaml functions are defined in the following `mod.ml` OCaml source:

```ocaml
(* File mod.ml -- some "useful" OCaml 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:

```c
/* File modwrap.c -- wrappers around the OCaml functions */
```
```c
#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 OCaml code to a C object file and put it in a C library along with the
stub code in modwrap.c and the OCaml runtime system:

    ocamlc -custom -output-obj -o modcaml.o mod.ml
    ocamlc -c modwrap.c
    cp `ocamlc -where`/libcamlrun.a mod.a && chmod +w 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 run-
     time 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 OCaml functions */

    #include <stdio.h>
    #include <caml/callback.h>

    extern int fib(int n);
    extern char * format_result(int n);

    int main(int argc, char ** argv)
    {
```
int result;

/* Initialize OCaml 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 -I `ocamlc -where` 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 OCaml’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.
• int (*compare_ext)(value v1, value v2)
  (Since 3.12.1) The compare_ext field contains a pointer to a C function that is called whenever one custom block and one unboxed integer are compared using OCaml’s generic comparison operators (=, <>, <=, >=, <, > and compare). As in the case of the compare field, the C function should return 0 if the two arguments are structurally equal, a negative integer if the first argument compares less than the second argument, and a positive integer if the first argument compares greater than the second argument.

The compare_ext field can be set to custom_compare_ext_default; this default comparison function simply raises Failure.

• intnat (*hash)(value v)
  The hash field contains a pointer to a C function that is called whenever OCaml’s generic hash operator (see module Hashtbl) is applied to a custom block. The C function can return an arbitrary 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, uintnat * wsize_32, uintnat * 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 OCaml 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.

• uintnat (*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 OCaml 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 wsize_32 result of the serialize operation if the architecture is 32 bits, or wsize_64 if the architecture is 64 bits.

The deserialize field can be set to custom_deserialize_default to indicate that deserialization is not supported. In this case, do not register the 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 OCaml allocation functions, and do not perform a callback into OCaml code. Do not use \texttt{CAMLparam} to register the parameters to these functions, and do not use \texttt{CAMLreturn} to return the result.

\section*{18.9.2 Allocating custom blocks}

Custom blocks must be allocated via the \texttt{caml\_alloc\_custom} function:

\begin{verbatim}
caml_alloc_custom(ops, size, used, max)
\end{verbatim}

returns a fresh custom block, with room for \texttt{size} bytes of user data, and whose associated 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 OCaml incremental major collector adjusts its speed relative to the allocation rate of the program. The faster the program allocates, the harder the GC works in order to reclaim quickly unreachable blocks and avoid having large amount of “floating garbage” (unreferenced objects that the GC has not yet collected).

Normally, the allocation rate is measured by counting the in-heap size of allocated blocks. However, it often happens that finalized objects contain pointers to out-of-heap memory blocks and other resources (such as file descriptors, X Windows bitmaps, etc.). For those blocks, the in-heap size of blocks is not a good measure of the quantity of resources allocated by the program.

The two arguments \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/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/max} = 1/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/max} ratio.

\section*{18.9.3 Accessing custom blocks}

The data part of a custom block \texttt{v} can be accessed via the pointer \texttt{Data\_custom\_val(v)}. This pointer has type \texttt{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 OCaml heap. In other terms, never store an OCaml 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>caml_serialize_int_1</td>
<td>Write a 1-byte integer</td>
</tr>
<tr>
<td>caml_serialize_int_2</td>
<td>Write a 2-byte integer</td>
</tr>
<tr>
<td>caml_serialize_int_4</td>
<td>Write a 4-byte integer</td>
</tr>
<tr>
<td>caml_serialize_int_8</td>
<td>Write a 8-byte integer</td>
</tr>
<tr>
<td>caml_serialize_float_4</td>
<td>Write a 4-byte float</td>
</tr>
<tr>
<td>caml_serialize_float_8</td>
<td>Write a 8-byte float</td>
</tr>
<tr>
<td>caml_serialize_block_1</td>
<td>Write an array of 1-byte quantities</td>
</tr>
<tr>
<td>caml_serialize_block_2</td>
<td>Write an array of 2-byte quantities</td>
</tr>
<tr>
<td>caml_serialize_block_4</td>
<td>Write an array of 4-byte quantities</td>
</tr>
<tr>
<td>caml_serialize_block_8</td>
<td>Write an array of 8-byte quantities</td>
</tr>
<tr>
<td>caml_deserialize_uint_1</td>
<td>Read an unsigned 1-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_sint_1</td>
<td>Read a signed 1-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_uint_2</td>
<td>Read an unsigned 2-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_sint_2</td>
<td>Read a signed 2-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_uint_4</td>
<td>Read an unsigned 4-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_sint_4</td>
<td>Read a signed 4-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_uint_8</td>
<td>Read an unsigned 8-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_sint_8</td>
<td>Read a signed 8-byte integer</td>
</tr>
<tr>
<td>caml_deserialize_float_4</td>
<td>Read a 4-byte float</td>
</tr>
<tr>
<td>caml_deserialize_float_8</td>
<td>Read an 8-byte float</td>
</tr>
<tr>
<td>caml_deserialize_block_1</td>
<td>Read an array of 1-byte quantities</td>
</tr>
<tr>
<td>caml_deserialize_block_2</td>
<td>Read an array of 2-byte quantities</td>
</tr>
<tr>
<td>caml_deserialize_block_4</td>
<td>Read an array of 4-byte quantities</td>
</tr>
<tr>
<td>caml_deserialize_block_8</td>
<td>Read an array of 8-byte quantities</td>
</tr>
<tr>
<td>caml_deserialize_error</td>
<td>Signal an error during deserialization; input_value or Marshal.from... raise a Failure 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
\texttt{struct custom\_operation} blocks for one with the same identifier, and calling its \texttt{deserialize}
function to fill the data part of the custom block.

\subsection*{18.9.5 Choosing identifiers}

Identifiers in \texttt{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 deserialisation 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.

\subsection*{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 \texttt{alloc\_final} function is still available to allocate a custom block
with a given finalization function, but default comparison, hashing and serialization functions.
\texttt{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 \( \text{used} \) and \( \text{max} \) are used to control the speed
of garbage collection, as described for \texttt{caml\_alloc\_custom}.

\subsection*{18.10 Advanced topic: cheaper C call}

This section describe how to make calling C functions cheaper.

\textbf{Note:} this only applies to the native compiler. So whenever you use any of these methods, you
have to provide an alternative byte-code stub that ignores all the special annotations.

\subsection*{18.10.1 Passing unboxed values}

We said earlier that all OCaml objects are represented by the C type \texttt{value}, and one has to use
macros such as \texttt{Int\_val} to decode data from the \texttt{value} type. It is however possible to tell OCaml
to do this for us and pass arguments unboxed to the C function. Similarly it is possible to tell
OCaml to expect the result unboxed and box it for us.

The motivation is that, by letting the OCaml compiler deal with boxing, it can often decide to
suppress it entirely.

For instance let’s consider this example:

\begin{verbatim}
external foo : float -> float -> float = "foo"

let f a b =
\end{verbatim}
let len = Array.length a in
assert (Array.length b = len);
let res = Array.make len 0. in
for i = 0 to len - 1 do
  res.(i) <- foo a.(i) b.(i)
done

Float arrays are unboxed in OCaml, however the C function foo expect its arguments as boxed
floats and returns a boxed float. Hence the OCaml compiler has no choice but to box a.(i) and
b.(i) and unbox the result of foo. This results in the allocation of \(3 \times \text{len}\) temporary float values.

Now if we annotate the arguments and result with [@unboxed], the compiler will be able to
avoid all these allocations:

```ocaml
external foo :
  (float [@unboxed])
-> (float [@unboxed])
-> (float [@unboxed])
= "foo_byte" "foo"
```

In this case the C functions must look like:

```c
CAMLprim double foo(double a, double b)
{
  ...
}

CAMLprim value foo_byte(value a, value b)
{
  return caml_copy_double(foo(Double_val(a), Double_val(b)))
}
```

For convenience, when all arguments and the result are annotated with [@unboxed], it is
possible to put the attribute only once on the declaration itself. So we can also write instead:

```ocaml
external foo : float -> float -> float = "foo_byte" "foo" [@unboxed]
```

The following table summarize what OCaml types can be unboxed, and what C types should
be used in correspondence:

<table>
<thead>
<tr>
<th>OCaml type</th>
<th>C type</th>
</tr>
</thead>
<tbody>
<tr>
<td>float</td>
<td>double</td>
</tr>
<tr>
<td>int32</td>
<td>int32_t</td>
</tr>
<tr>
<td>int64</td>
<td>int64_t</td>
</tr>
<tr>
<td>nativeint</td>
<td>intnat</td>
</tr>
</tbody>
</table>

Similarly, it is possible to pass untagged OCaml integers between OCaml and C. This is done
by annotating the arguments and/or result with [@unboxed]:

```ocaml
let len = Array.length a in
assert (Array.length b = len);
let res = Array.make len 0. in
for i = 0 to len - 1 do
  res.(i) <- foo a.(i) b.(i)
done
```
The corresponding C type must be `intnat`.  

**Note:** do not use the C `int` type in correspondence with `int [@untagged]`). This is because they often differ in size.

### 18.10.2 Direct C call

In order to be able to run the garbage collector in the middle of a C function, the OCaml compiler generates some bookkeeping code around C calls. Technically it wraps every C call with the C function `caml_c_call` which is part of the OCaml runtime.

For small functions that are called repeatedly, this indirection can have a big impact on performances. However this is not needed if we know that the C function doesn’t allocate and doesn’t raise exceptions. We can instruct the OCaml compiler of this fact by annotating the external declaration with the attribute `[@@noalloc]`:

```ocaml
external bar : int -> int -> int = "foo" [@@noalloc]
```

In this case calling `bar` from OCaml is as cheap as calling any other OCaml function, except for the fact that the OCaml compiler can’t inline C functions...

### 18.10.3 Example: calling C library functions without indirection

Using these attributes, it is possible to call C library functions with no indirection. For instance many math functions are defined this way in the OCaml standard library:

```ocaml
external sqrt : float -> float = "caml_sqrt_float" "sqrt"  
[@@unboxed] [@@noalloc]  
(** Square root. *)
```

```ocaml
external exp : float -> float = "caml_exp_float" "exp" [@@unboxed] [@@noalloc]  
(** Exponential. *)
```

```ocaml
external log : float -> float = "caml_log_float" "log" [@@unboxed] [@@noalloc]  
(** Natural logarithm. *)
```

### 18.11 Advanced topic: multithreading

Using multiple threads (shared-memory concurrency) in a mixed OCaml/C application requires special precautions, which are described in this section.

### 18.11.1 Registering threads created from C

Callbacks from C to OCaml are possible only if the calling thread is known to the OCaml runtime system. Threads created from OCaml (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 OCaml 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 OCaml 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 OCaml run-time system. Returns 1 on success, 0 on error. If the calling thread was not previously registered, does nothing and returns 0.

### 18.11.2 Parallel execution of long-running C code

The OCaml run-time system is not reentrant: at any time, at most one thread can be executing OCaml code or C code that uses the OCaml run-time system. Technically, this is enforced by a “master lock” that any thread must hold while executing such code.

When OCaml 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 OCaml 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 OCaml threads to run concurrently with its operations. The C code must re-acquire the master lock before returning to OCaml. 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 OCaml resources, enabling other threads to run OCaml code in parallel with the execution of the calling thread.

- **caml_acquire_runtime_system()** The calling thread re-acquires the master lock and other OCaml resources. It may block until no other thread uses the OCaml 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 OCaml data, nor call any function of the run-time system, nor call back into OCaml code. Consequently, arguments provided by OCaml to the C primitive must be copied into C data structures before calling **caml_release_runtime_system()**, and results to be returned to OCaml must be encoded as OCaml 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 OCaml 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
OCaml heap. */
name = stat_alloc(caml_string_length(vname) + 1);
name = caml_strdup (String_val(vname));
/* Release the OCaml run-time system */
caml_release_runtime_system();
/* Resolve the name */
h = gethostbyname(name);
/* Re-acquire the OCaml run-time system */
caml_acquire_runtime_system();
/* Encode the relevant fields of h as the OCaml value vres */
... /* Omitted */
/* Return to OCaml */
CAMLreturn (vres);
}

Callbacks from C to OCaml must be performed while holding the master lock to the OCaml
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 OCaml (e.g. an event loop in
a GUI application), the run-time system must be acquired before the callback and released after:

caml_acquire_runtime_system();
/* Resolve OCaml function vfun to be invoked */
/* Build OCaml argument varg to the callback */
vres = callback(vfun, varg);
/* Copy relevant parts of result vres to C data structures */
caml_release_runtime_system();

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 OCaml run-time system,
typically a blocking input/output operation.

18.12 Building mixed C/OCaml libraries: ocamlmklib

The ocamlmklib command facilitates the construction of libraries containing both OCaml 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:
• OCaml source files and object files (.cmo, .cmx, .ml) comprising the OCaml part of the library;

• C object files (.o, .a, respectively, .obj, .lib) comprising the C part of the library;

• Support libraries for the C part (-l<lib>).

It generates the following outputs:

• An OCaml bytecode library .cma incorporating the .cmo and .ml OCaml files given as arguments, and automatically referencing the C library generated with the C object files.

• An OCaml native-code library .cmxa incorporating the .cmx and .ml OCaml 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, -l, -rpath, -l, -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<lib>).

-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 OCaml library. ocamlmklib will generate output.cma and/or output.cmxa. If not specified, defaults to a.
Chapter 18. Interfacing C with OCaml

-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.

On native Windows, the following environment variable is also consulted:

OCAML_FLEXLINK
   Alternative executable to use instead of the configured value. Primarily used for bootstrapping.

Example  Consider an OCaml 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 an OCaml 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 OCaml 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 -dlllib -lzip \
   -cclib -lz -cclib -lz -ccopt -L/usr/local/zlib
ocamlopt -a -o zip.cmx.zip.cmx -cclib -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

Note: This example is on a Unix system. The exact command lines may be different on other systems.

If shared libraries are not supported, the following commands are performed instead:

ocamlc -a -custom -o zip.cma zip.cmo -cclib -lzip \
   -cclib -lz -ccopt -L/usr/local/zlib
ocamlopt -a -o zip.cmx.zip.cmx -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.
Chapter 19

Optimisation with Flambda

19.1 Overview

Flambda is the term used to describe a series of optimisation passes provided by the native code compilers as of OCaml 4.03.

Flambda aims to make it easier to write idiomatic OCaml code without incurring performance penalties.

To use the Flambda optimisers it is necessary to pass the -flambda option to the OCaml configure script. (There is no support for a single compiler that can operate in both Flambda and non-Flambda modes.) Code compiled with Flambda cannot be linked into the same program as code compiled without Flambda. Attempting to do this will result in a compiler error.

Whether or not a particular ocamlopt uses Flambda may be determined by invoking it with the -config option and looking for any line starting with "flambda:". If such a line is present and says "true", then Flambda is supported, otherwise it is not.

Flambda provides full optimisation across different compilation units, so long as the .cmx files for the dependencies of the unit currently being compiled are available. (A compilation unit corresponds to a single .ml source file.) However it does not yet act entirely as a whole-program compiler: for example, elimination of dead code across a complete set of compilation units is not supported.

Optimisation with Flambda is not currently supported when generating bytecode.

Flambda should not in general affect the semantics of existing programs. Two exceptions to this rule are: possible elimination of pure code that is being benchmarked (see section 19.14) and changes in behaviour of code using unsafe operations (see section 19.15).

Flambda does not yet optimise array or string bounds checks. Neither does it take hints for optimisation from any assertions written by the user in the code.

Consult the Glossary at the end of this chapter for definitions of technical terms used below.

19.2 Command-line flags

The Flambda optimisers provide a variety of command-line flags that may be used to control their behaviour. Detailed descriptions of each flag are given in the referenced sections. Those sections also describe any arguments which the particular flags take.
Commonly-used options:

- **-O2** Perform more optimisation than usual. Compilation times may be lengthened. (This flag is an abbreviation for a certain set of parameters described in section 19.5.)

- **-O3** Perform even more optimisation than usual, possibly including unrolling of recursive functions. Compilation times may be significantly lengthened.

- **-Oclassic**
  Make inlining decisions at the point of definition of a function rather than at the call site(s). This mirrors the behaviour of OCaml compilers not using Flambda. Compared to compilation using the new Flambda inlining heuristics (for example at -O2) it produces smaller .cmx files, shorter compilation times and code that probably runs rather slower. When using -Oclassic, only the following options described in this section are relevant: -inlining-report and -inline. If any other of the options described in this section are used, the behaviour is undefined and may cause an error in future versions of the compiler.

- **-inlining-report**
  Emit .inlining files (one per round of optimisation) showing all of the inliner’s decisions.

Less commonly-used options:

- **-remove-unused-arguments**
  Remove unused function arguments even when the argument is not specialised. This may have a small performance penalty. See section 19.10.3.

- **-unbox-closures**
  Pass free variables via specialised arguments rather than closures (an optimisation for reducing allocation). See section 19.9.3. This may have a small performance penalty.

Advanced options, only needed for detailed tuning:

- **-inline**
  The behaviour depends on whether -Oclassic is used.
  
  - When not in -Oclassic mode, -inline limits the total size of functions considered for inlining during any speculative inlining search. (See section 19.3.6.) Note that this parameter does not control the assessment as to whether any particular function may be inlined. Raising it to excessive amounts will not necessarily cause more functions to be inlined.
  
  - When in -Oclassic mode, -inline behaves as in previous versions of the compiler: it is the maximum size of function to be considered for inlining. See section 19.3.1.

- **-inline-toplevel**
  The equivalent of -inline but used when speculative inlining starts at toplevel. See section 19.3.6. Not used in -Oclassic mode.

- **-inline-branch-factor**
  Controls how the inliner assesses whether a code path is likely to be hot or cold. See section 19.3.5.
Chapter 19. Optimisation with Flambda

- **inline-alloc-cost, inline-branch-cost, inline-call-cost**
  Controls how the inliner assesses the runtime performance penalties associated with various operations. See section 19.3.5.

- **inline-indirect-cost, inline-prim-cost**
  Likewise.

- **inline-lifting-benefit**
  Controls inlining of functors at toplevel. See section 19.3.5.

- **inline-max-depth**
  The maximum depth of any speculative inlining search. See section 19.3.6.

- **inline-max-unroll**
  The maximum depth of any unrolling of recursive functions during any speculative inlining search. See section 19.3.6.

- **no-unbox-free-vars-of-closures**
  Do not unbox closure variables. See section 19.9.1.

- **no-unbox-specialised-args**
  Do not unbox arguments to which functions have been specialised. See section 19.9.2.

- **rounds**
  How many rounds of optimisation to perform. See section 19.2.1.

- **unbox-closures-factor**
  Scaling factor for benefit calculation when using -unbox-closures. See section 19.9.3.

Notes

- The set of command line flags relating to optimisation should typically be specified to be the same across an entire project. Flambda does not currently record the requested flags in the .cmx files. As such, inlining of functions from previously-compiled units will subject their code to the optimisation parameters of the unit currently being compiled, rather than those specified when they were previously compiled. It is hoped to rectify this deficiency in the future.

- Flambda-specific flags do not affect linking with the exception of affecting the optimisation of code in the startup file (containing generated functions such as currying helpers). Typically such optimisation will not be significant, so eliding such flags at link time might be reasonable.

- Flambda-specific flags are silently accepted even when the -flambda option was not provided to the configure script. (There is no means provided to change this behaviour.) This is intended to make it more straightforward to run benchmarks with and without the Flambda optimisers in effect.

- Some of the Flambda flags may be subject to change in future releases.
19.2.1 Specification of optimisation parameters by round

Flambda operates in *rounds*: one round consists of a certain sequence of transformations that may then be repeated in order to achieve more satisfactory results. The number of rounds can be set manually using the `-rounds` parameter (although this is not necessary when using predefined optimisation levels such as with `-O2` and `-O3`). For high optimisation the number of rounds might be set at 3 or 4.

Command-line flags that may apply per round, for example those with `-cost` in the name, accept arguments of the form:

\[ n \mid \text{rounds}=n[,\ldots] \]

- If the first form is used, with a single integer specified, the value will apply to all rounds.
- If the second form is used, zero-based *round* integers specify values which are to be used only for those rounds.

The flags `-Oclassic`, `-O2` and `-O3` are applied before all other flags, meaning that certain parameters may be overridden without having to specify every parameter usually invoked by the given optimisation level.

19.3 Inlining

*Inlining* refers to the copying of the code of a function to a place where the function is called. The code of the function will be surrounded by bindings of its parameters to the corresponding arguments.

The aims of inlining are:

- to reduce the runtime overhead caused by function calls (including setting up for such calls and returning afterwards);
- to reduce instruction cache misses by expressing frequently-taken paths through the program using fewer machine instructions; and
- to reduce the amount of allocation (especially of closures).

These goals are often reached not just by inlining itself but also by other optimisations that the compiler is able to perform as a result of inlining.

When a recursive call to a function (within the definition of that function or another in the same mutually-recursive group) is inlined, the procedure is also known as *unrolling*. This is somewhat akin to loop peeling. For example, given the following code:

```plaintext
let rec fact x =
  if x = 0 then
    1
  else
    x * fact (x - 1)

let n = fact 4
```
unrolling once at the call site \( \text{fact } 4 \) produces (with the body of \( \text{fact} \) unchanged):

\[
\text{let } n = \\
\quad \text{if } 4 = 0 \text{ then } 1 \\
\quad \text{else } 4 \times \text{fact} (4 - 1)
\]

This simplifies to:

\[
\text{let } n = 4 \times \text{fact } 3
\]

Flambda provides significantly enhanced inlining capabilities relative to previous versions of the compiler.

**Aside: when inlining is performed**

Inlining is performed together with all of the other Flambda optimisation passes, that is to say, after closure conversion. This has three particular advantages over a potentially more straightforward implementation prior to closure conversion:

- It permits higher-order inlining, for example when a non-inlinable function always returns the same function yet with different environments of definition. Not all such cases are supported yet, but it is intended that such support will be improved in future.

- It is easier to integrate with cross-module optimisation, since imported information about other modules is already in the correct intermediate language.

- It becomes more straightforward to optimise closure allocations since the layout of closures does not have to be estimated in any way: it is known. Similarly, it becomes more straightforward to control which variables end up in which closures, helping to avoid closure bloat.

**19.3.1 Classic inlining heuristic**

In -Oclassic mode the behaviour of the Flambda inliner mimics previous versions of the compiler. (Code may still be subject to further optimisations not performed by previous versions of the compiler: functors may be inlined, constants are lifted and unused code is eliminated all as described elsewhere in this chapter. See sections [19.3.3](#) [19.8.1](#) and [19.10](#) At the definition site of a function, the body of the function is measured. It will then be marked as eligible for inlining (and hence inlined at every direct call site) if:

- the measured size (in unspecified units) is smaller than that of a function call plus the argument of the -inline command-line flag; and

- the function is not recursive.

Non-Flambda versions of the compiler cannot inline functions that contain a definition of another function. However -Oclassic does permit this. Further, non-Flambda versions also cannot inline functions that are only themselves exposed as a result of a previous pass of inlining, but again this is permitted by -Oclassic. For example:
module M : sig
  val i : int
end = struct
  let f x =
    let g y = x + y in
    g
    let h = f 3
    let i = h 4 (* h is correctly discovered to be g and inlined *)
end

All of this contrasts with the normal Flambda mode, that is to say without -Oclassic, where:
• the inlining decision is made at the call site; and
• recursive functions can be handled, by specialisation (see below).

The Flambda mode is described in the next section.

19.3.2 Overview of “Flambda” inlining heuristics

The Flambda inlining heuristics, used whenever the compiler is configured for Flambda and
-Oclassic was not specified, make inlining decisions at call sites. This helps in situations where
the context is important. For example:

let f b x =
  if b then
    x
  else
    ... big expression ...

let g x = f true x

In this case, we would like to inline f into g, because a conditional jump can be eliminated and
the code size should reduce. If the inlining decision has been made after the declaration of f without
seeing the use, its size would have probably made it ineligible for inlining; but at the call site, its
final size can be known. Further, this function should probably not be inlined systematically: if
b is unknown, or indeed false, there is little benefit to trade off against a large increase in code
size. In the existing non-Flambda inliner this isn’t a great problem because chains of inlining were
cut off fairly quickly. However it has led to excessive use of overly-large inlining parameters such
as -inline 10000.

In more detail, at each call site the following procedure is followed:
• Determine whether it is clear that inlining would be beneficial without, for the moment, doing
  any inlining within the function itself. (The exact assessment of benefit is described below.)
  If so, the function is inlined.

• If inlining the function is not clearly beneficial, then inlining will be performed speculatively
  inside the function itself. The search for speculative inlining possibilities is controlled by two
  parameters: the inlining threshold and the inlining depth. (These are described in more detail
  below.)
If such speculation shows that performing some inlining inside the function would be beneficial, then such inlining is performed and the resulting function inlined at the original call site.

Otherwise, nothing happens.

Inlining within recursive functions of calls to other functions in the same mutually-recursive group is kept in check by an unrolling depth, described below. This ensures that functions are not unrolled to excess. (Unrolling is only enabled if -O3 optimisation level is selected and/or the -inline-max-unroll flag is passed with an argument greater than zero.)

### 19.3.3 Handling of specific language constructs

**Functors**

There is nothing particular about functors that inhibits inlining compared to normal functions. To the inliner, these both look the same, except that functors are marked as such.

Applications of functors at toplevel are biased in favour of inlining. (This bias may be adjusted: see the documentation for -inline-lifting-benefit below.)

Applications of functors not at toplevel, for example in a local module inside some other expression, are treated by the inliner identically to normal function calls.

**First-class modules**

The inliner will be able to consider inlining a call to a function in a first class module if it knows which particular function is going to be called. The presence of the first-class module record that wraps the set of functions in the module does not per se inhibit inlining.

**Objects**

Method calls to objects are not at present inlined by Flambda.

### 19.3.4 Inlining reports

If the -inlining-report option is provided to the compiler then a file will be emitted corresponding to each round of optimisation. For the OCaml source file basename.ml the files are named basename.round.inlining.org, with round a zero-based integer. Inside the files, which are formatted as “org mode”, will be found English prose describing the decisions that the inliner took.

### 19.3.5 Assessment of inlining benefit

Inlining typically results in an increase in code size, which if left unchecked, may not only lead to grossly large executables and excessive compilation times but also a decrease in performance due to worse locality. As such, the Flambda inliner trades off the change in code size against the expected runtime performance benefit, with the benefit being computed based on the number of operations that the compiler observes may be removed as a result of inlining.

For example given the following code:
let f b x =  
    if b then  
        x  
    else  
        ... big expression ...  

let g x = f true x  

it would be observed that inlining of \textit{f} would remove:  
\begin{itemize}  
    \item one direct call;  
    \item one conditional branch.  
\end{itemize}  

Formally, an estimate of runtime performance benefit is computed by first summing the cost of the operations that are known to be removed as a result of the inlining and subsequent simplification of the inlined body. The individual costs for the various kinds of operations may be adjusted using the various \texttt{-inline-...-cost} flags as follows. Costs are specified as integers. All of these flags accept a single argument describing such integers using the conventions detailed in section \texttt{19.2.1}.  

\texttt{-inline.alloc-cost}  
The cost of an allocation.  

\texttt{-inline.branch-cost}  
The cost of a branch.  

\texttt{-inline.call-cost}  
The cost of a direct function call.  

\texttt{-inline.indirect-cost}  
The cost of an indirect function call.  

\texttt{-inline.prim-cost}  
The cost of a \textit{primitive}. Primitives encompass operations including arithmetic and memory access.  

(Default values are described in section \texttt{19.5} below.)  

The initial benefit value is then scaled by a factor that attempts to compensate for the fact that the current point in the code, if under some number of conditional branches, may be cold. (Flambda does not currently compute hot and cold paths.) The factor—the estimated probability that the inliner really is on a \textit{hot} path—is calculated as \( (\frac{1}{1+f})^d \), where \( f \) is set by \texttt{-inline.branch.factor} and \( d \) is the nesting depth of branches at the current point. As the inliner descends into more deeply-nested branches, the benefit of inlining thus lessens.  

The resulting benefit value is known as the \textit{estimated benefit}.  

The change in code size is also estimated: morally speaking it should be the change in machine code size, but since that is not available to the inliner, an approximation is used.  

If the estimated benefit exceeds the increase in code size then the inlined version of the function will be kept. Otherwise the function will not be inlined.  

Applications of functors at toplevel will be given an additional benefit (which may be controlled by the \texttt{-inline.lifting-benefit} flag) to bias inlining in such situations towards keeping the inlined version.
19.3.6 Control of speculation

As described above, there are three parameters that restrict the search for inlining opportunities during speculation:

- the *inlining threshold*;
- the *inlining depth*;
- the *unrolling depth*.

These parameters are ultimately bounded by the arguments provided to the corresponding command-line flags (or their default values):

- `-inline` (or, if the call site that triggered speculation is at toplevel, `-inline-toplevel`);
- `-inline-max-depth`;
- `-inline-max-unroll`.

**Note in particular** that `-inline` does not have the meaning that it has in the previous compiler or in `-Oclassic` mode. In both of those situations `-inline` was effectively some kind of basic assessment of inlining benefit. However in Flambda inlining mode it corresponds to a constraint on the search; the assessment of benefit is independent, as described above.

When speculation starts the inlining threshold starts at the value set by `-inline` (or `-inline-toplevel` if appropriate, see above). Upon making a speculative inlining decision the threshold is reduced by the code size of the function being inlined. If the threshold becomes exhausted, at or below zero, no further speculation will be performed.

The inlining depth starts at zero and is increased by one every time the inliner descends into another function. It is then decreased by one every time the inliner leaves such function. If the depth exceeds the value set by `-inline-max-depth` then speculation stops. This parameter is intended as a general backstop for situations where the inlining threshold does not control the search sufficiently.

The unrolling depth applies to calls within the same mutually-recursive group of functions. Each time an inlining of such a call is performed the depth is incremented by one when examining the resulting body. If the depth reaches the limit set by `-inline-max-unroll` then speculation stops.

19.4 Specialisation

The inliner may discover a call site to a recursive function where something is known about the arguments: for example, they may be equal to some other variables currently in scope. In this situation it may be beneficial to *specialise* the function to those arguments. This is done by copying the declaration of the function (and any others involved in any same mutually-recursive declaration) and noting the extra information about the arguments. The arguments augmented by this information are known as *specialised arguments*. In order to try to ensure that specialisation is not performed uselessly, arguments are only specialised if it can be shown that they are *invariant*: in other words, during the execution of the recursive function(s) themselves, the arguments never change.
Unless overridden by an attribute (see below), specialisation of a function will not be attempted if:

- the compiler is in `-Oclassic` mode;
- the function is not obviously recursive;
- the function is not closed.

The compiler can prove invariance of function arguments across multiple functions within a recursive group (although this has some limitations, as shown by the example below).

It should be noted that the *unboxing of closures* pass (see below) can introduce specialised arguments on non-recursive functions. (No other place in the compiler currently does this.)

**Example: the well-known List.iter function**  This function might be written like so:

```ocaml
let rec iter f l =  
  match l with  
  | [] -> ()  
  | h :: t ->  
    f h;  
    iter f t

and used like this:

let print_int x =  
  print_endline (string_of_int x)

let run xs =  
  iter print_int (List.rev xs)
```

The argument `f` to `iter` is invariant so the function may be specialised:

```ocaml
let run xs =  
  let rec iter' f l =  
    (* The compiler knows: f holds the same value as foo throughout iter'. *)  
    match l with  
    | [] -> ()  
    | h :: t ->  
      f h;  
      iter' f t
  in  
  iter' print_int (List.rev xs)
```

The compiler notes down that for the function `iter'`, the argument `f` is specialised to the constant closure `print_int`. This means that the body of `iter'` may be simplified:
let run xs =
  let rec iter' f l =
    (* The compiler knows: f holds the same value as foo throughout iter'. *)
    match l with
    | []  -> ()
    | h :: t ->
      print_int h;  (* this is now a direct call *)
      iter' f t
    in
    iter' print_int (List.rev xs)

  The call to print_int can indeed be inlined:

let run xs =
  let rec iter' f l =
    (* The compiler knows: f holds the same value as foo throughout iter'. *)
    match l with
    | []  -> ()
    | h :: t ->
      print_endline (string_of_int h);
      iter' f t
    in
    iter' print_int (List.rev xs)

  The unused specialised argument f may now be removed, leaving:

let run xs =
  let rec iter' l =
    match l with
    | []  -> ()
    | h :: t ->
      print_endline (string_of_int h);
      iter' t
    in
    iter' (List.rev xs)

**Aside on invariant parameters.** The compiler cannot currently detect invariance in cases such as the following.

let rec iter_swap f g l =
  match l with
  | []  -> ()
  | 0 :: t ->
    iter_swap g f l
  | h :: t ->
    f h;
    iter_swap f g t
19.4.1 Assessment of specialisation benefit

The benefit of specialisation is assessed in a similar way as for inlining. Specialised argument information may mean that the body of the function being specialised can be simplified: the removed operations are accumulated into a benefit. This, together with the size of the duplicated (specialised) function declaration, is then assessed against the size of the call to the original function.

19.5 Default settings of parameters

The default settings (when not using \(-O\text{classic}\)) are for one round of optimisation using the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inline</td>
<td>10</td>
</tr>
<tr>
<td>-inline-branch-factor</td>
<td>0.1</td>
</tr>
<tr>
<td>-inline-alloc-cost</td>
<td>7</td>
</tr>
<tr>
<td>-inline-branch-cost</td>
<td>5</td>
</tr>
<tr>
<td>-inline-call-cost</td>
<td>5</td>
</tr>
<tr>
<td>-inline-indirect-cost</td>
<td>4</td>
</tr>
<tr>
<td>-inline-prim-cost</td>
<td>3</td>
</tr>
<tr>
<td>-inline-lifting-benefit</td>
<td>1300</td>
</tr>
<tr>
<td>-inline-toplevel</td>
<td>160</td>
</tr>
<tr>
<td>-inline-max-depth</td>
<td>1</td>
</tr>
<tr>
<td>-inline-max-unroll</td>
<td>0</td>
</tr>
<tr>
<td>-unbox-closures-factor</td>
<td>10</td>
</tr>
</tbody>
</table>

19.5.1 Settings at -O2 optimisation level

When \(-O2\) is specified two rounds of optimisation are performed. The first round uses the default parameters (see above). The second uses the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inline</td>
<td>25</td>
</tr>
<tr>
<td>-inline-branch-factor</td>
<td>Same as default</td>
</tr>
<tr>
<td>-inline-alloc-cost</td>
<td>Double the default</td>
</tr>
<tr>
<td>-inline-branch-cost</td>
<td>Double the default</td>
</tr>
<tr>
<td>-inline-call-cost</td>
<td>Double the default</td>
</tr>
<tr>
<td>-inline-indirect-cost</td>
<td>Double the default</td>
</tr>
<tr>
<td>-inline-prim-cost</td>
<td>Double the default</td>
</tr>
<tr>
<td>-inline-lifting-benefit</td>
<td>Same as default</td>
</tr>
<tr>
<td>-inline-toplevel</td>
<td>400</td>
</tr>
<tr>
<td>-inline-max-depth</td>
<td>2</td>
</tr>
<tr>
<td>-inline-max-unroll</td>
<td>Same as default</td>
</tr>
<tr>
<td>-unbox-closures-factor</td>
<td>Same as default</td>
</tr>
</tbody>
</table>
19.5.2 Settings at -O3 optimisation level

When -O3 is specified three rounds of optimisation are performed. The first two rounds are as for -O2. The third round uses the following parameters.

<table>
<thead>
<tr>
<th>Parameter</th>
<th>Setting</th>
</tr>
</thead>
<tbody>
<tr>
<td>-inline</td>
<td>50</td>
</tr>
<tr>
<td>-inline-branch-factor</td>
<td>Same as default</td>
</tr>
<tr>
<td>-inline-alloc-cost</td>
<td>Triple the default</td>
</tr>
<tr>
<td>-inline-branch-cost</td>
<td>Triple the default</td>
</tr>
<tr>
<td>-inline-indirect-cost</td>
<td>Triple the default</td>
</tr>
<tr>
<td>-inline-prim-cost</td>
<td>Triple the default</td>
</tr>
<tr>
<td>-inline-lifting-benefit</td>
<td>Same as default</td>
</tr>
<tr>
<td>-inline-toplevel</td>
<td>800</td>
</tr>
<tr>
<td>-inline-max-depth</td>
<td>3</td>
</tr>
<tr>
<td>-inline-max-unroll</td>
<td>1</td>
</tr>
<tr>
<td>-unbox-closures-factor</td>
<td>Same as default</td>
</tr>
</tbody>
</table>

19.6 Manual control of inlining and specialisation

Should the inliner prove recalcitrant and refuse to inline a particular function, or if the observed inlining decisions are not to the programmer’s satisfaction for some other reason, inlining behaviour can be dictated by the programmer directly in the source code. One example where this might be appropriate is when the programmer, but not the compiler, knows that a particular function call is on a cold code path. It might be desirable to prevent inlining of the function so that the code size along the hot path is kept smaller, so as to increase locality.

The inliner is directed using attributes. For non-recursive functions (and one-step unrolling of recursive functions, although @unroll is more clear for this purpose) the following are supported:

@@inline always or @@inline never

Attached to a declaration of a function or functor, these direct the inliner to either always or never inline, irrespective of the size/benefit calculation. (If the function is recursive then the body is substituted and no special action is taken for the recursive call site(s).) @@inline with no argument is equivalent to @@inline always.

@@inlined always or @inlined never

Attached to a function application, these direct the inliner likewise. These attributes at call sites override any other attribute that may be present on the corresponding declaration. @inlined with no argument is equivalent to @inlined always.

For recursive functions the relevant attributes are:

@@specialise always or @@specialise never

Attached to a declaration of a function or functor, this directs the inliner to either always or never specialise the function so long as it has appropriate contextual knowledge, irrespective of the size/benefit calculation. @@specialise with no argument is equivalent to @@specialise always.
@specialised always or @specialised never

Attached to a function application, this directs the inliner likewise. This attribute at a call site overrides any other attribute that may be present on the corresponding declaration. (Note that the function will still only be specialised if there exist one or more invariant parameters whose values are known.) @specialised with no argument is equivalent to @specialised always.

@unrolled $n$

This attribute is attached to a function application and always takes an integer argument. Each time the inliner sees the attribute it behaves as follows:

- If $n$ is zero or less, nothing happens.
- Otherwise the function being called is substituted at the call site with its body having been rewritten such that any recursive calls to that function or any others in the same mutually-recursive group are annotated with the attribute unrolled($n - 1$). Inlining may continue on that body.

As such, $n$ behaves as the “maximum depth of unrolling”.

A compiler warning will be emitted if it was found impossible to obey an annotation from an @inlined or @specialised attribute.

Example showing correct placement of attributes

```ocaml
module F (M : sig type t end) = struct
    let[@inline never] bar x =
        x * 3
    let foo x =
        (bar [@inlined]) (42 + x)
end [@inline never]

module X = F [@inlined] (struct type t = int end)
```

19.7 Simplification

Simplification, which is run in conjunction with inlining, propagates information (known as approximations) about which variables hold what values at runtime. Certain relationships between variables and symbols are also tracked: for example, some variable may be known to always hold the same value as some other variable; or perhaps some variable may be known to always hold the value pointed to by some symbol.

The propagation can help to eliminate allocations in cases such as:

```ocaml
let f x y =
    ...
let p = x, y in
    ...
    ... (fst p) ... (snd p) ...
```
Chapter 19. Optimisation with Flambda

The projections from \( p \) may be replaced by uses of the variables \( x \) and \( y \), potentially meaning that \( p \) becomes unused.

The propagation performed by the simplification pass is also important for discovering which functions flow to indirect call sites. This can enable the transformation of such call sites into direct call sites, which makes them eligible for an inlining transformation.

Note that no information is propagated about the contents of strings, even in safe-string mode, because it cannot yet be guaranteed that they are immutable throughout a given program.

19.8 Other code motion transformations

19.8.1 Lifting of constants

Expressions found to be constant will be lifted to symbol bindings—that is to say, they will be statically allocated in the object file—when they evaluate to boxed values. Such constants may be straightforward numeric constants, such as the floating-point number 42.0, or more complicated values such as constant closures.

Lifting of constants to toplevel reduces allocation at runtime.

The compiler aims to share constants lifted to toplevel such that there are no duplicate definitions. However if .cmx files are hidden from the compiler then maximal sharing may not be possible.

Notes about float arrays The following language semantics apply specifically to constant float arrays. (By “constant float array” is meant an array consisting entirely of floating point numbers that are known at compile time. A common case is a literal such as \([42.0; 43.0;]\).

- Constant float arrays at the toplevel are mutable and never shared. (That is to say, for each such definition there is a distinct symbol in the data section of the object file pointing at the array.)

- Constant float arrays not at toplevel are mutable and are created each time the expression is evaluated. This can be thought of as an operation that takes an immutable array (which in the source code has no associated name; let us call it the initialising array) and duplicates it into a fresh mutable array.
  - If the array is of size four or less, the expression will create a fresh block and write the values into it one by one. There is no reference to the initialising array as a whole.
  - Otherwise, the initialising array is lifted out and subject to the normal constant sharing procedure; creation of the array consists of bulk copying the initialising array into a fresh value on the OCaml heap.

19.8.2 Lifting of toplevel let bindings

Toplevel let-expressions may be lifted to symbol bindings to ensure that the corresponding bound variables are not captured by closures. If the defining expression of a given binding is found to be constant, it is bound as such (the technical term is a let-symbol binding).
Otherwise, the symbol is bound to a (statically-allocated) preallocated block containing one field. At runtime, the defining expression will be evaluated and the first field of the block filled with the resulting value. This initialise-symbol binding causes one extra indirection but ensures, by virtue of the symbol’s address being known at compile time, that uses of the value are not captured by closures.

It should be noted that the blocks corresponding to initialise-symbol bindings are kept alive forever, by virtue of them occurring in a static table of GC roots within the object file. This extended lifetime of expressions may on occasion be surprising. If it is desired to create some non-constant value (for example when writing GC tests) that does not have this extended lifetime, then it may be created and used inside a function, with the application point of that function (perhaps at toplevel)—or indeed the function declaration itself—marked as to never be inlined. This technique prevents lifting of the definition of the value in question (assuming of course that it is not constant).

19.9 Unboxing transformations

The transformations in this section relate to the splitting apart of boxed (that is to say, non-immediate) values. They are largely intended to reduce allocation, which tends to result in a runtime performance profile with lower variance and smaller tails.

19.9.1 Unboxing of closure variables

This transformation is enabled unless -no-unbox-free-vars-of-closures is provided.

Variables that appear in closure environments may themselves be boxed values. As such, they may be split into further closure variables, each of which corresponds to some projection from the original closure variable(s). This transformation is called unboxing of closure variables or unboxing of free variables of closures. It is only applied when there is reasonable certainty that there are no uses of the boxed free variable itself within the corresponding function bodies.

Example: In the following code, the compiler observes that the closure returned from the function \( f \) contains a variable pair (free in the body of \( f \)) that may be split into two separate variables.

