The OCaml system
release 4.08

Documentation and user’s manual

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

February 8, 2019

Copyright © 2019 Institut National de Recherche en Informatique et en Automatique
# Contents

## 1 An introduction to OCaml

### 1.1 Basics

### 1.2 Data types

### 1.3 Functions as values

### 1.4 Records and variants

### 1.5 Imperative features

### 1.6 Exceptions

### 1.7 Symbolic processing of expressions

### 1.8 Pretty-printing

### 1.9 Standalone OCaml programs

## 2 The module system

### 2.1 Structures

### 2.2 Signatures

### 2.3 Functors

### 2.4 Functors and type abstraction

### 2.5 Modules and separate compilation

## 3 Objects in OCaml

### 3.1 Classes and objects

### 3.2 Immediate objects

### 3.3 Reference to self

### 3.4 Initializers

### 3.5 Virtual methods

### 3.6 Private methods

### 3.7 Class interfaces

### 3.8 Inheritance

### 3.9 Multiple inheritance

### 3.10 Parameterized classes

### 3.11 Polymorphic methods

### 3.12 Using coercions

### 3.13 Functional objects

### 3.14 Cloning objects

### 3.15 Recursive classes
<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>63</td>
</tr>
<tr>
<td>3.17</td>
<td>Friends</td>
<td>65</td>
</tr>
<tr>
<td>4</td>
<td>Labels and variants</td>
<td>67</td>
</tr>
<tr>
<td>4.1</td>
<td>Labels</td>
<td>67</td>
</tr>
<tr>
<td>4.2</td>
<td>Polymorphic variants</td>
<td>73</td>
</tr>
<tr>
<td>5</td>
<td>Polymorphism and its limitations</td>
<td>77</td>
</tr>
<tr>
<td>5.1</td>
<td>Weak polymorphism and mutation</td>
<td>77</td>
</tr>
<tr>
<td>5.2</td>
<td>Polymorphic recursion</td>
<td>82</td>
</tr>
<tr>
<td>5.3</td>
<td>Higher-rank polymorphic functions</td>
<td>85</td>
</tr>
<tr>
<td>6</td>
<td>Advanced examples with classes and modules</td>
<td>87</td>
</tr>
<tr>
<td>6.1</td>
<td>Extended example: bank accounts</td>
<td>87</td>
</tr>
<tr>
<td>6.2</td>
<td>Simple modules as classes</td>
<td>93</td>
</tr>
<tr>
<td>6.3</td>
<td>The subject/observer pattern</td>
<td>98</td>
</tr>
<tr>
<td>II</td>
<td>The OCaml language</td>
<td>103</td>
</tr>
<tr>
<td>7</td>
<td>The OCaml language</td>
<td>105</td>
</tr>
<tr>
<td>7.1</td>
<td>Lexical conventions</td>
<td>105</td>
</tr>
<tr>
<td>7.2</td>
<td>Values</td>
<td>110</td>
</tr>
<tr>
<td>7.3</td>
<td>Names</td>
<td>112</td>
</tr>
<tr>
<td>7.4</td>
<td>Type expressions</td>
<td>115</td>
</tr>
<tr>
<td>7.5</td>
<td>Constants</td>
<td>118</td>
</tr>
<tr>
<td>7.6</td>
<td>Patterns</td>
<td>119</td>
</tr>
<tr>
<td>7.7</td>
<td>Expressions</td>
<td>124</td>
</tr>
<tr>
<td>7.8</td>
<td>Type and exception definitions</td>
<td>138</td>
</tr>
<tr>
<td>7.9</td>
<td>Classes</td>
<td>141</td>
</tr>
<tr>
<td>7.10</td>
<td>Module types (module specifications)</td>
<td>147</td>
</tr>
<tr>
<td>7.11</td>
<td>Module expressions (module implementations)</td>
<td>152</td>
</tr>
<tr>
<td>7.12</td>
<td>Compilation units</td>
<td>155</td>
</tr>
<tr>
<td>8</td>
<td>Language extensions</td>
<td>157</td>
</tr>
<tr>
<td>8.1</td>
<td>Recursive definitions of values</td>
<td>157</td>
</tr>
<tr>
<td>8.2</td>
<td>Recursive modules</td>
<td>158</td>
</tr>
<tr>
<td>8.3</td>
<td>Private types</td>
<td>160</td>
</tr>
<tr>
<td>8.4</td>
<td>Local opens for patterns</td>
<td>162</td>
</tr>
<tr>
<td>8.5</td>
<td>Locally abstract types</td>
<td>163</td>
</tr>
<tr>
<td>8.6</td>
<td>First-class modules</td>
<td>164</td>
</tr>
<tr>
<td>8.7</td>
<td>Recovering the type of a module</td>
<td>166</td>
</tr>
<tr>
<td>8.8</td>
<td>Substituting inside a signature</td>
<td>167</td>
</tr>
<tr>
<td>8.9</td>
<td>Type-level module aliases</td>
<td>169</td>
</tr>
<tr>
<td>8.10</td>
<td>Overriding in open statements</td>
<td>170</td>
</tr>
<tr>
<td>8.11</td>
<td>Generalized algebraic datatypes</td>
<td>171</td>
</tr>
<tr>
<td>Section</td>
<td>Page</td>
<td></td>
</tr>
<tr>
<td>----------------------------------------------</td>
<td>------</td>
<td></td>
</tr>
<tr>
<td>8.12 Syntax for Bigarray access</td>
<td>176</td>
<td></td>
</tr>
<tr>
<td>8.13 Attributes</td>
<td>177</td>
<td></td>
</tr>
<tr>
<td>8.14 Extension nodes</td>
<td>183</td>
<td></td>
</tr>
<tr>
<td>8.15 Extensible variant types</td>
<td>185</td>
<td></td>
</tr>
<tr>
<td>8.16 Generative functors</td>
<td>187</td>
<td></td>
</tr>
<tr>
<td>8.17 Extension-only syntax</td>
<td>188</td>
<td></td>
</tr>
<tr>
<td>8.18 Inline records</td>
<td>189</td>
<td></td>
</tr>
<tr>
<td>8.19 Documentation comments</td>
<td>190</td>
<td></td>
</tr>
<tr>
<td>8.20 Extended indexing operators</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>8.21 Empty variant types</td>
<td>193</td>
<td></td>
</tr>
<tr>
<td>8.22 Alerts</td>
<td>194</td>
<td></td>
</tr>
<tr>
<td>8.23 Generalized open statements</td>
<td>195</td>
<td></td>
</tr>
<tr>
<td>8.24 Binding operators</td>
<td>197</td>
<td></td>
</tr>
</tbody>
</table>

III The OCaml tools 201

9 Batch compilation (ocamlc) 203
   9.1 Overview of the compiler               203
   9.2 Options                                204
   9.3 Modules and the file system            216
   9.4 Common errors                          217
   9.5 Warning reference                      219

10 The toplevel system or REPL (ocaml) 223
   10.1 Options                               224
   10.2 Toplevel directives                   231
   10.3 The toplevel and the module system     234
   10.4 Common errors                         234
   10.5 Building custom toplevel systems: ocamlmktop 234
   10.6 The native toplevel: ocamlnat (experimental) 235

11 The runtime system (ocamlrun) 237
   11.1 Overview                              237
   11.2 Options                               238
   11.3 Dynamic loading of shared libraries    240
   11.4 Common errors                         241

12 Native-code compilation (ocamlopt) 243
   12.1 Overview of the compiler               243
   12.2 Options                               244
   12.3 Common errors                         257
   12.4 Running executables produced by ocamlopt 257
   12.5 Compatibility with the bytecode compiler 257
<table>
<thead>
<tr>
<th>Chapter</th>
<th>Title</th>
<th>Page</th>
</tr>
</thead>
<tbody>
<tr>
<td>13</td>
<td>Lexer and parser generators (ocamllex, ocamlyacc)</td>
<td>259</td>
</tr>
<tr>
<td>13.1</td>
<td>Overview of ocamllex</td>
<td>259</td>
</tr>
<tr>
<td>13.2</td>
<td>Syntax of lexer definitions</td>
<td>260</td>
</tr>
<tr>
<td>13.3</td>
<td>Overview of ocamlyacc</td>
<td>265</td>
</tr>
<tr>
<td>13.4</td>
<td>Syntax of grammar definitions</td>
<td>265</td>
</tr>
<tr>
<td>13.5</td>
<td>Options</td>
<td>268</td>
</tr>
<tr>
<td>13.6</td>
<td>A complete example</td>
<td>269</td>
</tr>
<tr>
<td>13.7</td>
<td>Common errors</td>
<td>270</td>
</tr>
<tr>
<td>13.8</td>
<td>Module Depend: Module dependencies.</td>
<td>271</td>
</tr>
<tr>
<td>14</td>
<td>The browser/editor (ocamlbrowser)</td>
<td>273</td>
</tr>
<tr>
<td>15</td>
<td>The documentation generator (ocamldoc)</td>
<td>275</td>
</tr>
<tr>
<td>15.1</td>
<td>Usage</td>
<td>275</td>
</tr>
<tr>
<td>15.2</td>
<td>Syntax of documentation comments</td>
<td>282</td>
</tr>
<tr>
<td>15.3</td>
<td>Custom generators</td>
<td>291</td>
</tr>
<tr>
<td>15.4</td>
<td>Adding command line options</td>
<td>294</td>
</tr>
<tr>
<td>16</td>
<td>The debugger (ocamldebug)</td>
<td>295</td>
</tr>
<tr>
<td>16.1</td>
<td>Compiling for debugging</td>
<td>295</td>
</tr>
<tr>
<td>16.2</td>
<td>Invocation</td>
<td>295</td>
</tr>
<tr>
<td>16.3</td>
<td>Commands</td>
<td>296</td>
</tr>
<tr>
<td>16.4</td>
<td>Executing a program</td>
<td>297</td>
</tr>
<tr>
<td>16.5</td>
<td>Breakpoints</td>
<td>300</td>
</tr>
<tr>
<td>16.6</td>
<td>The call stack</td>
<td>300</td>
</tr>
<tr>
<td>16.7</td>
<td>Examining variable values</td>
<td>301</td>
</tr>
<tr>
<td>16.8</td>
<td>Controlling the debugger</td>
<td>302</td>
</tr>
<tr>
<td>16.9</td>
<td>Miscellaneous commands</td>
<td>305</td>
</tr>
<tr>
<td>16.10</td>
<td>Running the debugger under Emacs</td>
<td>305</td>
</tr>
<tr>
<td>17</td>
<td>Profiling (ocamlprof)</td>
<td>307</td>
</tr>
<tr>
<td>17.1</td>
<td>Compiling for profiling</td>
<td>307</td>
</tr>
<tr>
<td>17.2</td>
<td>Profiling an execution</td>
<td>308</td>
</tr>
<tr>
<td>17.3</td>
<td>Printing profiling information</td>
<td>308</td>
</tr>
<tr>
<td>17.4</td>
<td>Time profiling</td>
<td>309</td>
</tr>
<tr>
<td>18</td>
<td>The ocamlbuild compilation manager</td>
<td>311</td>
</tr>
<tr>
<td>19</td>
<td>Interfacing C with OCaml</td>
<td>313</td>
</tr>
<tr>
<td>19.1</td>
<td>Overview and compilation information</td>
<td>313</td>
</tr>
<tr>
<td>19.2</td>
<td>The value type</td>
<td>320</td>
</tr>
<tr>
<td>19.3</td>
<td>Representation of OCaml data types</td>
<td>321</td>
</tr>
<tr>
<td>19.4</td>
<td>Operations on values</td>
<td>324</td>
</tr>
<tr>
<td>19.5</td>
<td>Living in harmony with the garbage collector</td>
<td>327</td>
</tr>
<tr>
<td>19.6</td>
<td>A complete example</td>
<td>331</td>
</tr>
<tr>
<td>19.7</td>
<td>Advanced topic: callbacks from C to OCaml</td>
<td>335</td>
</tr>
</tbody>
</table>
20 Optimisation with Flambda
20.1 Overview .............................................. 359
20.2 Command-line flags .................................. 359
20.3 Inlining ................................................. 362
20.4 Specialisation ......................................... 367
20.5 Default settings of parameters .................... 370
20.6 Manual control of inlining and specialisation ... 371
20.7 Simplification ......................................... 372
20.8 Other code motion transformations ............... 373
20.9 Unboxing transformations ......................... 374
20.10 Removal of unused code and values ............. 378
20.11 Other code transformations ....................... 378
20.12 Treatment of effects ............................... 379
20.13 Compilation of statically-allocated modules ... 380
20.14 Inhibition of optimisation ......................... 380
20.15 Use of unsafe operations ......................... 380
20.16 Glossary .............................................. 381
20.17 Module Spacetime: Profiling of a program’s space behaviour over time. ... 382

21 Fuzzing with afl-fuzz
21.1 Overview .............................................. 385
21.2 Generating instrumentation ....................... 385
21.3 Example ................................................. 385

22 Compiler plugins ................................. 387
22.1 Overview .............................................. 387
22.2 Basic example ........................................ 387

IV The OCaml library .................................. 389

23 The core library .................................. 391
23.1 Built-in types and predefined exceptions ......... 391
23.2 Module Stdlib: The OCaml Standard library. .... 394
23.3 Module Stdlib: The OCaml Standard library. .... 420
# The compiler front-end

24.1 Module `Ast_mapper` : The interface of a -ppx rewriter ............ 449
24.2 Module `Asttypes` : Auxiliary AST types used by parsetree and typedtree. ....... 453
24.3 Module `Location` : Source code locations (ranges of positions), used in parsetree . 454
24.4 Module `Longident` : Long identifiers, used in parsetree .................. 460
24.5 Module `Parse` : Entry points in the parser ............................. 460
24.6 Module `Parsetree` : Abstract syntax tree produced by parsing .......... 460
24.8 Module `Pparse` : Driver for the parser, external preprocessors and ast plugin hooks 471
24.9 Module `Typemod` : Type-checking of the module language and typed ast plugin hooks 472
24.10 Module `Simplif` : Lambda simplification and lambda plugin hooks ........ 474
24.11 Module `Clflags` : Command line flags .................................. 475

# The unix library: Unix system calls

25.1 Module `Unix` : Interface to the Unix system ................................ 481
25.2 Module `UnixLabels` : labelized version of the interface ................. 523

# The num library: arbitrary-precision rational arithmetic

26.1 Module `Num` : Interface to the Unix system ................................ 525

# The str library: regular expressions and string processing

27.1 Module `Str` : Regular expressions and high-level string processing ....... 527

# The threads library

28.1 Module `Thread` : Lightweight threads for Posix 1003.1c and Win32 ........ 534
28.2 Module `Mutex` : Locks for mutual exclusion .............................. 536
28.3 Module `Condition` : Condition variables to synchronize between threads ... 536
28.4 Module `Event` : First-class synchronous communication ........................ 537
28.5 Module `ThreadUnix` : Thread-compatible system calls ........................ 539

# The graphics library

29.1 Module `Graphics` : Machine-independent graphics primitives ............... 541

# The dynlink library: dynamic loading and linking of object files

30.1 Module `Dynlink` : Dynamic loading of .cmo, .cma and .cmxs files ........ 551

# The bigarray library

31.1 Module `Bigarray` : Machine-independent graphics primitives ............... 555

V Appendix

Index to the library ................................. 559
Index of keywords .................................. 571
Foreword

This manual documents the release 4.08 of the OCaml system. It is organized as follows.

- 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 (XP, Vista, 7, 8, 10).

License

The OCaml system is copyright © 1996–2019 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 © 2019 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 sites http://www.ocaml.org/ and http://caml.inria.fr/. The former 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, C or Java) 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 6 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**
  ```ocaml
  # (1 < 2) = false;;
  - : bool = false
  # let one = if true then 1 else 2;;
  val one : int = 1
  ```

- **characters**
  ```ocaml
  # 'a';;
  - : char = 'a'
  # int_of_char \n;;
  - : int = 10
  ```

- **immutable character strings**
  ```ocaml
  # "Hello" ^ " " ^ "world";;
  - : string = "Hello world"
  # {"This is a quoted string, here, neither \ nor " are special characters"};;
  - : string = "This is a quoted string, here, neither \ nor " are special characters"
  # {"\\"|}="\\\\\\\\";;
  - : bool = true
  # \{delimiter|the end of this\}quoted string is here|delimiter}\n
  # = "the end of this\}quoted string is here";;
  - : bool = true
  ```

Predefined data structures include tuples, arrays, and lists. There are also general mechanisms for defining your own data structures, such as records and variants, which will be covered in more detail later; for now, we concentrate on lists. Lists are either given in extension as a bracketed list of semicolon-separated elements, or built from the empty list `[]` (pronounce “nil”) by adding elements in front using the `::` (“cons”) operator.
# let l = ["is"; "a"; "tale"; "told"; "etc."];;
val l : string list = ["is"; "a"; "tale"; "told"; "etc."]

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

As with all other 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 exactly the same form as list expressions, with identifiers representing unspecified parts of the list. As an example, here is insertion sort on a list:

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

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

The type inferred for sort, 'a list -> 'a list, means that sort can actually apply to lists of any type, and returns a list of the same type. The type 'a is a type variable, and stands for any given type. The reason why sort can apply to lists of any type is that the comparisons (=, <=, etc.) are polymorphic in 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 a list in-place 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.

The OCaml notation for the type of a function with multiple arguments is arg1_type -> arg2_type -> ... -> return_type. For example, the type inferred for insert, 'a -> 'a list -> 'a list, means that insert takes two arguments, an element of any type 'a and a list with elements of the same type 'a and returns a list of the same type.
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 func-
tions”. 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:

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

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

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

1.4 Records and variants

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

# 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};;
Chapter 1. The core language

val add_ratio : ratio -> ratio -> ratio = <fun>

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

Record fields can also be accessed through pattern-matching:

# let integer_part r =
#   match r with
#     {num=num; denom=denom} -> num / denom;;
val integer_part : ratio -> int = <fun>

Since there is only one case in this pattern matching, it is safe to expand directly the argument r in a record pattern:

# let integer_part {num=num; denom=denom} = num / denom;;
val integer_part : ratio -> int = <fun>

Unneeded fields can be omitted:

# let get_denom {denom=denom} = denom;;
val get_denom : ratio -> int = <fun>

Optionally, missing fields can be made explicit by ending the list of fields with a trailing wildcard _:

# let get_num {num=num; _} = num;;
val get_num : ratio -> int = <fun>

When both sides of the = sign are the same, it is possible to avoid repeating the field name by eliding the =field part:

# let integer_part {num; denom} = num / denom;;
val integer_part : ratio -> int = <fun>

This short notation for fields also works when constructing records:

# let ratio num denom = {num; denom};;
val ratio : int -> int -> ratio = <fun>

At last, it is possible to update few fields of a record at once:

# let integer_product integer ratio = { ratio with num = integer * ratio.num };;
val integer_product : int -> ratio -> ratio = <fun>

With this functional update notation, the record on the left-hand side of with is copied except for the fields on the right-hand side which are updated.

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

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

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

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

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

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

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

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

Another interesting example of variant type is the built-in 'a option type which represents either a value of type 'a or an absence of value:

```
# type 'a option = Some of 'a | None;;
type 'a option = Some of 'a | None
```

This type is particularly useful when defining function that can fail in common situations, for instance

```
# let safe_square_root x = if x > 0. then Some(sqrt x) else None;;
val safe_square_root : float -> float option = <fun>
```

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

```
# type 'a btree = Empty | Node of 'a * 'a btree * 'a btree;;
type 'a btree = Empty | Node of 'a * 'a btree * 'a btree
```
This definition reads as follows: 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 also containing 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>
```

### 1.4.1 Record and variant disambiguation

( This subsection can be skipped on the first reading )

Astute readers may have wondered what happens when two or more record fields or constructors share the same name

```ocaml
# type first_record = { x:int; y:int; z:int }    
# type middle_record = { x:int; z:int }    
# type last_record = { x:int };;
# type first_variant = A | B | C    
# type last_variant = A;;
```

The answer is that when confronted with multiple options, OCaml tries to use locally available information to disambiguate between the various fields and constructors. First, if the type of the record or variant is known, OCaml can pick unambiguously the corresponding field or constructor. For instance:

```ocaml
# let look_at_x_then_z (r: first_record) =    
#   let x = r.x in    
#   x + r.z;;
val look_at_x_then_z : first_record -> int = <fun>

# let permute (x: first_variant) = match x with    
#   | A -> (B: first_variant)
```
# | B -> A
# | C -> C;;
val permute : first_variant -> first_variant = <fun>

val f : wrapped -> first_record * int = <fun>

In the first example, \((r: \text{first_record})\) is an explicit annotation telling OCaml that the type of \(r\) is \text{first_record}. With this annotation, OCaml knows that \(r.x\) refers to the \(x\) field of the first record type. Similarly, the type annotation in the second example makes it clear to OCaml that the constructors \(A\), \(B\) and \(C\) come from the first variant type. Contrarily, in the last example, OCaml has inferred by itself that the type of \(r\) can only be \text{first_record} and there are no needs for explicit type annotations.

Those explicit type annotations can in fact be used anywhere. Most of the time they are unnecessary, but they are useful to guide disambiguation, to debug unexpected type errors, or combined with some of the more advanced features of OCaml described in later chapters.

Secondly, for records, OCaml can also deduce the right record type by looking at the whole set of fields used in an expression or pattern:

```ocaml
# let project_and_rotate {x;y;_} = { x= - y; y = x ; z = 0} ;;
val project_and_rotate : first_record -> first_record = <fun>
```

Since the fields \(x\) and \(y\) can only appear simultaneously in the first record type, OCaml infers that the type of \(\text{project_and_rotate}\) is \text{first_record -> first_record}.

In last resort, if there is not enough information to disambiguate between different fields or constructors, OCaml picks the last defined type amongst all locally valid choices:

```ocaml
# let look_at_xz {x;z} = x;;
val look_at_xz : middle_record -> int = <fun>
```

Here, OCaml has inferred that the possible choices for the type of \({x;z}\) are \text{first_record} and \text{middle_record}, since the type \text{last_record} has no field \(z\). OCaml then picks the type \text{middle_record} as the last defined type between the two possibilities.

Beware that this last resort disambiguation is local: once OCaml has chosen a disambiguation, it sticks to this choice, even if it leads to an ulterior type error:

```ocaml
# let look_at_x_then_y r =
# let x = r.x in (* OCaml deduces \([r: \text{last_record}]\) *)
# x + r.y;;
Error: This expression has type last_record
  The field \(y\) does not belong to type last_record
```

```ocaml
# let is_a_or_b x = match x with
# | A -> true (* OCaml infers \([x: \text{last_variant}]\) *)
# | B -> true;;
Error: This variant pattern is expected to have type last_variant
  The constructor \(B\) does not belong to type last_variant
```
Moreover, being the last defined type is a quite unstable position that may change surreptitiously after adding or moving around a type definition, or after opening a module (see chapter 2). Consequently, adding explicit type annotations to guide disambiguation is more robust than relying on the last defined type disambiguation.

### 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 created by listing semicolon-separated element values 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.

```ocaml
# 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:

```ocaml
# 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, with operators `!` to fetch the current contents of the reference and `:=` to assign the contents. Variables can then be emulated by `let`-binding a reference. For instance, here is an in-place insertion sort over arrays:
# let insertion_sort a =
#   for i = 1 to Array.length a - 1 do
#     let val_i = a.(i) in
#     let j = ref i in
#     while !j > 0 && val_i < a.(!j - 1) do
#       a.(!j) <- a.(!j - 1);
#       j := !j - 1
#     done;
#     a.(!j) <- val_i
#   done;;
val insertion_sort : 'a array -> unit = <fun>

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

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

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

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

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

In some special cases, you may need to store a polymorphic function in a data structure, keeping
its polymorphism. Doing this requires user-provided type annotations, since polymorphism is only
introduced automatically for global definitions. However, you can explicitly give polymorphic types
to record fields.

# type idref = { mutable id: 'a. 'a -> 'a };;
val ( := ) : 'a ref -> 'a -> unit = <fun>

# r.id <- (fun x -> print_string "called id\n"; x);;
- : unit = ()
Chapter 1. The core language

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, although this should not be overused since it can make the code harder to understand. 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>
```

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 does pattern matching on the exception value with the same syntax and behavior as `match`. Thus, several exceptions can be caught by one `try...with` construct:

```ocaml
# let rec first_named_value values names =
#    try
#       List.assoc (head values) names
#    with
#       | Empty_list -> "no named value"
#       | Not_found -> first_named_value (List.tl values) names;;
val first_named_value : 'a list -> ('a * string) list -> string = <fun>

# first_named_value [ 0; 10 ] [ 1, "one"; 10, "ten"];;
- : string = "ten"
```

Also, finalization can be performed by trapping all exceptions, performing the finalization, then re-raising the exception:

```ocaml
# 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>
```

An alternative to `try...with` is to catch the exception while pattern matching:

```ocaml
# let assoc_may_map f x l =
#    match List.assoc x l with
#    | exception Not_found -> None
#    | Some _ as v -> f y;;
val assoc_may_map : ('a -> 'b option) -> 'c -> ('c * 'a) list -> 'b option = <fun>
```

Note that this construction is only useful if the exception is raised between `match...with`. Exception patterns can be combined with ordinary patterns at the toplevel,

```ocaml
# let flat_assoc_opt x l =
#    match List.assoc x l with
#    | None | exception Not_found -> None
#    | _ as v -> v;;
val flat_assoc_opt : 'a -> ('a * 'b option) list -> 'b option = <fun>
```

but they cannot be nested inside other patterns. For instance, the pattern `Some (exception A)` is invalid.

When exceptions are used as a control structure, it can be useful to make them as local as possible by using a locally defined exception. For instance, with
# let fixpoint f x =
#  let exception Done in
#  let x = ref x in
#  try while true do
#    let y = f !x in
#    if !x = y then raise Done else x := y
#  done; assert false
#  with Done -> !x;;
val fixpoint : ('a -> 'a) -> 'a -> 'a = <fun>

the function f cannot raise a Done exception, which removes an entire class of misbehaving functions.

## 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:

```ocaml
# 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.

```ocaml
# 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
# | Var v -> print_string v
# | Sum(f, g) ->

Chapter 1. The core language

# 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. The ;; used in the interactive examples is not required in source files created for use with OCaml compilers, but can be helpful to mark the end of a top-level expression unambiguously even when there are syntax errors. Here is a sample standalone program to print Fibonacci numbers:

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

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

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

More complex standalone OCaml programs are typically composed of multiple source files, and can link with precompiled libraries. Chapters 9 and 12 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)
```
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.

Another possibility is to open the module, which brings all identifiers defined inside the module in the scope of the current structure.

Opening a module enables lighter access to its components, at the cost of making it harder to identify in which module a identifier has been defined. In particular, opened modules can shadow identifiers present in the current scope, potentially leading to confusing errors:

A partial solution to this conundrum is to open modules locally, making the components of the module available only in the concerned expression. This can also make the code easier to read – the open statement is closer to where it is used– and to refactor – the code fragment is more self-contained. Two constructions are available for this purpose:
Chapter 2. The module system

and

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

In the second form, when the body of a local open is itself delimited by parentheses, braces or bracket, the parentheses of the local open can be omitted. For instance,

# PrioQueue.[empty] = PrioQueue.([empty]);;
- : bool = true

# PrioQueue.[|empty|] = PrioQueue.([|empty|]);;
- : bool = true

# PrioQueue.{ contents = empty } = PrioQueue.({ contents = empty });;
- : bool = true

becomes

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

It is also possible to copy the components of a module inside another module by using an include statement. This can be particularly useful to extend existing modules. As an illustration, we could add functions that returns an optional value rather than an exception when the priority queue is empty.

# module PrioQueueOpt =
# struct
#   include PrioQueue
#   let remove_top_opt x =
#     try Some(remove_top x) with Queue_is_empty -> None
#   let extract_opt x =
#     try Some(extract x) with Queue_is_empty -> None
# end;;
module PrioQueueOpt :
  sig
    type priority = int
    type 'a queue =
      'a PrioQueue.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
    val remove_top_opt : 'a queue -> 'a queue option
    val extract_opt : 'a queue -> (priority * 'a * 'a queue) option
  end
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).

```ocaml
# 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
  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:

```ocaml
# 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

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

An alternate syntax is provided for the above:

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

Like for modules, it is possible to include a signature to copy its components inside the current signature. For instance, we can extend the `PRIOQUEUE` signature with the `extract_opt` function:
Chapter 2. The module system

```ocaml
# module type PRIOQUEUE_WITH_OPT =
# sig
#   include PRIOQUEUE
#   val extract_opt : 'a queue -> (int * 'a * 'a queue) option
# end;;
module type PRIOQUEUE_WITH_OPT =
  sig
    type priority = int
    type 'a queue
    val empty : 'a queue
    val insert : 'a queue -> int -> 'a -> 'a queue
    val extract : 'a queue -> int * 'a * 'a queue
    exception Queue_is_empty
    val extract_opt : 'a queue -> (int * 'a * 'a queue) option
  end

2.3 Functors

Functors are “functions” from modules to modules. Functors let you create parameterized modules
and then provide other modules as parameter(s) to get a specific implementation. For instance, a
Set module implementing sets as sorted lists could be parameterized to work with any module that
provides an element type and a comparison function compare (such as OrderedString):

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

```ocaml
# 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

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

```ocaml
# 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
#
# StringSet.member "bar" (StringSet.add "foo" StringSet.empty);;
# - : bool = false
```

### 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:
Chapter 2. The module system

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

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

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

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

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

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

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

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

# WrongStringSet.add "gee" WrongStringSet.empty ;;
Error: This expression has type string but an expression was expected of type
  WrongStringSet.element = WrongSet(OrderedString).element

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

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.
Chapter 2. The module system

```ocaml
# module NoCaseString =
#   struct
#     type t = string
#     let compare s1 s2 =
#       OrderedString.compare (String.lowercase_ascii s1) (String.lowercase_ascii s2)
#   end;;
```

```ocaml
module NoCaseString :
  sig
    type t = string
    val compare : string -> string -> comparison
  end
```

```ocaml
# 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
```

```ocaml
# 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 practical 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 \texttt{A} as if the following definition was entered at top-level:

\begin{verbatim}
module A: sig (* contents of file A.mli *) end
  = struct (* contents of file A.ml *) end;;
\end{verbatim}

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

\begin{verbatim}
$ 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
\end{verbatim}

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

\begin{verbatim}
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;;
\end{verbatim}

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

The order in which the \texttt{.cmo} files are given to \texttt{ocamlc} during the linking phase determines the order in which the module definitions occur. Hence, in the example above, \texttt{Aux} appears first and \texttt{Main} can refer to it, but \texttt{Aux} cannot refer to \texttt{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 relationship between object, class and type in OCaml is different than in mainstream object-oriented languages such as Java and C++, so you shouldn’t assume that similar keywords mean the same thing. Object-oriented features are used much less frequently in OCaml than in those languages. OCaml has alternatives that are often more appropriate, such as modules and functors. Indeed, many OCaml programs do not use objects at all.

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

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

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

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

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

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

# 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
```

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

```ocaml
# class point x_init =
# object
# val mutable x = x_init
# method get_x = x
# method move d = x <- x + d
# end;;
class point :
int ->
object val mutable x : int method get_x : int method move : int -> unit end
```
Chapter 3. Objects in OCaml

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_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
\end{verbatim}

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 3.4.

\section{Immediate objects}

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

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

\begin{verbatim}
# let p = 
# object
#   val mutable x = 0
#   method get\_x = x
#   method move d = x <- x + d
# end;
val p : < get\_x : int; move : int -> unit > = <obj>
# p\#get\_x;;
- : int = 0
# p\#move 3;;
- : unit = ()
# p\#get\_x;;
- : int = 3
\end{verbatim}

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

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

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

A method or an initializer can invoke methods on 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 : '_weak1 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 '_weak1
Self type cannot escape its class
```

You can ignore the first two lines of the error message. What matters is the last one: putting \texttt{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;
```

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 6.1.

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.)

```ocaml
# 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
3.6 Private methods

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

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

# let p = new restricted_point 0;;
val p : restricted_point = <obj>
# p#move 10 ;;
Error: This expression has type restricted_point
  It has no method move

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

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

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

Private methods can be made public in a subclass.

```ocaml
# class point_again x =
#   object (self)
#     inherit restricted_point x
#     method virtual move : _
#   end;
class point_again :
```
Chapter 3. Objects in OCaml

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

The annotation \texttt{virtual} here is only used to mention a method without providing its definition. Since we didn't add the \texttt{private} annotation, this makes the method public, keeping the original definition.

An alternative definition is

\begin{verbatim}
# 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
\end{verbatim}

The constraint on self's type is requiring a public \texttt{move} method, and this is sufficient to override \texttt{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:

\begin{verbatim}
# 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
\end{verbatim}

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

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

```ocaml
# class type restricted_point_type =
#     object
#     method get_x : int
#     method bump : unit
# end;
class type restricted_point_type =
    object method bump : unit method get_x : int end
# fun (x : restricted_point_type) -> x;;
- : restricted_point_type -> restricted_point_type = <fun>
```

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

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

Or, equivalently:

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

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

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

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

```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 color. However, the function get_x below is a generic function applying method get_x to any object p that has this method (and possibly some others, which are represented by an ellipsis in the type). Thus, it applies to both points and colored points.

```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
```

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.

Note that for clarity’s sake, the method print is explicitly marked as overriding another definition by annotating the method keyword with an exclamation mark !. If the method print were not overriding the print method of printable_point, the compiler would raise an error:

```ocaml
# object
#  method! m = ()
# end;;
Error: The method `m' has no previous definition
```

This explicit overriding annotation also works for val and inherit:

```ocaml
# class another_printable_colored_point y c c' =
```
Chapter 3. Objects in OCaml

# object (self)
# inherit printable_point y
# inherit! printable_colored_point y c
# val! c = c'
# end;;
class another_printable_colored_point :
  int ->
  string ->
  string ->
  object
    val c : string
    val mutable x : int
    method color : string
    method get_x : int
    method move : int -> unit
    method print : unit
  end

3.10 Parameterized classes

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

# class oref x_init =
#   object
#     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 :
```ocaml
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>
```

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

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

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.

```ocaml
class [a] oref x_init =

  object
    val mutable x = (x_init : 'a)
    method get = x
    method set y = x <- y
  end;

class [a] oref :
  'a -> object val mutable x : 'a method get : 'a method set : 'a -> unit end
```

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

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

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.
Chapter 3. Objects in OCaml

### Class Definition

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

An alternate definition of `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`.

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

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

```ocaml
# class ['a] colored_circle c =
#   object
```
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.

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

At first look, we seem to have a polymorphic iterator, however this does not work in practice.

```plaintext
# let l = new intlist [1; 2; 3];;
val l : _weak2 intlist = <obj>
# l#fold (fun x y -> x+y) 0;;
- : int = 6
# l;;
- : int intlist = <obj>
# l#fold (fun s x -> s ^ Int.to_string x ^ " ") " " ;;
Error: This expression has type int but an expression was expected of type string
```

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.
Chapter 3. Objects in OCaml

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.

```ocaml
# class intlist (l : int list) =
# object
#   method empty = (l = [])
#   method fold : 'a. ('a -> int -> 'a) -> 'a -> 'a =
#     fun f accu -> List.fold_left f accu l
# end;;
class intlist :
  int list ->
  object method empty : bool method fold : ('a -> int -> 'a) -> 'a -> 'a end

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

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 -> 'b end;;
# class intlist' l =
# object (self : int #iterator)
#   method empty = (l = [])
#   method fold f accu = List.fold_left f accu l
# end;;
```
Note here the (self : int #iterator) idiom, which ensures that this object implements the interface iterator.

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

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

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

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

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

Another use of polymorphic methods is to allow some form of implicit subtyping in method arguments. We have already seen in section 3.8 how some functions may be polymorphic in the class of their argument. This can be extended to methods.

```plaintext
# 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
```
Chapter 3. Objects in OCaml

```
# (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.

```
# 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 given.

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

```
# let colored_point_to_point cp = (cp : colored_point :> point);;
val colored_point_to_point : colored_point -> point = <fun>
# let p = new point 3 and q = new colored_point 4 "blue";;
val p : point = <obj>
val q : colored_point = <obj>
# let l = [p; (colored_point_to_point q)];;
val l : point list = [<obj>; <obj>]
```

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

```
# (p : point :> colored_point);;
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 >
The first object type has no method color
```
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 `c1` is an abbreviation for the type `<m : 'a> as 'a`. The coercion from an object of type `c0` to an object of type `c1` is correct:

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

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

```ocaml
# class type c2 = object ('a) method m : 'a end;;
class type c2 = object ('a) method m : 'a end
```

```ocaml
# fun (x:c0) -> (x :> c2);;
- : c0 -> c2 = <fun>
```

While class types `c1` and `c2` are different, both object types `c1` and `c2` 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:

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

```ocaml
# let to_c2 x = (x :> c2);;
val to_c2 : #c2 -> c2 = <fun>
```
Note the difference between these two coercions: in the case of `to_c2`, the type

```
#c2 = < m : 'a; .. > as 'a
```

is polymorphically recursive (according to the explicit recursion in the class type of `c2`); hence the success of applying this coercion to an object of class `c0`. On the other hand, in the first case, `c1` 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 `to_c2` is `#c2 -> c2` while the type of `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

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

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

```
# class c = object 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 : '_weak3 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:

```ocaml
# 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:

```ocaml
# 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
#  method as_c = (self :> c')
# end ;;
class c : c'
and d : object method as_c : c' method m : int method n : int end
```

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'`.

```ocaml
# 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:

```ocaml
# 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:

```ocaml
# 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.

```ocaml
# class functional_point y =
  # object
  #   val x = y
  #   method get_x = x
  #   method move d = {< x = x + d >}
  #   method move_to x = {< x >}
  # end;

class functional_point :
  int ->
  object ('a)
    val x : int
    method get_x : int
    method move : int -> 'a
    method move_to : 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#move_to 15)#get_x;;
- : int = 15

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

As with records, the form `{< x >}` is an elided version of `{< x = x >}` which avoids the repetition of the instance variable name. 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)
  #   method move_to x = new bad_functional_point x
  # end;

class bad_functional_point :
```
int ->
object
  val x : int
  method get_x : int
  method move : int -> bad_functional_point
  method move_to : 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 6.2.1.

### 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.

```ocaml
# 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:

```ocaml
# class copy =
# object
#   method copy = {< >}
# end;;

class copy : object ('a) method copy : 'a end

# class copy =
# object (self)
#   method copy = Oo.copy self
# end;;

class copy : object ('a) method copy : 'a end
```

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

Cloning can also be used to provide facilities for saving and restoring the state of objects.

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

class backup :
  object ('a)
    val mutable copy : 'a option
    method restore : 'a
    method save : unit
  end
```

The above definition will only backup one level. The backup facility can be added to any class 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.)

# 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)
Chapter 3. Objects in OCaml

# 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`.

# 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
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 times : float -> 'a
  method value : float
end
```

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

```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 `money` or `money2`.

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

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

More examples of binary methods can be found in sections 6.2.1 and 6.2.3.

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

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

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

```ocaml
# class safe_money x =
#   object (self : 'a)
#     val repr : float
#     method print : unit
#     method times : float -> 'a
#     method value : float
#   end

class safe_money :
  float ->
  object ('a)
  val repr : float
  method leq : 'a -> bool
  method plus : 'a -> 'a
  method print : unit
  method times : float -> 'a
  method value : float
  end
```

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.

```ocaml
# module type MONEY =
```
Another example of friend functions may be found in section 6.2.3. 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 Labels in the standard library, you will see that function types have annotations you did not have in the functions you defined yourself.

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

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

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

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

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

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

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

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

Formal parameters and arguments are matched according to their respective labels\textsuperscript{1}, 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.

\begin{verbatim}
# 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>
\end{verbatim}

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.

\begin{verbatim}
# 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)
\end{verbatim}

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.

\begin{verbatim}
# f 3 2;;
- : int = 1
# ListLabels.map succ [1;2;3];;
- : int list = [2; 3; 4]
\end{verbatim}

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

\begin{verbatim}
# ListLabels.fold_left (+) 0 [1;2;3];;
Error: This expression has type int -> int -> int
but an expression was expected of type 'a list
\end{verbatim}

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.

\textsuperscript{1}This 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 (\texttt{~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 by totally applying the function, omitting all optional arguments and omitting all labels for all remaining arguments.

# 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 \( \sim y \) and then \( \sim x \), but \( f \) expects \( \sim x \) and then \( \sim y \). This is correctly handled if we know the type of \( g \) to be \( x : \text{int} \to y : \text{int} \to \text{int} \) 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 \( \text{bump} \).

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

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

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

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

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

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

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

### 4.1.3 Suggestions for labeling

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

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

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

To speak in an “object-oriented” way, one can consider that each function has a main argument, its object, and other arguments related with its action, the parameters. To permit the combination of functions through functionals in commuting label mode, the object will not be labeled. Its role is clear 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 a 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 `.

```plaintext
# `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.

```plaintext
# 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.

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

# f `E;;
val f `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.

```ml
# let f1 = function `A x -> x = 1 | `B -> true | `C -> false
val f1 : (>`A of int | `B | `C) -> bool = <fun>

# let f2 = function `A x -> x = "a" | `B -> true
val f2 : (>`A of string | `B) -> bool = <fun>

# let f x = f1 x && f2 x;;
val f x = f1 x && f2 x
```

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.

```ml
# 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 1 = (1 : 'a vlist -> 'a wlist);;
val wlist_of_vlist : 'a vlist -> 'a wlist = <fun>

# let open_vlist 1 = (1 : 'a vlist -> [`a vlist]);;
val open_vlist : 'a vlist -> ['a vlist] = <fun>

```

You may also selectively coerce values through pattern matching.

```ml
# 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.

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

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

To make this even more comfortable, you may use type definitions as abbreviations for or-patterns. That is, if you have defined

```hsaskell
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,

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

or combined with with aliases.

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

# 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

Polymorphism and its limitations

This chapter covers more advanced questions related to the limitations of polymorphic functions and types. There are some situations in OCaml where the type inferred by the type checker may be less generic than expected. Such non-genericity can stem either from interactions between side-effect and typing or the difficulties of implicit polymorphic recursion and higher-rank polymorphism.

This chapter details each of these situations and, if it is possible, how to recover genericity.

5.1 Weak polymorphism and mutation

5.1.1 Weakly polymorphic types

Maybe the most frequent examples of non-genericity derive from the interactions between polymorphic types and mutation. A simple example appears when typing the following expression

```
# let store = ref None ;;
val store : '_weak1 option ref = {contents = None}
```

Since the type of `None` is `'a option` and the function `ref` has type `'b -> 'b ref`, a natural deduction for the type of `store` would be `'a option ref`. However, the inferred type, `'_weak1 option ref`, is different. Type variables whose name starts with a `_weak` prefix like `'_weak1` are weakly polymorphic type variables, sometimes shortened as weak type variables. A weak type variable is a placeholder for a single type that is currently unknown. Once the specific type `t` behind the placeholder type `'_weak1` is known, all occurrences of `'_weak1` will be replaced by `t`. For instance, we can define another option reference and store an `int` inside:

```
# let another_store = ref None ;;
val another_store : '_weak2 option ref = {contents = None}

# another_store := Some 0;
# another_store ;;
- : int option ref = {contents = Some 0}
```

After storing an `int` inside `another_store`, the type of `another_store` has been updated from `'_weak2 option ref` to `int option ref`. This distinction between weakly and generic polymorphic type variable protects OCaml programs from unsoundness and runtime errors. To understand
from where unsoundness might come, consider this simple function which swaps a value x with the value stored inside a store reference, if there is such value:

```ocaml
# let swap store x = match !store with
#   | None -> store := Some x; x
#   | Some y -> store := Some x; y;;
val swap : 'a option ref -> 'a -> 'a = <fun>
```

We can apply this function to our store

```ocaml
# let one = swap store 1
# let one_again = swap store 2
# let two = swap store 3;;
val one : int = 1
val one_again : int = 1
val two : int = 2
```

After these three swaps the stored value is 3. Everything is fine up to now. We can then try to swap 3 with a more interesting value, for instance a function:

```ocaml
# let error = swap store (fun x -> x);;
Error: This expression should not be a function, the expected type is int
```

At this point, the type checker rightfully complains that it is not possible to swap an integer and a function, and that an int should always be traded for another int. Furthermore, the type checker prevents us to change manually the type of the value stored by store:

```ocaml
# store := Some (fun x -> x);;
Error: This expression should not be a function, the expected type is int
```

Indeed, looking at the type of store, we see that the weak type '_weak1 has been replaced by the type int

```ocaml
# store;;
- : int option ref = {contents = Some 3}
```

Therefore, after placing an int in store, we cannot use it to store any value other than an int. More generally, weak types protect the program from undue mutation of values with a polymorphic type.

Moreover, weak types cannot appear in the signature of toplevel modules: types must be known at compilation time. Otherwise, different compilation units could replace the weak type with different and incompatible types. For this reason, compiling the following small piece of code

```ocaml
let option_ref = ref None
```

yields a compilation error

```ocaml
Error: The type of this expression, '_weak1 option ref, contains type variables that cannot be generalized
```

To solve this error, it is enough to add an explicit type annotation to specify the type at declaration time:
let option_ref: int option ref = ref None

This is in any case a good practice for such global mutable variables. Otherwise, they will pick
out the type of first use. If there is a mistake at this point, this can result in confusing type errors
when later, correct uses are flagged as errors.

5.1.2 The value restriction

Identifying the exact context in which polymorphic types should be replaced by weak types in
a modular way is a difficult question. Indeed the type system must handle the possibility that
functions may hide persistent mutable states. For instance, the following function uses an internal
reference to implement a delayed identity function

```ocaml
# let make_fake_id () =
#   let store = ref None in
#   fun x -> swap store x ;;
val make_fake_id : unit -> 'a -> 'a = <fun>
# let fake_id = make_fake_id();;
val fake_id : '_weak3 -> '_weak3 = <fun>
```

It would be unsound to apply this `fake_id` function to values with different types. The function
`fake_id` is therefore rightfully assigned the type `'weak3 -> '_weak3` rather than `'a -> 'a`. At the
same time, it ought to be possible to use a local mutable state without impacting the type of a
function.

To circumvent these dual difficulties, the type checker considers that any value returned by a
function might rely on persistent mutable states behind the scene and should be given a weak type.
This restriction on the type of mutable values and the results of function application is called
the value restriction. Note that this value restriction is conservative: there are situations where the
value restriction is too cautious and gives a weak type to a value that could be safely generalized
to a polymorphic type:

```ocaml
# let not_id = (fun x -> x) (fun x -> x);;
val not_id : '_weak4 -> '_weak4 = <fun>
```

Quite often, this happens when defining function using higher order function. To avoid this problem,
a solution is to add an explicit argument to the function:

```ocaml
# let id_again = fun x -> (fun x -> x) (fun x -> x) x;;
val id_again : 'a -> 'a = <fun>
```

With this argument, `id_again` is seen as a function definition by the type checker and can there-
fore be generalized. This kind of manipulation is called eta-expansion in lambda calculus and is
sometimes referred under this name.

5.1.3 The relaxed value restriction

There is another partial solution to the problem of unnecessary weak type, which is implemented
directly within the type checker. Briefly, it is possible to prove that weak types that only appear
as type parameters in covariant positions –also called positive positions– can be safely generalized
to polymorphic types. For instance, the type `'a list` is covariant in `'a`:
# let f () = [];;
val f : unit -> 'a list = <fun>

# let empty = f ();;
val empty : 'a list = []

Remark that the type inferred for empty is 'a list and not '_weak5 list that should have occurred with the value restriction since f () is a function application.

The value restriction combined with this generalization for covariant type parameters is called the relaxed value restriction.

5.1.4 Variance and value restriction

Variance describes how type constructors behave with respect to subtyping. Consider for instance a pair of type x and xy with x a subtype of xy, denoted x :> xy:

# type x = [ 'X ];;
type x = [ 'X ]

# type xy = [ 'X | 'Y ];;
type xy = [ 'X | 'Y ]

As x is a subtype of xy, we can convert a value of type x to a value of type xy:

# let x:x = 'X;;
val x : x = 'X

# let x' = ( x :> xy );;
val x' : xy = 'X

Similarly, if we have a value of type x list, we can convert it to a value of type xy list, since we could convert each element one by one:

# let l:x list = [ 'X; 'X ];;
val l : x list = [ 'X; 'X ]

# let l' = ( l :> xy list );;
val l' : xy list = [ 'X; 'X ]

In other words, x :> xy implies that x list :> xy list, therefore the type constructor 'a list is covariant (it preserves subtyping) in its parameter 'a.

Contrarily, if we have a function that can handle values of type xy

# let f: xy -> unit = function
#   | 'X -> ()
#   | 'Y -> ();;
val f : xy -> unit = <fun>

it can also handle values of type x:

# let f' = (f :> x -> unit );;
val f' : x -> unit = <fun>
Note that we can rewrite the type of \(f\) and \(f'\) as

\[
\text{type } 'a \text{ proc } = 'a \to \text{unit} \\
\text{let } f' = (f: xy \text{ proc } :> x \text{ proc}); ; \\
\text{type } 'a \text{ proc } = 'a \to \text{unit} \\
\text{val } f': x \text{ proc } = \langle \text{fun} \rangle
\]

In this case, we have \(x :> xy\) implies \(xy \text{ proc } :> x \text{ proc}\). Notice that the second subtyping relation reverse the order of \(x\) and \(xy\): the type constructor \('a \text{ proc}\) is contravariant in its parameter \('a\). More generally, the function type constructor \('a \to 'b\) is covariant in its return type \('b\) and contravariant in its argument type \('a\).

A type constructor can also be invariant in some of its type parameters, neither covariant nor contravariant. A typical example is a reference:

\[
\text{let } x: x \text{ ref } = \text{ref } \langle X \rangle ; ; \\
\text{val } x : x \text{ ref } = \{ \text{contents } = \langle X \rangle \}
\]

If we were able to coerce \(x\) to the type \(xy \text{ ref}\) as a variable \(xy\), we could use \(xy\) to store the value \(\langle Y \rangle\) inside the reference and then use the \(x\) value to read this content as a value of type \(x\), which would break the type system.

More generally, as soon as a type variable appears in a position describing mutable state it becomes invariant. As a corollary, covariant variables will never denote mutable locations and can be safely generalized. For a better description, interested readers can consult the original article by Jacques Garrigue on [http://www.math.nagoya-u.ac.jp/~garrigue/papers/morepoly-long.pdf](http://www.math.nagoya-u.ac.jp/~garrigue/papers/morepoly-long.pdf)

Together, the relaxed value restriction and type parameter covariance help to avoid eta-expansion in many situations.

### 5.1.5 Abstract data types

Moreover, when the type definitions are exposed, the type checker is able to infer variance information on its own and one can benefit from the relaxed value restriction even unknowingly. However, this is not the case anymore when defining new abstract types. As an illustration, we can define a module type collection as:

\[
\text{module type COLLECTION } = \text{sig} \\
\text{type } 'a \ t \\
\text{val empty: unit } \to 'a \ t \\
\text{end} \\
\text{module Implementation } = \text{struct} \\
\text{type } 'a \ t = 'a \text{ list} \\
\text{let empty ()= [ ]} \\
\text{end} ; ; \\
\text{module type COLLECTION } = \text{sig type } 'a \ t \text{ val empty : unit } \to 'a \ t \text{ end} \\
\text{module Implementation } : \\
\text{ sig type } 'a \ t = 'a \text{ list} \text{ val empty : unit } \to 'a \text{ list } \text{ end}
\]

\[
\text{module List2: COLLECTION } = \text{Implementation}; ; \\
\text{module List2 : COLLECTION}
\]
In this situation, when coercing the module List2 to the module type COLLECTION, the type checker forgets that 'a List2.t was covariant in 'a. Consequently, the relaxed value restriction does not apply anymore:

```ocaml
# List2.empty ();;
- : '_weak5 List2.t = <abstr>
```

To keep the relaxed value restriction, we need to declare the abstract type 'a COLLECTION.t as covariant in 'a:

```ocaml
# module type COLLECTION = sig
# type +'a t
# val empty: unit -> 'a t
# end
# module List2: COLLECTION = Implementation;;
module type COLLECTION = sig type +'a t val empty : unit -> 'a t end
module List2 : COLLECTION
```

We then recover polymorphism:

```ocaml
# List2.empty ();;
- : 'a List2.t = <abstr>
```

## 5.2 Polymorphic recursion

The second major class of non-genericity is directly related to the problem of type inference for polymorphic functions. In some circumstances, the type inferred by OCaml might be not general enough to allow the definition of some recursive functions, in particular for recursive function acting on non-regular algebraic data type.

With a regular polymorphic algebraic data type, the type parameters of the type constructor are constant within the definition of the type. For instance, we can look at arbitrarily nested list defined as:

```ocaml
# type 'a regular_nested = List of 'a list | Nested of 'a regular_nested list
# let l = Nested[ List [1]; Nested [List[2;3]]; Nested[Nested[]] ];;
```

Note that the type constructor regular_nested always appears as 'a regular_nested in the definition above, with the same parameter 'a. Equipped with this type, one can compute a maximal depth with a classic recursive function

```ocaml
# let rec maximal_depth = function
# | List _ -> 1
# | Nested [] -> 0
# | Nested (a:q) -> 1 + max (maximal_depth a) (maximal_depth (Nested q));;
val maximal_depth : 'a regular_nested -> int = <fun>
```
Non-regular recursive algebraic data types correspond to polymorphic algebraic data types whose parameter types vary between the left and right side of the type definition. For instance, it might be interesting to define a datatype that ensures that all lists are nested at the same depth:

```ocaml
# type 'a nested = List of 'a list | Nested of 'a list nested;;
```

Intuitively, a value of type `'a nested` is a list of list ... of list of elements `a` with `k` nested list. We can then adapt the `maximal_depth` function defined on `regular_depth` into a `depth` function that computes this `k`. As a first try, we may define

```ocaml
# let rec depth = function
    # | List _ -> 1
    # | Nested n -> 1 + depth n;;
```

The type error here comes from the fact that during the definition of `depth`, the type checker first assigns to `depth` the type `'a -> 'b`. When typing the pattern matching, `'a -> 'b` becomes `'a nested -> 'b`, then `'a nested -> int` once the `List` branch is typed. However, when typing the application `depth n` in the `Nested` branch, the type checker encounters a problem: `depth n` is applied to `'a list nested`, it must therefore have the type `'a list nested -> 'b`. Unifying this constraint with the previous one leads to the impossible constraint `'a list nested = 'a nested`. In other words, within its definition, the recursive function `depth` is applied to values of type `'a t` with different types `'a` due to the non-regularity of the type constructor `nested`. This creates a problem because the type checker had introduced a new type variable `'a` only at the definition of the function `depth` whereas, here, we need a different type variable for every application of the function `depth`.

### 5.2.1 Explicitly polymorphic annotations

The solution of this conundrum is to use an explicitly polymorphic type annotation for the type `'a`:

```ocaml
# let rec depth: 'a. 'a nested -> int = function
    # | List _ -> 1
    # | Nested n -> 1 + depth n;;
```

In the type of `depth`, `'a. 'a nested -> int`, the type variable `'a` is universally quantified. In other words, `'a. 'a nested -> int` reads as “for all type `'a`, `depth` maps `'a nested` values to integers”. Whereas the standard type `'a nested -> int` can be interpreted as “let be a type variable `'a`, then `depth` maps `'a nested` values to integers”. There are two major differences with these two type expressions. First, the explicit polymorphic annotation indicates to the type checker that it needs
to introduce a new type variable every times the function \texttt{depth} is applied. This solves our problem with the definition of the function \texttt{depth}.

Second, it also notifies the type checker that the type of the function should be polymorphic. Indeed, without explicit polymorphic type annotation, the following type annotation is perfectly valid

```ocaml
# let sum: 'a -> 'b -> 'c = fun x y -> x + y;;
val sum : int -> int -> int = <fun>
```

since \texttt{'a,'b} and \texttt{'c} denote type variables that may or may not be polymorphic. Whereas, it is an error to unify an explicitly polymorphic type with a non-polymorphic type:

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

An important remark here is that it is not needed to explicit fully the type of \texttt{depth}: it is sufficient to add annotations only for the universally quantified type variables:

```ocaml
# let rec depth: 'a. 'a nested -> _ = function
# | List _ -> 1
# | Nested n -> 1 + depth n;;
val depth : 'a nested -> int = <fun>
```

5.2.2 More examples

With explicit polymorphic annotations, it becomes possible to implement any recursive function that depends only on the structure of the nested lists and not on the type of the elements. For instance, a more complex example would be to compute the total number of elements of the nested lists:

```ocaml
# let len nested =
#   let map_and_sum f = List.fold_left (fun acc x -> acc + f x) 0 in
#   let rec len: 'a. ('a list -> int) -> 'a nested -> int =
#     fun nested_len n ->
#       match n with
#       | List l -> nested_len l
#       | Nested n -> len (map_and_sum nested_len) n
#     in
#   len List.length nested;;
val len : 'a nested -> int = <fun>
```

Similarly, it may be necessary to use more than one explicitly polymorphic type variables, like for computing the nested list of list lengths of the nested list:
Chapter 5. Polymorphism and its limitations

5.3 Higher-rank polymorphic functions

Explicit polymorphic annotations are however not sufficient to cover all the cases where the inferred type of a function is less general than expected. A similar problem arises when using polymorphic functions as arguments of higher-order functions. For instance, we may want to compute the average depth or length of two nested lists:

```
# let average_depth x y = (depth x + depth y) / 2;;
val average_depth : 'a nested -> 'b nested -> int = <fun>
# let average_len x y = (len x + len y) / 2;;
val average_len : 'a nested -> 'b nested -> int = <fun>
# let one = average_len (List [2]) (List [[]]);;
val one : int = 1
```

It would be natural to factorize these two definitions as:

```
# let average f x y = (f x + f y) / 2;;
val average : ('a -> int) -> 'a -> 'a -> int = <fun>
```

However, the type of `average_len` is less generic than the type of `average_len`, since it requires the type of the first and second argument to be the same:

```
# average_len (List [2]) (List [[]]);;
- : int = 1
```

```
# average_len (List [2]) (List [[]]);;
Error: This expression has type 'a list
but an expression was expected of type int
```
As previously with polymorphic recursion, the problem stems from the fact that type variables are introduced only at the start of the let definitions. When we compute both \( f \ x \) and \( f \ y \), the type of \( x \) and \( y \) are unified together. To avoid this unification, we need to indicate to the type checker that \( f \) is polymorphic in its first argument. In some sense, we would want \( \text{average} \) to have type

\[
\text{val average}: (\text{'a nested} \to \text{int}) \to \text{'a nested} \to \text{'b nested} \to \text{int}
\]

Note that this syntax is not valid within OCaml: \( \text{average} \) has an universally quantified type \( \text{'a} \) inside the type of one of its argument whereas for polymorphic recursion the universally quantified type was introduced before the rest of the type. This position of the universally quantified type means that \( \text{average} \) is a second-rank polymorphic function. This kind of higher-rank functions is not directly supported by OCaml: type inference for second-rank polymorphic function and beyond is undecidable; therefore using this kind of higher-rank functions requires to handle manually these universally quantified types.

In OCaml, there are two ways to introduce this kind of explicit universally quantified types: universally quantified record fields,

```ocaml
# type 'a nested_reduction = { f:'elt. 'elt nested -> 'a };;
val type 'a nested_reduction : ('elt -> 'a) -> 'elt nested_reduction

# let boxed_len = { f = len };;
val boxed_len : int nested_reduction = {f = <fun>}
```

and universally quantified object methods:

```ocaml
# let obj_len = object method f:'a. 'a nested -> 'b = len end;;
val obj_len : < f : 'a. 'a nested -> int > = <obj>
```

To solve our problem, we can therefore use either the record solution:

```ocaml
# let average nsm x y = (nsm.f x + nsm.f y) / 2 ;;
val average : int nested_reduction -> 'a nested -> 'b nested -> int = <fun>
```

or the object one:

```ocaml
# let average (obj:<f:'a. 'a nested -> _ > ) x y = (obj#f x + obj#f y) / 2 ;;
val average : < f : 'a. 'a nested -> int > -> 'b nested -> 'c nested -> int = <fun>
```
Chapter 6

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 as virtual types through the example of window managers.

6.1 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
```

87
val mutable balance : Euro.c
method balance : Euro.c
method deposit : Euro.c -> unit
method withdraw : Euro.c -> Euro.c
end

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

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

# class account_with_interests =
# object (self)
#   inherit account
#   method private interest = self # deposit (self # balance # times 0.03)
# end;
class account_with_interests :
  object
  val mutable balance : Euro.c
  method balance : Euro.c
  method deposit : Euro.c -> unit
  method private interest : unit
  method withdraw : Euro.c -> Euro.c
end

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

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

# class safe_account =
# object
#   inherit account
#   method deposit x = if zero#leq x then balance <- balance#plus x
# end;
class safe_account :
  object
    val mutable balance : Euro.c
    method balance : Euro.c
    method deposit : Euro.c -> unit
    method withdraw : Euro.c -> Euro.c
  end

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

# class safe_account =
# object
#   inherit account as unsafe
#   method deposit x =
Chapter 6. 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

```ocaml
# 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 -> Euro.c
    end

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

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

class account_with_deposit :
    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 6. 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
#   method virtual deposit: m -> unit
# 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
#     method history : m operation list
#     method withdraw : m -> m
#     method balance : m
#     method mail : string -> unit
#   end
# module Client (B : sig class bank : client_view end) =
#   struct
#     class account x : client_view =
#       object
#       inherit B.bank
#       inherit A.check_client x
#       end
#     end
#   end
#

6.2 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.

6.2.1 Strings

A naive definition of strings as objects could be:

# 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.
As seen in section 3.16, the solution is to use functional update instead. We need to create an instance variable containing the representation \( s \) of the string.

```
# class better_string s =
#    object
#        val repr = s
#        method get n = String.get repr n
#        method 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 (a)
        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:

```
# 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 >}
```
Chapter 6. Advanced examples with classes and modules

```
# method concat (t : 'mytype) = {< repr = repr ^ t#repr >}
# end;;

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;;

exception Empty

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

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 : ('_weak1, '_weak2) 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 \( \text{Forall} \ 'b. ('b -> 'a -> 'b) -> 'b -> 'b \). 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

6.2.2 Hashtbl

A simplified version of object-oriented hash tables should have the following class type.
Chapter 6. Advanced examples with classes and modules

# 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:

# 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!

# 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

6.2.3 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.

# module type SET =
#   sig
#     type 'a tag
#     class ['a] c :

6.3 The subject/observer pattern

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

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

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

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

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

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.

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

class ['a] window_subject :
    object ('b)
        constraint 'a = < notify : 'b -> event -> unit; .. >
        val mutable observers : 'a list
val mutable position : int
method add_observer : 'a -> unit
method draw : unit
method identity : int
method move : int -> unit
method notify_observers : event -> unit
end

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

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

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 = ()

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

Classes window_observer and window_subject can still be extended by inheritance. For instance, one may enrich the subject with new behaviors and 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 notify_observers : event -> unit
method raise : unit
method resize : int -> unit
end

# class [subject] richer_window_observer =
# object
#    inherit [subject] window_observer as super
#    method notify s e = if e <> Raise then s#raise; super#notify s e
# end;
class [a] richer_window_observer :
    object
        constraint 'a = < draw : unit; raise : unit; .. >
        method notify : 'a -> event -> unit
    end

We can also create a different kind of observer:

# 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#addd_observer (new richer_window_observer);
- : unit = ()

# window#addd_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 7

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.

7.1 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

\[
\text{ident} ::= \ (\text{letter} \mid \_) \ \{\text{letter} \mid 0\ldots9 \mid \_ \mid '\}
\]

\[
\text{capitalized-ident} ::= \ (A\ldotsZ) \ \{\text{letter} \mid 0\ldots9 \mid \_ \mid '\}
\]

\[
\text{lowercase-ident} ::= \ (a\ldotsz \mid \_) \ \{\text{letter} \mid 0\ldots9 \mid \_ \mid '\}
\]

\[
\text{letter} ::= A\ldotsZ | a\ldotsz
\]

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

\[
\text{integer-literal} ::= [-] \ (0\ldots9) \ {0\ldots9 \mid \_}
\]

\[
\mid [-] \ (0x \mid 0X) \ (0\ldots9 \mid A\ldotsF \mid a\ldotsf) \ {0\ldots9 \mid A\ldotsF \mid a\ldotsf \mid \_}
\]

\[
\mid [-] \ (0o \mid 0O) \ (0\ldots7) \ {0\ldots7 \mid \_}
\]

\[
\mid [-] \ (0b \mid 0B) \ (0\ldots1) \ {0\ldots1 \mid \_}
\]

\[
\text{int32-literal} ::= \text{integer-literal} \ l
\]

\[
\text{int64-literal} ::= \text{integer-literal} \ L
\]

\[
\text{nativeint-literal} ::= \text{integer-literal} \ n
\]

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</td>
</tr>
<tr>
<td>0o, 0O</td>
<td>octal</td>
</tr>
<tr>
<td>0b, 0B</td>
<td>binary</td>
</tr>
</tbody>
</table>

(The initial 0 is the digit zero; the 0 for octal is the letter O.) An integer literal can be followed by one of the letters 1, L or n to indicate that this integer has type \text{int32}, \text{int64} or \text{nativeint} respectively, instead of the default type \text{int} for integer literals. The interpretation of integer literals that fall outside the range of representable integer values is undefined.

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

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

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 \texttt{e} or \texttt{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 \texttt{0x} or \texttt{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 \texttt{p} or \texttt{P}. It is written in decimal and interpreted as a power of 2.

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

Character literals

\[
\text{char-literal} ::= ' \text{regular-char}'
\|
\text{escape-sequence}'
\]

\[
\text{escape-sequence} ::= \ \text{(\textbackslash (backslash) \textbackslash)} \text{(double quote) \textbackslash (single quote) \textbackslash (space)}
\|
\ \text{(0 \ldots 9 \{0 \ldots 9 \} \{0 \ldots 9 \})}
\|
\ \text{x (0 \ldots 9 | A \ldots F | a \ldots f) \{0 \ldots 9 | A \ldots F | a \ldots f \}}
\|
\ \text{o (0 \ldots 3 \{0 \ldots 7 \} \{0 \ldots 7 \})}
\]

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

<table>
<thead>
<tr>
<th>Sequence</th>
<th>Character denoted</th>
</tr>
</thead>
<tbody>
<tr>
<td>\textbackslash</td>
<td>backslash (\textbackslash)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>double quote (\textbackslash)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>single quote (\textbackslash)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>linefeed (LF)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>carriage return (CR)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>horizontal tabulation (TAB)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>backspace (BS)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>space (SPC)</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>the character with ASCII code \textbackslashddd \textbackslash in decimal</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>the character with ASCII code \textbackslashhh \textbackslash in hexadecimal</td>
</tr>
<tr>
<td>\textbackslash</td>
<td>the character with ASCII code \textbackslashooo \textbackslash in octal</td>
</tr>
</tbody>
</table>

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

\[
\text{string-literal} \ ::= \ " \{\text{string-character}\} " \\
| \ \{\text{quoted-string-id} | \{\text{any-char} \} | \text{quoted-string-id} \}
\]

\[
\text{quoted-string-id} \ ::= \ \{a...z | \_\}
\]

\[
\text{string-character} \ ::= \ \text{regular-string-char} \\
| \ \text{escape-sequence} \\
| \ \backslash \{ \{0...9 | A...F | a...f\}^+ \} \\
| \ \backslash \text{newline} \{\text{space} | \text{tab}\}
\]

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, or a Unicode character escape sequence.

A Unicode character escape sequence is substituted by the UTF-8 encoding of the specified Unicode scalar value. The Unicode scalar value, an integer in the ranges 0x0000...0xD7FF or 0xE000...0x10FFFF, is defined using 1 to 6 hexadecimal digits; leading zeros are allowed.

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.

Quoted string literals provide an alternative lexical syntax for string literals. They are useful to represent strings of arbitrary content without escaping. Quoted strings are delimited by a matching pair of \{ quoted-string-id \} and \{ quoted-string-id \} with the same quoted-string-id on both sides. Quoted strings do not interpret any character in a special way but requires that the sequence \{ quoted-string-id \} does not occur in the string itself. The identifier quoted-string-id is a (possibly empty) sequence of lowercase letters and underscores that can be freely chosen to avoid such issue (e.g. \{|hello|\}, \{|ext|hello \{|world|\}\}|ext\}, ...).

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.

\[
\text{label-name} \ ::= \ \text{lowercase-ident} \\
\text{label} \ ::= \ ~ \text{label-name} : \\
\text{optlabel} \ ::= \ ? \ \text{label-name} :
\]

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

\[
\text{infix-symbol ::= ( } | \text{ operator-char } | \text{ operator-char }^+ \\
| \text{ operator-char }^-
\]

\[
\text{prefix-symbol ::= ! } \{ \text{ operator-char } \} \\
| ( ? \text{ operator-char } )^-
\]

\[
\text{operator-char ::= ! | $ | % | \& | \* | + | - | . | / \text{ operator-char }^-}
\]

See also the following language extensions: extension operators and extended indexing operators.

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>
<tr>
<td>lxor</td>
<td>match</td>
<td>method</td>
<td>mod</td>
<td>module</td>
<td>mutable</td>
</tr>
<tr>
<td>new</td>
<td>nonrec</td>
<td>object</td>
<td>of</td>
<td>open</td>
<td>or</td>
</tr>
<tr>
<td>private</td>
<td>rec</td>
<td>sig</td>
<td>struct</td>
<td>then</td>
<td>to</td>
</tr>
<tr>
<td>true</td>
<td>try</td>
<td>type</td>
<td>val</td>
<td>virtual</td>
<td>when</td>
</tr>
<tr>
<td>while</td>
<td>with</td>
<td></td>
<td></td>
<td></td>
<td></td>
</tr>
</tbody>
</table>

The following character sequences are also keywords:

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

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.
7.2 Values

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

7.2.1 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 $2^{57} - 9$.

7.2.2 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).
7.2.3 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).

7.2.4 Arrays

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

7.2.5 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 \(\text{constr} \); the latter case is written \(\text{constr}(v_1, \ldots, v_n)\), where the \( v_i \) are said to be the arguments of the non-constant constructor \(\text{constr}\). 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.

7.2.6 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`.

7.2.7 Functions

Functional values are mappings from values to values.

7.2.8 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.
7.3 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 7.1),
- 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

\[
\text{value-name} ::= \text{lowercase-ident} \\
\text{operator-name} ::= \text{prefix-symbol} | \text{infix-op} \\
\text{infix-op} ::= \text{infix-symbol} \\
\text{constr-name} ::= \text{capitalized-ident} \\
\text{tag-name} ::= \text{capitalized-ident} \\
\text{typeconstr-name} ::= \text{lowercase-ident} \\
\text{field-name} ::= \text{lowercase-ident} \\
\text{module-name} ::= \text{capitalized-ident} \\
\text{modtype-name} ::= \text{ident} \\
\text{class-name} ::= \text{lowercase-ident} \\
\text{inst-var-name} ::= \text{lowercase-ident} \\
\text{method-name} ::= \text{lowercase-ident}
\]

See also the following language extension: extended indexing operators.

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

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

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

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.
7.4 Type expressions

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

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

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

See also the following language extensions: first-class modules, attributes and extension nodes.

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>-&gt;</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 7.9.1); 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 (typeexpr) denotes the same type as typeexpr.

**Function types**

The type expression typeexpr₁ → typeexpr₂ denotes the type of functions mapping arguments of type typeexpr₁ to results of type typeexpr₂.

  - label-name : typeexpr₁ → typeexpr₂ denotes the same function type, but the argument is labeled label.
  - ? label-name : typeexpr₁ → typeexpr₂ denotes the type of functions mapping an optional labeled argument of type typeexpr₁ to results of type typeexpr₂. That is, the physical type of the function will be typeexpr₁ option → typeexpr₂.

**Tuple types**

The type expression typeexpr₁ × ... × typeexprₙ denotes the type of tuples whose elements belong to types typeexpr₁, ... typeexprₙ respectively.

**Constructed types**

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

  - The type expression typeexpr typeconstr, where typeconstr is a type constructor with one parameter, denotes the application of the unary type constructor typeconstr to the type typeexpr.
  - The type expression (typeexpr₁, ..., typeexprₙ) typeconstr, where typeconstr is a type constructor with n parameters, denotes the application of the n-ary type constructor typeconstr to the types typeexpr₁ through typeexprₙ.
  - 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 typeexpr as 'ident denotes the same type as typeexpr, and also binds the type variable ident to type typeexpr both in typeexpr 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 typeexpr, 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 typeexpr denotes either an object or polymorphic variant type, the row variable of typeexpr is captured by 'ident, and quantified upon.
Polymorphic variant types

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 \texttt{`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 \texttt{< \{} \texttt{method-type \{} ; method-type \}} \texttt{>} is a record of method types. Each method may have an explicit polymorphic type: \texttt{\{} \texttt{ident} \}^+ \texttt{. 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 \texttt{< \{} \texttt{method-type \}} \texttt{. . . >} is the type of an object whose method names and types are described by \texttt{method-type}_1, \ldots, \texttt{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 \to \# class-path
is usually not the same as type

\( \# \text{class-path as 'ident} \to ' \text{ident} \).

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

**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 7.8.1.

### 7.5 Constants

\[
\text{constant} ::= \hspace{1em} \text{integer-literal} \\
| \hspace{1em} \text{int32-literal} \\
| \hspace{1em} \text{int64-literal} \\
| \hspace{1em} \text{nativeint-literal} \\
| \hspace{1em} \text{float-literal} \\
| \hspace{1em} \text{char-literal} \\
| \hspace{1em} \text{string-literal} \\
| \hspace{1em} \text{constr} \\
| \hspace{1em} \text{false} \\
| \hspace{1em} \text{true} \\
| \hspace{1em} ( ) \\
| \hspace{1em} [ ] \\
| \hspace{1em} [ | ] \\
| \hspace{1em} \text{tag-name}
\]

See also the following language extension: extension literals.

The syntactic class of constants comprises literals from the four base types (integers, floating-point numbers, characters, character strings), the integer variants, and constant constructors from both normal and polymorphic variants, as well as the special constants \text{false, true, ( ), [ ], and [ | ]}, which behave like constant constructors, and begin end, which is equivalent to ( ).
Chapter 7. The OCaml language

7.6 Patterns

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

See also the following language extensions: local opens, first-class modules, attributes and extension nodes.

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 7.6)</td>
<td>–</td>
</tr>
<tr>
<td>Constructor application, Tag application</td>
<td>right</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.

Alias patterns

The pattern \texttt{pattern\textsubscript{1} as value-name} matches the same values as \texttt{pattern\textsubscript{1}}. If the matching against \texttt{pattern\textsubscript{1}} is successful, the name \texttt{value-name} is bound to the matched value, in addition to the bindings performed by the matching against \texttt{pattern\textsubscript{1}}.

Parenthesized patterns

The pattern \texttt{( pattern\textsubscript{1} )} matches the same values as \texttt{pattern\textsubscript{1}}. A type constraint can appear in a parenthesized pattern, as in \texttt{( pattern\textsubscript{1} : typexpr )}. This constraint forces the type of \texttt{pattern\textsubscript{1}} to be compatible with \texttt{typexpr}.

“Or” patterns

The pattern \texttt{pattern\textsubscript{1} | pattern\textsubscript{2}} represents the logical “or” of the two patterns \texttt{pattern\textsubscript{1}} and \texttt{pattern\textsubscript{2}}. A value matches \texttt{pattern\textsubscript{1} | pattern\textsubscript{2}} if it matches \texttt{pattern\textsubscript{1}} or \texttt{pattern\textsubscript{2}}. The two sub-patterns \texttt{pattern\textsubscript{1}} and \texttt{pattern\textsubscript{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 \textit{v} matches \texttt{pattern\textsubscript{1} | pattern\textsubscript{2}}, the bindings performed are those of \texttt{pattern\textsubscript{1}} when \textit{v} matches \texttt{pattern\textsubscript{1}}. Otherwise, value \textit{v} matches \texttt{pattern\textsubscript{2}} whose bindings are performed.

Variant patterns

The pattern \texttt{constr ( pattern\textsubscript{1} , \ldots , pattern\textsubscript{n} )} matches all variants whose constructor is equal to \texttt{constr}, and whose arguments match \texttt{pattern\textsubscript{1} \ldots pattern\textsubscript{n}}. It is a type error if \textit{n} is not the number of arguments expected by the constructor.

The pattern \texttt{constr \_} matches all variants whose constructor is \texttt{constr}.

The pattern \texttt{pattern\textsubscript{1} :: pattern\textsubscript{2}} matches non-empty lists whose heads match \texttt{pattern\textsubscript{1}}, and whose tails match \texttt{pattern\textsubscript{2}}.

The pattern \texttt{[ pattern\textsubscript{1} ; \ldots ; pattern\textsubscript{n} ]} matches lists of length \textit{n} whose elements match \texttt{pattern\textsubscript{1} \ldots pattern\textsubscript{n}}, respectively. This pattern behaves like \texttt{pattern\textsubscript{1} :: \ldots :: pattern\textsubscript{n} :: []}.

Polymorphic variant patterns

The pattern \texttt{ ` tag-name pattern\textsubscript{1} } matches all polymorphic variants whose tag is equal to \texttt{tag-name}, and whose argument matches \texttt{pattern\textsubscript{1}}.

Polymorphic variant abbreviation patterns

If the type \texttt{[('a,'b,\ldots)] typestr = [ ` tag-name\textsubscript{1} typexpr\textsubscript{1} | \ldots | ` tag-name\textsubscript{n} typexpr\textsubscript{n} ]} is defined, then the pattern \texttt{# typestr} is a shorthand for the following or-pattern: \texttt{( ` tag-name\textsubscript{1} (_ : typexpr\textsubscript{1} ) | \ldots | ` tag-name\textsubscript{n} (_ : typexpr\textsubscript{n} ) )}. It matches all values of type \texttt{[< typestr ]}. 
 Tuple patterns

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

 Record patterns

The pattern \{ field\(_1\) = \text{pattern}_1 \}; \ldots ; field\(_n\) = \text{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 \text{pattern}_i, for \( i = 1, \ldots, n \). A single identifier field\(_k\) stands for field\(_k\) = field\(_k\), and a single qualified identifier module-path . field\(_k\) stands for module-path . field\(_k\) = field\(_k\). The record value can define more fields than field\(_1\) . . . field\(_n\); the values associated to these extra fields are not taken into account for matching. 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. 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 \[ | \text{pattern}_1 ; \ldots ; \text{pattern}_n | \] matches arrays of length \( n \) such that the \( i \)-th array element matches the pattern \text{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.

 Lazy patterns

(Introduced in Objective Caml 3.11)

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

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 ??).
Exception patterns

(Introduced in OCaml 4.02)

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

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 with a try...with block). Since the bodies of all exception and value cases are 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.

A pattern match must contain at least one value case. It is an error if all cases are exceptions, because there would be no code to handle the return of a value.
7.7 Expressions

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

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

See also the following language extensions: first-class modules, overriding in open statements, syntax for Bigarray access, attributes, extension nodes and extended indexing operators.

### 7.7.1 Precedence and associativity

The table below shows the relative precedences and associativity of operators and non-closed constructions. The constructions with higher precedence come first. For infix and prefix symbols, we write “*...” to mean “any symbol starting with *".
Construction or operator | Associativity
--- | ---
prefix-symbol
. . (. . [ . { (see section 8.12) | –
#... function application, constructor application, tag application, assert, lazy | left
- -. (prefix) | –
**... lsl lsr asr | right
*... /... %... mod land lor lxor | left
+... -... | left
:... ^... | right
=... <... >... |... &... $... != | left
& && | right
or || | right
, | –
<- := | right
if | –
; | –
let match fun function try | –

### 7.7.2 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:

```plaintext
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 7.7.7 below).
Function application

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

The order in which the expressions \( \text{expr}, \ \text{argument}_1, \ldots, \text{argument}_n \) are evaluated is not specified.

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

If a parameter is specified as optional (label prefixed by \( ? \)) in the type of \( \text{expr} \), the corresponding argument will be automatically wrapped with the constructor \( \text{Some} \), 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 \( \text{None} \) 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 \(-\text{principal} \) mode.

Function definition

Two syntactic forms are provided to define functions. The first form is introduced by the keyword \text{function}:

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

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

The other form of function definition is introduced by the keyword \text{fun}:

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

This expression is equivalent to:

\[
\text{fun} \quad \text{parameter}_1 \rightarrow \ldots \text{fun} \quad \text{parameter}_n \rightarrow \text{expr}
\]
An optional type constraint `typexpr` can be added before `->` to enforce the type of the result to be compatible with the constraint `typexpr`:

```
fun parameter_1 ... parameter_n : typexpr -> expr
```

is equivalent to

```
fun parameter_1 -> ... fun parameter_n -> (expr : typexpr)
```

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

```
fun parameter_1 ... (parameter_n : typexpr) -> expr
```

and one on the result

```
fun parameter_1 ... parameter_n : 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 = expr_0 ) -> expr` is equivalent to

```
fun ? lab : ident -> let pattern = match ident with Some ident -> ident | None -> expr_0 in expr
```

where `ident` is a fresh variable, except that it is unspecified when `expr_0` is evaluated.

After these two transformations, expressions are of the form

```
fun [label_1] pattern_1 -> ... fun [label_n] pattern_n -> expr
```

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

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

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

**Guards in pattern-matchings**

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

```
function pattern_1 [when cond_1] -> expr_1
   | ... 
   | pattern_n [when cond_n] -> expr_n
```

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

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

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

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, \(\text{expr}\) is evaluated in the environment enriched by the bindings performed during matching, and the value of \(\text{expr}\) is returned as the value of the whole 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 let expression, one may instead write

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

Recursive definitions of names are introduced by let rec:

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

The only difference with the let construct described above is that the bindings of names to values performed by the pattern-matching are considered already performed when the expressions \(\text{expr}_1\) 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 \(\text{pattern}_1, \ldots, \text{pattern}_n\), and expect them to have the same value as in \(\text{expr}\), the body of the 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 \(\text{pattern}_1 \ldots \text{pattern}_n\) are just value names, as in:

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

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

The behavior of other forms of let rec definitions is implementation-dependent. The current implementation also supports a certain class of recursive definitions of non-functional values, as explained in section 8.1.

Explicit polymorphic type annotations

(Introduced in OCaml 3.12)

Polymorphic type annotations in let-definitions behave in a way similar to polymorphic methods:

\[
\text{let pattern}_1 : \text{typ}_1 \ldots \text{typ}_n \text{ typeexpr} = \text{expr}
\]

These annotations 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.
It is possible to define local exceptions in expressions: \texttt{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). For instance, the following assertion is true:

\begin{verbatim}
let gen () = let exception A in A
let () = assert(gen () <> gen ()),
\end{verbatim}

7.7.3 Control structures

Sequence

The expression \texttt{expr\textsubscript{1}; expr\textsubscript{2}} evaluates \texttt{expr\textsubscript{1}} first, then \texttt{expr\textsubscript{2}}, and returns the value of \texttt{expr\textsubscript{2}}.

Conditional

The expression \texttt{if expr\textsubscript{1} then expr\textsubscript{2} else expr\textsubscript{3}} evaluates to the value of \texttt{expr\textsubscript{2}} if \texttt{expr\textsubscript{1}} evaluates to the boolean \texttt{true}, and to the value of \texttt{expr\textsubscript{3}} if \texttt{expr\textsubscript{1}} evaluates to the boolean \texttt{false}.

The \texttt{else expr\textsubscript{3}} part can be omitted, in which case it defaults to \texttt{else ()}.

Case expression

The expression

\begin{verbatim}
match expr with
  pattern\textsubscript{1} \rightarrow expr\textsubscript{1}
| ... 
| pattern\textsubscript{n} \rightarrow expr\textsubscript{n}
\end{verbatim}

matches the value of \texttt{expr} against the patterns \texttt{pattern\textsubscript{1}} to \texttt{pattern\textsubscript{n}}. If the matching against \texttt{pattern\textsubscript{i}} succeeds, the associated expression \texttt{expr\textsubscript{i}} is evaluated, and its value becomes the value of the whole match expression. The evaluation of \texttt{expr\textsubscript{i}} takes place in an environment enriched by the bindings performed during matching. If several patterns match the value of \texttt{expr}, the one that occurs first in the \texttt{match} expression is selected. If none of the patterns match the value of \texttt{expr}, the exception \texttt{Match_failure} is raised.

Boolean operators

The expression \texttt{expr\textsubscript{1} && expr\textsubscript{2}} evaluates to \texttt{true} if both \texttt{expr\textsubscript{1}} and \texttt{expr\textsubscript{2}} evaluate to \texttt{true}; otherwise, it evaluates to \texttt{false}. The first component, \texttt{expr\textsubscript{1}}, is evaluated first. The second component, \texttt{expr\textsubscript{2}}, is not evaluated if the first component evaluates to \texttt{false}. Hence, the expression \texttt{expr\textsubscript{1} && expr\textsubscript{2}} behaves exactly as

\begin{verbatim}
if expr\textsubscript{1} then expr\textsubscript{2} else false.
\end{verbatim}

The expression \texttt{expr\textsubscript{1} || expr\textsubscript{2}} evaluates to \texttt{true} if one of the expressions \texttt{expr\textsubscript{1}} and \texttt{expr\textsubscript{2}} evaluates to \texttt{true}; otherwise, it evaluates to \texttt{false}. The first component, \texttt{expr\textsubscript{1}}, is evaluated first. The second component, \texttt{expr\textsubscript{2}}, is not evaluated if the first component evaluates to \texttt{true}. Hence, the expression \texttt{expr\textsubscript{1} || expr\textsubscript{2}} behaves exactly as
if expr₁ then true else expr₂.

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

Loops

The expression while expr₁ do expr₂ done repeatedly evaluates expr₂ while expr₁ evaluates to true. The loop condition expr₁ is evaluated and tested at the beginning of each iteration. The whole while...done expression evaluates to the unit value ().

The expression for name = expr₁ to expr₂ do expr₃ done first evaluates the expressions expr₁ and expr₂ (the boundaries) into integer values n and p. Then, the loop body expr₃ is repeatedly evaluated in an environment where name is successively bound to the values n, n + 1, ..., p − 1, p. The loop body is never evaluated if n > p.

The expression for name = expr₁ downto expr₂ do expr₃ done evaluates similarly, except that name is successively bound to the values n, n − 1, ..., p + 1, p. The loop body is never evaluated if n < p.

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

Exception handling

The expression

```plaintext
try
  expr
with pattern₁ → expr₁
| ... 
| patternₙ → exprₙ
```

evaluates the expression expr and returns its value if the evaluation of expr does not raise any exception. If the evaluation of expr raises an exception, the exception value is matched against the patterns pattern₁ to patternₙ. If the matching against pattern₁ succeeds, the associated expression expr₁ is evaluated, and its value becomes the value of the whole try expression. The evaluation of expr₁ 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.

7.7.4 Operations on data structures

Products

The expression expr₁, ..., exprₙ evaluates to the n-tuple of the values of expressions expr₁ to exprₙ. 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 ( expr₁, ..., exprₙ ) evaluates to the n-ary variant value whose constructor is constr and whose arguments are the values of expr₁, ..., exprₙ.
The expression constr ( expr₁, ..., exprₙ ) evaluates to the variant value whose constructor is constr, and whose arguments are the values of expr₁ ... exprₙ.

For lists, some syntactic sugar is provided. The expression expr₁ :: expr₂ stands for the constructor ( :: ) applied to the arguments ( expr₁, expr₂ ), and therefore evaluates to the list whose head is the value of expr₁ and whose tail is the value of expr₂. The expression [ expr₁ ; ... ; exprₙ ] is equivalent to expr₁ :: ... :: exprₙ :: [], and therefore evaluates to the list whose elements are the values of expr₁ to exprₙ.

**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 `{ field₁ = expr₁ ; ... ; fieldₙ = exprₙ }` evaluates to the record value `field₁ = v₁;...;fieldₙ = vₙ` where vᵢ is the value of exprᵢ for i = 1, ..., n. A single identifier fieldₖ stands for fieldₖ = fieldₖ, and a qualified identifier module-path . fieldₖ stands for module-path . fieldₖ = fieldₖ. The fields field₁ to fieldₙ 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 expr₁ to exprₙ are evaluated is not specified. Optional type constraints can be added after each field `{ field₁ : typexpr₁ = expr₁ ; ... ; fieldₙ : typexprₙ = exprₙ }` to force the type of fieldₖ to be compatible with typexprₖ.

The expression `{ expr with field₁ = expr₁ ; ... ; fieldₙ = exprₙ }` builds a fresh record with fields field₁,...,fieldₙ equal to expr₁,...,exprₙ, 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 field₁,...,fieldₙ, which are initialized to expr₁,...,exprₙ. As previously, single identifier fieldₖ stands for fieldₖ = fieldₖ, a qualified identifier module-path . fieldₖ stands for module-path . fieldₖ = fieldₖ and it is possible to add an optional type constraint on each field being updated with `{ expr with field₁ : typexpr₁ = expr₁ ; ... ; fieldₙ : typexprₙ = exprₙ }`.

The expression expr₁ . field evaluates expr₁ to a record value, and returns the value associated to field in this record value.

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

**Arrays**

The expression [ expr₁ ; ... ; exprₙ ] evaluates to a n-element array, whose elements are initialized with the values of expr₁ to exprₙ 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 \( expr_1 .( expr_2 ) \leftarrow expr_3 \) modifies in-place the array denoted by \( expr_1 \), replacing element number \( expr_2 \) by the value of \( expr_3 \). The exception \texttt{Invalid_argument} is raised if the access is out of bounds. The value of the whole expression is ()

**Strings**

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

The expression \( expr_1 .[ expr_2 ] \leftarrow expr_3 \) modifies in-place the string denoted by \( expr_1 \), replacing character number \( expr_2 \) by the value of \( expr_3 \). The exception \texttt{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 \texttt{Bytes.set} function.

### 7.7.5 Operators

Symbols from the class \texttt{infix-symbol}, as well as the keywords *, +, -, =, !=, <, >, or, ||, &&, :=, mod, land, lor, lxor, lsl, lsr, and asr can appear in infix position (between two expressions).

Symbols from the class \texttt{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 \( expr_1 \ infix-symbol \ expr_2 \) is interpreted as the application \( ( infix-symbol ) \ expr_1 \ expr_2 \).

The table below lists the symbols defined in the initial environment and their initial meaning. (See the description of the core library module \texttt{Pervasives} in chapter 23 for more details). Their meaning may be changed at any time using \texttt{let ( infix-op ) name_1 name_2 = ...}.

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

### 7.7.6 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 7.9.2 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 `Oo.copy` (see section ??). 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. A single instance variable name `id` stands for `id = id`. Other instance variables have the same value in the returned object as in self.

7.7.7 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 `typ1` is a subtype of `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 $typ_1$ to type $typ_2$, if the type of $expr$ is an instance of $typ_1$ (like for a type annotation), and $typ_1$ is a subtype of $typ_2$. The type of the coerced expression is an instance of $typ_2$. If the types contain variables, they may be instantiated by the subtyping algorithm, but this is only done after determining whether $typ_1$ is a potential subtype of $typ_2$. This means that typing may fail during this latter unification step, even if some instance of $typ_1$ is a subtype of some instance of $typ_2$. 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,

$$< \text{met}_1 : typ_1 ; \ldots ; \text{met}_n : typ_n >$$

is a supertype of

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

which may contain an ellipsis .. if every $typ_i$ is a supertype of the corresponding $typ'_i$.

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

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 $self$ to the (exact) type of its class: this is allowed if the type of $self$ 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 $typ$ is a subtype of another polymorphic variant type $typ'$ if the upper bound of $typ$ (i.e. the maximum set of constructors that may appear in an instance of $typ$) is included in the lower bound of $typ'$, and the types of arguments for the constructors of $typ$ are subtypes of those in $typ'$. Namely,

$$[ [ < ] \text{ ` C}_1 \text{ of } typ_1 | \ldots | \text{ ` C}_n \text{ of } typ_n ]$$

which may be a shrinkable type, is a subtype of

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

which may be an extensible type, if every $typ_i$ is a subtype of $typ'_i$.

**Variance**

Other types do not introduce new subtyping, but they may propagate the subtyping of their arguments. For instance, $typ_1 * typ_2$ is a subtype of $typ'_1 * typ'_2$ when $typ_1$ and $typ_2$ are respectively subtypes of $typ'_1$ and $typ'_2$. For function types, the relation is more subtle: $typ_1 \rightarrow typ_2$ is a subtype of $typ'_1 \rightarrow typ'_2$ if $typ_1$ is a supertype of $typ'_1$ and $typ_2$ is a subtype of $typ'_2$. 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 7.8.1), otherwise the default is nonvariant. This is also the case for constrained arguments in type definitions.

### 7.7.8 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 7.6).

#### Local modules

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

```ocaml
let remove_duplicates comparison_fun string_list =
  let module StringSet =
    Set.Make(struct type t = string
      let compare = comparison_fun end) in
  StringSet.elements
  (List.fold_right StringSet.add string_list StringSet.empty)
val remove_duplicates : (string -> string -> int) -> string list -> string list = <fun>
```
Local opens

The expressions `let open module-path in expr` and `module-path . ( expr )` are strictly equivalent. These constructions locally open the module referred to by the module path `module-path` in the respective scope of the expression `expr`.

When the body of a local open expression 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 . [1 expr 1]` is equivalent to `module-path . ([1 expr 1]).