```plaintext
let f x0 x1 =
    let pair = x0, x1 in
    Printf.printf "foo\n";
    fun y ->
        fst pair + snd pair + y
```

After some simplification one obtains:

```plaintext
let f x0 x1 =
    let pair_0 = x0 in
    let pair_1 = x1 in
    Printf.printf "foo\n";
    fun y ->
        pair_0 + pair_1 + y
```
and then:

```ml
let f x0 x1 =  
  Printf.printf "foo\n";  
  fun y ->  
    x0 + x1 + y
```

The allocation of the pair has been eliminated.

This transformation does not operate if it would cause the closure to contain more than twice as many closure variables as it did beforehand.

### 19.9.2 Unboxing of specialised arguments

This transformation is enabled unless `-no-unbox-specialised-args` is provided.

It may become the case during compilation that one or more invariant arguments to a function become specialised to a particular value. When such values are themselves boxed the corresponding specialised arguments may be split into more specialised arguments corresponding to the projections out of the boxed value that occur within the function body. This transformation is called *unboxing of specialised arguments*. It is only applied when there is reasonable certainty that the boxed argument itself is unused within the function.

If the function in question is involved in a recursive group then unboxing of specialised arguments may be immediately replicated across the group based on the dataflow between invariant arguments.

**Example:** Having been given the following code, the compiler will inline `loop` into `f`, and then observe `inv` being invariant and always the pair formed by adding 42 and 43 to the argument `x` of the function `f`.

```ml
let rec loop inv xs =  
  match xs with  
  | [] -> fst inv + snd inv  
  | x::xs -> x + loop2 xs inv  
and loop2 ys inv =  
  match ys with  
  | [] -> 4  
  | y::ys -> y - loop inv ys
```

```ml
let f x =  
  Printf.printf "%d\n" (loop (x + 42, x + 43) [1; 2; 3])
```

Since the functions have sufficiently few arguments, more specialised arguments will be added. After some simplification one obtains:

```ml
let f x =  
  let rec loop' xs inv_0 inv_1 =  
    match xs with  
    | [] -> inv_0 + inv_1  
    | x::xs -> x + loop2' xs inv_0 inv_1
```

and loop2' ys inv_0 inv_1 =
  match ys with
  | [] -> 4
  | y::ys -> y - loop' ys inv_0 inv_1

in
Printf.printf "%d\n" (loop' (x + 42) (x + 43) [1; 2; 3])

The allocation of the pair within f has been removed. (Since the two closures for loop' and loop2' are constant they will also be lifted to toplevel with no runtime allocation penalty. This would also happen without having run the transformation to unbox specialise arguments.)

The transformation to unbox specialised arguments never introduces extra allocation.

The transformation will not unbox arguments if it would result in the original function having sufficiently many arguments so as to inhibit tail-call optimisation.

The transformation is implemented by creating a wrapper function that accepts the original arguments. Meanwhile, the original function is renamed and extra arguments are added corresponding to the unboxed specialised arguments; this new function is called from the wrapper. The wrapper will then be inlined at direct call sites. Indeed, all call sites will be direct unless -unbox-closures is being used, since they will have been generated by the compiler when originally specialising the function. (In the case of -unbox-closures other functions may appear with specialised arguments; in this case there may be indirect calls and these will incur a small penalty owing to having to bounce through the wrapper. The technique of direct call surrogates used for -unbox-closures is not used by the transformation to unbox specialised arguments.)

19.9.3 Unboxing of closures

This transformation is not enabled by default. It may be enabled using the -unbox-closures flag.

The transformation replaces closure variables by specialised arguments. The aim is to cause more closures to become closed. It is particularly applicable, as a means of reducing allocation, where the function concerned cannot be inlined or specialised. For example, some non-recursive function might be too large to inline; or some recursive function might offer no opportunities for specialisation perhaps because its only argument is one of type unit.

At present there may be a small penalty in terms of actual runtime performance when this transformation is enabled, although more stable performance may be obtained due to reduced allocation. It is recommended that developers experiment to determine whether the option is beneficial for their code. (It is expected that in the future it will be possible for the performance degradation to be removed.)

Simple example: In the following code (which might typically occur when g is too large to inline) the value of x would usually be communicated to the application of the + function via the closure of g.

let f x =
  let g y =
    x + y
  in
    (g [@inlined never]) 42
Unboxing of the closure causes the value for \( x \) inside \( g \) to be passed as an argument to \( g \) rather than through its closure. This means that the closure of \( g \) becomes constant and may be lifted to toplevel, eliminating the runtime allocation.

The transformation is implemented by adding a new wrapper function in the manner of that used when unboxing specialised arguments. The closure variables are still free in the wrapper, but the intention is that when the wrapper is inlined at direct call sites, the relevant values are passed directly to the main function via the new specialised arguments.

Adding such a wrapper will penalise indirect calls to the function (which might exist in arbitrary places; remember that this transformation is not for example applied only on functions the compiler has produced as a result of specialisation) since such calls will bounce through the wrapper. To mitigate this, if a function is small enough when weighed up against the number of free variables being removed, it will be duplicated by the transformation to obtain two versions: the original (used for indirect calls, since we can do no better) and the wrapper/rewritten function pair as described in the previous paragraph. The wrapper/rewritten function pair will only be used at direct call sites of the function. (The wrapper in this case is known as a \textit{direct call surrogate}, since it takes the place of another function—the unchanged version used for indirect calls—at direct call sites.)

The \texttt{-unbox-closures-factor} command line flag, which takes an integer, may be used to adjust the point at which a function is deemed large enough to be ineligible for duplication. The benefit of duplication is scaled by the integer before being evaluated against the size.

\textbf{Harder example:} In the following code, there are two closure variables that would typically cause closure allocations. One is called \( fv \) and occurs inside the function \( baz \); the other is called \( z \) and occurs inside the function \( bar \). In this toy (yet sophisticated) example we again use an attribute to simulate the typical situation where the first argument of \( baz \) is too large to inline.

\begin{verbatim}
let foo c =
  let rec bar zs fv =
  match zs with
    | [] -> []
    | z::zs ->
    let rec baz f = function
      | [] -> []
      | a::l -> let r = fv + ((f @inlined never]) a) in r :: baz f l
    in
    (map2 (fun y -> z + y) [z; 2; 3; 4]) @ bar zs fv
  in
  Printf.printf "%%d" (List.length (bar [1; 2; 3; 4] c))
\end{verbatim}

The code resulting from applying \texttt{-O3 -unbox-closures} to this code passes the free variables via function arguments in order to eliminate all closure allocation in this example (aside from any that might be performed inside \texttt{printf}).
19.10 Removal of unused code and values

19.10.1 Removal of redundant let expressions

The simplification pass removes unused let bindings so long as their corresponding defining expressions have “no effects”. See the section “Treatment of effects” below for the precise definition of this term.

19.10.2 Removal of redundant program constructs

This transformation is analogous to the removal of let-expressions whose defining expressions have no effects. It operates instead on symbol bindings, removing those that have no effects.

19.10.3 Removal of unused arguments

This transformation is only enabled by default for specialised arguments. It may be enabled for all arguments using the -remove-unused-arguments flag.

The pass analyses functions to determine which arguments are unused. Removal is effected by creating a wrapper function, which will be inlined at every direct call site, that accepts the original arguments and then discards the unused ones before calling the original function. As a consequence, this transformation may be detrimental if the original function is usually indirectly called, since such calls will now bounce through the wrapper. (The technique of direct call surrogates used to reduce this penalty during unboxing of closure variables (see above) does not yet apply to the pass that removes unused arguments.)

19.10.4 Removal of unused closure variables

This transformation performs an analysis across the whole compilation unit to determine whether there exist closure variables that are never used. Such closure variables are then eliminated. (Note that this has to be a whole-unit analysis because a projection of a closure variable from some particular closure may have propagated to an arbitrary location within the code due to inlining.)

19.11 Other code transformations

19.11.1 Transformation of non-escaping references into mutable variables

Flambda performs a simple analysis analogous to that performed elsewhere in the compiler that can transform refs into mutable variables that may then be held in registers (or on the stack as appropriate) rather than being allocated on the OCaml heap. This only happens so long as the reference concerned can be shown to not escape from its defining scope.

19.11.2 Substitution of closure variables for specialised arguments

This transformation discovers closure variables that are known to be equal to specialised arguments. Such closure variables are replaced by the specialised arguments; the closure variables may then be removed by the “removal of unused closure variables” pass (see below).
19.12 Treatment of effects

The Flambda optimisers classify expressions in order to determine whether an expression:

- does not need to be evaluated at all; and/or
- may be duplicated.

This is done by forming judgements on the effects and the coeffects that might be performed were the expression to be executed. Effects talk about how the expression might affect the world; coeffects talk about how the world might affect the expression.

Effects are classified as follows:

**No effects:**

The expression does not change the observable state of the world. For example, it must not write to any mutable storage, call arbitrary external functions or change control flow (e.g. by raising an exception). Note that allocation is *not* classed as having “no effects” (see below).

- It is assumed in the compiler that expressions with no effects, whose results are not used, may be eliminated. (This typically happens where the expression in question is the defining expression of a `let`; in such cases the `let`-expression will be eliminated.) It is further assumed that such expressions with no effects may be duplicated (and thus possibly executed more than once).

- Exceptions arising from allocation points, for example “out of memory” or exceptions propagated from finalizers or signal handlers, are treated as “effects out of the ether” and thus ignored for our determination here of effectfulness. The same goes for floating point operations that may cause hardware traps on some platforms.

**Only generative effects:**

The expression does not change the observable state of the world save for possibly affecting the state of the garbage collector by performing an allocation. Expressions that only have generative effects and whose results are unused may be eliminated by the compiler. However, unlike expressions with “no effects”, such expressions will never be eligible for duplication.

**Arbitrary effects:**

All other expressions.

There is a single classification for coeffects:

**No coeffects:**

The expression does not observe the effects (in the sense described above) of other expressions. For example, it must not read from any mutable storage or call arbitrary external functions.

It is assumed in the compiler that, subject to data dependencies, expressions with neither effects nor coeffects may be reordered with respect to other expressions.
19.13 Compilation of statically-allocated modules

Compilation of modules that are able to be statically allocated (for example, the module corresponding to an entire compilation unit, as opposed to a first class module dependent on values computed at runtime) initially follows the strategy used for bytecode. A sequence of let-bindings, which may be interspersed with arbitrary effects, surrounds a record creation that becomes the module block. The Flambda-specific transformation follows: these bindings are lifted to toplevel symbols, as described above.

19.14 Inhibition of optimisation

Especially when writing benchmarking suites that run non-side-effecting algorithms in loops, it may be found that the optimiser entirely elides the code being benchmarked. This behaviour can be prevented by using the Sysopaque_identity function (which indeed behaves as a normal OCaml function and does not possess any “magic” semantics). The documentation of the Sys module should be consulted for further details.

19.15 Use of unsafe operations

The behaviour of the Flambda simplification pass means that certain unsafe operations, which may without Flambda or when using previous versions of the compiler be safe, must not be used. This specifically refers to functions found in the Obj module.

In particular, it is forbidden to change any value (for example using Obj.set_field or Obj.set_tag) that is not mutable. (Values returned from C stubs are always treated as mutable.) The compiler will emit warning 59 if it detects such a write—but it cannot warn in all cases. Here is an example of code that will trigger the warning:

```ocaml
let f x =
  let a = 42, x in
  (Obj.magic a : int ref) := 1;
  fst a
```

The reason this is unsafe is because the simplification pass believes that fst a holds the value 42; and indeed it must, unless type soundness has been broken via unsafe operations.

If it must be the case that code has to be written that triggers warning 59, but the code is known to actually be correct (for some definition of correct), then Sysopaque_identity may be used to wrap the value before unsafe operations are performed upon it. Great care must be taken when doing this to ensure that the opacity is added at the correct place. It must be emphasised that this use of Sysopaque_identity is only for exceptional cases. It should not be used in normal code or to try to guide the optimiser.

As an example, this code will return the integer 1:

```ocaml
let f x =
  let a = Sysopaque_identity (42, x) in
  (Obj.magic a : int ref) := 1;
  fst a
```
However the following code will still return 42:

```ocaml
let f x = 
  let a = 42, x in 
  Sys.opaque_identity (Obj.magic a : int ref) := 1; 
  fst a
```

High levels of inlining performed by Flambda may expose bugs in code thought previously to be correct. Take care, for example, not to add type annotations that claim some mutable value is always immediate if it might be possible for an unsafe operation to update it to a boxed value.

## 19.16 Glossary

The following terminology is used in this chapter of the manual.

**Call site**

See *direct call site* and *indirect call site* below.

**Closed function**

A function whose body has no free variables except its parameters and any to which are bound other functions within the same (possibly mutually-recursive) declaration.

**Closure**

The runtime representation of a function. This includes pointers to the code of the function together with the values of any variables that are used in the body of the function but actually defined outside of the function, in the enclosing scope. The values of such variables, collectively known as the *environment*, are required because the function may be invoked from a place where the original bindings of such variables are no longer in scope. A group of possibly mutually-recursive functions defined using *let rec* all share a single closure. (Note to developers: in the Flambda source code a *closure* always corresponds to a single function; a *set of closures* refers to a group of such.)

**Closure variable**

A member of the environment held within the closure of a given function.

**Constant**

Some entity (typically an expression) the value of which is known by the compiler at compile time. Constantness may be explicit from the source code or inferred by the Flambda optimisers.

**Constant closure**

A closure that is statically allocated in an object file. It is almost always the case that the environment portion of such a closure is empty.

**Defining expression**

The expression `e in let x = e in e'`. 
Direct call site
A place in a program’s code where a function is called and it is known at compile time which function it will always be.

Indirect call site
A place in a program’s code where a function is called but is not known to be a direct call site.

Program
A collection of symbol bindings forming the definition of a single compilation unit (i.e. .cmx file).

Specialised argument
An argument to a function that is known to always hold a particular value at runtime. These are introduced by the inliner when specialising recursive functions; and the unbox-closures pass. (See section 19.4.)

Symbol
A name referencing a particular place in an object file or executable image. At that particular place will be some constant value. Symbols may be examined using operating system-specific tools (for example objdump on Linux).

Symbol binding
Analogous to a let-expression but working at the level of symbols defined in the object file. The address of a symbol is fixed, but it may be bound to both constant and non-constant expressions.

Toplevel
An expression in the current program which is not enclosed within any function declaration.

Variable
A named entity to which some OCaml value is bound by a let expression, pattern-matching construction, or similar.

19.17 Module Spacetime : Profiling of a program’s space behaviour over time.
Currently only supported on x86-64 platforms running 64-bit code.
To use the functions in this module you must:

- configure the compiler with "-spacetime";
- compile to native code. Without these conditions being satisfied the functions in this module will have no effect.

Instead of manually taking profiling heap snapshots with this module it is possible to use an automatic snapshot facility that writes profiling information at fixed intervals to a file. To enable this, all that needs to be done is to build the relevant program using a compiler configured with
-spacetime; and set the environment variable OCAML_SPACETIME_INTERVAL to an integer number of milliseconds giving the interval between profiling heap snapshots. This interval should not be made excessively small relative to the running time of the program. A typical interval to start with might be 1/100 of the running time of the program. The program must exit ”normally” (i.e. by calling exit, with whatever exit code, rather than being abnormally terminated by a signal) so that the snapshot file is correctly completed.

When using the automatic snapshot mode the profiling output is written to a file called ”spacetime-<pid>” where <pid> is the process ID of the program. (If the program forks and continues executing then multiple files may be produced with different pid numbers.) The profiling output is by default written to the current working directory when the program starts. This may be customised by setting the OCAML_SPACETIME_SNAPSHOT_DIR environment variable to the name of the desired directory.

If using automatic snapshots the presence of the save_event_for_automatic_snapshots function, below, should be noted.

The functions in this module are thread safe.

For functions to decode the information recorded by the profiler, see the Spacetime offline library in otherlibs/.

module Series :
  sig
    type t

    Type representing a file that will hold a series of heap snapshots together with additional information required to interpret those snapshots.

    val create : path:string -> t

    create ~path creates a series file at path.

    val save_event : ?time:float -> t -> event_name:string -> unit

    save_event writes an event, which is an arbitrary string, into the given series file. This may be used for identifying particular points during program execution when analysing the profile. The optional time parameter is as for Snapshot.take.

    val save_and_close : ?time:float -> t -> unit

    save_and_close series writes information into series required for interpreting the snapshots that series contains and then closes the series file. This function must be called to produce a valid series file. The optional time parameter is as for Snapshot.take.

  end

module Snapshot :
  sig

    val take : ?time:float -> Spacetime.Series.t -> unit
take series takes a snapshot of the profiling annotations on the values in the minor and major heaps, together with GC stats, and write the result to the series file. This function triggers a minor GC but does not allocate any memory itself. If the optional time is specified, it will be used instead of the result of Sys.time as the timestamp of the snapshot. Such times should start from zero and be monotonically increasing. This parameter is intended to be used so that snapshots can be correlated against wall clock time (which is not supported in the standard library) rather than elapsed CPU time.

val save_event_for_automatic_snapshots : event_name:string -> unit

Like Series.save_event, but writes to the automatic snapshot file. This function is a no-op if OCAML_SPACETIME_INTERVAL was not set.
Part IV

The OCaml library
Chapter 20

The core library

This chapter describes the OCaml core library, which is composed of declarations for built-in types and exceptions, plus the module \texttt{Pervasives} that provides basic operations on these built-in types. The \texttt{Pervasives} module is special in two ways:

- It is automatically linked with the user’s object code files by the \texttt{ocamlc} command (chapter \ref{chap:compilation}.

- 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 \texttt{Pervasives} module, without adding a \texttt{open Pervasives} directive.

Conventions

The declarations of the built-in types and the components of module \texttt{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.

20.1 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

\begin{verbatim}
type int
  The type of integer numbers.

type char
  The type of characters.

type bytes
  The type of (writable) byte sequences.
\end{verbatim}
The type of (read-only) character strings.

The type of floating-point numbers.

The type of booleans (truth values).

The type of the unit value.

The type of exception values.

The type of arrays whose elements have type 'a.

The type of lists whose elements have type 'a.

The type of optional values of type 'a.

The type of signed 32-bit integers. See the Int32 module.

The type of signed 64-bit integers. See the Int64 module.

The type of signed, platform-native integers (32 bits on 32-bit processors, 64 bits on 64-bit processors). See the Nativeint module.

The type of format strings. 'a is the type of the parameters of the format, 'f is the result type for the printf-style functions, 'b is the type of the first argument given to %a and %t printing functions (see module Printf), 'c is the result type of these functions, and also the type of the argument transmitted to the first argument of kprintf-style functions, 'd is the result type for the scanf-style functions (see module Scanf), and 'e is the type of the receiver function for the scanf-style functions.

This type is used to implement the Lazy module. It should not be used directly.
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. The string gives some information to the programmer. As a general rule, this exception should not be caught, it denotes a programming error and the code should be modified not to trigger it.

exception Failure of string

Exception raised by library functions to signal that they are undefined on the given arguments. The string is meant to give some information to the programmer; you must not pattern match on the string literal because it may change in future versions (use Failure _ instead).

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. The string is meant to give some information to the programmer; you must not pattern match on the string literal because it may change in future versions (use Sys_error _ instead).

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 7.4.) The arguments are the location of the definition in the source code (file name, line number, column number).

20.2 Module Pervasives: The initially opened module.

This module provides the basic operations over the built-in types (numbers, booleans, byte sequences, 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 raise_notrace : exn -> 'a

A faster version raise which does not record the backtrace.

Since: 4.02.0

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.
Chapter 20. The core library

val (<>) : 'a -> 'a -> bool
   Negation of Pervasives.(=) [20.2].

val (<) : 'a -> 'a -> bool
   See Pervasives.(>=) [20.2].

val (>) : 'a -> 'a -> bool
   See Pervasives.(>=) [20.2].

val (<=) : 'a -> 'a -> bool
   See Pervasives.(>=) [20.2].

val (>=) : 'a -> 'a -> bool
   Structural ordering functions. These functions coincide with the usual orderings over integers, characters, strings, byte sequences and floating-point numbers, and extend them to a total ordering over all types. The ordering is compatible with ( = ). As in the case of ( = ), mutable structures are compared by contents. Comparison between functional values raises Invalid_argument. Comparison between cyclic structures may not terminate.

val compare : 'a -> 'a -> int
   compare x y returns 0 if x 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 [20.2]. 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 [21.31] and Map.Make [21.20] functors, as well as the List.sort [21.19] and Array.sort [21.2] 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, byte sequences, 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.(==) [20.2].

Boolean operations
val not : bool -> bool
    The boolean negation.
val (&&) : bool -> bool -> bool
    The boolean 'and'. Evaluation is sequential, left-to-right: in e1 && e2, e1 is evaluated first, and if it returns false, e2 is not evaluated at all.
val (&) : bool -> bool -> bool
    Deprecated. Pervasives.(&&) [20.2] should be used instead.
val (||) : bool -> bool -> bool
    The boolean 'or'. Evaluation is sequential, left-to-right: in e1 || e2, e1 is evaluated first, and if it returns true, e2 is not evaluated at all.
val (or) : bool -> bool -> bool
    Deprecated. Pervasives.(||) [20.2] should be used instead.

Debugging
val __LOC__ : string
    __LOC__ returns the location at which this expression appears in the file currently being parsed by the compiler, with the standard error format of OCaml: "File %S, line %d, characters %d-%d".
    Since: 4.02.0
val __FILE__ : string
    __FILE__ returns the name of the file currently being parsed by the compiler.
    Since: 4.02.0
val __LINE__ : int
__LINE__ returns the line number at which this expression appears in the file currently being parsed by the compiler.

Since: 4.02.0

val __MODULE__ : string

__MODULE__ returns the module name of the file being parsed by the compiler.

Since: 4.02.0

val __POS__ : string * int * int * int

__POS__ returns a tuple (file,lnum,cnum,enum), corresponding to the location at which this expression appears in the file currently being parsed by the compiler. file is the current filename, lnum the line number, cnum the character position in the line and enum the last character position in the line.

Since: 4.02.0

val __LOC_OF__ : 'a -> string * a

__LOC_OF__ expr returns a pair (loc, expr) where loc is the location of expr in the file currently being parsed by the compiler, with the standard error format of OCaml: "File %S, line %d, characters %d-%d".

Since: 4.02.0

val __LINE_OF__ : 'a -> int * a

__LINE__ expr returns a pair (line, expr), where line is the line number at which the expression expr appears in the file currently being parsed by the compiler.

Since: 4.02.0

val __POS_OF__ : 'a -> (string * int * int * int) * a

__POS_OF__ expr returns a pair (loc,expr), where loc is a tuple (file,lnum,cnum,enum) corresponding to the location at which the expression expr appears in the file currently being parsed by the compiler. file is the current filename, lnum the line number, cnum the character position in the line and enum the last character position in the line.

Since: 4.02.0

Composition operators

val (|>) : 'a -> ('a -> 'b) -> 'b

Reverse-application operator: x |> f |> g is exactly equivalent to g (f (x)).

Since: 4.01

val (@@) : ('a -> 'b) -> 'a -> 'b

Application operator: g @@ f @@ x is exactly equivalent to g (f (x)).

Since: 4.01
**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.

```ocaml
val (~-) : int -> int
  Unary negation. You can also write - e instead of ~- e.

val (~+) : int -> int
  Unary addition. You can also write + e instead of ~+ e.

val succ : int -> int
  succ x is x + 1.

val pred : int -> int
  pred x is x - 1.

val (+) : int -> int -> int
  Integer addition.

val (-) : int -> int -> int
  Integer subtraction.

val ( * ) : int -> int -> int
  Integer multiplication.

val (/) : int -> int -> int
  Integer division. Raise Division_by_zero if the second argument is 0. Integer division rounds the real quotient of its arguments towards zero. More precisely, if $x \geq 0$ and $y > 0$, $x / y$ is the greatest integer less than or equal to the real quotient of $x$ by $y$. 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 $\text{abs}(x \mod y) \leq \text{abs}(y) - 1$. 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.


**Chapter 20. The core library**

**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 \( m \geq \) 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 \( m \geq \) 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 \( m \geq \) bitsize.

**Floating-point arithmetic**

OCaml’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 (\(\cdot\)) : float \to float \to float
Floating-point subtraction

val (\(*\)) : float \to float \to float
Floating-point multiplication

val (\(/\)) : float \to float \to float
Floating-point division.

val (\(**\)) : float \to float \to float
Exponentiation.

val sqrt : float \to float
Square root.

val exp : float \to float
Exponential.

val log : float \to float
Natural logarithm.

val log10 : float \to float
Base 10 logarithm.

val expm1 : float \to float
\(\text{expm1} x\) computes \(\text{exp} x - 1.0\), giving numerically-accurate results even if \(x\) is close to 0.0.

Since: 3.12.0

val log1p : float \to float
\(\text{log1p} x\) computes \(\text{log}(1.0 + x)\) (natural logarithm), giving numerically-accurate results even if \(x\) is close to 0.0.

Since: 3.12.0

val cos : float \to float
Cosine. Argument is in radians.

val sin : float \to float
Sine. Argument is in radians.

val tan : float \to float
Tangent. Argument is in radians.

val acos : float \to 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 asin : float \(\rightarrow\) 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 atan : float \(\rightarrow\) float
Arc tangent. Result is in radians and is between \(-\pi/2\) and \(\pi/2\).

val atan2 : float \(\rightarrow\) float \(\rightarrow\) float
\(atan2\ y\ x\) returns the arc tangent of \(y \div x\). 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 hypot : float \(\rightarrow\) float \(\rightarrow\) float
\(hypot\ x\ y\) returns \(\sqrt{x \cdot x + y \cdot y}\), that is, the length of the hypotenuse of a right-angled triangle with sides of length \(x\) and \(y\), or, equivalently, the distance of the point \((x,y)\) to origin. If one of \(x\) or \(y\) is infinite, returns \texttt{infinity} even if the other is \texttt{nan}.

Since: 4.00.0

val cosh : float \(\rightarrow\) float
Hyperbolic cosine. Argument is in radians.

val sinh : float \(\rightarrow\) float
Hyperbolic sine. Argument is in radians.

val tanh : float \(\rightarrow\) float
Hyperbolic tangent. Argument is in radians.

val ceil : float \(\rightarrow\) float
Round above to an integer value. \texttt{ceil\ f\} returns the least integer value greater than or equal to \(f\). The result is returned as a float.

val floor : float \(\rightarrow\) float
Round below to an integer value. \texttt{floor\ f\} returns the greatest integer value less than or equal to \(f\). The result is returned as a float.

val abs_float : float \(\rightarrow\) float
\texttt{abs_float\ f\} returns the absolute value of \(f\).

val copysign : float \(\rightarrow\) float \(\rightarrow\) float
\texttt{copysign\ x\ y\} returns a float whose absolute value is that of \(x\) and whose sign is that of \(y\). If \(x\) is \texttt{nan}, returns \texttt{nan}. If \(y\) is \texttt{nan}, returns either \(x\) or \(-x\), but it is not specified which.

Since: 4.00.0
val mod_float : float -> float -> float

mod_float a b returns the remainder of a with respect to b. The returned value is a -. n *. b, where n is the quotient a /. b rounded towards zero to an integer.

val frexp : float -> float * int

frexp f returns the pair of the significant and the exponent of f. When f is zero, the significant x and the exponent n of f are equal to zero. When f is non-zero, they are defined by f = x *. 2 ** n and 0.5 <= x < 1.0.

val ldexp : float -> int -> float

ldexp x n returns x *. 2 ** n.

val modf : float -> float * float

modf f returns the pair of the fractional and integral part of f.

val float : int -> float

Same as Pervasives.float_of_int[20.2].

val float_of_int : int -> float

Convert an integer to floating-point.

val truncate : float -> int

Same as Pervasives.int_of_float[20.2].

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.

val infinity : float

Positive infinity.

val neg_infinity : float

Negative infinity.

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 nan as argument returns nan as result. As for floating-point comparisons, =, <, <=, > and >= return false and <> returns true if one or both of their arguments is nan.

val max_float : float

The largest positive finite value of type float.

val min_float : float
Chapter 20. The core library

The smallest positive, non-zero, non-denormalized value of type `float`.

```ocaml
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

The five classes of floating-point numbers, as determined by the `Pervasives.classify_float` function.

```ocaml
val classify_float : float -> fpclass
  Return the class of the given floating-point number: normal, subnormal, zero, infinite, or not a number.
```

String operations

More string operations are provided in module `String`. [21.37]

```ocaml
val (^) : string -> string -> string
  String concatenation.
```

Character operations

More character operations are provided in module `Char`. [21.6]

```ocaml
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. As the returned values may be shared, the user should not modify them directly.

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), in hexadecimal (if it begins with 0x or 0X), in octal (if it begins with 0o or 0O), or in binary (if it begins with 0b or 0B). The _ (underscore) character can appear anywhere in the string and is ignored. 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. The string is read in decimal (by default) or in hexadecimal (marked by 0x or 0X). The format of decimal floating-point numbers is [-] dd.ddd (e|E) [+/-] dd, where d stands for a decimal digit. The format of hexadecimal floating-point numbers is [-] 0(x|X) hh.hhh (p|P) [+/-] dd, where h stands for a hexadecimal digit and d for a decimal digit. In both cases, at least one of the integer and fractional parts must be given; the exponent part is optional. The _ (underscore) character can appear anywhere in the string and is ignored. Depending on the execution platforms, other representations of floating-point numbers can be accepted, but should not be relied upon. Raise Failure "float_of_string" if the given string is not a valid representation of a float.
Chapter 20. The core library

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[21.19].

val (@) : 'a list -> 'a list -> 'a list
   List concatenation. Not tail-recursive (length of the first argument).

Input/output

Note: all input/output functions can raise Sys_error when the system calls they invoke fail.

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_bytes : bytes -> unit
   Print a byte sequence on standard output.

Since: 4.02.0
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_bytes : bytes -> unit
    Print a byte sequence on standard error.
    Since: 4.02.0

val prerr_int : int -> unit
    Print an integer, in decimal, on standard error.

val prerr_float : float -> unit
    Print a floating-point number, in decimal, on standard error.

val prerr_endline : string -> unit
    Print a string, followed by a newline character on standard error and flush standard error.

val prerr_newline : unit -> unit
    Print a newline character on standard error, and flush standard error.
Input functions on standard input

val read_line : unit -> string
   Flush standard output, then read characters from standard input until a newline character
   is encountered. Return the string of all characters read, without the newline character at
   the end.

val read_int : unit -> int
   Flush standard output, then read one line from standard input and convert it to an integer.
   Raise Failure "int_of_string" if the line read is not a valid representation of an integer.

val read_float : unit -> float
   Flush standard output, then read one line from standard input and convert it to a
   floating-point number. The result is unspecified if the line read is not a valid representation
   of a floating-point number.

General output functions

type open_flag =
    — Open_rdonly
       open 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[20.2] and Pervasives.open_in_gen[20.2].

val open_out : string -> out_channel
   Open the named file for writing, and return a new output channel on that file, positioned 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.

val open_out_bin : string -> out_channel
Same as `Pervasives.open_out`\[20.2\], 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`\[20.2\].

```ocaml
def 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`\[20.2\] and `Pervasives.open_out_bin`\[20.2\] are special cases of this function.
```

```ocaml
def 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.
```

```ocaml
def val flush_all : unit -> unit

Flush all open output channels; ignore errors.
```

```ocaml
def val output_char : out_channel -> char -> unit

Write the character on the given output channel.
```

```ocaml
def val output_string : out_channel -> string -> unit

Write the string on the given output channel.
```

```ocaml
def val output_bytes : out_channel -> bytes -> unit

Write the byte sequence on the given output channel.

Since: 4.02.0
```

```ocaml
def val output : out_channel -> bytes -> int -> int -> unit

output oc buf pos len writes len characters from byte sequence buf, starting at offset pos, to the given output channel oc. Raise Invalid_argument "output" if pos and len do not designate a valid range of buf.
```

```ocaml
def val output_substring : out_channel -> string -> int -> int -> unit

Same as `output` but take a string as argument instead of a byte sequence.

Since: 4.02.0
```

```ocaml
def 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.
```

```ocaml
def 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 \texttt{Pervasives.input\_binary\_int} function. The format is compatible across all machines for a given version of OCaml.

\begin{verbatim}
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 \texttt{Pervasives.input_value}. See the description of module \texttt{Marshal} for more information. \texttt{Pervasives.output_value} is equivalent to \texttt{Marshal.to\_channel} with an empty list of flags.

val seek_out : out_channel -> int -> unit

\texttt{seek\_out} \texttt{chan pos} sets the current writing position to \texttt{pos} for channel \texttt{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 \texttt{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.

val close_out : out_channel -> unit

Close the given channel, flushing all buffered write operations. Output functions raise a \texttt{Sys\_error} exception when they are applied to a closed output channel, except \texttt{close\_out} and \texttt{flush}, which do nothing when applied to an already closed channel. Note that \texttt{close\_out} may raise \texttt{Sys\_error} if the operating system signals an error when flushing or closing.

val close_out\_noerr : out_channel -> unit

Same as \texttt{close\_out}, but ignore all errors.

val set_binary_mode_out : out_channel -> bool -> unit

\texttt{set\_binary\_mode\_out oc true} sets the channel \texttt{oc} to binary mode: no translations take place during output. \texttt{set\_binary\_mode\_out oc false} sets the channel \texttt{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 \texttt{\n} to \texttt{\r\n}. This function has no effect under operating systems that do not distinguish between text mode and binary mode.
General input functions

val open_in : string -> in_channel

Open the named file for reading, and return a new input channel on that file, positioned at the beginning of the file.

val open_in_bin : string -> in_channel

Same as Pervasives.open_in, but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like Pervasives.open_in.

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 and Pervasives.open_in_bin are special cases of this function.

val input_char : in_channel -> char

Read one character from the given input channel. Raise End_of_file if there are no more characters to read.

val input_line : in_channel -> string

Read characters from the given input channel, until a newline character is encountered. Return the string of all characters read, without the newline character at the end. Raise End_of_file if the end of the file is reached at the beginning of line.

val input : in_channel -> bytes -> int -> int -> int

input ic buf pos len reads up to len characters from the given channel ic, storing them in byte sequence 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 range of buf.

val really_input : in_channel -> bytes -> int -> int -> unit

really_input ic buf pos len reads len characters from channel ic, storing them in byte sequence 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 range of buf.

val really_input_string : in_channel -> int -> string
really_input_string ic len reads len characters from channel ic and returns them in a new string. Raise End_of_file if the end of file is reached before len characters have been read.

Since: 4.02.0

val input_byte : in_channel -> int
Same as Pervasives.input_char[20.2], 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[20.2]. Raise End_of_file if an end of file was reached while reading the integer.

val input_value : in_channel -> 'a
Read the representation of a structured value, as produced by Pervasives.output_value[20.2], and return the corresponding value. This function is identical to Marshal.from_channel[21.21]; see the description of module Marshal[21.21] for more information, in particular concerning the lack of type safety.

val seek_in : in_channel -> int -> unit
seek_in chan pos sets the current reading position to pos for channel chan. This works only for regular files. On files of other kinds, the behavior is unspecified.

val pos_in : in_channel -> int
Return the current reading position for the given channel.

val in_channel_length : in_channel -> int
Return the 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.

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 ; }

  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`.

```ocaml
val decr : int ref -> unit
Decrement the integer contained in the given reference. Equivalent to `fun r -> r := pred !r`.
```

### Result type

```ocaml
type ('a, 'b) result =
 | Ok of 'a
 | Error of 'b
```

### Operations on format strings

Format strings are character strings with special lexical conventions that defines the functionality of formatted input/output functions. Format strings are used to read data with formatted input functions from module `Scanf` and to print data with formatted output functions from modules `Printf` and `Format`.

Format strings have a general and highly polymorphic type `('a, 'b, 'c, 'd, 'e, 'f) format6`. The two simplified types, `format` and `format4` below are included for backward compatibility with earlier releases of OCaml.

The meaning of format string type parameters is as follows:

- `'a` is the type of the parameters of the format for formatted output functions (printf-style functions); `'a` is the type of the values read by the format for formatted input functions (scanf-style functions).

- `'b` is the type of input source for formatted input functions and the type of output target for formatted output functions. For printf-style functions from module `Printf`, `'b` is typically `out_channel`; for printf-style functions from module `Format`, `'b` is
typically `Format.formatter`; for `scanf`-style functions from module `Scanf`, `'b` is typically `Scanf.Scanning.in_channel`.

Type argument `'b` is also the type of the first argument given to user’s defined printing functions for `%a` and `%t` conversions, and user’s defined reading functions for `%r` conversion.

- `'c` is the type of the result of the `%a` and `%t` printing functions, and also the type of the argument transmitted to the first argument of `kprintf`-style functions or to the `kscanf`-style functions.
- `'d` is the type of parameters for the `scanf`-style functions.
- `'e` is the type of the receiver function for the `scanf`-style functions.
- `'f` is the final result type of a formatted input/output function invocation: for the `printf`-style functions, it is typically `unit`; for the `scanf`-style functions, it is typically the result type of the receiver function.

```ocaml
type ('a, 'b, 'c, 'd, 'e, 'f) format6 = ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6

val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string

val format_of_string : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6

val (^^) : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('f, 'b, 'c, 'e, 'g, 'h) format6 -> ('a, 'b, 'c, 'd, 'g, 'h) format6

f1 ^^ f2 catenates format strings f1 and f2. The result is a format string that behaves as the concatenation of format strings f1 and f2: in case of formatted output, it accepts arguments from f1, then arguments from f2; in case of formatted input, it returns results from f1, then results from f2.
```

**Program termination**

```ocaml
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.

```ocaml
def 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`, 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 21

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>String</td>
<td>502</td>
<td>string operations</td>
</tr>
<tr>
<td>Bytes</td>
<td>389</td>
<td>operations on byte sequences</td>
</tr>
<tr>
<td>Array</td>
<td>382</td>
<td>array operations</td>
</tr>
<tr>
<td>List</td>
<td>449</td>
<td>list operations</td>
</tr>
<tr>
<td>StdLabels</td>
<td>500</td>
<td>labeled versions of the above 4 modules</td>
</tr>
<tr>
<td>Char</td>
<td>396</td>
<td>character operations</td>
</tr>
<tr>
<td>Uchar</td>
<td>513</td>
<td>Unicode characters</td>
</tr>
<tr>
<td>Sort</td>
<td>496</td>
<td>(deprecated)</td>
</tr>
<tr>
<td>Hashtbl</td>
<td>432</td>
<td>hash tables and hash functions</td>
</tr>
<tr>
<td>Random</td>
<td>480</td>
<td>pseudo-random number generator</td>
</tr>
<tr>
<td>Set</td>
<td>492</td>
<td>sets over ordered types</td>
</tr>
<tr>
<td>Map</td>
<td>454</td>
<td>association tables over ordered types</td>
</tr>
<tr>
<td>MoreLabels</td>
<td>461</td>
<td>labeled versions of Hashtbl, Set, and Map</td>
</tr>
<tr>
<td>Oo</td>
<td>469</td>
<td>useful functions on objects</td>
</tr>
<tr>
<td>Stack</td>
<td>499</td>
<td>last-in first-out stacks</td>
</tr>
<tr>
<td>Queue</td>
<td>479</td>
<td>first-in first-out queues</td>
</tr>
<tr>
<td>Buffer</td>
<td>386</td>
<td>buffers that grow on demand</td>
</tr>
<tr>
<td>Lazy</td>
<td>445</td>
<td>delayed evaluation</td>
</tr>
<tr>
<td>Weak</td>
<td>514</td>
<td>references that don’t prevent objects from being garbage-collected</td>
</tr>
<tr>
<td>Ephemeron</td>
<td>400</td>
<td>ephemeron and weak hash tables</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>397</td>
<td>Complex numbers</td>
</tr>
<tr>
<td>Int32</td>
<td>439</td>
<td>operations on 32-bit integers</td>
</tr>
<tr>
<td>Int64</td>
<td>442</td>
<td>operations on 64-bit integers</td>
</tr>
<tr>
<td>Nativeint</td>
<td>466</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>410</td>
<td>pretty printing with automatic indentation and line breaking</td>
</tr>
<tr>
<td>Marshal</td>
<td>459</td>
<td>marshaling of data structures</td>
</tr>
<tr>
<td>Printf</td>
<td>476</td>
<td>formatting printing functions</td>
</tr>
<tr>
<td>Scanf</td>
<td>483</td>
<td>formatted input functions</td>
</tr>
<tr>
<td>Digest</td>
<td>398</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>431</td>
<td>a generic lexer over streams</td>
</tr>
<tr>
<td>Lexing</td>
<td>447</td>
<td>the run-time library for lexers generated by ocamlllex</td>
</tr>
<tr>
<td>Parsing</td>
<td>470</td>
<td>the run-time library for parsers generated by ocamlyacc</td>
</tr>
<tr>
<td>Stream</td>
<td>500</td>
<td>basic functions over streams</td>
</tr>
</tbody>
</table>
Chapter 21. The standard library

System interface:

<table>
<thead>
<tr>
<th>Module</th>
<th>Page</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>Arg</td>
<td>379</td>
<td>parsing of command line arguments</td>
</tr>
<tr>
<td>Callback</td>
<td>395</td>
<td>registering OCaml functions to be called from C</td>
</tr>
<tr>
<td>Filename</td>
<td>408</td>
<td>operations on file names</td>
</tr>
<tr>
<td>Gc</td>
<td>425</td>
<td>memory management control and statistics</td>
</tr>
<tr>
<td>Printexc</td>
<td>471</td>
<td>a catch-all exception handler</td>
</tr>
<tr>
<td>Spacetime</td>
<td>497</td>
<td>memory profiler</td>
</tr>
<tr>
<td>Sys</td>
<td>506</td>
<td>system interface</td>
</tr>
</tbody>
</table>

21.1 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. For compatibility with GNU getopt_long, keyword=arg is also allowed. Arguments not preceded by a keyword are called anonymous arguments.

Examples (cmd is assumed to be the command name):

- cmd -flag (a unit option)
- cmd -int 1 (an int option with argument 1)
- cmd -string foobar (a string option with argument "foobar")
- cmd -float 12.34 (a float option with argument 12.34)
- cmd a b c (three anonymous arguments: "a", "b", and "c")
- cmd a b -- c d (two anonymous arguments and a rest option with two arguments)

type spec =
<table>
<thead>
<tr>
<th>Unit of (unit -&gt; unit)</th>
</tr>
</thead>
<tbody>
<tr>
<td>Call the function with unit argument</td>
</tr>
<tr>
<td>Bool of (bool -&gt; unit)</td>
</tr>
<tr>
<td>Call the function with a bool argument</td>
</tr>
<tr>
<td>Set of bool Pervasives.ref</td>
</tr>
<tr>
<td>Set the reference to true</td>
</tr>
<tr>
<td>Clear of bool Pervasives.ref</td>
</tr>
<tr>
<td>Set the reference to false</td>
</tr>
</tbody>
</table>
| 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. Beware: options 
  that have an empty doc string will not be included in the list.
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_dynamic :
(key * spec * doc) list Pervasives.ref ->
anon_fun -> usage_msg -> unit

Same as Arg.parse[21.1], except that the speclist argument is a reference and may be updated during the parsing. A typical use for this feature is to parse command lines of the form:

- command subcommand options where the list of options depends on the value of the subcommand argument.

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.

val parse_argv_dynamic :
?current:int Pervasives.ref ->
string array ->
(key * spec * doc) list Pervasives.ref ->
anon_fun -> string -> unit

Same as Arg.parse_argv[21.1], except that the speclist argument is a reference and may be updated during the parsing. See Arg.parse_dynamic[21.1].

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 :
  ?limit:int ->
  (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.

21.2 Module Array : Array operations.

val length : 'a array -> int
Return the length (number of elements) of the given array.

val get : 'a array -> int -> 'a
Array.get a n returns the element number n of array a. The first element has number 0. The last element has number Array.length a - 1. 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).

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.

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.

val create : int -> 'a -> 'a array

Deprecated. Array.create is an alias for Array.make.21.2.

val create_float : int -> float array

Array.create_float n returns a fresh float array of length n, with uninitialized data.

Since: 4.03

val make_float : int -> float array

Deprecated. Array.make_float is an alias for Array.create_float.21.2.

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.

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.

val create_matrix : int -> int -> 'a -> 'a array array

Deprecated. Array.create_matrix is an alias for Array.make_matrix.21.2.

val append : 'a array -> 'a array -> 'a array

Array.append v1 v2 returns a fresh array containing the concatenation of the arrays v1 and v2.

val concat : 'a array list -> 'a array

Same as Array.append, but concatenates a list of arrays.

val sub : 'a array -> int -> int -> 'a array

A two-dimensional array of elements stored in row-major order (i.e., the ith row is stored before the (i + 1)st row).
Array.sub a start len returns a fresh array of length len, containing the elements number start to start + len - 1 of array a.

Raise Invalid_argument "Array.sub" if start and len do not designate a valid subarray of a; that is, if start < 0, or len < 0, or start + len > Array.length a.

val copy : 'a array -> 'a array
    Array.copy a returns a copy of a, that is, a fresh array containing the same elements as a.

val fill : 'a array -> int -> int -> 'a -> unit
    Array.fill a ofs len x modifies the array a in place, storing x in elements number ofs to ofs + len - 1.
    Raise Invalid_argument "Array.fill" if ofs and len do not designate a valid subarray of a.

val blit : 'a array -> int -> 'a array -> int -> int -> unit
    Array.blit v1 o1 v2 o2 len copies len elements from array v1, starting at element number o1, to array v2, starting at element number o2. It works correctly even if v1 and v2 are the same array, and the source and destination chunks overlap.
    Raise Invalid_argument "Array.blit" if o1 and len do not designate a valid subarray of v1, or if o2 and len do not designate a valid subarray of v2.

val to_list : 'a array -> 'a list
    Array.to_list a returns the list of all the elements of a.

val of_list : 'a list -> 'a array
    Array.of_list l returns a fresh array containing the elements of l.

Iterators

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 iteri : (int -> 'a -> unit) -> 'a array -> unit
    Same as Array.iter [21.2], but the function is applied with the index of the element as first argument, and the element itself as second argument.

val map : ('a -> 'b) -> 'a array -> 'b array
    Array.map f a applies function f to all the elements of a, and builds an array with the results returned by f: [| f a.(0); f a.(1); ...; f a.(Array.length a - 1) |].

val mapi : (int -> 'a -> 'b) -> 'a array -> 'b array
Same as `Array.map`[21.2], but the function is applied to the index of the element as first argument, and the element itself as second argument.