7.8 Type and exception definitions

7.8.1 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 ::= = [l] constr-decl {1 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
```

See also the following language extensions: private types, generalized algebraic datatypes, attributes, extension nodes, extensible variant types and inline records.

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 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 7.7.7).

For instance, \texttt{type +'a t} declares \texttt{t} as an abstract type that is covariant in its parameter; this means that if the type \texttt{τ} is a subtype of the type \texttt{σ}, then \texttt{t} is a subtype of \texttt{σ t}. Similarly, \texttt{type -'a t} declares that the abstract type \texttt{t} is contravariant in its parameter: if \texttt{τ} is a subtype of \texttt{σ}, then \texttt{σ t} is a subtype of \texttt{τ 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 \texttt{type \texttt{'}a t} means that \texttt{t} is neither a subtype nor a supertype of \texttt{σ t} if \texttt{τ} is subtype of \texttt{σ}.

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 \texttt{constraint 'ident = typexpr} allows the specification of type parameters. Any actual type argument corresponding to the type parameter \texttt{ident} has to be an instance of \texttt{typexpr} (more precisely, \texttt{ident} and \texttt{typexpr} are unified). Type variables of \texttt{typexpr} can appear in the type equation and the type declaration.

### 7.8.2 Exception definitions

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

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 \texttt{exception constr-decl} generates a new exception, distinct from all other exceptions in the system. The form \texttt{exception constr-name = constr} gives an alternate name to an existing exception.
7.9 Classes

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

7.9.1 Class types

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

\[
\text{class-type} ::= [[? \text{label-name}:] \text{typexpr ->} \text{class-type}}
\mid \text{class-body-type}
\]

\[
\text{class-body-type} ::= \text{object} \[(\text{typexpr})\] \{\text{class-field-spec}\} \text{end}
\mid \{[(\text{typexpr}, \text{typexpr})]\} \text{classtype-path}
\mid \text{let open module-path in class-body-type}
\]

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

See also the following language extensions: attributes and extension nodes.

Simple class expressions

The expression classtype-path is equivalent to the class type bound to the name classtype-path. Similarly, the expression [ typexpr\_1, \ldots, 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 -> class-type is the type of class functions (functions from values to classes) that take as argument a value of type typexpr and return as result a class of type class-type.

Class body type

The class type expression object [( typexpr )] {class-field-spec} end is the type of a class body. It specifies its instance variables and methods. In this type, typexpr is 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.

**Local opens**

Local opens are supported in class types since OCaml 4.06.

**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.
7.9.2 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.

\[
\text{class-expr ::= class-path} \\
\quad \mid \ [ \ \text{typexpr} \ \{, \ \text{typexpr}\} ] \ \text{class-path} \\
\quad \mid ( \ \text{class-expr} ) \\
\quad \mid ( \ \text{class-expr} : \ \text{class-type} ) \\
\quad \mid \ \text{class-expr} \ \{ \ \text{argument}\}^+ \\
\quad \mid \ \text{fun} \ \{\ \text{parameter}\}^+ \ \rightarrow \ \text{class-expr} \\
\quad \mid \ \text{let} \ \{\ \text{rec}\} \ \text{let-binding} \ \{\ \text{and} \ \text{let-binding}\} \ \text{in} \ \text{class-expr} \\
\quad \mid \ \text{object} \ \text{class-body} \ \text{end} \\
\quad \mid \ \text{let open} \ \text{module-path} \ \text{in} \ \text{class-expr}
\]

\[
\text{class-field ::= inherit} \ \text{class-expr} \ [\ \text{as} \ \text{lowercase-ident}] \\
\quad \mid \ \text{inherit!} \ \text{class-expr} \ [\ \text{as} \ \text{lowercase-ident}] \\
\quad \mid \ \text{val} \ [\ \text{mutable}] \ \text{inst-var-name} \ [\ : \ \text{typexpr}] = \ \text{expr} \\
\quad \mid \ \text{val!} \ [\ \text{mutable}] \ \text{inst-var-name} \ [\ : \ \text{typexpr}] = \ \text{expr} \\
\quad \mid \ \text{val} \ [\ \text{mutable}] \ \text{virtual} \ \text{inst-var-name} : \ \text{typexpr} \\
\quad \mid \ \text{val virtual} \ \text{mutable} \ \text{inst-var-name} : \ \text{typexpr} \\
\quad \mid \ \text{method} \ [\ \text{private}] \ \text{method-name} \ \{\ \text{parameter}\} \ [\ : \ \text{typexpr}] = \ \text{expr} \\
\quad \mid \ \text{method!} \ [\ \text{private}] \ \text{method-name} \ \{\ \text{parameter}\} \ [\ : \ \text{typexpr}] = \ \text{expr} \\
\quad \mid \ \text{method} \ [\ \text{private}] \ \text{method-name} : \ \text{poly-typexpr} = \ \text{expr} \\
\quad \mid \ \text{method!} \ [\ \text{private}] \ \text{method-name} : \ \text{poly-typexpr} = \ \text{expr} \\
\quad \mid \ \text{method} \ [\ \text{private}] \ \text{virtual} \ \text{method-name} : \ \text{poly-typexpr} \\
\quad \mid \ \text{method virtual} \ \text{mutable} \ \text{method-name} : \ \text{poly-typexpr} \\
\quad \mid \ \text{constraint} \ \text{typexpr} = \ \text{typexpr} \\
\quad \mid \ \text{initializer} \ \text{expr}
\]

See also the following language extensions: locally abstract types, attributes and extension nodes.

Simple class expressions

The expression \text{class-path} evaluates to the class bound to the name \text{class-path}. Similarly, the expression \[ \text{typexpr}_1, \ldots, \text{typexpr}_n \] \text{class-path} evaluates to the parametric class bound to the name \text{class-path}, in which type parameters have been instantiated respectively to \text{typexpr}_1, \ldots, \text{typexpr}_n.

The expression ( \text{class-expr} ) evaluates to the same module as \text{class-expr}.

The expression ( \text{class-expr} : \text{class-type} ) checks that \text{class-type} matches the type of \text{class-expr} (that is, that the implementation \text{class-expr} meets the type specification \text{class-type}). The whole expression evaluates to the same class as \text{class-expr}, except that all components not specified in \text{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

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

is a short form for

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

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.

Local opens

Local opens are supported in class expressions since OCaml 4.06.

Class body

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

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

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

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

The expression \( \text{inst-var-name} \leftarrow expr \) modifies in-place the current object by replacing the value associated to \( \text{inst-var-name} \) by the value of \( expr \). Of course, this instance variable must have been declared mutable.

The expression \( \{< \text{inst-var-name}_1 = expr_1 ; \ldots ; \text{inst-var-name}_n = expr_n > \} \) evaluates to a copy of the current object in which the values of instance variables \( \text{inst-var-name}_1, \ldots, \text{inst-var-name}_n \) have been replaced by the values of the corresponding expressions \( expr_1, \ldots, expr_n \).

**Virtual method definition**

A method specification is written \texttt{method [private] virtual method-name : poly-typeexpr}. 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.

**Explicit overriding**

Since Ocaml 3.12, the keywords \texttt{inherit!}, \texttt{val!} and \texttt{method!} have the same semantics as \texttt{inherit}, \texttt{val} and \texttt{method}, but they additionally require the definition they introduce to be overriding. Namely, \texttt{method!} requires \texttt{method-name} to be already defined in this class, \texttt{val!} requires \texttt{inst-var-name} to be already defined in this class, and \texttt{inherit!} requires \texttt{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.

**Constraints on type parameters**

The construct \texttt{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.

**Initializers**

A class initializer \texttt{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.

### 7.9.3 Class definitions

\[
\text{class-definition} ::= \text{class} \text{ class-binding \{and class-binding\}} \\
\text{class-binding} ::= [\text{virtual}] [\{ type-parameters \}] \text{class-name \{parameter\} :: class-type} \\
\text{= class-expr} \\
\text{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 7.4).

**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.

### 7.9.4 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.

### 7.9.5 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.

### 7.10 Module types (module specifications)

Module types are the module-level equivalent of type expressions: they specify the general shape and type properties of modules.
module-type ::= modtype-path
| sig {specification [;;]} end
| functor ( module-name : module-type ) -> module-type
| module-type -> 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

specification ::= val value-name : typexpr
| external value-name : typexpr = external-declaration
| type-definition
| exception constr-decl
| class-specification
| class-type-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

See also the following language extensions: recovering the type of a module, substitution inside a signature, type-level module aliases, attributes, extension nodes and generative functors.

7.10.1 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.

7.10.2 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.
Chapter 7. The OCaml language

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 19).

Type specifications

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

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

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

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

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

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

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

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

Class specifications are described more precisely in section 7.9.4.

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 7.9.5.

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.

7.10.3 Functor types

The module type expression functor (module-name : module-type\textsubscript{1}) -> module-type\textsubscript{2} is the type of functors (functions from modules to modules) that take as argument a module of type module-type\textsubscript{1} and return as result a module of type module-type\textsubscript{2}. The module type module-type\textsubscript{2} can use the name module-name to refer to type components of the actual argument of the functor. If the type module-type\textsubscript{2} does not depend on type components of module-name, the module type expression can be simplified with the alternative short syntax module-type\textsubscript{1} -> module-type\textsubscript{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).

7.10.4 The with operator

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

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

$$\text{sig type t module M: (sig type u end) end}$$

then S with type t=int denotes the signature

$$\text{sig type t=int module M: (sig type u end) end}$$

and S with module M = N denotes the signature

$$\text{sig type t module M: (sig type u=N.u end) end}$$

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

$$\text{functor (A: S) (B: S with type t = A.t) ...}$$

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

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

\[
\text{module-expr} ::= \text{module-path} \\
| \text{struct} [\text{module-items}] \text{end} \\
| \text{functor} (\text{module-name} : \text{module-type}) \to \text{module-expr} \\
| \text{module-expr} (\text{module-expr}) \\
| (\text{module-expr}) \\
| (\text{module-expr} : \text{module-type})
\]

\[
\text{module-items} ::= \{;\} (\text{definition} | \text{expr}) \{(;\} (\text{definition} | ;; \text{expr})\} \{;\}
\]

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

See also the following language extensions: recursive modules, first-class modules, overriding in open statements, attributes, extension nodes and generative functors.

Simple module expressions

The expression \text{module-path} evaluates to the module bound to the name \text{module-path}.

The expression \((\text{module-expr})\) evaluates to the same module as \text{module-expr}.

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

Structures

Structures \text{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 10), optional ;; are allowed after and before each definition in a structure. These ;; have no semantic meanings. Similarly, an expr preceded by ;; is allowed as a component of a structure. It is equivalent to let _ = expr, i.e. expr is evaluated for its side-effects but is not bound to any identifier. If expr is the first component of a structure, the preceding ;; can be omitted.

Value definitions
A value definition let [ rec ] let-binding { and let-binding } bind value names in the same way as a let...in... expression (see section 7.7.2). The value names appearing in the left-hand sides of the bindings are bound to the corresponding values in the right-hand sides.

A value definition external value-name : typexpr = external-declaration implements value-name as the external function specified in external-declaration (see chapter 19).

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

Exception definitions
Exceptions are defined with the syntax exception constr-decl or exception constr-name = constr.

Class definitions
A definition of one or several classes is written class class-binding { and class-binding } and consists of a sequence of mutually recursive definitions of class names. Class definitions are described more precisely in section 7.9.3.

Class type definitions
A definition of one or several classes is written class type classtype-def { and classtype-def } and consists of a sequence of mutually recursive definitions of class type names. Class type definitions are described more precisely in section 7.9.5.

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

One can write

    module module-name : module-type = module-expr

instead of

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

Another derived form is
module module-name ( name_1 : module-type_1 ) ... ( name_n : module-type_n ) = module-expr

which is equivalent to

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

Module type definitions

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

Opening a module path

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

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 you define a module S as below

module S = struct type t = int let x = 2 end

defining the module B as

module B = struct include S let y = (x + 1 : t) end

is equivalent to defining it as

module B = struct type t = S.t let x = S.x 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.

7.11.3 Functors

Functor definition

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

Functor application

The expression module-expr_1 ( module-expr_2 ) evaluates module-expr_1 to a functor and module-expr_2 to a module, and applies the former to the latter. The type of module-expr_2 must match the type expected for the arguments of the functor module-expr_1.
Chapter 7. The OCaml language

7.12 Compilation units

\[
\text{unit-interface} \ ::= \ \{ \text{specification}[:;]\}
\]
\[
\text{unit-implementation} \ ::= \ [\text{module-items}]
\]

Compilation units bridge the module system and the separate compilation system. A compilation unit is composed of two parts: an interface and an implementation. The interface contains a sequence of specifications, just as the inside of a \texttt{sig...end} signature expression. The implementation contains a sequence of definitions and expressions, just as the inside of a \texttt{struct...end} module expression. A compilation unit also has a name \texttt{unit-name}, derived from the names of the files containing the interface and the implementation (see chapter 9 for more details). A compilation unit behaves roughly as the module definition

\[
\text{module } \texttt{unit-name} : \texttt{sig} \texttt{ unit-interface end} = \texttt{struct} \texttt{ unit-implementation end}
\]

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

\[
\texttt{name}_1 : \texttt{sig specification}_1 \texttt{ end} \ldots \texttt{name}_n : \texttt{sig specification}_n \texttt{ end}
\]

where \texttt{name}_1 \ldots \texttt{name}_n are the names of the other compilation units available in the search path (see chapter 9 for more details) and \texttt{specification}_1 \ldots \texttt{specification}_n are their respective interfaces.
Chapter 8

Language extensions

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

8.1 Recursive definitions of values

(Introduced in Objective Caml 1.00)

As mentioned in section 7.7.2, the let rec binding construct, in addition to the definition of recursive functions, also supports a certain class of recursive definitions of non-functional values, such as

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

which binds \text{name}_1 to the cyclic list \(1::2::1::2::\ldots\), and \text{name}_2 to the cyclic list \(2::1::2::1::\ldots\). Informally, the class of accepted definitions consists of those definitions where the defined names occur only inside function bodies or as argument to a data constructor.

More precisely, consider the expression:

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

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

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

- \(e\) has no free occurrence of any of \(\text{name}_1\ldots\text{name}_n\)
- \(e\) is a variable
- \(e\) has the form \textbf{fun} \ldots \rightarrow \ldots
- \(e\) has the form \textbf{function} \ldots \rightarrow \ldots
- \(e\) has the form \textbf{lazy} (\ldots)
• $e$ has one of the following forms, where each one of $\text{expr}_1 \ldots \text{expr}_m$ is statically constructive with respect to $\text{name}_1 \ldots \text{name}_n$, and $\text{expr}_0$ is statically constructive with respect to $\text{name}_1 \ldots \text{name}_n$, $\text{xname}_1 \ldots \text{xname}_m$:

- let [rec] $\text{xname}_1 = \text{expr}_1$ and $\ldots$ and $\text{xname}_m = \text{expr}_m$ in $\text{expr}_0$
- let module... in $\text{expr}_1$
- constr ( $\text{expr}_1$, $\ldots$, $\text{expr}_m$ )
- `tag-name ( $\text{expr}_1$, $\ldots$, $\text{expr}_m$ )
- [$1$ $\text{expr}_1 ; \ldots ; \text{expr}_m$ $1$]
- { $\text{field}_1 = \text{expr}_1$ ; $\ldots$ ; $\text{field}_m = \text{expr}_m$ }
- { $\text{expr}_1$ with $\text{field}_2 = \text{expr}_2$ ; $\ldots$ ; $\text{field}_m = \text{expr}_m$ } where $\text{expr}_1$ is not immediately linked to $\text{name}_1 \ldots \text{name}_n$
- ( $\text{expr}_1$, $\ldots$, $\text{expr}_m$ )
- $\text{expr}_1 ; \ldots ; $ $\text{expr}_m$

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

• $e$ is $\text{name}$

• $e$ has the form $\text{expr}_1 ; \ldots ; $ $\text{expr}_m$ where $\text{expr}_m$ is immediately linked to $\text{name}$

• $e$ has the form let [rec] $\text{xname}_1 = \text{expr}_1$ and $\ldots$ and $\text{xname}_m = \text{expr}_m$ in $\text{expr}_0$ where $\text{expr}_0$ is immediately linked to $\text{name}$ or to one of the $\text{xname}_i$ such that $\text{expr}_i$ is immediately linked to $\text{name}$.

### 8.2 Recursive modules

(Introduced in Objective Caml 3.07)

\[
\text{definition} ::= \ldots \ |
\text{module rec} \ \text{module-name} : \text{module-type} = \text{module-expr} \ |
\{ \text{and} \ \text{module-name} : \text{module-type} = \text{module-expr} \} \\
\text{specification} ::= \ldots \ |
\text{module rec} \ \text{module-name} : \text{module-type} \{ \text{and} \ \text{module-name} : \text{module-type} \}
\]

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

\[
\text{module rec A : sig} \\
\text{type t = Leaf of string | Node of ASet.t} \\
\text{val compare: t -> t -> int}
\]
end = struct
  type t = Leaf of string | Node of ASet.t
  let compare t1 t2 = 
    match (t1, t2) with
    | (Leaf s1, Leaf s2) -> Stdlib.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

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

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

Currently, the compiler requires that all dependency cycles between the recursively-defined module identifiers go through at least one “safe” module. A module is “safe” if all value definitions that it contains have function types typexpr₁ \rightarrow typexpr₂. Evaluation of a recursive module definition proceeds by building initial values for the safe modules involved, binding all (functional) values to fun _ \rightarrow 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 at runtime:

module rec M : sig val f : unit \rightarrow int end = struct let f () = N.x end
and N : sig val x : int end = struct let x = M.f () end
Exception: Undefined_recursive_module ("exten.etex", 1, 43).

If there are no safe modules along a dependency cycle, an error is raised

module rec M : sig val x : int end = struct let x = N.y end
and N : sig val x : int val y:int end = struct let x = M.x let y = 0 end
Error: Cannot safely evaluate the definition of the following cycle
of recursively-defined modules: M \rightarrow N \rightarrow M.

There are no safe modules in this cycle (see manual section 8.2).
File "exten.etex", line 1, characters 18-28:
1 | module rec M : sig val x : int end = struct let x = N.y end
   ^^^^^^^^^
Module M defines an unsafe value, x .
File "exten.etex", line 2, characters 10-20:
Module \( N \) defines an unsafe value, \( x \).

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

### 8.3 Private types

Private type declarations in module signatures, of the form \texttt{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 8.3.1), for type abbreviations (section 8.3.2), and for row types (section 8.3.3).

#### 8.3.1 Private variant and record types

(Introduced in Objective Caml 3.07)

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

\[
| = \text{private [l]} \text{ constr-decl} \{ | \text{ constr-decl}\}
\]

\[
| = \text{private record-decl}
\]

Values of a variant or record type declared \texttt{private} can be de-structured normally in pattern-matching or via the \texttt{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 \texttt{private} declaration ensures that in any value of type \( M.t \), the argument to the \texttt{B} constructor is always a positive integer.

With respect to the variance of their parameters, private types are handled like abstract types. That is, if a private type has parameters, their variance is the one explicitly given by prefixing the parameter by a ‘+’ or a ‘-‘, it is invariant otherwise.
8.3.2 Private type abbreviations
(Introduced in Objective Caml 3.11)

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

Unlike a regular type abbreviation, a private type abbreviation declares a type that is distinct from its implementation type \text{typexpr}. However, coercions from the type to \text{typexpr} are permitted. Moreover, the compiler “knows” the implementation type and can take advantage of this knowledge to perform type-directed optimizations.

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 \(N.to\_int\ \ x\). Deep coercions are also supported: if \(l\) has type \(N.t\ \ \text{list}\), the coercion \((l :> \text{int}\ \text{list})\) returns the list of underlying integers, like \(\text{List.map}\ N.to\_int\ l\) but without copying the list \(l\).

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 \((l : N.t\ \text{list} :> \text{int}\ \text{list})\) for the above examples.

8.3.3 Private row types
(Introduced in Objective Caml 3.09)

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

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.
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.

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

type t = [ `A of int | `B of bool ]

can be refined in two ways. A definition [u] may add new field to [t], and the declaration

type u = 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

type v = private [< t > `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, (`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.

8.4 Local opens for patterns

(Introduced in OCaml 4.04)

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

| module-path .( pattern )
| module-path .( pattern )
| module-path .[ pattern ]
| module-path .[ pattern ]
| module-path .{ pattern }


For patterns, local opens are limited to the `module-path . ( pattern )` construction. This construction locally open the module referred to by the module path `module-path` in the scope of the pattern `pattern`.

When the body of a local open pattern is delimited by `[ ]`, `[]| []`, or `{ }`, the parentheses can be omitted. For example, `module-path . [ pattern ]` is equivalent to `module-path . ([ pattern ])`, and `module-path . [ | pattern | ]` is equivalent to `module-path . ([ | pattern | ])`.

### 8.5 Locally abstract types

(Introduced in OCaml 3.12, short syntax added in 4.03)

```plaintext
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:

```ocaml
let f = fun (type t) (foo : t list) -> ...  
```

and even use the alternative syntax for declaring functions:

```ocaml
let f (type t) (foo : t list) = ...  
```

If several locally abstract types need to be introduced, it is possible to use the syntax `fun ( type typeconstr-name1...typeconstr-name_n ) -> expr` as syntactic sugar for `fun ( type typeconstr-name1 ) -> ... -> fun ( type typeconstr-name_n ) -> expr`. For instance,

```ocaml
let f = fun (type t u v) -> fun (foo : (t * u * v) list) -> ...  
let f' (type t u v) (foo : (t * u * v) 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.

```ocaml
let f (type t) () =  
  let module M = struct exception E of t end in  
  (fun x -> M.E x), (function M.E x -> Some x | _ -> None)
```

Here is another example:

```ocaml
let sort_uniq (type s) (cmp : s -> s -> int) =  
  let module S = Set.Make(struct type t = s let compare = cmp end) in  
  fun l ->  
    S.elements (List.fold_right S.add l S.empty)
```
It is also extremely useful for first-class modules (see section 8.6) and generalized algebraic datatypes (GADTs: see section 8.11).

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

\[
\begin{array}{l}
\text{let-binding} ::= ...
| \text{value-name : type \{typeconstr-name\}+ . typexpr = expr} \\
\text{class-field} ::= ...
| \text{method [private] method-name : type \{typeconstr-name\}+ . typexpr = expr}
| \text{method! [private] method-name : type \{typeconstr-name\}+ . typexpr = expr}
\end{array}
\]

The (type 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:

\[
\begin{aligned}
\text{let rec } f : \text{type } t1 t2. \ t1 \star t2 \text{ list }&\rightarrow t1 = ...
\end{aligned}
\]

is automatically expanded into

\[
\begin{aligned}
\text{let rec } f : 't1 't2. 't1 \star 't2 \text{ list }&\rightarrow 't1 = \\
\text{fun (type } t1) (\text{type } t2) &\rightarrow (\ldots : t1 \star t2 \text{ list }\rightarrow t1)
\end{aligned}
\]

This syntax can be very useful when defining recursive functions involving GADTs, see the section 8.11 for a more detailed explanation.

The same feature is provided for method definitions.

### 8.6 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.; fewer parens required starting from 4.05)

\[
\begin{array}{l}
\text{typexpr ::= ...}
| (\text{module package-type}) \\
\text{module-expr ::= ...}
| (\text{val expr [: package-type]}) \\
\text{expr ::= ...}
| (\text{module module-expr [: package-type]}) \\
\text{pattern ::= ...}
| (\text{module module-name [: package-type]}) \\
\text{package-type ::= modtype-path}
| \text{modtype-path with package-constraint \{and package-constraint\}} \\
\text{package-constraint ::= type typeconstr = typexpr}
\end{array}
\]
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. If the module expression is already parenthesized, like the arguments of functors are, no additional parens are needed: `Map.Make(val key)`.

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 8.16. 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:

```
type picture = ...
module type DEVICE = sig
  val draw : picture -> unit
  ...
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:

```
let parse.Cmdline () = ...
```
module Device =
  (val (let device_name = parse_cmdline () in
    try Hashtbl.find devices device_name
    with Not_found ->
      Printf.eprintf "Unknown device %s\n" device_name;
    exit 2)
  ) : DEVICE)
Alternatively, the selection can be performed within a function:

let draw_using_device device_name picture =
  let module Device =
    (val (Hashtbl.find devices device_name) : DEVICE)
  in
  Device.draw picture

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

let sort (type s) (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 = <fun>

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

let make_set (type s) cmp =
  let module S = Set.Make(struct
    type t = s
    let compare = cmp
  end) in
  (module S : Set.S with type elt = s)
val make_set : ('a -> 'a -> int) -> (module Set.S with type elt = 'a) = <fun>

8.7 Recovering the type of a module

(Introduced in OCaml 3.12)

```
module-type ::= ...
| module type of module-expr
```

The construction module type of module-expr expands to the module type (signature or functor
type) inferred for the module expression 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 include construct, is to extend the sig-
nature 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.
module type MYHASH = sig
  include module type of struct include Hashtbl end
  val replace: ('a, 'b) t -> 'a -> 'b -> unit
end

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

module MyHash : MYHASH = struct
  include Hashtbl
  let replace t k v = remove t k; add t k v
end

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

module MySet : module type of Set = struct
  ...
end

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

8.8 Substituting inside a signature

8.8.1 Destructive substitutions

(Introduced in OCaml 3.12, generalized in 4.06)

\[
\text{mod-constraint} ::= ...
  \mid \text{type} [\text{type-params}] \text{typeconstr-name} := \text{tyexpr}
  \mid \text{module} \text{module-path} := \text{extended-module-path}
\]

A “destructive” substitution (with... :=...) behaves essentially like normal signature constraints (with... =...), but it additionally removes the redefined type or module from the signature.

Prior to OCaml 4.06, there were a number of restrictions: one could only remove types and modules at the outermost level (not inside submodules), and in the case of with type the definition had to be another type constructor with the same type parameters.

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

module type Printable = sig
  type t
  val print : Format.formatter -> t -> unit
end

module type Comparable = sig
  type t
  val compare : t -> t -> int
module type PrintableComparable = sig
  include Printable
  include Comparable with type t := t
end

One can also use this to completely remove a field:

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

or to rename one:

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

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

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

8.8.2 Local substitution declarations
(Introduced in OCaml 4.08)

\[
\text{specification} ::= \ldots
| \text{type type-subst \{and type-subst\}}
| \text{module module-name ::= extended-module-path}
\]

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

Local substitutions behave like destructive substitutions (with... ::=...) but instead of being
applied to a whole signature after the fact, they are introduced during the specification of the
signature, and will apply to all the items that follow.

This provides a convenient way to introduce local names for types and modules when defining
a signature:

module type S = sig
  type t
  module Sub : sig
    type outer := t
    type t
    val to_outer : t -> outer
end
module type S =
  sig type t module Sub : sig type t val to Outer : t/1 -> t/2 end end

Note that, unlike type declarations, type substitution declarations are not recursive, so substitutions like the following are rejected:

# module type S = sig
#  type 'a poly_list := [ `Cons of 'a * 'a poly_list | `Nil ]
# end ;;
Error: Unbound type constructor poly_list

8.9 Type-level module aliases

(Introduced in OCaml 4.02)

\[
\text{specification ::= ...} \\
\text{| module module-name = module-path}
\]

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 \texttt{-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 mechan-
ism. 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
-no-alias-deps -open Mylib (the last one is equivalent to adding the line open! Mylib at the top of each
file).

```
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.

```
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.

Note the use of double underscores in Mylib__A and Mylib__B. These were chosen on pur-
pose; the compiler uses the following heuristic when printing paths: given a path Lib__fooBar,
if Lib.FooBar exists and is an alias for Lib__fooBar, then the compiler will always display
Lib.FooBar instead of Lib__fooBar. This way the long Mylib__ names stay hidden and all the
user sees is the nicer dot names. This is how the OCaml standard library is compiled.

### 8.10 Overriding in open statements

(Introduced in OCaml 4.01)
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.

This is also available (since OCaml 4.06) for local opens in class expressions and class type expressions.

### 8.11 Generalized algebraic datatypes

(Introduced in OCaml 4.00)

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

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

expr ::= ...  | let open! module-path in expr

class-body-type ::= ...  | let open! module-path in class-body-type

class-expr ::= ...  | let open! module-path in class-expr
```

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:

type _ term =
  | Int : int -> int term
  | Add : (int -> int -> int) term
  | App : ('b -> 'a) term * 'b term -> 'a term

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

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 -> 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-> a). The prefix $ in $App_ 'b denotes an existential type named by the compiler (see 8.11). 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).

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

### Refutation cases  (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:

```
matching-case ::= pattern [when expr] -> expr
               | pattern -> .
```

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:

```ocaml
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 because `deep` expects a non-existing `char t` as the first element of the tuple. 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:

```ocaml
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 -> Int.to_string 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.

```ocaml
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.

```ocaml
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:

  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:

  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:

  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
          The type constructor $'arg would escape its scope
• $n$ (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 : (t,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.

### 8.12 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><code>Bigarray.Array1.get expr0 expr1</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 } &lt;= expr</td>
<td><code>Bigarray.Array1.set expr0 expr1 expr</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 , expr2 }</td>
<td><code>Bigarray.Array2.get expr0 expr1 expr2</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 , expr2 } &lt;= expr</td>
<td><code>Bigarray.Array2.set expr0 expr1 expr2 expr</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 , expr2 , expr3 }</td>
<td><code>Bigarray.Array3.get expr0 expr1 expr2 expr3</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 , expr2 , expr3 } &lt;= expr</td>
<td><code>Bigarray.Array3.set expr0 expr1 expr2 expr3 expr</code></td>
</tr>
<tr>
<td>expr0 .{ expr1 , ... , exprn }</td>
<td>`Bigarray.Genarray.get expr0 [</td>
</tr>
<tr>
<td>expr0 .{ expr1 , ... , exprn } &lt;= expr</td>
<td>`Bigarray.Genarray.set expr0 [</td>
</tr>
</tbody>
</table>

The last two entries are valid for any $n > 3$. 
8.13 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:

```
attr-id ::= lowercase-ident
        | capitalized-ident
        | attr-id . attr-id

attr-payload ::= [module-items]
               | : typexpr
               | : [specification]
               | ? pattern [when expr]
```

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

```
attribute ::= [@ attr-id attr-payload ]

expr ::= ...
       | expr attribute

typexpr ::= ...
         | typexpr attribute

pattern ::= ...
         | pattern attribute

module-expr ::= ...
         | module-expr attribute

module-type ::= ...
         | module-type attribute

class-expr ::= ...
         | class-expr attribute

class-type ::= ...
         | class-type attribute
```

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:
field-decl ::= [mutable] field-name : poly-typexpr {attribute}
constr-decl ::= (constr-name | ()) [of constr-args] {attribute}

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 { item-attribute }
    | { 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:

\[
\begin{align*}
\text{floating-attribute} & \ ::= \ [\text{@@@ attr-id attr-payload }] \\
\text{definition} & \ ::= \ ...
\quad | \quad \text{floating-attribute} \\
\text{specification} & \ ::= \ ...
\quad | \quad \text{floating-attribute} \\
\text{class-field-spec} & \ ::= \ ...
\quad | \quad \text{floating-attribute} \\
\text{class-field} & \ ::= \ ...
\quad | \quad \text{floating-attribute}
\end{align*}
\]

(Note: contrary to what the grammar above describes, 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{verbatim}
let[@foo] x = 2 in x + 1 === (let x = 2 [@@foo] in x + 1)
begin[@foo][@bar x] ... end === (begin ... end)[@foo][@@bar x]
module[@foo] M = ... === module M = ... [@@foo]
type[@foo] t = T === type t = T [@@foo]
method[@foo] m = ... === method m = ... [@@foo]
\end{verbatim}

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

\begin{verbatim}
let[@foo] x = 2 and y = 3 in x + y === (let x = 2 [@@foo] and y = 3 in x + y)
let[@foo] x = 2
and[@bar] y = 3 in x + y === (let x = 2 [@@foo] and y = 3 [@bar] in x + y)
\end{verbatim}

\subsection{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 attached to any kind of syntactic item which support attributes (such as an expression, or a type expression) in which case its scope is limited to that item. Note that it is not well-defined which scope is used for a specific warning. This is implementation dependant and can change between versions. Some warnings are even completely outside the control of "ocaml.warning" (for instance, warnings 1, 2, 14, 29 and 50).
“ocaml.warnerror” or “warnerror”, with a string literal payload. Same as “ocaml.warning”, for the -warn-error command-line option.

“ocaml.alert” or “alert”: see section 8.22.

“ocaml.deprecated” or “deprecated”: alias for the “deprecated” alert, see section 8.22.

“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.l <- expr”), the “deprecated” alert will be triggered. If the payload of the attribute is a string literal, the alert 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 it 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 19.11 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.local` or `local` take either `never`, `always`, `maybe` or nothing as payload on a function definition. If no payload is provided, the default is `always`. The attribute controls an optimization which consists in compiling a function into a static continuation. Contrary to inlining, this optimization does not duplicate the function’s body. This is possible when all references to the function are full applications, all sharing the same continuation (for instance, the returned value of several branches of a pattern matching). `never` disables the optimization, `always` asserts that the optimization applies (otherwise a warning 55 is emitted) and `maybe` lets the optimization apply when possible (this is the default behavior when the attribute is not specified). The optimization is implicitly disabled when using the bytecode compiler in debug mode (-g), and for functions marked with an `ocaml.inline always` or `ocaml.unrolled` attribute which supersede `ocaml.local`.

```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

let fires_warning_22 x =
    assert (x >= 0) [@ppwarning "TODO: remove this later"]
Warning 22: TODO: remove this later

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

let rec not_a_tail_call = function
    | [] -> []
    | x :: q -> x :: (not_a_tail_call[@tailcall]) q
Warning 51: expected tailcall

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 fragile_match_1 = function
| Int 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 9.5)
val fragile_match_1 : fragile -> unit = <fun>
```

let fragile_match_2 = function
| String "constant" -> ()
| _ -> ()

```
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 9.5)
val fragile_match_2 : fragile -> unit = <fun>
```

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

### 8.14 Extension nodes

(Introduced in OCaml 4.02, infix notations for constructs other than expressions added in 4.03, infix notation (e1 ;%ext e2) added in 4.04.)

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 8.13.

The first form of extension node is used for “algebraic” categories:
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]
begin%foo ... end === [%foo begin ... end]
x ;%foo 2 === [%foo x; 2]
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:

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

Quoted strings `{...}` are particularly interesting for payloads that embed foreign syntax fragments. Those fragments can be interpreted by a preprocessor and turned into OCaml code without requiring escaping quotes. For instance, you can use `[%sql {1...1}]` to represent arbitrary SQL statements – assuming you have a ppx-rewriter that recognizes the `%sql` extension.

Note that the word-delimited form, for example `{sql|...|sql}`, should not be used for signaling that an extension is in use. Indeed, the user cannot see from the code whether this string literal has different semantics than they expect. Moreover, giving semantics to a specific delimiter limits the freedom to change the delimiter to avoid escaping issues.

### 8.14.1 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 8.15) and return its extension constructor slot.

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

# x <> y;;
- : bool = true
```

### 8.15 Extensible variant types

(Introduced in OCaml 4.02)

```
type-representation ::=  ...
   |  = ..

specification ::=  ...
   |  type [type-params] typeconstr type-extension-spec

definition ::=  ...
   |  type [type-params] typeconstr type-extension-def

type-extension-spec ::=  += [private] [l] constr-decl {1 constr-decl}
type-extension-def ::=  += [private] [l] constr-def {1 constr-def}

constr-def ::=  constr-decl
   |  constr-name = constr
```
Extensible variant types are variant types which can be extended with new variant constructors. Extensible variant types are defined using ... New variant constructors are added using +=.

```ocaml
module Expr = struct
  type attr = ..
  type attr += Str of string
  type attr +=
    | Int of int
    | Float of float
end
```

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

```ocaml
let to_string = function
  | Expr.Str s -> s
  | Expr.Int i -> Int.to_string i
  | Expr.Float f -> string_of_float f
  | _ -> "?"
```

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

```ocaml
type exn += Exc of int
```

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

```ocaml
# let not_in_scope = Str "Foo";;
Error: Unbound constructor Str
```

```ocaml
type Expr.attr += Str = Expr.Str
```

```ocaml
# let now_works = Str "foo";;
val now_works : Expr.attr = Expr.Str "foo"
```

Extensible variant constructors can be declared private. As with regular variants, this prevents them from being constructed directly by constructor application while still allowing them to be de-structured in pattern-matching.

```ocaml
module B : sig
  type Expr.attr += private Bool of int
  val bool : bool -> Expr.attr
end = struct
  type Expr.attr += Bool of int
  let bool p = if p then Bool 1 else Bool 0
end
```
# let inspection_works = function
#     | B.Bool p -> (p = 1)
#     | _ -> true;;
val inspection_works : Expr.attr -> bool = <fun>

# let construction_is_forbidden = B.Bool 1;;
Error: Cannot create values of the private type Expr.attr

## 8.15.1 Private extensible variant types
(Introduced in OCaml 4.06)

\[
\text{type-representation ::= } \ldots
\| \text{= private ..}
\]

Extensible variant types can be declared private. This prevents new constructors from being declared directly, but allows extension constructors to be referred to in interfaces.

module Msg : sig
  type t = private ..
  module MkConstr (X : sig type t end) : sig
    type t += C of X.t
  end
end

end = struct
  type t = ..
  module MkConstr (X : sig type t end) = struct
    type t += C of X.t
  end
end

## 8.16 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.

8.17 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.

8.17.1 Extension operators

(Introduced in OCaml 4.02.2)

```
infix-symbol ::= ... |
                 # {operator-chars} # {operator-char | #}
```

Operator names starting with a # character and containing more than one # character are reserved for extensions.
8.17.2 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.

8.18 Inline records

(Introduced in OCaml 4.03)

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

The arguments of 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.

```ocaml
type t =
    | Point of {width: int; mutable x: float; mutable y: float}
    | Other
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}
    | Other -> Other
let print = function
    | Point {x; y; _} -> Printf.printf "%f/%f" x y
    | Other -> ()
let reset = function
```
let invalid = function
  | Point p -> p
Error: This form is not allowed as the type of the inlined record could escape.

8.19 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 8.13) 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 15). The three comment forms recognised by the compiler are a subset of the forms accepted by ocamldoc (see 15.2).

8.19.1 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:

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

8.19.2 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:

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

or
(** 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.

### 8.19.3 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:
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:

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 "]
8.20 Extended indexing operators

(Introduced in 4.06)

\[
\text{dot-ext} ::= \\
\quad | ( ! | $ | % | * | + | - | / | : | = | > | ? | @ | ^ | ] | \{ ) \{operator-char\}
\]

\[
\text{expr} ::= \ldots \\
\quad | \text{expr} . \text{module-path} . \text{dot-ext} ( ( \text{expr} ) | [ \text{expr} ] | \{ \text{expr} \} ) \triangleleft \text{expr}
\]

\[
\text{operator-name} ::= \ldots \\
\quad | . \text{dot-ext} ( \ldots | [] | \{ \} ) \triangleleft
\]

This extension provides syntactic sugar for getting and setting elements for user-defined indexed types. For instance, we can define python-like dictionaries with

```ocaml
module Dict = struct
 include Hashtbl
 let ( .%{} ) tabl index = find tabl index
 let ( .%{}<- ) tabl index value = add tabl index value
end
let dict =
 let dict = Dict.create 10 in
 let () =
  dict.Dict.%{"one"} <- 1;
 let open Dict in
  dict.%{"two"} <- 2 in
 dict

# dict.Dict.%{"one"};;
- : int = 1

# let open Dict in dict.%{"two"};;
- : int = 2
```

8.21 Empty variant types

(Introduced in 4.07.0)

\[
\text{type-representation} ::= \ldots \\
\quad | = |
\]

This extension allows user to define empty variants. Empty variant type can be eliminated by refutation case of pattern matching.

```ocaml
type t = |
let f (x: t) = match x with _ -> .
```
8.22 Alerts

(Introduced in 4.08)

Since OCaml 4.08, it is possible to mark components (such as value or type declarations) in signatures with “alerts” that will be reported when those components are referenced. This generalizes the notion of “deprecated” components which were previously reported as warning 3. Those alerts can be used for instance to report usage of unsafe features, or of features which are only available on some platforms, etc.

Alert categories are identified by a symbolic identifier (a lowercase identifier, following the usual lexical rules) and an optional message. The identifier is used to control which alerts are enabled, and which ones are turned into fatal errors. The message is reported to the user when the alert is triggered (i.e. when the marked component is referenced).

The `ocaml.alert` or `alert` attribute serves two purposes: (i) to mark component with an alert to be triggered when the component is referenced, and (ii) to control which alert names are enabled. In the first form, the attribute takes an identifier possibly followed by a message. Here is an example of a value declaration marked with an alert:

```ocaml
module U: sig
val fork: unit -> bool
  [@@alert unix "This function is only available under Unix."]
end
```

Here `unix` is the identifier for the alert. If this alert category is enabled, any reference to `U.fork` will produce a message at compile time, which can be turned or not into a fatal error.

And here is another example 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 file, so that any reference to that unit will trigger the alert:

`[@@alert unsafe "This module is unsafe!"]`

Controlling which alerts are enabled and whether they are turned into fatal errors is done either through the compiler’s command-line option `-alert <spec>` or locally in the code through the `alert` or `ocaml.alert` attribute taking a single string payload `<spec>`. In both cases, the syntax for `<spec>` is a concatenation of items of the form:

- `+id` enables alert `id`.
- `-id` disables alert `id`.
- `++id` turns alert `id` into a fatal error.
- `-id` turns alert `id` into non-fatal mode.
- `@id` equivalent to `++id+id` (enables `id` and turns it into a fatal-error)

As a special case, if `id` is `all`, it stands for all alerts.

Here are some examples:
Chapter 8. Language extensions

(* Disable all alerts, reenables just unix (as a soft alert) and window (as a fatal-error), for the rest of the current structure *)

[@@alert "-all--all+unix@window"]
...

let x =
(* Locally disable the window alert *)
begin[@alert "-window"]
...
end

Before OCaml 4.08, there was support for a single kind of deprecation alert. It is now known as the deprecated alert, but legacy attributes to trigger it and the legacy ways to control it as warning 3 are still supported. For instance, passing -w +3 on the command-line is equivalent to -alert +deprecated, and:

val x: int
[@@ocaml.deprecated "Please do something else"]

is equivalent to:

val x: int
[@@ocaml.alert deprecated "Please do something else"]

8.23 Generalized open statements
(Introduced in 4.08)

\[
definition ::= ...
| open module-expr
| open! module-expr

\[
specification ::= ...
| open extended-module-path
| open! extended-module-path

\[
expr ::= ...
| let open module-expr in expr
| let open! module-expr in expr
\]

This extension makes it possible to open any module expression in module structures and expressions. A similar mechanism is also available inside module types, but only for extended module paths (e.g. F(X).G(Y)).

For instance, a module can be constrained when opened with
module M = struct let x = 0 let hidden = 1 end
open (M:sig val x: int end)
let y = hidden
Error: Unbound value hidden

Another possibility is to immediately open the result of a functor application

let sort (type x) (x:x list) =
  let open Set.Make(struct type t = x let compare=compare end) in
  elements (of_list x)
val sort : 'a list -> 'a list = <fun>

Going further, this construction can introduce local components inside a structure,

module M = struct
  let x = 0
  open! struct
    let x = 0
    let y = 1
  end
  let w = x + y
end
module M : sig val x : int val w : int end

One important restriction is that types introduced by open struct... end cannot appear in the signature of the enclosing structure, unless they are defined equal to some non-local type. So:

module M = struct
  open struct type 'a t = 'a option = None | Some of 'a end
  let x : int t = Some 1
end
module M : sig val x : int option end

is OK, but:

module M = struct
  open struct type t = A end
  let x = A
end
Error: The type t/3400 introduced by this open appears in the signature
File "exten.etex", line 3, characters 6-7:
The value x has no valid type if t/3400 is hidden

is not because x cannot be given any type other than t, which only exists locally. Although the above would be OK if x too was local:
Chapter 8. Language extensions

module M: sig end = struct
  open struct
  type t = A
  end
  ...
  open struct let x = A end
  ...
end
module M : sig end

Inside signatures, extended opens are limited to extended module paths,

module type S = sig
  module F: sig end -> sig type t end
  module X: sig end
  open F(X)
  val f: t
end
module type S =
  sig
    module F : sig end -> sig type t end
  module X : sig end
    val f : F(X).t
  end

  and not

  open struct type t = int end

  In those situations, local substitutions(see 8.8.2) can be used instead.
  Beware that this extension is not available inside class definitions:

class c =
  let open Set.Make(Int) in
  ...

8.24 Binding operators

(Introduced in 4.08.0)
Users can define \textit{let} operators:

\begin{verbatim}
let ( let* ) o f = 
  match o with
  | None -> None
  | Some x -> f x
let return x = Some x
val ( let* ) : 'a option -> ('a -> 'b option) -> 'b option = <fun>
val return : 'a -> 'a option = <fun>
\end{verbatim}

and then apply them using this convenient syntax:

\begin{verbatim}
let find_and_sum tbl k1 k2 = 
  let* x1 = Hashtbl.find_opt tbl k1 in 
  let* x2 = Hashtbl.find_opt tbl k2 in 
  return (x1 + x2)
val find_and_sum : ('a, int) Hashtbl.t -> 'a -> 'a -> int option = <fun>
\end{verbatim}

which is equivalent to this expanded form:

\begin{verbatim}
let find_and_sum tbl k1 k2 = 
  ( let* ) (Hashtbl.find_opt tbl k1) 
  (fun x1 ->
    ( let* ) (Hashtbl.find_opt tbl k2) 
    (fun x2 -> return (x1 + x2)))
val find_and_sum : ('a, int) Hashtbl.t -> 'a -> 'a -> int option = <fun>
\end{verbatim}

Users can also define \textit{and} operators:

\begin{verbatim}
module ZipSeq = struct
  type 'a t = 'a Seq.t
  open Seq
\end{verbatim}
Chapter 8. Language extensions

```
let rec return x = 
  fun () -> Cons(x, return x)
let rec prod a b = 
  fun () ->
    match a (), b () with
    | Nil, _ | _, Nil -> Nil
    | Cons(x, a), Cons(y, b) -> Cons((x, y), prod a b)
let ( let+ ) f s = map s f
let ( and+ ) a b = prod a b
```

```
module ZipSeq :
  sig
    type 'a t = 'a Seq.t
    val return : 'a -> 'a Seq.t
    val prod : 'a Seq.t -> 'b Seq.t -> ('a * 'b) Seq.t
    val ( let+ ) : 'a Seq.t -> ('a -> 'b) -> 'b Seq.t
    val ( and+ ) : 'a Seq.t -> 'b Seq.t -> ('a * 'b) Seq.t
  end
```

```
open ZipSeq
let sum3 z1 z2 z3 =
  let+ x1 = z1
  and+ x2 = z2
  and+ x3 = z3 in
  x1 + x2 + x3
val sum3 : int Seq.t -> int Seq.t -> int Seq.t -> int Seq.t = <fun>
```

```
which is equivalent to this expanded form:

open ZipSeq
let sum3 z1 z2 z3 =
  ( let+ ) (( and+ ) (( and+ ) z1 z2) z3)
  (fun ((x1, x2), x3) -> x1 + x2 + x3)
val sum3 : int Seq.t -> int Seq.t -> int Seq.t -> int Seq.t = <fun>
```

8.24.1 Rationale

This extension is intended to provide a convenient syntax for working with monads and applicatives.

An applicative should provide a module implementing the following interface:

```
module type Applicative_syntax = sig
  type 'a t
  val ( let+ ) : 'a t -> ('a -> 'b) -> 'b t
  val ( and+ ) : 'a t -> 'b t -> ('a * 'b) t
end
```
where \(\text{let}+\) is bound to the \texttt{map} operation and \(\texttt{and}+\) is bound to the monoidal product operation.

A monad should provide a module implementing the following interface:

```ocaml
module type Monad_syntax = sig
  include Applicative_syntax

  val ( let* ) : 'a t -> ('a -> 'b t) -> 'b t
  val ( and* ) : 'a t -> 'b t -> ('a * 'b) t
end
```

where \(\text{let}^*\) is bound to the \texttt{bind} operation, and \(\texttt{and}^*\) is also bound to the monoidal product operation.
Part III

The OCaml tools
Chapter 9

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.

9.1 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

\[
\text{ocamlrun } a.\text{out } arg_1 \arg_2 \ldots \arg_n
\]

executes the compiled code contained in `a.out`, passing it as arguments the character strings `\arg_1` to `\arg_n`. (See chapter 11 for more details.)

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

\[
./a.\text{out } arg_1 \arg_2 \ldots \arg_n
\]

The produced file has the executable bit set, and it manages to launch the bytecode interpreter by itself.

### 9.2 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 .cmalibrary. 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.
Chapter 9. Batch compilation (ocamlc)

-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.

-args filename
Read additional newline-terminated command line arguments from filename.

-args0 filename
Read additional null character terminated command line arguments from filename.

-bin-annot
Dump detailed information about the compilation (types, bindings, tail-calls, etc) in binary format. The information for file src.ml (resp. src.mli) is put into file src.cmt (resp. src.cmti). In case of a type error, dump all the information inferred by the type-checker before the error. The *.cmt and *.cmti 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.
The environment variable OCAML_COLOR is considered if -color is not provided. Its values are auto/always/never as above.

-error-style mode
Control the way error messages and warnings are printed. The following modes are supported:

  short
    only print the error and its location;

  contextual
    like short, but also display the source code snippet corresponding to the location of the error.

The default setting is contextual.
The environment variable OCAML_ERROR_STYLE is considered if -error-style is not provided. Its values are short/contextual as above.

-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.

-config-var var
Print the value of a specific configuration variable from the -config output, then exit. If the variable does not exist, the exit code is non-zero. This option is only available since OCaml 4.08, so script authors should have a fallback for older versions.

-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 19.

Unix:

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

Security warning: never set the “setuid” or “setgid” bits on executables produced by ocamlc -custom, this would make them vulnerable to attacks.

-dep ocamldep-args

Compute dependencies, as the ocamldep command would do. The remaining arguments are interpreted as if they were given to the ocamldep command.

-dlllib -l dbname

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

-dllpath dir

Adds the directory dir to the run-time search path for shared C libraries. At link-time, shared libraries are searched in the standard search path (the one corresponding to the -I option). The -dllpath option simply stores dir in the produced executable file, where ocamlrun can find it and use it as described in section 11.3.

-for-pack module-path

Generate an object file (.cmo) that can later be included as a sub-module (with the given access 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 16), and to produce stack backtraces when the program terminates on an uncaught exception (see section 11.2).

-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 .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 +unix adds the subdirectory unix 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. When compiling a module (option -c), setting the -linkall option ensures
   that this module will always be linked if it is put in a library and this library is linked.

-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 ocaml -use-runtime runtime-name
   option. See section 19.1.6 for more information.

-match-context-rows
   Set the number of rows of context used for optimization during pattern matching compilation.
   The default value is 32. Lower values cause faster compilation, but less optimized code. This
   advanced option is meant for use in the event that a pattern-match-heavy program leads to
   significant increases in compilation time.

-no-alias-deps
   Do not record dependencies for module aliases. See section 8.9 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 (.cmi), 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.

-opaque
When the native compiler compiles an implementation, by default it produces a .cmx file containing information for cross-module optimization. It also expects .cmx files to be present for the dependencies of the currently compiled source, and uses them for optimization. Since OCaml 4.03, the compiler will emit a warning if it is unable to locate the .cmx file of one of those dependencies.

The -opaque option, available since 4.04, disables cross-module optimization information for the currently compiled unit. When compiling .mli interface, using -opaque marks the compiled .cmi interface so that subsequent compilations of modules that depend on it will not rely on the corresponding .cmx file, nor warn if it is absent. When the native compiler compiles a .ml implementation, using -opaque generates a .cmx that does not contain any cross-module optimization information.

Using this option may degrade the quality of generated code, but it reduces compilation time, both on clean and incremental builds. Indeed, with the native compiler, when the implementation of a compilation unit changes, all the units that depend on it may need to be recompiled – because the cross-module information may have changed. If the compilation unit whose implementation changed was compiled with -opaque, no such recompilation needs to occur. This option can thus be used, for example, to get faster edit-compile-test feedback loops.

-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 19, section 19.7.5. 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 (.cmi) that combines the object files given on the command line, making them appear as sub-modules of the output .cmo file. The name of the output .cmo file must be given with the -o option. For instance,

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

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

-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 section 24.1, 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.
-stop-after pass
   Stop compilation after the given compilation pass. The currently supported passes are:
   parsing, typing.

-safe-string
   Enforce the separation between types string and bytes, thereby making strings read-only.
   This is the default.

-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.

-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. 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. This is intended for
   compatibility with old source code and should not be used with new software.

-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 19.1.6
   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
   Deprecated since OCaml 4.08.0. Compile or link multithreaded programs, in combination with the VM-level threads library described in chapter 28.

-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 synonym for the 'deprecated' alert.
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.
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.
15. Private method made public implicitly.
17. Undeclared virtual method.
18. Non-principal type.
19. Type without principality.
20. Unused function argument.
22. Preprocessor warning.
23. Useless record `with` clause.
24. Bad module name: the source file name is not a valid OCaml module name.
25. Deprecated: now part of warning 8.
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.
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.

Constructor or label name used out of scope.

Ambiguous constructor or label name.

Disambiguated constructor or label name (compatibility warning).

Nonoptional label applied as optional.

Open statement shadows an already defined identifier.

Open statement shadows an already defined label or constructor.

Error in environment variable.

Illegal attribute payload.

Implicit elimination of optional arguments.

Absent cmi file when looking up module alias.

Unexpected documentation comment.

Warning on non-tail calls if @tailcall present.

(see 9.5.2)

Fragile constant pattern.

Attribute cannot appear in this context

Attribute used more than once on an expression

Inlining impossible

Unreachable case in a pattern-matching (based on type information).

(see 9.5.3)

Ambiguous or-pattern variables under guard

Missing cmx file

Assignment to non-mutable value

Unused module declaration

Unboxable type in primitive declaration
Chapter 9. Batch compilation (ocamlc)

62 Type constraint on GADT type declaration
63 Erroneous printed signature
64 -unsafe used with a preprocessor returning a syntax tree
65 Type declaration defining a new '()' constructor
66 Unused open! statement
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..42-44-45-48-50-60. It is displayed by ocamlc -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.

**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)

A set of arguments that will be inserted before or after the arguments from the command line. Arguments are specified in a comma-separated list of name=value pairs. A _ is used to specify the position of the command line arguments, i.e. a=x,,b=y means that a=x should be executed before parsing the arguments, and b=y after. Finally, an alternative separator can be specified as the first character of the string, within the set :|; ,.

**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.

### 9.3 Modules and the file system

This short section is intended to clarify the relationship between the names of the modules corresponding to compilation units and the names of the files that contain their compiled interface and compiled implementation.

The compiler always derives the module name by taking the capitalized base name of the source file (.ml or .mli file). That is, it strips the leading directory name, if any, as well as the .ml or .mli suffix; then, it set the first letter to uppercase, in order to comply with the requirement that module names must be capitalized. For instance, compiling the file mylib/misc.ml provides an implementation for the module named Misc. Other compilation units may refer to components defined in mylib/misc.ml under the names Misc.name; they can also do open Misc, then use unqualified names name.

The .cmi and .cmo files produced by the compiler have the same base name as the source file. Hence, the compiled files always have their base name equal (modulo capitalization of the first letter) to the name of the module they describe (for .cmi files) or implement (for .cmo files).

When the compiler encounters a reference to a free module identifier Mod, it looks in the search path for a file named Mod.cmi or mod.cmi and loads the compiled interface contained in that file. As
a consequence, renaming .cmi files is not advised: the name of a .cmi file must always correspond to the name of the compilation unit it implements. It is admissible to move them to another directory, if their base name is preserved, and the correct -I options are given to the compiler. The compiler will flag an error if it loads a .cmi file that has been renamed.

Compiled bytecode files (.cmo files), on the other hand, can be freely renamed once created. That’s because the linker never attempts to find by itself the .cmo file that implements a module with a given name: it relies instead on the user providing the list of .cmo files by hand.

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. The filename is either a compiled interface file (.cmi file), or a compiled bytecode file (.cmo file). If filename has the format mod.cmi, this means you are trying to compile a file that references identifiers from module mod, but you have not yet compiled an interface for module mod. Fix: compile mod.mli or mod.ml first, to create the compiled interface mod.cmi.

If filename has the format mod.cmo, this means you are trying to link a bytecode object file that does not exist yet. Fix: compile mod.ml first.

If your program spans several directories, this error can also appear because you haven’t specified the directories to look into. Fix: add the correct -I options to the command line.

Corrupted compiled interface filename

The compiler produces this error when it tries to read a compiled interface file (.cmi file) that has the wrong structure. This means something went wrong when this .cmi file was written: the disk was full, the compiler was interrupted in the middle of the file creation, and so on. This error can also appear if a .cmi file is modified after its creation by the compiler. Fix: remove the corrupted .cmi file, and rebuild it.

This expression has type \( t_1 \), but is used with type \( t_2 \)

This is by far the most common type error in programs. Type \( t_1 \) is the type inferred for the expression (the part of the program that is displayed in the error message), by looking at the expression itself. Type \( t_2 \) is the type expected by the context of the expression; it is deduced by looking at how the value of this expression is used in the rest of the program. If the two types \( t_1 \) and \( t_2 \) are not compatible, then the error above is produced.

In some cases, it is hard to understand why the two types \( t_1 \) and \( t_2 \) are incompatible. For instance, the compiler can report that “expression of type \( \text{foo} \) cannot be used with type \( \text{foo} \)”, and it really seems that the two types \( \text{foo} \) are compatible. This is not always true. Two type constructors can have the same name, but actually represent different types. This can happen if a type constructor is redefined. Example:

\[
\begin{align*}
\text{type foo} & = A \mid B \\
\text{let f = function A -> 0 | B -> 1}
\end{align*}
\]
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 ($a, b, \ldots$) in a type $t$ can be in either of two states: generalized (which means that the type $t$ is valid for all possible instantiations of the variables) and not generalized (which means that the type $t$ is valid only for one instantiation of the variables). In a let binding let name = expr, the type-checker normally generalizes as many type variables as possible in the type of expr. However, this leads to unsoundness (a well-typed program can crash) in conjunction with polymorphic mutable data structures. To avoid this, generalization is performed at let bindings only if the bound expression expr belongs to the class of “syntactic values”, which includes constants, identifiers, functions, tuples of syntactic values, etc. In all other cases (for instance, expr is a function application), a polymorphic mutable could have been created and generalization is therefore turned off 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 name whose type contains non-generalized type variables. There are two ways to fix this error:

- Add a type constraint or a .mli file to give a monomorphic type (without type variables) to name. For instance, instead of writing

  let sort_int_list = List.sort Stdlib.compare
  (* inferred type 'a list -> 'a list, with 'a not generalized *)

  write

  let sort_int_list = (List.sort Stdlib.compare : 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 general-
  ized *)

  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 \textit{mod} on the command line (typically, the file named \texttt{mod.cmo}, or a library containing that file). Fix: add the missing \texttt{.ml} or \texttt{.cmo} file to the command line. Or, you have provided an implementation for the module named \textit{mod}, but it comes too late on the command line: the implementation of \textit{mod} must come before all bytecode object files that reference \textit{mod}. Fix: change the order of \texttt{.ml} and \texttt{.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 \texttt{Mod1.f} calls function \texttt{Mod2.g}, and function \texttt{Mod2.g} calls function \texttt{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 \texttt{f} and \texttt{g} in the same module.
- Parameterize one function by the other. That is, instead of having

\begin{verbatim}
mod1.ml: let f x = ... Mod2.g ...
mod2.ml: let g y = ... Mod1.f ...
\end{verbatim}

define

\begin{verbatim}
mod1.ml: let f g x = ... g ...
mod2.ml: let rec g y = ... Mod1.f g ...
\end{verbatim}

and link \texttt{mod1.cmo} before \texttt{mod2.cmo}.

- Use a reference to hold one of the two functions, as in :

\begin{verbatim}
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
\end{verbatim}

The external function \texttt{f} is not available

This error appears when trying to link code that calls external functions written in C. As explained in chapter 19, such code must be linked with C libraries that implement the required \texttt{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 \texttt{-custom} option.

9.5 Warning reference

This section describes and explains in detail some warnings:

9.5.1 Warning 9: missing fields in a record pattern

When pattern matching on records, it can be useful to match only few fields of a record. Eliding fields can be done either implicitly or explicitly by ending the record pattern with ; _ -. However, implicit field elision is at odd with pattern matching exhaustiveness checks. Enabling warning 9 prioritizes exhaustiveness checks over the convenience of implicit field elision and will warn on
implicit field elision in record patterns. In particular, this warning can help to spot exhaustive
record pattern that may need to be updated after the addition of new fields to a record type.

```ocaml
type 'a point = {x='a ;y='a}
let dx { x } = x (* implicit field elision: trigger warning 9 *)
let dy { y; _ } = y (* explicit field elision: do not trigger warning 9 *)
```

### 9.5.2 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
`ocaml.warn_on_literal_pattern` (see the manual section on builtin attributes in 8.13.1):

```ocaml
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
```

> " | 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: \texttt{Invalid_argument}, \texttt{Failure}, \texttt{Sys_error} will all raise this warning if you match for a specific string argument.

Additionally, built-in exceptions with a structured argument that includes a string also have the attribute set: \texttt{Assert_failure} and \texttt{Match_failure} will raise the warning for a pattern that uses a literal string to match the first element of their tuple argument.

If your code raises this warning, you should \textit{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.

\begin{verbatim}
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,

\begin{verbatim}
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 \texttt{exception} patterns documented in Section 7.6, one can write:

\begin{verbatim}
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 raise 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.

\end{verbatim}
\end{verbatim}

9.5.3 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 \( p \mid q \) 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: \( p \texttt{ when } g \rightarrow e \), 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 10

The toplevel system or REPL (ocaml)

This chapter describes the toplevel system for OCaml, that permits interactive use of the OCaml system through a read-eval-print loop (REPL). 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:

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

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

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 7.7). 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 10.2.

Unix:

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

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

options are described below. objects are filenames ending in .cmo or .cma; they are loaded into the interpreter immediately after options are set. scriptfile is any file name not ending in .cmo or .cma.
If no scriptfile is given on the command line, the toplevel system enters interactive mode: phrases are read on standard input, results are printed on standard output, errors on standard error. End-of-file on standard input terminates ocaml (see also the #quit directive in section 10.2).

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 10.2. The evaluation outcode for each phrase are not displayed. If the current directory does not contain an .ocamlinit file, but the user’s home directory (environment variable HOME) does, the latter is read and executed as described below.

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

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing ctrl-C (or, more precisely, by sending the INTR signal to the ocaml process). The toplevel then immediately returns to the # prompt.

If scriptfile is given on the command-line to ocaml, the toplevel system enters script mode: the contents of the file are read as a sequence of OCaml phrases and executed, as per the #use directive (section 10.2). 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:

```bash
#!/usr/local/bin/ocamlrun /usr/local/bin/ocaml
```

### 10.1 Options

The following command-line options are recognized by the ocaml command.

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

- `-args filename`
  Read additional newline-terminated command line arguments from filename. It is not possible to pass a scriptfile via file to the toplevel.

- `-args0 filename`
  Read additional null character terminated command line arguments from filename. It is not possible to pass a scriptfile via file to the toplevel.
Chapter 10. The toplevel system or REPL (ocaml)

- 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. See also option -nostdlib.

  If the given directory starts with +, it is taken relative to the standard library directory. For instance, -I +unix adds the subdirectory unix 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 10.2).

- 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 section 24.1, 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.

-safestring
    Enforce the separation between types string and bytes, thereby making strings read-only.
This is the default.

-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
    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.

-unsafe-string
    Identify the types string and bytes, thereby making strings writable. This is intended for
compatibility with old source code and should not be used with new software.

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

-verbose
    Print all external commands before they are executed, Useful to debug C library problems.

-version
    Print version string and exit.
- **vnum**
  Print short version number and exit.

- **no-version**
  Do not print the version banner at startup.

- **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 synonym for the 'deprecated' alert
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.
25 Deprecated: now part of warning 8.
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 9.5.2)
   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 9.5.3)
   Ambiguous or-pattern variables under guard
58 Missing cmx file
59 Assignment to non-mutable value
60 Unused module declaration
61 Unboxable type in primitive declaration
62 Type constraint on GADT type declaration
63 Erroneous printed signature
64 -unsafe used with a preprocessor returning a syntax tree
65 Type declaration defining a new '(' constructor
66 Unused open! statement
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 \texttt{-w +a-4-6-7-9-27-29-32..42-44-45-48-50-60}. It is displayed by \texttt{-help}. Note that warnings 5 and 10 are not always triggered, depending on the internals of the type checker.

\texttt{-warn-error warning-list}

Mark as fatal the warnings specified in the argument \texttt{warning-list}. The compiler will stop with an error when one of these warnings is emitted. The \texttt{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 \texttt{@} 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 -a+31} (only warning 31 is fatal).

\texttt{-warn-help}

Show the description of all available warning numbers.

\texttt{- file}

Use \texttt{file} as a script file name, even when it starts with a hyphen (-).

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

Display a short usage summary and exit.

\texttt{Unix:}

The following environment variables are also consulted:
10.2 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.

For directives that take file names as arguments, if the given file name specifies no directory, the file is searched in the following directories:

1. In script mode, the directory containing the script currently executing; in interactive mode, the current working directory.
2. Directories added with the #directory directive.
3. Directories given on the command line with -I options.
4. The standard library directory.

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 \texttt{false}, or switch back to default behaviour (commuting style) if argument is \texttt{true}.

#ppx "file-name";;
After parsing, pipe the abstract syntax tree through the preprocessor command.

#principal bool;;
If the argument is \texttt{true}, check information paths during type-checking, to make sure that all types are derived in a principal way. If the argument is \texttt{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.
10.3 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 7.3).

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.

10.4 Common errors

This section describes and explains the most frequently encountered error messages.

**Cannot find file filename**

The named file could not be found in the current directory, nor in the directories of the search path.

If filename has the format mod.cmi, this means you have referenced the compilation unit mod, but its compiled interface could not be found. Fix: compile mod.mli or mod.ml first, to create the compiled interface mod.cmi.

If filename has the format mod.cmo, this means you are trying to load with #load a bytecode object file that does not exist yet. Fix: compile mod.ml first.

If your program spans several directories, this error can also appear because you haven’t specified the directories to look into. Fix: use the #directory directive to add the correct directories to the search path.

**This expression has type t₁, but is used with type t₂**

See section 9.4.

**Reference to undefined global mod**

You have neglected to load in memory an implementation for a module with #load. See section 10.3 above.

10.5 Building custom toplevel systems: ocamlmktop

The ocamlmktop command builds OCaml toplevels that contain user code preloaded at start-up.

The ocamlmktop command takes as argument a set of .cmo and .cma files, and links them with the object files that implement the OCaml toplevel. The typical use is:

```
ocamlmktop -o mytoplevel foo.cmo bar.cmo gee.cmo
```
This creates the bytecode file mytoplevel, containing the OCaml toplevel system, plus the code from the three .cmo files. This toplevel is directly executable and is started by:

    ./mytoplevel

This enters a regular toplevel loop, except that the code from foo.cmo, bar.cmo and gee.cmo is already loaded in memory, just as if you had typed:

    #load "foo.cmo";;
    #load "bar.cmo";;
    #load "gee.cmo";;

on entrance to the toplevel. The modules Foo, Bar and Gee are not opened, though; you still have to do

    open Foo;;

yourself, if this is what you wish.

10.5.1 Options

The following command-line options are recognized by ocamlmktop.

-cclib libname
    Pass the -l libname option to the C linker when linking in “custom runtime” mode. See the corresponding option for ocamlc, in chapter 9.

-ccopt option
    Pass the given option to the C compiler and linker, when linking in “custom runtime” mode. See the corresponding option for ocamlc, in chapter 9.

-custom
    Link in “custom runtime” mode. See the corresponding option for ocamlc, in chapter 9.

-I directory
    Add the given directory to the list of directories searched for compiled object code files (.cmo and .cma).

-o exec-file
    Specify the name of the toplevel file produced by the linker. The default is a.out.

10.6 The native toplevel: ocamlnat (experimental)

This section describes a tool that is not yet officially supported but may be found useful.

OCaml code executing in the traditional toplevel system uses the bytecode interpreter. When increased performance is required, or for testing programs that will only execute correctly when compiled to native code, the native toplevel may be used instead.
For the majority of installations the native toplevel will not have been installed along with the rest of the OCaml toolchain. In such circumstances it will be necessary to build the OCaml distribution from source. From the built source tree of the distribution you may use `make natrun-top` to build and execute a native toplevel. (Alternatively `make ocamlnat` can be used, which just performs the build step.)

If the `make install` command is run after having built the native toplevel then the `ocamlnat` program (either from the source or the installation directory) may be invoked directly rather than using `make natruntop`. 
Chapter 11

The runtime system (ocamlrun)

The ocamlrun command executes bytecode files produced by the linking phase of the ocamlc command.

11.1 Overview

The ocamlrun command comprises three main parts: the bytecode interpreter, that actually executes bytecode files; the memory allocator and garbage collector; and a set of C functions that implement primitive operations such as input/output.

The usage for ocamlrun is:

```
ocamlrun options bytecode-executable arg1 ... argn
```

The first non-option argument is taken to be the name of the file containing the executable bytecode. (That file is searched in the executable path as well as in the current directory.) The remaining arguments are passed to the 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 argn.

As mentioned in chapter 9, the bytecode executable files produced by the ocamlc command are self-executable, and manage to launch the ocamlrun command on themselves automatically. That is, assuming a.out is a bytecode executable file,

```
a.out arg1 ... argn
```

works exactly as

```
ocamlrun a.out arg1 ... argn
```

Notice that it is not possible to pass options to ocamlrun 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 ....

237
11.2 Options

The following command-line options are recognized by ocamlrun.

- **b**  When the program aborts due to an uncaught exception, print a detailed “back trace” of the execution, showing where the exception was raised and which function calls were outstanding at this point. The back trace is printed only if the bytecode executable contains debugging information, i.e. was compiled and linked with the -g option to ocamlc set. This is equivalent to setting the b flag in the OCAMLRUNPARAM environment variable (see below).

- **I** *dir*  Search the directory *dir* for dynamically-loaded libraries, in addition to the standard search path (see section 11.3).

- **m**  Print the magic number of the bytecode executable given as argument and exit.

- **M**  Print the magic number expected by this version of the runtime and exit.

- **p**  Print the names of the primitives known to this version of ocamlrun and exit.

- **v**  Direct the memory manager to print some progress messages on standard error. This is equivalent to setting v=63 in the OCAMLRUNPARAM environment variable (see below).

- **version**  Print version string and exit.

- **vnum**  Print short version number and exit.

The following environment variables are also consulted:

**CAML_LD_LIBRARY_PATH**  Additional directories to search for dynamically-loaded libraries (see section 11.3).

**OCAMLLIB**  The directory containing the OCaml standard library. (If OCAMLLIB is not set, CAMLLIB will be used instead.) Used to locate the ld.conf configuration file for dynamic loading (see section 11.3). If not set, default to the library directory specified when compiling OCaml.

**OCAMLRUNPARAM**  Set the runtime system options and garbage collection parameters. (If OCAMLRUNPARAM is not set, CAMLRUNPARAM will be used instead.) This variable must be a sequence of parameter specifications 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 ??.

  **b**  (backtrace) Trigger the printing of a stack backtrace when an uncaught exception aborts the program. This option takes no argument.
Chapter 11. The runtime system (ocamlrun)

p (parser trace) Turn on debugging support for ocamlyacc-generated parsers. When this option is on, the pushdown automaton that executes the parsers prints a trace of its actions. This option takes no argument.

R (randomize) Turn on randomization of all hash tables by default (see section ??). 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.

1024 (= 0x400)
   Output GC statistics at program exit.

c (cleanup_on_exit) Shut the runtime down gracefully on exit (see caml_shutdown in section 19.7.5). The option also enables pooling (as in caml_startup_pooled). This mode can be used to detect leaks with a third-party memory debugger.
M (custom_major_ratio) Target ratio of floating garbage to major heap size for out-of-heap memory held by custom values (e.g. bigarrays) located in the major heap. The GC speed is adjusted to try to use this much memory for dead values that are not yet collected. Expressed as a percentage of major heap size. Default: 44. Note: this only applies to values allocated with caml_alloc_custom_mem.

m (customMinorRatio) Bound on floating garbage for out-of-heap memory held by custom values in the minor heap. A minor GC is triggered when this much memory is held by custom values located in the minor heap. Expressed as a percentage of minor heap size. Default: 100. Note: this only applies to values allocated with caml_alloc_custom_mem.

n (customMinorMaxSize) Maximum amount of out-of-heap memory for each custom value allocated in the minor heap. When a custom value is allocated on the minor heap and holds more than this many bytes, only this value is counted against custom_minor_ratio and the rest is directly counted against custom_major_ratio. Default: 8192 bytes. Note: this only applies to values allocated with caml_alloc_custom_mem.

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.

### 11.3 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 19.1.4), 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.

### 11.4 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 16), or run it with `ocamlrun -b` or with the `OCAMLRUNPARAM` environment variable set to `b=1`.
Out of memory

The program being executed requires more memory than available. Either the program builds excessively large data structures; or the program contains too many nested function calls, and the stack overflows. In some cases, your program is perfectly correct, it just requires more memory than your machine provides. In other cases, the “out of memory” message reveals an error in your program: non-terminating recursive function, allocation of an excessively large array, 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 12

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 links 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`.

12.1 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 9).
• 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 (lib.cmx and lib.a/.lib) a set of object files (.cmx and .o/.obj files). Libraries are build with ocamlopt -a (see the description of the -a option below). The object files contained in the library are linked as regular .cmx files (see above), in the order specified when the library was built. The only difference is that if an object file contained in a library is not referenced anywhere in the program, then it is not linked in.

• Arguments ending in .c are passed to the C compiler, which generates a .o/.obj object file. This object file is linked with the program.

• Arguments ending in .o, .a or .so (.obj, .lib and .dll under Windows) are assumed to be C object files and libraries. They are linked with the program.

The output of the linking phase is a regular Unix or Windows executable file. It does not need ocamlrun to run.

12.2 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 .cmxalibrary. 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.

-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.
-args filename
Read additional newline-terminated command line arguments from filename.

-args0 filename
Read additional null character terminated command line arguments from filename.

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

-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.

-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.

The environment variable OCAML_COLOR is considered if -color is not provided. Its values are auto/always/never as above.

-error-style mode
Control the way error messages and warnings are printed. The following modes are supported:
short
only print the error and its location;

contextual
like short, but also display the source code snippet corresponding to the location of the error.

The default setting is contextual.

The environment variable OCAML_ERROR_STYLE is considered if -error-style is not provided. Its values are short/contextual as above.

-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.

-config-var var
Print the value of a specific configuration variable from the -config output, then exit. If the variable does not exist, the exit code is non-zero. This option is only available since OCaml 4.08, so script authors should have a fallback for older versions.

-depend ocamldep-args
Compute dependencies, as the ocamldep command would do. The remaining arguments are interpreted as if they were given to the ocamldep command.

-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. 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 produce stack backtraces when the program terminates on an uncaught exception (see section 11.2).

-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 +unix` adds the subdirectory `unix` 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 (option `-a`), setting the `-linkall` option forces all subsequent links of programs involving that library to link all the modules contained in the library. When compiling a module (option `-c`), setting the `-linkall` option ensures that this module will always be linked if it is put in a library and this library is linked.

```
-linscan
```

Use linear scan register allocation. Compiling with this allocator is faster than with the usual graph coloring allocator, sometimes quite drastically so for long functions and modules. On the other hand, the generated code can be a bit slower.

```
-match-context-rows
```

Set the number of rows of context used for optimization during pattern matching compilation. The default value is 32. Lower values cause faster compilation, but less optimized code. This advanced option is meant for use in the event that a pattern-match-heavy program leads to significant increases in compilation time.

```
-no-alias-deps
```

Do not record dependencies for module aliases. See section 8.9 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 .cmxa libraries, ignore -cclib and -ccopt options potentially contained in the libraries (if these options were given when building the libraries). This can be useful if a library contains incorrect specifications of C libraries or C options; in this case, during linking, set -noautolink and pass the correct C libraries and options on the command line.

-nodynlink
Allow the compiler to use some optimizations that are valid only for code that is never dynlinked.

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

-nostdlib
Do not automatically add the standard library directory to the list of directories searched for compiled interface files (.cmi), compiled object code files (.cmx), 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 camlprogs.exe under Windows. If the -a option is given, specify the name of the library produced. If the -pack option is given, specify the name of the packed object file produced. If the -output-obj option is given, specify the name of the output file produced. If the -shared option is given, specify the name of plugin file produced.

-opaque
When the native compiler compiles an implementation, by default it produces a .cmx file containing information for cross-module optimization. It also expects .cmx files to be present for the dependencies of the currently compiled source, and uses them for optimization. Since OCaml 4.03, the compiler will emit a warning if it is unable to locate the .cmx file of one of those dependencies.

The -opaque option, available since 4.04, disables cross-module optimization information for the currently compiled unit. When compiling .ml interface, using -opaque marks the compiled .cmi interface so that subsequent compilations of modules that depend on it will not rely on the corresponding .cmx file, nor warn if it is absent. When the native compiler compiles a .ml implementation, using -opaque generates a .cmx that does not contain any cross-module optimization information.
Using this option may degrade the quality of generated code, but it reduces compilation
time, both on clean and incremental builds. Indeed, with the native compiler, when the
implementation of a compilation unit changes, all the units that depend on it may need to
be recompiled – because the cross-module information may have changed. If the compilation
unit whose implementation changed was compiled with -opaque, no such recomilation needs
to occur. This option can thus be used, for example, to get faster edit-compile-test feedback
loops.

-`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 an executable file. This is useful to wrap
OCaml code as a C library, callable from any C program. See chapter 19, section 19.7.5. 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 17 for more
information on profiling.) The -p option must be given both at compile-time and at link-time.
Linking object files not compiled with -p is possible, but results in less precise profiling.

Unix:

See the Unix manual page for `gprof(1)` for more information about the 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:

```bash
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 section 24.1, 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.

-stop-after pass

Stop compilation after the given compilation pass. The currently supported passes are: parsing, typing.
-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 is the default.

-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.

-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. This is intended for compatibility with old source code and should not be used with new software.

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

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

-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 synonym for the ‘deprecated’ alert.
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.
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.
15. Private method made public implicitly.
17. Undeclared virtual method.
18. Non-principal type.
19. Type without principality.
20. Unused function argument.
22. Preprocessor warning.
23. Useless record with clause.
24. Bad module name: the source file name is not a valid OCaml module name.
25. Deprecated: now part of warning 8.
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.
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.

Constructor or label name used out of scope.

Ambiguous constructor or label name.

Disambiguated constructor or label name (compatibility warning).

Nonoptional label applied as optional.

Open statement shadows an already defined identifier.

Open statement shadows an already defined label or constructor.

Error in environment variable.

Illegal attribute payload.

Implicit elimination of optional arguments.

Absent cmi file when looking up module alias.

Unexpected documentation comment.

Warning on non-tail calls if @tailcall present.

(see 9.5.2)

Fragile constant pattern.

Attribute cannot appear in this context

Attribute used more than once on an expression

Inlining impossible

Unreachable case in a pattern-matching (based on type information).

(see 9.5.3)

Ambiguous or-pattern variables under guard

Missing cmx file

Assignment to non-mutable value

Unused module declaration

Unboxable type in primitive declaration
Type constraint on GADT type declaration

Erroneous printed signature

-unsafe used with a preprocessor returning a syntax tree

Type declaration defining a new '()' constructor

Unused open! statement

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-4-6-7-9-27-29-32-42-44-45-48-50-60. 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.

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)
   A set of arguments that will be inserted before or after the arguments from the command line. Arguments are specified in a comma-separated list of name=value pairs. A _ is used to specify the position of the command line arguments, i.e. a=x,_,b=y means that a=x should be executed before parsing the arguments, and b=y after. Finally, an alternative separator can be specified as the first character of the string, within the set :|; ,.

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.
12.3 Common errors

The error messages are almost identical to those of ocamlc. See section 9.4.

12.4 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 11.2), 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.

12.5 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 not always turned into a Stack_overflow exception like the bytecode compiler does. The runtime system makes a best effort to trap stack overflows and raise the Stack_overflow exception, but sometimes it fails and a “segmentation fault” or another system fault occurs instead.

- On ARM and PowerPC processors (32 and 64 bits), fused multiply-add (FMA) instructions can be generated for a floating-point multiplication followed by a floating-point addition or subtraction, as in x *. y +. z. The FMA instruction avoids rounding the intermediate result x *. y, which is generally beneficial, but produces floating-point results that differ slightly from those produced by the bytecode interpreter.

- 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.

- The native-code compiler performs a number of optimizations that the bytecode compiler does not perform, especially when the Flambda optimizer is active. In particular, the native-code
compiler identifies and eliminates “dead code”, i.e. computations that do not contribute to the results of the program. For example,

```ocaml
let _ = ignore M.f
```

contains a reference to compilation unit M when compiled to bytecode. This reference forces M to be linked and its initialization code to be executed. The native-code compiler eliminates the reference to M, hence the compilation unit M may not be linked and executed. A workaround is to compile M with the `-linkall` flag so that it will always be linked and executed, even if not referenced. See also the `Sys.opaque_identity` function from the `Sys` standard library module.
Chapter 13

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).

13.1 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 ??.)

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.)
13.1.1 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 improves performance when using the native compiler, but decreases it when using the bytecode compiler.

-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.

13.2 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 13.2.7.
13.2.1 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.

13.2.2 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`.

13.2.3 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.

13.2.4 Regular expressions

The regular expressions are in the style of `lex`, with a more OCaml-like syntax.

```
regexp ::= ...

' regular-char | escape-sequence '

A character constant, with the same syntax as OCaml character constants. Match the denoted character.

_  (underscore) Match any character.

eof Match the end of the lexer input.

Note: On some systems, with interactive input, an end-of-file may be followed by more characters. However, `ocamllex` will not correctly handle regular expressions that contain `eof` followed by something else.
" {string-character} "
A string constant, with the same syntax as OCaml string constants. Match the corresponding sequence of characters.

[ character-set ]
Match any single character belonging to the given character set. Valid character sets are:
- single character constants 'c';
- ranges of characters 'c1'-'c2' (all characters between c1 and c2, inclusive);
- and the union of two or more character sets, denoted by concatenation.

\[^ character-set \]
Match any single character not belonging to the given character set.

regexp1 \# regexp2
(difference of character sets) Regular expressions regexp1 and regexp2 must be character sets defined with \[ ... \] (or a 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.

13.2.5 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.

\texttt{entrypoint \[exp\_1 \ldots \ exp\_n\] lexbuf}
   (Where \texttt{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.

\subsection*{13.2.6 Variables in regular expressions}

The \texttt{as} construct is similar to “\texttt{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:

\begin{itemize}
   \item A variable is a \texttt{char} variable when all its occurrences bind \texttt{char} occurrences in the previous sense.
   \item A variable is an \texttt{option} variable when the overall expression can be matched without binding this variable.
\end{itemize}

For instance, in (\texttt{'a' as x} | ( \texttt{'a' (_ as x)} ) the variable \texttt{x} is of type \texttt{char}, whereas in (\texttt{"ab" as x} | ( \texttt{'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 (\texttt{('a'|"ab") as x} (\texttt{"ba"|'a') as y}) may result in binding either \texttt{x} to \texttt{"ab"} and \texttt{y} to \texttt{"a"}, or \texttt{x} to \texttt{"a"} and \texttt{y} to \texttt{"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.
13.2.7 Refill handlers

By default, when ocamllex reaches the end of its lexing buffer, it will silently call the refill buffer 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

\[(\text{Lexing.lexbuf} \rightarrow 'a) \rightarrow \text{Lexing.lexbuf} \rightarrow 'a\]

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:

```
{ type token = EOL | INT of int | PLUS

module Make (M : sig
  type 'a t
  val return : 'a -> 'a t
  val bind : 'a t -> ('a -> 'b t) -> 'b t
  val fail : string -> 'a t

  (* Set up lexbuf *)
  val on_refill : Lexing.lexbuf -> unit t
end) = struct

let refill_handler k lexbuf =
  M.bind (M.on_refill lexbuf) (fun () -> k lexbuf)
}

refill {refill_handler}

rule token = parse
  | ''/'' ''\t''
    { token lexbuf }
  | ''\n''
    { M.return EOL }
  | ['0'-'9']+ as i
    { M.return (INT (int_of_string i)) }
  | ''+''
    { M.return PLUS }
```
| _
|   { M.fail "unexpected character" }
| {
|     end
| }

13.2.8 Reserved identifiers
All identifiers starting with __ocaml_lex are reserved for use by ocamllex; do not use any such identifier in your programs.

13.3 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.

13.4 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.
13.4.1 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.

13.4.2 Declarations

Declarations are given one per line. They all start with a % sign.

\%token constr ... constr

Declare the given symbols constr ... constr as tokens (terminal symbols). These symbols are added as constant constructors for the token concrete type.

\%token < typexpr > constr ... constr

Declare the given symbols constr ... constr as tokens with an attached attribute of the given type. These symbols are added as constructors with arguments of the given type for the token concrete type. The typexpr part is an arbitrary 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

Declare the given symbols as entry points for the grammar. For each entry point, a parsing function with the same name is defined in the output module. Non-terminals that are not declared as entry points have no such parsing function. Start symbols must be given a type with the %type directive below.

\%type < typexpr > symbol ... symbol

Specify the type of the semantic attributes for the given symbols. This is mandatory for start symbols only. Other nonterminal symbols need not be given types by hand: these types will be inferred when running the output files through the OCaml compiler (unless the -s option is in effect). The typexpr part is an arbitrary 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.

### 13.4.3 Rules

The syntax for rules is as usual:

```ocaml
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).
13.4.4 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 \texttt{parse\_error} with the string "syntax error" as argument. The default \texttt{parse\_error} function does nothing and returns, thus initiating error recovery (see below). The user can define a customized \texttt{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 \texttt{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 \texttt{Parsing.Parse\_error} exception.

Refer to documentation on \texttt{yacc} for more details and guidance in how to use error recovery.

13.5 Options

The \texttt{ocamlyacc} command recognizes the following options:

\begin{itemize}
  \item \texttt{-b\ prefix}
    \begin{itemize}
      \item Name the output files \texttt{prefix.ml}, \texttt{prefix.mli}, \texttt{prefix.output}, instead of the default naming convention.
    \end{itemize}
  \item \texttt{-q}
    \begin{itemize}
      \item This option has no effect.
    \end{itemize}
  \item \texttt{-v}
    \begin{itemize}
      \item Generate a description of the parsing tables and a report on conflicts resulting from ambiguities in the grammar. The description is put in file \texttt{grammar.output}.
    \end{itemize}
  \item \texttt{-version}
    \begin{itemize}
      \item Print version string and exit.
    \end{itemize}
  \item \texttt{-vnum}
    \begin{itemize}
      \item Print short version number and exit.
    \end{itemize}
  \item \texttt{-}
    \begin{itemize}
      \item Read the grammar specification from standard input. The default output file names are \texttt{stdin.ml} and \texttt{stdin.mli}.
    \end{itemize}
  \item \texttt{- file}
    \begin{itemize}
      \item Process \texttt{file} as the grammar specification, even if its name starts with a dash (-) character. This option must be the last on the command line.
    \end{itemize}
\end{itemize}

At run-time, the \texttt{ocamlyacc}-generated parser can be debugged by setting the \texttt{p} option in the \texttt{OCAMLRUNPARAM} environment variable (see section 11.2). 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 \texttt{grammar.output} generated by \texttt{ocamlyacc -v}.
13.6 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
  [' ' '	'] { token lexbuf } (* skip blanks *)
| ['\n'] { EOL }
| ['0'-'9']+ as lxm { INT(int_of_string lxm) }
| '+' { PLUS }
| '-' { MINUS }
| '*' { TIMES }
| '/' { DIV }
| '(' { LPAREN }
| ')' { RPAREN }
| eof { raise Eof }
```
Here is the main program, that combines the parser with the lexer:

```ocaml
(* File calc.ml *)
let _ =
  try
    let lexbuf = Lexing.from_channel stdin in
    while true do
      let result = Parser.main Lexer.token lexbuf in
      print_int result; print_newline(); flush stdout
    done
  with Lexer.Eof ->
    exit 0
```

To compile everything, execute:

```
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
```

### 13.7 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.

```ocaml
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:

```ocaml
{ let keyword_table = Hashtbl.create 53
let _ =
  List.iter (fun (kwd, tok) -> Hashtbl.add keyword_table kwd tok)
```
Chapter 13. 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.

13.8 Module Depend : Module dependencies.

Warning: this module is unstable and part of compiler-libs[24].

module String :
  Misc.Stdlib.String

  type map_tree =
    | Node of String.Set.t * bound_map
  type bound_map = map_tree String.Map.t
val make_leaf : string -> map_tree
val make_node : bound_map -> map_tree
val weaken_map : String.Set.t -> map_tree -> map_tree
val free_structure_names : String.Set.t ref
val pp_deps : string list 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 14

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 15

The documentation generator (ocamldoc)

This chapter describes OCamldoc, a tool that generates documentation from special comments embedded in source files. The comments used by OCamldoc are of the form (**...**) and follow the format described in section 15.2.

OCamldoc can produce documentation in various formats: HTML, \LaTeX, TeXinfo, Unix man pages, and dot dependency graphs. Moreover, users can add their own custom generators, as explained in section 15.3.

In this chapter, we use the word element to refer to any of the following parts of an OCaml source file: a type declaration, a value, a module, an exception, a module type, a type constructor, a record field, a class, a class type, a class method, a class value or a class inheritance clause.

15.1 Usage

15.1.1 Invocation

OCamldoc is invoked via the command ocamldoc, as follows:

```
ocamldoc options sourcefiles
```

Options for choosing the output format

The following options determine the format for the generated documentation.

- `html`
  Generate documentation in HTML default format. The generated HTML pages are stored in the current directory, or in the directory specified with the `-d` option. You can customize the style of the generated pages by editing the generated `style.css` file, or by providing your own style sheet using option `-css-style`. The file `style.css` is not generated if it already exists 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 http://www.research.att.com/sw/tools/graphviz/. The textual representation of the graph is written to the file `ocamldoc.out`, or to the file specified with the `-o` option. Use `dot ocamldoc.out` to display it.

`-g file.cm[o,a,xs]` Dynamically load the given file, which defines a custom documentation generator. See section 15.4.1. 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: `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.

-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 15.1.2 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 15.2.5).

-no-stop
Keep elements placed after/between the (**/**) special comment(s) (see section 15.2).

-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 ocamlproto 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.

-csstyle 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 \textit{n} to the given \LaTeX{} sectioning command \textit{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 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 \texttt{-man} option:

\texttt{-man-mini}

Generate man pages only for modules, module types, classes and class types, instead of pages for all elements.

\texttt{-man-suffix\ suffix}

Set the suffix used for generated man filenames. Default is '3o', as in \texttt{List.3o}.

\texttt{-man-section\ section}

Set the section number used for generated man filenames. Default is '3'.

15.1.2 Merging of module information

Information on a module can be extracted either from the \texttt{.mli} or \texttt{.ml} file, or both, depending on the files given on the command line. When both \texttt{.mli} and \texttt{.ml} files are given for the same module, information extracted from these files is merged according to the following rules:

\begin{itemize}
\item Only elements (values, types, classes, ...) declared in the \texttt{.mli} file are kept. In other terms, definitions from the \texttt{.ml} file that are not exported in the \texttt{.mli} file are not documented.
\item 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 \texttt{.ml} file is concatenated to the one in the \texttt{.mli} file, if the corresponding \texttt{-m} flag is given on the command line. If a description is present in the \texttt{.ml} file and not in the \texttt{.mli} file, the \texttt{.ml} description is kept. In either case, all the information given in the \texttt{.mli} file is kept.
\end{itemize}

15.1.3 Coding rules

The following rules must be respected in order to avoid name clashes resulting in cross-reference errors:

\begin{itemize}
\item 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 \texttt{ab} and \texttt{AB} will be printed to the same file on case insensitive file systems.
\item In a module, there must not be two classes, two class types or a class and a class type with the same name.
\item In a module, there must not be two values, two types, or two exceptions with the same name.
\item Values defined in tuple, as in \texttt{let (x,y,z) = (1,2,3)} are not kept by OCamlDoc.
\item Avoid the following construction:
\end{itemize}
open Foo (* which has a module Bar with a value x *)
module Foo =
  struct
    module Bar =
      struct
        let x = 1
      end
    end
let dummy = Bar.x

In this case, OCamlDoc will associate Bar.x to the x of module Foo defined just above, instead of to the Bar.x defined in the opened module Foo.

### 15.2 Syntax of documentation comments

Comments containing documentation material are called *special comments* and are written between (** and *). Special comments must start exactly with (**. Comments beginning with ( and more than two * are ignored.

#### 15.2.1 Placement of documentation comments

OCamlDoc can associate comments to some elements of the language encountered in the source files. The association is made according to the locations of comments with respect to the language elements. The locations of comments in .mli and .ml files are different.

**Comments in .mli files**

A special comment is associated to an element if it is placed before or after the element. A special comment before an element is associated to this element if:

- There is no blank line or another special comment between the special comment and the element. However, a regular comment can occur between the special comment and the element.
- The special comment is not already associated to the previous element.
- The special comment is not the first one of a toplevel module.

A special comment after an element is associated to this element if there is no blank line or comment between the special comment and the element.

There are two exceptions: for 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 = {
  foo : int ; (** Comment for field foo *)
  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 \rightarrow int
end
(** The comment for the class type my_class_type *)
class type my_class_type =
  object
    (** The comment for variable x. *)
    val mutable x : int
    (** The commend for method m. *)
    method m : int \rightarrow int
  end
(** The comment for module Foo *)
module Foo : 
sig
  (** The comment for x *)
  val x : int
  (** A special comment that is kept but not associated to any element *)
end
(** 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 :
    sig
      (** 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 = {
  foo : int ; (** Comment for field foo *)
  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 comment for method m. *)
    method m : int -> int
  end
(** The comment for module Foo *)
module Foo =
  struct
    (** The comment for x *)
    let x = 0
    (** 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

15.2.2 The Stop special comment

The special comment (**/**) tells OCaml to discard elements placed after this comment, up to the end of the current class, class type, module or module type, or up to the next stop comment. For instance:

class type foo =
  object
    (** comment for method m *)
    method m : string
    (**/**)
    (** This method won't appear in the documentation *)
    method bar : int
    end
(** This value appears in the documentation, since the Stop special comment in the class does not affect the parent module of the class.*)

val foo : string
(**/**)
(** The value bar does not appear in the documentation.*)

val bar : string
(**/**)
(** The type t appears since in the documentation since the previous stop comment toggled off the "no documentation mode". *)

The -no-stop option to ocamlc causes the Stop special comments to be ignored.

15.2.3 Syntax of documentation comments

The inside of documentation comments (**...**) consists of free-form text with optional formatting annotations, followed by optional tags giving more specific information about parameters, version,
authors, ... The tags are distinguished by a leading @ character. Thus, a documentation comment has the following shape:

(** The comment begins with a description, which is text formatted according to the rules described in the next section. The description continues until the first non-escaped '@' character.
   @author Mr Smith
   @param x description for parameter x
*)

Some elements support only a subset of all @-tags. Tags that are not relevant to the documented element are simply ignored. For instance, all tags are ignored when documenting type constructors, record fields, and class inheritance clauses. Similarly, a @param tag on a class instance variable is ignored.

At last, (***) is the empty documentation comment.

15.2.4 Text formatting

Here is the BNF grammar for the simple markup language used to format text descriptions.
\text ::= \{\text-element\}^+

text-element ::=  
| \{0\ldots9\}^+text \} format text as a section header; the integer following \{ indicates the sectioning level.  
| \{0\ldots9\}^+ : label text \} same, but also associate the name label to the current point. This point can be referenced by its fully-qualified label in a \{! command, just like any other element.  
| \{b text \} set text in bold.  
| \{i text \} set text in italic.  
| \{e text \} emphasize text.  
| \{C text \} center text.  
| \{L text \} left align text.  
| \{R text \} right align text.  
| \{ul list \} build a list.  
| \{ol list \} build an enumerated list.  
| \{:: string \} text \} put a link to the given address (given as \text) on the given text.  
| [ string \} set the given string in source code style.  
| {[[ string ]]} set the given string in preformatted source code style.  
| \{v string \} set the given string in verbatim style.  
| \{%, string %\} target-specific content (\LaTeX\ code by default, see details in \ref{15.2.4.4})  
| \{! string \} insert a cross-reference to an element (see section \ref{15.2.4.2} 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.

15.2.4.1 List formatting

list ::=  
| \{^ text \}^+  
| \{li text \}^+

A shortcut syntax exists for lists and enumerated lists:

(** Here is a \{b list\}
- item 1
- item 2
- item 3

The list is ended by the blank line.*)

is equivalent to:

(** Here is a {b list}
{ul {- item 1}
 {- item 2}
 {- item 3}}
The list is ended by the blank line.*)

The same shortcut is available for enumerated lists, using ‘+’ instead of ‘-’. Note that only one list can be defined by this shortcut in nested lists.

15.2.4.2 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.
15.2.4.3 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 }.

15.2.4.4 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.

15.2.4.5 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] \}.

15.2.5 Documentation tags (@-tags)

Predefined tags

The following table gives the list of predefined @-tags, with their syntax and meaning.
Chapter 15. The documentation generator (ocamldoc)

<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:

(** My comment to show you a custom tag.
@foo this is the text argument to the [foo] custom tag.
*)

To handle custom tags, you need to define a custom generator, as explained in section 15.3.2.

15.3 Custom generators

OCamldoc operates in two steps:

1. analysis of the source files;
2. generation of documentation, through a documentation generator, which is an object of class Odoc_args.class_generator.

Users can provide their own documentation generator to be used during step 2 instead of the default generators. All the information retrieved during the analysis step is available through the Odoc_info module, which gives access to all the types and functions representing the elements found in the given modules, with their associated description.

The files you can use to define custom generators are installed in the ocamldoc sub-directory of the OCaml standard library.
15.3.1 The generator 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 8.6).

The easiest way to define a custom generator is the following this example, here extending the current HTML generator. We don’t have to know if this is the original HTML generator defined in ocamltdoc 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;;

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`.

### 15.3.2 Handling custom tags

Making a custom generator handle custom tags (see 15.2.5) is very simple.

#### For HTML

Here is how to develop a HTML generator handling your custom tags.

The class `Odoc_html.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.
15.4 Adding command line options

The command line analysis is performed after loading the module containing the documentation generator, thus allowing command line options to be added to the list of existing ones. Adding an option can be done with the function

\[
\text{Odoc_args.add_option : string * Arg.spec * string -> unit}
\]

Note: Existing command line options can be redefined using this function.

15.4.1 Compilation and usage

Defining a custom generator class in one file

Let \texttt{custom.ml} be the file defining a new generator class. Compilation of \texttt{custom.ml} can be performed by the following command:

```
ocamlc -I +ocamldoc -c custom.ml
```

The file \texttt{custom.cmo} is created and can be used this way:

```
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{file\_1.ml[i]}, \texttt{file\_2.ml[i]}, ..., \texttt{file\_n.ml[i]}. A .cma library file must be created, including all these files.