```ocaml
val fold_left : ('a -> 'b -> 'a) -> 'a 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.
```

```ocaml
val fold_right : ('b -> 'a -> 'a) -> 'b array -> 'a -> 'a
  Array.fold_right f a x computes f a.(0) (f a.(1) ( ... (f a.(n-1) x) ...)),
  where n is the length of the array a.
```

### Iterators on two arrays

```ocaml
val iter2 : ('a -> 'b -> unit) -> 'a array -> 'b array -> unit
  Array.iter2 f a b applies function f to all the elements of a and b. Raise
  Invalid_argument if the arrays are not the same size.
  Since: 4.03.0
```

```ocaml
val map2 : ('a -> 'b -> 'c) -> 'a array -> 'b array -> 'c array
  Array.map2 f a b applies function f to all the elements of a and b, and builds an array
  with the results returned by f: [| f a.(0) b.(0); ...; f a.(Array.length a - 1)
  b.(Array.length b - 1)|]. Raise Invalid_argument if the arrays are not the same size.
  Since: 4.03.0
```

### Array scanning

```ocaml
val for_all : ('a -> bool) -> 'a array -> bool
  Array.for_all p [|a1; ...; an|] checks if all elements of the array satisfy the predicate
  p. That is, it returns (p a1) && (p a2) && ... && (p an).
  Since: 4.03.0
```

```ocaml
val exists : ('a -> bool) -> 'a array -> bool
  Array.exists p [|a1; ...; an|] checks if at least one element of the array satisfies the
  predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).
  Since: 4.03.0
```

```ocaml
val mem : 'a -> 'a array -> bool
  mem a l is true if and only if a is equal to an element of l.
  Since: 4.03.0
```

```ocaml
val memq : 'a -> 'a array -> bool
  Same as Array.mem[21.2], but uses physical equality instead of structural equality to
  compare array elements.
  Since: 4.03.0
```
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.

The current implementation uses Heap Sort. It runs in constant 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`.

val fast_sort : ('a -> 'a -> int) -> 'a array -> unit

Same as `Array.sort` or `Array.stable_sort`, whichever is faster on typical input.

21.3 Module Buffer : Extensible buffers.

This module implements buffers that automatically expand as necessary. It provides accumulative concatenation of strings in quasi-linear time (instead of quadratic time when strings are concatenated pairwise).

type t

The abstract type of buffers.

val create : int -> t
create n returns a fresh buffer, initially empty. The n parameter is the initial size of the internal byte sequence that holds the buffer contents. That byte sequence 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, 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 to_bytes : t -> bytes
Return a copy of the current contents of the buffer. The buffer itself is unchanged.

Since: 4.02

val sub : t -> int -> int -> string
Buffer.sub b off len returns a copy of len bytes from the current contents of the buffer b, starting at offset off.
Raise Invalid_argument if srcoff and len do not designate a valid range of b.

val blit : t -> int -> bytes -> 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 dst, starting at character dstoff.
Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if dstoff and len do not designate a valid range 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 byte sequence holding the buffer contents, replacing it with the initial internal byte sequence 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 buffer b.

val add_string : t -> string -> unit
add_string b s appends the string s at the end of buffer b.

val add_bytes : t -> bytes -> unit
add_bytes b s appends the byte sequence s at the end of buffer b.
Since: 4.02

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 buffer b.

val add_subbytes : t -> bytes -> int -> int -> unit
add_subbytes b s ofs len takes len characters from offset ofs in byte sequence s and appends them at the end of buffer b.
Since: 4.02

val add_substitute : t -> (string -> string) -> string -> unit
add_substitute b f s appends the string pattern s at the end of 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 at most n characters 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. In this case, the characters are still added to the buffer, so as to avoid loss of data.

val output_buffer : Pervasives.out_channel -> t -> unit
output_buffer oc b writes the current contents of buffer b on the output channel oc.
21.4 Module Bytes: Byte sequence operations.

A byte sequence is a mutable data structure that contains a fixed-length sequence of bytes. Each byte can be indexed in constant time for reading or writing.

Given a byte sequence \( s \) of length \( l \), we can access each of the \( l \) bytes of \( s \) via its index in the sequence. Indexes start at 0, and we will call an index valid in \( s \) if it falls within the range \([0...1-1]\) (inclusive). A position is the point between two bytes or at the beginning or end of the sequence. We call a position valid in \( s \) if it falls within the range \([0...1]\) (inclusive). Note that the byte at index \( n \) is between positions \( n \) and \( n+1 \).

Two parameters \( \texttt{start} \) and \( \texttt{len} \) are said to designate a valid range of \( s \) if \( \texttt{len} \geq 0 \) and \( \texttt{start} \) and \( \texttt{start+len} \) are valid positions in \( s \).

Byte sequences can be modified in place, for instance via the \texttt{set} and \texttt{blit} functions described below. See also strings (module \texttt{String}\[21.37\]), which are almost the same data structure, but cannot be modified in place.

Bytes are represented by the OCaml type \texttt{char}.

Since: 4.02.0

<table>
<thead>
<tr>
<th>Function</th>
<th>Description</th>
</tr>
</thead>
<tbody>
<tr>
<td>\texttt{val length : bytes -&gt; int}</td>
<td>Returns the length (number of bytes) of the argument.</td>
</tr>
<tr>
<td>\texttt{val get : bytes -&gt; int -&gt; char}</td>
<td>\texttt{get s n} returns the byte at index ( n ) in argument ( s ). Raise \texttt{Invalid_argument} if ( n ) not a valid index in ( s ).</td>
</tr>
<tr>
<td>\texttt{val set : bytes -&gt; int -&gt; char -&gt; unit}</td>
<td>\texttt{set s n c} modifies ( s ) in place, replacing the byte at index ( n ) with ( c ). Raise \texttt{Invalid_argument} if ( n ) is not a valid index in ( s ).</td>
</tr>
<tr>
<td>\texttt{val create : int -&gt; bytes}</td>
<td>\texttt{create n} returns a new byte sequence of length ( n ). The sequence is uninitialized and contains arbitrary bytes. Raise \texttt{Invalid_argument} if ( n &lt; 0 ) or ( n &gt; \text{Sys.max_string_length}[21.38].</td>
</tr>
<tr>
<td>\texttt{val make : int -&gt; char -&gt; bytes}</td>
<td>\texttt{make n c} returns a new byte sequence of length ( n ), filled with the byte ( c ). Raise \texttt{Invalid_argument} if ( n &lt; 0 ) or ( n &gt; \text{Sys.max_string_length}[21.38].</td>
</tr>
<tr>
<td>\texttt{val init : int -&gt; (int -&gt; char) -&gt; bytes}</td>
<td>\texttt{Bytes.init n f} returns a fresh byte sequence of length ( n ), with character ( i ) initialized to the result of ( f \ i ) (in increasing index order). Raise \texttt{Invalid_argument} if ( n &lt; 0 ) or ( n &gt; \text{Sys.max_string_length}[21.38].</td>
</tr>
<tr>
<td>\texttt{val empty : bytes}</td>
<td></td>
</tr>
</tbody>
</table>


A byte sequence of size 0.

```ocaml
val copy : bytes -> bytes
Return a new byte sequence that contains the same bytes as the argument.

val of_string : string -> bytes
Return a new byte sequence that contains the same bytes as the given string.

val to_string : bytes -> string
Return a new string that contains the same bytes as the given byte sequence.

val sub : bytes -> int -> int -> bytes
    sub s start len
    returns a new byte sequence of length len, containing the subsequence of s
    that starts at position start and has length len.
    Raise Invalid_argument if start and len do not designate a valid range of s.

val sub_string : bytes -> int -> int -> string
    Same as sub but return a string instead of a byte sequence.

val extend : bytes -> int -> int -> bytes
    extend s left right
    returns a new byte sequence that contains the bytes of s, with left
    uninitialized bytes prepended and right uninitialized bytes appended to it. If left or right
    is negative, then bytes are removed (instead of appended) from the corresponding side of s.
    Raise Invalid_argument if the result length is negative or longer than Sys.max_string_length bytes.

val fill : bytes -> int -> int -> char -> unit
    fill s start len c modifies s in place, replacing len characters with c, starting at start.
    Raise Invalid_argument if start and len do not designate a valid range of s.

val blit : bytes -> int -> bytes -> int -> int -> unit
    blit src srcoff dst dstoff len copies len bytes from sequence src, starting at index
    srcoff, to sequence dst, starting at index dstoff. It works correctly even if src and dst
    are the same byte sequence, and the source and destination intervals overlap.
    Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if
    dstoff and len do not designate a valid range of dst.

val blit_string : string -> int -> bytes -> int -> int -> unit
    blit src srcoff dst dstoff len copies len bytes from string src, starting at index
    srcoff, to byte sequence dst, starting at index dstoff.
    Raise Invalid_argument if srcoff and len do not designate a valid range of src, or if
    dstoff and len do not designate a valid range of dst.
```
val concat : bytes -> bytes list -> bytes
  concat sep sl concatenates the list of byte sequences sl, inserting the separator byte sequence sep between each, and returns the result as a new byte sequence.
  Raise Invalid_argument if the result is longer than Sys.max_string_length\textsuperscript{21.38} bytes.

val cat : bytes -> bytes -> bytes
  cat s1 s2 concatenates s1 and s2 and returns the result as new byte sequence.
  Raise Invalid_argument if the result is longer than Sys.max_string_length\textsuperscript{21.38} bytes.

val iter : (char -> unit) -> bytes -> unit
  iter f s applies function f in turn to all the bytes of s. It is equivalent to f (get s 0); f (get s 1); \ldots; f (get s (length s - 1)); ()).

val iteri : (int -> char -> unit) -> bytes -> unit
  Same as Bytes.iter\textsuperscript{21.4}, but the function is applied to the index of the byte as first argument and the byte itself as second argument.

val map : (char -> char) -> bytes -> bytes
  map f s applies function f in turn to all the bytes of s (in increasing index order) and stores the resulting bytes in a new sequence that is returned as the result.

val mapi : (int -> char -> char) -> bytes -> bytes
  mapi f s calls f with each character of s and its index (in increasing index order) and stores the resulting bytes in a new sequence that is returned as the result.

val trim : bytes -> bytes
  Return a copy of the argument, without leading and trailing whitespace. The bytes regarded as whitespace are the ASCII characters ' ', '\012', '\n', '\r', and '\t'.

val escaped : bytes -> bytes
  Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml. All characters outside the ASCII printable range (32..126) are escaped, as well as backslash and double-quote.
  Raise Invalid_argument if the result is longer than Sys.max_string_length\textsuperscript{21.38} bytes.

val index : bytes -> char -> int
  index s c returns the index of the first occurrence of byte c in s.
  Raise Not_found if c does not occur in s.

val rindex : bytes -> char -> int
  rindex s c returns the index of the last occurrence of byte c in s.
  Raise Not_found if c does not occur in s.
val \texttt{index} : \texttt{bytes} \to \texttt{int} \to \texttt{char} \to \texttt{int}

\texttt{index} \ s \ i \ c \ \mathrm{returns} \ \text{the index of the first occurrence of byte} \ c \ \text{in} \ s \ \text{after position} \ i. \ \texttt{Bytes.index} \ s \ c \ \text{is equivalent to} \ \texttt{Bytes.index_from} \ s \ 0 \ c. 

Raise \texttt{Invalid_argument} \ \text{if} \ i \ \text{is not a valid position in} \ s. \ \texttt{Raise Not_found} \ \text{if} \ c \ \text{does not occur in} \ s \ \text{after position} \ i.

val \texttt{rindex} : \texttt{bytes} \to \texttt{int} \to \texttt{char} \to \texttt{int}

\texttt{rindex} \ s \ i \ c \ \text{returns the index of the last occurrence of byte} \ c \ \text{in} \ s \ \text{before position} \ i+1. \ \texttt{rindex} \ s \ c \ \text{is equivalent to} \ \texttt{rindex_from} \ s \ (\texttt{Bytes.length} \ s - 1) \ c. 

Raise \texttt{Invalid_argument} \ \text{if} \ i+1 \ \text{is not a valid position in} \ s. \ \texttt{Raise Not_found} \ \text{if} \ c \ \text{does not occur in} \ s \ \text{before position} \ i+1.

val \texttt{contains} : \texttt{bytes} \to \texttt{char} \to \texttt{bool}

\texttt{contains} \ s \ c \ \text{tests if} \ \text{byte} \ c \ \text{appears in} \ s.

val \texttt{contains_from} : \texttt{bytes} \to \texttt{int} \to \texttt{char} \to \texttt{bool}

\texttt{contains_from} \ s \ \text{start} \ c \ \text{tests if} \ \text{byte} \ c \ \text{appears in} \ s \ \text{after position} \ \text{start}. \ \texttt{contains} \ s \ c \ \text{is equivalent to} \ \texttt{contains_from} \ s \ 0 \ c. 

Raise \texttt{Invalid_argument} \ \text{if} \ \text{start} \ \text{is not a valid position in} \ s.

val \texttt{rcontains_from} : \texttt{bytes} \to \texttt{int} \to \texttt{char} \to \texttt{bool}

\texttt{rcontains_from} \ s \ \text{stop} \ c \ \text{tests if} \ \text{byte} \ c \ \text{appears in} \ s \ \text{before position} \ \text{stop+1}. 

Raise \texttt{Invalid_argument} \ \text{if} \ \text{stop} < 0 \ \text{or} \ \text{stop+1} \ \text{is not a valid position in} \ s.

val \texttt{uppercase} : \texttt{bytes} \to \texttt{bytes}

\textit{Deprecated.} \ \texttt{Functions operating on Latin-1 character set are deprecated.} \texttt{Return a copy of} \ \text{the argument, with all lowercase letters translated to uppercase, including accented letters of the ISO Latin-1 (8859-1) character set.}

val \texttt{lowercase} : \texttt{bytes} \to \texttt{bytes}

\textit{Deprecated.} \ \texttt{Functions operating on Latin-1 character set are deprecated.} \texttt{Return a copy of} \ \text{the argument, with all uppercase letters translated to lowercase, including accented letters of the ISO Latin-1 (8859-1) character set.}

val \texttt{capitalize} : \texttt{bytes} \to \texttt{bytes}

\textit{Deprecated.} \ \texttt{Functions operating on Latin-1 character set are deprecated.} \texttt{Return a copy of} \ \text{the argument, with the first character set to uppercase, using the ISO Latin-1 (8859-1) character set..}

val \texttt{uncapitalize} : \texttt{bytes} \to \texttt{bytes}

\textit{Deprecated.} \ \texttt{Functions operating on Latin-1 character set are deprecated.} \texttt{Return a copy of} \ \text{the argument, with the first character set to lowercase, using the ISO Latin-1 (8859-1) character set..}
val uppercase_ascii : bytes -> bytes
   Return a copy of the argument, with all lowercase letters translated to uppercase, using the US-ASCII character set.
   Since: 4.03.0

val lowercase_ascii : bytes -> bytes
   Return a copy of the argument, with all uppercase letters translated to lowercase, using the US-ASCII character set.
   Since: 4.03.0

val capitalize_ascii : bytes -> bytes
   Return a copy of the argument, with the first character set to uppercase, using the US-ASCII character set.
   Since: 4.03.0

val uncapitalize_ascii : bytes -> bytes
   Return a copy of the argument, with the first character set to lowercase, using the US-ASCII character set.
   Since: 4.03.0

type t = bytes
   An alias for the type of byte sequences.

val compare : t -> t -> int
   The comparison function for byte sequences, with the same specification as Pervasives.compare[20.2]. Along with the type t, this function compare allows the module Bytes to be passed as argument to the functors Set.Make[21.31] and Map.Make[21.20].

val equal : t -> t -> bool
   The equality function for byte sequences.
   Since: 4.03.0

Unsafe conversions (for advanced users)

This section describes unsafe, low-level conversion functions between bytes and string. They do not copy the internal data; used improperly, they can break the immutability invariant on strings provided by the -safe-string option. They are available for expert library authors, but for most purposes you should use the always-correct Bytes.to_string[21.4] and Bytes.of_string[21.4] instead.

val unsafe_to_string : bytes -> string
   Unsafe convert a byte sequence into a string.

To reason about the use of unsafe_to_string, it is convenient to consider an "ownership" discipline. A piece of code that manipulates some data "owns" it; there are several disjoint ownership modes, including:
Unique ownership: the data may be accessed and mutated

• Shared ownership: the data has several owners, that may only access it, not mutate it.

Unique ownership is linear: passing the data to another piece of code means giving up ownership (we cannot write the data again). A unique owner may decide to make the data shared (giving up mutation rights on it), but shared data may not become uniquely-owned again.

`unsafe_to_string s` can only be used when the caller owns the byte sequence `s` – either uniquely or as shared immutable data. The caller gives up ownership of `s`, and gains ownership of the returned string.

There are two valid use-cases that respect this ownership discipline:

1. Creating a string by initializing and mutating a byte sequence that is never changed after initialization is performed.

   ```
   let string_init len f : string =
   let s = Bytes.create len in
   for i = 0 to len - 1 do Bytes.set s i (f i) done;
   Bytes.unsafe_to_string s
   ```

   This function is safe because the byte sequence `s` will never be accessed or mutated after `unsafe_to_string` is called. The `string_init` code gives up ownership of `s`, and returns the ownership of the resulting string to its caller.

   Note that it would be unsafe if `s` was passed as an additional parameter to the function `f` as it could escape this way and be mutated in the future – `string_init` would give up ownership of `s` to pass it to `f`, and could not call `unsafe_to_string` safely.

   We have provided the `String.init`, `String.map` and `String.mapi` functions to cover most cases of building new strings. You should prefer those over `to_string` or `unsafe_to_string` whenever applicable.

2. Temporarily giving ownership of a byte sequence to a function that expects a uniquely owned string and returns ownership back, so that we can mutate the sequence again after the call ended.

   ```
   let bytes_length (s : bytes) =
   String.length (Bytes.unsafe_to_string s)
   ```

   In this use-case, we do not promise that `s` will never be mutated after the call to `bytes_length s`. The `String.length` function temporarily borrows unique ownership of the byte sequence (and sees it as a `string`), but returns this ownership back to the caller, which may assume that `s` is still a valid byte sequence after the call. Note that this is only correct because we know that `String.length` does not capture its argument – it could escape by a side-channel such as a memoization combinator.
The caller may not mutate \texttt{s} while the string is borrowed (it has temporarily given up ownership). This affects concurrent programs, but also higher-order functions: if \texttt{String.length} returned a closure to be called later, \texttt{s} should not be mutated until this closure is fully applied and returns ownership.

\begin{verbatim}
val unsafe_of_string : string -> bytes
Unsafely convert a shared string to a byte sequence that should not be mutated.

The same ownership discipline that makes \texttt{unsafe_to_string} correct applies to \texttt{unsafe_of_string}: you may use it if you were the owner of the \texttt{string} value, and you will own the return \texttt{bytes} in the same mode.

In practice, unique ownership of string values is extremely difficult to reason about correctly. You should always assume strings are shared, never uniquely owned.

For example, string literals are implicitly shared by the compiler, so you never uniquely own them.

let incorrect = Bytes.unsafe_of_string "hello"
let s = Bytes.of_string "hello"
\end{verbatim}

The first declaration is incorrect, because the string literal "hello" could be shared by the compiler with other parts of the program, and mutating \texttt{incorrect} is a bug. You must always use the second version, which performs a copy and is thus correct.

Assuming unique ownership of strings that are not string literals, but are (partly) built from string literals, is also incorrect. For example, mutating \texttt{unsafe_of_string \("foo" \^ s\)} could mutate the shared string "foo" – assuming a rope-like representation of strings. More generally, functions operating on strings will assume shared ownership, they do not preserve unique ownership. It is thus incorrect to assume unique ownership of the result of \texttt{unsafe_of_string}.

The only case we have reasonable confidence is safe is if the produced \texttt{bytes} is shared – used as an immutable byte sequence. This is possibly useful for incremental migration of low-level programs that manipulate immutable sequences of bytes (for example \texttt{Marshal.from_bytes[21.21]} and previously used the \texttt{string} type for this purpose.

\section*{21.5 Module Callback : Registering OCaml values with the C runtime.}

This module allows OCaml values to be registered with the C runtime under a symbolic name, so that C code can later call back registered OCaml functions, or raise registered OCaml exceptions.

\begin{verbatim}
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 \texttt{caml_named_value(n)}.
\end{verbatim}
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.

21.6 Module Char: Character operations.

val code : char -> int
  Return the ASCII code of the argument.

val chr : int -> char
  Return the character with the given ASCII code. Raise Invalid_argument "Char.chr" if the argument is outside the range 0–255.

val escaped : char -> string
  Return a string representing the given character, with special characters escaped following the lexical conventions of OCaml. All characters outside the ASCII printable range (32..126) are escaped, as well as backslash, double-quote, and single-quote.

val lowercase : char -> char
  Deprecated. Functions operating on Latin-1 character set are deprecated. Convert the given character to its equivalent lowercase character, using the ISO Latin-1 (8859-1) character set.

val uppercase : char -> char
  Deprecated. Functions operating on Latin-1 character set are deprecated. Convert the given character to its equivalent uppercase character, using the ISO Latin-1 (8859-1) character set.

val lowercase_ascii : char -> char
  Convert the given character to its equivalent lowercase character, using the US-ASCII character set.
  Since: 4.03.0

val uppercase_ascii : char -> char
  Convert the given character to its equivalent uppercase character, using the US-ASCII character set.
  Since: 4.03.0

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[20.2]`. Along with the type `t`, this function `compare` allows the module
`Char` to be passed as argument to the functors `Set.Make[21.31]` and `Map.Make[21.20]`.

```ocaml
val equal : t -> t -> bool
  The equal function for chars.
  Since: 4.03.0
```

### 21.7 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`).

```ocaml
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 >= 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
  polar norm arg returns the complex having norm norm and argument arg.

val exp : t -> t
  Exponentiation. \(\exp z\) 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.

21.8 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. This module should not be used for secure and sensitive
cryptographic applications. For these kind of applications more recent and stronger cryptographic
primitives should be used instead.

type t = string
  The type of digests: 16-character strings.
val compare : t -> t -> int
  The comparison function for 16-character digest, with the same specification as
  Pervasives.compare[20.2] and the implementation shared with String.compare[21.37].
  Along with the type t, this function compare allows the module Digest to be passed as
  argument to the functors Set.Make[21.31] and Map.Make[21.20].
  Since: 4.00.0

val equal : t -> t -> bool
  The equal function for 16-character digest.
  Since: 4.03.0

val string : string -> t
  Return the digest of the given string.

val bytes : bytes -> t
  Return the digest of the given byte sequence.
  Since: 4.02.0

val substring : string -> int -> int -> t
  Digest.substring s ofs len returns the digest of the substring of s starting at index ofs
  and containing len characters.

val subbytes : bytes -> int -> int -> t
  Digest.subbytes s ofs len returns the digest of the subsequence of s starting at index
  ofs and containing len bytes.
  Since: 4.02.0

val channel : Pervasives.in_channel -> int -> t
  If len is nonnegative, Digest.channel ic len reads len characters from channel ic and
  returns their digest, or raises End_of_file if end-of-file is reached before len characters are
  read. If len is negative, Digest.channel ic len reads all characters from ic until
  end-of-file is reached and return their digest.

val file : string -> t
  Return the digest of the file whose name is given.

val output : Pervasives.out_channel -> t -> unit
  Write a digest on the given output channel.

val input : Pervasives.in_channel -> t
  Read a digest from the given input channel.

val to_hex : t -> string
Return the printable hexadecimal representation of the given digest.

```ocaml
val from_hex : string -> t
```

Convert a hexadecimal representation back into the corresponding digest. Raise `Invalid_argument` if the argument is not exactly 32 hexadecimal characters.

Since: 4.00.0

---

### 21.9 Module Ephemeron: Ephemerons and weak hash table

Ephemerons and weak hash table

Ephemerons and weak hash table are useful when one wants to cache or memorize the computation of a function, as long as the arguments and the function are used, without creating memory leaks by continuously keeping old computation results that are not useful anymore because one argument or the function is freed. An implementation using is not suitable because all associations would keep in memory the arguments and the result.

Ephemerons can also be used for "adding" a field to an arbitrary boxed ocaml value: you can attach an information to a value created by an external library without memory leaks.

Ephemerons hold some keys and one or no data. They are all boxed ocaml values. The keys of an ephemeron have the same behavior than weak pointers according to the garbage collector. In fact ocaml weak pointers are implemented as ephemersons without data.

The keys and data of an ephemeron are said to be full if they point to a value, empty if the value have never been set, have been unset, or was erased by the GC. In the function that accesses the keys or data these two states are represented by the `option` type.

The data is considered by the garbage collector alive if all the full keys are alive and if the ephemeron is alive. When one of the keys is not considered alive anymore by the GC, the data is emptied from the ephemeron. The data could be alive for another reason and in that case the GC will not free it, but the ephemeron will not hold the data anymore.

The ephemeron complicate the notion of liveness of values, because it is not anymore an equivalence with the reachability from root value by usual pointers (not weak and not ephemersons). With ephemersons the notion of liveness is constructed by the least fixpoint of: A value is alive if:

- it is a root value
- it is reachable from alive value by usual pointers
- it is the data of an alive ephemeron with all its full keys alive

Notes:

- All the types defined in this module cannot be marshaled using `Pervasives.output_value` or the functions of the `Marshal` module.

Ephemerons are defined in a language agnostic way in this paper: B. Hayes, Ephemerons: a New Finalization Mechanism, OOPSLA'9

```ocaml
module type S =
sig
```
Propose the same interface as usual hash table. However since the bindings are weak, even if `mem h k` is true, a subsequent `find h k` may raise `Not_found` because the garbage collector can run between the two.

Moreover, the table shouldn’t be modified during a call to `iter`. Use `filter_map_inplace` in this case.

```ocaml
include Hashtbl.S
val clean : 'a t -> unit
  remove all dead bindings. Done automatically during automatic resizing.
val stats_alive : 'a t -> Hashtbl.statistics
  same as `Hashtbl.SeededS.stats` but only count the alive bindings
```

The output signature of the functor `Ephemeron.K1.Make` and `Ephemeron.K2.Make`. These hash tables are weak in the keys. If all the keys of a binding are alive the binding is kept, but if one of the keys of the binding is dead then the binding is removed.

```ocaml
module type SeededS =
  sig
    include Hashtbl.SeededS
    val clean : 'a t -> unit
      remove all dead bindings. Done automatically during automatic resizing.
    val stats_alive : 'a t -> Hashtbl.statistics
      same as `Hashtbl.SeededS.stats` but only count the alive bindings
  end
```


```ocaml
module K1 :
  sig
    type ('k, 'd) t
      an ephemeron with one key
    val create : unit -> ('k, 'd) t
      `Ephemeron.K1.create ()` creates an ephemeron with one key. The data and the key are empty
```
val get_key : ('k, 'd) t -> 'k option

Ephemeron.K1.get_key eph returns None if the key of eph is empty, Some x (where x is the key) if it is full.

val get_key_copy : ('k, 'd) t -> 'k option

Ephemeron.K1.get_key_copy eph returns None if the key of eph is empty, Some x (where x is a (shallow) copy of the key) if it is full. This function has the same GC friendliness as Weak.get_copy

val set_key : ('k, 'd) t -> 'k -> unit

Ephemeron.K1.set_key eph el sets the key of eph to be a (full) key to el

val unset_key : ('k, 'd) t -> unit

Ephemeron.K1.unset_key eph el sets the key of eph to be an empty key. Since there is only one key, the ephemeron starts behaving like a reference on the data.

val check_key : ('k, 'd) t -> bool

Ephemeron.K1.check_key eph returns true if the key of the eph is full, false if it is empty. Note that even if Ephemeron.K1.check_key eph returns true, a subsequent Ephemeron.K1.get_key eph can return None.

val blit_key : ('k, 'a) t -> ('k, 'b) t -> unit

Ephemeron.K1.blit_key eph1 eph2 sets the key of eph2 with the key of eph1. Contrary to using Ephemeron.K1.get_key followed by Ephemeron.K1.set_key or Ephemeron.K1.unset_key this function does not prevent the incremental GC from erasing the value in its current cycle.

val get_data : ('k, 'd) t -> 'd option

Ephemeron.K1.get_data eph returns None if the data of eph is empty, Some x (where x is the data) if it is full.

val get_data_copy : ('k, 'd) t -> 'd option

Ephemeron.K1.get_data_copy eph returns None if the data of eph is empty, Some x (where x is a (shallow) copy of the data) if it is full. This function has the same GC friendliness as Weak.get_copy

val set_data : ('k, 'd) t -> 'd -> unit

Ephemeron.K1.set_data eph el sets the data of eph to be a (full) data to el

val unset_data : ('k, 'd) t -> unit

Ephemeron.K1.unset_data eph el sets the key of eph to be an empty key. The ephemeron starts behaving like a weak pointer.
val check_data : ('k, 'd) t -> bool
Ephemeron.K1.check_data eph returns true if the data of the eph is full, false if it is empty. Note that even if Ephemeron.K1.check_data eph returns true, a subsequent Ephemeron.K1.get_data\[21.9\] eph can return None.

val blit_data : ('a, 'd) t -> ('b, 'd) t -> unit
Ephemeron.K1.blit_data eph1 eph2 sets the data of eph2 with the data of eph1. Contrary to using Ephemeron.K1.get_data followed by Ephemeron.K1.set_data or Ephemeron.K1.unset_data this function does not prevent the incremental GC from erasing the value in its current cycle.

module Make :
functor (H : Hashtbl.HashedType) -> Ephemeron.S with type key = H.t
Functor building an implementation of a weak hash table

module MakeSeeded :
functor (H : Hashtbl.SeededHashedType) -> Ephemeron.SeededS with type key = H.t
Functor building an implementation of a weak hash table. The seed is similar to the one of Hashtbl.MakeSeeded\[21.14\].

end

module K2 :
sig
  type ('k1, 'k2, 'd) t
  an ephemeron with two keys

  val create : unit -> ('k1, 'k2, 'd) t
  Same as Ephemeron.K1.create\[21.9\]

  val get_key1 : ('k1, 'k2, 'd) t -> 'k1 option
  Same as Ephemeron.K1.get_key\[21.9\]

  val get_key1_copy : ('k1, 'k2, 'd) t -> 'k1 option
  Same as Ephemeron.K1.get_key_copy\[21.9\]

  val set_key1 : ('k1, 'k2, 'd) t -> 'k1 -> unit
  Same as Ephemeron.K1.set_key\[21.9\]

  val unset_key1 : ('k1, 'k2, 'd) t -> unit
val check_key1 : ('k1, 'k2, 'd) t -> bool
    Same as Ephemeron.K1.check_key
val get_key2 : ('k1, 'k2, 'd) t -> 'k2 option
    Same as Ephemeron.K1.get_key
val get_key2_copy : ('k1, 'k2, 'd) t -> 'k2 option
    Same as Ephemeron.K1.get_key_copy
val set_key2 : ('k1, 'k2, 'd) t -> 'k2 -> unit
    Same as Ephemeron.K1.set_key
val unset_key2 : ('k1, 'k2, 'd) t -> unit
    Same as Ephemeron.K1.unset_key
val check_key2 : ('k1, 'k2, 'd) t -> bool
    Same as Ephemeron.K1.check_key
val blit_key1 : ('k1, 'a, 'b) t -> ('k1, 'c, 'd) t -> unit
    Same as Ephemeron.K1.blit_key
val blit_key2 : ('a, 'k2, 'b) t -> ('c, 'k2, 'd) t -> unit
    Same as Ephemeron.K1.blit_key
val blit_key12 : ('k1, 'k2, 'a) t -> ('k1, 'k2, 'b) t -> unit
    Same as Ephemeron.K1.blit_key
val get_data : ('k1, 'k2, 'd) t -> 'd option
    Same as Ephemeron.K1.get_data
val get_data_copy : ('k1, 'k2, 'd) t -> 'd option
    Same as Ephemeron.K1.get_data_copy
val set_data : ('k1, 'k2, 'd) t -> 'd -> unit
    Same as Ephemeron.K1.set_data
val unset_data : ('k1, 'k2, 'd) t -> unit
    Same as Ephemeron.K1.unset_data
val check_data : ('k1, 'k2, 'd) t -> bool
    Same as Ephemeron.K1.check_data[21.9]

val blit_data : ('k1, 'k2, 'd) t -> ('k1, 'k2, 'd) t -> unit
    Same as Ephemeron.K1.blit_data[21.9]

module Make :
    functor (H1 : Hashtbl.HashedType) ->
        functor (H2 : Hashtbl.HashedType) ->
        Ephemeron.S with type key = H1.t * H2.t

    Functor building an implementation of a weak hash table

module MakeSeeded :
    functor (H1 : Hashtbl.SeededHashedType) ->
        functor (H2 : Hashtbl.SeededHashedType) ->
        Ephemeron.SeededS with type key = H1.t * H2.t

    Functor building an implementation of a weak hash table. The seed is similar to the
    one of Hashtbl.MakeSeeded[21.14].

end

module Kn :
    sig

        type ('k, 'd) t
            an ephemeron with an arbitrary number of keys of the same type

        val create : int -> ('k, 'd) t
            Same as Ephemeron.K1.create[21.9]

        val get_key : ('k, 'd) t -> int -> 'k option
            Same as Ephemeron.K1.get_key[21.9]

        val get_key_copy : ('k, 'd) t -> int -> 'k option
            Same as Ephemeron.K1.get_key_copy[21.9]

        val set_key : ('k, 'd) t -> int -> 'k -> unit
            Same as Ephemeron.K1.set_key[21.9]

        val unset_key : ('k, 'd) t -> int -> unit
            Same as Ephemeron.K1.unset_key[21.9]

        val check_key : ('k, 'd) t -> int -> bool
val blit_key : ('k, 'a) t -> int -> ('k, 'b) t -> int -> int -> unit
  Same as Ephemeron.K1.blit_key

val get_data : ('k, 'd) t -> 'd option
  Same as Ephemeron.K1.get_data

val get_data_copy : ('k, 'd) t -> 'd option
  Same as Ephemeron.K1.get_data_copy

val set_data : ('k, 'd) t -> 'd -> unit
  Same as Ephemeron.K1.set_data

val unset_data : ('k, 'd) t -> unit
  Same as Ephemeron.K1.unset_data

val check_data : ('k, 'd) t -> bool
  Same as Ephemeron.K1.check_data

val blit_data : ('k, 'd) t -> ('k, 'd) t -> unit
  Same as Ephemeron.K1.blit_data

module Make :
  functor (H : Hashtbl.HashedType) -> Ephemeron.S with type key = H.t array
    Functor building an implementation of a weak hash table

module MakeSeeded :
  functor (H : Hashtbl.SeededHashedType) -> Ephemeron.SeededS with type key = H.t array
    Functor building an implementation of a weak hash table. The seed is similar to the one of Hashtbl.MakeSeeded

end

module GenHashTable :
  sig
    Define a hash table on generic containers which have a notion of "death" and aliveness. If a binding is dead the hash table can automatically remove it.
    type equal =
      — ETrue — EFalse — EDead
module MakeSeeded :
functor (H : sig
  type t
  keys

  type 'a container
  contains keys and the associated data

  val hash : int -> t -> int
  same as Hashtbl.SeededHashedType

  val equal : 'a container ->
    t -> Ephemeron.GenHashTable.equal
  equality predicate used to compare a key with the one in a container. Can return
  EDead if the keys in the container are dead

  val create : t ->
    'a -> 'a container
  create key data creates a container from some initials keys and one data

  val get_key : 'a container ->
    t option
  get_key cont returns the keys if they are all alive

  val get_data : 'a container -> 'a option
  get_data cont return the data if it is alive

  val set_key_data : 'a container ->
    t -> 'a -> unit
  set_key_data cont modify the key and data

  val check_key : 'a container -> bool
  check_key cont checks if all the keys contained in the data are alive

end ) -> Ephemeron.SeededS with type key = H.t

Functor building an implementation of an hash table that use the container for keeping
the information given

val current_dir_name : string
   The conventional name for the current directory (e.g. . in Unix).

val parent_dir_name : string
   The conventional name for the parent of the current directory (e.g. .. in Unix).

val dir_sep : string
   The directory separator (e.g. / in Unix).

Since: 3.11.2

val concat : string -> string -> string
   concat dir file returns a file name that designates file file in directory dir.

val is_relative : string -> bool
   Return true if the file name is relative to the current directory, false if it is absolute (i.e. in Unix, starts with /).

val is_implicit : string -> bool
   Return true if the file name is relative and does not start with an explicit reference to the current directory (. or ../ in Unix), false if it starts with an explicit reference to the root directory or the current directory.

val check_suffix : string -> string -> bool
   check_suffix name suff returns true if the filename name ends with the suffix suff.

val chop_suffix : string -> string -> string
   chop_suffix name suff removes the suffix suff from the filename name. The behavior is undefined if name does not end with the suffix suff.

val extension : string -> string
   extension name is the shortest suffix ext of name0 where:
      • name0 is the longest suffix of name that does not contain a directory separator;
      • ext starts with a period;
      • ext is preceded by at least one non-period character in name0.

   If such a suffix does not exist, extension name is the empty string.

Since: 4.04

val remove_extension : string -> string
Return the given file name without its extension, as defined in Filename.remove_extension\[21.10\]. If the extension is empty, the function returns the given file name.

The following invariant holds for any file name \( s \):
\[
remove\_extension s ^\ extension s = s
\]

Since: 4.04

val chop_extension : string -> string
Same as Filename.remove_extension\[21.10\], but raise Invalid_argument if the given name has an empty extension.

val basename : string -> string
Split a file name into directory name / base file name. If \( name \) is a valid file name, then \( \text{concat (dirname name) (basename name)} \) returns a file name which is equivalent to \( name \). Moreover, after setting the current directory to dirname name (with Sys.chdir\[21.38\]), references to basename name (which is a relative file name) designate the same file as name before the call to Sys.chdir\[21.38\].

This function conforms to the specification of POSIX.1-2008 for the basename utility.

val dirname : string -> string
See Filename.basename\[21.10\]. This function conforms to the specification of POSIX.1-2008 for the dirname utility.

val temp_file : ?temp_dir:string -> string -> string -> string
\( \text{temp_file prefix suffix} \) 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 the current result of Filename.get_temp_dir_name\[21.10\]. 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 ->
?perms:int ->
?temp_dir:string -> string -> string -> string * Pervasives.out_channel
Same as Filename.temp_file\[21.10\], 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). The file is created with permissions \texttt{perms} (defaults to readable and writable only by the file owner, \texttt{0o600}).

Before 4.03.0 no \texttt{?perms} optional argument
Before 3.11.2 no \texttt{?temp_dir} optional argument

\textbf{Raises} \texttt{Sys_error} if the file could not be opened.

\begin{verbatim}
val get_temp_dir_name : unit -> string

The name of the temporary directory: Under Unix, the value of the \texttt{TMPDIR} environment variable, or "/tmp" if the variable is not set. Under Windows, the value of the \texttt{TEMP} environment variable, or "." if the variable is not set. The temporary directory can be changed with \texttt{Filename.set_temp_dir_name[21.10]}. 

Since: 4.00.0

val set_temp_dir_name : string -> unit

Change the temporary directory returned by \texttt{Filename.get_temp_dir_name[21.10]} and used by \texttt{Filename.temp_file[21.10]} and \texttt{Filename.open_temp_file[21.10]}. 