The following commands create the \texttt{custom.cma} file from files \texttt{file\_1.ml[i]}, ..., \texttt{file\_n.ml[i]}:

```
ocamlc -I +ocamldoc -c file\_1.ml[i]
ocamlc -I +ocamldoc -c file\_2.ml[i]
...  
ocamlc -I +ocamldoc -c file\_n.ml[i]
ocamlc -o custom.cma -a file\_1.cmo file\_2.cmo ... file\_n.cmo
```

Then, the following command uses \texttt{custom.cma} as custom generator:

```
ocamldoc -g custom.cma other-options source-files
```
Chapter 16

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.

16.1 Compiling for debugging

Before the debugger can be used, the program must be compiled and linked with the -g option: all .cmo and .cma files that are part of the program should have been created with ocamlc -g, and they must be linked together with ocamlc -g.

Compiling with -g entails no penalty on the running time of programs: object files and bytecode executable files are bigger and take longer to produce, but the executable files run at exactly the same speed as if they had been compiled without -g.

16.2 Invocation

16.2.1 Starting the debugger

The OCaml debugger is invoked by running the program 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 16.10 for information on how to
run the debugger under Emacs.)

-I directory
Add directory to the list of directories searched for source files and compiled files. (See also
the directory command.)

-s socket
Use socket for communicating with the debugged program. See the description of the com-
mand set socket (section 16.8.6) for the format of socket.

-version
Print version string and exit.

-vnum
Print short version number and exit.

-help or -help
Display a short usage summary and exit.

16.2.2 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.

16.2.3 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.

16.3 Commands
A debugger command is a single line of input. It starts with a command name, which is followed
by arguments depending on this name. Examples:

run
goto 1000
set arguments arg1 arg2
A command name can be truncated as long as there is no ambiguity. For instance, `go 1000` is understood as `goto 1000`, since there are no other commands whose name starts with `go`. For the most frequently used commands, ambiguous abbreviations are allowed. For instance, `r` stands for `run` even though there are others commands starting with `r`. You can test the validity of an abbreviation using the `help` command.

If the previous command has been successful, a blank line (typing just `RET`) will repeat it.

### 16.3.1 Getting help

The OCaml debugger has a simple on-line help system, which gives a brief description of each command and variable.

- **help**
  - Print the list of commands.

- **help command**
  - Give help about the command `command`.

- **help set variable, help show variable**
  - Give help about the variable `variable`. The list of all debugger variables can be obtained with `help set`.

- **help info topic**
  - Give help about `topic`. Use `help info` to get a list of known topics.

### 16.3.2 Accessing the debugger state

- **set variable value**
  - Set the debugger variable `variable` to the value `value`.

- **show variable**
  - Print the value of the debugger variable `variable`.

- **info subject**
  - Give information about the given subject. For instance, `info breakpoints` will print the list of all breakpoints.

### 16.4 Executing a program

#### 16.4.1 Events

Events are “interesting” locations in the source code, corresponding to the beginning or end of evaluation of “interesting” sub-expressions. Events are the unit of single-stepping (stepping goes to the next or previous event encountered in the program execution). Also, breakpoints can only be set at events. Thus, events play the role of line numbers in debuggers for conventional languages.

During program execution, a counter is incremented at each event encountered. The value of this counter is referred as the `current time`. Thanks to reverse execution, it is possible to jump back and forth to any time of the execution.

Here is where the debugger events (written $\triangledown$) are located in the source code:
• Following a function application:
   \((f \text{ arg})\zx\)

• On entrance to a function:
   \text{fun } x \ y \ z \rightarrow \zx \ldots

• On each case of a pattern-matching definition (function, \texttt{match...with} construct, \texttt{try...with} construct):
   \text{function } \text{pat1 } \rightarrow \zx \text{expr1}
   \mid \ldots
   \mid \text{patN } \rightarrow \zx \text{exprN}

• Between subexpressions of a sequence:
   \text{expr1}; \zx \text{expr2}; \zx \ldots; \zx \text{exprN}

• In the two branches of a conditional expression:
   \text{if } \text{cond } \rightarrow \zx \text{expr1 } \text{else } \zx \text{expr2}

• At the beginning of each iteration of a loop:
   \text{while } \text{cond } \rightarrow \zx \text{body } \text{done}
   \text{for } i = a \text{ to } b \rightarrow \zx \text{body } \text{done}

Exceptions: A function application followed by a function return is replaced by the compiler by a jump (tail-call optimization). In this case, no event is put after the function application.

16.4.2 Starting the debugged program

The debugger starts executing the debugged program only when needed. This allows setting breakpoints or assigning debugger variables before execution starts. There are several ways to start execution:

\texttt{run} \hspace{1em} \text{Run the program until a breakpoint is hit, or the program terminates.}

\texttt{goto 0} \hspace{2em} \text{Load the program and stop on the first event.}

\texttt{goto time} \hspace{1em} \text{Load the program and execute it until the given time. Useful when you already know approximately at what time the problem appears. Also useful to set breakpoints on function values that have not been computed at time 0 (see section 16.5).}

The execution of a program is affected by certain information it receives when the debugger starts it, such as the command-line arguments to the program and its working directory. The debugger provides commands to specify this information (\texttt{set arguments} and \texttt{cd}). These commands must be used before program execution starts. If you try to change the arguments or the working directory after starting your program, the debugger will kill the program (after asking for confirmation).
16.4.3 Running the program

The following commands execute the program forward or backward, starting at the current time. The execution will stop either when specified by the command or when a breakpoint is encountered.

**run**
Execute the program forward from current time. Stops at next breakpoint or when the program terminates.

**reverse**
Execute the program backward from current time. Mostly useful to go to the last breakpoint encountered before the current time.

**step [count]**
Run the program and stop at the next event. With an argument, do it \textit{count} times. If \textit{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 \textit{count} times.

**next [count]**
Run the program and stop at the next event, skipping over function calls. With an argument, do it \textit{count} times.

**previous [count]**
Run the program backward and stop at the previous event, skipping over function calls. With an argument, do it \textit{count} times.

**finish**
Run the program until the current function returns.

**start**
Run the program backward and stop at the first event before the current function invocation.

16.4.4 Time travel

You can jump directly to a given time, without stopping on breakpoints, using the \texttt{goto} command.

As you move through the program, the debugger maintains an history of the successive times you stop at. The \texttt{last} command can be used to revisit these times: each \texttt{last} command moves one step back through the history. That is useful mainly to undo commands such as \texttt{step} and \texttt{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 \textit{count} times.

**set history size**
Set the size of the execution history.
16.4.5 Killing the program

**kill**

Kill the program being executed. This command is mainly useful if you wish to recompile the program without leaving the debugger.

16.5 Breakpoints

A breakpoint causes the program to stop whenever a certain point in the program is reached. It can be set in several ways using the `break` command. Breakpoints are assigned numbers when set, for further reference. The most comfortable way to set breakpoints is through the Emacs interface (see section 16.10).

**break**

Set a breakpoint at the current position in the program execution. The current position must be on an event (i.e., neither at the beginning, nor at the end of the program).

**break function**

Set a breakpoint at the beginning of `function`. This works only when the functional value of the identifier `function` has been computed and assigned to the identifier. Hence this command cannot be used at the very beginning of the program execution, when all identifiers are still undefined; use `goto time` to advance execution until the functional value is available.

**break @ [module] line**

Set a breakpoint in module `module` (or in the current module if `module` is not given), at the first event of line `line`.

**break @ [module] line column**

Set a breakpoint in module `module` (or in the current module if `module` is not given), at the event closest to line `line`, column `column`.

**break @ [module] # character**

Set a breakpoint in module `module` at the event closest to character number `character`.

**break address**

Set a breakpoint at the code address `address`.

**delete [breakpoint-numbers]**

Delete the specified breakpoints. Without argument, all breakpoints are deleted (after asking for confirmation).

**info breakpoints**

Print the list of all breakpoints.

16.6 The call stack

Each time the program performs a function application, it saves the location of the application (the return address) in a block of data called a stack frame. The frame also contains the local variables
of the caller function. All the frames are allocated in a region of memory called the call stack. The command `backtrace` (or `bt`) displays parts of the call stack.

At any time, one of the stack frames is “selected” by the debugger; several debugger commands refer implicitly to the selected frame. In particular, whenever you ask the debugger for the value of a local variable, the value is found in the selected frame. The commands `frame`, `up` and `down` select whichever frame you are interested in.

When the program stops, the debugger automatically selects the currently executing frame and describes it briefly as the `frame` command does.

```plaintext
frame
Describe the currently selected stack frame.

frame frame-number
Select a stack frame by number and describe it. The frame currently executing when the program stopped has number 0; its caller has number 1; and so on up the call stack.

backtrace [count], bt [count]
Print the call stack. This is useful to see which sequence of function calls led to the currently executing frame. With a positive argument, print only the innermost `count` frames. With a negative argument, print only the outermost `-count` frames.

up [count]
Select and display the stack frame just “above” the selected frame, that is, the frame that called the selected frame. An argument says how many frames to go up.

down [count]
Select and display the stack frame just “below” the selected frame, that is, the frame that was called by the selected frame. An argument says how many frames to go down.
```

### 16.7 Examining variable values

The debugger can print the current value of simple expressions. The expressions can involve program variables: all the identifiers that are in scope at the selected program point can be accessed.

Expressions that can be printed are a subset of OCaml expressions, as described by the following grammar:

```plaintext
simple-expr ::= lowercase-ident
              | \{capitalized-ident .\} lowercase-ident
              | *
              | $ integer
              | simple-expr . lowercase-ident
              | simple-expr . ( integer )
              | simple-expr . [ integer ]
              | ! simple-expr
              | ( simple-expr )
```

The first two cases refer to a value identifier, either unqualified or qualified by the path to the structure that define it. `*` refers to the result just computed (typically, the value of a function
application), and is valid only if the selected event is an “after” event (typically, a function application). $integer refer to a previously printed value. The remaining four forms select part of an expression: respectively, a record field, an array element, a string element, and the current contents of a reference.

print variables
Print the values of the given variables. print can be abbreviated as p.

display variables
Same as print, but limit the depth of printing to 1. Useful to browse large data structures without printing them in full. display can be abbreviated as d.

When printing a complex expression, a name of the form $integer is automatically assigned to its value. Such names are also assigned to parts of the value that cannot be printed because the maximal printing depth is exceeded. Named values can be printed later on with the commands p $integer or d $integer. Named values are valid only as long as the program is stopped. They are forgotten as soon as the program resumes execution.

set print_depth d
Limit the printing of values to a maximal depth of d.

set print_length l
Limit the printing of values to at most l nodes printed.

16.8 Controlling the debugger

16.8.1 Setting the program name and arguments

set program file
Set the program name to file.

set arguments arguments
Give arguments as command-line arguments for the program.

A shell is used to pass the arguments to the debugged program. You can therefore use wildcards, shell variables, and file redirections inside the arguments. To debug programs that read from standard input, it is recommended to redirect their input from a file (using set arguments < input-file), otherwise input to the program and input to the debugger are not properly separated, and inputs are not properly replayed when running the program backwards.

16.8.2 How programs are loaded
The loadingmode variable controls how the program is executed.

set loadingmode direct
The program is run directly by the debugger. This is the default mode.
set loadingmode runtime
The debugger execute the OCaml runtime ocamlrun on the program. Rarely useful; moreover it prevents the debugging of programs compiled in “custom runtime” mode.

set loadingmode manual
The user starts manually the program, when asked by the debugger. Allows remote debugging (see section 16.8.6).

16.8.3 Search path for files
The debugger searches for source files and compiled interface files in a list of directories, the search path. The search path initially contains the current directory . and the standard library directory. The directory command adds directories to the path.
Whenever the search path is modified, the debugger will clear any information it may have cached about the files.

directory directorynames
Add the given directories to the search path. These directories are added at the front, and will therefore be searched first.

directory directorynames for modulename
Same as directory directorynames, but the given directories will be searched only when looking for the source file of a module that has been packed into modulename.

directory
Reset the search path. This requires confirmation.

16.8.4 Working directory
Each time a program is started in the debugger, it inherits its working directory from the current working directory of the debugger. This working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in the debugger with the cd command or the -cd command-line option.

cd directory
Set the working directory for ocamldebug to directory.
pwd Print the working directory for ocamldebug.

16.8.5 Turning reverse execution on and off
In some cases, you may want to turn reverse execution off. This speeds up the program execution, and is also sometimes useful for interactive programs.

Normally, the debugger takes checkpoints of the program state from time to time. That is, it makes a copy of the current state of the program (using the Unix system call fork). If the variable checkpoints is set to off, the debugger will not take any checkpoints.

set checkpoints on/off
Select whether the debugger makes checkpoints or not.
16.8.6 Communication between the debugger and the program

The debugger communicate with the program being debugged through a Unix socket. You may need to change the socket name, for example if you need to run the debugger on a machine and your program on another.

```
set socket socket
```

Use `socket` for communication with the program. `socket` can be either a file name, or an Internet port specification `host:port`, where `host` is a host name or an Internet address in dot notation, and `port` is a port number on the host.

On the debugged program side, the socket name is passed through the `CAML_DEBUG_SOCKET` environment variable.

16.8.7 Fine-tuning the debugger

Several variables enables to fine-tune the debugger. Reasonable defaults are provided, and you should normally not have to change them.

```
set processcount count
```

Set the maximum number of checkpoints to `count`. More checkpoints facilitate going far back in time, but use more memory and create more Unix processes.

As checkpointing is quite expensive, it must not be done too often. On the other hand, backward execution is faster when checkpoints are taken more often. In particular, backward single-stepping is more responsive when many checkpoints have been taken just before the current time. To fine-tune the checkpointing strategy, the debugger does not take checkpoints at the same frequency for long displacements (e.g. `run`) and small ones (e.g. `step`). The two variables `bigstep` and `smallstep` contain the number of events between two checkpoints in each case.

```
set bigstep count
```

Set the number of events between two checkpoints for long displacements.

```
set smallstep count
```

Set the number of events between two checkpoints for small displacements.

The following commands display information on checkpoints and events:

```
info checkpoints
```

Print a list of checkpoints.

```
info events [module]
```

Print the list of events in the given module (the current module, by default).
16.8.8 User-defined printers

Just as in the toplevel system (section 10.2), 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.

\texttt{load\_printer \textasciitilde file-name}

Load in the debugger the indicated \texttt{.cmo} or \texttt{.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 \texttt{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.

\texttt{install\_printer printer-name}

Register the function named \texttt{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 \texttt{printer-name} when it has such an object to print. The printing function \texttt{printer-name} must use the \texttt{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 \texttt{printer-name} must refer to one of the functions defined by the object files loaded using \texttt{load\_printer}. It cannot reference the functions of the program being debugged.

\texttt{remove\_printer printer-name}

Remove the named function from the table of value printers.

16.9 Miscellaneous commands

\texttt{list [module] [beginning] [end]}

List the source of module \texttt{module}, from line number \texttt{beginning} to line number \texttt{end}. By default, 20 lines of the current module are displayed, starting 10 lines before the current position.

\texttt{source filename}

Read debugger commands from the script \texttt{filename}.

16.10 Running the debugger under Emacs

The most user-friendly way to use the debugger is to run it under Emacs. See the file \texttt{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 \texttt{M-x camldebug}, with argument the name of the executable file \texttt{profilename} to debug. Communication with the debugger takes place in an Emacs buffer named \texttt{*camldebug-profilename*}. 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 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 R
(command run): execute the program forward to next breakpoint.

C-c V
(command reverse): execute the program backward to latest breakpoint.

C-c L
(command last): go back one step in the command history.

C-c T
(command backtrace): display backtrace of function calls.

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 17

Profiling (ocamlprof)

This chapter describes how the execution of OCaml programs can be profiled, by recording how many times functions are called, branches of conditionals are taken, ...

17.1 Compiling for profiling

Before profiling an execution, the program must be compiled in profiling mode, using the ocamlcp front-end to the ocamlc compiler (see chapter 9) or the ocamloptp front-end to the ocamlopt compiler (see chapter 12). 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

307
t try ... with ... branches: a count point is set at the beginning of the body of each branch

For instance, compiling with ocamlcp -P film profiles function calls, if...then...else..., loops
and pattern matching.

Calling ocamlcp 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.

17.2 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).

17.3 Printing profiling information

The ocamlprof command produces a source listing of the program modules where execution counts
have been inserted as comments. For instance,

    ocamlprof foo.ml

prints the source code for the foo module, with comments indicating how many times the
functions in this module have been called. Naturally, this information is accurate only if the source
file has not been modified after it was compiled.

The following options are recognized by ocamlprof:

-args filename
    Read additional newline-terminated command line arguments from filename.

-args0 filename
    Read additional null character terminated command line arguments from filename.

-f dumpfile
    Specifies an alternate dump file of profiling information to be read.
-F string
  Specifies an additional string to be output with profiling information. By default, ocamlprof will annotate programs with comments of the form (* n *) where n is the counter value for a profiling point. With option -F s, the annotation will be (* sn *).

-impl filename
  Process the file filename as an implementation file, even if its extension is not .ml.

-intf filename
  Process the file filename as an interface file, even if its extension is not .mli.

-version
  Print version string and exit.

-vnum
  Print short version number and exit.

-help or -help
  Display a short usage summary and exit.

17.4 Time profiling

Profiling with ocamlprof only records execution counts, not the actual time spent 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 18

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 19

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.

19.1 Overview and compilation information

19.1.1 Declaring primitives

\[
\textit{definition} ::= \ldots
\]

\[
| \textit{external} \ \textit{value-name} : \textit{typexpr} = \textit{external-declaration}
\]

\[
\textit{external-declaration} ::= \textit{string-literal} \ [\textit{string-literal} \ [\textit{string-literal}]]
\]

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}:

\[
\texttt{external input : in\_channel \to bytes \to int \to int \to int}
\]

\[
= \ "\text{input}\"
\]

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,

```ocaml
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,

```ocaml
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.

### 19.1.2 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:

```c
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 19.4.5)</td>
</tr>
<tr>
<td>caml/callback.h</td>
<td>callback from C to OCaml (see section 19.7)</td>
</tr>
<tr>
<td>caml/custom.h</td>
<td>operations on custom blocks (see section 19.9)</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 19.9).</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 19.12).</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.

### 19.1.3 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 19.1.4 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:

```
ocamlc -a -o mylib.cma -custom a.cmo b.cmo -cclib -lmylib
```

users of the library can simply link with `mylib.cma`:

```
ocamlc -o myprog mylib.cma ...
```

and the system will automatically add the `-custom` and `-cclib -lmylib` options, achieving the same effect as

```
ocamlc -o myprog -custom a.cmo b.cmo ... -cclib -lmylib
```

The alternative is of course to build the library without extra options:

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

and then ask users to provide the `-custom` and `-cclib -lmylib` options themselves at link-time:

```
ocamlc -o myprog -custom mylib.cma ... -cclib -lmylib
```

The former alternative is more convenient for the final users of the library, however.
19.1.4 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 ocamlrun then loads dynamically these libraries, and resolves references to the required primitives, before executing the bytecode.

This facility is currently supported and known to work well under Linux, MacOS X, and Windows. It is supported, but not fully tested yet, under FreeBSD, Tru64, Solaris and Irix. It is not supported yet under other Unixes.

To dynamically link C code with 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 appropriate C compiler flags for producing position-independent code (when required by the operating system), and 2- building a shared library from the resulting object files. The resulting shared library or DLL file must be installed in a place where ocamlrun can find it later at program start-up time (see section 11.3). 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 -dll -lname.

Do not set the -custom flag, otherwise you’re back to static linking as described in section 19.1.3. The ocamlmklib tool (see section 19.14) 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:

```
ocamlc -a -o mylib.cma a.cmo b.cmo -dll -lmylib
```

users of the library can simply link with mylib.cma:

```
ocamlc -o myprog mylib.cma ...
```

and the system will automatically add the -dll -lmylib option, achieving the same effect as

```
ocamlc -o myprog a.cmo b.cmo ... -dll -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.

19.1.5 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 bytecode executables. That is, the bytecode executable contains no machine code, and can therefore be
Chapter 19. 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 19.14) 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.

19.1.6 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 *ocamlc* knows which C primitives are required) and also when building the bytecode executable (so that the bytecode from *unix.cma* and *threads.cma* is actually linked in).

## 19.2 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).

### 19.2.1 Integer values

Integer values encode 63-bit signed integers (31-bit on 32-bit architectures). They are unboxed (unallocated).

### 19.2.2 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:
<table>
<thead>
<tr>
<th>Tag</th>
<th>Contents of the block</th>
</tr>
</thead>
<tbody>
<tr>
<td>0 to No_scan_tag−1</td>
<td>A structured block (an array of OCaml objects). Each field is a value.</td>
</tr>
<tr>
<td>Closure_tag</td>
<td>A closure representing a functional value. The first word is a pointer to a piece of code, the remaining words are value containing the environment.</td>
</tr>
<tr>
<td>String_tag</td>
<td>A character string or a byte sequence.</td>
</tr>
<tr>
<td>Double_tag</td>
<td>A double-precision floating-point number.</td>
</tr>
<tr>
<td>Double_array_tag</td>
<td>An array or record of double-precision floating-point numbers.</td>
</tr>
<tr>
<td>Abstract_tag</td>
<td>A block representing an abstract datatype.</td>
</tr>
<tr>
<td>Custom_tag</td>
<td>A block representing an abstract datatype with user-defined finalization, comparison, hashing, serialization and deserialization functions attached.</td>
</tr>
</tbody>
</table>

19.2.3 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.

19.3 Representation of OCaml data types

This section describes how OCaml data types are encoded in the value type.

19.3.1 Atomic types

<table>
<thead>
<tr>
<th>OCaml type</th>
<th>Encoding</th>
</tr>
</thead>
<tbody>
<tr>
<td>int</td>
<td>Unboxed integer values.</td>
</tr>
<tr>
<td>char</td>
<td>Unboxed integer values (ASCII code).</td>
</tr>
<tr>
<td>float</td>
<td>Blocks with tag Double_tag.</td>
</tr>
<tr>
<td>bytes</td>
<td>Blocks with tag String_tag.</td>
</tr>
<tr>
<td>string</td>
<td>Blocks with tag String_tag.</td>
</tr>
<tr>
<td>int32</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
<tr>
<td>int64</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
<tr>
<td>nativeint</td>
<td>Blocks with tag Custom_tag.</td>
</tr>
</tbody>
</table>

19.3.2 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 unboxable 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.

19.3.3 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.

19.3.4 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:

```haskell
type t =
| A          (* First constant constructor -> integer "Val_int(0)" *)
| B of string (* First non-constant constructor -> block with tag 0 *)
| C          (* Second constant constructor -> integer "Val_int(1)" *)
| D of bool  (* Second non-constant constructor -> block with tag 1 *)
| E of t * t  (* Third non-constant constructor -> block with tag 2 *)
```

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 19.3.2.

### 19.3.5 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:

```c
callback(caml_get_public_method(foo, hash_variant("bar")), foo);
```

### 19.3.6 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.
19.4 Operations on values

19.4.1 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.

19.4.2 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.

19.4.3 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 v, with type char * or, when OCaml is configured with -force-safe-string, with type const char *. This pointer is a valid C string: there is a null byte after the last byte in the string. However, OCaml strings can contain embedded null bytes, which will confuse the usual C functions over strings.
• Bytes_val($v$) returns a pointer to the first byte of the byte sequence $v$, with type unsigned char *.

• 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).

19.4.4 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_alloc_initialized_string($n$, $p$) returns a byte sequence (or string) value of length $n$ bytes. The value is initialized from the $n$ bytes starting at address $p$. 
• **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 $n \leq Max_young_wosize$ words, with tag $t$. If this block is a structured block (i.e., if $t < No_scan_tag$), then the fields of the block (initially containing garbage) must be initialized with legal values (using direct assignment to the fields of the block) before the next allocation.

• **caml_alloc_shr**($n$, $t$) returns a fresh block of size $n$, with tag $t$. The size of the block can be greater than `Max_young_wosize`. (It can also be smaller, but in this case it is more efficient to call `caml_alloc_small` instead of `caml_alloc_shr`.) If this block is a structured block
(i.e. if \( t < \text{No\_scan\_tag} \)), then the fields of the block (initially containing garbage) must be initialized with legal values (using the \text{caml\_initialize} function described below) before the next allocation.

### 19.4.5 Raising exceptions

Two functions are provided to raise two standard exceptions:

- \text{caml\_failwith(s)}, where \( s \) is a null-terminated C string (with type \text{char *}), raises exception \text{Failure} with argument \( s \).
- \text{caml\_invalid\_argument(s)}, where \( s \) is a null-terminated C string (with type \text{char *}), raises exception \text{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 19.7.3. Once the exception identifier is recovered in C, the following functions actually raise the exception:

- \text{caml\_raise\_constant(id)} raises the exception \( id \) with no argument;
- \text{caml\_raise\_with\_arg(id, v)} raises the exception \( id \) with the OCaml value \( v \) as argument;
- \text{caml\_raise\_with\_args(id, n, v)} raises the exception \( id \) with the OCaml values \( v[0] \), \ldots, \( v[n-1] \) as arguments;
- \text{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.

### 19.5 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.

#### 19.5.1 Simple interface

All the macros described in this section are declared in the \text{memory.h} header file.

**Rule 1** A function that has parameters or local variables of type \text{value} must begin with a call to one of the \text{CAMLparam} macros and return with \text{CAMLreturn}, \text{CAMLreturn0}, or \text{CAMLreturnT}. In particular, \text{CAMLlocal} and \text{CAMLxparam} can only be called after \text{CAMLparam}.

There are six \text{CAMLparam} macros: \text{CAMLparam0} to \text{CAMLparam5}, which take zero to five arguments respectively. If your function has no more than 5 parameters of type \text{value}, use the corresponding macros with these parameters as arguments. If your function has more than 5 parameters of type \text{value}, use \text{CAMLparam5} with five of these parameters, and use one or more calls to the \text{CAMLxparam} macros for the remaining parameters (\text{CAMLxparam1} to \text{CAMLxparam5}).

The macros \text{CAMLreturn}, \text{CAMLreturn0}, and \text{CAMLreturnT} are used to replace the C keyword \text{return}. Every occurrence of \text{return x} must be replaced by \text{CAMLreturn (x)} if \( x \) has type \text{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:
value bar (value v1, value v2, value v3)
{
    CAMLparam3 (v1, v2, v3);
    CAMLlocal1 (result);
    result = caml_alloc (3, 0);
    Store_field (result, 0, v1);
    Store_field (result, 1, v2);
    Store_field (result, 2, v3);
    CAMLreturn (result);
}

Warning: The first argument of Store_field and Store_double_field must be a variable declared by CAMLparam* or a parameter declared by CAMLlocal* to ensure that a garbage collection triggered by the evaluation of the other arguments will not invalidate the first argument after it is computed.

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 CAML macros use identifiers (local variables, type identifiers, structure tags) that start with caml__. Do not use any identifier starting with caml__ in your programs.

19.5.2 Low-level interface

We now give the GC rules corresponding to the low-level allocation functions caml_alloc_small and caml_alloc_shr. You can ignore those rules if you stick to the simplified allocation function caml_alloc.

Rule 5 After a structured block (a block with tag less than No_scan_tag) is allocated with the low-level functions, all fields of this block must be filled with well-formed values before the next
allocation operation. If the block has been allocated with `caml_alloc_small`, filling is performed by
direct assignment to the fields of the block:

\[
\text{Field}(v, n) = v_n;
\]

If the block has been allocated with `caml_alloc_shr`, filling is performed through the
`caml_initialize` function:

\[
\text{caml_initialize}(&\text{Field}(v, n), v_n);
\]

The next allocation can trigger a garbage collection. The garbage collector assumes that all
structured blocks contain well-formed values. Newly created blocks contain random data, which
generally do not represent well-formed values.

If you really need to allocate before the fields can receive their final value, first initialize with
a constant value (e.g. `Val_unit`), then allocate, then modify the fields with the correct value (see
rule 6).

**Rule 6** Direct assignment to a field of a block, as in

\[
\text{Field}(v, n) = w;
\]

is safe only if \(v\) is a block newly allocated by `caml_alloc_small`; that is, if no allocation took
place between the allocation of \(v\) and the assignment to the field. In all other cases, never assign
directly. If the block has just been allocated by `caml_alloc_shr`, use `caml_initialize` to assign a
value to a field for the first time:

\[
\text{caml_initialize}(&\text{Field}(v, n), w);
\]

Otherwise, you are updating a field that previously contained a well-formed value; then, call the
`caml_modify` function:

\[
\text{caml_modify}(&\text{Field}(v, n), w);
\]

To illustrate the rules above, here is a C function that builds and returns a list containing the
two integers given as parameters. First, we write it using the simplified allocation functions:

```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 \texttt{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 \texttt{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 \texttt{caml_alloc_small}, and filled by direct assignments on their fields.

\begin{verbatim}
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);
}
\end{verbatim}

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 \texttt{caml_modify}.

\begin{verbatim}
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);
}
\end{verbatim}

It would be incorrect to perform \texttt{Field(r, 1) = tail} directly, because the allocation of \texttt{tail} has taken place since \texttt{r} was allocated.

19.6 A complete example

This section outlines how the functions from the Unix \texttt{curses} library can be made available to OCaml programs. First of all, here is the interface \texttt{curses.ml} that declares the \texttt{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"
external wrefresh : window -> unit = "caml_curses_wrefresh"
external newwin: int -> int -> int -> int -> window = "caml_curses_newwin"
external addch: char -> unit = "caml_curses_addch"
external mvwaddch: window -> int -> int -> char -> unit = "caml_curses_mvwaddch"
external addstr: string -> unit = "caml_curses_addstr"
external mvwaddstr: window -> int -> int -> string -> unit = "caml_curses_mvwaddstr"
(* lots more omitted *)

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:

/* File curses_stubs.c -- stub code for curses */
#include <curses.h>
#include <caml/mlvalues.h>
#include <caml/memory.h>
#include <caml/alloc.h>
#include <caml/custom.h>

/* Encapsulation of opaque window handles (of type WINDOW *) as 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_fixed_length_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));
    CAMLreturn (Val_unit);
}

value caml_curses_addstr(value s)
{
    CAMLparam1 (s);
    addstr(String_val(s));
    CAMLreturn (Val_unit);
}

value caml_curses_mvwaddstr(value win, value x, value y, value s)
{
    CAMLparam4 (win, x, y, s);
    mvwaddstr(Window_val(win), Int_val(x), Int_val(y), String_val(s));
    CAMLreturn (Val_unit);
}

/* This goes on for pages. */

The file curses_stubs.c can be compiled with:

    cc -c -I`ocamlc -where`/curses_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.)

### 19.7 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.

#### 19.7.1 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:**
value call_caml_f_ex(value closure, value arg)
{
  CAMLparam2(closure, arg);
  CAMLlocal2(res, tmp);
  res = caml_callback_exn(closure, arg);
  if(Is_exception_result(res)) {
    res = Extract_exception(res);
    tmp = caml_alloc(3, 0); /* Safe to allocate: res contains valid value. */
    ...
  }
  CAMLreturn (res);
}

19.7.2 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:

```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));
}
```

### 19.7.3 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 19.4.5) 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);
}
```

### 19.7.4 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 19.7.1).

### 19.7.5 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 19.7.4, 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 `caml_startup` function calls the uncaught exception handler (or enters the debugger, if running under ocamldebug) if an exception escapes from a top-level module initialiser. Such exceptions may be caught in the C code by instead using the `caml_startup_exn` function and testing the result using `Is_exception_result` (followed by `Extract_exception` if appropriate).

The `-output-obj` option can also be used to obtain the C source file. More interestingly, the same option can also produce directly a shared library (.so file, .dll under Windows) that contains the OCaml code, 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` (or `caml_startup_exn`) 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, `libasmarun.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 \texttt{ocamlc}, you should do:

\begin{verbatim}
ocamlc -output-obj -o camlcode.o unix.cma other .cmo and .cma files cc -o myprog C objects and libraries \ 
  camlcode.o -L'ocamlc -where' -lunix -lcamlrun
\end{verbatim}

With \texttt{ocamlopt}, you should do:

\begin{verbatim}
ocamlopt -output-obj -o camlcode.o unix.cmxa other .cma and .cmxa files cc -o myprog C objects and libraries \ 
  camlcode.o -L'ocamlc -where' -lunix -lasmrun
\end{verbatim}

\textbf{Warning:} On some ports, special options are required on the final linking phase that links together the object file produced by the \texttt{-output-obj} option and the remainder of the program. Those options are shown in the configuration file \texttt{Makefile.config} generated during compilation of OCaml, as the variable \texttt{OC_LDFLAGS}.

- Windows with the MSVC compiler: the object file produced by OCaml have been compiled with the \texttt{/MD} flag, and therefore all other object files linked with it should also be compiled with \texttt{/MD}.
- other systems: you may have to add one or more of \texttt{-lcurses}, \texttt{-lm}, \texttt{-ldl}, depending on your OS and C compiler.

\textbf{Stack backtraces.} When OCaml bytecode produced by \texttt{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 \texttt{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 \texttt{Printexc.record_backtrace true} in the initialization of one of the OCaml modules. This can also be achieved from the C side by calling \texttt{caml_record_backtrace(Val_int(1))}; in the OCaml-C glue code.

\textbf{Unloading the runtime.}

In case the shared library produced with \texttt{-output-obj} is to be loaded and unloaded repeatedly by a single process, care must be taken to unload the OCaml runtime explicitly, in order to avoid various system resource leaks.

Since 4.05, \texttt{caml_shutdown} function can be used to shut the runtime down gracefully, which equals the following:

- Running the functions that were registered with \texttt{Pervasives.at_exit}.
- Triggering finalization of allocated custom blocks (see section \texttt{19.9}). For example, \texttt{Pervasives.in_channel} and \texttt{Pervasives.out_channel} are represented by custom blocks that enclose file descriptors, which are to be released.
• Unloading the dependent shared libraries that were loaded by the runtime, including `dynlink` plugins.

• Freeing the memory blocks that were allocated by the runtime with `malloc`. Inside C primitives, it is advised to use `caml_stat_*` functions from `memory.h` for managing static (that is, non-moving) blocks of heap memory, as all the blocks allocated with these functions are automatically freed by `caml_shutdown`. For ensuring compatibility with legacy C stubs that have used `caml_stat_*` incorrectly, this behaviour is only enabled if the runtime is started with a specialized `caml_startup_pooled` function.

As a shared library may have several clients simultaneously, it is made for convenience that `caml_startup` (and `caml_startup_pooled`) may be called multiple times, given that each such call is paired with a corresponding call to `caml_shutdown` (in a nested fashion). The runtime will be unloaded once there are no outstanding calls to `caml_startup`.

Once a runtime is unloaded, it cannot be started up again without reloading the shared library and reinitializing its static data. Therefore, at the moment, the facility is only useful for building reloadable shared libraries.

### 19.8 Advanced example with callbacks

This section illustrates the callback facilities described in section 19.7. 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 */

#include <stdio.h>
#include <string.h>
#include <caml/mlvalues.h>
#include <caml/callback.h>

int fib(int n)
{
```

```
Chapter 19. Interfacing C with OCaml

341

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 (or caml_startup_exn) 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));
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`.)

### 19.9 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.

#### 19.9.1 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 * bsize_32, uintnat * bsize_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 **bsize_32** and **bsize_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 **register_custom_operations** (see below).

- **const struct custom_fixed_length* fixed_length**
  Normally, space in the serialized output is reserved to write the **bsize_32** and **bsize_64** fields returned by **serialize**. However, for very short custom blocks, this space can be larger than the data itself! As a space optimisation, if **serialize** always returns the same values for **bsize_32** and **bsize_64**, then these values may be specified in the **fixed_length** structure, and do not consume space in the serialized output.
Note: the finalize, compare, hash, serialize and 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 CAMLparam to register the parameters to these functions, and do not use CAMLreturn to return the result.

19.9.2 Allocating custom blocks

Custom blocks must be allocated via caml_alloc_custom or caml_alloc_custom_mem:

```
caml_alloc_custom(ops, size, used, max)
```

returns a fresh custom block, with room for size bytes of user data, and whose associated operations are given by ops (a pointer to a struct custom_operations, usually statically allocated as a C global variable).

The two parameters used and max are used to control the speed of garbage collection when the finalized object contains pointers to out-of-heap resources. Generally speaking, the 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 used and max give the GC an idea of how much out-of-heap resources are consumed by the finalized block being allocated: you give the amount of resources allocated to this object as parameter used, and the maximum amount that you want to see in floating garbage as parameter max. The units are arbitrary: the GC cares only about the ratio used/max.

For instance, if you are allocating a finalized block holding an X Windows bitmap of w by h pixels, and you’d rather not have more than 1 mega-pixels of unreclaimed bitmaps, specify used = w * h and max = 1000000.

Another way to describe the effect of the used and max parameters is in terms of full GC cycles. If you allocate many custom blocks with 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 N allocations. For instance, if used = 1 and 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 used = 0 and max = 1. But if you later find that the finalization functions are not called “often enough”, consider increasing the used/max ratio.

```
caml_alloc_custom_mem(ops, size, used)
```

Use this function when your custom block holds only out-of-heap memory (memory allocated with malloc or caml_stat_alloc) and no other resources. used should be the number of bytes of out-of-heap memory that are held by your custom block. This function works like caml_alloc_custom except that the max parameter is under the control of the user (via the custom_major_ratio,
custom_minor_ratio, and custom_minor_max_size parameters) and proportional to the heap sizes.

19.9.3 Accessing custom blocks

The data part of a custom block \( v \) can be accessed via the pointer `Data_custom_val(v)`. This pointer has type `void *` and should be cast to the actual type of the data stored in the custom block.

The contents of custom blocks are not scanned by the garbage collector, and must therefore not contain any pointer inside the 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.

19.9.4 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 `struct custom_operation` blocks for one with the same identifier, and calling its `deserialize` function to fill the data part of the custom block.

### 19.9.5 Choosing identifiers

Identifiers in `struct custom_operations` must be chosen carefully, since they must identify uniquely the data structure for serialization and deserialization operations. In particular, consider including a version number in the identifier; this way, the format of the data can be changed later, yet backward-compatible deserialisation functions can be provided.

Identifiers starting with `_` (an underscore character) are reserved for the OCaml runtime 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.

19.9.6 Finalized blocks

Custom blocks generalize the finalized blocks that were present in OCaml prior to version 3.00. For backward compatibility, the format of custom blocks is compatible with that of finalized blocks, and the alloc_final function is still available to allocate a custom block with a given finalization function, but default comparison, hashing and serialization functions.

caml_alloc_final(n, f, used, max) returns a fresh custom block of size n+1 words, with finalization function f. The first word is reserved for storing the custom operations; the other n words are available for your data. The two parameters used and max are used to control the speed of garbage collection, as described for caml_alloc_custom.

19.10 Advanced topic: Bigarrays and the OCaml-C interface

This section explains how C stub code that interfaces C or Fortran code with OCaml code can use Bigarrays.

19.10.1 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.

19.10.2 Accessing an OCaml bigarray from C or Fortran

If v is a OCaml value representing a Bigarray, 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 Bigarray can be consulted from C as follows:

<table>
<thead>
<tr>
<th>C expression</th>
<th>Returns</th>
</tr>
</thead>
<tbody>
<tr>
<td>Caml_ba_array_val(v)-&gt;num_dims</td>
<td>number of dimensions</td>
</tr>
<tr>
<td>Caml_ba_array_val(v)-&gt;dim[i]</td>
<td>i-th dimension</td>
</tr>
<tr>
<td>Caml_ba_array_val(v)-&gt;flags &amp; BIGARRAY_KIND_MASK</td>
<td>kind of array elements</td>
</tr>
</tbody>
</table>

The kind of array elements is one of the following constants:
The following example shows the passing of a two-dimensional Bigarray to a C function and a Fortran function.

```ocaml
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;
}
```

### 19.10.3 Wrapping a C or Fortran array as an OCaml Bigarray

A pointer `p` to an already-allocated C or Fortran array can be wrapped and returned to OCaml as a Bigarray using the `caml_ba_alloc` or `caml_ba_alloc_dims` functions.

- **caml_ba_alloc(kind | layout, numdims, p, dims)**

  Return an OCaml Bigarray wrapping the data pointed to by `p`. `kind` is the kind of array elements (one of the `CAML_BA_` kind constants above). `layout` is `CAML_BA_C_LAYOUT` for an array with C layout and `CAML_BA_FORTRAN_LAYOUT` for an array with Fortran layout. `numdims` is the number of dimensions in the array. `dims` is an array of `numdims` long integers, giving the sizes of the array in each dimension.

- **caml_ba_alloc_dims(kind | layout, numdims, p, (long) dim1, (long) dim2, ..., (long) dimnumdims)**

  Same as `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.
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);
}

19.11 Advanced topic: cheaper C call

This section describe how to make calling C functions cheaper.

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.

19.11.1 Passing unboxed values

We said earlier that all OCaml objects are represented by the C type value, and one has to use macros such as Int_val to decode data from the value type. It is however possible to tell the OCaml native-code compiler 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 ‘ocamlopt’ deal with boxing, it can often decide to suppress it entirely.

For instance let’s consider this example:

external foo : float -> float -> float = "foo"

let f a b =
    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 native-code compiler will be able to avoid all these allocations:

```
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 convenicence, 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:

```
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 [@untagged]:

```
external f : string -> (int [@untagged]) = "f_byte" "f"
```

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.
19.11.2 Direct C call

In order to be able to run the garbage collector in the middle of a C function, the OCaml native-code 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 native-code 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...

19.11.3 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. */)
```

19.12 Advanced topic: multithreading

Using multiple threads (shared-memory concurrency) in a mixed OCaml/C application requires special precautions, which are described in this section.

19.12.1 Registering threads created from C

Callbacks from C to OCaml are possible only if the calling thread is known to the OCaml run-time 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.

### 19.12.2 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;
  char * name;

  /* Copy the string argument to a C string, allocated outside the OCaml heap. */
  name = caml_stat_strdup(String_val(vname));
  /* Release the OCaml run-time system */
  caml_release_runtime_system();
  /* Resolve the name */
```
h = gethostbyname(name);
/* Free the copy of the string, which we might as well do before
acquiring the runtime system to benefit from parallelism. */
caml_stat_free(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.

19.13 Advanced topic: interfacing with Windows Unicode APIs

This section contains some general guidelines for writing C stubs that use Windows Unicode APIs.
Note: This is an experimental feature of OCaml: the set of APIs below, as well as their exact
semantics are not final and subject to change in future releases.

The OCaml system under Windows can be configured at build time in one of two modes:

• legacy mode: All path names, environment variables, command line arguments, etc. on the
  OCaml side are assumed to be encoded using the current 8-bit code page of the system.
• Unicode mode: All path names, environment variables, command line arguments, etc. on
  the OCaml side are assumed to be encoded using UTF-8.
In what follows, we say that a string has the *OCaml encoding* if it is encoded in UTF-8 when in Unicode mode, in the current code page in legacy mode, or is an arbitrary string under Unix. A string has the *platform encoding* if it is encoded in UTF-16 under Windows or is an arbitrary string under Unix.

From the point of view of the writer of C stubs, the challenges of interacting with Windows Unicode APIs are twofold:

- The Windows API uses the UTF-16 encoding to support Unicode. The runtime system performs the necessary conversions so that the OCaml programmer only needs to deal with the OCaml encoding. C stubs that call Windows Unicode APIs need to use specific runtime functions to perform the necessary conversions in a compatible way.

- When writing stubs that need to be compiled under both Windows and Unix, the stubs need to be written in a way that allow the necessary conversions under Windows but that also work under Unix, where typically nothing particular needs to be done to support Unicode.

The native C character type under Windows is `WCHAR`, two bytes wide, while under Unix it is `char`, one byte wide. A type `char_os` is defined in `<caml/misc.h>` that stands for the concrete C character type of each platform. Strings in the platform encoding are of type `char_os *`.

The following functions are exposed to help write compatible C stubs. To use them, you need to include both `<caml/misc.h>` and `<caml/osdeps.h>`.

- `char_os* caml_stat_strdup_to_os(const char *)` copies the argument while translating from OCaml encoding to the platform encoding. This function is typically used to convert the `char *` underlying an OCaml string before passing it to an operating system API that takes a Unicode argument. Under Unix, it is equivalent to `caml_stat_strdup`. **Note:** For maximum backwards compatibility in Unicode mode, if the argument is not a valid UTF-8 string, this function will fall back to assuming that it is encoded in the current code page.

- `char* caml_stat_strdup_of_os(const char_os *)` copies the argument while translating from the platform encoding to the OCaml encoding. It is the inverse of `caml_stat_strdup_to_os`. This function is typically used to convert a string obtained from the operating system before passing it on to OCaml code. Under Unix, it is equivalent to `caml_stat_strdup`. **Note:** The strings returned by `caml_stat_strdup_to_os` and `caml_stat_strdup_of_os` are allocated using `caml_stat_alloc`, so they need to be deallocated using `caml_stat_free` when they are no longer needed.

- `value caml_copy_string_of_os(char_os *)` allocates an OCaml string with contents equal to the argument string converted to the OCaml encoding. This function is essentially equivalent to `caml_stat_strdup_of_os` followed by `caml_copy_string`, except that it avoids the allocation of the intermediate string returned by `caml_stat_strdup_of_os`. Under Unix, it is equivalent to `caml_copy_string`. **Note:** The strings returned by `caml_stat_strdup_to_os` and `caml_stat_strdup_of_os` are allocated using `caml_stat_alloc`, so they need to be deallocated using `caml_stat_free` when they are no longer needed.
Example  We want to bind the function `getenv` in a way that works both under Unix and Windows. Under Unix this function has the prototype:

```c
char *getenv(const char *);
```

While the Unicode version under Windows has the prototype:

```c
WCHAR *_wgetenv(const WCHAR *);
```

In terms of `char_os`, both functions take an argument of type `char_os *` and return a result of the same type. We begin by choosing the right implementation of the function to bind:

```c
#ifdef _WIN32
#define getenv_os _wgetenv
#else
#define getenv_os getenv
#endif
```

The rest of the binding is the same for both platforms:

```c
/* The following define is necessary because the API is experimental */
#define CAML_INTERNALS

#include <caml/mlvalues.h>
#include <caml/misc.h>
#include <caml/alloc.h>
#include <caml/fail.h>
#include <caml/osdeps.h>
#include <stdlib.h>

CAMLprim value stub_getenv(value var_name)
{
  CAMLparam1(var_name);
  CAMLlocal1(var_value);
  char_os *var_name_os, *var_value_os;

  var_name_os = caml_stat_strdup_to_os(String_val(var_name));
  var_value_os = getenv_os(var_name_os);
  caml_stat_free(var_name_os);

  if (var_value_os == NULL)
    caml_raise_not_found();

  var_value = caml_copy_string_of_os(var_value_os);

  CAMLreturn(var_value);
}
```
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, -Wl,-rpath, -Wl,-R
```

These options are passed as is to the C compiler. Refer to the documentation of the C compiler.

```
-custom
```

Force the construction of a statically linked library only, even if dynamic linking is supported.

```
-failsafe
```

Fall back to building a statically linked library if a problem occurs while building the shared library (e.g. some of the support libraries are not available as shared libraries).

```
-Ldir
```

Add `dir` to the search path for support libraries (-l<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.

-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 -dllib -lzip \
    -cclib -lzip -cclib -lz -ccopt -L/usr/local/zlib
occamlopt -a -o zip.cmx zip.cmx -cclib -lzip \
    -cclib -lzip -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
occamlopt -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

\texttt{ocamlmklib -o zip zip.cmx -lz -L/usr/local/zlib}

builds the native-code library zip.cmxa, and

\texttt{ocamlmklib -o zip zipstubs.o -lz -L/usr/local/zlib}

builds the C libraries \texttt{dllzip.so} and \texttt{libzip.a}. Notice that the support libraries (-lz) and the corresponding options (-L/usr/local/zlib) must be given on all three invocations of \texttt{ocamlmklib}, because they are needed at different times depending on whether shared libraries are supported.
Chapter 20

Optimisation with Flambda

20.1 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 20.14) and changes in behaviour of code using unsafe operations (see section 20.15).

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.

20.2 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:
-02 Perform more optimisation than usual. Compilation times may be lengthened. (This flag is an abbreviation for a certain set of parameters described in section 20.5.)

-03 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 -02) 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 20.10.3.

-unbox-closures
Pass free variables via specialised arguments rather than closures (an optimisation for reducing allocation). See section 20.9.3. 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 20.3.6.) 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 20.3.1.

-inline-toplevel
The equivalent of -inline but used when speculative inlining starts at toplevel. See section 20.3.6. 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 20.3.5.
Chapter 20. 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 20.3.5.

- **-inline-indirect-cost**, **-inline-prim-cost**
  Likewise.

- **-inline-lifting-benefit**
  Controls inlining of functors at toplevel. See section 20.3.5.

- **-inline-max-depth**
  The maximum depth of any speculative inlining search. See section 20.3.6.

- **-inline-max-unroll**
  The maximum depth of any unrolling of recursive functions during any speculative inlining search. See section 20.3.6.

- **-no-unbox-free-vars-of-closures**
  Do not unbox closure variables. See section 20.9.1.

- **-no-unbox-specialised-args**
  Do not unbox arguments to which functions have been specialised. See section 20.9.2.

- **-rounds**
  How many rounds of optimisation to perform. See section 20.2.1.

- **-unbox-closures-factor**
  Scaling factor for benefit calculation when using **-unbox-closures**. See section 20.9.3.

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.
20.2.1 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 \(-\text{rounds}\) parameter (although this is not necessary when using predefined optimisation levels such as with \(-\text{O2}\) and \(-\text{O3}\)). 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 \(-\text{cost}\) in the name, accept arguments of the form:

\[ n \ | \ round^{=n},... \]

- 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 \(-\text{Oclassic}\), \(-\text{O2}\) and \(-\text{O3}\) 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.

20.3 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:

```ocaml
let rec fact x =
  if x = 0 then
    1
  else
    x * fact (x - 1)

let n = fact 4
```
unrolling once at the call site `fact 4` produces (with the body of `fact` unchanged):

```haskell
let n = 
  if 4 = 0 then 
    1 
  else 
    4 * fact (4 - 1) 
```

This simplifies to:

```haskell
let n = 4 * 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.

### 20.3.1 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 20.3.3, 20.8.1 and 20.10. 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.

20.3.2 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 \( \text{false} \), 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 \(-\text{inline} 10000\).

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.)
Chapter 20. Optimisation with Flambda

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.)

20.3.3 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.

20.3.4 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.

20.3.5 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 f would remove:

- one direct call;
- one conditional branch.

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-\ldots-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{20.2.1}.

\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{20.5} 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.
20.3.6 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):

- \(-\text{inline}\) (or, if the call site that triggered speculation is at toplevel, \(-\text{inline-toplevel}\));
- \(-\text{inline-max-depth}\);
- \(-\text{inline-max-unroll}\).

Note in particular that \(-\text{inline}\) does not have the meaning that it has in the previous compiler or in \(-O\text{classic}\) mode. In both of those situations \(-\text{inline}\) 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 \(-\text{inline}\) (or \(-\text{inline-toplevel}\) 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 \(-\text{inline-max-depth}\) 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 \(-\text{inline-max-unroll}\) then speculation stops.

20.4 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 \textit{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 \textit{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 \textit{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 (Int.to_string x)

let run xs =  
  iter print_int (List.rev xs)

The argument `f` to `iter` is invariant so the function may be specialised:

```
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 (Int.to_string 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 (Int.to_string 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
20.4.1 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.

20.5 Default settings of parameters

The default settings (when not using -Oclassic) 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>

20.5.1 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>
20.5.2 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-call-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>

20.6 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

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)

20.7 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:

let f x y =
  ...
let p = x, y in
  ...
  ... (fst p) ... (snd p) ...

...
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 \texttt{safe-string} mode, because it cannot yet be guaranteed that they are immutable throughout a given program.

### 20.8 Other code motion transformations

#### 20.8.1 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 .\texttt{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 \textit{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.

#### 20.8.2 Lifting of toplevel let bindings

Toplevel \texttt{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 \textit{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).

## 20.9 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.

### 20.9.1 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.

```ocaml
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:

```ocaml
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:

```ocaml
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.

20.9.2 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`.

```ocaml
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

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:

```ocaml
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

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' [1; 2; 3] (x + 42) (x + 43))

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.)

### 20.9.3 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.

```ml
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 direct call surrogate, since it takes the place of another function—the unchanged version used for indirect calls—at direct call sites.)

The -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.

**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.

```ml
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))
```

The code resulting from applying -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 printf).
20.10 Removal of unused code and values

20.10.1 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.

20.10.2 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.

20.10.3 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.)

20.10.4 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.)

20.11 Other code transformations

20.11.1 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.

20.11.2 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).
20.12 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.
20.13 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.

20.14 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 Sys.opaque_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.

20.15 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 Sys.opaque_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 Sys.opaque_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 = Sys.opaque_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.

### 20.16 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 \( \text{let } x = e \text{ in } e' \).
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 20.4.)

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.

20.17 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 func-
tion, 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/.

val enabled : bool

enabled is true if the compiler is configured with spacetime and false otherwise

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

dep "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 Spacetime.Snapshot.take[20.17].

val save_and_close : ?time:float -> t -> unit

save_and_close 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
Spacetime.Snapshot.take[20.17].
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
\texttt{time} is specified, it will be used instead of the result of \texttt{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 \texttt{Spacetime.Series.save_event}, but writes to the automatic snapshot file. This
function is a no-op if \texttt{OCAML_SPACETIME_INTERVAL} was not set.
Chapter 21

Fuzzing with afl-fuzz

21.1 Overview

American fuzzy lop ("afl-fuzz") is a fuzzer, a tool for testing software by providing randomly-generated inputs, searching for those inputs which cause the program to crash.

Unlike most fuzzers, afl-fuzz observes the internal behaviour of the program being tested, and adjusts the test cases it generates to trigger unexplored execution paths. As a result, test cases generated by afl-fuzz cover more of the possible behaviours of the tested program than other fuzzers.

This requires that programs to be tested are instrumented to communicate with afl-fuzz. The native-code compiler "ocamlopt" can generate such instrumentation, allowing afl-fuzz to be used against programs written in OCaml.

For more information on afl-fuzz, see the website at http://lcamtuf.coredump.cx/afl/

21.2 Generating instrumentation

The instrumentation that afl-fuzz requires is not generated by default, and must be explicitly enabled, by passing the -afl-instrument option to ocamlopt.

To fuzz a large system without modifying build tools, OCaml’s configure script also accepts the afl-instrument option. If OCaml is configured with afl-instrument, then all programs compiled by ocamlopt will be instrumented.

21.2.1 Advanced options

In rare cases, it is useful to control the amount of instrumentation generated. By passing the -afl-inst-ratio N argument to ocamlopt with N less than 100, instrumentation can be generated for only N% of branches. (See the afl-fuzz documentation on the parameter AFL INST_RATIO for the precise effect of this).

21.3 Example

As an example, we fuzz-test the following program, readline.ml:
let _ =
  let s = read_line () in
  match Array.to_list (Array.init (String.length s) (String.get s)) with
  | ['s'; 'e'; 'c'; 'r'; 'e'; 't'; ' '; 'c'; 'o'; 'd'; 'e'] -> failwith "uh oh"
  | _ -> ()

There is a single input (the string “secret code”) which causes this program to crash, but finding it by blind random search is infeasible.

Instead, we compile with afl-fuzz instrumentation enabled:

    ocamlopt -afl-instrument readline.ml -o readline

    Next, we run the program under afl-fuzz:

    mkdir input
    echo asdf > input/testcase
    mkdir output
    afl-fuzz -i input -o output ./readline

    By inspecting instrumentation output, the fuzzer finds the crashing input quickly.
Chapter 22

Compiler plugins

22.1 Overview

Starting from OCaml 4.03, it is possible to extend the native and bytecode compilers with plugins using the -plugin command line option of both tools. This possibility is also available for ocamldep for OCaml version ulterior to 4.05. Beware however that plugins are an advanced feature of which the design is still in flux and breaking changes may happen in the future. Plugins features are based on the compiler library API. In complement, new hooks have been added to the compiler to increase its flexibility.

In particular, hooks are available in the Pparse module (see section 24.8) to transform the parsed abstract syntax tree, providing similar functionality to extension point based preprocessors. Other hooks are available to analyze the typed tree in the Typemod module (see section 24.9) after the type-checking phase of the compiler. Since the typed tree relies on numerous invariants that play a vital part in ulterior phases of the compiler, it is not possible however to transform the typed tree. Similarly, the intermediary lambda representation can be modified by using the hooks provided in the Simplif module (see section 24.10). A plugin can also add new options to a tool through the Clflags.add_arguments function (see Clflags module (see section 24.11)).

Plugins are dynamically loaded and need to be compiled in the same mode (i.e. native or bytecode) that the tool they extend.

22.2 Basic example

As an illustration, we shall build a simple Hello world plugin that adds a simple statement print_endline "Hello from:$sourcefile" to a compiled file.

The simplest way to implement this feature is to modify the abstract syntax tree. We will therefore add an hooks to the Pparse.ImplementationHooks. Since the proposed modification is very basic, we could implement the hook directly. However, for the sake of this illustration, we use the Ast_mapper structure that provides a better path to build more interesting plugins.

The first step is to build the AST fragment corresponding to the evaluation of print_endline:

```ocaml
let print_endline name =
  let open Ast_helper in
  let print_endline = Exp.ident
```

387
Then, we can construct an AST mapper that adds this fragment to the parsed AST tree.

```ocaml
let add_hello name (mapper:Ast_mapper.mapper) structure =
  let default = Ast_mapper.default_mapper in
  (print_endline name) :: (default.structure default structure)

let ast_mapper name =
  { Ast_mapper.default_mapper with structure = add_hello name }

Once this AST mapper is constructed, we need to convert it to a hook and adds this hook to
the Pparse.ImplementationsHooks.

let transform hook_info structure =
  let astm = ast_mapper hook_info.Misc.sourcefile in
  astm.structure astm structure

let () = Pparse.ImplementationHooks.add_hook "Hello world hook" transform

The resulting simplistic plugin can then be compiled with

$ ocamlopt -I +compiler-libs -shared plugin.ml -o plugin.cmxs

Compiling other files with this plugin enabled is then as simple as

$ ocamlopt -plugin plugin.cmxs test.ml -o test
Part IV

The OCaml library
Chapter 23

The core library

This chapter describes the OCaml core library, which is composed of declarations for built-in types and exceptions, plus the module `Pervasives` that provides basic operations on these built-in types. The `Pervasives` module is special in two ways:

- It is automatically linked with the user’s object code files by the `ocamlc` command (chapter 9).
- It is automatically “opened” when a compilation starts, or when the toplevel system is launched. Hence, it is possible to use unqualified identifiers to refer to the functions provided by the `Pervasives` module, without adding a `open Pervasives` directive.

Conventions

The declarations of the built-in types and the components of module `Pervasives` are printed one by one in typewriter font, followed by a short comment. All library modules and the components they provide are indexed at the end of this report.

23.1 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**

```ocaml
type int
    The type of integer numbers.

type char
    The type of characters.

type bytes
    The type of (writable) byte sequences.
```
type string
   The type of (read-only) character strings.

type float
   The type of floating-point numbers.

type bool = false | true
   The type of booleans (truth values).

type unit = ()
   The type of the unit value.

type exn
   The type of exception values.

type 'a array
   The type of arrays whose elements have type 'a.

type 'a list = [] | :: of 'a * 'a list
   The type of lists whose elements have type 'a.

type 'a option = None | Some of 'a
   The type of optional values of type 'a.

type int32
   The type of signed 32-bit integers. Literals for 32-bit integers are suffixed by l. See the
   Int32[?]

   int32 module.

type int64
   The type of signed 64-bit integers. Literals for 64-bit integers are suffixed by L. See the
   Int64[?]

   int64 module.

type nativeint
   The type of signed, platform-native integers (32 bits on 32-bit processors, 64 bits on 64-bit
   processors). Literals for native integers are suffixed by n. See the Nativeint[?]

   nativeint module.

type ('a, 'b, 'c, 'd, 'e, 'f) format6
   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.

type 'a lazy_t
   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 12.5.)

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 8.2.) The arguments are the location of the definition in the source code (file name, line number, column number).

23.2 Module Stdlib: The OCaml Standard library.

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 Stdlib.

It particular, it provides the basic operations over the built-in types (numbers, booleans, byte sequences, strings, exceptions, references, lists, arrays, input-output channels, ...).

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.

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.)

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. The arguments are the location of the definition in the source code (file name, line number, column number).
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. Left-associative operator, see Ocaml_operators[??] for more information.

val (<>) : 'a -> 'a -> bool

Negation of (=)[23.3]. Left-associative operator, see Ocaml_operators[??] for more information.

val (<) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators[??] for more information.

val (>) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators[??] for more information.

val (<=) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators[??] for more information.

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. Left-associative operator, see Ocaml_operators[??] for more information.

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 nan[23.3]. 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[??] and Map.Make[??] functors, as well as the List.sort[??] and Array.sort[??] 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 \texttt{nan}.

\begin{verbatim}
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 \texttt{nan}.
\end{verbatim}

\begin{verbatim}
val (==) : 'a -> 'a -> bool
  \texttt{e1 == e2} tests for physical equality of \texttt{e1} and \texttt{e2}. On mutable types such as references, arrays, byte sequences, records with mutable fields and objects with mutable instance variables, \texttt{e1 == e2} is true if and only if physical modification of \texttt{e1} also affects \texttt{e2}. On non-mutable types, the behavior of \texttt{(==)} is implementation-dependent; however, it is guaranteed that \texttt{e1 == e2} implies \texttt{compare e1 e2 = 0}. Left-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}

\begin{verbatim}
val (!=) : 'a -> 'a -> bool
  Negation of \texttt{(==)}[23.3]. Left-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}

\subsection*{Boolean operations}

\begin{verbatim}
val not : bool -> bool
  The boolean negation.
\end{verbatim}

\begin{verbatim}
val (&&) : bool -> bool -> bool
  The boolean 'and'. Evaluation is sequential, left-to-right: in \texttt{e1 && e2}, \texttt{e1} is evaluated first, and if it returns \texttt{false}, \texttt{e2} is not evaluated at all. Right-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}

\begin{verbatim}
val (&) : bool -> bool -> bool
  Deprecated. (&&)[23.3] should be used instead. Right-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}

\begin{verbatim}
val (||) : bool -> bool -> bool
  The boolean 'or'. Evaluation is sequential, left-to-right: in \texttt{e1 || e2}, \texttt{e1} is evaluated first, and if it returns \texttt{true}, \texttt{e2} is not evaluated at all. Right-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}

\begin{verbatim}
val (or) : bool -> bool -> bool
  Deprecated. (||)[23.3] should be used instead. Right-associative operator, see \texttt{Ocaml_operators} for more information.
\end{verbatim}
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_OF__ 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)).
Left-associative operator, see 0caml_operators for more information.
Since: 4.01

val (@@) : ('a -> 'b) -> 'a -> 'b
Application operator: g @@ f @@ x is exactly equivalent to g (f (x)). Right-associative operator, see 0caml_operators for more information.
Since: 4.01

Integer arithmetic

Integers are Sys.int_size bits wide. All operations are taken modulo 2^{Sys.int_size}. They do not fail on overflow.

val (~-) : int -> int
Unary negation. You can also write - e instead of ~- e. Unary operator, see 0caml_operators for more information.

val (~+) : int -> int
Unary addition. You can also write + e instead of ~+ e. Unary operator, see 0caml_operators for more information.
Since: 3.12.0

val succ : int -> int
succ x is x + 1.

val pred : int -> int
pred x is x - 1.

val (+) : int -> int -> int
Integer addition. Left-associative operator, see 0caml_operators for more information.

val (-) : int -> int -> int
Integer subtraction. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
val ( * ) : int -> int -> int
```

Integer multiplication. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
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, \( (-x) / y = x / (-y) = - (x / y) \). Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
val (mod) : int -> int -> int
```

Integer remainder. If \( y \) is not zero, the result of \( x \mod y \) satisfies the following properties: \( x = (x / y) \times y + x \mod y \) 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. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
val abs : int -> int
```

Return the absolute value of the argument. Note that this may be negative if the argument is `min_int`.

```ocaml
val max_int : int
```

The greatest representable integer.

```ocaml
val min_int : int
```

The smallest representable integer.

**Bitwise operations**

```ocaml
val (land) : int -> int -> int
```

Bitwise logical and. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
val (lor) : int -> int -> int
```

Bitwise logical or. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
val (lxor) : int -> int -> int
```

Bitwise logical exclusive or. Left-associative operator, see `Ocaml_operators` for more information.

```ocaml
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 > Sys.int_size. Right-associative operator, see Ocaml_operators[??] for more information.

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 > Sys.int_size. Right-associative operator, see Ocaml_operators[??] for more information.

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 > Sys.int_size. Right-associative operator, see Ocaml_operators[??] for more information.

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. Unary operator, see Ocaml_operators[??] for more information.

val (~+.) : float -> float
Unary addition. You can also write +. e instead of ~+. e. Unary operator, see Ocaml_operators[??] for more information.

Since: 3.12.0

val (+.) : float -> float -> float
Floating-point addition. Left-associative operator, see Ocaml_operators[??] for more information.

val (-.) : float -> float -> float
Floating-point subtraction. Left-associative operator, see Ocaml_operators[??] for more information.

val ( *. ) : float -> float -> float
Floating-point multiplication. Left-associative operator, see Ocaml_operators[??] for more information.
val (/.) : float -> float -> float
   Floating-point division. Left-associative operator, see \texttt{Ocaml_operators} for more information.

val ( ** ) : float -> float -> float
   Exponentiation. Right-associative operator, see \texttt{Ocaml_operators} for more information.

val sqrt : float -> float
   Square root.

val exp : float -> float
   Exponential.

val log : float -> float
   Natural logarithm.

val log10 : float -> float
   Base 10 logarithm.

val expm1 : float -> float
   \texttt{expm1} \texttt{x} computes \texttt{exp x} -. 1.0, giving numerically-accurate results even if \texttt{x} is close to 0.0.
   \textbf{Since}: 3.12.0

val log1p : float -> float
   \texttt{log1p} \texttt{x} computes \texttt{log(1.0 +. x)} (natural logarithm), giving numerically-accurate results even if \texttt{x} is close to 0.0.
   \textbf{Since}: 3.12.0

val cos : float -> float
   Cosine. Argument is in radians.

val sin : float -> float
   Sine. Argument is in radians.

val tan : float -> float
   Tangent. Argument is in radians.

val acos : float -> float
   Arc cosine. The argument must fall within the range \([-1.0, 1.0]\). Result is in radians and is between 0.0 and pi.

val asin : float -> float
Arc sine. The argument must fall within the range $[-1.0, 1.0]$. Result is in radians and is between $-\pi/2$ and $\pi/2$.

```ocaml
definition val atan : float -> float
  Arc tangent. Result is in radians and is between $-\pi/2$ and $\pi/2$.

definition val atan2 : float -> float -> float
  atan2 y x returns the arc tangent of $y / x$. The signs of $x$ and $y$ are used to determine the quadrant of the result. Result is in radians and is between $-\pi$ and $\pi$.

definition val hypot : float -> float -> 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 infinity even if the other is nan.
  Since: 4.00.0

definition val cosh : float -> float
  Hyperbolic cosine. Argument is in radians.

definition val sinh : float -> float
  Hyperbolic sine. Argument is in radians.

definition val tanh : float -> float
  Hyperbolic tangent. Argument is in radians.

definition val ceil : float -> float
  Round above to an integer value. ceil f returns the least integer value greater than or equal to $f$. The result is returned as a float.

definition val floor : float -> float
  Round below to an integer value. floor f returns the greatest integer value less than or equal to $f$. The result is returned as a float.

definition val abs_float : float -> float
  abs_float f returns the absolute value of $f$.

definition val copysign : float -> float -> float
  copysign x y returns a float whose absolute value is that of $x$ and whose sign is that of $y$. If $x$ is nan, returns nan. If $y$ is nan, returns either $x$ or $-x$, but it is not specified which.
  Since: 4.00.0

definition val mod_float : float -> float -> float
  mod_float a b returns the remainder of $a$ with respect to $b$. The returned value is $a \mod b$, 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 float_of_int[23.3].

val float_of_int : int -> float
Convert an integer to floating-point.

val truncate : float -> int
Same as int_of_float[23.3].

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
The smallest positive, non-zero, non-denormalized value of type float.

val epsilon_float : float
The difference between 1.0 and the smallest exactly representable floating-point number 
greater than 1.0.
Chapter 23. The core library

405

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 classify_float function.

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.

val (^) : string -> string -> string
  String concatenation. Right-associative operator, see Ocaml_operators for more information.

Character operations

More character operations are provided in module Char.

val int_of_char : char -> int
  Return the ASCII code of the argument.

val char_of_int : int -> char
  Return the character with the given ASCII code. Raise Invalid_argument "char_of_int" if the argument is outside the range 0–255.

Unit operations

val ignore : 'a -> unit
  Discard the value of its argument and return (). For instance, ignore(f x) discards the result of the side-effecting function f. It is equivalent to f x; (), except that 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_opt : string -> bool option
    Convert the given string to a boolean.
    Return None if the string is not "true" or "false".
    Since: 4.05

val bool_of_string : string -> bool
    Same as bool_of_string_opt[23.3], but raise Invalid_argument "bool_of_string" instead of returning None.

val string_of_int : int -> string
    Return the string representation of an integer, in decimal.

val int_of_string_opt : string -> int option
    Convert the given string to an integer. The string is read in decimal (by default, or if the string begins with 0u), 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 0u prefix reads the input as an unsigned integer in the range [0, 2*max_int+1]. If the input exceeds max_int[23.3] it is converted to the signed integer min_int + input - max_int - 1.
    The _ (underscore) character can appear anywhere in the string and is ignored.
    Return None 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.
    Since: 4.05

val int_of_string : string -> int
    Same as int_of_string_opt[23.3], but raise Failure "int_of_string" instead of returning None.

val string_of_float : float -> string
    Return the string representation of a floating-point number.

val float_of_string_opt : string -> float option
    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 an 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.

Return None if the given string is not a valid representation of a float.

Since: 4.05

val float_of_string : string \rightarrow\ float

Same as float_of_string_opt[23.3], but raise Failure "float_of_string" instead of returning None.

Pair operations

val fst : 'a * 'b \rightarrow\ 'a

Return the first component of a pair.

val snd : 'a * 'b \rightarrow\ 'b

Return the second component of a pair.

List operations

More list operations are provided in module List[??].

val (@) : 'a list \rightarrow\ 'a list \rightarrow\ 'a list

List concatenation. Not tail-recursive (length of the first argument). Right-associative operator, see Ocaml_operators[??] for more information.

Input/output

Note: all input/output functions can raise Sys_error when the system calls they invoke fail.

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.

cval prerr_float : float -> unit
   Print a floating-point number, in decimal, on standard error.

cval prerr_endline : string -> unit
   Print a string, followed by a newline character on standard error and flush standard error.

cval 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.

cval read_int_opt : unit -> int option
   Flush standard output, then read one line from standard input and convert it to an integer.
   Return None if the line read is not a valid representation of an integer.
   Since: 4.05

val read_int : unit -> int
   Same as read_int_opt[23.3], but raise Failure "int_of_string" instead of returning None.

val read_float_opt : unit -> float option
   Flush standard output, then read one line from standard input and convert it to a floating-point number.
   Return None if the line read is not a valid representation of a floating-point number.
   Since: 4.05.0

val read_float : unit -> float
   Same as read_float_opt[23.3], but raise Failure "float_of_string" instead of returning None.
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 open_out_gen[23.3] and open_in_gen[23.3].

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 open_out[23.3], but the file is opened in binary mode, so that no translation takes
  place during writes. On operating systems that do not distinguish between text mode and
  binary mode, this function behaves like open_out[23.3].

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 specifies the opening mode. The extra argument perm specifies
  the file permissions, in case the file must be created. open_out[23.3] and
  open_out_bin[23.3] are special cases of this function.

val flush : out_channel -> unit
Flush the buffer associated with the given output channel, performing all pending writes on that channel. Interactive programs must be careful about flushing standard output and standard error at the right time.

val flush_all : unit -> unit
Flush all open output channels; ignore errors.

val output_char : out_channel -> char -> unit
Write the character on the given output channel.

val output_string : out_channel -> string -> unit
Write the string on the given output channel.

val output_bytes : out_channel -> bytes -> unit
Write the byte sequence on the given output channel.
Since: 4.02.0

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.

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

val output_byte : out_channel -> int -> unit
Write one 8-bit integer (as the single character with that code) on the given output channel. The given integer is taken modulo 256.

val output_binary_int : out_channel -> int -> unit
Write one integer in binary format (4 bytes, big-endian) on the given output channel. The given integer is taken modulo 2^{32}. The only reliable way to read it back is through the \input_binary_int function. The format is compatible across all machines for a given version of OCaml.

val output_value : out_channel -> 'a -> unit
Write the representation of a structured value of any type to a channel. Circularities and sharing inside the value are detected and preserved. The object can be read back, by the function \input_value. See the description of module Marshal for more information. output_value is equivalent to Marshal.to_channel with an empty list of flags.

val seek_out : out_channel -> int -> unit
seek_out chan pos sets the current writing position to pos for channel chan. This works only for regular files. On files of other kinds (such as terminals, pipes and sockets), the behavior is unspecified.

val pos_out : out_channel -> int
   Return the current writing position for the given channel. Does not work on channels opened with the Open_append flag (returns unspecified results).

val out_channel_length : out_channel -> int
   Return the size (number of characters) of the regular file on which the given channel is opened. If the channel is opened on a file that is not a regular file, the result is meaningless.

val close_out : out_channel -> unit
   Close the given channel, flushing all buffered write operations. Output functions raise a Sys_error exception when they are applied to a closed output channel, except close_out and flush, which do nothing when applied to an already closed channel. Note that close_out may raise Sys_error if the operating system signals an error when flushing or closing.

val close_out_noerr : out_channel -> unit
   Same as close_out, but ignore all errors.

val set_binary_mode_out : out_channel -> bool -> unit
   set_binary_mode_out oc true sets the channel oc to binary mode: no translations take place during output. set_binary_mode_out oc false sets the channel oc to text mode: depending on the operating system, some translations may take place during output. For instance, under Windows, end-of-lines will be translated from \n to \r\n. This function has no effect under operating systems that do not distinguish between text mode and binary mode.

General input functions

val open_in : string -> in_channel
   Open the named file for reading, and return a new input channel on that file, positioned at the beginning of the file.

val open_in_bin : string -> in_channel
   Same as open_in[23.3], but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like open_in[23.3].

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. open_in[23.3] and open_in_bin[23.3] are special cases of this function.
Chapter 23. The core library

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 really_input[23.3] 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 input_char[23.3], 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 output_binary_int[23.3]. Raise End_of_file if an end of file was reached
  while reading the integer.

val input_value : in_channel -> 'a
  Read the representation of a structured value, as produced by output_value[23.3], and
  return the corresponding value. This function is identical to Marshal.from_channel[??];
  see the description of module Marshal[??] for more information, in particular concerning
  the lack of type safety.
val seek_in : in_channel -> int -> unit
         seek_in chan pos sets the current reading position to pos for channel chan. This works
         only for regular files. On files of other kinds, the behavior is unspecified.

val pos_in : in_channel -> int
         Return the current reading position for the given channel.

val in_channel_length : in_channel -> int
         Return the 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 : out_channel -> int64 -> unit
         val pos_out : out_channel -> int64
         val out_channel_length : out_channel -> int64
         val seek_in : in_channel -> int64 -> unit
         val pos_in : in_channel -> int64
         val in_channel_length : 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. Unary
operator, see Ocaml_operators for more information.

val (:=) : 'a ref -> 'a -> unit
r := a stores the value of a in reference r. Equivalent to fun r v -> r.contents <- v.
Right-associative operator, see Ocaml_operators for more information.

val incr : int ref -> unit
Increment the integer contained in the given reference. Equivalent to fun r -> r := succ
!r.

val decr : int ref -> unit
Decrement the integer contained in the given reference. Equivalent to fun r -> r := pred
!r.

Result type

type ('a, 'b) result =
| Ok of 'a
| Error of 'b

Since: 4.03.0

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 are made of three kinds of entities:

- **conversions specifications**, introduced by the special character '%', followed by one or more
  characters specifying what kind of argument to read or print,
- *formatting indications*, introduced by the special character '@' followed by one or more characters specifying how to read or print the argument,

- *plain characters* that are regular characters with usual lexical conventions. Plain characters specify string literals to be read in the input or printed in the output.

There is an additional lexical rule to escape the special characters '%', '@' in format strings: if a special character follows a '%' character, it is treated as a plain character. In other words, "%%" is considered as a plain '%' and "%@" as a plain '@'.

For more information about conversion specifications and formatting indications available, read the documentation of modules `Printf`, `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.

```
type ('a, 'b, 'c, 'd, 'e, 'f) format6 = ('a, 'b, 'c, 'd, 'e, 'f) CamlinternalFormatBasics.format6
type ('a, 'b, 'c, 'd) format4 = ('a, 'b, 'c, 'c, 'd) format6
type ('a, 'b, 'c) format = ('a, 'b, 'c, 'c) format4
val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string
  Converts a format string into a string.
```
Chapter 23. The core library

val format_of_string : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> ('a, 'b, 'c, 'd, 'e, 'f) format6

format_of_string s returns a format string read from the string literal s. Note: format_of_string can not convert a string argument that is not a literal. If you need this functionality, use the more general Scanf.format_from_string function.

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. Right-associative operator, see Ocaml_operators for more information.

Program termination

val exit : int -> 'a

Terminate the process, returning the given status code to the operating system: usually 0 to indicate no errors, and a small positive integer to indicate failure. All open output channels are flushed with flush_all. An implicit exit 0 is performed each time a program terminates normally. An implicit exit 2 is performed if the program terminates early because of an uncaught exception.

val at_exit : (unit -> unit) -> unit

Register the given function to be called at program termination time. The functions registered with at_exit will be called when the program does any of the following:

- executes exit[23.3]
- terminates, either normally or because of an uncaught exception
- executes the C function caml_shutdown. The functions are called in 'last in, first out’ order: the function most recently added with at_exit is called first.

Standard library modules

module Arg :
  Arg

module Array :
  Array

module ArrayLabels :
ArrayLabels
module Bigarray :
  Bigarray
module Bool :
  Bool
module Buffer :
  Buffer
module Bytes :
  Bytes
module BytesLabels :
  BytesLabels
module Callback :
  Callback
module Char :
  Char
module Complex :
  Complex
module Digest :
  Digest
module Ephemeron :
  Ephemeron
module Filename :
  Filename
module Float :
  Float
module Format :
  Format
module Fun :
  Fun
module Gc :
  Gc
module Genlex :
  Genlex
module Hashtbl :
  Hashtbl
module Int :
  Int
module Int32 :
  Int32
module Int64 :
  Int64
Chapter 23. The core library

module Lazy :
    Lazy
module Lexing :
    Lexing
module List :
    List
module ListLabels :
    ListLabels
module Map :
    Map
module Marshal :
    Marshal
module MoreLabels :
    MoreLabels
module Nativeint :
    Nativeint
module Obj :
    Obj
module Oo :
    Oo
module Option :
    Option
module Parsing :
    Parsing
module Pervasives :
    Pervasives
module Printexc :
    Printexc
module Printf :
    Printf
module Queue :
    Queue
module Random :
    Random
module Result :
    Result
module Scanf :
    Scanf
module Seq :
    Seq
module Set :
Set
module Spacetime :
  Spacetime
module Stack :
  Stack
module StdLabels :
  StdLabels
module Stream :
  Stream
module String :
  String
module StringLabels :
  StringLabels
module Sys :
  Sys
module Uchar :
  Uchar
module Weak :
  Weak

23.3 Module Stdlib : The OCaml Standard library.

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 Stdlib.

It particular, it provides the basic operations over the built-in types (numbers, booleans, byte sequences, strings, exceptions, references, lists, arrays, input-output channels, ...).

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.

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.)

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. The arguments are the location of the definition in the source code (file name, line number, column number).

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.

Left-associative operator, see Ocaml_operators for more information.

val (<>) : 'a -> 'a -> bool

Negation of (=)[23.3]. Left-associative operator, see Ocaml_operators for more information.

val (<) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators for more information.

val (>) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators for more information.

val (<=) : 'a -> 'a -> bool

See (>=)[23.3]. Left-associative operator, see Ocaml_operators for more information.

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 data structures may not terminate.

Left-associative operator, see Ocaml_operators for more information.

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 \(=, <, \text{ and } >\) defined above, with one difference on the treatment of the float value nan\[23.3\]. 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[??] and Map.Make[??] functors, as well as the List.sort[??] and Array.sort[??] 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. Left-associative operator, see Ocaml_operators[??] for more information.

val (!=) : 'a -> 'a -> bool

Negation of \((==)\)[23.3]. Left-associative operator, see Ocaml_operators[??] for more information.

**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. Right-associative operator, see Ocaml_operators[??] for more information.

val (\&) : bool -> bool -> bool
Deprecated. ([&][23.3]) should be used instead. Right-associative operator, see Ocaml_operators[??] for more information.

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. Right-associative operator, see Ocaml_operators[??] for more information.

val (or) : bool -> bool -> bool
  Deprecated. ([&][23.3]) should be used instead. Right-associative operator, see Ocaml_operators[??] for more information.

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
Chapter 23. The core library

```ocaml
__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_OF__ 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

```ocaml
val (|>) : 'a -> ('a -> 'b) -> 'b
Reverse-application operator: x |> f |> g is exactly equivalent to g (f (x)). Left-associative operator, see Ocaml_operators for more information.
Since: 4.01
```

```ocaml
val (@@) : ('a -> 'b) -> 'a -> 'b
Application operator: g @@ f @@ x is exactly equivalent to g (f (x)). Right-associative operator, see Ocaml_operators for more information.
Since: 4.01
```

Integer arithmetic

Integers are Sys.int_size bits wide. All operations are taken modulo 2^Sys.int_size. They do not fail on overflow.

```ocaml
val (~-) : int -> int
Unary negation. You can also write - e instead of ~- e. Unary operator, see Ocaml_operators for more information.
```

```ocaml
val (~+) : int -> int
Unary addition. You can also write + e instead of ~+ e. Unary operator, see Ocaml_operators for more information.
Since: 3.12.0
```
val succ : int -> int
    succ x is x + 1.

val pred : int -> int
    pred x is x - 1.

val (+) : int -> int -> int
    Integer addition. Left-associative operator, see [Ocaml_operators](#) for more information.

val (-) : int -> int -> int
    Integer subtraction. Left-associative operator, see [Ocaml_operators](#) for more information.

val ( * ) : int -> int -> int
    Integer multiplication. Left-associative operator, see [Ocaml_operators](#) for more information.

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, \((-x) / y = x / (-y) = - (x / y)\). Left-associative operator, see [Ocaml_operators](#) for more information.

val (mod) : int -> int -> int
    Integer remainder. If \( y \) is not zero, the result of \( x \mod y \) satisfies the following properties: \( x = (x / y) \times y + x \mod y \) 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. Left-associative operator, see [Ocaml_operators](#) for more information.

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 23. The core library

Bitwise operations

val (land) : int -> int -> int

   Bitwise logical and. Left-associative operator, see `Ocaml_operators` for more information.

val (lor) : int -> int -> int

   Bitwise logical or. Left-associative operator, see `Ocaml_operators` for more information.

val (lxor) : int -> int -> int

   Bitwise logical exclusive or. Left-associative operator, see `Ocaml_operators` for more information.

val lnot : int -> int

   Bitwise logical negation.

val (lsl) : int -> int -> int

   \( n \text{lsl} m \) shifts \( n \) to the left by \( m \) bits. The result is unspecified if \( m < 0 \) or \( m > \) `Sys.int_size`. Right-associative operator, see `Ocaml_operators` for more information.

val (lsr) : int -> int -> int

   \( n \text{lsr} m \) 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 \( m > \) `Sys.int_size`. Right-associative operator, see `Ocaml_operators` for more information.

val (asr) : int -> int -> int

   \( n \text{asr} m \) 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 \( m > \) `Sys.int_size`. Right-associative operator, see `Ocaml_operators` for more information.

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 \( 1.0 \div 0.0 \), `neg_infinity` for \(-1.0 \div 0.0\), and `nan` (‘not a number’) for \( 0.0 \div 0.0 \). These special numbers then propagate through floating-point computations as expected: for instance, \( 1.0 \div \text{infinity} \) 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`. Unary operator, see `Ocaml_operators` for more information.

val (~+..) : float -> float
Unary addition. You can also write \( + \ e \) instead of \( ^+\ e \). Unary operator, see `Ocaml_operators` for more information.

**Since:** 3.12.0

```ocaml
val (+.) : float -> float -> float

Floating-point addition. Left-associative operator, see `Ocaml_operators` for more information.
```

```ocaml
val (-.) : float -> float -> float

Floating-point subtraction. Left-associative operator, see `Ocaml_operators` for more information.
```

```ocaml
val (*. ) : float -> float -> float

Floating-point multiplication. Left-associative operator, see `Ocaml_operators` for more information.
```

```ocaml
val (/.) : float -> float -> float

Floating-point division. Left-associative operator, see `Ocaml_operators` for more information.
```

```ocaml
val ( ** ) : float -> float -> float

Exponentiation. Right-associative operator, see `Ocaml_operators` for more information.
```

```ocaml
val sqrt : float -> float

Square root.
```

```ocaml
val exp : float -> float

Exponential.
```

```ocaml
val log : float -> float

Natural logarithm.
```

```ocaml
val log10 : float -> float

Base 10 logarithm.
```

```ocaml
val expm1 : float -> float

expm1 x computes \( \exp x - 1.0 \), giving numerically-accurate results even if \( x \) is close to 0.0.

**Since:** 3.12.0
```

```ocaml
val log1p : float -> float

log1p x computes \( \log(1.0 + x) \) (natural logarithm), giving numerically-accurate results even if \( x \) is close to 0.0.

**Since:** 3.12.0
```
val \texttt{cos} : \texttt{float} \to \texttt{float}
  Cosine. Argument is in radians.

val \texttt{sin} : \texttt{float} \to \texttt{float}
  Sine. Argument is in radians.

val \texttt{tan} : \texttt{float} \to \texttt{float}
  Tangent. Argument is in radians.

val \texttt{acos} : \texttt{float} \to \texttt{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 \texttt{asin} : \texttt{float} \to \texttt{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 \texttt{atan} : \texttt{float} \to \texttt{float}
  Arc tangent. Result is in radians and is between \(-\pi/2\) and \(\pi/2\).

val \texttt{atan2} : \texttt{float} \to \texttt{float} \to \texttt{float}
  \texttt{atan2} \(y / x\) returns the arc tangent of \(y / x\). The signs of \(x\) and \(y\) are used to determine the quadrant of the result. Result is in radians and is between \(-\pi\) and \(\pi\).

val \texttt{hypot} : \texttt{float} \to \texttt{float} \to \texttt{float}
  \texttt{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}.
  \textbf{Since:} 4.0.0

val \texttt{cosh} : \texttt{float} \to \texttt{float}
  Hyperbolic cosine. Argument is in radians.

val \texttt{sinh} : \texttt{float} \to \texttt{float}
  Hyperbolic sine. Argument is in radians.

val \texttt{tanh} : \texttt{float} \to \texttt{float}
  Hyperbolic tangent. Argument is in radians.

val \texttt{ceil} : \texttt{float} \to \texttt{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.
Round below to an integer value. floor f returns the greatest integer value less than or equal to f. The result is returned as a float.

val abs_float : float -> float
  abs_float f returns the absolute value of f.

val copysign : float -> float -> float
  copysign x y returns a float whose absolute value is that of x and whose sign is that of y. If x is nan, returns nan. If y is 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 float_of_int.[23.3].

val float_of_int : int -> float
  Convert an integer to floating-point.

val truncate : float -> int
  Same as int_of_float.[23.3].

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
The smallest positive, non-zero, non-denormalized value of type float.

val epsilon_float : float
The difference between 1.0 and the smallest exactly representable floating-point number greater than 1.0.

type fpclass =
| FP_normal
  Normal number, none of the below
| FP_subnormal
  Number very close to 0.0, has reduced precision
| FP_zero
  Number is 0.0 or -0.0
| FP_infinite
  Number is positive or negative infinity
| FP_nan
  Not a number: result of an undefined operation

The five classes of floating-point numbers, as determined by the classify_float function.

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.

val (^) : string -> string -> string
String concatenation. Right-associative operator, see Ocaml_operators for more information.
Character operations

More character operations are provided in module Char[??].

val int_of_char : char -> int
  Return the ASCII code of the argument.

val char_of_int : int -> char
  Return the character with the given ASCII code. Raise Invalid_argument "char_of_int"
  if the argument is outside the range 0–255.