Since: 4.00.0

val temp_dir_name : string

\textit{Deprecated}. You should use \texttt{Filename.get_temp_dir_name[21.10]} instead. The name of the initial temporary directory: Under Unix, the value of the \texttt{TMPDIR} environment variable, or "/tmp" if the variable is not set. Under Windows, the value of the \texttt{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.
\end{verbatim}

\subsection{21.11 Module Format : Pretty printing.}

This module implements a pretty-printing facility to format values within \texttt{pretty-printing boxes}. The pretty-printer splits lines at specified break hints, and indents lines according to the box structure.

For a gentle introduction to the basics of pretty-printing using \texttt{Format}, read \url{http://caml.inria.fr/resources/doc/guides/format.en.html}.

You may consider this module as providing an extension to the \texttt{printf} facility to provide automatic line splitting. The addition of pretty-printing annotations to your regular \texttt{printf} formats gives you fancy indentation and line breaks. Pretty-printing annotations are described below in the documentation of the function \texttt{Format.fprintf[21.11]}. 

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 `printf` 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 "%s %i\n" "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 `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 splitting. 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.

### Boxes

```ocaml
val open_box : int -> unit
open_box d opens a new pretty-printing box with offset d.

This box prints material as much as possible on every line.

A break hint splits the line if there is no more room on the line to print the remainder of the box. A break hint also splits the line if the splitting “moves to the left” (i.e. it gives an indentation smaller than the one of the current line).

This box is the general purpose pretty-printing box.

If the pretty-printer splits the line in the box, offset d is added to the current indentation.

val close_box : unit -> unit
Closes the most recently opened pretty-printing box.
```
Formatting functions

val print_string : string -> unit
    print_string str prints str in the current box.

val print_as : int -> string -> unit
    print_as len str prints str in the current box. The pretty-printer formats str as if it were of length len.

val print_int : int -> unit
    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.

Break hints

A 'break hint' tells the pretty-printer to output some space or split the line whichever way is more appropriate to the current box splitting rules.

Break hints are used to separate printing items and are mandatory to let the pretty-printer correctly split lines and indent items.

Simple break hints are:

• the 'space': output a space or split the line if appropriate,

• the 'cut': split the line if appropriate.

Note: the notions of space and line splitting are abstract for the pretty-printing engine, since those notions can be completely defined by the programmer. However, in the pretty-printer default setting, “output a space” simply means printing a space character (ASCII code 32) and “split the line” is printing a newline character (ASCII code 10).

val print_space : unit -> unit
    print_space () the 'space' break hint: the pretty-printer may split the line at this point, otherwise it prints one space. It is equivalent to print_break 1 0.

val print_cut : unit -> unit
    print_cut () the 'cut' break hint: the pretty-printer may split the line at this point, otherwise it prints nothing. It is equivalent to print_break 0 0.

val print_break : int -> int -> unit
print_break nspaces offset the 'full' break hint: the pretty-printer may split the line at
this point, otherwise it prints nspaces spaces.

If the pretty-printer splits the line, offset is added to the current indentation.

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 new line in the current box. Not the normal way of pretty-printing, since the new
line does not reset the current line count. You should prefer using break hints within a
vertical box.

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 right margin to d (in characters): the pretty-printer splits lines that
overflow the right margin according to the break hints given. 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 \(10^9\)).

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 maximum indentation limit of lines to d (in characters): once
this limit is reached, new 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 \(10^{-9}\)).

val get_max_indent : unit -> int

Return 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_hbox : unit -> unit

open_hbox () opens a new 'horizontal' pretty-printing box.
This box prints material on a single line.
Break hints in a horizontal box never split the line. (Line splitting may still occur inside boxes nested deeper).

val open_vbox : int -> unit

open_vbox d opens a new ‘vertical’ pretty-printing box with offset d.
This box prints material on as many lines as break hints in the box.
Every break hint in a vertical box splits the line.
If the pretty-printer splits the line in the box, d is added to the current indentation.

val open_hvbox : int -> unit

open_hvbox d opens a new 'horizontal-vertical' pretty-printing box with offset d.
This box behaves as an horizontal box if it fits on a single line, otherwise it behaves as a vertical box.
If the pretty-printer splits the line in the box, d is added to the current indentation.

val open_hovbox : int -> unit

open_hovbox d opens a new 'horizontal-or-vertical' pretty-printing box with offset d.
This box prints material as much as possible on every line.
A break hint splits the line if there is no more room on the line to print the remainder of the box.
If the pretty-printer splits the line in the box, d is added to the current indentation.
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.

Semantic Tags

type tag = string

Semantic 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 semantic
entities (e.g. HTML or TeX elements or terminal escape sequences).

By default, those tags do not influence line splitting calculation: the tag ‘markers’ are not
considered as part of the printing material that drives line splitting (in other words, the length of
those strings is considered as zero for line splitting).

Thus, tag handling is in some sense transparent to pretty-printing and does not interfere with
usual indentation. Hence, a single pretty printing routine can output both simple ‘verbatim’ materi-
al 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 splitting 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 splitting (in
other words, the length of the strings corresponding to tag markers is considered as zero for line
splitting). 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 split 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 t is "<t>" and the closing marker "</t>".

Default tag printing functions just do nothing.

Tag marking and tag printing functions are user definable and can be set by calling
set_formatter_tag_functions.
val open_tag : tag -> unit
  open_tag t opens the tag named t; the print_open_tag function of the formatter is called
with t as argument; the tag marker mark_open_tag t will be flushed into the output device
of the formatter.
val close_tag : unit -> unit
    close_tag () closes the most recently opened tag t. In addition, the print_close_tag
    function of the formatter is called with t as argument. The marker mark_close_tag t will
    be flushed into the output device of the formatter.

val set_tags : bool -> unit
    set_tags b turns on or off the treatment of tags (default is off).

val set_print_tags : bool -> unit
    set_print_tags b turns on or off the printing of tags.

val set_mark_tags : bool -> unit
    set_mark_tags b turns on or off the output of tag markers.

val get_print_tags : unit -> bool
    Return the current status of tags printing.

val get_mark_tags : unit -> bool
    Return the current status of tags marking.

Redirecting the standard formatter output

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.)

val set_formatter_output_functions :
    (string -> int -> int -> unit) -> (unit -> unit) -> unit
    set_formatter_output_functions out flush redirects the pretty-printer output
    functions to the functions out and flush.
    The out function performs all 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 %!, or
    pretty-printing indications @? or @., or using low level functions 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 splitting, and even printing of all the characters that have to be printed!

type formatter_out_functions =
{    out_string : string -> int -> int -> unit ;
    out_flush : unit -> unit ;
    out_newline : unit -> unit ;
    out_spaces : int -> unit ;
}

val set_formatter_out_functions : formatter_out_functions -> unit

set_formatter_out_functions f
Redirect the pretty-printer output to the functions
f.out_string and f.out_flush as described in set_formatter_output_functions. In addition, the pretty-printer function that outputs a newline is set to the function
f.out_newline and the function that outputs indentation spaces is set to the function
f.out_spaces.

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
f.out_spaces and f.out_newline are normally connected to f.out_string and
f.out_flush: respective default values for f.out_space and f.out_newline are
f.out_string (String.make n ' ') 0 n and f.out_string "\n" 0 1.

val get_formatter_out_functions : unit -> formatter_out_functions

Return the current output functions of the pretty-printer, including line splitting and indentation functions. Useful to record the current setting and restore it afterwards.

Changing the meaning of printing semantic 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 a formatter: 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_set_tags : formatter -> bool -> unit
val pp_set_print_tags : formatter -> bool -> unit
val pp_set_mark_tags : formatter -> bool -> unit
val pp_get_print_tags : formatter -> unit -> bool
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_formatter_tag_functions :
  formatter -> formatter_tag_functions -> unit
val pp_get_formatter_tag_functions :
  formatter -> unit -> formatter_tag_functions
val pp_set_formatter_out_functions :
  formatter -> formatter_out_functions -> unit
val pp_get_formatter_out_functions :
  formatter -> unit -> formatter_out_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.

val pp_flush_formatter : formatter -> unit
pp_flush_formatter fmt flushes fmt’s internal queue, ensuring that all the printing and
flushing actions have been performed. In addition, this operation will close all boxes and
reset the state of the formatter.

This will not flush fmt’s output. In most cases, the user may want to use
Convenience formatting functions.

val pp_print_list :
  ?pp_sep:(formatter -> unit -> unit) ->
  (formatter -> 'a -> unit) -> formatter -> 'a list -> unit

  pp_print_list ?pp_sep pp_v ppf 1 prints items of list 1, using pp_v to print each item, and calling pp_sep between items (pp_sep defaults to Format.pp_print_cut[21.11]). Does nothing on empty lists.

  Since: 4.02.0

val pp_print_text : formatter -> string -> unit


  Since: 4.02.0

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. 'h' stands for an 'horizontal' box, 'v' stands for a 'vertical' box, 'hv' stands for an 'horizontal-vertical' box, 'b' stands for an 'horizontal-or-vertical' box demonstrating indentation, 'hov' stands a simple '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 'cut' break hint, as with print_cut ()

  • @ : output a 'space' break hint, as with print_space ()

  • @: output a 'full' break hint as with print_break. The nspaces and offset parameters of the break hint may be optionally specified with the following syntax: the < character, followed by an integer nspaces value, then an integer offset, and a closing > character. If no parameters are provided, the good break defaults to a 'space' break hint.
• @:: flush the pretty printer and split the line, as with \texttt{printf newn}.

• @<n>: print the following item as if it were of length \( n \). Hence, \texttt{printf "@<0>%s" arg} prints \texttt{arg} as a zero length string. If @<n> is not followed by a conversion specification, then the following character of the format is printed as if it were of length \( n \).

• @\{: open a tag. The name of the tag may be optionally specified with the following syntax: the \(<\) character, followed by an optional string specification, and the closing \(>\) character. The string specification is any character string that does not contain the closing character \(>\). If omitted, the tag name defaults to the empty string. For more details about tags, see the functions \texttt{open_tag} and \texttt{close_tag}.

• @\}: close the most recently opened tag.

• @?: flush the pretty printer as with \texttt{printf newn}. This is equivalent to the conversion \texttt{%!}.

• @\n: force a newline, as with \texttt{force newn}, not the normal way of pretty-printing, you should prefer using break hints inside a vertical box.

Note: If you need to prevent the interpretation of a @ character as a pretty-printing indication, you must escape it with a % character. Old quotation mode @@ is deprecated since it is not compatible with formatted input interpretation of character \'@\'.

Example: \texttt{printf "@[%s@ %d@]@." \"x =\" 1} is equivalent to \texttt{open_box(); print_string \"x =\"; print_space(); print_int 1; close_box(); print_newline();}. It prints \texttt{x = 1} within a pretty-printing 'horizontal-or-vertical' box.

val printf : (\'a, formatter, unit) Pervasives.format -> \'a

val eprintf : (\'a, formatter, unit) Pervasives.format -> \'a

val sprintf : (\'a, unit, string) Pervasives.format -> \'a

val asprintf : (\'a, formatter, unit, string) Pervasives.format4 -> \'a

Since: 4.01.0
Chapter 21. The standard library

val ifprintf : formatter -> ('a, formatter, unit) Pervasives.format -> 'a

Same as 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.

Since: 3.12.0

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.

val kasprintf : (string -> 'a) -> ('b, formatter, unit, 'a) Pervasives.format4 -> 'b

Same as asprintf above, but instead of returning the string, passes it to the first argument.

Since: 4.03

Deprecated

val bprintf : Buffer.t -> ('a, formatter, unit) Pervasives.format -> 'a

Deprecated. This function is error prone. 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

Deprecated. An alias for ksprintf.

val set_all_formatter_output_functions :
    out:(string -> int -> int -> unit) ->
    flush:(unit -> unit) ->
    newline:(unit -> unit) ->
    spaces:(int -> unit) -> unit
Deprecated. Subsumed by `set_formatter_out_functions`.

```ml
val get_all_formatter_output_functions :  
  unit ->  
  (string -> int -> int -> unit) * (unit -> unit) * (unit -> unit) *  
  (int -> unit)  
  Deprecated. Subsumed by `get_formatter_out_functions`.
```

```ml
val pp_set_all_formatter_output_functions :  
  formatter ->  
  out:(string -> int -> int -> unit) ->  
  flush:(unit -> unit) ->  
  newline:(unit -> unit) -> spaces:(int -> unit) -> unit  
  Deprecated. Subsumed by `pp_set_formatter_out_functions`.
```

```ml
val pp_get_all_formatter_output_functions :  
  formatter ->  
  unit ->  
  (string -> int -> int -> unit) * (unit -> unit) * (unit -> unit) *  
  (int -> unit)  
  Deprecated. Subsumed by `pp_get_formatter_out_functions`.
```

Tabulation boxes are deprecated.

```ml
val pp_open_tbox : formatter -> unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val pp_close_tbox : formatter -> unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val pp_print_tbreak : formatter -> int -> int -> unit  
  Deprecated. since 4.03.0
```

```ml
val pp_set_tab : formatter -> unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val pp_print_tab : formatter -> unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val open_tbox : unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val close_tbox : unit -> unit  
  Deprecated. since 4.03.0
```

```ml
val print_tbreak : int -> int -> unit
```
Chapter 21. The standard library

Deprecated. since 4.03.0

val set_tab : unit -> unit

Deprecated. since 4.03.0

val print_tab : unit -> unit

Deprecated. since 4.03.0

21.12 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.

  major_words : float ;
    Number of words allocated in the major heap, including the promoted words, since the program was started.

  minor_collections : int ;
    Number of minor collections since the program was started.

  major_collections : int ;
    Number of major collection cycles completed since the program was started.

  heap_words : int ;
    Total size of the major heap, in words.

  heap_chunks : int ;
    Number of contiguous pieces of memory that make up the major heap.

  live_words : int ;
    Number of words of live data in the major heap, including the header words.

  live_blocks : int ;
    Number of live blocks in the major heap.

  free_words : int ;
    Number of words in the free list.
The memory management counters are returned in a `stat` record.

The total amount of memory allocated by the program since it was started is (in words) `minor_words + major_words - promoted_words`. Multiply by the word size (4 on a 32-bit machine, 8 on a 64-bit machine) to get the number of bytes.

type control =
{
mutable minor_heap_size : int ;
    The size (in words) of the minor heap. Changing this parameter will trigger a minor collection. Default: 256k.
mutable major_heap_increment : int ;
    How much to add to the major heap when increasing it. If this number is less than or equal to 1000, it is a percentage of the current heap size (i.e. setting it to 100 will double the heap size at each increase). If it is more than 1000, it is a fixed number of words that will be added to the heap. Default: 15.
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:
Chapter 21. The standard library

- 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 and shared library search at start-up.
- 0x200 Computation of compaction-triggering condition.
- 0x400 Output GC statistics at program exit. Default: 0.

```ocaml
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. If compaction is permanently disabled, it is strongly suggested to set allocation_policy to 1. 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: 1024k.

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.

Since: 3.11.0

window_size : int ;

The size of the window used by the major GC for smoothing out variations in its workload. This is an integer between 1 and 50. Default: 1.

Since: 4.03.0
```

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.

```ocaml
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.

```ocaml
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.

```ocaml
val counters : unit -> float * float * float
```

Return (minor_words, promoted_words, major_words). This function is as fast as quick_stat.

```ocaml
val minor_words : unit -> 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.

In native code this function does not allocate.

**Since:** 4.04

```ocaml
val get : unit -> control
```

Return the current values of the GC parameters in a control record.

```ocaml
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 } 

```ocaml
val minor : unit -> unit
```

Trigger a minor collection.

```ocaml
val major_slice : int -> int
```

major_slice n Do a minor collection and a slice of major collection. n is the size of the slice: the GC will do enough work to free (on average) n words of memory. If n = 0, the GC will try to do enough work to ensure that the next automatic slice has no work to do. This function returns an unspecified integer (currently: 0).

```ocaml
val major : unit -> unit
```

Do a minor collection and finish the current major collection cycle.

```ocaml
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.

```ocaml
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 get_minor_free : unit -> int

Return the current size of the free space inside the minor heap.

Since: 4.03.0

val get_bucket : int -> int

get_bucket n returns the current size of the n-th future bucket of the GC smoothing system. The unit is one millionth of a full GC. Raise Invalid_argument if n is negative, return 0 if n is larger than the smoothing window.

Since: 4.03.0

val get_credit : unit -> int

get_credit () returns the current size of the "work done in advance" counter of the GC smoothing system. The unit is one millionth of a full GC.

Since: 4.03.0

val huge_fallback_count : unit -> int

Return the number of times we tried to map huge pages and had to fall back to small pages. This is always 0 if OCAMLRUNPARAM contains H=1.

Since: 4.03.0

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 (including through weak pointers) 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.

In the presence of multiple OCaml threads it should be assumed that any particular finaliser may be executed in any of the threads.

Anything reachable from the closure of finalisation functions is considered reachable, so the following code will not work as expected:

\[
\text{• let } v = \ldots \text{ in } \text{Gc.finalise } (\text{fun } _{\rightarrow} \ldots v\ldots) v
\]

Instead you should make sure that \(v\) is not in the closure of the finalisation function by writing:

\[
\text{• let } f = \text{fun } x_{\rightarrow} \ldots \text{ let } v = \ldots \text{ in } \text{Gc.finalise } f v
\]

The \(f\) function can use all features of OCaml, 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 guaranteed to be 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. Note that values of types float are sometimes allocated and sometimes not, so finalising them is unsafe, and finalise will also raise Invalid_argument for them. Values of type 'a Lazy.t (for any 'a) are like float in this respect, except that the compiler sometimes optimizes them in a way that prevents finalise from detecting them. In this case, it will not raise Invalid_argument, but you should still avoid calling finalise on lazy values.

The results of calling String.make[21.37], Bytes.make[21.4], Bytes.create[21.4], Array.make[21.2], and Pervasives.ref[20.2] are guaranteed to be heap-allocated and non-constant except when the length argument is 0.

val finalise_last : (unit -> unit) -> 'a -> unit

same as Gc.finalise[21.12] except the value is not given as argument. So you can’t use the given value for the computation of the finalisation function. The benefit is that the function is called after the value is unreachable for the last time instead of the first time. So contrary to Gc.finalise[21.12] the value will never be reachable again or used again. In particular every weak pointer and ephemeron that contained this value as key or data is unset before running the finalisation function. Moreover the finalisation function attached with ‘GC.finalise’ are always called before the finalisation function attached with ‘GC.finalise_last’.

Since: 4.04
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.

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.

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.

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.


This module implements a simple 'standard' lexical analyzer, presented as a function from char-
acter streams to token streams. It implements roughly the lexical conventions of OCaml, but is
parameterized by the set of keywords of your language.

Example: a lexer suitable for a desk calculator is obtained by

let lexer = make_lexer ['+';'-';'*';'/';'let';'=';'('';'')'"

The associated parser would be a function from token stream to, for instance, int, and would
have rules such as:

let rec parse_expr = parser
| [< n1 = parse_atom; n2 = parse_remainder n1 >] -> n2

and parse_atom = parser
| [<'Int n>] -> n
| [<'Kwd "("; n = parse_expr; 'Kwd ")" >] -> n

and parse_remainder n1 = parser
| [<'Kwd "+"; n2 = parse_expr >] -> n1+n2
| [<>] -> n1

One should notice that the use of the parser keyword and associated notation for streams are
only available through camlp4 extensions. This means that one has to preprocess its sources e. g.
by using the "-pp" command-line switch of the compilers.

type token =
| Kwd of string
| Ident of string
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).

```plaintext
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 `Stream.Error` with the offending lexeme as its parameter) otherwise. Blanks and newlines are skipped. Comments delimited by `/*` and `*/` are skipped as well, and can be nested. A `Stream.Failure` exception is raised if end of stream is unexpectedly reached.

### 21.14 Module Hashtbl : Hash tables and hash functions.

Hash tables are hashed association tables, with in-place modification.

**Generic interface**

```plaintext
type ('a, 'b) t
```

The type of hash tables from type `'a` to type `'b`.

```plaintext
val create : ?random:bool -> 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.

The optional `random` parameter (a boolean) controls whether the internal organization of the hash table is randomized at each execution of `Hashtbl.create` or deterministic over all executions.

A hash table that is created with `~random:false` uses a fixed hash function (`Hashtbl.hash`) to distribute keys among buckets. As a consequence, collisions between keys happen deterministically. In Web-facing applications or other security-sensitive applications, the deterministic collision patterns can be exploited by a malicious user to create a denial-of-service attack: the attacker sends input crafted to create many collisions in the table, slowing the application down.

A hash table that is created with `~random:true` uses the seeded hash function `Hashtbl.seeded_hash` with a seed that is randomly chosen at hash table creation.
time. In effect, the hash function used is randomly selected among $2^{30}$ different hash functions. All these hash functions have different collision patterns, rendering ineffective the denial-of-service attack described above. However, because of randomization, enumerating all elements of the hash table using `Hashtbl.fold` or `Hashtbl.iter` is no longer deterministic: elements are enumerated in different orders at different runs of the program.

If no `~random` parameter is given, hash tables are created in non-random mode by default. This default can be changed either programmatically by calling `Hashtbl.randomize` or by setting the R flag in the `OCAMLRUNPARAM` environment variable.

**Before 4.00.0** the `random` parameter was not present and all hash tables were created in non-randomized mode.

```ocaml
val clear : ('a, 'b) t -> unit

Empty a hash table. Use `reset` instead of `clear` to shrink the size of the bucket table to its initial size.

val reset : ('a, 'b) t -> unit

Empty a hash table and shrink the size of the bucket table to its initial size.

Since: 4.00.0
```

```ocaml
val copy : ('a, 'b) t -> ('a, 'b) t

Return a copy of the given hashtable.

val add : ('a, 'b) t -> 'a -> 'b -> unit

`Hashtbl.add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `Hashtbl.remove tbl x`, the previous binding for `x`, if any, is restored. (Same behavior as with association lists.)

val find : ('a, 'b) t -> 'a -> 'b

`Hashtbl.find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

val find_all : ('a, 'b) t -> 'a -> 'b list

`Hashtbl.find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

val mem : ('a, 'b) t -> 'a -> bool

`Hashtbl.mem tbl x` checks if `x` is bound in `tbl`.

val remove : ('a, 'b) t -> 'a -> unit

`Hashtbl.remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.
```
val 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.

If the hash table was created in non-randomized mode, the order in which the bindings are enumerated is reproducible between successive runs of the program, and even between minor versions of OCaml. For randomized hash tables, the order of enumeration is entirely random.

The behavior is not defined if the hash table is modified by f during the iteration.

val filter_map_inplace : ('a -> 'b -> 'b option) -> ('a, 'b) t -> unit

Hashtbl.filter_map_inplace f tbl applies f to all bindings in table tbl and update each binding depending on the result of f. If f returns None, the binding is discarded. If it returns Some new_val, the binding is update to associate the key to new_val.

Other comments for Hashtbl.iter apply as well.

Since: 4.03.0

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.

If the hash table was created in non-randomized mode, the order in which the bindings are enumerated is reproducible between successive runs of the program, and even between minor versions of OCaml. For randomized hash tables, the order of enumeration is entirely random.

The behavior is not defined if the hash table is modified by f during the iteration.

val length : ('a, 'b) t -> int

Hashtbl.length tbl returns the number of bindings in tbl. It takes constant time. Multiple bindings are counted once each, so Hashtbl.length gives the number of times Hashtbl.iter calls its first argument.
val randomize : unit -> unit

After a call to Hashtbl.randomize(), hash tables are created in randomized mode by default: Hashtbl.create \[21.14\] returns randomized hash tables, unless the `random=false` optional parameter is given. The same effect can be achieved by setting the \( R \) parameter in the OCamlRUNPARAM environment variable.

It is recommended that applications or Web frameworks that need to protect themselves against the denial-of-service attack described in Hashtbl.create \[21.14\] call Hashtbl.randomize() at initialization time.

Note that once Hashtbl.randomize() was called, there is no way to revert to the non-randomized default behavior of Hashtbl.create \[21.14\]. This is intentional. Non-randomized hash tables can still be created using Hashtbl.create `random=false`.

Since: 4.00.0

val is_randomized : unit -> bool

return if the tables are currently created in randomized mode by default

Since: 4.02.0

type statistics =

\{ num_bindings : int ;

Number of bindings present in the table. Same value as returned by Hashtbl.length \[21.14\].

num_buckets : int ;

Number of buckets in the table.

max_bucket_length : int ;

Maximal number of bindings per bucket.

bucket_histogram : int array ;

Histogram of bucket sizes. This array \( \text{histo} \) has length \( \text{max_bucket_length} + 1 \). The value of \( \text{histo}.(i) \) is the number of buckets whose size is \( i \).

\}

val stats : ('a, 'b) t -> statistics

Hashtbl.stats tbl returns statistics about the table tbl: number of buckets, size of the biggest bucket, distribution of buckets by size.

Since: 4.00.0

Functorial interface

The functorial interface allows the use of specific comparison and hash functions, either for performance/security concerns, or because keys are not hashable/comparable with the polymorphic builtins.

For instance, one might want to specialize a table for integer keys:
module IntHash =
  struct
  type t = int
  let equal i j = i=j
  let hash i = i land max_int
  end

module IntHashtbl = Hashtbl.Make(IntHash)

let h = IntHashtbl.create 17 in
IntHashtbl.add h 12 "hello"

This creates a new module IntHashtbl, with a new type 'a IntHashtbl.t of tables from int to 'a. In this example, h contains string values so its type is string IntHashtbl.t.

Note that the new type 'a IntHashtbl.t is not compatible with the type ('a,'b) Hashtbl.t of the generic interface. For example, Hashtbl.length h would not type-check, you must use IntHashtbl.length.

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 (provided objects do not contain floats)
      • ((fun x y -> compare x y = 0), Hashtbl.hash) for comparing objects by structure and handling Pervasives.nan correctly
      • (==), Hashtbl.hash) for comparing objects by physical equality (e.g. for mutable or cyclic objects).
  end

The input signature of the functor Hashtbl.Make.

module type S =
  sig
type key
val create : int -> 'a t
val clear : 'a t -> unit
val reset : '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 filter_map_inplace : (key -> 'a -> 'a option) -> 'a t -> unit
val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
val length : 'a t -> int
val stats : 'a t -> Hashtbl.statistics
end

The output signature of the functor Hashtbl.Make[21.14].

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. Since the hash function is not seeded, the create
  operation of the result structure always returns non-randomized hash tables.

module type SeededHashedType =
  sig
    type t
      The type of the hashtable keys.
    val equal : t -> t -> bool
      The equality predicate used to compare keys.
    val hash : int -> t -> int
  end
A seeded hashing function on keys. The first argument is the seed. It must be the case that if equal \( x \) \( y \) is true, then \( \text{hash seed} \ x = \text{hash seed} \ y \) for any value of \( \text{seed} \). A suitable choice for \( \text{hash} \) is the function \texttt{Hashtbl.seeded_hash}\[21.14\] below.

end

The input signature of the functor \texttt{Hashtbl.MakeSeeded}\[21.14\].

Since: 4.00.0

module type SeededS =

sig

  type key
  type 'a t

  val create : ?random:bool -> int -> 'a t
  val clear : 'a t -> unit
  val reset : '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 filter_map_inplace : (key -> 'a -> 'a option) -> 'a t -> unit
  val fold : (key -> 'a -> 'b -> 'b) -> 'a t -> 'b -> 'b
  val length : 'a t -> int
  val stats : 'a t -> Hashtbl.statistics

end

The output signature of the functor \texttt{Hashtbl.MakeSeeded}\[21.14\].

Since: 4.00.0

module MakeSeeded :

functor (H : SeededHashedType) -> SeededS with type key = H.t

Functor building an implementation of the hashtable structure. The functor
\texttt{Hashtbl.MakeSeeded} returns a structure containing a type \texttt{key} of keys and a type \texttt{'a t} of
hash tables associating data of type \texttt{'a} to keys of type \texttt{key}. The operations perform
similarly to those of the generic interface, but use the seeded hashing and equality functions
specified in the functor argument \texttt{H} instead of generic equality and hashing. The \texttt{create}
operation of the result structure supports the \texttt{~random} optional parameter and returns
randomized hash tables if `random: true` is passed or if randomization is globally on (see `Hashtbl.randomize`).

Since: 4.00.0

The polymorphic hash functions

```ocaml
val hash : 'a -> int

Hashtbl.hash x associates a nonnegative integer to any value of any type. It is guaranteed that if \( x = y \) or \( \text{Pervasives.compare} \ x \ y = 0 \), then \( \text{hash} \ x = \text{hash} \ y \). Moreover, \text{hash} always terminates, even on cyclic structures.
```

```ocaml
val seeded_hash : int -> 'a -> int

A variant of `Hashtbl.hash` that is further parameterized by an integer seed.

Since: 4.00.0
```

```ocaml
val hash_param : int -> int -> 'a -> int

Hashtbl.hash_param meaningful total x computes a hash value for x, with the same properties as for \text{hash}. The two extra integer parameters meaningful and total give more precise control over hashing. Hashing performs a breadth-first, left-to-right traversal of the structure x, stopping after meaningful meaningful nodes were encountered, or total nodes (meaningful or not) were encountered. If total as specified by the user exceeds a certain value, currently 256, then it is capped to that value. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of meaningful and total 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 meaningful and total govern the tradeoff between accuracy and speed. As default choices, `Hashtbl.hash` and `Hashtbl.seeded_hash` take meaningful = 10 and total = 100.
```

```ocaml
val seeded_hash_param : int -> int -> 'a -> int

A variant of `Hashtbl.hash_param` that is further parameterized by an integer seed.

Usage: `Hashtbl.seeded_hash_param` that is further parameterized by an integer seed.

Since: 4.00.0
```


This module provides operations on the type \text{int32} of signed 32-bit integers. Unlike the built-in \text{int} type, the type \text{int32} is guaranteed to be exactly 32-bit wide on all platforms. All arithmetic operations over \text{int32} are taken modulo \( 2^{32} \).

Performance notice: values of type \text{int32} occupy more memory space than values of type \text{int}, and arithmetic operations on \text{int32} are generally slower than those on \text{int}. Use \text{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.(//)\[20.2\].
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  
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 $[\text{Int32.min_int}^{[21.15]}, \text{Int32.max_int}^{[21.15]}]$.

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. Along with the type t, this function compare allows the module Int32 to be passed as argument to the functors Set.Make and Map.Make.

val equal : t -> t -> bool
The equal function for int32s.
Since: 4.03.0

21.16 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.(//)[20.2].

val rem : int64 -> int64 -> int64
   Integer remainder. If y is not zero, the result of Int64.rem x y satisfies the following property: x = Int64.add (Int64.mul (Int64.div x y) y) (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 >= 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 $[\text{Int64.min_int}, \text{Int64.max_int}]$.

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.
Chapter 21. The standard library

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[20.2]. Along with the type t, this function compare allows the module
    Int64 to be passed as argument to the functors Set.Make[21.31] and Map.Make[21.20].

val equal : t -> t -> bool
    The equal function for int64s.
    Since: 4.03.0

21.17 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 from_fun : (unit -> 'a) -> 'a t

from_fun f is the same as lazy (f ()) but slightly more efficient.

from_fun should only be used if the function f is already defined. In particular it is always less efficient to write from_fun (fun () -> expr) than lazy expr.

Since: 4.00.0

val from_val : 'a -> 'a t

from_val v returns an already-forced suspension of v. This is for special purposes only and should not be confused with lazy (v).

Since: 4.00.0

val is_val : 'a t -> bool

is_val x returns true if x has already been forced and did not raise an exception.

Since: 4.00.0

val lazy_from_fun : (unit -> 'a) -> 'a t
Deprecation. synonym for from_fun.

val lazy_from_val : 'a -> 'a t
Deprecation. synonym for from_val.

val lazy_is_val : 'a t -> bool
Deprecation. synonym for is_val.

21.18 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 lexbuf and the beginning of the line); pos_cnum is the offset of the position (number of characters between the beginning of the lexbuf and the position). The difference between pos_cnum and pos_bol is the character offset within the line (i.e. the column number, assuming each character is one column wide).

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 : bytes ;
  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 : (bytes -> 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 byte sequence s and a byte count n. The function should put n bytes or fewer in s, starting at index 0, and return the number of bytes 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.


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 cons : 'a -> 'a list -> 'a list
cons x xs is x :: xs
Since: 4.03.0

val hd : 'a list -> 'a
Return the first element of the given list. Raise Failure "hd" if the list is empty.
val tl : 'a list -> 'a list
    Return the given list without its first element. Raise Failure "tl" if the list is empty.

val nth : 'a list -> int -> 'a
    Return the n-th element of the given list. The first element (head of the list) is at position
    0. Raise Failure "nth" if the list is too short. 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
    Concatenate two lists. Same as the infix operator @. Not tail-recursive (length of the first
    argument).

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[21.19] 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
    An alias for concat.

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 iteri : (int -> 'a -> unit) -> 'a list -> unit
    Same as List.iter[21.19], but the function is applied to the index of the element as first
    argument (counting from 0), and the element itself as second argument.
    Since: 4.00.0

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 mapi : (int -> 'a -> 'b) -> 'a list -> 'b list
Same as List.map\[21.19\], but the function is applied to the index of the element as first argument (counting from 0), and the element itself as second argument. Not tail-recursive.

Since: 4.00.0

val rev_map : ('a -> 'b) -> 'a list -> 'b list

List.rev_map f l gives the same result as List.rev\[21.19\] (List.map\[21.19\] 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

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 are determined to have different lengths.

val map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list

List.map2 f [a1; ...; an] [b1; ...; bn] is [f a1 b1; ...; f an bn]. Raise Invalid_argument if the two lists are determined to have different lengths. Not tail-recursive.

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\[21.19\] (List.map2\[21.19\] f l1 l2), but is tail-recursive and more efficient.

val fold_left2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a

List.fold_left2 f a [b1; ...; bn] [c1; ...; cn] is f (... (f (f a b1 c1) b2 c2) ...) bn cn. Raise Invalid_argument if the two lists are determined to have different lengths.

val fold_right2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c

List.fold_right2 f [a1; ...; an] [b1; ...; bn] c is f a1 b1 (f a2 b2 (... (f an bn c) ...)). Raise Invalid_argument if the two lists are determined to have different lengths. Not tail-recursive.
List scanning

val for_all : ('a -> bool) -> 'a list -> bool

for_all p [a1; ...; an] checks if all elements of the list satisfy the predicate p. That is, it returns (p a1) && (p a2) && ... && (p an).

val exists : ('a -> bool) -> 'a list -> bool

exists p [a1; ...; an] checks if at least one element of the list satisfies the predicate p. That is, it returns (p a1) || (p a2) || ... || (p an).

val for_all2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool

Same as List.for_all [21.19], but for a two-argument predicate. Raise Invalid_argument if the two lists are determined to have different lengths.

val exists2 : ('a -> 'b -> bool) -> 'a list -> 'b list -> bool

Same as List.exists [21.19], but for a two-argument predicate. Raise Invalid_argument if the two lists are determined to have different lengths.

val mem : 'a -> 'a list -> bool

mem a l is true if and only if a is equal to an element of l.

val memq : 'a -> 'a list -> bool

Same as List.mem [21.19], 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 [21.19].

val partition : ('a -> bool) -> 'a list -> 'a list * 'a list

partition p l returns a pair of lists (11, 12), where 11 is the list of all the elements of l that satisfy the predicate p, and 12 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 [21.19], but uses physical equality instead of structural equality to compare keys.

val mem_assoc : 'a -> ('a * 'b) list -> bool

Same as List.assoc [21.19], 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 [21.19], 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 [21.19], 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 sort_uniq : ('a -> 'a -> int) -> 'a list -> 'a list
  Same as List.sort, but also remove duplicates.

Since: 4.02.0

val merge : ('a -> 'a -> int) -> 'a list -> '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).

21.20 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.

For instance:

```ocaml
define module IntPairs =
  struct
    type t = int * int
    let compare (x0,y0) (x1,y1) =
      match Pervasives.compare x0 x1 with
        0 -> Pervasives.compare y0 y1
      | c -> c
```
module PairsMap = Map.Make(IntPairs)

let m = PairsMap.(empty |> add (0,1) "hello" |> add (1,0) "world")

This creates a new module PairsMap, with a new type 'a PairsMap.t of maps from int * int to 'a. In this example, m contains string values so its type is string PairsMap.t.

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.

  end
val add : key -> 'a -> 'a t -> 'a t

add x y m returns a map containing the same bindings as m, plus a binding of x to y. If x was already bound in m to a value that is physically equal to y, m is returned unchanged (the result of the function is then physically equal to m). Otherwise, the previous binding of x in m disappears.

**Before 4.03** Physical equality was not ensured.

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. If x was not in m, m is returned unchanged (the result of the function is then physically equal to m).

**Before 4.03** Physical equality was not ensured.

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 union : (key -> 'a -> 'a -> 'a option) ->
'a t -> 'a t -> 'a t

union f m1 m2 computes a map whose keys is the union of keys of m1 and of m2. When the same binding is defined in both arguments, the function f is used to combine them.

Since: 4.03.0

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) -> 'a t -> '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. If p satisfies every binding in m, m is returned unchanged (the result of the function is then physically equal to m)
Before 4.03 Physical equality was not ensured.
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[21.20].
Since: 3.12.0

val min_binding : 'a t -> key * 'a
Return the smallest binding of the given map (with respect to the \texttt{Ord.compare} ordering), or raise \texttt{Not_found} if the map is empty.

\textbf{Since:} 3.12.0

\begin{verbatim}
val max_binding : 'a t -> key * 'a

Same as \texttt{Map.S.min_binding[21.20]}, but returns the largest binding of the given map.

\textbf{Since:} 3.12.0
\end{verbatim}

\begin{verbatim}
val choose : 'a t -> key * 'a

Return one binding of the given map, or raise \texttt{Not_found} if the map is empty. Which binding is chosen is unspecified, but equal bindings will be chosen for equal maps.

\textbf{Since:} 3.12.0
\end{verbatim}

\begin{verbatim}
val split : key -> 'a t -> 'a t * 'a option * 'a t

split \texttt{x m} returns a triple \((l, \text{data}, r)\), 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 \(x\); \texttt{data} is \texttt{None} if \(m\) contains no binding for \(x\), or \texttt{Some \(v\)} if \(m\) binds \(v\) to \(x\).

\textbf{Since:} 3.12.0
\end{verbatim}

\begin{verbatim}
val find : key -> 'a t -> 'a

\texttt{find \(x m\)} returns the current binding of \(x\) in \(m\), or raises \texttt{Not_found} if no such binding exists.
\end{verbatim}

\begin{verbatim}
val map : ('a -> 'b) -> 'a t -> 'b t

\texttt{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.
\end{verbatim}

\begin{verbatim}
val mapi : (key -> 'a -> 'b) -> 'a t -> 'b t

Same as \texttt{Map.S.map[21.20]}, but the function receives as arguments both the key and the associated value for each binding of the map.
\end{verbatim}

\textbf{end}

Output signature of the functor \texttt{Map.Make[21.20]}.

\textbf{module Make :}

\begin{verbatim}
functor (Ord : OrderedType) -> S with type key = Ord.t

Functor building an implementation of the map structure given a totally ordered type.
\end{verbatim}
21.21 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 OCaml.

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 OCaml 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.

Values of extensible variant types, for example exceptions (of extensible type exn), returned by the unmarshaller should not be pattern-matched over through match ... with or try ... with, because unmarshalling does not preserve the information required for matching their constructors. Structural equalities with other extensible variant values does not work either. Most other uses such as Printexc.to_string, will still work as expected.

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
  | Closures
  | Compat_32

The flags to the Marshal.to_* functions below.

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, functional values, and compatibility between 32- and 64-bit platforms.

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 neither detected nor preserved, 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 \( \text{flags} \) does not contain \texttt{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 \( \text{flags} \) contains \texttt{Marshal.Closures}, functional values will be marshaled as a the position in the code of the program together with the values corresponding to the free variables captured in the closure. 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.)

The exact definition of which free variables are captured in a closure is not specified and can vary between bytecode and native code (and according to optimization flags). In particular, a function value accessing a global reference may or may not include the reference in its closure. If it does, unmarshaling the corresponding closure will create a new reference, different from the global one.

If \( \text{flags} \) contains \texttt{Marshal.Compat_32}, marshaling fails when it encounters an integer value outside the range \([-2^{30}, 2^{30}-1]\) of integers that are representable on a 32-bit platform. This ensures that marshaled data generated on a 64-bit platform can be safely read back on a 32-bit platform. If \( \text{flags} \) does not contain \texttt{Marshal.Compat_32}, integer values outside the range \([-2^{30}, 2^{30}-1]\) are marshaled, and can be read back on a 64-bit platform, but will cause an error at un-marshaling time when read back on a 32-bit platform. The \texttt{Marshal.Compat_32} flag only matters when marshaling is performed on a 64-bit platform; it has no effect if marshaling is performed on a 32-bit platform.

val \texttt{to\_bytes} : 'a \to \text{extern\_flags list} \to \text{bytes}

\texttt{Marshal.to\_bytes} \( v \) \( \text{flags} \) returns a byte sequence containing the representation of \( v \). The \text{flags} argument has the same meaning as for \texttt{Marshal.to\_channel}[21.21].

Since: 4.02.0

val \texttt{to\_string} : 'a \to \text{extern\_flags list} \to \text{string}

Same as \texttt{to\_bytes} but return the result as a string instead of a byte sequence.

val \texttt{to\_buffer} : \text{bytes} \to \text{int} \to \text{int} \to 'a \to \text{extern\_flags list} \to \text{int}

\texttt{Marshal.to\_buffer} \( \text{buff} \) \( \text{ofs} \) \( \text{len} \) \( v \) \( \text{flags} \) marshals the value \( v \), storing its byte representation in the sequence \( \text{buff} \), starting at index \( \text{ofs} \), and writing at most \( \text{len} \) bytes. It returns the number of bytes actually written to the sequence. If the byte representation of \( v \) does not fit in \( \text{len} \) characters, the exception \texttt{Failure} is raised.

val \texttt{from\_channel} : \text{Pervasives.in\_channel} \to 'a

\texttt{Marshal.from\_channel} \( \text{chan} \) reads from channel \( \text{chan} \) the byte representation of a structured value, as produced by one of the \texttt{Marshal.to\_*} functions, and reconstructs and returns the corresponding value.

It raises \texttt{End\_of\_file} if the function has already reached the end of file when starting to read from the channel, and raises \texttt{Failure} "input\_value: truncated object" if it reaches the end of file later during the unmarshalling.
Chapter 21. The standard library

val from_bytes : bytes -> int -> 'a

Marshal.from_bytes 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 byte sequence buff, starting at position ofs. The byte
sequence is not mutated.

Since: 4.02.0

val from_string : string -> int -> 'a

Same as from_bytes but take a string as argument instead of a byte sequence.