Unit operations

val ignore : 'a -> unit
  Discard the value of its argument and return (). For instance, ignore(f x) discards the
  result of the side-effecting function f. It is equivalent to f x; (), except that 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_opt : string -> bool option
  Convert the given string to a boolean.
  Return None if the string is not "true" or "false".
  Since: 4.05

val bool_of_string : string -> bool
  Same as bool_of_string_opt[23.3], but raise Invalid_argument "bool_of_string"
  instead of returning None.

val string_of_int : int -> string
  Return the string representation of an integer, in decimal.

val int_of_string_opt : string -> int option
  Convert the given string to an integer. The string is read in decimal (by default, or if the
  string begins with 0u), 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 0u prefix reads the input as an unsigned integer in the range [0, 2*max_int+1]. If the
  input exceeds max_int[23.3] it is converted to the signed integer min_int + input -
  max_int - 1.
The _ (underscore) character can appear anywhere in the string and is ignored.
Return None 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.

Since: 4.05

val int_of_string : string -> int
    Same as int_of_string_opt[23.3], but raise Failure "int_of_string" instead of returning None.

val string_of_float : float -> string
    Return the string representation of a floating-point number.

val float_of_string_opt : string -> float option
    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 an 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.
    Return None if the given string is not a valid representation of a float.

Since: 4.05

val float_of_string : string -> float
    Same as float_of_string_opt[23.3], but raise Failure "float_of_string" instead of returning None.

Pair operations

val fst : 'a * 'b -> 'a
    Return the first component of a pair.

val snd : 'a * 'b -> 'b
    Return the second component of a pair.
List operations

More list operations are provided in module List[??].

val (@) : 'a list -> 'a list -> 'a list
    List concatenation. Not tail-recursive (length of the first argument). Right-associative
    operator, see Ocaml_operators[??] for more information.

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 \(\rightarrow\) unit*

Print a string, followed by a newline character, on standard output and flush standard output.

*val print_newline : unit \(\rightarrow\) 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 \(\rightarrow\) unit*

Print a character on standard error.

*val prerr_string : string \(\rightarrow\) unit*

Print a string on standard error.

*val prerr_bytes : bytes \(\rightarrow\) unit*

Print a byte sequence on standard error.

**Since:** 4.02.0

*val prerr_int : int \(\rightarrow\) unit*

Print an integer, in decimal, on standard error.

*val prerr_float : float \(\rightarrow\) unit*

Print a floating-point number, in decimal, on standard error.

*val prerr_endline : string \(\rightarrow\) unit*

Print a string, followed by a newline character on standard error and flush standard error.

*val prerr_newline : unit \(\rightarrow\) unit*

Print a newline character on standard error, and flush standard error.

**Input functions on standard input**

*val read_line : unit \(\rightarrow\) 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_opt : unit \(\rightarrow\) int option*

Flush standard output, then read one line from standard input and convert it to an integer. Return \texttt{None} if the line read is not a valid representation of an integer.

**Since:** 4.05
val read_int : unit -> int
  Same as read_int_opt[23.3], but raise Failure "int_of_string" instead of returning None.

val read_float_opt : unit -> float option
  Flush standard output, then read one line from standard input and convert it to a floating-point number.
  Return None if the line read is not a valid representation of a floating-point number.
  Since: 4.05.0

val read_float : unit -> float
  Same as read_float_opt[23.3], but raise Failure "float_of_string" instead of returning None.

**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 open_out_gen[23.3] and open_in_gen[23.3].

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.

```plaintext
val open_out_bin : string -> out_channel
```
Same as `open_out[23.3]`, but the file is opened in binary mode, so that no translation takes place during writes. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_out[23.3]`.

```plaintext
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` specifies the opening mode. The extra argument `perm` specifies the file permissions, in case the file must be created. `open_out[23.3]` and `open_out_bin[23.3]` are special cases of this function.

```plaintext
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.

```plaintext
val flush_all : unit -> unit
```
Flush all open output channels; ignore errors.

```plaintext
val output_char : out_channel -> char -> unit
```
Write the character on the given output channel.

```plaintext
val output_string : out_channel -> string -> unit
```
Write the string on the given output channel.

```plaintext
val output_bytes : out_channel -> bytes -> unit
```
Write the byte sequence on the given output channel.

```plaintext
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`.

```plaintext
val output_substring : out_channel -> string -> int -> int -> unit
```
Same as `output` but take a string as argument instead of a byte sequence.

```plaintext
val output_byte : out_channel -> int -> unit
```

Write one 8-bit integer (as the single character with that code) on the given output channel. The given integer is taken modulo 256.

val output_binary_int : out_channel -> int -> unit
Write one integer in binary format (4 bytes, big-endian) on the given output channel. The given integer is taken modulo \(2^{32}\). The only reliable way to read it back is through the function \input_binary_int\[23.3\]. The format is compatible across all machines for a given version of OCaml.

val output_value : out_channel -> 'a -> unit
Write the representation of a structured value of any type to a channel. Circularities and sharing inside the value are detected and preserved. The object can be read back, by the function \input_value\[23.3\]. See the description of module \texttt{Marshal}\[??\] for more information. \output_value[23.3] is equivalent to \texttt{Marshal.to_channel}[??] with an empty list of flags.

val seek_out : out_channel -> int -> unit
\seek_out\ chan pos sets the current writing position to pos for channel chan. This works only for regular files. On files of other kinds (such as terminals, pipes and sockets), the behavior is unspecified.

val pos_out : out_channel -> int
Return the current writing position for the given channel. Does not work on channels opened with the \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
\set_binary_mode_out\ oc true sets the channel oc to binary mode: no translations take place during output. \set_binary_mode_out\ oc false sets the channel oc to text mode: depending on the operating system, some translations may take place during output. For instance, under Windows, end-of-lines will be translated from \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 open_in[23.3], but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like open_in[23.3].

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. open_in[23.3] and open_in_bin[23.3] 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 really_input[23.3] 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 input_char[23.3], 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 output_binary_int[23.3]. Raise End_of_file if an end of file was reached while reading the integer.

val input_value : in_channel -> 'a

Read the representation of a structured value, as produced by output_value[23.3], and return the corresponding value. This function is identical to Marshal.from_channel[??]; see the description of module Marshal[??] for more information, in particular concerning the lack of type safety.

val seek_in : in_channel -> int -> unit

seek_in chan pos sets the current reading position to pos for channel chan. This works only for regular files. On files of other kinds, the behavior is unspecified.

val pos_in : in_channel -> int

Return the current reading position for the given channel.

val in_channel_length : in_channel -> int

Return the 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 : out_channel -> int64 -> unit
    val pos_out : out_channel -> int64
    val out_channel_length : out_channel -> int64
    val seek_in : in_channel -> int64 -> unit
    val pos_in : in_channel -> int64
    val in_channel_length : 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. Unary operator, see Ocaml_operators for more information.

val (:=) : 'a ref -> 'a -> unit

r := a stores the value of a in reference r. Equivalent to fun r v -> r.contents <- v. Right-associative operator, see Ocaml_operators for more information.

val incr : int ref -> unit
Increment the integer contained in the given reference. Equivalent to \texttt{fun } r \rightarrow r := \texttt{succ} \!r.

\begin{verbatim}
val decr : int ref \rightarrow unit
  Decrement the integer contained in the given reference. Equivalent to \texttt{fun } r \rightarrow r := \texttt{pred} \!r.
\end{verbatim}

\textbf{Result type}

\begin{verbatim}
type ('a, 'b) result =
  | Ok of 'a
  | Error of 'b
  Since: 4.03.0
\end{verbatim}

\textbf{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 \texttt{Scanf[??]} and to print data with formatted output functions from modules \texttt{Printf[??]} and \texttt{Format[??]}.

Format strings are made of three kinds of entities:

- \textit{conversions specifications}, introduced by the special character \texttt{%} followed by one or more characters specifying what kind of argument to read or print,

- \textit{formatting indications}, introduced by the special character \texttt{@} followed by one or more characters specifying how to read or print the argument,

- \textit{plain characters} that are regular characters with usual lexical conventions. Plain characters specify string literals to be read in the input or printed in the output.

There is an additional lexical rule to escape the special characters \texttt{%} and \texttt{@} in format strings: if a special character follows a \texttt{%} character, it is treated as a plain character. In other words, \texttt{%%} is considered as a plain \texttt{%} and \texttt{%@} as a plain \texttt{@}.

For more information about conversion specifications and formatting indications available, read the documentation of modules \texttt{Scanf[??]}, \texttt{Printf[??]} and \texttt{Format[??]}.

Format strings have a general and highly polymorphic type \texttt{('a, 'b, 'c, 'd, 'e, 'f) format6}. The two simplified types, \texttt{format} and \texttt{format4} below are included for backward compatibility with earlier releases of OCaml.

The meaning of format string type parameters is as follows:

- \texttt{'a} is the type of the parameters of the format for formatted output functions (\texttt{printf}-style functions); \texttt{'a} is the type of the values read by the format for formatted input functions (\texttt{scanf}-style functions).
Chapter 23. The core library

• '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.

type ('a, 'b, 'c, 'd, 'e, 'f) format6 = (‘a, ‘b, ‘c, ‘d, ‘e, ‘f) CamlinternalFormatBasics.format6
type ('a, 'b, 'c, 'd) format4 = (‘a, ‘b, ‘c, ‘c, ‘d) format6
type ('a, 'b, 'c) format = (‘a, ‘b, ‘c, ‘c) format4
val string_of_format : ('a, 'b, 'c, 'd, 'e, 'f) format6 -> string

Converts a format string into a string.

val format_of_string :
  ('a, 'b, 'c, 'd, 'e, 'f) format6 ->
  ('a, 'b, 'c, 'd, 'e, 'f) format6

  format_of_string s returns a format string read from the string literal s. Note:
  format_of_string can not convert a string argument that is not a literal. If you need this
  functionality, use the more general Scanf.format_from_string[] function.

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. Right-associative operator, see Ocaml_operators[] for
  more information.
Program termination

val exit : int -> 'a

Terminate the process, returning the given status code to the operating system: usually 0 to indicate no errors, and a small positive integer to indicate failure. All open output channels are flushed with flush_all. An implicit exit 0 is performed each time a program terminates normally. An implicit exit 2 is performed if the program terminates early because of an uncaught exception.

val at_exit : (unit -> unit) -> unit

Register the given function to be called at program termination time. The functions registered with at_exit will be called when the program does any of the following:

- executes exit[23.3]
- terminates, either normally or because of an uncaught exception
- executes the C function caml_shutdown. The functions are called in 'last in, first out' order: the function most recently added with at_exit is called first.

Standard library modules

module Arg :
    Arg
module Array :
    Array
module ArrayLabels :
    ArrayLabels
module Bigarray :
    Bigarray
module Bool :
    Bool
module Buffer :
    Buffer
module Bytes :
    Bytes
module BytesLabels :
    BytesLabels
module Callback :
    Callback
module Char :
    Char
module Complex :
    Complex
module Digest :
  Digest
module Ephemeron :
  Ephemeron
module Filename :
  Filename
module Float :
  Float
module Format :
  Format
module Fun :
  Fun
module Gc :
  Gc
module Genlex :
  Genlex
module Hashtbl :
  Hashtbl
module Int :
  Int
module Int32 :
  Int32
module Int64 :
  Int64
module Lazy :
  Lazy
module Lexing :
  Lexing
module List :
  List
module ListLabels :
  ListLabels
module Map :
  Map
module Marshal :
  Marshal
module MoreLabels :
  MoreLabels
module Nativeint :
  Nativeint
module Obj :
Obj
module Oo :
  Oo
module Option :
  Option
module Parsing :
  Parsing
module Pervasives :
  Pervasives
module Printexc :
  Printexc
module Printf :
  Printf
module Queue :
  Queue
module Random :
  Random
module Result :
  Result
module Scanf :
  Scanf
module Seq :
  Seq
module Set :
  Set
module Spacetime :
  Spacetime
module Stack :
  Stack
module StdLabels :
  StdLabels
module Stream :
  Stream
module String :
  String
module StringLabels :
  StringLabels
module Sys :
  Sys
module Uchar :
  Uchar
module Weak :
  Weak
Chapter 24

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 9 and 12) and plugins invoked via the *-plugin* flag (see chapter 22).

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";;
```

24.1 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*[24.1] allows to implement AST rewriting using open recursion. A typical mapper would be based on *Ast_mapper.default_mapper*[24.1], 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.

Warning: this module is unstable and part of compiler-libs[24].

A generic Parsetree mapper

type mapper =
  { attribute : mapper -> Parsetree.attribute -> Parsetree.attribute ;
    attributes : mapper -> Parsetree.attribute list -> Parsetree.attribute list ;
    binding_op : mapper -> Parsetree.binding_op -> Parsetree.binding_op ;
    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_substitution : mapper -> Parsetree.module_substitution -> Parsetree.module_substitution;
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_declaration : mapper -> Parsetree.open_declaration -> Parsetree.open_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_exception : mapper -> Parsetree.type_exception -> Parsetree.type_exception;
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.

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", "ocamlcmdoc", "ocamldep", "ocaml", ... Some global variables that reflect
command-line options are automatically synchronized between the calling tool and the ppx preprocessor: Clflags.include_dir[24.11], Load_path[??], Clflags.open_modules[24.11], Clflags.for_package[24.11], Clflags.debug[24.11].

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[24.3]. This function implements proper error reporting for uncaught exceptions.

Registration API

val register_function : (string -> (string list -> mapper) -> unit) 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.

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.
Chapter 24. The compiler front-end

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.

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.

val drop_ppx_context_sig :
  restore:bool -> Parsetree.signature -> Parsetree.signature

  Same as drop_ppx_context_str, but for signatures.

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).

val set_cookie : string -> Parsetree.expression -> unit
val get_cookie : string -> Parsetree.expression option

24.2 Module Asttypes: Auxiliary AST types used by parsetree
and typedtree.

Warning: this module is unstable and part of compiler-libs[24].

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 =
  | Nonrecursive
Recursive
type direction_flag =
| Upto
| Downto
type private_flag =
| Private
| Public
type mutable_flag =
| Immutable
| Mutable
type virtual_flag =
| Virtual
| Concrete
type override_flag =
| Override
| Fresh
type closed_flag =
| Closed
| Open
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 =
| Covariant
| Contravariant
| Invariant

24.3 Module Location: Source code locations (ranges of positions), used in parsetree

Warning: this module is unstable and part of compiler-libs[24].

type t = Warnings.loc =
{ 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 rhs_interval : int -> int -> t
val get_pos_info : Lexing.position -> string * int * int

file, line, char

type 'a loc =
{
  txt : 'a ;
  loc : t ;
}
val mknoloc : 'a -> 'a loc
val mkloc : 'a -> t -> 'a loc

Input info

val input_name : string ref
val input_lexbuf : Lexing.lexbuf option ref

Toplevel-specific functions

val echo_eof : unit -> unit
val reset : unit -> unit
Printing locations

val rewrite_absolute_path : string -> string
    rewrite absolute path to honor the BUILD_PATH_PREFIX_MAP variable
    (https://reproducible-builds.org/specs/build-path-prefix-map/) if it is set.

val absolute_path : string -> string
val show_filename : string -> string
    In -absname mode, return the absolute path for this filename. Otherwise, returns the
    filename unchanged.

val print_filename : Format.formatter -> string -> unit
val print_loc : Format.formatter -> t -> unit
val print_locs : Format.formatter -> t list -> unit

Toplevel-specific location highlighting

val highlight_terinfo : Lexing.lexbuf -> Format.formatter -> t list -> unit

Reporting errors and warnings

The type of reports and report printers

type msg = (Format.formatter -> unit) loc
val msg : ?loc:t ->
    ('a, Format.formatter, unit, msg) format4 -> 'a

type report_kind =
    | Report_error
    | Report_warning of string
    | Report_warning_as_error of string
    | Report_alert of string
    | Report_alert_as_error of string

type report =
    { kind : report_kind ;
      main : msg ;
      sub : msg list ;
    }

type report_printer =
    { pp : report_printer -> Format.formatter -> report -> unit ;
      pp_report_kind : report_printer ->
      report -> Format.formatter -> report_kind -> unit ;
      pp_main_loc : report_printer ->
      report -> Format.formatter -> t -> unit ;
      pp_main_txt : report_printer ->
    }
A printer for reports, defined using open-recursion. The goal is to make it easy to define new printers by re-using code from existing ones.

Report printers used in the compiler

val batch_mode_printer : report_printer
val terminfo_toplevel_printer : Lexing.lexbuf -> report_printer
val best_toplevel_printer : unit -> report_printer

Detects the terminal capabilities and selects an adequate printer

Printing a report

val print_report : Format.formatter -> report -> unit

Display an error or warning report.

val report_printer : (unit -> report_printer) ref

Hook for redefining the printer of reports.

The hook is a unit -> report_printer and not simply a report_printer: this is useful so that it can detect the type of the output (a file, a terminal, ...) and select a printer accordingly.

val default_report_printer : unit -> report_printer

Original report printer for use in hooks.

Reporting warnings

Converting a Warnings.t into a report

val report_warning : t -> Warnings.t -> report option

report_warning loc w produces a report for the given warning w, or None if the warning is not to be printed.
val warning_reporter : (t -> Warnings.t -> report option) ref
   Hook for intercepting warnings.

val default_warning_reporter : t -> Warnings.t -> report option
   Original warning reporter for use in hooks.

Printing warnings

val formatter_for_warnings : Format.formatter ref
val print_warning : t -> Format.formatter -> Warnings.t -> unit
   Prints a warning. This is simply the composition of report_warning and print_report.

val prerr_warning : t -> Warnings.t -> unit
   Same as print_warning, but uses !formatter_for_warnings as output formatter.

Reporting alerts

Converting an Alert.t into a report

val report_alert : t -> Warnings.alert -> report option
   report_alert loc w produces a report for the given alert w, or None if the alert is not to be printed.

val alert_reporter : (t -> Warnings.alert -> report option) ref
   Hook for intercepting alerts.

val default_alert_reporter : t -> Warnings.alert -> report option
   Original alert reporter for use in hooks.

Printing alerts

val print_alert : t -> Format.formatter -> Warnings.alert -> unit
   Prints an alert. This is simply the composition of report_alert and print_report.

val prerr_alert : t -> Warnings.alert -> unit
   Same as print_alert, but uses !formatter_for_warnings as output formatter.

val deprecated : ?def:t -> ?use:t -> string -> unit
   Prints a deprecation alert.

val alert : ?def:t ->
   ?use:t -> kind:string -> t -> string -> unit
   Prints an arbitrary alert.
Chapter 24. The compiler front-end

Reporting errors

type error = report

An error is a report which report_kind must be Report_error.

val error : ?loc:t -> ?sub:msg list -> string -> error
val errorf :
  ?loc:t ->
  ?sub:msg list ->
  ('a, Format.formatter, unit, error) format4 -> 'a
val error_of_printer :
  ?loc:t ->
  ?sub:msg list ->
  (Format.formatter -> 'a -> unit) -> 'a -> error
val error_of_printer_file : (Format.formatter -> 'a -> unit) -> 'a -> error

Automatically reporting errors for raised exceptions

val register_error_of_exn : (exn -> error option) -> unit

Each compiler module which defines a custom type of exception which can surface as a
user-visible error should register a "printer" for this exception using
register_error_of_exn. The result of the printer is an error value containing a location,
a message, and optionally sub-messages (each of them being located as well).

val error_of_exn : exn -> [ Already_displayed | Ok of error ] option
exception Error of error
  Raising Error e signals an error e; the exception will be caught and the error will be
  printed.

exception Already_displayed_error
  Raising Already_displayed_error signals an error which has already been printed. The
  exception will be caught, but nothing will be printed

val raise_errorf :
  ?loc:t ->
  ?sub:msg list ->
  ('a, Format.formatter, unit, 'b) format4 -> 'a
val report_exception : Format.formatter -> exn -> unit
  Reraise the exception if it is unknown.
24.4 Module Longident : Long identifiers, used in parsetree.

**Warning:** this module is unstable and part of compiler-libs[24].

```ocaml
type t =
  | Lident of string
  | Ldot of t * string
  | Lapply of t * t
val flatten : t -> string list
val unflatten : string list -> t option
val last : t -> string
val parse : string -> t

To print a longident, see [Pprintast.longident][24.7], using [Format.asprintf][??] to convert to a string.
```

24.5 Module Parse : Entry points in the parser

**Warning:** this module is unstable and part of compiler-libs[24].

```ocaml
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
```

24.6 Module Parsetree : Abstract syntax tree produced by parsing

**Warning:** this module is unstable and part of compiler-libs[24].

```ocaml
type constant =
  | Pconst_integer of string * char option
  | Pconst_char of char
  | Pconst_string of string * string option
  | Pconst_float of string * char option

Extension points

type attribute =
```
{  attr_name : string Asttypes.loc  
    attr_payload : payload  
    attr_loc : Location.t  
}

{  attr_name : string Asttypes.loc  
    attr_payload : payload  
    attr_loc : Location.t  
}

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

Core language

type core_type =
{  ptyp_desc : core_type_desc  
    ptyp_loc : Location.t  
    ptyp_loc_stack : Location.t list  
    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 object_field 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 Asttypes.loc list * core_type
  | Ptyp_package of package_type
  | Ptyp_extension of extension

Core language (continued)

type package_type = Longident.t Asttypes.loc *
  (Longident.t Asttypes.loc * core_type) list

type row_field =
{  prf_desc : row_field_desc  
    prf_loc : Location.t  
    prf_attributes : attributes  
}

type row_field_desc =
  | Rtag of Asttypes.label Asttypes.loc * bool * core_type list
  | Rinherit of core_type

type object_field =
{ pof_desc : object_field_desc ;
    pof_loc : Location.t ;
    pof_attributes : attributes ;
}

type object_field_desc =
    | Otag of Asttypes.label Asttypes.loc * core_type
    | Oinherit of core_type

type pattern =
{ ppat_desc : pattern_desc ;
    ppat_loc : Location.t ;
    ppat_loc_stack : Location.t list ;
    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_loc_stack : Location.t list ;
    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
* expression
  | Pexp_apply of expression * (Asttypes.arg_label * expression) list
  | Pexp_match of expression * case list
  | Pexp_try of expression * case list
  | 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
  | * expression option
  | 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
  | * Asttypes.direction_flag * expression
  | Pexp_constraint of expression * core_type
  | Pexp_coerce of expression * core_type option * core_type
  | Pexp_send of expression * Asttypes.label Asttypes.loc
  | Pexp_new of Longident.t Asttypes.loc
  | Pexp_setinstvar of Asttypes.label Asttypes.loc * expression
  | Pexp_override of (Asttypes.label 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 Asttypes.loc * expression
  | Pexp_pack of module_expr
  | Pexp_open of open_declaration * expression
  | Pexp_letop of { let_ : binding_op ;
                ands : binding_op list ;
                body : expression ;
            };
  | Pexp_extension of extension
  | Pexp_unreachable

type case =
{ pc_lhs : pattern ;
  pc_guard : expression option ;
  pc_rhs : expression ;
}

type binding_op =
{ pbop_op : string Asttypes.loc ;
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
| Ptype_open


{ ptyext_path : Longident.t Asttypes.loc ;
  ptyext_params : (core_type * Asttypes.variance) list ;
  ptyext_constructors : extension_constructor list ;
  ptyext_private : Asttypes.private_flag ;
  ptyext_loc : Location.t ;
  ptyext_attributes : attributes ;
}

type extension_constructor =
{ pext_name : string Asttypes.loc ;
  pext_kind : extension_constructor_kind ;
  pext_loc : Location.t ;
  pext_attributes : attributes ;
}

type type_exception =
{ ptyexn_constructor : extension_constructor ;
  ptyexn_loc : Location.t ;
  ptyexn_attributes : attributes ;
}

type extension_constructor_kind =
| Pext_decl of constructor_arguments * core_type option
| Pext_rebind of Longident.t Asttypes.loc

Class language

type class_type =
{ pcty_desc : class_type_desc ;
  pcty_loc : Location.t ;
  pcty_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
| Pcty_open of open_description * class_type

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 (Asttypes.label Asttypes.loc * Asttypes.mutable_flag * Asttypes.virtual_flag * core_type)
    | Pctf_method of (Asttypes.label Asttypes.loc * 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
    | Pcl_open of open_description * class_expr


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 =
Chapter 24. The compiler front-end

| Pcf_inherit of Asttypes.override_flag * class_expr * string Asttypes.loc option |
| Pcf_val of (Asttypes.label Asttypes.loc * Asttypes.mutable_flag * class_field_kind) |
| Pcf_method of (Asttypes.label 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

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 |
| Pmty_alias of Longident.t Asttypes.loc |

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_typesubst of type_declaration list |
| Psig_typext of type_extension |
| Psig_exception of type_exception |
| Psig_module of module_declaration |
| Psig_modsubst of module_substitution |
| Psig_recmodule of module_declaration list |
| Psig_modtype of module_type_declaration |
type module_declaration =
{ pmd_name : string Asttypes.loc ;
  pmd_type : module_type ;
  pmd_attributes : attributes ;
  pmd_loc : Location.t ;
}

type module_substitution =
{ pms_name : string Asttypes.loc ;
  pms_manifest : Longident.t Asttypes.loc ;
  pms_attributes : attributes ;
  pms_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 'a open_infos =
{ popen_expr : 'a ;
  popen_override : Asttypes.override_flag ;
  popen_loc : Location.t ;
  popen_attributes : attributes ;
}

type open_description = Longident.t Asttypes.loc open_infos

type open_declaration = module_expr open_infos

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 Longident.t Asttypes.loc * type_declaration
| "with_modsubst of Longident.t 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_typeext of type_extension
| Pstr_exception of type_exception
| Pstr_module of module_binding
| Pstr_recmodule of module_binding list
| Pstr_modtype of module_type_declaration
| Pstr_open of open_declaration
| 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 ;
  pvb_loc : Location.t ;
}

type module_binding =
{ pmb_name : string Asttypes.loc ;
  pmb_expr : module_expr ;
}
pmb_attributes : attributes ;
pmb_loc : Location.t ;
}

**Toplevel**

type toplevel_phrase =
  | Ptop_def of structure
  | Ptop_dir of toplevel_directive

type toplevel_directive =
{  pdir_name : string Asttypes.loc ;
  pdir_arg : directive_argument option ;
  pdir_loc : Location.t ;
}

type directive_argument =
{  pdira_desc : directive_argument_desc ;
  pdira_loc : Location.t ;
}

type directive_argument_desc =
  | Pdir_string of string
  | Pdir_int of string * char option
  | Pdir_ident of Longident.t
  | Pdir_bool of bool

### 24.7 Module Pprintast : Pretty-printers for Parsetree[24.6]

**Warning:** this module is unstable and part of compiler-libs[24].

type space_formatter = (unit, Format.formatter, unit) format

val longident : Format.formatter -> Longident.t -> unit
val expression : Format.formatter -> Parsetree.expression -> unit
val string_of_expression : Parsetree.expression -> string
val pattern : Format.formatter -> Parsetree.pattern -> unit
val core_type : Format.formatter -> Parsetree.core_type -> unit
val signature : Format.formatter -> Parsetree.signature -> unit
val structure : Format.formatter -> Parsetree.structure -> unit
val string_of_structure : Parsetree.structure -> string
val toplevel_phrase : Format.formatter -> Parsetree.toplevel_phrase -> unit
val top_phrase : Format.formatter -> Parsetree.toplevel_phrase -> unit
val tyvar : Format.formatter -> string -> unit

Print a type variable name, taking care of the special treatment required for the single quote character in second position.
24.8 Module Pparse: Driver for the parser, external preprocessors and ast plugin hooks

Warning: this module is unstable and part of compiler-libs[24].

```ocaml
type error =
  | CannotRun of string
  | WrongMagic of string
exception Error of error
val preprocess : string -> string
val remove_preprocessed : string -> unit

type 'a ast_kind =
  | Structure : Parsetree.structure ast_kind
  | Signature : Parsetree.signature ast_kind
val read_ast : 'a ast_kind -> string -> 'a
val write_ast : 'a ast_kind -> string -> 'a -> unit
val file :
  tool_name:string ->
  string -> (Lexing.lexbuf -> 'a) -> 'a ast_kind -> 'a
val apply_rewriters :
  ?restore:bool ->
  tool_name:string -> 'a ast_kind -> 'a -> 'a

  If restore = true (the default), cookies set by external rewriters will be kept for later calls.

val apply_rewriters_str :
  ?restore:bool ->
  tool_name:string -> Parsetree.structure -> Parsetree.structure
val apply_rewriters_sig :
  ?restore:bool ->
  tool_name:string -> Parsetree.signature -> Parsetree.signature
val report_error : Format.formatter -> error -> unit
val parse_implementation : tool_name:string -> string -> Parsetree.structure
val parse_interface : tool_name:string -> string -> Parsetree.signature
val call_external_preprocessor : string -> string -> string
val open_and_check_magic : string -> string -> in_channel * bool

module ImplementationHooks :
  Misc.HookSig with type t = Parsetree.structure

module InterfaceHooks :
  Misc.HookSig with type t = Parsetree.signature
```
Module Typemod: Type-checking of the module language and typed ast plugin hooks

Warning: this module is unstable and part of compiler-libs[24].

module Signature_names :
  sig
    type t
    val simplify : Env.t -> t -> Types.signature -> Types.signature
  end

val type_module : Env.t -> Parsetree.module_expr -> Typedtree.module_expr
val type_structure :
  Env.t ->
  Parsetree.structure ->
  Location.t ->
  Typedtree.structure * Types.signature * Signature_names.t * Env.t
val type_toplevel_phrase :
  Env.t ->
  Parsetree.structure ->
  Typedtree.structure * Types.signature * Signature_names.t * Env.t
val type_implementation :
  string ->
  string ->
  Env.t ->
  Parsetree.structure -> Typedtree.structure * Typedtree.module_coercion
val type_interface :
  string -> Env.t -> Parsetree.signature -> Typedtree.signature
val transl_signature : Env.t -> Parsetree.signature -> Typedtree.signature
val check_nongen_schemes : Env.t -> Types.signature -> unit
val modtype_of_package :
  Env.t ->
  Location.t ->
  Path.t -> Longident.t list -> Types.type_expr list -> Types.module_type
val path_of_module : Typedtree.module_expr -> Path.t option
val save_signature :
  string ->
  Typedtree.signature ->
  string -> string -> Env.t -> Cmi_format.cmi_infos -> unit
val package_units :
  Env.t -> string list -> string -> string -> Typedtree.module_coercion
val initial_env :
   loc:Location.t ->
   safe_string:bool ->
   initially_opened_module:string option ->
   open_implicit_modules:string list -> Env.t

module Sig_component_kind :
  sig
    type t =
      | Value
      | Type
      | Module
      | Module_type
      | Extension_constructor
      | Class
      | Class_type
    val to_string : t -> string
  end

type hiding_error =
  | Illegal_shadowing of { shadowed_item_id : Ident.t ;
                          shadowed_item_kind : Sig_component_kind.t ;
                          shadowed_item_loc : Location.t ;
                          shadower_id : Ident.t ;
                          user_id : Ident.t ;
                          user_kind : Sig_component_kind.t ;
                          user_loc : Location.t ;
                      }
  | Appears_in_signature of { opened_item_id : Ident.t ;
                            opened_item_kind : Sig_component_kind.t ;
                            user_id : Ident.t ;
                            user_kind : Sig_component_kind.t ;
                            user_loc : Location.t ;
                      }

  type error =
    | Cannot_apply of Types.module_type
    | Not_included of Includemod.error list
    | Cannot_eliminate_dependency of Types.module_type
    | Signature_expected
    | Structure_expected of Types.module_type
    | With_no_component of Longident.t
    | With_mismatch of Longident.t * Includemod.error list
    | With_makes_applicative_functor_ill_typed of Longident.t * Path.t * Include-
      mod.error list
    | With_changes_module_alias of Longident.t * Ident.t * Path.t

| With_cannot_remove_constrained_type |
| Repeated_name of Sig_component_kind.t * string |
| Non_generalizable of Types.type_expr |
| Non_generalizable_class of Ident.t * Types.class_declaration |
| Non_generalizable_module of Types.module_type |
| Implementation_is_required of string |
| Interface_not_compiled of string |
| Not_allowed_in_functor_body |
| Not_a_packed_module of Types.type_expr |
| Incomplete_packed_module of Types.type_expr |
| Scoping_pack of Longident.t * Types.type_expr |
| Recursive_module_require_explicit_type |
| Apply_generative |
| Cannot_scrape_alias of Path.t |
| Badly_formed_signature of string * Typedecl.error |
| Cannot_hide_id of hiding_error |
| Incomplete_packed_module of Types.type_expr |

exception Error of Location.t * Env.t * error
exception Error_forward of Location.error
val report_error : Env.t -> Format.formatter -> error -> unit

module ImplementationHooks :
  Misc.HookSig with type t = Typedtree.structure * Typedtree.module_coercion
module InterfaceHooks :
  Misc.HookSig with type t = Typedtree.signature

24.10 Module Simplif : Lambda simplification and lambda plugin hooks

Warning: this module is unstable and part of compiler-libs[24].

val simplify_lambda : string -> Lambda.lambda -> Lambda.lambda
val split_default_wrapper :
  id:Ident.t ~> kind:Lambda.function_kind ~> 
  params:(Ident.t * Lambda.value_kind) list ~> 
  return:Lambda.value_kind ~> 
  body:Lambda.lambda ~> 
  attr:Lambda.function_attribute ~> 
  loc:Location.t ~> (Ident.t * Lambda.lambda) list
val is_tail_native_heuristic : (int ~> bool) ref

module Hooks :
  Misc.HookSig with type t = lambda
24.11 Module CFLAGS: Command line flags

module Int_arg_helper :
sig
  type parsed
  val parse : string -> string -> parsed ref -> unit
  type parse_result =
    | Ok
    | Parse_failed of exn
  val parse_no_error : string ->
    parsed ref ->
    parse_result
  val get : key:int -> parsed -> int
end

Optimization parameters represented as ints indexed by round number.

module Float_arg_helper :
sig
  type parsed
  val parse : string -> string -> parsed ref -> unit
  type parse_result =
    | Ok
    | Parse_failed of exn
  val parse_no_error : string ->
    parsed ref ->
    parse_result
  val get : key:int -> parsed -> float
end

Optimization parameters represented as floats indexed by round number.

type inlining_arguments =
{  inline_call_cost : int option ;
  inline_alloc_cost : int option ;
  inline_prim_cost : int option ;
  inline_branch_cost : int option ;
  inline_indirect_cost : int option ;
  inline_lifting_benefit : int option ;
  inline_branch_factor : float option ;
  inline_max_depth : int option ;
  inline_max_unroll : int option ;
inline_threshold : float option;
inline_toplevel_threshold : int option;
}
val classic_arguments : inlining_arguments
val o1_arguments : inlining_arguments
val o2_arguments : inlining_arguments
val o3_arguments : inlining_arguments
val use_inlining_arguments_set : ?round:int -> inlining_arguments -> unit
Set all the inlining arguments for a round. The default is set if no round is provided.
val objfiles : string list ref
val ccobjs : string list ref
val dllibs : string list ref
val compile_only : bool ref
val output_name : string option ref
val include_dirs : string list ref
val no_std_include : bool ref
val print_types : bool ref
val make_archive : bool ref
val debug : bool ref
val debug_full : bool ref
val unsafe : bool ref
val use_linscan : bool ref
val link_everything : bool ref
val custom_runtime : bool ref
val no_check_prims : bool ref
val bytecode_compatible_32 : bool ref
val output_c_object : bool ref
val output_complete_object : bool ref
val all_ccopts : string list ref
val classic : bool ref
val nopervasives : bool ref
val match_context_rows : int ref
val open_modules : string list ref
val preprocessor : string option ref
val all_ppx : string list ref
val absname : bool ref
val annotations : bool ref
val binary_annotations : bool ref
val use_threads : bool ref
val use_vmtthreads : bool ref
val noassert : bool ref
val verbose : bool ref
val noprompt : bool ref
val nopromptcont : bool ref
val init_file : string option ref
val noinit : bool ref
val noversion : bool ref
val use_prims : string ref
val use_runtime : string ref
val principal : bool ref
val real_paths : bool ref
val recursive_types : bool ref
val strict_sequence : bool ref
val strict_formats : bool ref
val applicative_functors : bool ref
val make_runtime : bool ref
val gprofile : bool ref
val c_compiler : string option ref
val no_auto_link : bool ref
val dllpaths : string list ref
val make_package : bool ref
val for_package : string option ref
val error_size : int ref
val float_const_prop : bool ref
val transparent_modules : bool ref
val unique_ids : bool ref
val dump_source : bool ref
val dump_parsetree : bool ref
val dump_typedtree : bool ref
val dump_rawlambda : bool ref
val dump_lambda : bool ref
val dump_rawclambda : bool ref
val dump_clambda : bool ref
val dump_rawflambda : bool ref
val dump_flambda : bool ref
val dump_flambda_let : int option ref
val dump_instr : bool ref
val keep_camlprimc_file : bool ref
val keep_asm_file : bool ref
val optimize_for_speed : bool ref
val dump_cmm : bool ref
val dump_selection : bool ref
val dump_cse : bool ref
val dump_live : bool ref
val dump_avail : bool ref
val debug_runavail : bool ref
val dump_spill : bool ref
val dump_split : bool ref
val dump_interf : bool ref
val dump_prefer : bool ref
val dump_regalloc : bool ref
val dump_reload : bool ref
val dump_scheduling : bool ref
val dump_linear : bool ref
val dump_interval : bool ref
val keep_startup_file : bool ref
val dump_combine : bool ref
val native_code : bool ref
val default_inline_threshold : float
val inline_threshold : Float_arg_helper.parsed ref
val inlining_report : bool ref
val simplify_rounds : int option ref
val default_simplify_rounds : int ref
val rounds : unit -> int
val default_inline_max_unroll : int
val inline_max_unroll : Int_arg_helper.parsed ref
val default_inline_toplevel_threshold : int
val inline_toplevel_threshold : Int_arg_helper.parsed ref
val default_inline_call_cost : int
val default_inline_alloc_cost : int
val default_inline_prim_cost : int
val default_inline_branch_cost : int
val default_inline_indirect_cost : int
val default_inline_lifting_benefit : int
val inline_call_cost : Int_arg_helper.parsed ref
val inline_alloc_cost : Int_arg_helper.parsed ref
val inline_prim_cost : Int_arg_helper.parsed ref
val inline_branch_cost : Int_arg_helper.parsed ref
val inline_indirect_cost : Int_arg_helper.parsed ref
val inline_lifting_benefit : Int_arg_helper.parsed ref
val default_inline_branch_factor : float
val inline_branch_factor : Float_arg_helper.parsed ref
val dont_write_files : bool ref
val std_include_flag : string -> string
val std_include_dir : unit -> string list
val shared : bool ref
val dlcode : bool ref
val pic_code : bool ref
val runtime_variant : string ref
val force_slash : bool ref
val keep_docs : bool ref
val keep_locs : bool ref
val unsafe_string : bool ref
val opaque : bool ref
val profile_columns : Profile.column list ref
val flambda_invariant_checks : bool ref
val unbox_closures : bool ref
val unbox_closures_factor : int ref
val default_unbox_closures_factor : int
val unbox_free_vars_of_closures : bool ref
val unbox_specialised_args : bool ref
val clambda_checks : bool ref
val default_inline_max_depth : int
val inline_max_depth : Int_arg_helper.parsed ref
val remove_unused_arguments : bool ref
val dump_flambda_verbose : bool ref
val classic_inlining : bool ref
val afl_instrument : bool ref
val afl_inst_ratio : int ref
val all_passes : string list ref
val dumped_pass : string -> bool
val set_dumped_pass : string -> bool -> unit
val dump_into_file : bool ref

type 'a env_reader =
  { parse : string -> 'a option ;
    usage : string ;
    env_var : string ;
  }
val color : Misc.Color.setting option ref
val color_reader : Misc.Color.setting env_reader
val error_style : Misc.Error_style.setting option ref
val error_style_reader : Misc.Error_style.setting env_reader
val unboxed_types : bool ref
module Compiler_pass :
  sig
    type t =
      | Parsing
      | Typing
    val of_string : string -> t option
    val to_string : t -> string
    val passes : t list
    val pass_names : string list
  end
val stop_after : Compiler_pass.t option ref
val should_stop_after : Compiler_pass.t -> bool
val arg_spec : (string * Arg.spec * string) list ref
val add_arguments : string -> (string * Arg.spec * string) list -> unit
val parse_arguments : Arg.anon_fun -> string -> unit
val print_arguments : string -> unit
val reset_arguments : unit -> unit
Chapter 25

The unix library: Unix system calls

The unix library makes many Unix system calls and system-related library functions available to OCaml programs. This chapter describes briefly the functions provided. Refer to sections 2 and 3 of the Unix manual for more details on the behavior of these functions.

Not all functions are provided by all Unix variants. If some functions are not available, they will raise Invalid_arg when called.

Programs that use the unix library must be linked as follows:

```
ocamlc other options unix.cma other files
ocamlopt other options unix.cmxa other files
```

For interactive use of the unix library, do:

```
ocamlmktop -o mytop unix.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type

```
#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.

25.1 Module Unix : Interface to the Unix system.

Note: all the functions of this module (except Unix.error_message[25.1] and Unix.handle_unix_error[25.1]) are liable to raise the Unix.Unix_error[25.1] exception whenever the underlying system call signals an error.

Error report

```
type error =
  | E2BIG
```

481
Argument list too long

| EACCESS | 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. |
| EXIST | 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 | |
Chapter 25. The unix library: Unix system calls

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 |
| ENOBUFS  | No buffer space available |
| EISCONN  | Socket is already connected |
| ENOTCONN  | Socket is not connected |
| ESHUTDOWN  | Can’t send after socket shutdown |
| ETOMANYREFS  | Too many references: can’t splice |
| ETIMEDOUT  | Connection timed out |
| ECONNREFUSED  | Connection refused |
| EHOSTDOWN  | Host is down |
| EHOSTUNREACH  | No route to host |
| ELOOP  | Too many levels of symbolic links |
| EOVERFLOW  | File size or position not representable |
| EUNKNOWNERR of int  |
Unknown error

The type of error codes. Errors defined in the POSIX standard and additional errors from UNIX98 and BSD. All other errors are mapped to EUNKNOWNERR.

exception Unix_error of error * string * string

Raised by the system calls below when an error is encountered. The first component is the error code; the second component is the function name; the third component is the string parameter to the function, if it has one, or the empty string otherwise.

val error_message : error -> string

Return a string describing the given error code.

val handle_unix_error : ('a -> 'b) -> 'a -> 'b

handle_unix_error f x applies f to x and returns the result. If the exception Unix.Unix_error[25.1] 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”. The returned array is empty if the process has special privileges.

val unsafe_environment : unit -> string array

Return the process environment, as an array of strings with the format “variable=value”. Unlike Unix.environment[25.1], this function returns a populated array even if the process has special privileges. See the documentation for Unix.unsafe_getenv[25.1] for more details.

Since: 4.06.0

val getenv : string -> string

Return the value associated to a variable in the process environment, unless the process has special privileges.

Raises Not_found if the variable is unbound or the process has special privileges.

(This function is identical to Sys.getenv[??].)

val unsafe_getenv : string -> string

Return the value associated to a variable in the process environment.

Unlike Unix.getenv[25.1], this function returns the value even if the process has special privileges. It is considered unsafe because the programmer of a setuid or setgid program must be careful to avoid using maliciously crafted environment variables in the search path for executables, the locations for temporary files or logs, and the like.

Since: 4.06.0

Raises Not_found if the variable is unbound.
val putenv : string -> string -> unit

Unix.putenv name value sets the value associated to a variable in the process environment. name is the name of the environment variable, and value its new associated value.

Process handling

type process_status =
  | WEXITED of int

  The process terminated normally by exit; the argument is the return code.

  | WSIGNALED of int

  The process was killed by a signal; the argument is the signal number.

  | WSTOPPED of int

  The process was stopped by a signal; the argument is the signal number.

  The termination status of a process. See module Sys for the definitions of the standard signal numbers. Note that they are not the numbers used by the OS.

type wait_flag =
  | WNOHANG

  Do not block if no child has died yet, but immediately return with a pid equal to 0.

  | WUNTRACED

  Report also the children that receive stop signals.

  Flags for Unix.waitpid.

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.

val execve : string -> string array -> string array -> 'a

Same as Unix.execv, except that the third argument provides the environment to the program executed.

val execvp : string -> string array -> 'a

Same as Unix.execv, except that the program is searched in the path.

val execvpe : string -> string array -> string array -> 'a

Same as Unix.execve, except that the program is searched in the path.

val fork : unit -> int
Fork a new process. The returned integer is 0 for the child process, the pid of the child process for the parent process.