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 bytes, of the header. Marshal.data_size
buff ofs is the size, in bytes, 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
bytes, 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 byte sequence, the program
needs to read first Marshal.header_size bytes into the sequence, then determine the
length of the remainder of the representation using Marshal.data_size, make sure
the sequence is large enough to hold the remaining data, then read it, and finally call
Marshal.from_bytes to unmarshal the value.

val data_size : bytes -> int -> int

See Marshal.header_size.

val total_size : bytes -> int -> int

See Marshal.header_size.


This meta-module provides labelized version of the Hashtbl, Map and Set modules.

They only differ by their labels. They are provided to help porting from previous versions of
OCaml. The contents of this module are subject to change.

module Hashtbl :

  sig
    type ('a, 'b) t = ('a, 'b) Hashtbl.t
    val create : ?random:bool -> int -> ('a, 'b) t
    val clear : ('a, 'b) t -> unit
val reset : ('a, 'b) t -> unit
val copy : ('a, 'b) t -> ('a, 'b) t
val add : ('a, 'b) t -> key:'a -> data:'b -> unit
val find : ('a, 'b) t -> 'a -> 'b
val find_all : ('a, 'b) t -> 'a -> 'b list
val mem : ('a, 'b) t -> 'a -> bool
val remove : ('a, 'b) t -> 'a -> unit
val replace : ('a, 'b) t -> key:'a -> data:'b -> unit
val iter : f:(key:'a -> data:'b -> unit) -> ('a, 'b) t -> unit
val filter_map_inplace :
  f:(key:'a -> data:'b -> 'b option) -> ('a, 'b) t -> unit
val fold : f:(key:'a -> data:'b -> 'c -> 'c) ->
  ('a, 'b) t -> init:'c -> 'c
val length : ('a, 'b) t -> int
val randomize : unit -> unit
type statistics = Hashtbl.statistics
val stats : ('a, 'b) t -> statistics
module type HashedType =
  Hashtbl.HashedType
module type SeededHashedType =
  Hashtbl.SeededHashedType
module type S =
sig
  type key
  type 'a t
  val create : int -> 'a t
  val clear : 'a t -> unit
  val reset : 'a t -> unit
  val copy : 'a t -> 'a t
  val add : 'a t -> key:'a -> data:'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 -> data:'a -> unit
  val mem : 'a t -> key -> bool
  val iter : f:(key:'a -> data:'a -> unit) ->
    'a t -> unit
val filter_map_inplace :
  f:(key:key -> data:'a -> 'a option) ->
  'a t -> unit
val fold : f:(key:key -> data:'a -> 'b -> 'b) ->
  'a t -> init:'b -> 'b
val length : 'a t -> int
val stats : 'a t -> MoreLabels.Hashtbl.statistics
end

module type SeededS =
  sig
    type key
    type 'a t
    val create : ?random:bool -> int -> 'a t
    val clear : 'a t -> unit
    val reset : 'a t -> unit
    val copy : 'a t -> 'a t
    val add : 'a t ->
      key:key -> data:'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:key -> data:'a -> unit
    val mem : 'a t -> key -> bool
    val iter : f:(key:key -> data:'a -> unit) ->
      'a t -> unit
    val filter_map_inplace :
      f:(key:key -> data:'a -> 'a option) ->
      'a t -> unit
    val fold : f:(key:key -> data:'a -> 'b -> 'b) ->
      'a t -> init:'b -> 'b
    val length : 'a t -> int
    val stats : 'a t -> MoreLabels.Hashtbl.statistics
  end

module Make :
  functor (H : HashedType) -> S with type key = H.t

module MakeSeeded :
  functor (H : SeededHashedType) -> SeededS with type key = H.t
val hash : 'a -> int
val seeded_hash : int -> 'a -> int
val hash_param : int -> int -> 'a -> int
val seeded_hash_param : int -> int -> int -> 'a -> int

end

module Map :
sig

module type OrderedType =
  Map.OrderedType
module type S =
  sig
    type key
    type +'a t
    val empty : 'a t
    val is_empty : 'a t -> bool
    val mem : key -> 'a t -> bool
    val add : key: key ->
      data:'a -> 'a t
    val singleton : key -> 'a -> 'a t
    val remove : key -> 'a t -> 'a t
    val merge :
      f: (key -> 'a option -> 'b option -> 'c option) ->
        'a t -> 'b t -> 'c t
    val union :
      f: (key -> 'a -> 'a -> 'a option) ->
        'a t -> 'a t -> 'a t
    val compare :
      cmp: ('a -> 'a -> int) ->
        'a t -> 'a t -> int
    val equal :
      cmp: ('a -> 'a -> bool) ->
        'a t -> 'a t -> bool
    val iter :
      f: (key: key -> data: 'a -> unit) ->
        'a t -> unit
    val fold :
      f: (key: key -> data: 'a -> 'b -> 'b) ->
        'a t -> init: 'b -> 'b
    val for_all :
      f: (key -> 'a -> bool) -> 'a t -> bool
    val exists :
      f: (key -> 'a -> bool) -> 'a t -> bool
    val filter :
      f: (key -> 'a -> bool) ->
        'a t -> 'a t
    val partition :
      f: (key -> 'a -> bool) ->
Chapter 21. The standard library

'a t -> 'a t * 'a t
val cardinal : 'a t -> int
val bindings : 'a t -> (key * 'a) list
val min_binding : 'a t -> key * 'a
val max_binding : 'a t -> key * 'a
val choose : 'a t -> key * 'a
val split : key ->
  'a t ->
  'a t * 'a option * 'a t
val find : key -> 'a t -> 'a
val map : f:'a -> 'b -> 'a t -> 'b t
val mapi : f:(key -> 'a -> 'b) -> 'a t -> 'b t

end

module Make :
  functor (Ord : OrderedType) -> S with type key = Ord.t
end

module Set :
  sig

    module type OrderedType =
      Set.OrderedType

    module type S =
      sig

        type elt
        type t
        val empty : t
        val is_empty : t -> bool
        val mem : elt -> t -> bool
        val add : elt -> t -> t
        val singleton : elt -> t
        val remove : elt -> t -> t
        val union : t -> t -> t
        val inter : t -> t -> t
        val diff : t -> t -> t
        val compare : t -> t -> int
        val equal : t -> t -> bool

  end


val subset : t -> t -> bool
val iter : f:(elt -> unit) -> t -> unit
val map : f:(elt -> elt) ->
    t -> t
val fold : f:(elt -> 'a -> 'a) -> t -> init:'a -> 'a
val for_all : f:(elt -> bool) -> t -> bool
val exists : f:(elt -> bool) -> t -> bool
val filter : f:(elt -> bool) -> t -> t
val partition : f:(elt -> bool) ->
    t -> t * t
val cardinal : t -> int
val elements : t -> elt list
val min_elt : t -> elt
val max_elt : t -> elt
val choose : t -> elt
val split : elt ->
    t -> t * bool * t
val find : elt -> t -> elt
val of_list : elt list -> t
end

module Make :
    functor (Ord : OrderedType) -> S with type elt = Ord.t
end


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 pointer 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.

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.(/)[20.2].

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 \([\text{Nativeint.min_int}[21.23], \text{Nativeint.max_int}[21.23]\].

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.

```ocaml
val to_int32 : nativeint -> int32
```

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.

```ocaml
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`.

```ocaml
val to_string : nativeint -> string
```

Return the string representation of its argument, in decimal.

```ocaml
type t = nativeint
```

An alias for the type of native integers.

```ocaml
val compare : t -> t -> int
```

The comparison function for native integers, with the same specification as `Pervasives.compare[20.2]`. Along with the type `t`, this function `compare` allows the module `Nativeint` to be passed as argument to the functors `Set.Make[21.31]` and `Map.Make[21.20]`.

```ocaml
val equal : t -> t -> bool
```

The equal function for natives ints.

Since: 4.03.0

### 21.24 Module `Oo`: Operations on objects

```ocaml
val copy : (‘a ..) -> ‘a
```

`Oo.copy o` returns a copy of object `o`, that is a fresh object with the same methods and instance variables as `o`.

```ocaml
val id : (‘a ..) -> int
```

Return an integer identifying this object, unique for the current execution of the program. The generic comparison and hashing functions are based on this integer. When an object is obtained by unmarshaling, the id is refreshed, and thus different from the original object. As a consequence, the internal invariants of data structures such as hash table or sets containing objects are broken after unmarshaling the data structures.

val symbol_start : unit -> int
    symbol_start and Parsing.symbol_end 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.

val rhs_start : int -> int
    Same as Parsing.symbol_start and Parsing.symbol_end, 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.

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


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 [21.26], 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 \( b = \text{true} \)) or off (if \( b = \text{false} \)). 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).

When using this mechanism, one should be aware that an exception backtrace is attached to the thread that saw it raised, rather than to the exception itself. Practically, it means that the code related to fn should not use the backtrace if it has itself raised an exception before.

Since: 3.11.2

Raw backtraces

type raw_backtrace

The abstract type raw_backtrace stores a backtrace in a low-level format, instead of directly exposing them as string as the get_backtrace() function does.

This allows delaying the formatting of backtraces to when they are actually printed, which may be useful if you record more backtraces than you print.

Raw backtraces cannot be marshalled. If you need marshalling, you should use the array returned by the backtrace_slots function of the next section.

Since: 4.01.0

val get_raw_backtrace : unit -> raw_backtrace

Printexc.get_raw_backtrace () returns the same exception backtrace that Printexc.print_backtrace would print, but in a raw format.

Since: 4.01.0

val print_raw_backtrace : Pervasives.out_channel -> raw_backtrace -> unit
Print a raw backtrace in the same format Printexc.print_backtrace uses.

Since: 4.01.0

val raw_backtrace_to_string : raw_backtrace -> string
    Return a string from a raw backtrace, in the same format Printexc.get_backtrace uses.

Since: 4.01.0

Current call stack

val get_callstack : int -> raw_backtrace
    Printexc.get_callstack n returns a description of the top of the call stack on the current program point (for the current thread), with at most n entries. (Note: this function is not related to exceptions at all, despite being part of the Printexc module.)

Since: 4.01.0

Uncaught exceptions

val set_uncaught_exception_handler : (exn -> raw_backtrace -> unit) -> unit
    Printexc.set_uncaught_exception_handler fn registers fn as the handler for uncaught exceptions. The default handler prints the exception and backtrace on standard error output.

    Note that when fn is called all the functions registered with Pervasives.at_exit have already been called. Because of this you must make sure any output channel fn writes on is flushed.

    Also note that exceptions raised by user code in the interactive toplevel are not passed to this function as they are caught by the toplevel itself.

    If fn raises an exception, both the exceptions passed to fn and raised by fn will be printed with their respective backtrace.

Since: 4.02.0

Manipulation of backtrace information

These functions are used to traverse the slots of a raw backtrace and extract information from them in a programmer-friendly format.

type backtrace_slot
    The abstract type backtrace_slot represents a single slot of a backtrace.

Since: 4.02

val backtrace_slots : raw_backtrace -> backtrace_slot array option
Returns the slots of a raw backtrace, or None if none of them contain useful information.

In the return array, the slot at index 0 corresponds to the most recent function call, raise, or primitive get_backtrace call in the trace.

Some possible reasons for returning None are as follow:

- none of the slots in the trace come from modules compiled with debug information (-g)
- the program is a bytecode program that has not been linked with debug information enabled (ocamlc -g)

Since: 4.02.0

type location =
{ filename : string ;
  line_number : int ;
  start_char : int ;
  end_char : int ;
}

The type of location information found in backtraces. start_char and end_char are positions relative to the beginning of the line.

Since: 4.02

module Slot :
  sig

type t = Printexc.backtrace_slot
val is_raise : t -> bool

  is_raise slot is true when slot refers to a raising point in the code, and false when it comes from a simple function call.

  Since: 4.02

val is_inline : t -> bool

  is_inline slot is true when slot refers to a call that got inlined by the compiler, and false when it comes from any other context.

  Since: 4.04.0

val location : t -> Printexc.location option

  location slot returns the location information of the slot, if available, and None otherwise.

  Some possible reasons for failing to return a location are as follow:

  - the slot corresponds to a compiler-inserted raise
  - the slot corresponds to a part of the program that has not been compiled with debug information (-g)

  Since: 4.02
val format : int -> t -> string option

format pos slot returns the string representation of slot as raw_backtrace_to_string would format it, assuming it is the pos-th element of the backtrace: the 0-th element is pretty-printed differently than the others. Whole-backtrace printing functions also skip some uninformative slots; in that case, format pos slot returns None.

Since: 4.02

end

Raw backtrace slots

type raw_backtrace_slot

This type allows direct access to raw backtrace slots, without any conversion in an OCaml-usable data-structure. Being process-specific, they must absolutely not be marshalled, and are unsafe to use for this reason (marshalling them may not fail, but un-marshalling and using the result will result in undefined behavior).

Elements of this type can still be compared and hashed: when two elements are equal, then they represent the same source location (the converse is not necessarily true in presence of inlining, for example).

Since: 4.02.0

val raw_backtrace_length : raw_backtrace -> int

raw_backtrace_length bckt returns the number of slots in the backtrace bckt.

Since: 4.02

val get_raw_backtrace_slot : raw_backtrace -> int -> raw_backtrace_slot

get_slot bckt pos returns the slot in position pos in the backtrace bckt.

Since: 4.02

val convert_raw_backtrace_slot : raw_backtrace_slot -> backtrace_slot

Extracts the user-friendly backtrace_slot from a low-level raw_backtrace_slot.

Since: 4.02

val get_raw_backtrace_next_slot :

raw_backtrace_slot -> raw_backtrace_slot option

get_raw_backtrace_next_slot slot returns the next slot inlined, if any.

Since: 4.04.0
Exception slots

val exn_slot_id : exn -> int

Printexc.exn_slot_id returns an integer which uniquely identifies the constructor used to
create the exception value exn (in the current runtime).

Since: 4.02.0

val exn_slot_name : exn -> string

Printexc.exn_slot_id exn returns the internal name of the constructor used to create the
exception value exn.

Since: 4.02.0

21.27 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 string 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: convert an integer argument to signed decimal.
- u, n, 1, L, or N: convert an integer argument to unsigned decimal. Warning: n, 1, L,
  and N are used for scanf, and should not be used for printf.
- 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: convert a string argument to OCaml syntax (double quotes, escapes).
- c: insert a character argument.
- C: convert a character argument to OCaml syntax (single quotes, escapes).
• f: convert a floating-point argument to decimal notation, in the style \texttt{dddd.ddd}.
• F: convert a floating-point argument to OCaml syntax (\texttt{dddd.} or \texttt{dddd.ddd} or \texttt{d.ddd e+-dd}).
• e or E: convert a floating-point argument to decimal notation, in the style \texttt{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).
• h or H: convert a floating-point argument to hexadecimal notation, in the style \texttt{0xh.hhhh e+-dd} (hexadecimal mantissa, exponent in decimal and denotes a power of 2).
• B: convert a boolean argument to the string \texttt{true} or \texttt{false}
• b: convert a boolean argument (deprecated; do not use in new programs).
• ld, li, lu, lx, lX, lo: convert an \texttt{int32} argument to the format specified by the second letter (decimal, hexadecimal, etc).
• nd, ni, nu, nx, nX, no: convert a \texttt{nativeint} argument to the format specified by the second letter.
• Ld, Li, Lu, Lx, LX, Lo: convert an \texttt{int64} argument to the format specified by the second letter.
• a: user-defined printer. Take two arguments and apply the first one to \texttt{outchan} (the current output channel) and to the second argument. The first argument must therefore have type \texttt{out_channel -> 'b -> unit} and the second \texttt{'b}. The output produced by the function is inserted in the output of \texttt{fprintf} at the current point.
• t: same as \texttt{%a}, but take only one argument (with type \texttt{out_channel -> unit}) and apply it to \texttt{outchan}.
• \texttt{{ fmt %}}: convert a format string argument to its type digest. The argument must have the same type as the internal format string \texttt{fmt}.
• \texttt{( fmt %)}: format string substitution. Take a format string argument and substitute it to the internal format string \texttt{fmt} to print following arguments. The argument must have the same type as the internal format string \texttt{fmt}.
• !: take no argument and flush the output.
• %: take no argument and output one \% character.
• @: take no argument and output one @ character.
• ;: take no argument and output nothing: a 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 signed numerical conversions, prefix number with a + sign if positive.
• space: for signed numerical conversions, prefix number with a space if positive.
• #: request an alternate formatting style for the hexadecimal and octal integer types (x, X, o, lx, 1x, lo, Lx, LX, Lo).

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, \%.*f 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[21.27], but output on stdout.

val eprintf : ('a, Pervasives.out_channel, unit) Pervasives.format -> 'a
Same as Printf.fprintf[21.27], but output on stderr.

val sprintf : ('a, unit, string) Pervasives.format -> 'a
Same as Printf.fprintf[21.27], 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[21.27], but instead of printing on an output channel, append the formatted arguments to the given extensible buffer (see module Buffer[21.3]).

val ifprintf : 'b -> ('a, 'b, 'c, unit) Pervasives.format4 -> 'a
Same as Printf.fprintf[21.27], but does not print anything. Useful to ignore some material when conditionally printing.

Since: 3.10.0

Formatted output functions with continuations.

val kfprintf :
  (Pervasives.out_channel -> 'd) ->
  Pervasives.out_channel ->
  ('a, Pervasives.out_channel, unit, 'd) Pervasives.format4 -> 'a
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 ikfprintf : ('b -> 'd) -> 'b -> ('a, 'b, 'c, 'd) Pervasives.format4 -> 'a
Same as \texttt{kfprintf} above, but does not print anything. Useful to ignore some material when conditionally printing.

\textbf{Since:} 4.0

\begin{description}
\item[\textbf{val ksprintf :}]\texttt{(string -> 'd) -> ('a, unit, string, 'd) Pervasives.format4 -> 'a}
\end{description}

Same as \texttt{sprintf} above, but instead of returning the string, passes it to the first argument.

\textbf{Since:} 3.09.0

\begin{description}
\item[\textbf{val kbprintf :}]\texttt{(Buffer.t -> 'd) -> Buffer.t -> ('a, Buffer.t, unit, 'd) Pervasives.format4 -> 'a}
\end{description}

Same as \texttt{bprintf}, but instead of returning immediately, passes the buffer to its first argument at the end of printing.

\textbf{Since:} 3.10.0

\begin{description}
\item[\textbf{val kprintf :}]\texttt{(string -> 'b) -> ('a, unit, string, 'b) Pervasives.format4 -> 'a}
\end{description}

A deprecated synonym for \texttt{ksprintf}.

\section{Module Queue: First-in first-out queues.}

This module implements queues (FIFOs), with in-place modification.

\textbf{Warning} This module is not thread-safe: each \texttt{Queue.t} value must be protected from concurrent access (e.g. with a \texttt{Mutex.t}). Failure to do so can lead to a crash.

\begin{description}
\item[type 'a t]The type of queues containing elements of type 'a.
\item[exception Empty]Raised when \texttt{Queue.take} or \texttt{Queue.peek} is applied to an empty queue.
\item[\textbf{val create : unit -> 'a t}]Return a new queue, initially empty.
\item[\textbf{val add : 'a -> 'a t -> unit}]\texttt{add x q} adds the element \texttt{x} at the end of the queue \texttt{q}.
\item[\textbf{val push : 'a -> 'a t -> unit}]\texttt{push} is a synonym for \texttt{add}.
\item[\textbf{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 : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b
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
Chapter 21. The standard library

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[21.29] but takes more data as seed.

val self_init : unit -> unit
  Initialize the generator with a random seed chosen in a system-dependent way. If
/dev/urandom is available on the host machine, it is used to provide a highly random initial
seed. Otherwise, a less random seed is computed from system parameters (current time,
process IDs).

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 and bound
  (inclusive). 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.

```ocaml
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.

  end

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.
```
21.30 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`. 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` is `bscanf ic fmt f`, where:

- `ic` is a source of characters (typically a formatted input channel with type `Scanf.Scanning.in_channel`),
- `fmt` is a format string (the same format strings as those used to print material with module `Printf` or `Format`),
- `f` is a function that has as many arguments as the number of values to read in the input according to `fmt`.

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, the result we get is 42.
Formatted input as a functional feature

The OCaml 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 OCaml 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 OCaml formatted input facility is fully type-checked at compile time.

Formatted input channel

module Scanning :
  sig
    type in_channel

    The notion of input channel for the Scanf module: those channels provide all the machinery necessary to read from any source of characters, including a !Pervasives.in_channel value. A Scanf.Scanning.in_channel value is also called a formatted input channel or equivalently a scanning buffer. The type Scanning.scanbuf below is an alias for Scanning.in_channel.

    Since: 3.12.0

    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 Scanning.in_channel formatted input channel attached to !Pervasives.stdin.
    Note: in the interactive system, when input is read from !Pervasives.stdin, the newline character that triggers evaluation is part of 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

    type file_name = string

    A convenient alias to designate a file name.
    Since: 4.00.0
val open_in : file_name -> in_channel

    Scanning.open_in fname returns a !Scanning.in_channel formatted input channel
    for bufferized reading in text mode from 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 : file_name -> in_channel

    Scanning.open_in_bin fname returns a !Scanning.in_channel formatted input
    channel for bufferized reading in binary mode from file fname.
    Since: 3.12.0

val close_in : in_channel -> unit

    Closes the !Pervasives.in_channel associated with the given
    !Scanning.in_channel formatted input channel.
    Since: 3.12.0

val from_file : file_name -> in_channel

    An alias for !Scanning.open_in above.

val from_file_bin : string -> in_channel

    An alias for !Scanning.open_in_bin above.

val from_string : string -> in_channel

    Scanning.from_string s returns a !Scanning.in_channel 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 !Scanning.in_channel 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 !Scanning.in_channel formatted input
    channel which reads from the regular !Pervasives.in_channel input channel ic
    argument. Reading starts at current reading position of ic.
val end_of_input : in_channel -> bool

Scanning.end_of_input ic tests the end-of-input condition of the given !Scanning.in_channel formatted input channel.

val beginning_of_input : in_channel -> bool

Scanning.beginning_of_input ic tests the beginning of input condition of the given !Scanning.in_channel formatted input channel.

val name_of_input : in_channel -> string

Scanning.name_of_input ic returns the name of the character source for the given !Scanning.in_channel formatted input channel.

Since: 3.09.0

val stdib : in_channel

A deprecated alias for !Scanning.stdin, the scanning buffer reading from !Pervasives.stdin.

end

Type of formatted input functions

type ('a, 'b, 'c, 'd) scanner = ('a, Scanning.in_channel, 'b, 'c, 'a -> 'd, 'd) Pervasives.format

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 all the arguments specified by format string fmt, when scan has read those arguments from the !Scanning.in_channel formatted input channel ic.

For instance, the !Scanf.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 formatted input functions must be provided before receiver function f. 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

exception Scan_failure of string

When the input can not be read according to the format string specification, formatted input functions typically raise exception Scan_failure.
Chapter 21. The standard library

The general formatted input function

```ocaml
val bscanf : Scanning.in_channel -> ('a, 'b, 'c, 'd) scanner
bscanf ic fmt r1 ... rN f reads characters from the !Scanning.in_channel formatted
input channel ic and converts them to values according to format string fmt. As a final step,
receiver function f is applied to the values read and gives the result of the 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
the %r conversions specified in the format string.
```

Format string description

The format string 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 \[21.30\]),
- conversion specifications, each of which causes reading and conversion of one argument for
the function f (see \[21.30\]),
- scanning indications to specify boundaries of tokens (see scanning \[21.30\]).

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 (0-9+).
- 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).
• u: reads an unsigned decimal integer.
• x or X: reads an unsigned hexadecimal integer (\([0-9a-fA-F]+\)).
• 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 21.30),
  – a scanning indication (see scanning 21.30) 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 OCaml).
• 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 OCaml).
• f, e, E, g, G: reads an optionally signed floating-point number in decimal notation, in the style \( \text{dddd.ddd e/E+-dd} \).
• h, H: reads an optionally signed floating-point number in hexadecimal notation.
• F: reads a floating point number according to the lexical conventions of OCaml (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, nx, no: reads a nativeint argument to the format specified by the second letter for regular integers.
• Ld, Li, Lu, Lx, 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, %[0-9a-f] returns a string of hexadecimal digits. If a closing bracket appears in a range, it must occur as the first character of the range (or just after the ^ in case of range negation); hence [] matches a ] character and [^[ matches any character that is not ]. Use %% and %@ to include a % or a @ in a range.

- 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\n"", then Scanf.sscanf s "fmt: %{%i%}" succeeds and returns the format string "number is %u".

- ( fmt %): scanning sub-format substitution. Reads a format string rf in the input, then goes on scanning with rf instead of scanning with fmt. The format string rf 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. The conversion returns the format string read rf, and then a value read using rf. Hence, if s is the string "\%4d\%1234.00", then Scanf.sscanf s "%(%i%)" (fun fmt i -> fmt, i) evaluates to ("%4d", 1234). This behaviour is not mere format substitution, since the conversion returns the format string read as additional argument. If you need pure format substitution, use special flag _ to discard the extraneous argument: conversion %_ (fmt %) reads a format string rf and then behaves the same as format string rf. 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.
- @: matches one @ character in the input.
- ,: does nothing.

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-\255] 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 OCaml 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 plain 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 @ character appears anywhere else in the format string, it is treated as a plain character.

Note:

• As usual in format strings, % and @ characters must be escaped using %% and %@; this rule still holds within range specifications and scanning indications. For instance, format "%s@%s" reads a string up to the next % character, and format "%s@%s" reads a string up to the next @.

• 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

```plaintext
val sscanf : string -> ('a, 'b, 'c, 'd) scanner
  Same as `Scanf.bscanf` but reads from the given string.

val scanf : ('a, 'b, 'c, 'd) scanner
  Same as `Scanf.bscanf` but reads from the predefined formatted input channel `Scanf.Scanning.stdin` that is connected to `!Pervasives.stdin`.

val kscanf : Scanning.in_channel ->
  (Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner
  Same as `Scanf.bscanf` 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.

val ksscanf : string ->
  (Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner
  Same as `Scanf.kscanf` but reads from the given string.

Since: 4.02.0
```

Reading format strings from input

```plaintext
val bscanf_format :
  Scanning.in_channel ->
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 ->
  (('a, 'b, 'c, 'd, 'e, 'f) Pervasives.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
```

```plaintext
val sscanf_format :
  string ->
  ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 ->
  (('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 -> 'g) -> 'g
  Same as `Scanf.bscanf_format` but reads from the given string.

Since: 3.09.0
```
val format_from_string : string -> ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6 -> ('a, 'b, 'c, 'd, 'e, 'f) Pervasives.format6

format_from_string s fmt converts a string argument to a format string, according to the given format string fmt. Raise Scan_failure if s, considered as a format string, does not have the same type as fmt.

Since: 3.10.0

val unescaped : string -> string

unescaped s return a copy of s with escape sequences (according to the lexical conventions of OCaml) replaced by their corresponding special characters. More precisely, Scanf.unescaped has the following property: for all string s, Scanf.unescaped (String.escaped s) = s.

Always return a copy of the argument, even if there is no escape sequence in the argument. Raise Scan_failure if s is not properly escaped (i.e. s has invalid escape sequences or special characters that are not properly escaped). For instance, String.unescaped "\\" will fail.

Since: 4.00.0

Deprecated

val fscanf : Pervasives.in_channel -> ('a, 'b, 'c, 'd) scanner

Deprecated. Scanf.fscanf is error prone and deprecated since 4.03.0.

This function violates the following invariant of the Scanf module: To preserve scanning semantics, all scanning functions defined in Scanf must read from a user defined Scanning.in_channel formatted input channel.

If you need to read from a !Pervasives.in_channel input channel ic, simply define a !Scanning.in_channel formatted input channel as in let ib = Scanning.from_channel ic, then use !Scanf.bscanf ib as usual.

val kfscanf :

Pervasives.in_channel ->

(Scanning.in_channel -> exn -> 'd) -> ('a, 'b, 'c, 'd) scanner

Deprecated. Scanf.kfscanf is error prone and deprecated since 4.03.0.

21.31 Module Set : Sets over ordered types.

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.
Chapter 21. The standard library

The Make functor constructs implementations for any type, given a compare function. For instance:

```ocaml
module IntPairs = 
  struct
    type t = int * int
    let compare (x0,y0) (x1,y1) =
      match Pervasives.compare x0 x1 with
      0 -> Pervasives.compare y0 y1
      | c -> c
  end

module PairsSet = Set.Make(IntPairs)

let m = PairsSet.(empty |> add (2,3) |> add (5,7) |> add (11,13))
```

This creates a new module PairsSet, with a new type PairsSet.t of sets of int * int.

```ocaml
module type OrderedType =
  sig
    type t
      The type of the set elements.
    val compare : t -> t -> int
      A total ordering function over the set elements. This is a two-argument function f such that f e1 e2 is zero if the elements e1 and e2 are equal, f e1 e2 is strictly negative if e1 is smaller than e2, and f e1 e2 is strictly positive if e1 is greater than e2.
      Example: a suitable ordering function is the generic structural comparison function Pervasives.compare[20.2].
  end
```

Input signature of the functor Set.Make[21.31].

```ocaml
module type S =
  sig
    type elt
      The type of the set elements.
    type t
      The type of sets.
    val empty : t
```

The empty set.

val is_empty : t -> bool

Test whether a set is empty or not.

val mem : elt -> t -> bool

mem x s tests whether x belongs to the set s.

val add : elt -> t -> t

add x s returns a set containing all elements of s, plus x. If x was already in s, s is returned unchanged (the result of the function is then physically equal to s).
Before 4.03 Physical equality was not ensured.

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 (the result of the function is then physically equal to s).
Before 4.03 Physical equality was not ensured.

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 map : (elt -> elt) -> t -> t

\( \text{map} \ f \ s \) is the set whose elements are \( f \ a_0 \ . \ f \ a_1 \ldots \ f \ a_N \), where \( a_0,a_1\ldots a_N \) are the elements of \( s \).

The elements are passed to \( f \) in increasing order with respect to the ordering over the type of the elements.

If no element of \( s \) is changed by \( f \), \( s \) is returned unchanged. (If each output of \( f \) is physically equal to its input, the returned set is physically equal to \( s \).)

val fold : (elt -> 'a -> 'a) -> t -> 'a -> 'a

\( \text{fold} \ f \ s \ a \) computes \( (f \ x_N \ldots \ (f \ x_2 \ (f \ x_1 \ a))\ldots) \), where \( x_1 \ldots x_N \) are the elements of \( s \), in increasing order.

val for_all : (elt -> bool) -> t -> bool

\( \text{for_all} \ p \ s \) checks if all elements of the set satisfy the predicate \( p \).

val exists : (elt -> bool) -> t -> bool

\( \text{exists} \ p \ s \) checks if at least one element of the set satisfies the predicate \( p \).

val filter : (elt -> bool) -> t -> t

\( \text{filter} \ p \ s \) returns the set of all elements in \( s \) that satisfy predicate \( p \). If \( p \) satisfies every element in \( s \), \( s \) is returned unchanged (the result of the function is then physically equal to \( s \)).

Before 4.03 Physical equality was not ensured.

val partition : (elt -> bool) -> t -> t * t

\( \text{partition} \ p \ s \) returns a pair of sets \( (s_1, s_2) \), where \( s_1 \) is the set of all the elements of \( s \) that satisfy the predicate \( p \), and \( s_2 \) 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 \texttt{Ord.compare}, where \texttt{Ord} is the argument given to \texttt{Set.Make\[21.31\].}

val min_elt : t -> elt
Return the smallest element of the given set (with respect to the \texttt{Ord.compare} ordering), or raise \texttt{Not_found} if the set is empty.

\begin{verbatim}
val max_elt : t \rightarrow elt

Same as \texttt{Set.S.min_elt}\texttt{[21.31]}, but returns the largest element of the given set.
\end{verbatim}

\begin{verbatim}
val choose : t \rightarrow elt

Return one element of the given set, or raise \texttt{Not_found} if the set is empty. Which element is chosen is unspecified, but equal elements will be chosen for equal sets.
\end{verbatim}

\begin{verbatim}
val split : elt \rightarrow t \rightarrow t \times bool \times t

\texttt{split} \texttt{x} \texttt{s} returns a triple \((l, \texttt{present}, r)\), 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 \(x\); \texttt{present} is \texttt{false} if \(s\) contains no element equal to \(x\), or \texttt{true} if \(s\) contains an element equal to \(x\).
\end{verbatim}

\begin{verbatim}
val find : elt \rightarrow t \rightarrow elt

\texttt{find} \texttt{x} \texttt{s} returns the element of \(s\) equal to \(x\) (according to \texttt{Ord.compare}), or raise \texttt{Not_found} if no such element exists.
\texttt{Since: 4.01.0}
\end{verbatim}

\begin{verbatim}
val of_list : elt list \rightarrow t

\texttt{of_list} \texttt{l} creates a set from a list of elements. This is usually more efficient than folding \texttt{add} over the list, except perhaps for lists with many duplicated elements.
\texttt{Since: 4.02.0}
\end{verbatim}

\begin{verbatim}
end

Output signature of the functor \texttt{Set.Make}\texttt{[21.31]}.
\end{verbatim}

\begin{verbatim}
module Make :
  functor (Ord : OrderedType) \rightarrow S with type elt = Ord.t

  Functor building an implementation of the set structure given a totally ordered type.
\end{verbatim}

\subsection{21.32 Module \texttt{Sort} \text{\:} Sorting and merging lists.}

This module is obsolete and exists only for backward compatibility. The sorting functions in \texttt{Array}\texttt{[21.2]} and \texttt{List}\texttt{[21.19]} should be used instead. The new functions are faster and use less memory.

\begin{verbatim}
val list : ('a \rightarrow 'a \rightarrow bool) \rightarrow 'a list \rightarrow 'a list
\end{verbatim}
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.

```ocaml
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.

```ocaml
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.

## 21.33 Module Spacetime: Profiling of a program’s space behaviour over time.

Currently only supported on x86-64 platforms running 64-bit code.

To use the functions in this module you must:

- configure the compiler with `-spacetime`;
- compile to native code. Without these conditions being satisfied the functions in this module will have no effect.

Instead of manually taking profiling heap snapshots with this module it is possible to use an automatic snapshot facility that writes profiling information at fixed intervals to a file. To enable this, all that needs to be done is to build the relevant program using a compiler configured with `-spacetime`; and set the environment variable `OCAML_SPACETIME_INTERVAL` to an integer number of milliseconds giving the interval between profiling heap snapshots. This interval should not be made excessively small relative to the running time of the program. A typical interval to start with might be 1/100 of the running time of the program. The program must exit "normally" (i.e. by calling `exit`, with whatever exit code, rather than being abnormally terminated by a signal) so that the snapshot file is correctly completed.

When using the automatic snapshot mode the profiling output is written to a file called "spacetime-<pid>" where `<pid>` is the process ID of the program. (If the program forks and continues executing then multiple files may be produced with different pid numbers.) The profiling output is by default written to the current working directory when the program starts. This may be customised by setting the `OCAML_SPACETIME_SNAPSHOT_DIR` environment variable to the name of the desired directory.

If using automatic snapshots the presence of the `save_event_for_automatic_snapshots` function, below, should be noted.

The functions in this module are thread safe.

For functions to decode the information recorded by the profiler, see the Spacetime offline library in `otherlibs/`.
module Series :
  sig

  type t

  Type representing a file that will hold a series of heap snapshots together with
  additional information required to interpret those snapshots.

  val create : path:string -> t

  create ~path creates a series file at path.

  val save_event : ?time:float -> t -> event_name:string -> unit

  save_event writes an event, which is an arbitrary string, into the given series file.
  This may be used for identifying particular points during program execution when
  analysing the profile. The optional time parameter is as for Snapshot.take.

  val save_and_close : ?time:float -> t -> unit

  save_and_close series writes information into series required for interpreting the
  snapshots that series contains and then closes the series file. This function must be
  called to produce a valid series file. The optional time parameter is as for
  Snapshot.take.

end

module Snapshot :
  sig

  val take : ?time:float -> Spacetime.Series.t -> unit

  take series takes a snapshot of the profiling annotations on the values in the minor
  and major heaps, together with GC stats, and write the result to the series file. This
  function triggers a minor GC but does not allocate any memory itself. If the optional
  time is specified, it will be used instead of the result of Sys.time as the timestamp of
  the snapshot. Such times should start from zero and be monotonically increasing. This
  parameter is intended to be used so that snapshots can be correlated against wall clock
  time (which is not supported in the standard library) rather than elapsed CPU time.

end

val save_event_for_automatic_snapshots : event_name:string -> unit

Like Series.save_event, but writes to the automatic snapshot file. This function is a
no-op if OCAML_SPACETIME_INTERVAL was not set.
Chapter 21. The standard library

21.34 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` or `Stack.top` is applied to an empty stack.

**val create : unit -> 'a t**

Return a new stack, initially empty.

**val push : 'a -> 'a t -> unit**

push x s adds the element x at the top of stack s.

**val pop : 'a t -> 'a**

pop s removes and returns the topmost element in stack s, or raises Empty if the stack is empty.

**val 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. Time complexity O(1)

**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.

**val fold : ('b -> 'a -> 'b) -> 'b -> 'a t -> 'b**

fold f accu s is (f (... (f (f accu x1) x2) ...) xn) where x1 is the top of the stack, x2 the second element, and xn the bottom element. The stack is unchanged.

**Since:** 4.03
21.35 Module StdLabels : Standard labeled libraries.

This meta-module provides labelized version of the Array, Bytes, List and String modules.

They only differ by their labels. Detailed interfaces can be found in arrayLabels.mli, bytesLabels.mli, listLabels.mli and stringLabels.mli.

module Array :
  ArrayLabels
module Bytes :
  BytesLabels
module List :
  ListLabels
module String :
  StringLabels

21.36 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

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.

Do note that the indices passed to f may not start at 0 in the general case. For example, [< '0'; '1'; Stream.from f >] would call f the first time with count 2.

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_bytes : bytes -> char t
Return the stream of the characters of the bytes parameter.
Since: 4.02.0

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.

A string is an immutable data structure that contains a fixed-length sequence of (single-byte) characters. Each character can be accessed in constant time through its index.

Given a string \( s \) of length \( l \), we can access each of the \( l \) characters of \( s \) via its index in the sequence. Indexes start at 0, and we will call an index valid in \( s \) if it falls within the range \([0...l-1]\) (inclusive). 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 \([0...l]\) (inclusive). Note that the character at index \( n \) is between positions \( n \) and \( n+1 \).

Two parameters \( \text{start} \) and \( \text{len} \) are said to designate a valid substring of \( s \) if \( \text{len} \geq 0 \) and \( \text{start} \) and \( \text{start}+\text{len} \) are valid positions in \( s \).

OCaml strings used to be modifiable in place, for instance via the \text{String.set} and \text{String.blit} functions described below. This usage is deprecated and only possible when the compiler is put in "unsafe-string" mode by giving the \text{-unsafe-string} command-line option (which is currently the default for reasons of backward compatibility). This is done by making the types \text{string} and \text{bytes} (see module \text{Bytes} ) interchangeable so that functions expecting byte sequences can also accept strings as arguments and modify them.

All new code should avoid this feature and be compiled with the \text{-safe-string} command-line option to enforce the separation between the types \text{string} and \text{bytes}.

val length : string -> int

Return the length (number of characters) of the given string.

val get : string -> int -> char

\text{String.get} \( s \) \( n \) returns the character at index \( n \) in string \( s \). You can also write \( s.[n] \) instead of \text{String.get} \( s \) \( n \).

Raise \text{Invalid_argument} if \( n \) not a valid index in \( s \).

val set : bytes -> int -> char -> unit

\text{Deprecated.} This is a deprecated alias of \text{Bytes.set} \( s \) \( n \) \( c \) modifies byte sequence \( s \) in place, replacing the byte at index \( n \) with \( c \). You can also write \( s.[n] \leftarrow c \) instead of \text{String.set} \( s \) \( n \) \( c \).

Raise \text{Invalid_argument} if \( n \) is not a valid index in \( s \).

val create : int -> bytes

\text{Deprecated.} This is a deprecated alias of \text{Bytes.create} \( n \) returns a fresh byte sequence of length \( n \). The sequence is uninitialized and contains arbitrary bytes.

Raise \text{Invalid_argument} if \( n < 0 \) or \( n > \text{Sys.max_string_length} \).

val make : int -> char -> string

\text{String.make} \( n \) \( c \) returns a fresh string of length \( n \), filled with the character \( c \).

Raise \text{Invalid_argument} if \( n < 0 \) or \( n > \text{Sys.max_string_length} \).

val init : int -> (int -> char) -> string
String.init n f returns a string of length n, with character i initialized to the result of f i (called in increasing index order).

Raise Invalid_argument if n < 0 or n > Sys.max_string_length[21.38].

Since: 4.02.0

val copy : string -> string

 Deprecated. Because strings are immutable, it doesn’t make much sense to make identical copies of them. 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 : bytes -> int -> int -> char -> unit

 Deprecated. This is a deprecated alias of Bytes.fill[21.4]. String.fill s start len c modifies byte sequence s in place, replacing len bytes with c, starting at start.

Raise Invalid_argument if start and len do not designate a valid range of s.

val blit : string -> int -> bytes -> int -> int -> unit

Same as Bytes.blit_string[21.4].

val concat : string -> string list -> string

String.concat sep sl concatenates the list of strings sl, inserting the separator string sep between each.

Raise Invalid_argument if the result is longer than Sys.max_string_length[21.38] bytes.

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 iteri : (int -> char -> unit) -> string -> unit

Same as String.iter[21.37], but the function is applied to the index of the element as first argument (counting from 0), and the character itself as second argument.

Since: 4.00.0

val map : (char -> char) -> string -> string

String.map f s applies function f in turn to all the characters of s (in increasing index order) and stores the results in a new string that is returned.

Since: 4.00.0

val mapi : (int -> char -> char) -> string -> string
**String.map f** calls **f** with each character of **s** and its index (in increasing index order) and stores the results in a new string that is returned.

**Since:** 4.02.0

**val trim : string -> string**

Return a copy of the argument, without leading and trailing whitespace. The characters regarded as whitespace are: `' '`, `'\012'`, `'\n'`, `'\r'`, and `'\t'`. If there is neither leading nor trailing whitespace character in the argument, return the original string itself, not a copy.

**Since:** 4.00.0

**val escaped : string -> string**

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of OCaml. All characters outside the ASCII printable range (32..126) are escaped, as well as backslash and double-quote.

If there is no special character in the argument that needs escaping, return the original string itself, not a copy.

Raise **Invalid_argument** if the result is longer than **Sys.max_string_length** bytes. The function **Scanf.unescaped** is a left inverse of **escaped**, i.e. **Scanf.unescaped (escaped s) = s** for any string **s** (unless **escape s** fails).

**val index : string -> char -> int**

**String.index** **s** **c** returns the index 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 index 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 index 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 index 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
    Deprecated. Functions operating on Latin-1 character set are deprecated.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
    Deprecated. Functions operating on Latin-1 character set are deprecated.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
    Deprecated. Functions operating on Latin-1 character set are deprecated.Return a copy of
    the argument, with the first character set to uppercase, using the ISO Latin-1 (8859-1)
    character set..

val uncapitalize : string -> string
    Deprecated. Functions operating on Latin-1 character set are deprecated.Return a copy of
    the argument, with the first character set to lowercase, using the ISO Latin-1 (8859-1)
    character set..

val uppercase_ascii : string -> string
    Return a copy of the argument, with all lowercase letters translated to uppercase, using the
    US-ASCII character set.
    Since: 4.03.0

val lowercase_ascii : string -> string
    Return a copy of the argument, with all uppercase letters translated to lowercase, using the
    US-ASCII character set.
    Since: 4.03.0

val capitalize_ascii : string -> string
    Return a copy of the argument, with the first character set to uppercase, using the
    US-ASCII character set.
Return a copy of the argument, with the first character set to uppercase, using the US-ASCII character set.

**Since:** 4.03.0

```ocaml
val uncapitalize_ascii : string -> string
```

Return a copy of the argument, with the first character set to lowercase, using the US-ASCII character set.

**Since:** 4.03.0

```ocaml
type t = string
```

An alias for the type of strings.

```ocaml
val compare : t -> t -> int
```

The comparison function for strings, with the same specification as `Pervasives.compare`\[20.2\]. Along with the type `t`, this function `compare` allows the module `String` to be passed as argument to the functors `Set.Make`\[21.31\] and `Map.Make`\[21.20\].

```ocaml
val equal : t -> t -> bool
```

The equal function for strings.

**Since:** 4.03.0

```ocaml
val split_on_char : char -> string -> string list
```

`String.split_on_char sep s` returns the list of all (possibly empty) substrings of `s` that are delimited by the `sep` character.

The function’s output is specified by the following invariants:

- The list is not empty.
- Concatenating its elements using `sep` as a separator returns a string equal to the input `(String.concat (String.make 1 sep) (String.split_on_char sep s) = s)`.
- No string in the result contains the `sep` character.

**Since:** 4.04.0

### 21.38 Module Sys : System interface.

Every function in this module raises `Sys_error` with an informative message when the underlying system call signal an error.

```ocaml
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**.

```ocaml
val os_type : string
```

Operating system currently executing the OCaml 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).

```ocaml
type backend_type =
  | Native | Bytecode | Other of string
```

Currently, the official distribution only supports **Native** and **Bytecode**, but it can be other backends with alternative compilers, for example, javascript.

**Since:** 4.04.0

```ocaml
val backend_type : backend_type
```

Backend type currently executing the OCaml program. @ since 4.04.0

```ocaml
val unix : bool
```

True if **Sys.os_type** = "Unix".

**Since:** 4.01.0

```ocaml
val win32 : bool
```

True if **Sys.os_type** = "Win32".

**Since:** 4.01.0

```ocaml
val cygwin : bool
```

True if **Sys.os_type** = "Cygwin".

**Since:** 4.01.0

```ocaml
val word_size : int
```

Size of one word on the machine currently executing the OCaml program, in bits: 32 or 64.

```ocaml
val int_size : int
```

Size of an int. It is 31 bits (resp. 63 bits) when using the OCaml compiler on a 32 bits (resp. 64 bits) platform. It may differ for other compilers, e.g. it is 32 bits when compiling to JavaScript.

**Since:** 4.03.0

```ocaml
val big_endian : bool
```
Whether the machine currently executing the Caml program is big-endian.

**Since:** 4.00.0

```ml
val max_string_length : int
  Maximum length of strings and byte sequences.
```

```ml
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.
```

```ml
val runtime_variant : unit -> string
  Return the name of the runtime variant the program is running on. This is normally the
  argument given to `-runtime-variant` at compile time, but for byte-code it can be changed
  after compilation.

  **Since:** 4.03.0
```

```ml
val runtime_parameters : unit -> string
  Return the value of the runtime parameters, in the same format as the contents of the
  OCAMLRUNPARAM environment variable.

  **Since:** 4.03.0
```

### Signal handling

```ml
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.

val sigabrt : int  
   Abnormal termination

val sigalrm : int  
   Timeout

val sigfpe : int   
   Arithmetic exception

val sighup : int  
   Hangup on controlling terminal

val sigill : int   
   Invalid hardware instruction

val sigint : int   
   Interactive interrupt (ctrl-C)

val sigkill : int  
   Termination (cannot be ignored)

val sigpipe : int  
   Broken pipe

val sigquit : int  
   Interactive termination

val sigsevgy : 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  
   Child process resumed from state indicated by sigchld
val sigstop : int
Stop

val sigtstp : int
Interactive stop

val sigttin : int
Terminal read from background process

val sigttou : int
Terminal write from background process

val sigvtalrm : int
Timeout in virtual time

val sigprof : int
Profiling interrupt

val sigbus : int
Bus error
Since: 4.03

val sigpoll : int
Pollable event
Since: 4.03

val sigsys : int
Bad argument to routine
Since: 4.03

val sigtrap : int
Trace/breakpoint trap
Since: 4.03

val sigurg : int
Urgent condition on socket
Since: 4.03

val sigxcpu : int
Timeout in cpu time
Since: 4.03
val sigxfsz : int
    File size limit exceeded
Since: 4.03

exception Break
    Exception raised on interactive interrupt if Sys.catch_break is on.

val catch_break : bool -> unit
    catch_break governs whether interactive interrupt (ctrl-C) terminates the program or
    raises the Break exception. Call catch_break true to enable raising Break, and
    catch_break false to let the system terminate the program on user interrupt.

val ocaml_version : string
    ocaml_version is the version of OCaml. 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.

val enable_runtime_warnings : bool -> unit
    Control whether the OCaml runtime system can emit warnings on stderr. Currently, the
    only supported warning is triggered when a channel created by open_* functions is finalized
    without being closed. Runtime warnings are enabled by default.
Since: 4.03.0

val runtime_warnings_enabled : unit -> bool
    Return whether runtime warnings are currently enabled.
Since: 4.03.0

Optimization

val opaque_identity : 'a -> 'a
    For the purposes of optimization, opaque_identity behaves like an unknown (and thus
    possibly side-effecting) function.
    At runtime, opaque_identity disappears altogether.
    A typical use of this function is to prevent pure computations from being optimized
    away in benchmarking loops. For example:

    for _round = 1 to 100_000 do
        ignore (Sys.opaque_identity (my_pure_computation ()))
    done

Since: 4.03.0

Since: 4.03

type t

The type for Unicode characters.

A value of this type represents a Unicode scalar value which is an integer in the ranges 0x0000...0xD7FF or 0xE000...0x10FFFF.

val min : t

min is U+0000.

val max : t

max is U+10FFFF.

val succ : t -> t

succ u is the scalar value after u in the set of Unicode scalar values.

Raises Invalid_argument if u is Uchar.max[21.39].

val pred : t -> t

pred u is the scalar value before u in the set of Unicode scalar values.

Raises Invalid_argument if u is Uchar.min[21.39].

val is_valid : int -> bool

is_valid n is true iff n is an Unicode scalar value (i.e. in the ranges 0x0000...0xD7FF or 0xE000...0x10FFFF).

val of_int : int -> t

of_int i is i as an Unicode character.

Raises Invalid_argument if i does not satisfy Uchar.is_valid[21.39].

val to_int : t -> int

to_int u is u as an integer.

val is_char : t -> bool

is_char u is true iff u is a latin1 OCaml character.

val of_char : char -> t

of_char c is c as an Unicode character.

val to_char : t -> char
to_char u is u as an OCaml latin1 character.

Raises Invalid_argument if u does not satisfy Uchar.is_char\textsuperscript{21.39}.

val equal : t -> t -> bool
equal u u' is u = u'.

val compare : t -> t -> int
compare u u' is Pervasives.compare u u'.

val hash : t -> int
hash u associates a non-negative integer to u.

val dump : Format.formatter -> t -> unit
dump ppf u prints a representation of u on ppf using only US-ASCII encoded characters
according to the Unicode notational convention for code points\textsuperscript{http://www.unicode.org/versions/latest/appA.pdf}.


Low-level functions

\begin{verbatim}
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\textsuperscript{20.2} nor the
functions of the Marshal\textsuperscript{21.21} module.
\end{verbatim}

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\textsuperscript{21.38}-1.

val length : 'a t -> int

Weak.length ar returns the length (number of elements) of ar.
Chapter 21. The standard library

val set : 'a t -> int -> 'a option -> unit
    Weak.set ar n (Some el) sets the nth cell of ar to be a (full) pointer to el; Weak.set ar n None sets the nth cell of ar to empty. Raise Invalid_argument "Weak.set" if n is not in the range 0 to Weak.length a - 1.

val get : 'a t -> int -> 'a option
    Weak.get ar n returns None if the nth 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 a - 1.

val get_copy : 'a t -> int -> 'a option
    Weak.get_copy ar n returns None if the nth 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 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 a.

val blit : 'a t -> int -> 'a t -> int -> int -> unit
    Weak.blit ar1 off1 ar2 off2 len copies len weak pointers from ar1 (starting at off1) to ar2 (starting at off2). It works correctly even if ar1 and ar2 are the same. Raise Invalid_argument "Weak.blit" if off1 and len do not designate a valid subarray of ar1, or if off2 and len do not designate a valid subarray of ar2.

Weak hash sets

A weak hash set 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 sets 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 sets 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 set, 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 \( f \ t \) \( \text{init} \) computes \((f \ d_1 \ldots \ (f \ d_N \ \text{init}))\) where \(d_1 \ldots \ d_N\) are the elements of \(t\) in some unspecified order. It is not specified what happens if \(f\) tries to change \(t\) itself.

val count : \(t\) \(
\rightarrow\) int

Count the number of elements in the table. \(\text{count} \ t\) gives the same result as \(\text{fold} (\text{fun } _n \rightarrow n+1) \ t \ 0\) but does not delay the deallocation of the dead elements.

val stats : \(t\) \(
\rightarrow\) 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.

end

The output signature of the functor Weak.Make\([21.40]\).

module Make :
functor (H : Hashtbl.HashedType) \(
\rightarrow\) S with type data = H.t

Functor building an implementation of the weak hash set structure. \(H.\text{equal}\) can’t be the physical equality, since only shallow copies of the elements in the set are given to it.
Chapter 22

The compiler front-end

This chapter describes the OCaml front-end, which declares the abstract syntax tree used by the compiler, provides a way to parse, print and pretty-print OCaml code, and ultimately allows to write abstract syntax tree preprocessors invoked via the -ppx flag (see chapters 8 and 11).

It is important to note that the exported front-end interface follows the evolution of the OCaml language and implementation, and thus does not provide any backwards compatibility guarantees.

The front-end is a part of compiler-libs library. Programs that use the compiler-libs library should be built as follows:

```
ocamlfind ocamlc other options -package compiler-libs.common other files
ocamlfind ocamlopt other options -package compiler-libs.common other files
```

Use of the ocamlfind utility is recommended. However, if this is not possible, an alternative method may be used:

```
ocamlc other options -I +compiler-libs ocamlcommon.cma other files
ocamlopt other options -I +compiler-libs ocamlcommon.cmxa other files
```

For interactive use of the compiler-libs library, start ocaml and type

```
#load "compiler-libs/ocamlcommon.cma";;
```

22.1 Module Ast_mapper : The interface of a -ppx rewriter

A -ppx rewriter is a program that accepts a serialized abstract syntax tree and outputs another, possibly modified, abstract syntax tree. This module encapsulates the interface between the compiler and the -ppx rewriters, handling such details as the serialization format, forwarding of command-line flags, and storing state.

Ast_mapper.mapper[22.1.1] allows to implement AST rewriting using open recursion. A typical mapper would be based on Ast_mapper.default_mapper[22.1.1], a deep identity mapper, and will fall back on it for handling the syntax it does not modify. For example:

```
open Asttypes
open Parsetree
open Ast_mapper
```
let test_mapper argv =
{ default_mapper with
   expr = fun mapper expr ->
      match expr with
      | { pexp_desc = Pexp_extension ({ txt = "test" }, PStr [])} ->
        Ast_helper.Exp.constant (Const_int 42)
      | other -> default_mapper.expr mapper other; }

let () =
  register "ppx_test" test_mapper

This -ppx rewriter, which replaces [%test] in expressions with the constant 42, can be compiled using ocamlc -o ppx_test -I +compiler-libs ocamlcommon.cma ppx_test.ml.

### 22.1.1 A generic Parsetree mapper

type mapper =
{ attribute : mapper -> Parsetree.attribute -> Parsetree.attribute ;
 attributes : mapper -> Parsetree.attribute list -> Parsetree.attribute list ;
 case : mapper -> Parsetree.case -> Parsetree.case ;
 cases : mapper -> Parsetree.case list -> Parsetree.case list ;
 class_declaration : mapper -> Parsetree.class_declaration -> Parsetree.class_declaration ;
 class_description : mapper -> Parsetree.class_description -> Parsetree.class_description ;
 class_expr : mapper -> Parsetree.class_expr -> Parsetree.class_expr ;
 class_field : mapper -> Parsetree.class_field -> Parsetree.class_field ;
 class_signature : mapper -> Parsetree.class_signature -> Parsetree.class_signature ;
 class_structure : mapper -> Parsetree.class_structure -> Parsetree.class_structure ;
 class_type : mapper -> Parsetree.class_type -> Parsetree.class_type ;
 class_type_declaration : mapper -> Parsetree.class_type_declaration -> Parsetree.class_type_declaration ;
 class_type_field : mapper -> Parsetree.class_type_field -> Parsetree.class_type_field ;
 constructor_declaration : mapper -> Parsetree.constructor_declaration -> Parsetree.constructor_declaration ;
 expr : mapper -> Parsetree.expression -> Parsetree.expression ;
 extension : mapper -> Parsetree.extension -> Parsetree.extension ;
 extension_constructor : mapper -> Parsetree.extension_constructor -> Parsetree.extension_constructor ;
 include_declaration : mapper -> Parsetree.include_declaration -> Parsetree.include_declaration ;
 include_description : mapper -> Parsetree.include_description -> Parsetree.include_description ;
 label_declaration : mapper ->
Parsetree.label_declaration -> Parsetree.label_declaration ;
location : mapper -> Location.t -> Location.t ;
module_binding : mapper -> Parsetree.module_binding -> Parsetree.module_binding ;
module_declaration : mapper ->
Parsetree.module_declaration -> Parsetree.module_declaration ;
module_expr : mapper -> Parsetree.module_expr -> Parsetree.module_expr ;
module_type : mapper -> Parsetree.module_type -> Parsetree.module_type ;
module_type_declaration : mapper ->
Parsetree.module_type_declaration -> Parsetree.module_type_declaration ;
open_description : mapper -> Parsetree.open_description -> Parsetree.open_description ;
pat : mapper -> Parsetree.pattern -> Parsetree.pattern ;
payload : mapper -> Parsetree.payload -> Parsetree.payload ;
signature : mapper -> Parsetree.signature -> Parsetree.signature ;
signature_item : mapper -> Parsetree.signature_item -> Parsetree.signature_item ;
structure : mapper -> Parsetree.structure -> Parsetree.structure ;
structure_item : mapper -> Parsetree.structure_item -> Parsetree.structure_item ;
typ : mapper -> Parsetree.core_type -> Parsetree.core_type ;
type_declaration : mapper -> Parsetree.type_declaration -> Parsetree.type_declaration ;
type_extension : mapper -> Parsetree.type_extension -> Parsetree.type_extension ;
type_kind : mapper -> Parsetree.type_kind -> Parsetree.type_kind ;
value_binding : mapper -> Parsetree.value_binding -> Parsetree.value_binding ;
value_description : mapper ->
Parsetree.value_description -> Parsetree.value_description ;
with_constraint : mapper -> Parsetree.with_constraint -> Parsetree.with_constraint ;
}

A mapper record implements one "method" per syntactic category, using an open recursion style: each method takes as its first argument the mapper to be applied to children in the syntax tree.

val default_mapper : mapper

A default mapper, which implements a "deep identity" mapping.

22.1.2 Apply mappers to compilation units

val tool_name : unit -> string

Can be used within a ppx preprocessor to know which tool is calling it "ocamlc", "ocamlopt", "ocamldoc", "ocamldep", "ocaml", ... Some global variables that reflect command-line options are automatically synchronized between the calling tool and the ppx preprocessor: Clflags.include_dirs, Config.load_path, Clflags.open_modules, Clflags.for_package, Clflags.debug.

val apply : source:string -> target:string -> mapper -> unit

Apply a mapper (parametrized by the unit name) to a dumped parsetree found in the source file and put the result in the target file. The structure or signature field of the mapper is applied to the implementation or interface.
val run_main : (string list -> mapper) -> unit

Entry point to call to implement a standalone -ppx rewriter from a mapper, parametrized
by the command line arguments. The current unit name can be obtained from
Location.input_name. This function implements proper error reporting for uncaught
exceptions.

22.1.3 Registration API

val register_function :
  (string -> (string list -> mapper) -> unit) Pervasives.ref
val register : string -> (string list -> mapper) -> unit

Apply the register_function. The default behavior is to run the mapper immediately,
taking arguments from the process command line. This is to support a scenario where a
mapper is linked as a stand-alone executable.

It is possible to overwrite the register_function to define ”-ppx drivers”, which combine
several mappers in a single process. Typically, a driver starts by defining
register_function to a custom implementation, then lets ppx rewriters (linked statically
or dynamically) register themselves, and then run all or some of them. It is also possible to
have -ppx drivers apply rewriters to only specific parts of an AST.

The first argument to register is a symbolic name to be used by the ppx driver.

22.1.4 Convenience functions to write mappers

val map_opt : ('a -> 'b) -> 'a option -> 'b option
val extension_of_error : Location.error -> Parsetree.extension

Encode an error into an ‘ocaml.error’ extension node which can be inserted in a generated
Parsetree. The compiler will be responsible for reporting the error.

val attribute_of_warning : Location.t -> string -> Parsetree.attribute

Encode a warning message into an ‘ocaml.ppwarning’ attribute which can be inserted in a
generated Parsetree. The compiler will be responsible for reporting the warning.

22.1.5 Helper functions to call external mappers

val add_ppx_context_str :
  tool_name:string -> Parsetree.structure -> Parsetree.structure

Extract information from the current environment and encode it into an attribute which is
prepended to the list of structure items in order to pass the information to an external
processor.

val add_ppx_context_sig :
  tool_name:string -> Parsetree.signature -> Parsetree.signature
Same as `add_ppx_context_str`, but for signatures.

```ocaml
val drop_ppx_context_str : restore:bool -> Parsetree.structure -> Parsetree.structure
  Drop the ocaml.ppx.context attribute from a structure. If `restore` is true, also restore the
  associated data in the current process.
```

```ocaml
val drop_ppx_context_sig : restore:bool -> Parsetree.signature -> Parsetree.signature
  Same as `drop_ppx_context_str`, but for signatures.
```

### 22.1.6 Cookies

Cookies are used to pass information from a ppx processor to a further invocation of itself, when
called from the OCaml toplevel (or other tools that support cookies).

```ocaml
val set_cookie : string -> Parsetree.expression -> unit
val get_cookie : string -> Parsetree.expression option
```

### 22.2 Module Asttypes : Auxiliary AST types used by parsetree and typedtree.

```ocaml
type constant =
  | Const_int of int
  | Const_char of char
  | Const_string of string * string option
  | Const_float of string
  | Const_int32 of int32
  | Const_int64 of int64
  | Const_nativeint of nativeint

type rec_flag =
type direction_flag =
type private_flag =
type mutable_flag =
type virtual_flag =
type override_flag =
type closed_flag =
type label = string

type arg_label =
  | Nolabel
  | Labelled of string
  | Optional of string
```
type 'a loc = 'a Location.loc =
{ txt : 'a ;
  loc : Location.t ;
}
type variance =

22.3 Module Location: Source code locations (ranges of positions), used in parsetree.

type t =
{ loc_start : Lexing.position ;
  loc_end : Lexing.position ;
  loc_ghost : bool ;
}

Note on the use of Lexing.position in this module. If pos_fname = "", then use !input_name instead. If pos_lnum = -1, then pos_bol = 0. Use pos_cnum and re-parse the file to get the line and character numbers. Else all fields are correct.

val none : t
  An arbitrary value of type t; describes an empty ghost range.

val in_file : string -> t
  Return an empty ghost range located in a given file.

val init : Lexing.lexbuf -> string -> unit
  Set the file name and line number of the lexbuf to be the start of the named file.

val curr : Lexing.lexbuf -> t
  Get the location of the current token from the lexbuf.

val symbol_rloc : unit -> t
val symbol_gloc : unit -> t
val rhs_loc : int -> t
  rhs_loc n returns the location of the symbol at position n, starting at 1, in the current parser rule.

val input_name : string Pervasives.ref
val input_lexbuf : Lexing.lexbuf option Pervasives.ref
val get_pos_info : Lexing.position -> string * int * int
val print_loc : Format.formatter -> t -> unit
val print_error : Format.formatter -> t -> unit
val print_error_cur_file : Format.formatter -> unit -> unit
val print_warning : t -> Format.formatter -> Warnings.t -> unit
val formatter_for_warnings : Format.formatter Pervasives.ref
val prerr_warning : t -> Warnings.t -> unit
val echo_eof : unit -> unit
val reset : unit -> unit
val warning_printer :
  (t -> Format.formatter -> Warnings.t -> unit) Pervasives.ref
  Hook for intercepting warnings.
val default_warning_printer : t -> Format.formatter -> Warnings.t -> unit
  Original warning printer for use in hooks.
val highlight_locations : Format.formatter -> t list -> bool
type 'a loc =
  {  txt : 'a ;
      loc : t ;
  }
val mknoloc : 'a -> 'a loc
val mkloc : 'a -> t -> 'a loc
val print : Format.formatter -> t -> unit
val print_compact : Format.formatter -> t -> unit
val print_filename : Format.formatter -> string -> unit
val absolute_path : string -> string
  In -absname mode, return the absolute path for this filename. Otherwise, returns the
  filename unchanged.
val absname : bool Pervasives.ref
type error =
  {  loc : t ;
      msg : string ;
      sub : error list ;
      if_highlight : string ;
  }
exception Error of error
val print_error_prefix : Format.formatter -> unit -> unit
val error :
  ?loc:t ->
  ?sub:error list -> ?if_highlight:string -> string -> error
val errorf :
?loc:t ->
?sub:error list ->
?if_highlight:string ->
('a, Format.formatter, unit, error) Pervasives.format4 -> 'a

val raise_errorf :
?loc:t ->
?sub:error list ->
?if_highlight:string ->
('a, Format.formatter, unit, 'b) Pervasives.format4 -> 'a

val error_of_printer : t -> (Format.formatter -> 'a -> unit) -> 'a -> error
val error_of_printer_file : (Format.formatter -> 'a -> unit) -> 'a -> error
val error_of_exn : exn -> error option
val register_error_of_exn : (exn -> error option) -> unit
val report_error : Format.formatter -> error -> unit

val error_reporter : (Format.formatter -> error -> unit) Pervasives.ref
  Hook for intercepting error reports.

val default_error_reporter : Format.formatter -> error -> unit
  Original error reporter for use in hooks.

val report_exception : Format.formatter -> exn -> unit

22.4 Module Longident : Long identifiers, used in parsetree.

type t =
  | Lident of string
  | Ldot of t * string
  | Lapply of t * t
val flatten : t -> string list
val last : t -> string
val parse : string -> t

22.5 Module Parse : Entry points in the parser

val implementation : Lexing.lexbuf -> Parsetree.structure
val interface : Lexing.lexbuf -> Parsetree.signature
val toplevel_phrase : Lexing.lexbuf -> Parsetree.toplevel_phrase
val use_file : Lexing.lexbuf -> Parsetree.toplevel_phrase list
val core_type : Lexing.lexbuf -> Parsetree.core_type
val expression : Lexing.lexbuf -> Parsetree.expression
val pattern : Lexing.lexbuf -> Parsetree.pattern

22.6 Module Parsetree: Abstract syntax tree produced by parsing

type constant =
  | Pconst_integer of string * char option
  | Pconst_char of char
  | Pconst_string of string * string option
  | Pconst_float of string * char option

22.6.1 Extension points

type attribute = string Asttypes.loc * payload

type extension = string Asttypes.loc * payload


type attributes = attribute list

type payload =
  | PStr of structure
  | PSig of signature
  | PTyp of core_type
  | PPat of pattern * expression option

22.6.2 Core language

type core_type =

{  ptyp_desc : core_type_desc ;
    ptyp_loc : Location.t ;
    ptyp_attributes : attributes ;
}


type core_type_desc =

  | Ptyp_any    | Ptyp_var of string
  | Ptyp_arrow of Asttypes.arg_label * core_type * core_type
  | Ptyp_tuple of core_type list
  | Ptyp_constr of Longident.t Asttypes.loc * core_type list
  | Ptyp_object of (string * attributes * core_type) list
    * Asttypes.closed_flag
  | Ptyp_class of Longident.t Asttypes.loc * core_type list
  | Ptyp_alias of core_type * string
  | Ptyp_variant of row_field list * Asttypes.closed_flag * Asttypes.label list option
  | Ptyp_poly of string list * core_type
  | Ptyp_package of package_type
| Ptyp_extension of extension

type package_type = Longident.t Asttypes.loc *
  (Longident.t Asttypes.loc * core_type) list

type row_field =
  | Rtag of Asttypes.label * attributes * bool * core_type list
  | Rinherit of core_type

type pattern =
  { ppat_desc : pattern_desc ;
    ppat_loc : Location.t ;
    ppat_attributes : attributes ;
  }

type pattern_desc =
  | Ppat_any | Ppat_var of string Asttypes.loc
  | Ppat_alias of pattern * string Asttypes.loc
  | Ppat_constant of constant
  | Ppat_interval of constant * constant
  | Ppat_tuple of pattern list
  | Ppat_construct of Longident.t Asttypes.loc * pattern option
  | Ppat_variant of Asttypes.label * pattern option
  | Ppat_record of (Longident.t Asttypes.loc * pattern) list * Asttypes.closed_flag
  | Ppat_array of pattern list
  | Ppat_or of pattern * pattern
  | Ppat_constraint of pattern * core_type
  | Ppat_type of Longident.t Asttypes.loc
  | Ppat_lazy of pattern
  | Ppat_unpack of string Asttypes.loc
  | Ppat_exception of pattern
  | Ppat_extension of extension
  | Ppat_open of Longident.t Asttypes.loc * pattern

type expression =
  { pexp_desc : expression_desc ;
    pexp_loc : Location.t ;
    pexp_attributes : attributes ;
  }

type expression_desc =
  | Pexp_ident of Longident.t Asttypes.loc
  | Pexp_constant of constant
  | Pexp_let of Asttypes.rec_flag * value_binding list * expression
  | Pexp_function of case list
  | Pexp_fun of Asttypes.arg_label * expression option * pattern
  | Pexp_type of Longident.t Asttypes.loc
  | Pexp_lazy of pattern
  | Pexp_unpack of string Asttypes.loc
  | Pexp_match of expression * case list
  | Pexp_try of expression * case list
Chapter 22. The compiler front-end

| Pexp_tuple of expression list |
| Pexp_construct of Longident.t Asttypes.loc * expression option |
| Pexp_variant of Asttypes.label * expression option |
| Pexp_record of (Longident.t Asttypes.loc * expression) list |
| Pexp_field of expression * Longident.t Asttypes.loc |
| Pexp_setfield of expression * Longident.t Asttypes.loc * expression |
| Pexp_array of expression list |
| Pexp_ifthenelse of expression * expression * expression option |
| Pexp_sequence of expression * expression |
| Pexp_while of expression * expression |
| Pexp_for of pattern * expression * expression |
| Pexp_setfield of expression * Asttypes.direction_flag * expression |
| Pexp_constraint of expression * core_type |
| Pexp_coerce of expression * core_type option * core_type |
| Pexp_send of expression * string |
| Pexp_new of Longident.t Asttypes.loc |
| Pexp_setinstvar of string Asttypes.loc * expression |
| Pexp_override of (string Asttypes.loc * expression) list |
| Pexp_letmodule of string Asttypes.loc * module_expr * expression |
| Pexp_letexception of extension_constructor * expression |
| Pexp_assert of expression |
| Pexp_lazy of expression |
| Pexp_poly of expression * core_type option |
| Pexp_object of class_structure |
| Pexp_newtype of string * expression |
| Pexp_pack of module_expr |
| Pexp_open of Asttypes.override_flag * Longident.t Asttypes.loc * expression |
| Pexp_extension of extension |

type case =
{ pc_lhs : pattern ;
  pc_guard : expression option ;
  pc_rhs : expression ;
}

type value_description =
{ pval_name : string Asttypes.loc ;
  pval_type : core_type ;
  pval_prim : string list ;
  pval_attributes : attributes ;
  pval_loc : Location.t ;
}

type type_declaration =
{ ptype_name : string Asttypes.loc ;
  ptype_params : (core_type * Asttypes.variance) list ;

ptype_cstrs : (core_type * core_type * Location.t) list ;
ptype_kind : type_kind ;
ptype_private : Asttypes.private_flag ;
ptype_manifest : core_type option ;
ptype_attributes : attributes ;
ptype_loc : Location.t ;

{ }
type type_kind =
| Ptype_abstract | Ptype_variant of constructor_declaration list
| Ptype_record of label_declaration list
type label_declaration =
{ pld_name : string Asttypes.loc ;
pld Mutable : Asttypes.mutable_flag ;
pld_type : core_type ;
pld_loc : Location.t ;
pld attributes : attributes ;
}
type constructor_declaration =
{ pcd_name : string Asttypes.loc ;
pcd_args : constructor_arguments ;
pcd_res : core_type option ;
pcd_loc : Location.t ;
pced attributes : attributes ;
}
type constructor_arguments =
| Pcstr_tuple of core_type list
| Pcstr_record of label_declaration list
type type_extension =
{ ptyext_path : Longident.t Asttypes.loc ;
ptyext_params : (core_type * Asttypes.variance) list ;
ptyext_constructors : extension_constructor list ;
ptyext_private : Asttypes.private_flag ;
ptyext attributes : attributes ;
}
type extension_constructor =
{ pext_name : string Asttypes.loc ;
pext_kind : extension_constructor_kind ;
pext_loc : Location.t ;
pext_attributes : attributes ;
}
type extension_constructor_kind =
| Pext_decl of constructor_arguments * core_type option
| Pext_rebind of Longident.t Asttypes.loc
22.6.3 Class language

type class_type =
{  pctype_desc : class_type_desc ;
  pctype_loc : Location.t ;
  pctype_attributes : attributes ;
} ;
type class_type_desc =
|  Pcty_constr of Longident.t Asttypes.loc * core_type list
|  Pcty_signature of class_signature
|  Pcty_arrow of Asttypes.arg_label * core_type * class_type
|  Pcty_extension of extension

type class_signature =
{  pcsig_self : core_type ;
  pcsig_fields : class_type_field list ;
} ;
type class_type_field =
{  pctf_desc : class_type_field_desc ;
  pctf_loc : Location.t ;
  pctf_attributes : attributes ;
} ;
type class_type_field_desc =
|  Pctf_inherit of class_type
|  Pctf_val of (string * Asttypes.mutable_flag * Asttypes.virtual_flag * core_type)
|  Pctf_method of (string * Asttypes.private_flag * Asttypes.virtual_flag * core_type)
|  Pctf_constraint of (core_type * core_type)
|  Pctf_attribute of attribute
|  Pctf_extension of extension

type 'a class_infos =
{  pci_virt : Asttypes.virtual_flag ;
  pci_params : (core_type * Asttypes.variance) list ;
  pci_name : string Asttypes.loc ;
  pci_expr : 'a ;
  pci_loc : Location.t ;
  pci_attributes : attributes ;
} ;
type class_description = class_type class_infos

type class_type_declaration = class_type class_infos

type class_expr =
{  pcl_desc : class_expr_desc ;
  pcl_loc : Location.t ;
  pcl_attributes : attributes ;
} ;
type class_expr_desc =
| Pcl_constr of Longident.t Asttypes.loc * core_type list
| Pcl_structure of class_structure
| Pcl_fun of Asttypes.arg_label * expression option * pattern
  * class_expr
| Pcl_apply of class_expr * (Asttypes.arg_label * expression) list
| Pcl_let of Asttypes.rec_flag * value_binding list * class_expr
| Pcl_constraint of class_expr * class_type
| Pcl_extension of extension

type class_structure =
{ pcstr_self : pattern ;
  pcstr_fields : class_field list ;
}

type class_field =
{ pcf_desc : class_field_desc ;
  pcf_loc : Location.t ;
  pcf_attributes : attributes ;
}

type class_field_desc =
| Pcf_inherit of Asttypes.override_flag * class_expr * string option
| Pcf_val of (string Asttypes.loc * Asttypes.mutable_flag * class_field_kind)
| Pcf_method of (string Asttypes.loc * Asttypes.private_flag * class_field_kind)
| Pcf_constraint of (core_type * core_type)
| Pcf_initializer of expression
| Pcf_attribute of attribute
| Pcf_extension of extension

type class_field_kind =
| Cfk_virtual of core_type
| Cfk_concrete of Asttypes.override_flag * expression

type class_declaration = class_expr class_infos

22.6.4 Module language

type module_type =
{ pmty_desc : module_type_desc ;
  pmty_loc : Location.t ;
  pmty_attributes : attributes ;
}

type module_type_desc =
| Pmty_ident of Longident.t Asttypes.loc
| Pmty_signature of signature
| Pmty_functor of string Asttypes.loc * module_type option * module_type
| Pmty_with of module_type * with_constraint list
| Pmty_typeof of module_expr
| Pmty_extension of extension
type signature = signature_item list

type signature_item =
  { psig_desc : signature_item_desc ;
    psig_loc : Location.t ;
  }

type signature_item_desc =
  | Psig_value of value_description
  | Psig_type of Asttypes.rec_flag * type_declaration list
  | Psig_typext of type_extension
  | Psig_exception of extension_constructor
  | Psig_module of module_declaration
  | Psig_recmodule of module_declaration list
  | Psig_modtype of module_type_declaration
  | Psig_open of open_description
  | Psig_include of include_description
  | Psig_class of class_description list
  | Psig_class_type of class_type_declaration list
  | Psig_attribute of attribute
  | Psig_extension of extension * attributes

type module_declaration =
  { pmd_name : string Asttypes.loc ;
    pmd_type : module_type ;
    pmd_attributes : attributes ;
    pmd_loc : Location.t ;
  }

type module_type_declaration =
  { pmtd_name : string Asttypes.loc ;
    pmtd_type : module_type option ;
    pmtd_attributes : attributes ;
    pmtd_loc : Location.t ;
  }

type open_description =
  { popen_lid : Longident.t Asttypes.loc ;
    popen_override : Asttypes.override_flag ;
    popen_loc : Location.t ;
    popen_attributes : attributes ;
  }

type 'a include_infos =
  { pincl_mod : 'a ;
    pincl_loc : Location.t ;
    pincl_attributes : attributes ;
  }

type include_description = module_type include_infos
type include_declaration = module_expr include_infos

type with_constraint =
  | Pwith_type of Longident.t Asttypes.loc * type_declaration
  | Pwith_module of Longident.t Asttypes.loc * Longident.t Asttypes.loc
  | Pwith_typesubst of type_declaration
  | Pwith_modsubst of string Asttypes.loc * Longident.t Asttypes.loc

type module_expr =
  { pmod_desc : module_expr_desc ;
    pmod_loc : Location.t ;
    pmod_attributes : attributes ;
  }

type module_expr_desc =
  | Pmod_ident of Longident.t Asttypes.loc
  | Pmod_structure of structure
  | Pmod_functor of string Asttypes.loc * module_type option * module_expr
  | Pmod_apply of module_expr * module_expr
  | Pmod_constraint of module_expr * module_type
  | Pmod_unpack of expression
  | Pmod_extension of extension

type structure = structure_item list

type structure_item =
  { pstr_desc : structure_item_desc ;
    pstr_loc : Location.t ;
  }

type structure_item_desc =
  | Pstr_eval of expression * attributes
  | Pstr_value of Asttypes.rec_flag * value_binding list
  | Pstr_primitive of value_description
  | Pstr_type of Asttypes.rec_flag * type_declaration list
  | Pstr_typext of type_extension
  | Pstr_exception of extension_constructor
  | Pstr_module of module_binding
  | Pstr_recmodule of module_binding list
  | Pstr_modtype of module_type_declaration
  | Pstr_open of open_description
  | Pstr_class of class_declaration list
  | Pstr_class_type of class_type_declaration list
  | Pstr_include of include_declaration
  | Pstr_attribute of attribute
  | Pstr_extension of extension * attributes

type value_binding =
  { pvb_pat : pattern ;
    pvb_expr : expression ;
    pvb_attributes : attributes ;
  }
Chapter 22. The compiler front-end

pvb_loc : Location.t ;
}

module_binding =
{
  pmb_name : string Asttypes.loc ;
  pmb_expr : module_expr ;
  pmb_attributes : attributes ;
  pmb_loc : Location.t ;
}

22.6.5 Toplevel

toplevel_phrase =
  | Ptop_def of structure
  | Ptop_dir of string * directive_argument

directive_argument =
  | Pdir_none | Pdir_string of string
  | Pdir_int of string * char option
  | Pdir_ident of Longident.t
  | Pdir_bool of bool

22.7 Module Pprintast

type space_formatter = (unit, Format.formatter, unit) Pervasives.format

val toplevel_phrase : Format.formatter -> Parsetree.toplevel_phrase -> unit
val expression : Format.formatter -> Parsetree.expression -> unit
val string_of_expression : Parsetree.expression -> string
val top_phrase : Format.formatter -> Parsetree.toplevel_phrase -> unit
val core_type : Format.formatter -> Parsetree.core_type -> unit
val pattern : Format.formatter -> Parsetree.pattern -> unit
val signature : Format.formatter -> Parsetree.signature -> unit
val structure : Format.formatter -> Parsetree.structure -> unit
val string_of_structure : Parsetree.structure -> string
Chapter 23

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:

```bash
ocamlc other options unix.cma other files
ocamlopt other options unix.cmxa other files
```

For interactive use of the unix library, do:

```bash
ocamlmktop -o mytop unix.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type

```bash
#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.

23.1 Module Unix: Interface to the Unix system.

Note: all the functions of this module (except error_message and handle_unix_error) are liable to raise the Unix_error exception whenever the underlying system call signals an error.

Error report

```ocaml
type error =
   — E2BIG
```
Argument list too long
— EACCES  
  Permission denied
— EAGAIN  
  Resource temporarily unavailable; try again
— EBADF  
  Bad file descriptor
— EBUSY  
  Resource unavailable
— ECHILD  
  No child process
— EDEADLK  
  Resource deadlock would occur
— EDOM  
  Domain error for math functions, etc.
— EEXIST  
  File exists
— EFAULT  
  Bad address
— EFBIG  
  File too large
— EINTR  
  Function interrupted by signal
— EINVAL  
  Invalid argument
— EIO  
  Hardware I/O error
— EISDIR  
  Is a directory
— EMFILE  
  Too many open files by the process
— EMLINK  
  Too many links
— ENAMETOOLONG  
  Filename too long
— ENFILE  
  Too many open files in the system
— ENODEV  
  No such device
— ENOENT  
  No such file or directory
— ENOEXEC  
  Not an executable file
— ENOLCK  
  No locks available
— ENOMEM
  Not enough memory
— ENOSPC
  No space left on device
— ENOSYS
  Function not supported
— ENOTDIR
  Not a directory
— ENOTEMPTY
  Directory not empty
— ENOTTY
  Inappropriate I/O control operation
— ENXIO
  No such device or address
— EPERM
  Operation not permitted
— EPIPE
  Broken pipe
— ERANGE
  Result too large
— EROFS
  Read-only file system
— ESPIPE
  Invalid seek e.g. on a pipe
— ESRCH
  No such process
— EXDEV
  Invalid link
— EWOULDBLOCK
  Operation would block
— EINPROGRESS
  Operation now in progress
— EALREADY
  Operation already in progress
— ENOTSOCK
  Socket operation on non-socket
— EDESTADDRREQ
  Destination address required
— EMSGSIZE
  Message too long
— EPROTOTYPE
  Protocol wrong type for socket
— ENOPROTOOPT
  Protocol not available
— EPROTONOSUPPORT
Protocol not supported
— ESOCKTNOSUPPORT
   Socket type not supported
— EOPNOTSUPP
   Operation not supported on socket
— EPFNOSUPPORT
   Protocol family not supported
— EAFNOSUPPORT
   Address family not supported by protocol family
— EADDRINUSE
   Address already in use
— EADDRNOTAVAIL
   Can’t assign requested address
— ENETDOWN
   Network is down
— ENETUNREACH
   Network is unreachable
— ENETRESET
   Network dropped connection on reset
— ECONNABORTED
   Software caused connection abort
— ECONNRESET
   Connection reset by peer
— ENOBUS
   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
Chapter 23. The unix library: Unix system calls

| 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.
  Raises 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.

```ocaml
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.
```

```ocaml
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.

Raises Unix.Unix_error on failure.
```

```ocaml
val execve : string -> string array -> string array -> 'a

Same as Unix.execv, except that the third argument provides the environment to the program executed.
```

```ocaml
val execvp : string -> string array -> 'a

Same as Unix.execv, except that the program is searched in the path.
```

```ocaml
val execvpe : string -> string array -> string array -> 'a

Same as Unix.execve, except that the program is searched in the path.
```

```ocaml
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.

On Windows: not implemented, use Unix.create_process or threads.
```

```ocaml
val wait : unit -> int * process_status

Wait until one of the children processes die, and return its pid and termination status.

On Windows: Not implemented, use Unix.waitpid.
```

```ocaml
val waitpid : wait_flag list -> int -> int * process_status

Same as Unix.wait, 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, and whether it should report stopped children.

On Windows, this function can only wait for a given PID, not any child process.
```

```ocaml
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 (or the command interpreter cmd.exe on Windows) 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. On Windows: not implemented (because it is meaningless).

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. On Windows: not implemented.

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
The flags to `Unix.openfile[23.1]`.

The type of file access rights, e.g. `0o640` is read and write for user, read for group, none for others

```ocaml
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 (see `Unix.umask[23.1]`). Return a file descriptor on the named file.

```ocaml
val close : file_descr -> unit
```

Close a file descriptor.

```ocaml
val read : file_descr -> bytes -> int -> int -> int
```

`read fd buff ofs len` reads `len` bytes from descriptor `fd`, storing them in byte sequence `buff`, starting at position `ofs` in `buff`. Return the number of bytes actually read.

```ocaml
val write : file_descr -> bytes -> int -> int -> int
```

`write fd buff ofs len` writes `len` bytes to descriptor `fd`, taking them from byte sequence `buff`, starting at position `ofs` in `buff`. Return the number of bytes actually written. `write` repeats the writing operation until all bytes have been written or an error occurs.

```ocaml
val single_write : file_descr -> bytes -> int -> int -> int
```

Same as `write`, but attempts to write only once. Thus, if an error occurs, `single_write` guarantees that no data has been written.

```ocaml
val write_substring : file_descr -> string -> int -> int -> int
```

Same as `write`, but take the data from a string instead of a byte sequence.
val single_write_substring : file_descr -> string -> int -> int -> int
    Same as single_write, but take the data from a string instead of a byte sequence.

Interfacing with the standard input/output library

val in_channel_of_descr : file_descr -> Pervasives.in_channel
    Create an input channel reading from the given descriptor. The channel is initially in binary
    mode; use set_binary_mode_in ic false if text mode is desired. Text mode is supported
    only if the descriptor refers to a file or pipe, but is not supported if it refers to a socket. On
    Windows, set_binary_mode_in always fails on channels created with this function.
    Beware that channels are buffered so more characters may have been read from the file
    descriptor than those accessed using channel functions. Channels also keep a copy of the
    current position in the file.
    You need to explicitly close all channels created with this function. Closing the channel also
    closes the underlying file descriptor (unless it was already closed).

val out_channel_of_descr : file_descr -> Pervasives.out_channel
    Create an output channel writing on the given descriptor. The channel is initially in binary
    mode; use set_binary_mode_out oc false if text mode is desired. Text mode is supported
    only if the descriptor refers to a file or pipe, but is not supported if it refers to a socket. On
    Windows, set_binary_mode_out always fails on channels created with this function.
    Beware that channels are buffered so you may have to flush them to ensure that all data
    has been sent to the file descriptor. Channels also keep a copy of the current position in the
    file.
    You need to explicitly close all channels created with this function. Closing the channel
    flushes the data and closes the underlying file descriptor (unless it has already been closed,
    in which case the buffered data is lost).

val descr_of_in_channel : Pervasives.in_channel -> file_descr
    Return the descriptor corresponding to an input channel.

val descr_of_out_channel : Pervasives.out_channel -> file_descr
    Return the descriptor corresponding to an output channel.

Seeking and truncating

type seek_command =
    | SEEK_SET
    | SEEK_CUR
    | SEEK_END
    indicates positions relative to the beginning of the file
    indicates positions relative to the current position
    indicates positions relative to the end of the file
indicates positions relative to the end of the file

Positioning modes for Unix.lseek\[23.1\].

val lseek : file_descr -> int -> seek_command -> int
  Set the current position for a file descriptor, and return the resulting offset (from the
  beginning of the file).

val truncate : string -> int -> unit
  Truncates the named file to the given size.
  On Windows: not implemented.

val ftruncate : file_descr -> int -> unit
  Truncates the file corresponding to the given descriptor to the given size.
  On Windows: not implemented.

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_SOCKET
    Socket

type stats =
  { st_dev : int ;
    Device number
    st_ino : int ;
    Inode number
    st_kind : file_kind ;
    Kind of the file
    st_perm : file_perm ;
    Access rights
  }
Chapter 23. The unix library: Unix system calls

```plaintext
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 : int ;
    Size in bytes
st_atime : float ;
    Last access time
st_mtime : float ;
    Last modification time
st_ctime : float ;
    Last status change time
```

The information returned by the Unix.stat calls.

```plaintext
val stat : string -> stats
    Return the information for the named file.

val lstat : string -> stats
    Same as Unix.stat, but in case the file is a symbolic link, return the information for the link itself.

val fstat : file_descr -> stats
    Return the information for the file associated with the given descriptor.

val isatty : file_descr -> bool
    Return true if the given file descriptor refers to a terminal or console window, false otherwise.
```

File operations on large files

```plaintext
module LargeFile :
    sig
        val lseek : Unix.file_descr -> int64 -> Unix.seek_command -> int64
```
See Unix.lseek\[23.1\].

val truncate : string -> int64 -> unit

See Unix.truncate\[23.1\].

val ftruncate : Unix.file_descr -> int64 -> unit

See Unix.ftruncate\[23.1\].

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
Chapter 23. The unix library: Unix system calls

```ml
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\[23.1\] (for positioning a file descriptor), Unix.truncate\[23.1\] and Unix.ftruncate\[23.1\] (for changing the size of a file), and Unix.stat\[23.1\], Unix.lstat\[23.1\] and Unix.fstat\[23.1\] (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

```ml
val unlink : string -> unit
    Unlink operation.

val rename : string -> string -> unit
    Rename operation.

val link : string -> string -> unit
    Link operation.
```

File permissions and ownership

```ml
type access_permission =
  — R_OK
    Read permission
  — W_OK
    Write permission
  — X_OK
    Execution permission
  — F_OK
    File exists

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. On Windows: not implemented.
```
val chown : string -> int -> int -> unit
    Change the owner uid and owner gid of the named file. On Windows: not implemented
    (make no sense on a DOS file system).

val fchown : file_descr -> int -> int -> unit
    Change the owner uid and owner gid of an opened file. On Windows: not implemented
    (make no sense on a DOS file system).

val umask : int -> int
    Set the process's file mode creation mask, and return the previous mask. On Windows: not
    implemented.

val access : string -> access_permission list -> unit
    Check that the process has the given permissions over the named file.
    Raises Unix_error otherwise.
    On Windows, execute permission X_OK, cannot be tested, it just tests for read permission
    instead.

Operations on file descriptors

val dup : file_descr -> file_descr
    Return a new file descriptor referencing the same file as the given descriptor.

val dup2 : file_descr -> file_descr -> unit
    dup2 fd1 fd2 duplicates fd1 to fd2, closing fd2 if already opened.

val set_nonblock : file_descr -> unit
    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.

val clear_nonblock : file_descr -> unit
    Clear the “non-blocking” flag on the given descriptor. See Unix.set_nonblock[23.1].

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.

val clear_close_on_exec : file_descr -> unit
    Clear the “close-on-exec” flag on the given descriptor. See Unix.set_close_on_exec[23.1].
Directories

val mkdir : string -> file_perm -> unit
  Create a directory with the given permissions (see Unix.umask\[23.1\]).

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. On Windows: not implemented.

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 (see Unix.umask\[23.1\]). On Windows: not implemented.
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 (or cmd.exe on Windows), 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 -> Pervasives.in_channel * Pervasives.out_channel * Pervasives.in_channel
Similar to Unix.open_process\[23.1\], 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.

val close_process_in : Pervasives.in_channel -> process_status
Close channels opened by Unix.open_process_in\[23.1\], 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\[23.1\], 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\[23.1\], 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\[23.1\], wait for the associated command to terminate, and return its termination status.

Symbolic links

val symlink : ?to_dir:bool -> string -> string -> unit
symlink ?to_dir source dest creates the file dest as a symbolic link to the file source. On Windows, ~to_dir indicates if the symbolic link points to a directory or a file; if omitted, symlink examines source using stat and picks appropriately, if source does not exist then false is assumed (for this reason, it is recommended that the ~to_dir parameter be specified in new code). On Unix, ~to_dir ignored.

Windows symbolic links are available in Windows Vista onwards. There are some important differences between Windows symlinks and their POSIX counterparts.

Windows symbolic links come in two flavours: directory and regular, which designate whether the symbolic link points to a directory or a file. The type must be correct - a directory symlink which actually points to a file cannot be selected with chdir and a file symlink which actually points to a directory cannot be read or written (note that Cygwin’s emulation layer ignores this distinction).

When symbolic links are created to existing targets, this distinction doesn’t matter and symlink will automatically create the correct kind of symbolic link. The distinction matters when a symbolic link is created to a non-existent target.

The other caveat is that by default symbolic links are a privileged operation. Administrators will always need to be running elevated (or with UAC disabled) and by
default normal user accounts need to be granted the SeCreateSymbolicLinkPrivilege via
Local Security Policy (secpol.msc) or via Active Directory.

Unix.has_symlink can be used to check that a process is able to create symbolic links.

val has_symlink : unit -> bool
    Returns true if the user is able to create symbolic links. On Windows, this indicates that
the user not only has the SeCreateSymbolicLinkPrivilege but is also running elevated, if
necessary. On other platforms, this is simply indicates that the symlink system call is
available.

val readlink : string -> string
    Read the contents of a symbolic link.

Polling

val select :
  file_descr list ->
  file_descr list ->
  file_descr list ->
  float -> file_descr list * file_descr list * file_descr list
    Wait until some input/output operations become possible on some channels. The three list
arguments are, respectively, a set of descriptors to check for reading (first argument), for
writing (second argument), or for exceptional conditions (third argument). The fourth
argument is the maximal timeout, in seconds; a negative fourth argument means no timeout
(unbounded wait). The result is composed of three sets of descriptors: those ready for
reading (first component), ready for writing (second component), and over which an
exceptional condition is pending (third component).

Locking

type lock_command =
   — F_UNLOCK
       Unlock a region
   — F_LOCK
       Lock a region for writing, and block if already locked
   — F_TLOCK
       Lock a region for writing, or fail if already locked
   — F_TEST
       Test a region for other process locks
   — F_RLOCK
       Lock a region for reading, and block if already locked
   — F_TRLOCK
       Lock a region for reading, or fail if already locked
Commands for Unix.lockf[23.1].

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[23.1]), 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_UNLOCK 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[21.38] and Sys.set_signal[21.38].

val kill : int -> int -> unit

kill pid sig sends signal number sig to the process with id pid. On Windows, only the Sys.sigkill signal is emulated.

type sigprocmask_command =

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.
On Windows: not implemented (no inter-process signals on Windows).

val sigpending : unit -> int list

Return the set of blocked signals that are currently pending.
On Windows: not implemented (no inter-process signals on Windows).

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.
On Windows: not implemented (no inter-process signals on Windows).
val pause : unit -> unit
    Wait until a non-ignored, non-blocked signal is delivered.
    On Windows: not implemented (no inter-process signals on Windows).

Time functions

type process_times =
{  tms_utime : float ;
    User time for the process
    tms_stime : float ;
    System time for the process
    tms_cutime : float ;
    User time for the children processes
    tms_cstime : float ;
    System time for the children processes
}
    The execution times (CPU times) of a process.

type tm =
{  tm_sec : int ;
    Seconds 0..60
    tm_min : int ;
    Minutes 0..59
    tm_hour : int ;
    Hours 0..23
    tm_mday : int ;
    Day of month 1..31
    tm_mon : int ;
    Month of year 0..11
    tm_year : int ;
    Year - 1900
    tm_wday : int ;
    Day of week (Sunday is 0)
    tm_yday : int ;
    Day of year 0..365
    tm_isdst : bool ;

Daylight time savings in effect

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[23.1], but with resolution better than 1 second.

val gmtime : float -> tm
  Convert a time in seconds, as returned by Unix.time[23.1], 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[23.1], 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[23.1]. 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.
  On Windows: not implemented.

val sleep : int -> unit
  Stop execution for the given number of seconds.

val sleepf : float -> unit
  Stop execution for the given number of seconds. Like sleep, but fractions of seconds are supported.
  Since: 4.03.0

val times : unit -> process_times
  Return the execution times of the process. On Windows, it is partially implemented, will not report timings for child processes.

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. If both times are 0.0, the access and last modification times are both set to the current time.

```ocaml
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.

val getitimer : interval_timer -> interval_timer_status
val setitimer : interval_timer
  interval_timer_status
  interval_timer_status

User id, group id

val getuid : unit -> int
val geteuid : unit -> int
Return the effective user id under which the process runs. On Windows, always return 1.

```ocaml
val setuid : int -> unit
Set the real user id and effective user id for the process. On Windows: not implemented.
```

```ocaml
val getgid : unit -> int
Return the group id of the user executing the process. On Windows, always return 1.
```

```ocaml
val getegid : unit -> int
Return the effective group id under which the process runs. On Windows, always return 1.
```

```ocaml
val setgid : int -> unit
Set the real group id and effective group id for the process. On Windows: not implemented.
```

```ocaml
val getgroups : unit -> int array
Return the list of groups to which the user executing the process belongs. On Windows, always return [|1|].
```

```ocaml
val setgroups : int array -> unit
setgroups groups sets the supplementary group IDs for the calling process. Appropriate privileges are required. On Windows: not implemented.
```

```ocaml
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. On Windows: not implemented.
```

```ocaml
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.
```

```ocaml
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.
Raises Not_found if no such entry exist.
On Windows, always raise Not_found.

val getgrnam : string -> group_entry
Find an entry in group with the given name.
Raises Not_found if no such entry exist.
On Windows, always raise Not_found.

val getpwuid : int -> passwd_entry
Find an entry in passwd with the given user id.
Raises Not_found if no such entry exist.
On Windows, always raise Not_found.

val getgrgid : int -> group_entry
Find an entry in group with the given group id.
Raises Not_found if no such entry exist.
On Windows, always 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.

Raises 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 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
  | PF_INET
  | PF_INET6
  Unix domain
  Internet domain (IPv4)
  Internet domain (IPv6)
  The type of socket domains. Not all platforms support IPv6 sockets (type PF_INET6).
  Windows does not support PF_UNIX.

type socket_type =
  | SOCK_STREAM
  | SOCK_DGRAM
  | SOCK_RAW
  | SOCK_SEQPACKET
  Stream socket
  Datagram socket
  Raw socket
  Sequenced packets socket
  The type of socket kinds, specifying the semantics of communications. SOCK_SEQPACKET is included for completeness, but is rarely supported by the OS, and needs system calls that are not available in this library.

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

val shutdown : file_descr -> shutdown_command -> unit
  Shutdown a socket connection. SHUTDOWN_SEND as second argument causes reads on the
  other end of the connection to return an end-of-file condition. SHUTDOWN_RECEIVE causes
  writes on the other end of the connection to return a closed pipe condition (SIGPIPE signal).

val getsockname : file_descr -> sockaddr
  Return the address of the given socket.

val getpeername : file_descr -> sockaddr
Return the address of the host connected to the given socket.

type msg_flag =
    | MSG_OOB — MSG_DONTROUTE — MSG_PEEK
The flags for Unix.recv[23.1], Unix.recvfrom[23.1], Unix.send[23.1] and
    | Unix.sendto[23.1].

val recv : file_descr -> bytes -> int -> int -> msg_flag list -> int
    | Receive data from a connected socket.

val recvfrom : file_descr ->
    | bytes -> int -> int -> msg_flag list -> int * sockaddr
    | Receive data from an unconnected socket.

val send : file_descr -> bytes -> int -> int -> msg_flag list -> int
    | Send data over a connected socket.

val send_substring :
    | file_descr -> string -> int -> int -> msg_flag list -> int
    | Same as send, but take the data from a string instead of a byte sequence.

val sendto :
    | file_descr ->
    | bytes -> int -> int -> msg_flag list -> sockaddr -> int
    | Send data over an unconnected socket.

val sendto_substring :
    | file_descr ->
    | string -> int -> int -> msg_flag list -> sockaddr -> int
    | Same as sendto, but take the data from a string instead of a byte sequence.

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 `Unix.getsockopt` and modified with `Unix.setsockopt`. These options have a boolean (true/false) value.

```haskell
type socket_int_option =
  — SO_SNDBUF
    Size of send buffer
— SO_RCVBUF
    Size of received buffer
— SO_ERROR
  Deprecated. Use `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 `Unix.getsockopt_int` and modified with `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 `Unix.getsockopt_optint` and modified with `Unix.setsockopt_optint`. These options have a value of type `int option`, with 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 `Unix.getsockopt_float` and modified with `Unix.setsockopt_float`. These options have a floating-point value representing a time in seconds. The value 0 means infinite timeout.

```haskell
val getsockopt : file_descr -> socket_bool_option -> bool
Return the current status of a boolean-valued option in the given socket.

```ocaml
val setsockopt : file_descr -> socket_bool_option -> bool -> unit
```
Set or clear a boolean-valued option in the given socket.

```ocaml
val getsockopt_int : file_descr -> socket_int_option -> int
```
Same as `Unix.getsockopt` for an integer-valued socket option.

```ocaml
val setsockopt_int : file_descr -> socket_int_option -> int -> unit
```
Same as `Unix.setsockopt` for an integer-valued socket option.

```ocaml
val getsockopt_optint : file_descr -> socket_optint_option -> int option
```
Same as `Unix.getsockopt` for a socket option whose value is an `int` option.

```ocaml
val setsockopt_optint :
  file_descr -> socket_optint_option -> int option -> unit
```
Same as `Unix.setsockopt` for a socket option whose value is an `int` option.

```ocaml
val getsockopt_float : file_descr -> socket_float_option -> float
```
Same as `Unix.getsockopt` for a socket option whose value is a floating-point number.

```ocaml
val setsockopt_float : file_descr -> socket_float_option -> float -> unit
```
Same as `Unix.setsockopt` for a socket option whose value is a floating-point number.

```ocaml
val getsockopt_error : file_descr -> error option
```
Return the error condition associated with the given socket, and clear it.

**High-level network connection functions**

```ocaml
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` on the output channel at the right times to ensure correct synchronization.

```ocaml
val shutdown_connection : Pervasives.in_channel -> unit
```
“Shut down” a connection established with `Unix.open_connection`; that is, transmit an end-of-file condition to the server reading on the other side of the connection. This does not fully close the file descriptor associated with the channel, which you must remember to free via `Pervasives.close_in`.

```ocaml
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` never returns normally. On Windows, it is not implemented. Use threads.

**Host and protocol databases**

```ocaml
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.
  Raises `Not_found` if no such entry exist.

val gethostbyaddr : inet_addr -> host_entry
  Find an entry in `hosts` with the given address.
  Raises `Not_found` if no such entry exist.

val getprotobynumber : string -> protocol_entry
```
Chapter 23. The unix library: Unix system calls

Find an entry in protocols with the given name.

**Raises** *Not_found* if no such entry exist.

```ocaml
val getprotobynumber : int -> protocol_entry
                             Find an entry in protocols with the given protocol number.
                             **Raises** *Not_found* if no such entry exist.
```

```ocaml
val servbyport : int -> string -> service_entry
                             Find an entry in services with the given service number.
                             **Raises** *Not_found* if no such entry exist.
```

```ocaml
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* [23.1].
```

```ocaml
type getaddrinfo_option =
    | AI_FAMILY of socket_domain
      Impose the given socket domain
    | AI_SOCKTYPE of socket_type
      Impose the given socket type
    | AI_PROTOCOL of int
      Impose the given protocol
    — AI_NUMERICHOST
      Do not call name resolver, expect numeric IP address
    — AI_CANONNAME
```
Fill the ai_canonname field of the result
— AI_PASSIVE

Set address to “any” address for use with Unix.bind
Options to 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).

type name_info =
{ niHostname : string ;
  Name or IP address of host

  niService : string ;
  Name of service or port number
}

Host and service information returned by Unix.getnameinfo.

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.

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.

Raises 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.

type terminal_io =
{ mutable c_ignbrk : bool ;
  Ignore the break condition.

  mutable c_brkint : bool ;
  Signal interrupt on break condition.

  mutable c_ignpar : bool ;
  Ignore characters with parity errors.

  mutable c_parmrk : bool ;
  Mark parity errors.

  mutable c_inpck : bool ;
  Enable parity check on input.

  mutable c_istrip : bool ;
  Strip 8th bit on input characters.

  mutable c_inlcr : bool ;
  Map NL to CR on input.

  mutable c_igncr : bool ;
  Ignore CR on input.

  mutable c_icrnl : bool ;
  Map CR to NL on input.

  mutable c_ixon : bool ;
  Recognize XON/XOFF characters on input.

  mutable c_ixoff : bool ;
  Emit XON/XOFF chars to control input flow.

  mutable c_opost : bool ;
  Enable output processing.

  mutable c_obaud : int ;
Output baud rate (0 means close connection).

mutable c_ibaud : int;
Input baud rate.

mutable c_csize : int;
Number of bits per character (5-8).

mutable c_cstopb : int;
Number of stop bits (1-2).

mutable c_cread : bool;
Reception is enabled.

mutable c_parenb : bool;
Enable parity generation and detection.

mutable c_parodd : bool;
Specify odd parity instead of even.

mutable c_hupcl : bool;
Hang up on last close.

mutable c_clocal : bool;
Ignore modem status lines.

mutable c_isig : bool;
Generate signal on INTR, QUIT, SUSP.

mutable c_icanon : bool;
Enable canonical processing (line buffering and editing)

mutable c_noflsh : bool;
Disable flush after INTR, QUIT, SUSP.

mutable c_echo : bool;
Echo input characters.

mutable c_echoe : bool;
Echo ERASE (to erase previous character).

mutable c_echok : bool;
Echo KILL (to erase the current line).

mutable c_echonl : bool;
Echo NL even if c_echo is not set.

mutable c_vintr : char;
Interrupt character (usually ctrl-C).

mutable c_vquit : char;
Chapter 23. The unix library: Unix system calls

Quit character (usually ctrl-“).

mutable c_verase : char ;
Erase character (usually DEL or ctrl-H).

mutable c_vkkill : char ;
Kill line character (usually ctrl-U).

mutable c_veof : char ;
End-of-file character (usually ctrl-D).

mutable c_veol : char ;
Alternate end-of-line char. (usually none).

mutable c_vmin : int ;
Minimum number of characters to read before the read request is satisfied.

mutable c_vtime : int ;
Maximum read wait (in 0.1s units).

mutable c_vstart : char ;
Start character (usually ctrl-Q).

mutable c_vstop : char ;
Stop character (usually ctrl-S).

val tcgetattr : file_descr -> terminal_io
Return the status of the terminal referred to by the given file descriptor. On Windows, not implemented.

type setattr_when =
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 (TCSAFNULL). TCSADRAIN is recommended when changing the output parameters; TCSAFNULL, when changing the input parameters. On Windows, not implemented.

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). On Windows, not implemented.

val tcdrain : file_descr -> unit
Waits until all output written on the given file descriptor has been transmitted.
On Windows, not implemented.

type flush_queue =
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.
On Windows, not implemented.

type flow_action =
val tcflow : file_descr -> flow_action -> unit
Suspend or restart reception or transmission of data on the given file descriptor, depending on the second argument: TCOOFF suspends output, TCOON restarts output, TCIOFF transmits a STOP character to suspend input, and TCION transmits a START character to restart input.
On Windows, not implemented.

val setsid : unit -> int
Put the calling process in a new session and detach it from its controlling terminal.
On Windows, not implemented.

23.2 Module UnixLabels: labelized version of the interface
This module is identical to Unix [23.1], 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>getpid</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</td>
</tr>
<tr>
<td>link</td>
<td>implemented (since 3.02)</td>
</tr>
<tr>
<td>symlink, readlink</td>
<td>implemented (since 4.03.0)</td>
</tr>
<tr>
<td>access</td>
<td>execute permission <code>X_OK</code> cannot be tested, it just tests for read permission instead</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>mkfifo</td>
<td>not implemented</td>
</tr>
<tr>
<td>kill</td>
<td>partially implemented (since 4.00.0): only the <code>SIGKILL</code> signal is implemented</td>
</tr>
<tr>
<td>pause</td>
<td>not implemented (no inter-process signals in Windows)</td>
</tr>
<tr>
<td>alarm</td>
<td>not implemented</td>
</tr>
<tr>
<td>times</td>
<td>partially implemented, will not report timings for child processes</td>
</tr>
<tr>
<td>getitimer, setitimer</td>
<td>not implemented</td>
</tr>
<tr>
<td>getuid, geteuid, getgid, getegid</td>
<td>always return 1</td>
</tr>
<tr>
<td>getgroups</td>
<td>always returns $[11]$ (since 2.00)</td>
</tr>
<tr>
<td>setuid, setgid, setgroups</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 socket_domain</td>
<td><code>PF_INET</code> is fully supported; <code>PF_INET6</code> is fully supported (since 4.01.0); <code>PF_UNIX</code> is not supported</td>
</tr>
<tr>
<td>establish_server</td>
<td>not implemented; use threads</td>
</tr>
<tr>
<td>terminal functions (tc*)</td>
<td>not implemented</td>
</tr>
</tbody>
</table>
Chapter 24

The num library: arbitrary-precision rational arithmetic

The num library implements integer arithmetic and rational arithmetic in arbitrary precision.

More documentation on the functions provided in this library can be found in The CAML Numbers Reference Manual by Valérie Ménissier-Morain, technical report 141, INRIA, july 1992 (available electronically, http://hal.inria.fr/docs/00/07/00/27/PDF/RT-0141.pdf).

Programs that use the num library must be linked as follows:

ocamlc other options nums.cma other files
ocamlopt other options nums.cmxa other files

For interactive use of the nums library, do:

ocamlmktop -o mytop nums.cma
./mytop

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type

#load "nums.cma";;

24.1 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).

type num =
  | Int of int
  | Big_int of Big_int.big_int
  | Ratio of Ratio.ratio

 The type of numbers.
Arithmetic operations

`val (+/) : num -> num -> num`

Same as `Num.add_num` \[24.1\].

`val add_num : num -> num -> num`

Addition

`val minus_num : num -> num`

Unary negation.

`val (-/) : num -> num -> num`

Same as `Num.sub_num` \[24.1\].

`val sub_num : num -> num -> num`

Subtraction

`val ( */ ) : num -> num -> num`

Same as `Num.mult_num` \[24.1\].

`val mult_num : num -> num -> num`

Multiplication

`val square_num : num -> num`

Squaring

`val (//) : num -> num -> num`

Same as `Num.div_num` \[24.1\].

`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` \[24.1\].

`val power_num : num -> num -> num`

Exponentiation

`val abs_num : num -> num`
Absolute value.

val succ_num : num -> num
  succ n is \textit{n+1}

val pred_num : num -> num
  pred n is \textit{n-1}

val incr_num : num Pervasives.ref -> unit
  incr r is \textit{r:=r+1}, where \textit{r} is a reference to a number.

val decr_num : num Pervasives.ref -> unit
  decr r is \textit{r:=r-1}, where \textit{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[24.1].
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[24.1] 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. Raise Failure "num_of_string" if the given string is not a valid representation of an integer

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
24.2 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 * r <= 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 * b + r and 0 <= 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.

val num_bits_big_int : big_int -> int
  Return the number of significant bits in the absolute value of the given big integer.
  \[
  \text{num_bits_big_int } a = \begin{cases} 
  0 & \text{if } a = 0; \\
  \text{positive integer } n & \text{otherwise such that } 2^{(n-1)} \leq |a| < 2^n.
  \end{cases}
  \]
  Since: 4.03.0

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, \text{is_int_big_int } a \text{ returns true if and only if } a \text{ is between } 2^{30} \text{ and } 2^{30}-1. \text{ On a 64-bit platform, } is_int_big_int a \text{ returns true if and only if } a \text{ is between } -2^{62} \text{ 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 [Nativeint.min_int, 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.

24.3 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[24.3].

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[24.3].

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[24.3].

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[24.3].

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[24.3].
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 25

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:

```
ocamlc  other options  str.cma  other files
ocamlopt  other options  str.cmxa  other files
```

For interactive use of the str library, do:

```
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";;
```

25.1 Module Str : Regular expressions and high-level string processing

Regular expressions

type regexp

The type of compiled regular expressions.

val regexp : string -> regexp

Compile a regular expression. The 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 ‘n’ character.

• $ Matches at end of line: either at the end of the matched string, or just before a ‘n’ character.

• \| (infix) Alternative between two expressions.

• \(\ldots\) Grouping and naming of the enclosed expression.

• \1 The text matched by the first \(\ldots\) expression (\2 for the second expression, and so on up to \9).

• \b Matches word boundaries.

• \ Quotes word boundaries. The special characters are $\^\$.+?\[\].

Note: the argument to regexp is usually a string literal. In this case, any backslash character in the regular expression must be doubled to make it past the OCaml string parser. For example, the following expression:

```ocaml
let r = Str.regexp "hello \((A-Za-z)+\)" in
Str.replace_first r "\1" "hello world"
```

returns the string "world".

In particular, if you want a regular expression that matches a single backslash character, you need to quote it in the argument to regexp (according to the last item of the list above) by adding a second backslash. Then you need to quote both backslashes (according to the syntax of string constants in OCaml) by doubling them again, so you need to write four backslash characters: Str.regexp "\\\\".

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\[25.1\], 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.

Raises 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.

Raises Not_found if no substring matches.

val string_partial_match : regexp -> string -> int -> bool

Similar to Str.string_match[25.1], 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 last call to one of the following matching or searching functions:

- Str.string_match[25.1]
- Str.search_forward[25.1]
- Str.search_backward[25.1]
- Str.string_partial_match[25.1]
- Str.global_substitute[25.1]
- Str.substitute_first[25.1]

provided that none of the following functions was called inbetween:

- Str.global_replace[25.1]
- Str.replace_first[25.1]
- Str.split[25.1]
- Str.bounded_split[25.1]
- Str.split_delim[25.1]
- Str.bounded_split_delim[25.1]
• Str.full_split
• Str.bounded_full_split

Note: in the case of global_substitute and substitute_first, a call to matched_string is only valid within the subst argument, not after global_substitute or substitute_first returns.

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 the last call to a matching or searching function (see Str.matched_string for details).

val match_end : unit -> int

match_end() returns the position of the character following the last character of the substring that was matched by the last call to a matching or searching function (see Str.matched_string for details).

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 that was matched by the last call to a matching or searching function (see Str.matched_string for details). The user must make sure that the parameter s is the same string that was passed to the matching or searching function.

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 that was matched by the last call to a matching or searching function (see Str.matched_string for details).

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 n-th group of the regular expression that was matched by the last call to a matching or searching function (see Str.matched_string for details).

Raises
• \texttt{Not\_found} if the \texttt{n}th group of the regular expression was not matched.
• \texttt{Invalid\_argument} if there are fewer than \texttt{n} groups in the regular expression.

\textbf{Replacement}

\begin{verbatim}
val global_replace : regexp -> string -> string -> string
    global_replace regexp templ s returns a string identical to \texttt{s}, except that all substrings
    of \texttt{s} that match \texttt{regexp} have been replaced by \texttt{templ}. The replacement template \texttt{templ} can
    contain \texttt{\1}, \texttt{\2}, etc; these sequences will be replaced by the text matched by the
    corresponding group in the regular expression. \texttt{\0} stands for the text matched by the whole
    regular expression.
\end{verbatim}

\begin{verbatim}
val replace_first : regexp -> string -> string -> string
    Same as Str.global_replace[25.1], except that only the first substring matching the
    regular expression is replaced.
\end{verbatim}

\begin{verbatim}
val global_substitute : regexp -> (string -> string) -> string -> string
    global_substitute regexp subst s returns a string identical to \texttt{s}, except that all
    substrings of \texttt{s} that match \texttt{regexp} have been replaced by the result of function \texttt{subst}. The
    function \texttt{subst} is called once for each matching substring, and receives \texttt{s} (the whole text) as
    argument.
\end{verbatim}

\begin{verbatim}
val substitute_first : regexp -> (string -> string) -> string -> string
    Same as Str.global_substitute[25.1], except that only the first substring matching the
    regular expression is replaced.
\end{verbatim}

\begin{verbatim}
val replace_matched : string -> string -> string
    replace_matched repl s returns the replacement text \texttt{repl} in which \texttt{\1}, \texttt{\2}, etc. have
    been replaced by the text matched by the corresponding groups in the regular expression
    that was matched by the last call to a matching or searching function (see
    Str.matched_string[25.1] for details). \texttt{s} must be the same string that was passed to the
    matching or searching function.
\end{verbatim}

\textbf{Splitting}

\begin{verbatim}
val split : regexp -> string -> string list
    split r s splits \texttt{s} into substrings, taking as delimiters the substrings that match \texttt{r}, and
    returns the list of substrings. For instance, split (regexp "\[ \ t\]+") s splits \texttt{s} into
    blank-separated words. An occurrence of the delimiter at the beginning or at the end of the
    string is ignored.
\end{verbatim}

\begin{verbatim}
val bounded_split : regexp -> string -> int -> string list
\end{verbatim}
Same as `Str.split`[25.1], 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`[25.1] but splits into at most n substrings, where n is the extra integer parameter. For instance, `split_delim (regexp "\s") " 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`[25.1] but splits into at most n substrings, where n is the extra integer parameter. For instance, `bounded_split_delim (regexp "\s") " abc "` returns `[""; "abc"; "]`, while `bounded_split` with the same arguments returns `["abc"]`.

```type split_result =```
``| Text of string``
``| Delim of string``

```val full_split : regexp -> string -> split_result list```
Same as `Str.full_split`[25.1], 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_full_split`[25.1], 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`[25.1].

```val last_chars : string -> int -> string```
`last_chars s n` returns the last n characters of s.
Chapter 26

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:

```
  ocamlc -thread other options unix.cma threads.cma other files
  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:

```
  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).
26.1 Module Thread: Lightweight threads for Posix 1003.1c and Win32.

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 [26.1].
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[26.1].

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[26.1].
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.

26.2 Module Mutex : Locks for mutual exclusion.

Mutexes (mutual-exclusion locks) are used to implement critical sections and protect shared mutable data structures against concurrent accesses. The typical use is (if m is the mutex associated with the data structure D):

   Mutex.lock m;
   (* Critical section that operates over D *);
   Mutex.unlock m


type t
   The type of mutexes.

val create : unit -> t
   Return a new mutex.

val lock : t -> unit
   Lock the given mutex. Only one thread can have the mutex locked at any time. A thread that attempts to lock a mutex already locked by another thread will suspend until the other thread unlocks the mutex.

val try_lock : t -> bool
   Same as Mutex.lock[26.2], but does not suspend the calling thread if the mutex is already locked: just return false immediately in that case. If the mutex is unlocked, lock it and return true.

val unlock : t -> unit
   Unlock the given mutex. Other threads suspended trying to lock the mutex will restart.

26.3 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):

\[
\begin{align*}
\text{Mutex.lock } m; \\
\text{while (* some predicate P over D is not satisfied *) do} \\
& \quad \text{Condition.wait } c \ m \\
& \quad \text{done;} \\
& \quad (* \text{ Modify D *}) \\
& \quad \text{if (* the predicate P over D is now satisfied *) then Condition.signal } c; \\
\text{Mutex.unlock } m
\end{align*}
\]

```plaintext
type t
The type of condition variables.
val create : unit -> t
Return a new condition variable.
val wait : t -> Mutex.t -> unit
wait c m atomically unlocks the mutex \(m\) and suspends the calling process on the condition variable \(c\). The process will restart after the condition variable \(c\) has been signalled. The mutex \(m\) is locked again before \text{wait} returns.
val signal : t -> unit
signal c restarts one of the processes waiting on the condition variable \(c\).
val broadcast : t -> unit
broadcast c restarts all processes waiting on the condition variable \(c\).
```

26.4 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.

```plaintext
type 'a channel
The type of communication channels carrying values of type 'a.
val new_channel : unit -> 'a channel
Return a new channel.
type +'a event
The type of communication events returning a result of type 'a.
```
val send : 'a channel -> 'a -> unit event

send ch v returns the event consisting in sending the value v over the channel ch. The result value of this event is ()

val receive : 'a channel -> 'a event

receive ch returns the event consisting in receiving a value from the channel ch. The result value of this event is the value received.

val always : 'a -> 'a event

always v returns an event that is always ready for synchronization. The result value of this event is v.

val choose : 'a event list -> 'a event

choose evl returns the event that is the alternative of all the events in the list evl.

val wrap : 'a event -> ('a -> 'b) -> 'b event

wrap ev fn returns the event that performs the same communications as ev, then applies the post-processing function fn on the return value.

val 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.
26.5 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).

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 -> bytes -> int -> int -> int
val write : Unix.file_descr -> bytes -> int -> int -> int
val write_substring : Unix.file_descr -> string -> int -> int -> int

Input/output with timeout

val timed_read : Unix.file_descr -> bytes -> int -> int -> float -> int
    See ThreadUnix.timed_write.
val timed_write : Unix.file_descr -> bytes -> 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.
val timed_write_substring :
    Unix.file_descr -> string -> int -> int -> float -> int
    See ThreadUnix.timed_write.

Polling

val select :
    Unix.filedescr list ->
    Unix.filedescr list ->
    Unix.filedescr list ->
    float -> Unix.filedescr list * Unix.filedescr list * Unix.filedescr list
Pipes and redirections

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 -> bytes -> int -> int -> Unix.msg_flag list -> int
val recvfrom :
  Unix.file_descr ->
  bytes -> int -> int -> Unix.msg_flag list -> int * Unix.sockaddr
val send :
  Unix.file_descr -> bytes -> int -> int -> Unix.msg_flag list -> int
val send_substring :
  Unix.file_descr -> string -> int -> int -> Unix.msg_flag list -> int
val sendto :
  Unix.file_descr ->
  bytes -> int -> int -> Unix.msg_flag list -> Unix.sockaddr -> int
val sendto_substring :
  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 27

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 coordinates diagram]

### 27.1 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 [27.1].

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 [27.1].

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 [27.1] 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
**Chapter 27. The graphics library**

**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 a1 a2** 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[27.1].

**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.

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.

val text_size : string -> int * int
Return the dimensions of the given text, if it were drawn with the current font and size.

### Filling

val fill_rect : int -> int -> int -> int -> unit
fill_rect x y w h fills the rectangle with lower left corner at x,y, width w and height h, with the current color. Raise Invalid_argument if w or h is negative.

val fill_poly : (int * int) array -> unit
Fill the given polygon with the current color. The array contains the coordinates of the vertices of the polygon.

val fill_arc : int -> int -> int -> int -> int -> int -> unit
Fill an elliptical pie slice with the current color. The parameters are the same as for Graphics.draw_arc[27.1].

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[27.1].

val fill_circle : int -> int -> int -> unit
Fill a circle with the current color. The parameters are the same as for Graphics.draw_circle[27.1].

### Images

type image
The abstract type for images, in internal representation. Externally, images are represented as matrices of colors.

val transp : color
In matrices of colors, this color represents 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.

\[
\text{val make_image : color array array} \rightarrow \text{image}
\]

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 \text{Graphic_failure} is raised.

\[
\text{val dump_image : image} \rightarrow \text{color array array}
\]

Convert an image to a color matrix.

\[
\text{val draw_image : image} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{unit}
\]

Draw the given image with lower left corner at the given point.

\[
\text{val get_image : int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{image}
\]

Capture the contents of a rectangle on the screen as an image. The parameters are the same as for \text{Graphics.fill_rect[27.1]}.

\[
\text{val create_image : int} \rightarrow \text{int} \rightarrow \text{image}
\]

\text{create_image w h} returns a new image \( w \) pixels wide and \( h \) pixels tall, to be used in conjunction with \text{blit_image}. The initial image contents are random, except that no point is transparent.

\[
\text{val blit_image : image} \rightarrow \text{int} \rightarrow \text{int} \rightarrow \text{unit}
\]

\text{blit_image img x y} copies screen pixels into the image \text{img}, modifying \text{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 \text{img} are left unchanged.

Mouse and keyboard events

\[
\text{type status} = \\
\{ \text{mouse_x : int} ; \\
\quad \text{X coordinate of the mouse} \\
\text{mouse_y : int} ; \\
\quad \text{Y coordinate of the mouse} \\
\text{button : bool} ; \\
\quad \text{true if a mouse button is pressed} \\
\text{keypressed : bool} ; \\
\quad \text{true if a key has been pressed} \\
\text{key : char} ;
\}
\]
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.

val loop_at_exit : event list -> (status -> unit) -> unit

Loop before exiting the program, the list given as argument is the list of handlers and the
events on which these handlers are called. To exit cleanly the loop, the handler should raise
Exit. Any other exception will be propagated outside of the loop.

Since: 4.01

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.
Chapter 27. The graphics library

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 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 above).
    Default remember mode is on.
Chapter 28

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.

28.1 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

```ocaml
val allow_only : string list -> unit
  allow_only units restricts the compilation units that dynamically-linked units can reference: it forbids all references to units other than those named in the list units. References to any other compilation unit will cause a `Unavailable_unit` error during `loadfile` or `loadfile_private`.

Initially (or after calling `default_available_units`) all compilation units composing the program currently running are available for reference from dynamically-linked units. `allow_only` can be used to restrict access to a subset of these units, 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. If `allow_only` is called several times, access will be restricted to the intersection of the given lists (i.e. a call to `allow_only` can never increase the set of available units).

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

```ocaml
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\[28.1\], 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 OCaml standard library directory.

val clear_available_units : unit -> unit
  Empty the list of compilation units accessible to dynamically-linked programs.

**Deprecated, initialization**

val init : unit -> unit
  *Deprecated.* Initialize the Dynlink library. This function is called automatically when needed.

**Error reporting**

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.

val error_message : error -> string
  Convert an error description to a printable message.
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 OCaml arrays described in section 21.2. The main differences between “big arrays” and standard OCaml arrays are as follows:

- Big arrays are not limited in size, unlike OCaml 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, OCaml 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 OCaml arrays can contain arbitrary OCaml 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 OCaml 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 OCaml 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 OCaml 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 OCaml ad-hoc polymorphic operations:

- comparisons (`=`, `<>`, `<=`, etc, as well as `Pervasives.compare`);
- hashing (module `Hash`);
- and structured input-output (the functions from the `Marshal` module, as well as `Pervasives.output_value` and `Pervasives.input_value`).

#### Element kinds

Big arrays can contain elements of the following kinds:

- IEEE single precision (32 bits) floating-point numbers (`Bigarray.float32_elt`),
- IEEE double precision (64 bits) floating-point numbers (`Bigarray.float64_elt`),
- IEEE single precision (2 * 32 bits) floating-point complex numbers (`Bigarray.complex32_elt`),
- IEEE double precision (2 * 64 bits) floating-point complex numbers (`Bigarray.complex64_elt`),
- 8-bit integers (signed or unsigned) (`Bigarray.int8_signed_elt` or `Bigarray.int8_unsigned_elt`),
- 16-bit integers (signed or unsigned) (`Bigarray.int16_signed_elt` or `Bigarray.int16_unsigned_elt`),
- OCaml integers (signed, 31 bits on 32-bit architectures, 63 bits on 64-bit architectures) (`Bigarray.int_elt`),
- 32-bit signed integer (`Bigarray.int32_elt`),
- 64-bit signed integers (`Bigarray.int64_elt`),
- platform-native signed integers (32 bits on 32-bit architectures, 64 bits on 64-bit architectures) (`Bigarray.nativeint_elt`).

Each element kind is represented at the type level by one of the *_elt types defined below (defined with a single constructor instead of abstract types for technical injectivity reasons).

```ocaml
type float32_elt =
        type float64_elt =
```
type int8_signed_elt =
type int8_unsigned_elt =
type int16_signed_elt =
type int16_unsigned_elt =
type int32_elt =
type int64_elt =
type int_elt =
type nativeint_elt =
type complex32_elt =
type complex64_elt =
type ('a, 'b) kind =
  | Float32 : -> (float, float32_elt) kind
  | Float64 : -> (float, float64_elt) kind
  | Int8_signed : -> (int, int8_signed_elt) kind
  | Int8_unsigned : -> (int, int8_unsigned_elt) kind
  | Int16_signed : -> (int, int16_signed_elt) kind
  | Int16_unsigned : -> (int, int16_unsigned_elt) kind
  | Int32 : -> (int32, int32_elt) kind
  | Int64 : -> (int64, int64_elt) kind
  | Int : -> (int, int_elt) kind
  | Nativeint : -> (nativeint, nativeint_elt) kind
  | Complex32 : -> (Complex.t, complex32_elt) kind
  | Complex64 : -> (Complex.t, complex64_elt) kind
  | Char : -> (char, int8_unsigned_elt) kind

To each element kind is associated an OCaml type, which is the type of OCaml 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 OCaml uses the OCaml type float, which is 64-bit double precision floats.
The GADT type ('a, 'b) kind captures this association of an OCaml 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. Its constructors list all possible associations of OCaml types with
element kinds, and are re-exported below for backward-compatibility reasons.

Using a generalized algebraic datatype (GADT) here allows to write well-typed polymorphic
functions whose return type depend on the argument type, such as:

let zero : type a b. (a, b) kind -> a = function
  | Float32 -> 0.0 | Complex32 -> Complex.zero
  | Float64 -> 0.0 | Complex64 -> Complex.zero
  | Int8_signed -> 0 | Int8_unsigned -> 0
  | Int16_signed -> 0 | Int16_unsigned -> 0
  | Int32 -> 0l | Int64 -> 0L
| Int -> 0 | Nativeint -> 0n |
| Char -> '\000'

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 OCaml type `float`. Big arrays of complex kinds `complex32_elt`, `complex64_elt` are accessed with the OCaml type `Complex.t`. Big arrays of integer kinds are accessed using the smallest OCaml integer type large enough to represent the array elements: `int` for 8- and 16-bit integer bigarrays, as well as OCaml-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`.

val kind_size_in_bytes : ('a, 'b) kind -> int
    kind_size_in_bytes k is the number of bytes used to store an element of type k.
    Since: 4.03.0

Array layouts

type c_layout =
    See Bigarray.fortran_layout.

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 phantom types Bigarray.c_layout and Bigarray.fortran_layout respectively.

Supported layouts

The GADT type `'a layout` represents one of the two supported memory layouts: C-style or Fortran-style. Its constructors are re-exported as values below for backward-compatibility reasons.

type 'a layout =
    | C_layout : -> c_layout layout
    | Fortran_layout : -> fortran_layout layout

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 OCaml 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 OCaml 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.

    val num_dims : ('a, 'b, 'c) t -> int

        Return the number of dimensions of the given big array.

    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.
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.

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 change_layout : ('a, 'b, 'c) t ->
'd Bigarray.layout -> ('a, 'b, 'd) t

Genarray.change_layout a layout returns a bigarray with the specified layout,
sharing the data with a (and hence having the same dimensions as a). No copying of
elements is involved: the new array and the original array share the same storage
space. The dimensions are reversed, such that get v [1| a; b |] in C layout becomes
get v [ | b+1; a+1 |] in Fortran layout.

Since: 4.04.0

val size_in_bytes : ('a, 'b, 'c) t -> int

size_in_bytes a is the number of elements in a multiplied by a's
Bigarray.kind_size_in_bytes[29.1].

Since: 4.03.0

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.)

val set : ('a, 'b, 'c) t -> int array -> 'a -> unit
Assign an element of a generic big array. \texttt{Genarray.set a \[|i1; \ldots; iN|\] v} stores the value \(v\) in the element of \(a\) whose coordinates are \(i_1\) in the first dimension, \(i_2\) in the second dimension, \ldots, \(i_N\) 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 \texttt{Genarray.get}; otherwise, \texttt{Invalid_argument} is raised. If \(N > 3\), alternate syntax is provided: you can write \(a\{|i_1, i_2, \ldots, i_N\} <- v\) instead of \texttt{Genarray.set a \[|i1; \ldots; iN|\] v}. (The syntax \(a\{|\ldots\} <- v\) with one, two or three coordinates is reserved for updating one-, two- and three-dimensional arrays as described below.)

\begin{verbatim}
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.
  \texttt{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 \([ofs \ldots ofs + len - 1]\) 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 \(a\).

  \texttt{Genarray.sub_left} applies only to big arrays in C layout. Raise \texttt{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. \texttt{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 \([ofs \ldots ofs + len - 1]\) 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 \([|i1; \ldots; iN+ofs|]\) of the
  original array \(a\).

  \texttt{Genarray.sub_right} applies only to big arrays in Fortran layout. Raise
  \texttt{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
\end{verbatim}
Chapter 29. The bigarray library

Extract a sub-array of lower dimension from the given big array by fixing one or several of the first (left-most) coordinates. \texttt{Genarray.slice_left a \[|i_1; \ldots; i_M|\]} returns the 'slice' of a obtained by setting the first \(M\) coordinates to \(i_1, \ldots, i_M\). If a has \(N\) dimensions, the slice has dimension \(N - M\), and the element at coordinates \([|j_1; \ldots; j(N-M)|]\) in the slice is identical to the element at coordinates \([|i_1; \ldots; i_M; j_1; \ldots; j(N-M)|]\) in the original array a. No copying of elements is involved: the slice and the original array share the same storage space.

\texttt{Genarray.slice_left} applies only to big arrays in C layout. Raise \texttt{Invalid_argument} if \(M \geq N\), or if \([|i_1; \ldots; i_M|]\) is outside the bounds of a.

\texttt{val slice_right :}
\texttt{
('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. \texttt{Genarray.slice_right a \[|i_1; \ldots; i_M|\]} 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 \([|j_1; \ldots; j(N-M)|]\) in the slice is identical to the element at coordinates \([|j_1; \ldots; j(N-M); i_1; \ldots; i_M|]\) in the original array a. No copying of elements is involved: the slice and the original array share the same storage space.

\texttt{Genarray.slice_right} applies only to big arrays in Fortran layout. Raise \texttt{Invalid_argument} if \(M \geq N\), or if \([|i_1; \ldots; i_M|]\) is outside the bounds of a.

\texttt{val blit :}
\texttt{
('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
}

Copy all elements of a big array in another big array. \texttt{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 \texttt{Genarray.blit} to sub-array or slices of src and dst.

\texttt{val fill :}
\texttt{
('a, 'b, 'c) t -> 'a -> unit
}

Set all elements of a big array to a given value. \texttt{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 \texttt{Genarray.fill} to a sub-array or a slice of \(a\).

\texttt{val map_file :}
\texttt{
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. \texttt{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 \texttt{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. Array accesses are bounds-checked, but the bounds are determined by the initial call to map_file. Therefore, you should make sure no other process modifies the mapped file while you're accessing it, or a SIGBUS signal may be raised. This happens, for instance, if the file is shrunk.

This function raises Sys_error in the case of any errors from the underlying system calls. Invalid_argument or Failure may be raised in cases where argument validation fails.

end

One-dimensional arrays

module Array1 :
  sig

  type ('a, 'b, 'c) t

  The type of one-dimensional big arrays whose elements have OCaml 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 size_in_bytes : ('a, 'b, 'c) t -> int
size_in_bytes a is the number of elements in a multiplied by a’s
Bigarray.kind_size_in_bytes[29.1].
Since: 4.03.0

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 for more details.

val unsafe_get : ('a, 'b, 'c) t -> int -> 'a

  Like Bigarray.Array1.get, 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, 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, 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 OCaml 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.

    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 size_in_bytes : ('a, 'b, 'c) t -> int

size_in_bytes a is the number of elements in a multiplied by a’s
Bigarray.kind_size_in_bytes

Since: 4.03.0

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; 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; 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 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 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

val slice_right :
('a, 'b, Bigarray.fortran_layout) t ->
int -> ('a, 'b, Bigarray.fortran_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\[29.1\] for more details. Array2.slice_left applies only to arrays with C layout.

\[
\text{val slice_right : ('a, 'b, Bigarray.fortran_layout) t -> int -> ('a, 'b, Bigarray.fortran_layout) Bigarray.Array1.t}
\]

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\[29.1\] for more details. Array2.slice_right applies only to arrays with Fortran layout.

\[
\text{val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit}
\]

Copy the first big array to the second big array. See Bigarray.Genarray.blit\[29.1\] for more details.

\[
\text{val fill : ('a, 'b, 'c) t -> 'a -> unit}
\]

Fill the given big array with the given value. See Bigarray.Genarray.fill\[29.1\] for more details.

\[
\text{val of_array : ('a, 'b) Bigarray.kind -> 'c Bigarray.layout -> 'a array array -> ('a, 'b, 'c) t}
\]

Build a two-dimensional big array initialized from the given array of arrays.

\[
\text{val map_file : Unix.file_descr -> ?pos:int64 -> ('a, 'b) Bigarray.kind -> 'c Bigarray.layout -> 'a array array -> ('a, 'b, 'c) t}
\]

Memory mapping of a file as a two-dimensional big array. See Bigarray.Genarray.map_file\[29.1\] for more details.

\[
\text{val unsafe_get : ('a, 'b, 'c) t -> int -> int -> 'a}
\]

Like Bigarray.Array2.get\[29.1\], but bounds checking is not always performed.

\[
\text{val unsafe_set : ('a, 'b, 'c) t -> int -> int -> 'a -> unit}
\]

Like Bigarray.Array2.set\[29.1\], but bounds checking is not always performed.

Two-dimensional arrays. The Array2 structure provides operations similar to those of Bigarray.Genarray\[29.1\], 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 OCaml type 'a, representation kind 'b, and memory layout 'c.

    val create :
      ('a, 'b) Bigarray.kind ->
      'c Bigarray.layout -> int -> int -> int -> ('a, 'b, 'c) t
    Array3.create kind layout dim1 dim2 dim3 returns a new bigarray of three dimension, whose size is dim1 in the first dimension, dim2 in the second dimension, and dim3 in the third. 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 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 size_in_bytes : ('a, 'b, 'c) t -> int
    size_in_bytes a is the number of elements in a multiplied by a’s Bigarray.kind_size_in_bytes\[29.1\].
    Since: 4.03.0

    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.

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 \texttt{Bigarray.Genarray.slice\_right} for more details. \texttt{Array3.slice\_right\_2} applies only to arrays with Fortran layout.

\begin{verbatim}
val blit : ('a, 'b, 'c) t -> ('a, 'b, 'c) t -> unit
Copy the first big array to the second big array. See \texttt{Bigarray.Genarray.blit} for more details.

val fill : ('a, 'b, 'c) t -> 'a -> unit
Fill the given big array with the given value. See \texttt{Bigarray.Genarray.fill} 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 \texttt{Bigarray.Genarray.map\_file} for more details.

val unsafe\_get : ('a, 'b, 'c) t -> int -> int -> int -> 'a
Like \texttt{Bigarray.Array3.get}, but bounds checking is not always performed.

val unsafe\_set : ('a, 'b, 'c) t -> int -> int -> int -> 'a -> unit
Like \texttt{Bigarray.Array3.set}, but bounds checking is not always performed.
\end{verbatim}

Three-dimensional arrays. The \texttt{Array3} structure provides operations similar to those of \texttt{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.

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` for reshaping to two-dimensional arrays.

\[
\text{val reshape}_3 : \\
('a, 'b, 'c) Genarray.t \to \\
\text{int} \to \text{int} \to \text{int} \to ('a, 'b, 'c) Array3.t
\]

Specialized version of `Bigarray.reshape` for reshaping to three-dimensional arrays.

### 29.2 Big arrays in the OCaml-C interface

C stub code that interface C or Fortran code with OCaml 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 an OCaml bigarray from C or Fortran

If \( v \) is a OCaml value representing a big array, the expression `Caml_ba_data_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 OCaml big array can be consulted from C as follows:

<table>
<thead>
<tr>
<th>C expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td><code>Caml_ba_array_val(v)-&gt;num_dims</code></td>
<td>number of dimensions</td>
</tr>
<tr>
<td><code>Caml_ba_array_val(v)-&gt;dim[i]</code></td>
<td>( i )-th dimension</td>
</tr>
<tr>
<td><code>Caml_ba_array_val(v)-&gt;flags &amp; BIGARRAY_KIND_MASK</code></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><code>CAML_BA_FLOAT32</code></td>
<td>32-bit single-precision floats</td>
</tr>
<tr>
<td><code>CAML_BA_FLOAT64</code></td>
<td>64-bit double-precision floats</td>
</tr>
<tr>
<td><code>CAML_BA_SINT8</code></td>
<td>8-bit signed integers</td>
</tr>
<tr>
<td><code>CAML_BA_UINT8</code></td>
<td>8-bit unsigned integers</td>
</tr>
<tr>
<td><code>CAML_BA_SINT16</code></td>
<td>16-bit signed integers</td>
</tr>
<tr>
<td><code>CAML_BA_UINT16</code></td>
<td>16-bit unsigned integers</td>
</tr>
<tr>
<td><code>CAML_BA_INT32</code></td>
<td>32-bit signed integers</td>
</tr>
<tr>
<td><code>CAML_BA_INT64</code></td>
<td>64-bit signed integers</td>
</tr>
<tr>
<td><code>CAML_BA_CAML_INT</code></td>
<td>31- or 63-bit signed integers</td>
</tr>
<tr>
<td><code>CAML_BA_NATIVE_INT</code></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.
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 = Caml_ba_array_val(bigarray)->dim[0];
    int dimy = Caml_ba_array_val(bigarray)->dim[1];
    /* C passes scalar parameters by value */
    my_c_function(Caml_ba_data_val(bigarray), dimx, dimy);
    /* Fortran passes all parameters by reference */
    my_fortran_function_(Caml_ba_data_val(bigarray), &dimx, &dimy);
    return Val_unit;
}

29.2.3 Wrapping a C or Fortran array as an OCaml big array

A pointer \( p \) to an already-allocated C or Fortran array can be wrapped and returned to OCaml as a big array using the \texttt{caml_ba_alloc} or \texttt{caml_ba_alloc_dims} functions.

- \texttt{caml_ba_alloc(kind | layout, numdims, p, dims)}
  
  Return an OCaml big array wrapping the data pointed to by \( p \). \texttt{kind} is the kind of array elements (one of the \texttt{CAML_BA_kind} constants above). \texttt{layout} is \texttt{CAML_BA_C_LAYOUT} for an array with C layout and \texttt{CAML_BA_FORTRAN_LAYOUT} for an array with Fortran layout. \texttt{numdims} is the number of dimensions in the array. \texttt{dims} is an array of \texttt{numdims} long integers, giving the sizes of the array in each dimension.

- \texttt{caml_ba_alloc_dims(kind | layout, numdims, p, (long) dim1, (long) dim2, \ldots, (long) dim_{numdims})}
  
  Same as \texttt{caml_ba_alloc}, 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 OCaml.

```ocaml
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 caml_ba_alloc(CAML_BA_NATIVE_INT | CAML_BA_C_LAYOUT,
                         2, my_c_array, dims);
}

value caml_get_fortran_array(value unit)
```

{ 
    return caml_ba_alloc_dims(CAML_BA_FLOAT32 | CAML_BA_FORTRAN_LAYOUT, 
        2, my_fortran_array_, 300L, 400L);
}

Part V

Appendix
Index to the library

( * ), 358
( */ ), 576
( ** ), 360
( **/ ), 576
( * . ), 360
(&), 356
(&&), 356
( @ ), 358
(@@), 356
(!), 372
(!=), 372
( = ), 354
( %= ), 354
(>=), 355
(=/.), 577
(’), 363
(’’), 374
(’’’), 374
( > ), 355
(>=/), 577
(>/), 577
(<), 355
(<=/), 577
(<>) , 355
(<= ), 355
(abs), 358
( abs_big_int ), 359
( abs_float ), 361
( abs_num ), 359
( absname ), 358
( absolute_path ), 359
( accept ), 359
( access ), 359
( access_permission ), 359
( acos ), 360
( adapt_filename ), 360
(add), 358
(add_available_units), 359
(add_big_int), 360
(add_buffer), 360
(add_bytes), 360
(add_channel), 360
(add_char), 360
(add_interfaces), 360
(add_int_big_int), 360
(add_num), 360
(add_ppx_context_sig), 360
INDEX TO THE LIBRARY

dump, 514
dump_image, 605
dup, 550
dup2, 550
Dynlink, 609
echo_eof, 525
elements, 466, 495
e lt, 465, 493
Empty, 479, 499
empty, 389, 453, 464, 465, 493, 501
enable_runtime_warnings, 512
end_of_input, 486
End_of_file, 353
environment, 541
Ephemeron, 400
eprintf, 422, 478
epsilon_float, 363
eq_big_int, 580
eq_num, 517
er rror_formatter, 418
Error, 500, 525, 611
error, 525, 541, 611
error_message, 541, 611
error_of_exn, 526
error_of_printer, 526
error_of_printer_file, 526
error_reporter, 526
errorf, 526
escaped, 391, 396, 504
establish_server, 565
Event, 595
event, 395, 609
executable_name, 507
execv, 542, 597
execve, 542, 597
execvp, 542, 597
execvpe, 542
exists, 385, 452, 457, 463, 466, 495
exists2, 452
Exit, 354
exit, 374, 592
exn, 352
exn_slot_id, 476
exn_slot_name, 476
exp, 360, 398
expml, 360
expression, 527, 528, 535
expression_desc, 529
extend, 390
extension, 408, 527
extension_constructor, 530
extension_constructor_kind, 530
extension_of_error, 522
extern_flags, 459
extract_big_int, 583
Failure, 353, 500
failwith, 354
fast_sort, 386, 454
fchmod, 549
fchown, 550
file, 399
file_descr, 543
file_exists, 507
file_kind, 546
filename, 484
file_perm, 544
Filename, 408
fill, 384, 390, 503, 515, 621, 623, 626, 629
fill_arc, 604
fill_circle, 604
fill_ellipse, 604
fill_poly, 604
fill_rect, 604
filter, 452, 457, 464, 466, 495
filter_map_inplace, 434, 437, 438, 462, 463
finalise, 429
finalise_last, 430
finalise_release, 431
find, 433, 437, 438, 452, 458, 462, 463, 465, 466, 496, 516
find_all, 433, 437, 438, 452, 458, 462, 463, 465, 466, 496, 516
first_chars, 590
flatten, 450, 526
float, 352, 362, 481, 482
float_of_big_int, 582
float_of_bits, 442, 445
get_formatter_output_functions, 416
get_formatter_tag_functions, 418
get_image, 605
get_key, 402, 403, 407
get_key_copy, 402, 405
get_key1, 403
get_key1_copy, 403
get_key2, 404
get_key2_copy, 404
get_margin, 413
get_mark_tags, 416
get_max_boxes, 414
get_max_indent, 413
get_minor_free, 429
get_normalize_ratio, 583
get_normalize_ratio_when_printing, 583
get_pos_info, 524
get_print_tags, 416
get_raw_backtrace, 472
get_raw_backtrace_next_slot, 475
get_raw_backtrace_slot, 475
get_state, 482
get_temp_dir_name, 410
getaddrinfo, 568
getaddrinfo_option, 568
getcwd, 507, 551
getegid, 559
getenv, 507, 541
geteuid, 558
getgid, 559
getgrgid, 560
getgrnam, 560
getgroups, 559
gethostbyaddr, 566
gethostbyname, 566
gethostname, 566
getitimer, 558
getlogin, 560
getnameinfo, 568
getnameinfo_option, 568
getpeereid, 562
getpid, 543
getppid, 543
getprotobynumber, 567
getpwnam, 560
getpwuid, 560
getservbyname, 567
getservbyport, 567
getsockname, 562
getsockopt, 564
getsockopt_error, 563
getsockopt_float, 565
getsockopt_int, 565
getsockopt_optint, 565
gettimeofday, 557
getuid, 558
global_replace, 589
global_substitute, 589
gmtime, 557
Graphic_failure, 600
Graphics, 600
green, 601
group_beginning, 588
group_end, 588
group_entry, 559
gt_big_int, 580
gt_num, 578
guard, 596
handle_unix_error, 541
has_symlink, 554
hash, 407, 436, 437, 439, 464, 514
hash_param, 439, 464
HashedType, 436, 462
Hashbl, 432, 461
hd, 449
header_size, 461
Help, 381
highlight_locations, 525
host_entry, 566
huge_fallback_count, 429
hypot, 361
i, 397
id, 469, 592
ifprintf, 423, 478
ignore, 364
ikfprintf, 423, 478
image, 604
implementation, 526
INDEX TO THE LIBRARY

kill, 555, 592
kind, 615, 619, 623, 625, 627
kind_size_in_bytes, 617
Kn, 405
kprintf, 423, 479
kscanf, 491
ksprintf, 423, 479
ksscanf, 491
label, 523
label_declaration, 530
LargeFile, 372, 547
last, 526
last_chars, 590
layout, 617, 619, 623, 625, 627
Lazy, 445
Lazy (module), 141
lazy_from_fun, 446
lazy_from_val, 447
lazy_is_val, 447
lazy_t, 352
ldexp, 362
le_big_int, 580
le_num, 578
length, 382, 387, 389, 394, 397, 398, 402, 409, 412
lexbuf, 448
lexeme, 448
lexeme_char, 448
lexeme_end, 449
lexeme_end_p, 449
lexeme_start, 448
lexeme_start_p, 449
Lexing, 447
lineto, 602
link, 549
linking_error, 611
List, 549, 500
list, 352, 496
listen, 502
lnot, 359
loadfile, 609
loadfile_private, 609
loc, 524, 525
localtime, 557
Location, 524
location, 474
lock, 594
lock_command, 554
lockf, 555
log, 360, 398
log10, 360
log1p, 360
logand, 441, 443, 468
lognot, 441, 444, 468
logor, 441, 443, 468
logxor, 441, 444, 468
Longident, 526
loop_at_exit, 606
lowercase, 392, 396, 505
lowercase_ascii, 393, 396, 505
lseek, 546, 547
lstat, 547, 549
lt_big_int, 580
lt_num, 577
magenta, 602
major, 428
major_slice, 428
Make, 403, 405, 406, 407, 438, 463, 465, 466
make, 382, 389, 482, 502
make_float, 383
make_formatter, 419
make_image, 605
make_leaf, 239
make_lexer, 432
make_matrix, 383
make_node, 239
make_self_init, 482
MakeSeeded, 403, 405, 407, 438, 463
Map, 454, 464
map, 384, 391, 450, 458, 465, 466, 495, 503
map_file, 621, 624, 626, 629
map_opt, 522
map_tree, 239
map2, 385, 451
mapi, 384, 391, 450, 458, 465, 503
mapper, 521
Marshal, 459
INDEX TO THE LIBRARY

num_of_string, 578
ocaml_version, 512
of_array, 623, 626, 629
of_bytes, 501
of_channel, 501
of_char, 513
of_float, 441, 444, 468
of_int, 441, 444, 468, 513
of_int32, 444, 468
of_list, 384, 466, 496, 500
of_nativeint, 445
of_string, 390, 441, 445, 469, 500
one, 397, 440, 442, 466
0, 469
opaque_identity, 512
open_box, 411
open_connection, 565, 598
open_description, 533
open_flag, 367, 544
open_graph, 600
open_hbox, 414
open_hovbox, 414
open_hvbox, 414
open_in, 370, 485
open_in_gen, 370
open_module, 239
open_out, 367
open_out_bin, 367
open_out_gen, 368
open_process, 552, 598
open_process_full, 552
open_process_in, 552, 598
open_process_out, 552, 598
open_tag, 415
open_tbox, 424
open_temp_file, 409
open_vbox, 414
opendir, 551
openfile, 544
option, 552
or_big_int, 582
OrderedType, 455, 464, 465, 493
os_type, 508
out_channel, 365
out_channel_length, 369, 372
out_channel_of_descr, 545
Out_of_memory, 353
output, 368, 399
output_binary_int, 368
output_buffer, 388
output_byte, 368
output_bytes, 368
output_char, 368
output_string, 368
output_substring, 368
output_value, 369
over_max_boxes, 414
override_flag, 523
package_type, 528
parent_dir_name, 408
Parse, 526
parse, 380, 526
parse_argv, 381
parse_argv_dynamic, 381
parse_dynamic, 381
Parse_error, 470
Parsetree, 527
Parsing, 470
partition, 452, 457, 465, 466, 495
passwd_entry, 559
pattern, 527, 528, 535
pattern_desc, 528
pause, 556
payload, 527
peek, 480, 501
Pervasives, 354
pipe, 531, 598
plot, 602
plots, 602
point_color, 602
polar, 398
poll, 596
pop, 480, 499
pos_in, 371, 372
pos_out, 369, 372
position, 447
pow, 398
<table>
<thead>
<tr>
<th>Function</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>print_newline</td>
<td>366, 413</td>
</tr>
<tr>
<td>print_raw_backtrace</td>
<td>472</td>
</tr>
<tr>
<td>print_space</td>
<td>412</td>
</tr>
<tr>
<td>print_stat</td>
<td>429</td>
</tr>
<tr>
<td>print_string</td>
<td>365, 412</td>
</tr>
<tr>
<td>print_tab</td>
<td>425</td>
</tr>
<tr>
<td>print_tbreak</td>
<td>424</td>
</tr>
<tr>
<td>print_warning</td>
<td>525</td>
</tr>
<tr>
<td>Printexc</td>
<td>471</td>
</tr>
<tr>
<td>printf</td>
<td>476</td>
</tr>
<tr>
<td>private_flag</td>
<td>523</td>
</tr>
<tr>
<td>process_status</td>
<td>542</td>
</tr>
<tr>
<td>process_times</td>
<td>556</td>
</tr>
<tr>
<td>prohibit</td>
<td>610</td>
</tr>
<tr>
<td>protocol_entry</td>
<td>566</td>
</tr>
<tr>
<td>push</td>
<td>479, 499</td>
</tr>
<tr>
<td>putenv</td>
<td>541</td>
</tr>
<tr>
<td>Queue</td>
<td>479</td>
</tr>
<tr>
<td>quick_stat</td>
<td>428</td>
</tr>
<tr>
<td>quo_num</td>
<td>576</td>
</tr>
<tr>
<td>quomod_big_int</td>
<td>580</td>
</tr>
<tr>
<td>quote</td>
<td>410, 586</td>
</tr>
<tr>
<td>raise</td>
<td>354</td>
</tr>
<tr>
<td>raise_errorf</td>
<td>526</td>
</tr>
<tr>
<td>raise_notrace</td>
<td>354</td>
</tr>
<tr>
<td>Random</td>
<td>480</td>
</tr>
<tr>
<td>randomize</td>
<td>433, 462</td>
</tr>
<tr>
<td>ratio_of_num</td>
<td>578</td>
</tr>
<tr>
<td>raw_backtrace</td>
<td>472</td>
</tr>
<tr>
<td>raw_backtrace_length</td>
<td>475</td>
</tr>
<tr>
<td>raw_backtrace_slot</td>
<td>475</td>
</tr>
<tr>
<td>raw_backtrace_to_string</td>
<td>473</td>
</tr>
<tr>
<td>rcontains_from</td>
<td>392, 503</td>
</tr>
<tr>
<td>read</td>
<td>544, 597</td>
</tr>
<tr>
<td>read_float</td>
<td>367</td>
</tr>
<tr>
<td>read_int</td>
<td>367</td>
</tr>
<tr>
<td>read_key</td>
<td>606</td>
</tr>
<tr>
<td>read_line</td>
<td>367</td>
</tr>
<tr>
<td>readdir</td>
<td>507, 551</td>
</tr>
<tr>
<td>readlink</td>
<td>554</td>
</tr>
<tr>
<td>really_input</td>
<td>370</td>
</tr>
<tr>
<td>really_input_string</td>
<td>370</td>
</tr>
<tr>
<td>rec_flag</td>
<td>523</td>
</tr>
<tr>
<td>receive</td>
<td>596</td>
</tr>
<tr>
<td>record_backtrace</td>
<td>471</td>
</tr>
<tr>
<td>recv</td>
<td>503, 598</td>
</tr>
<tr>
<td>recvfrom</td>
<td>503, 598</td>
</tr>
<tr>
<td>red</td>
<td>601</td>
</tr>
<tr>
<td>ref</td>
<td>372</td>
</tr>
<tr>
<td>regexp</td>
<td>585</td>
</tr>
<tr>
<td>regexp_case_fold</td>
<td>586</td>
</tr>
<tr>
<td>regexp_string</td>
<td>586</td>
</tr>
<tr>
<td>regexp_string_case_fold</td>
<td>586</td>
</tr>
<tr>
<td>register</td>
<td>395, 522</td>
</tr>
<tr>
<td>register_error_of_exn</td>
<td>526</td>
</tr>
<tr>
<td>register_exception</td>
<td>396</td>
</tr>
<tr>
<td>register_function</td>
<td>522</td>
</tr>
<tr>
<td>register_printer</td>
<td>472</td>
</tr>
<tr>
<td>rem</td>
<td>440, 443, 467</td>
</tr>
<tr>
<td>remember_mode</td>
<td>607</td>
</tr>
<tr>
<td>remove</td>
<td>433, 437, 456, 462, 465, 494, 507, 516</td>
</tr>
<tr>
<td>remove_assoc</td>
<td>453</td>
</tr>
<tr>
<td>remove_assq</td>
<td>453</td>
</tr>
<tr>
<td>remove_extension</td>
<td>408</td>
</tr>
<tr>
<td>rename</td>
<td>507, 549</td>
</tr>
<tr>
<td>replace</td>
<td>433, 437, 438, 462, 463</td>
</tr>
<tr>
<td>replace_first</td>
<td>589</td>
</tr>
<tr>
<td>replace_matched</td>
<td>589</td>
</tr>
<tr>
<td>report_error</td>
<td>526</td>
</tr>
<tr>
<td>report_exception</td>
<td>526</td>
</tr>
<tr>
<td>reset</td>
<td>387, 433, 437, 458, 462, 463, 525</td>
</tr>
<tr>
<td>reshape</td>
<td>630</td>
</tr>
<tr>
<td>reshape_1</td>
<td>630</td>
</tr>
<tr>
<td>reshape_2</td>
<td>630</td>
</tr>
<tr>
<td>reshape_3</td>
<td>631</td>
</tr>
<tr>
<td>resize_window</td>
<td>601</td>
</tr>
<tr>
<td>result</td>
<td>373</td>
</tr>
<tr>
<td>rev</td>
<td>450</td>
</tr>
<tr>
<td>rev_append</td>
<td>450</td>
</tr>
<tr>
<td>rev_map</td>
<td>451</td>
</tr>
<tr>
<td>rev_map2</td>
<td>451</td>
</tr>
<tr>
<td>rewinddir</td>
<td>551</td>
</tr>
<tr>
<td>rgb</td>
<td>601</td>
</tr>
<tr>
<td>rhs_end</td>
<td>479</td>
</tr>
<tr>
<td>rhs_end_pos</td>
<td>479</td>
</tr>
<tr>
<td>rhs_loc</td>
<td>523</td>
</tr>
<tr>
<td>rhs_start</td>
<td>479</td>
</tr>
</tbody>
</table>
rhs_start_pos, 470
rindex, 391, 504
rindex_from, 392, 504
rlineto, 602
rmdir, 551
rmoveto, 602
round_num, 577
row_field, 528
run_main, 522
runtime_parameters, 509
runtime_variant, 509
runtime_warnings_enabled, 512
S, 400, 436, 455, 462, 464, 465, 493, 515
save_and_close, 347, 498
save_event, 347, 498
save_event_for_automatic_snapshots, 348, 498
Scan_failure, 486
scanbuf, 489
Scann, 483
scans, 491
scanner, 486
Scanning, 484
search_backward, 587
search_forward, 587
seeded_hash, 439, 464
seeded_hash_param, 439, 464
SeededHashedType, 437, 462
SeededS, 401, 438, 463
seek_command, 546
seek_in, 371, 372
seek_out, 369, 372
select, 594, 593, 596, 597
self, 592
self_init, 481
send, 563, 596, 598
send_substring, 563, 598
sendto, 563, 598
sendto_substring, 563, 598
Series, 347, 498
service_entry, 566
Set, 385, 592
set, 382, 389, 428, 502, 515, 619, 623, 625, 627
set_all_formatter_output_functions, 423
set_approx_printing, 583
set_binary_mode_in, 371
set_binary_mode_out, 369
set_close_on_exec, 550
set_color, 501
set_cookie, 523
set_data, 402, 404, 406
set_ellipsis_text, 415
set_error_when_null_denominator, 583
set_float_precision, 584
set_font, 604
set_formatter_out_channel, 416
set_formatter_out_functions, 417
set_formatter_output_functions, 416
set_formatter_tag_functions, 417
set_key, 402, 405
set_key_data, 407
set_key1, 403
set_key2, 404
set_line_width, 603
set_margin, 413
set_mark_tags, 416
set_max_boxes, 414
set_max_indent, 413
set_nonblock, 550
set_normalize_ratio, 583
set_normalize_ratio_when_printing, 583
set_print_tags, 416
set_signal, 509
set_state, 482
set_tab, 425
set_tags, 416
set_temp_dir_name, 410
set_temp_dir_name, 571
setattr, 571
setattr, 470
set_uncaught_exception_handler, 473
set_window_title, 600
setattr_when, 571
setgid, 559
setgroups, 559
setitimer, 558
setsid, 572
setsockopt, 565
<table>
<thead>
<tr>
<th>term</th>
<th>page</th>
</tr>
</thead>
<tbody>
<tr>
<td>setsockopt_float</td>
<td>565</td>
</tr>
<tr>
<td>setsockopt_int</td>
<td>565</td>
</tr>
<tr>
<td>setsockopt_optint</td>
<td>565</td>
</tr>
<tr>
<td>setuid</td>
<td>559</td>
</tr>
<tr>
<td>shift_left</td>
<td>441</td>
</tr>
<tr>
<td>shift_left_big_int</td>
<td>582</td>
</tr>
<tr>
<td>shift_right</td>
<td>441</td>
</tr>
<tr>
<td>shift_right_big_int</td>
<td>582</td>
</tr>
<tr>
<td>shift_right_towards_zero_big_int</td>
<td>582</td>
</tr>
<tr>
<td>show_filename</td>
<td>525</td>
</tr>
<tr>
<td>shutdown</td>
<td>562</td>
</tr>
<tr>
<td>shutdown_command</td>
<td>562</td>
</tr>
<tr>
<td>shutdown_connection</td>
<td>565</td>
</tr>
<tr>
<td>sigabrt</td>
<td>510</td>
</tr>
<tr>
<td>sigalrm</td>
<td>510</td>
</tr>
<tr>
<td>sigbus</td>
<td>511</td>
</tr>
<tr>
<td>sigchld</td>
<td>510</td>
</tr>
<tr>
<td>sigcont</td>
<td>510</td>
</tr>
<tr>
<td>sigfpe</td>
<td>510</td>
</tr>
<tr>
<td>sighup</td>
<td>510</td>
</tr>
<tr>
<td>sigill</td>
<td>510</td>
</tr>
<tr>
<td>sigint</td>
<td>510</td>
</tr>
<tr>
<td>sigkill</td>
<td>510</td>
</tr>
<tr>
<td>sigmask</td>
<td>593</td>
</tr>
<tr>
<td>sign_big_int</td>
<td>580</td>
</tr>
<tr>
<td>sign_num</td>
<td>577</td>
</tr>
<tr>
<td>signal</td>
<td>509</td>
</tr>
<tr>
<td>signal_behavior</td>
<td>509</td>
</tr>
<tr>
<td>signature</td>
<td>533</td>
</tr>
<tr>
<td>signature_item</td>
<td>533</td>
</tr>
<tr>
<td>signature_item_desc</td>
<td>533</td>
</tr>
<tr>
<td>sigpending</td>
<td>555</td>
</tr>
<tr>
<td>sigpipe</td>
<td>510</td>
</tr>
<tr>
<td>sigpoll</td>
<td>511</td>
</tr>
<tr>
<td>sigprocmask</td>
<td>555</td>
</tr>
<tr>
<td>sigprocmask_command</td>
<td>555</td>
</tr>
<tr>
<td>sigprof</td>
<td>511</td>
</tr>
<tr>
<td>sigquit</td>
<td>510</td>
</tr>
<tr>
<td>sigsev</td>
<td>510</td>
</tr>
<tr>
<td>sigstop</td>
<td>511</td>
</tr>
<tr>
<td>sigsuspend</td>
<td>555</td>
</tr>
<tr>
<td>sigsys</td>
<td>511</td>
</tr>
<tr>
<td>sigterm</td>
<td>510</td>
</tr>
<tr>
<td>sigtrap</td>
<td>511</td>
</tr>
<tr>
<td>sigtstp</td>
<td>511</td>
</tr>
<tr>
<td>sigttin</td>
<td>511</td>
</tr>
<tr>
<td>sigttou</td>
<td>511</td>
</tr>
<tr>
<td>sigurg</td>
<td>511</td>
</tr>
<tr>
<td>sigusr1</td>
<td>510</td>
</tr>
<tr>
<td>sigusr2</td>
<td>510</td>
</tr>
<tr>
<td>sigvtalrm</td>
<td>511</td>
</tr>
<tr>
<td>sigxcpu</td>
<td>511</td>
</tr>
<tr>
<td>sigxfsz</td>
<td>512</td>
</tr>
<tr>
<td>sin</td>
<td>360</td>
</tr>
<tr>
<td>single_write</td>
<td>544</td>
</tr>
<tr>
<td>single_write_substring</td>
<td>545</td>
</tr>
<tr>
<td>singleton</td>
<td>456</td>
</tr>
<tr>
<td>sinh</td>
<td>361</td>
</tr>
<tr>
<td>size</td>
<td>467</td>
</tr>
<tr>
<td>size_in_bytes</td>
<td>619</td>
</tr>
<tr>
<td>size_x</td>
<td>601</td>
</tr>
<tr>
<td>size_y</td>
<td>601</td>
</tr>
<tr>
<td>sleep</td>
<td>557</td>
</tr>
<tr>
<td>sleepf</td>
<td>557</td>
</tr>
<tr>
<td>slice_left</td>
<td>620</td>
</tr>
<tr>
<td>slice_left_1</td>
<td>628</td>
</tr>
<tr>
<td>slice_left_2</td>
<td>628</td>
</tr>
<tr>
<td>slice_right</td>
<td>621</td>
</tr>
<tr>
<td>slice_right_1</td>
<td>628</td>
</tr>
<tr>
<td>slice_right_2</td>
<td>628</td>
</tr>
<tr>
<td>Slot</td>
<td>474</td>
</tr>
<tr>
<td>Snapshot</td>
<td>347</td>
</tr>
<tr>
<td>snd</td>
<td>365</td>
</tr>
<tr>
<td>sockaddr</td>
<td>562</td>
</tr>
<tr>
<td>socket</td>
<td>562</td>
</tr>
<tr>
<td>socket_bool_option</td>
<td>564</td>
</tr>
<tr>
<td>socket_domain</td>
<td>561</td>
</tr>
<tr>
<td>socket_float_option</td>
<td>564</td>
</tr>
<tr>
<td>socket_int_option</td>
<td>564</td>
</tr>
<tr>
<td>socket_optint_option</td>
<td>564</td>
</tr>
<tr>
<td>socket_type</td>
<td>561</td>
</tr>
<tr>
<td>socketpair</td>
<td>562</td>
</tr>
<tr>
<td>Sort</td>
<td>496</td>
</tr>
<tr>
<td>sort</td>
<td>386</td>
</tr>
<tr>
<td>sort_uniq</td>
<td>454</td>
</tr>
<tr>
<td>sound</td>
<td>607</td>
</tr>
<tr>
<td>space_formatter</td>
<td>535</td>
</tr>
<tr>
<td>Spacetime</td>
<td>346</td>
</tr>
<tr>
<td>spec</td>
<td>380</td>
</tr>
</tbody>
</table>
INDEX TO THE LIBRARY

tcgetattr, 571
tcsetattr, 571
tcgetattr, 571
temp_dir_name, 410
temp_file, 409
terminal_io, 571
text_size, 604
Thread, 592
ThreadUnix, 597
time, 507, 557
timed_read, 597
timed_write, 597
timed_write_substring, 597
times, 557
tl, 450
tm, 557
to_buffer, 460
to_bytes, 387, 460
to_channel, 459
to_char, 513
to_float, 441, 444, 468
to_hex, 399
to_int, 441, 444, 468, 513
to_int32, 444, 469
to_list, 384
to_nativeint, 445
to_string, 390, 442, 445, 460, 469, 471
token, 432
tool_name, 521
top, 480, 499
top_phrase, 535
toplevel_phrase, 526, 535
total_size, 461
transfer, 480
transp, 604
trim, 391, 504
truncate, 362, 546, 548
try_lock, 594
type_declaration, 530
type_extension, 530
type_kind, 530
Uchar, 513
umask, 550
uncapitalize, 392, 505
uncapitalize_ascii, 393, 506
Undefined, 446
Undefined_recursive_module, 354
unescape, 492
union, 456, 464, 465, 494
unit, 352
unit_big_int, 579
Unix, 537
unix, 508
Unix_error, 511
UnixLabels (module), 572
unlink, 549
unlock, 594
unsafe_get, 624, 626, 629
unsafe_of_string, 395
unsafe_set, 624, 626, 629
unsafe_to_string, 393
unset_data, 402, 404, 406
unset_key, 402, 405
unset_key1, 403
unset_key2, 404
uppercase, 392, 396, 505
uppercase_ascii, 393, 396, 505
usage, 381
usage_msg, 380
usage_string, 382
use_file, 526
utimes, 597
value_binding, 535
value_description, 529
variance, 524
virtual_flag, 523
wait, 542, 595, 597
wait_flag, 542
wait_next_event, 606
wait_pid, 593
wait_read, 592
wait_signal, 594
wait_timed_read, 593
wait_timed_write, 593
wait_write, 593
waitpid, 542, 597
warning_printer, 525
Weak, 514
weaken_map, 239
white, 601
win32, 508
with_constraint, 534
word_size, 508
wrap, 596
wrap_abort, 596
write, 544 597
write_substring, 544 597
xor_big_int, 582
yellow, 602
yield, 593

zero, 397 440 442 466
zero_big_int, 579
Index of keywords

and, 108 121 125 129 130 134 141 149
as, 100 101 104 105 125 127
asr, 98 109 116
assert, 120

begin, 103 108 110

class, 129 130 132 134 135
constraint, 121 123 125 128

do, see while, for
done, see while, for
downto, see for

else, see if
der, 103 108 110
for, 108 114
fun, 108 109 111 125 147
function, 108 109 111
functor, 130 133 134 136

if, 108 109 113
in, see let
include, 130 133 134 136 151
inherit, 123 125 127
inherit!, 154
initializer, 125 129

land, 98 109 116
lazy, 108 120 141
let, 108 109 112 120 125 134 135 145
        147 173
lor, 98 109 116
lsl, 98 109 116
lsr, 98 109 116
lxor, 98 109 116

match, 108 109 113 155
method, 123 125 128
method!, 154
mod, 98 109 116

module, 120 130 132 134 136 141 149 151
        152
mutable, 121 123 125 127 128

new, 108 117
nonrec, 121

object, 108 118 123 125
of, 102 121 151
open, 130 133 134 136 145
open!, 154
or, 108 109 114

private, 123 125 128 142 144

rec, see let, module

sig, 130
struct, 134 136

then, see if
to, see for
ture, 103
try, 108 109 114
type, 121 129 132 134 136 147 151 155

val, 123 125 127 128 130 131 149
val!, 154
virtual, see val, method, class

when, 108 112 160
while, 114

with, 108 130 133 146 149 151