On Windows: not implemented, use Unix.create_process[25.1] or threads.

```ml
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[25.1].
```

```ml
val waitpid : wait_flag list -> int -> int * process_status
Same as Unix.wait[25.1], but waits for the child process whose pid is given. A pid of -1 means wait for any child. A pid of 0 means wait for any child in the same process group as the current process. Negative pid arguments represent process groups. The list of options indicates whether waitpid should return immediately without waiting, and whether it should report stopped children.
On Windows, this function can only wait for a given PID, not any child process.
```

```ml
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.
```

```ml
val getpid : unit -> int
Return the pid of the process.
```

```ml
val getppid : unit -> int
Return the pid of the parent process. On Windows: not implemented (because it is meaningless).
```

```ml
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

```ml
type file_descr
The abstract type of file descriptors.
```

```ml
val stdin : file_descr
File descriptor for standard input.
```

```ml
val stdout : file_descr
```
File descriptor for standard output.

val stderr : file_descr

File descriptor for standard error.

type open_flag =
    | O_RDONLY
        Open for reading
    | O_WRONLY
        Open for writing
    | O_RDWR
        Open for reading and writing
    | O_NONBLOCK
        Open in non-blocking mode
    | O_APPEND
        Open for append
    | O_CREAT
        Create if nonexistent
    | O_TRUNC
        Truncate to 0 length if existing
    | O_EXCL
        Fail if existing
    | O_NOCTTY
        Don’t make this dev a controlling tty
    | O_DSYNC
        Writes complete as ‘Synchronised I/O data integrity completion’
    | O_SYNC
        Writes complete as ‘Synchronised I/O file integrity completion’
    | O_RSYNC
        Reads complete as writes (depending on O_SYNC/O_DSYNC)
    | O_SHARE_DELETE
        Windows only: allow the file to be deleted while still open
    | O_CLOEXEC
        Set the close-on-exec flag on the descriptor returned by Unix.openfile[25.1]. See Unix.set_close_on_exec[25.1] for more information.
    | O_KEEPEXEC
Clear the close-on-exec flag. This is currently the default.

The flags to Unix.openfile[25.1].

type file_perm = int

The type of file access rights, e.g. 0o640 is read and write for user, read for group, none for others

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[25.1]). Return a file descriptor on the named file.

val close : file_descr -> unit

Close a file descriptor.

val fsync : file_descr -> unit

Flush file buffers to disk.

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.

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.

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.

val write_substring : file_descr -> string -> int -> int -> int
Same as write, but take the data from a string instead of a byte sequence.

Since: 4.02.0

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.

Since: 4.02.0
Interfacing with the standard input/output library

val in_channel_of_descr : file_descr -> in_channel
Create an input channel reading from the given descriptor. The channel is initially in binary mode; use \texttt{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, \texttt{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 -> out_channel
Create an output channel writing on the given descriptor. The channel is initially in binary mode; use \texttt{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, \texttt{set\_binary\_mode\_out} always fails on channels created with this function.

Beware that channels are buffered so you may have to \texttt{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 : in_channel -> file_descr
Return the descriptor corresponding to an input channel.

val descr_of_out_channel : out_channel -> file_descr
Return the descriptor corresponding to an output channel.

Seeking and truncating

type seek_command =
  | SEEK_SET
    indicates positions relative to the beginning of the file
  | SEEK_CUR
    indicates positions relative to the current position
  | SEEK_END
    indicates positions relative to the end of the file
Positioning modes for Unix.lseek[25.1].
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_SOCK
    Socket
type stats =
{ st_dev : int ;
  Device number
  st_ino : int ;
  Inode number
  st_kind : file_kind ;
  Kind of the file
  st_perm : file_perm ;
Chapter 25. The unix library: Unix system calls

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 ID (if special file)

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[25.1] calls.

val stat : string -> stats
Return the information for the named file.

val lstat : string -> stats
Same as Unix.stat[25.1], 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

module LargeFile :
  sig
    val lseek : Unix.file_descr -> int64 -> Unix.seek_command -> int64
      See Unix.lseek[25.1].
    val truncate : string -> int64 -> unit
      See Unix.truncate[25.1].
    val ftruncate : Unix.file_descr -> int64 -> unit
      See Unix.ftruncate[25.1].

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 ID (if special file)
  st_size : int64 ;
        Size in bytes
  st_atime : float ;
        Last access time
Chapter 25. The unix library: Unix system calls

```ocaml
st_mtime : float ;
  Last modification time
st_ctime : float ;
  Last status change time
```

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 (for positioning a file descriptor), Unix.truncate and Unix.ftruncate (for changing the size of a file), and Unix.stat, Unix.lstat and Unix.fstat (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.

Mapping files into memory


Memory mapping of a file as a Bigarray. map_file fd kind layout shared dims returns a Bigarray of kind kind, layout layout, and dimensions as specified in dims. The data contained in this Bigarray are the contents of the file referred to by the file descriptor fd (as opened previously with Unix.openfile, for example). The optional pos parameter is the byte offset in the file of the data being mapped; it defaults to 0 (map from the beginning of the file).

If shared is true, all modifications performed on the array are reflected in the file. This requires that fd be opened with write permissions. If shared is false, modifications performed on the array are done in memory only, using copy-on-write of the modified pages; the underlying file is not affected.

Genarray.map_file is much more efficient than reading the whole file in a Bigarray, modifying that Bigarray, and writing it afterwards.

To adjust automatically the dimensions of the Bigarray 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 \texttt{Failure} is raised.

If all dimensions of the \texttt{Bigarray} are given, the file size is matched against the size of the \texttt{Bigarray}. If the file is larger than the \texttt{Bigarray}, only the initial portion of the file is mapped to the \texttt{Bigarray}. If the file is smaller than the big array, the file is automatically grown to the size of the \texttt{Bigarray}. This requires write permissions on \texttt{fd}.

Array accesses are bounds-checked, but the bounds are determined by the initial call to \texttt{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.

\texttt{Invalid_argument} or \texttt{Failure} may be raised in cases where argument validation fails.

\textbf{Since:} 4.06.0

\textbf{Operations on file names}

\begin{verbatim}
val unlink : string -> unit

Removes the named file.

If the named file is a directory, raises:

\begin{itemize}
  \item \texttt{EPERM} on POSIX compliant system
  \item \texttt{EISDIR} on Linux $\geq 2.1.132$
  \item \texttt{EACCESS} on Windows
\end{itemize}

val rename : string -> string -> unit

\texttt{rename old new} changes the name of a file from \texttt{old} to \texttt{new}, moving it between directories if needed. If \texttt{new} already exists, its contents will be replaced with those of \texttt{old}. Depending on the operating system, the metadata (permissions, owner, etc) of \texttt{new} can either be preserved or be replaced by those of \texttt{old}.

val link : ?follow:bool -> string -> string -> unit

\texttt{link ?follow source dest} creates a hard link named \texttt{dest} to the file named \texttt{source}.

\textbf{Raises}

\begin{itemize}
  \item \texttt{ENOSYS} On \texttt{Unix} if \texttt{~follow:$_{}$} is requested, but \texttt{linkat} is unavailable.
  \item \texttt{ENOSYS} On \texttt{Windows} if \texttt{~follow:$_{}false$} is requested.
\end{itemize}
\end{verbatim}
Chapter 25. The unix library: Unix system calls

File permissions and ownership

type access_permission =
  | R_OK
      Read permission
  | W_OK
      Write permission
  | X_OK
      Execution permission
  | F_OK
      File exists

Flags for the Unix.access call.

val chmod : string -> file_perm -> unit
  Change the permissions of the named file.

val fchmod : file_descr -> file_perm -> unit
  Change the permissions of an opened file. 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 : ?cloexec:bool -> file_descr -> file_descr
  Return a new file descriptor referencing the same file as the given descriptor. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

val dup2 : ?cloexec:bool -> file_descr -> file_descr -> unit
  dup2 fd1 fd2 duplicates fd1 to fd2, closing fd2 if already opened. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

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[25.1].

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, create_process and open_process functions.

  It is often a security hole to leak file descriptors opened on, say, a private file to an external program: the program, then, gets access to the private file and can do bad things with it. Hence, it is highly recommended to set all file descriptors “close-on-exec”, except in the very few cases where a file descriptor actually needs to be transmitted to another program.

  The best way to set a file descriptor “close-on-exec” is to create it in this state. To this end, the openfile function has O_CLOEXEC and O_KEEPEXEC flags to enforce “close-on-exec” mode or “keep-on-exec” mode, respectively. All other operations in the Unix module that create file descriptors have an optional argument ?cloexec:bool to indicate whether the file descriptor should be created in “close-on-exec” mode (by writing ~cloexec:true) or in “keep-on-exec” mode (by writing ~cloexec:false). For historical reasons, the default file descriptor creation mode is “keep-on-exec”, if no cloexec optional argument is given. This is not a safe default, hence it is highly recommended to pass explicit cloexec arguments to operations that create file descriptors.

  The cloexec optional arguments and the 0_KEEPEXEC flag were introduced in OCaml 4.05. Earlier, the common practice was to create file descriptors in the default, “keep-on-exec” mode, then call set_close_on_exec on those freshly-created file descriptors. This is not as safe as creating the file descriptor in “close-on-exec” mode because, in multithreaded programs, a window of vulnerability exists between the time when the file descriptor is created and the time set_close_on_exec completes. If another thread spawns another program during this window, the descriptor will leak, as it is still in the “keep-on-exec” mode.
Regarding the atomicity guarantees given by `cloexec: true` or by the use of the
0 _CL OEXEC flag: on all platforms it is guaranteed that a concurrently-executing Caml thread
cannot leak the descriptor by starting a new process. On Linux, this guarantee extends to
concurrently-executing C threads. As of Feb 2017, other operating systems lack the
necessary system calls and still expose a window of vulnerability during which a C thread
can see the newly-created file descriptor in “keep-on-exec” mode.

val clear_close_on_exec : file_descr -> unit
Clear the “close-on-exec” flag on the given descriptor. See Unix.set_close_on_exec[25.1].

Directories

val mkdir : string -> file_perm -> unit
Create a directory with the given permissions (see Unix.umask[25.1]).

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 : ?cloexec:bool -> 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. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

val mkfifo : string -> file_perm -> unit

Create a named pipe with the given permissions (see Unix.umask[25.1]). 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 ->
  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[25.1], except that the extra argument env specifies the environment passed to the program.

val open_process_in : string -> 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 -> out_channel

Same as Unix.open_process_in[25.1], 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 flush[23.3] at the right times to ensure correct synchronization.
Chapter 25. The unix library: Unix system calls

val open_process : string -> in_channel * out_channel
  Same as Unix.open_process_out[25.1], but redirects both the standard input and standard output of the command to pipes connected to the two returned channels. The input channel is connected to the output of the command, and the output channel to the input of the command.

val open_process_full :
  string ->
  string array -> in_channel * out_channel * in_channel
  Similar to Unix.open_process[25.1], 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 open_process_args_in : string -> string array -> in_channel
  High-level pipe and process management. The first argument specifies the command to run, and the second argument specifies the argument array passed to the command. This function runs the 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.
  Since: 4.08.0

val open_process_args_out : string -> string array -> out_channel
  Same as Unix.open_process_args_in[25.1], 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 flush[23.3] at the right times to ensure correct synchronization.
  Since: 4.08.0

val open_process_args : string -> string array -> in_channel * out_channel
  Same as Unix.open_process_args_out[25.1], 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.
  Since: 4.08.0

val open_process_args_full :
  string ->
  string array ->
  string array -> in_channel * out_channel * in_channel
  Similar to Unix.open_process_args[25.1], but the third 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.
  Since: 4.08.0

val process_in_pid : in_channel -> int
Return the pid of a process opened via `Unix.open_process_in`\[25.1\] or `Unix.open_process_args_in`\[25.1\].

**Since:** 4.08.0

```ocaml
val process_out_pid : out_channel -> int
```

Return the pid of a process opened via `Unix.open_process_out`\[25.1\] or `Unix.open_process_args_out`\[25.1\].

**Since:** 4.08.0

```ocaml
val process_pid : in_channel * out_channel -> int
```

Return the pid of a process opened via `Unix.open_process`\[25.1\] or `Unix.open_process_args`\[25.1\].

**Since:** 4.08.0

```ocaml
val process_full_pid : in_channel * out_channel * in_channel -> int
```

Return the pid of a process opened via `Unix.open_process_full`\[25.1\] or `Unix.open_process_args_full`\[25.1\].

**Since:** 4.08.0

```ocaml
val close_process_in : in_channel -> process_status
```

Close channels opened by `Unix.open_process_in`\[25.1\], wait for the associated command to terminate, and return its termination status.

```ocaml
val close_process_out : out_channel -> process_status
```

Close channels opened by `Unix.open_process_out`\[25.1\], wait for the associated command to terminate, and return its termination status.

```ocaml
val close_process : in_channel * out_channel -> process_status
```

Close channels opened by `Unix.open_process`\[25.1\], wait for the associated command to terminate, and return its termination status.

```ocaml
val close_process_full : in_channel * out_channel * in_channel -> process_status
```

Close channels opened by `Unix.open_process_full`\[25.1\], wait for the associated command to terminate, and return its termination status.

**Symbolic links**

```ocaml
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 is 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[25.1] 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.

Since: 4.03.0

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

```ml
type lock_command =
  | F_ULOCK
  | Unlock a region
  | F_LOCK
  | Lock a region for writing, and block if already locked
  | F_TLOCK
  | Lock a region for writing, or fail if already locked
  | F_TEST
  | Test a region for other process locks
  | F_RLOCK
  | Lock a region for reading, and block if already locked
  | F_TRLOCK
  | Lock a region for reading, or fail if already locked

Commands for Unix.lockf[25.1].
```

```ml
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[25.1]), and extends size bytes
forward if size is positive, size bytes backwards if size is negative, or to the end of the file
if size is zero. A write lock prevents any other process from acquiring a read or write lock
on the region. A read lock prevents any other process from acquiring a write lock on the
region, but lets other processes acquire read locks on it.

The F_LOCK and F_TLOCK commands attempts to put a write lock on the specified region.
The F_RLOCK and F_TRLOCK commands attempts to put a read lock on the specified region.
If one or several locks put by another process prevent the current process from acquiring the
lock, F_LOCK and F_RLOCK block until these locks are removed, while F_TLOCK and F_TRLOCK
fail immediately with an exception. The F_ULOCK removes whatever locks the current
process has on the specified region. Finally, the F_TEST command tests whether a write lock
can be acquired on the specified region, without actually putting a lock. It returns
immediately if successful, or fails otherwise.

What happens when a process tries to lock a region of a file that is already locked by the
same process depends on the OS. On POSIX-compliant systems, the second lock operation
succeeds and may "promote" the older lock from read lock to write lock. On Windows, the
second lock operation will block or fail.
Signals

Note: installation of signal handlers is performed via the functions `Sys.signal` and `Sys.set_signal`.

```ocaml
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.
```

```ocaml
type sigprocmask_command =
    | SIG_SETMASK
    | SIG_BLOCK
    | SIG_UNBLOCK
val sigprocmask : sigprocmask_command -> int list -> int list
    sigprocmask cmd sigs changes the set of blocked signals. If cmd is SIG_SETMASK, blocked signals are set to those in the list sigs. If cmd is SIG_BLOCK, the signals in sigs are added to the set of blocked signals. If cmd is SIG_UNBLOCK, the signals in sigs are removed from the set of blocked signals. sigprocmask returns the set of previously blocked signals.
    On Windows: not implemented (no inter-process signals on Windows).
```

```ocaml
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).
```

```ocaml
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).
```

```ocaml
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

```ocaml
type process_times =
    { tms_utime : float ;
        User time for the process
    tms_stime : float ;
        System time for the process
    tms_cutime : float ;
        System time for the process
    }
User time for the children processes

    tms_cstime : float ;
    System time for the children processes

}\n
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
}

The type representing wallclock time and calendar date.

val time : unit -> float
Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

val gettimeofday : unit -> float
Same as Unix.time[25.1], but with resolution better than 1 second.

val gmtime : float -> tm
Convert a time in seconds, as returned by Unix.time[25.1], into a date and a time. Assumes UTC (Coordinated Universal Time), also known as GMT. To perform the inverse conversion, set the TZ environment variable to "UTC", use Unix.mktime[25.1], and then restore the original value of TZ.
val localtime : float -> tm

Convert a time in seconds, as returned by Unix.time[25.1], into a date and a time. Assumes
the local time zone. The function performing the inverse conversion is Unix.mktime[25.1].

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[25.1]. 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.

type interval_timer =
  | ITIMER_REAL
    decrements in real time, and sends the signal SIGALRM when expired.
  | ITIMER_VIRTUAL
    decrements in process virtual time, and sends SIGVTALRM when expired.
  | ITIMER_PROF
    (for profiling) decrements both when the process is running and when the system is
    running on behalf of the process; it sends SIGPROF when expired.

The three kinds of interval timers.
type interval_timer_status =
{ it_interval : float ;
  Period
  it_value : float ;
  Current value of the timer
}

The type describing the status of an interval timer

val getimer : interval_timer -> interval_timer_status
  Return the current status of the given interval timer.
  On Windows: not implemented.

val setimer :
  interval_timer ->
  interval_timer_status -> interval_timer_status
  settimer t s sets the interval timer t and returns its previous status.
  The s argument is interpreted as follows:
  s.it_value, if nonzero, is the time to the next timer expiration;
  s.it_interval, if nonzero, specifies a value to be used in reloading
  it_value when the timer expires. Setting s.it_value to zero disables
  the timer. Setting s.it_interval to zero causes the timer to be
  disabled after its next expiration.
  On Windows: not implemented.

User id, group id

val getuid : unit -> int
  Return the user id of the user executing the process. On Windows, always
  return 1.

val geteuid : unit -> int
  Return the effective user id under which the process runs. On Windows, always
  return 1.

val setuid : int -> unit
  Set the real user id and effective user id for the process. On Windows: not implemented.

val getgid : unit -> int
  Return the group id of the user executing the process. On Windows, always return 1.

val getegid : unit -> int
  Return the effective group id under which the process runs. On Windows, always return 1.

val setgid : int -> unit
  Set the real group id and effective group id for the process. On Windows: not implemented.
val getgroups : unit -> int array
Return the list of groups to which the user executing the process belongs. On Windows, always return [11].

val setgroups : int array -> unit
setgroups groups sets the supplementary group IDs for the calling process. Appropriate privileges are required. On Windows: not implemented.

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.

type passwd_entry =
{ pw_name : string;
  pw_passwd : string;
  pw_uid : int;
  pw_gid : int;
  pw_gecos : string;
  pw_dir : string;
  pw_shell : string;
}
Structure of entries in the passwd database.

type group_entry =
{ gr_name : string;
  gr_passwd : string;
  gr_gid : int;
  gr_mem : string array;
}
Structure of entries in the groups database.

val getlogin : unit -> string
Return the login name of the user executing the process.

val getpwnam : string -> passwd_entry
Find an entry in passwd with the given name.

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

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).
Chapter 25. The unix library: Unix system calls

Sockets

type socket_domain =
  | PF_UNIX
    Unix domain
  | PF_INET
    Internet domain (IPv4)
  | PF_INET6
    Internet domain (IPv6)

The type of socket domains. Not all platforms support IPv6 sockets (type PF_INET6). Windows does not support PF_UNIX.

type socket_type =
  | SOCK_STREAM
    Stream socket
  | SOCK_DGRAM
    Datagram socket
  | SOCK_RAW
    Raw socket
  | SOCK_SEQPACKET
    Sequenced packets socket

The type of socket kinds, specifying the semantics of communications. 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 :
  ?cloexec:bool ->
  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. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

val domain_of_sockaddr : sockaddr -> socket_domain
Return the socket domain adequate for the given socket address.

val socketpair : ?cloexec:bool ->
  socket_domain ->
  socket_type -> int -> file_descr * file_descr

Create a pair of unnamed sockets, connected together. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

val accept : ?cloexec:bool -> 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. See Unix.set_close_on_exec[25.1] for documentation on the cloexec optional argument.

val bind : file_descr -> sockaddr -> unit

Bind a socket to an address.

val connect : file_descr -> sockaddr -> unit

Connect a socket to an address.

val listen : file_descr -> int -> unit

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

type shutdown_command =
  | SHUTDOWN_RECEIVE
    Close for receiving
  | SHUTDOWN_SEND
    Close for sending
  | SHUTDOWN_ALL
    Close both

The type of commands for shutdown.

val shutdown : file_descr -> shutdown_command -> unit

Shutdown a socket connection. SHUTDOWN_SEND as second argument causes reads on the other end of the connection to return an end-of-file condition. SHUTDOWN_RECEIVE causes writes on the other end of the connection to return a closed pipe condition (SIGPIPE signal).

val getsockname : file_descr -> sockaddr

Return the address of the given socket.

val getpeername : file_descr -> sockaddr

Return the address of the host connected to the given socket.
type msg_flag =
  | MSG_OOB
  | MSG_DONTROUTE
  | MSG_PEEK

  The flags for Unix.recv[25.1], Unix.recvfrom[25.1], Unix.send[25.1] and
  Unix.sendto[25.1].

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.

  Since: 4.02.0

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.

  Since: 4.02.0

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`[25.1] and modified with `Unix.setsockopt`[25.1]. These options have a boolean (`true/false`) value.

```
type socket_int_option =
| SO_SNDBUF      |
| Size of send buffer |

| SO_RCVBUF      |
| Size of received buffer |

| SO_ERROR       |

| 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`[25.1] and modified with `Unix.setsockopt_int`[25.1]. These options have an integer value.

```
type socket_optint_option =
| SO_LINGER      |
| Whether to linger on closed connections that have data present, and for how long (in seconds) |
The socket options that can be consulted with \texttt{Unix.getsockopt\_optint[25.1]} and modified with \texttt{Unix.setsockopt\_optint[25.1]}. These options have a value of type \texttt{int option}, with \texttt{None} meaning “disabled”.

\begin{verbatim}
|  \texttt{SO_RCVTIMEO}  \\
|  Timeout for input operations  \\
|  \texttt{SO_SNDTIMEO}  \\
|  Timeout for output operations
\end{verbatim}

The socket options that can be consulted with \texttt{Unix.getsockopt\_float[25.1]} and modified with \texttt{Unix.setsockopt\_float[25.1]}. These options have a floating-point value representing a time in seconds. The value 0 means infinite timeout.

\begin{verbatim}
val getsockopt : file\_descr -> socket\_bool\_option -> bool  
  \hspace*{1em} Return the current status of a boolean-valued option in the given socket.
val setsockopt : file\_descr -> socket\_bool\_option -> bool -> unit  
  \hspace*{1em} Set or clear a boolean-valued option in the given socket.
val getsockopt\_int : file\_descr -> socket\_int\_option -> int  
  \hspace*{1em} Same as \texttt{Unix.getsockopt[25.1]} for an integer-valued socket option.
val setsockopt\_int : file\_descr -> socket\_int\_option -> int -> unit  
  \hspace*{1em} Same as \texttt{Unix.setsockopt[25.1]} for an integer-valued socket option.
val getsockopt\_optint : file\_descr -> socket\_optint\_option -> int\_option  
  \hspace*{1em} Same as \texttt{Unix.getsockopt[25.1]} for a socket option whose value is an \texttt{int option}.
val setsockopt\_optint :  
  \hspace*{1em} file\_descr -> socket\_optint\_option -> int\_option -> unit  
  \hspace*{1em} Same as \texttt{Unix.setsockopt[25.1]} for a socket option whose value is an \texttt{int option}.
val getsockopt\_float : file\_descr -> socket\_float\_option -> float  
  \hspace*{1em} Same as \texttt{Unix.getsockopt[25.1]} for a socket option whose value is a floating-point number.
val setsockopt\_float : file\_descr -> socket\_float\_option -> float -> unit  
  \hspace*{1em} Same as \texttt{Unix.setsockopt[25.1]} for a socket option whose value is a floating-point number.
val getsockopt\_error : file\_descr -> error\_option  
  \hspace*{1em} Return the error condition associated with the given socket, and clear it.
\end{verbatim}
High-level network connection functions

val open_connection : sockaddr -> in_channel * out_channel

Connect to a server at the given address. Return a pair of buffered channels connected to
the server. Remember to call flush[23.3] on the output channel at the right times to ensure
correct synchronization.

val shutdown_connection : in_channel -> unit

“Shut down” a connection established with Unix.open_connection[25.1]; 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 close_in[23.3].

val establish_server :
  (in_channel -> 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[25.1] never returns normally.

On Windows, it is not implemented. Use threads.

Host and protocol databases

type host_entry =
  {  h_name : string ;
      h_aliases : string array ;
      h_addrtype : socket_domain ;
      h_addr_list : inet_addr array ;
  }

Structure of entries in the hosts database.

type protocol_entry =
  {  p_name : string ;
      p_aliases : string array ;
      p_proto : int ;
  }

Structure of entries in the protocols database.

type service_entry =
  {  s_name : string ;
      s_aliases : string array ;
      s_port : int ;
      s_proto : string ;
  }

Structure of entries in the services database.
val gethostname : unit -> string
Return the name of the local host.

val gethostbyname : string -> host_entry
Find an entry in hosts with the given name.
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 : int -> protocol_entry
Find an entry in protocols with the given protocol number.
Raises Not_found if no such entry exist.

val getservbyname : string -> string -> service_entry
Find an entry in services with the given name.
Raises Not_found if no such entry exist.

val getservbyport : int -> string -> service_entry
Find an entry in services with the given service number.
Raises Not_found if no such entry exist.

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[25.1].

```haskell
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[25.1]

Options to Unix.getaddrinfo[25.1].

val getaddrinfo :
  string -> string -> getaddrinfo_option list -> addr_info list
getaddrinfo host service opts returns a list of Unix.addr_info[25.1] 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).

```
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[25.1].

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;
    Quit character (usually ctrl-\).
mutable c_verase : char;
    Erase character (usually DEL or ctrl-H).
mutable c_vkill : char;
    Kill line character (usually ctrl-U).
mutable c_veof : char;
    End-of-file character (usually ctrl-D).
mutable c_veol : char;
    Alternate end-of-line char. (usually none).
mutable c_vmin : int;
    Minimum number of characters to read before the read request is satisfied.
mutable c_vtime : int;
    Maximum read wait (in 0.1s units).
mutable c_vstart : char;
    Start character (usually ctrl-Q).
mutable c_vstop : char;
    Stop character (usually ctrl-S).
}
val tcgetattr : file_descr -> terminal_io
    Return the status of the terminal referred to by the given file descriptor. On Windows, not implemented.
type setattr_when =
  | TCSANOW
  | TCSADRAIN
  | TCSAFLUSH
val tcsetattr : file_descr -> setattr_when -> terminal_io -> unit
Set the status of the terminal referred to by the given file descriptor. The second argument indicates when the status change takes place: immediately (TCSANOW), when all pending output has been transmitted (TCSADRAIN), or after flushing all input that has been received but not read (TCSAFLUSH). TCSADRAIN is recommended when changing the output parameters; TCSAFLUSH, when changing the input parameters.

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 =
| TCIFLUSH
| TCOFLUSH
| TCI0FLUSH
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 =
| TCOOFF
| TCOON
| TCIOFF
| TCION
val tcflow : file_descr -> flow_action -> unit
Suspend or restart reception or transmission of data on the given file descriptor, depending on the second argument: TCOOFF suspends output, TCOON restarts output, TCIOFF transmits a STOP character to suspend input, and TCION transmits a START character to restart input.
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.
25.2 Module UnixLabels: labelized version of the interface

This module is identical to Unix (25.1), and only differs by the addition of labels. You may see these labels directly by looking at unixLabels.mli, or by using the ocamlbrowser tool.
Windows:
The Cygwin port of OCaml fully implements all functions from the Unix module. The native Win32 ports implement a subset of them. Below is a list of the functions that are not implemented, or only partially implemented, by the Win32 ports. Functions not mentioned are fully implemented and behave as described previously in this chapter.

<table>
<thead>
<tr>
<th>Functions</th>
<th>Comment</th>
</tr>
</thead>
<tbody>
<tr>
<td>fork</td>
<td>not implemented, use <code>create_process</code> or threads</td>
</tr>
<tr>
<td>wait</td>
<td>not implemented, use <code>waitpid</code></td>
</tr>
<tr>
<td>waitpid</td>
<td>can only wait for a given PID, not any child process</td>
</tr>
<tr>
<td>getppid</td>
<td>not implemented (meaningless under Windows)</td>
</tr>
<tr>
<td>nice</td>
<td>not implemented</td>
</tr>
<tr>
<td>truncate, ftruncate</td>
<td>not implemented</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 filesystem)</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 <code>[11]</code> (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 <code>socket_domain</code></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 26

The num library: arbitrary-precision rational arithmetic

The num library implements integer arithmetic and rational arithmetic in arbitrary precision. It was split off the core OCaml distribution starting with the 4.06.0 release, and can now be found at https://github.com/ocaml/num.

New applications that need arbitrary-precision arithmetic should use the Zarith library (https://github.com/ocaml/Zarith) instead of the Num library, and older applications that already use Num are encouraged to switch to Zarith. Zarith delivers much better performance than Num and has a nicer API.
Chapter 27

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";;
```

27.1 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.
\(..\) Grouping and naming of the enclosed expression.
\1 The text matched by the first \(..\) expression (\2 for the second expression,
and so on up to \9).
\b Matches word boundaries.
\Quotes 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:

```
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[27.1], but the regexp
  matches in a case-insensitive way.
String matching and searching

```plaintext
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[27.1], 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[27.1]
- Str.search_forward[27.1]
- Str.search_backward[27.1]
- Str.string_partial_match[27.1]
- Str.global_substitute[27.1]
- Str.substitute_first[27.1]

provided that none of the following functions was called inbetween:

- Str.global_replace[27.1]
- Str.replace_first[27.1]
- Str.split[27.1]
- Str.bounded_split[27.1]
- Str.split_delim[27.1]
- Str.bounded_split_delim[27.1]
```
• Str.full_split[27.1]
• Str.bounded_full_split[27.1]

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[27.1] 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[27.1] for details).

val matched_group : int -> string -> string

matched_group n s returns the substring of s that was matched by the nth group \(\ldots\) of the regular expression that was matched by the last call to a matching or searching function (see Str.matched_string[27.1] 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 nth 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 \(\text{(a\*)}\), 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 nth group of the regular expression that was matched by the last call to a matching or searching function (see Str.matched_string[27.1] for details).

Raises

• Not_found if the nth group of the regular expression was not matched.
• Invalid_argument if there are fewer than n groups in the regular expression.

val group_end : int -> int

group_end n returns the position of the character following the last character of substring that was matched by the nth group of the regular expression that was matched by the last call to a matching or searching function (see Str.matched_string[27.1] for details).

Raises
• **Not_found** if the \texttt{n}th group of the regular expression was not matched.
• **Invalid_argument** if there are fewer than \texttt{n} groups in the regular expression.

**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 \texttt{Str.global_replace\[27.1\]}, 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 \texttt{Str.global_substitute\[27.1\]}, 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
  \texttt{Str.matched_string\[27.1\]} for details). \texttt{s} must be the same string that was passed to the
  matching or searching function.
\end{verbatim}

**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,
  \texttt{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[27.1], 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[27.1] but occurrences of the delimiter at the beginning and at the end of the string are recognized and returned as empty strings in the result. For instance, `split_delim (regexp " ") " abc "` returns `["]"; "abc"; ""]`, while `split` with the same arguments returns `"abc"`.

val bounded_split_delim : regexp -> string -> int -> string list

Same as Str.bounded_split[27.1], but occurrences of the delimiter at the beginning and at the end of the string are recognized and returned as empty strings in the result.

type split_result =
  | Text of string
  | Delim of string

val full_split : regexp -> string -> split_result list

Same as Str.split_delim[27.1], but returns the delimiters as well as the substrings contained between delimiters. The former are tagged Delim in the result list; the latter are tagged Text. For instance, `full_split (regexp "[\{\}]\") "\{ab\}"` returns `[Delim "{"; Text "ab"; Delim "}"]`.

val bounded_full_split : regexp -> string -> int -> split_result list

Same as Str.bounded_split_delim[27.1], 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[27.1].

val last_chars : string -> int -> string

`last_chars s n` returns the last \( n \) characters of \( s \).
Chapter 28

The threads library

Warning: the threads library is deprecated since version 4.08.0 of OCaml. Please switch to system threads, which have the same API. Lightweight threads with VM-level scheduling are provided by third-party libraries such as Lwt, but with a different API.

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 -I +threads other options unix.cma threads.cma other files
  ocamlopt -I +threads other options unix.cmxa threads.cmxa other files
```

Compilation units that use the threads library must also be compiled with the -I +threads option (see chapter 9).

Programs that use VM-level threads must be compiled with the -vmthread option to ocamlc (see chapter 9), 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 9).
28.1 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[28.1].
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_write[28.1].

val wait_timed_write : Unix.file_descr -> float -> bool
  Suspend the execution of the calling thread until at least one character or EOF 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[28.1]. 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 waitsignal.

28.2 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):

```ml
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[28.2], 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.

28.3 Module Condition: Condition variables to synchronize between threads.

Condition variables are used when one thread wants to wait until another thread has finished doing something: the former thread 'waits' on the condition variable, the latter thread 'signals' the
condition when it is done. Condition variables should always be protected by a mutex. The typical use is (if D is a shared data structure, m its mutex, and c is a condition variable):

```ocaml
Mutex.lock m;
while (* some predicate P over D is not satisfied *) do
  Condition.wait c m
done;
(* Modify D *)
if (* the predicate P over D is now satisfied *) then Condition.signal c;
Mutex.unlock m
```

type t
The type of condition variables.

val create : unit -> t
Return a new condition variable.

val wait : t -> Mutex.t -> unit
wait c m atomically unlocks the mutex m and suspends the calling process on the condition variable c. The process will restart after the condition variable c has been signalled. The mutex m is locked again before wait returns.

val signal : t -> unit
signal c restarts one of the processes waiting on the condition variable c.

val broadcast : t -> unit
broadcast c restarts all processes waiting on the condition variable c.

### 28.4 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.

```ocaml
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[28.4]: 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.
28.5 Module ThreadUnix: Thread-compatible system calls.

The functionality of this module has been merged back into the Unix[25.1] module. Threaded programs can now call the functions from module Unix[25.1] directly, and still get the correct behavior (block the calling thread, if required, but do not block all threads in the process). Thread-compatible system calls.

Process handling

val execv : string -> string array -> unit
val execve : string -> string array -> string array -> unit
val execvp : string -> string array -> unit
val wait : unit -> int * Unix.process_status
val waitpid : Unix.wait_flag list -> int -> int * Unix.process_status
val system : string -> Unix.process_status

Basic input/output

val read : Unix.file_descr -> 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[28.5].
val timed_write : Unix.file_descr -> bytes -> int -> int -> float -> int
   Behave as ThreadUnix.read[28.5] and ThreadUnix.write[28.5], 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[28.5].

Polling

val select :
   Unix.file_descr list ->
   Unix.file_descr list ->
   Unix.file_descr list ->
   float -> Unix.file_descr list * Unix.file_descr list * Unix.file_descr list
Pipes and redirections

val pipe : ?cloexec:bool -> unit -> Unix.file_descr * Unix.file_descr
val open_process_in : string -> in_channel
val open_process_out : string -> out_channel
val open_process : string -> in_channel * out_channel

Time

val sleep : int -> unit

Sockets

val socket :
  ?cloexec:bool ->
  Unix.socket_domain -> Unix.socket_type -> int -> Unix.file_descr
val accept :
  ?cloexec:bool -> Unix.file_descr -> Unix.file_descr * Unix.sockaddr
val connect : Unix.file_descr -> Unix.sockaddr -> unit
val recv :
  Unix.filedescr -> 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 -> in_channel * out_channel
Chapter 29

The graphics library

The graphics library provides a set of portable drawing primitives. Drawing takes place in a separate window that is created when \texttt{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:

\begin{verbatim}
ocamlc other options graphics.cma other files
\end{verbatim}

For interactive use of the graphics library, do:

\begin{verbatim}
ocamlmktop -o mytop graphics.cma
./mytop
\end{verbatim}

or (if dynamic linking of C libraries is supported on your platform), start \texttt{ocaml} and type

\begin{verbatim}
#load "graphics.cma";;
\end{verbatim}

Here are the graphics mode specifications supported by \texttt{Graphics.open\_graph} on the X11 implementation of this library: the argument to \texttt{Graphics.open\_graph} has the format "display-name geometry", where \texttt{display-name} is the name of the X-windows display to connect to, and \texttt{geometry} is a standard X-windows geometry specification. The two components are separated by a space. Either can be omitted, or both. Examples:

\begin{verbatim}
Graphics.open_graph "foo:0"
\end{verbatim}

connects to the display \texttt{foo:0} and creates a window with the default geometry

\begin{verbatim}
Graphics.open_graph "foo:0 300x100+50-0"
\end{verbatim}

connects to the display \texttt{foo:0} and creates a window 300 pixels wide by 100 pixels tall, at location (50,0)

\begin{verbatim}
Graphics.open_graph " 300x100+50-0"
\end{verbatim}

connects to the default display and creates a window 300 pixels wide by 100 pixels tall, at location (50,0)

\begin{verbatim}
Graphics.open_graph ""
\end{verbatim}

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.

29.1 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
Chapter 29. The graphics library

Resize and erase the graphics window.

val clear_graph : unit -> unit
  Erase the graphics window.

val size_x : unit -> int
  See Graphics.size_y[29.1].

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. Some implementation (e.g. X Windows) represent coordinates by 16-bit integers, hence wrong clipping may occur with coordinates below -32768 or above 32676.

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[29.1].

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[29.1] 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 moves the current point to this point.

val curveto : int * int -> int * int -> int * int -> unit
    curveto b c d draws a cubic Bezier curve starting from the current point to point d, with control points b and c, and moves the current point to d.

val draw_rect : int -> int -> int -> int -> unit
    draw_rect x y w h draws the rectangle with lower left corner at x,y, width w and height h. The current point is unchanged. Raise Invalid_argument if w or h is negative.

val draw_poly_line : (int * int) array -> unit
**val draw_poly : (int * int) array -> unit**

`draw_poly` polygon draws the given polygon. The array contains the coordinates of the vertices of the polygon. The current point is unchanged.

**val draw_segments : (int * int * int * int) array -> unit**

`draw_segments` segments draws the segments given in the array argument. Each segment is specified as a quadruple \((x_0, y_0, x_1, y_1)\) where \((x_0, y_0)\) and \((x_1, y_1)\) are the coordinates of the end points of the segment. The current point is unchanged.

**val draw_arc : int -> int -> int -> int -> int -> int -> unit**

`draw_arc` \(x, y, rx, ry, a_1, a_2\) draws an elliptical arc with center \(x, y\), horizontal radius \(rx\), vertical radius \(ry\), from angle \(a_1\) to angle \(a_2\) (in degrees). The current point is unchanged. Raise `Invalid_argument` if \(rx\) or \(ry\) is negative.

**val draw_ellipse : int -> int -> int -> int -> unit**

`draw_ellipse` \(x, y, rx, ry\) draws an ellipse with center \(x, y\), horizontal radius \(rx\) and vertical radius \(ry\). The current point is unchanged. Raise `Invalid_argument` if \(rx\) or \(ry\) is negative.

**val draw_circle : int -> int -> int -> unit**

`draw_circle` \(x, y, r\) draws a circle with center \(x, y\) and radius \(r\). The current point is unchanged. Raise `Invalid_argument` if \(r\) is negative.

**val set_line_width : int -> unit**

Set the width of points and lines drawn with the functions above. Under X Windows, `set_line_width 0` selects a width of 1 pixel and a faster, but less precise drawing algorithm than the one used when `set_line_width 1` is specified. Raise `Invalid_argument` if the argument is negative.

**Text drawing**

**val draw_char : char -> unit**

See `Graphics.draw_string[29.1]`.

**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[29.1].

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[29.1].

val fill_circle : int -> int -> int -> unit
  Fill a circle with the current color. The parameters are the same as for Graphics.draw_circle[29.1].

Images

type image
  The abstract type for images, in internal representation. Externally, images are represented as matrices of colors. Images are bound to the current graphics window and should not be reused after closing this graphics window with Graphics.close_graph[29.1].

val transp : color
  In matrices of colors, this color represent a 'transparent' point: when drawing the corresponding image, all pixels on the screen corresponding to a transparent pixel in the image will not be modified, while other points will be set to the color of the corresponding point in the image. This allows superimposing an image over an existing background.

val make_image : color array array -> image
Convert the given color matrix to an image. Each sub-array represents one horizontal line. All sub-arrays must have the same length; otherwise, exception Graphic_failure is raised.

val dump_image : image -> color array array
  Convert an image to a color matrix.

val draw_image : image -> int -> int -> unit
  Draw the given image with lower left corner at the given point.

val get_image : int -> int -> int -> int -> image
  Capture the contents of a rectangle on the screen as an image. The parameters are the same as for Graphics.fill_rect.[29.1].

val create_image : int -> int -> image
  create_image w h returns a new image w pixels wide and h pixels tall, to be used in conjunction with blit_image. The initial image contents are random, except that no point is transparent.

val blit_image : image -> int -> int -> unit
  blit_image img x y copies screen pixels into the image img, modifying img in-place. The pixels copied are those inside the rectangle with lower left corner at x,y, and width and height equal to those of the image. Pixels that were transparent in img are left unchanged.

Mouse and keyboard events

type status =
  { mouse_x : int ;
    X coordinate of the mouse
    mouse_y : int ;
    Y coordinate of the mouse
    button : bool ;
    true if a mouse button is pressed
    keypressed : bool ;
    true if a key has been pressed
    key : char ;
    the character for the key pressed
  }
  To report events.

type event =
  | Button_down
A mouse button is pressed
  | Button_up
    A mouse button is released
  | Key_pressed
    A key is pressed
  | Mouse_motion
    The mouse is moved
  | Poll
    Don’t wait; return immediately
To specify events to wait for.

val wait_next_event : event list -> status
  Wait until one of the events specified in the given event list occurs, and return the status of
  the mouse and keyboard at that time. If Poll is given in the event list, return immediately
  with the current status. If the mouse cursor is outside of the graphics window, the mouse_x
  and mouse_y fields of the event are outside the range 0..size_x()-1, 0..size_y()-1. Keypresses
  are queued, and dequeued one by one when the Key_pressed event is specified and the Poll event
  is not 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.

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[29.1] below). Default display mode is on.

val remember_mode : bool -> unit

Set remember mode on or off. When turned on, drawings are done in the backing store; when turned off, the backing store is unaffected by drawings. This occurs independently of drawing onto the graphics window (see the function Graphics.display_mode[29.1] above). Default remember mode is on.
Chapter 30

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.

30.1 Module Dynlink: Dynamic loading of .cmo, .cma and .cmxs 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 itself register its entry points with the main program (or a previously-loaded library) e.g. by modifying tables of functions.
An exception will be raised if the given library defines toplevel modules whose names clash with modules existing either in the main program or a shared library previously loaded with `loadfile`. Modules from shared libraries previously loaded with `loadfile_private` are not included in this restriction.

The compilation units loaded by this function are added to the "allowed units" list (see `Dynlink.set_allowed_units[30.1]`).

```ml
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.

An exception will be raised if the given library defines toplevel modules whose names clash with modules existing in either the main program or a shared library previously loaded with `loadfile`. Modules from shared libraries previously loaded with `loadfile_private` are not included in this restriction.

An exception will also be raised if the given library defines toplevel modules whose name matches that of an interface depended on by a module existing in either the main program or a shared library previously loaded with `loadfile`. This applies even if such dependency is only a "module alias" dependency (i.e. just on the name rather than the contents of the interface).

The compilation units loaded by this function are not added to the "allowed units" list (see `Dynlink.set_allowed_units[30.1]`) since they cannot be referenced from other compilation units.

```ml
val adapt_filename : string -> string
```

In bytecode, the identity function. In native code, replace the last extension with `.cmxs`.

### Access control

```ml
val set_allowed_units : string list -> unit
```

Set the list of compilation units that may be referenced from units that are dynamically loaded in the future to be exactly the given value.

Initially all compilation units composing the program currently running are available for reference from dynamically-linked units. `set_allowed_units` 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.

Note that `Dynlink.loadfile[30.1]` changes the allowed-units list.

```ml
val allow_only : string list -> unit
```

`allow_only units` sets the list of allowed units to be the intersection of the existing allowed units and the given list of units. As such it can never increase the set of allowed units.

```ml
val prohibit : string list -> unit
```
prohibit units prohibits dynamically-linked units from referencing the units named in list
units by removing such units from the allowed units list. This can be used to prevent
access to selected units, e.g. private, internal modules of the running program.

val main_program_units : unit -> string list
  Return the list of compilation units that form the main program (i.e. are not dynamically
  linked).

val public_dynamically_loaded_units : unit -> string list
  Return the list of compilation units that have been dynamically loaded via loadfile (and
  not via loadfile_private). Note that compilation units loaded dynamically cannot be
  unloaded.

val all_units : unit -> string list
  Return the list of compilation units that form the main program together with those that
  have been dynamically loaded via loadfile (and not via loadfile_private).

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.

Error reporting

type linking_error = private
  | Undefined_global of string
  | Unavailable_primitive of string
  | Uninitialized_global of string
type error = private
  | 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
  | Cannot_open_dynamic_library of exn
  | Library's_module_initializers_failed of exn
  | Inconsistent_implementation of string
  | Module_already_loaded of string
  | Private_library_cannot_implement_interface of string

exception Error of error
Errors in dynamic linking are reported by raising the `Error` exception with a description of the error. A common case is the dynamic library not being found on the system: this is reported via `Cannot_open_dynamic_library` (the enclosed exception may be platform-specific).

```haskell
val error_message : error -> string
```

Convert an error description to a printable message.
Chapter 31

The bigarray library

The bigarray library has now been integrated into OCaml’s standard library.

The bigarray functionality may now be found in the standard library Bigarray module, except for the map_file function which is now part of the Unix library. The documentation has been integrated into the documentation for the standard library.

The legacy bigarray library bundled with the compiler is a compatibility library with exactly the same interface as before, i.e. with map_file included.

We strongly recommend that you port your code to use the standard library version instead, as the changes required are minimal.

If you choose to use the compatibility library, you must link your programs as follows:

```
ocamlc other options bigarray.cma other files
ocamlopt other options bigarray.cmxa other files
```

For interactive use of the bigarray compatibility library, do:

```
ocamlmkttop -o mytop bigarray.cma
./mytop
```

or (if dynamic linking of C libraries is supported on your platform), start ocaml and type #load "bigarray.cma";;.
Part V

Appendix
Index to the library

( * ), 400, 426
( ** ), 402, 428
( * . ), 401, 428
(&), 397, 423
(&&), 397, 423
(@), 407, 434
(@@), 399, 425
(!), 415, 441
(!=), 397, 423
(=), 396, 422
(==), 397, 423
(^), 405, 431
(^^), 417, 443
(>), 396, 422
(>=), 396, 422
(<), 396, 422
(<=), 396, 422
(<>), 399, 426
(-), 399, 426
(-.), 401, 428
(>), 399, 425
(||), 397, 424
(+), 399, 426
(+.), 401, 428
(/), 400, 426
(/.), 402, 428
(+-), 399, 425
(+-), 401, 427
(+-), 399, 425
(asr), 401, 427
(land), 400, 427
(lor), 400, 427
(lsl), 401, 427
(lsr), 401, 427
(lxor), 400, 427
(mod), 400, 426
(or), 397, 424
_angry_file_, 398, 424
_angry_line_, 398, 424
_angry_line_of_, 398, 425
abs, 400, 426
abs_float, 403, 430
absname, 476
absolute_path, 456
accept, 512, 540
access, 497
access_permission, 497
acos, 402, 429
adap_filenamelname, 552
add_arguments, 480
add_implementation, 271
add_implementation_binding, 271
add_ppx_context_sig, 453
add_ppx_context_str, 453
add_signature, 271
add_signature_binding, 271
add_use_file, 271
addr_info, 517
afl_inst_ratio, 480
afl_instrument, 479
alarm, 507
alert, 458
alert_reporter, 458
all_ccopts, 476
all_passes, 480
all_ppx, 476
all_units, 553
allow_only, 552
allow_unsafe_modules, 553
Already_displayed_error, 459
always, 538
annotations, 476
applcative_FUNCTORS, 477
apply, 452
apply_rewriters, 471
apply_rewriters_sig, 471
apply_rewriters_str, 471
Arg, 417, 444
arg_label, 454
arg_spec, 480
Array, 417, 444
array, 392
ArrayLabels, 417, 444
asin, 402, 429
Assert_failure, 394, 421
Assert_failure, 137, 393
ast_kind, 471
Ast_mapper, 449
Asttypes, 453
at_exit, 417, 444
atan, 403, 429
atan2, 403, 429
attribute, 461
attribute_of_warning, 452
attributes, 461
auto_synchronize, 549
background, 543
batch_mode_printer, 457
best_toplevel_printer, 457
Bigarray, 418, 444
binary_annotations, 477
bind, 512
binding_op, 464
black, 543
blit_image, 547
blue, 543
Bool, 418, 444
bool, 392
bool_of_string, 406, 432
bool_of_string_opt, 406, 432
bound_map, 271
bounded_full_split, 532
bounded_split, 531
bounded_split_delim, 532
broadcast, 537
Buffer, 418, 444
button_down, 548
bytecode_compatible_32, 476
Bytes, 418, 444
bytes, 391
BytesLabels, 418, 444
C_compiler, 477
call_external_preprocessor, 471
Callback, 418, 444
case, 463
cclambdas, 476
ceil, 403, 429
channel, 537
Char, 418, 444
char, 391
char_of_int, 405, 432
chdir, 499
check_nongen_schemes, 472
chmod, 497
choose, 538
chown, 497
chroot, 499
clambda_checks, 479
class_declaration, 467
class_description, 466
class_expr, 466
class_expr_desc, 466
class_field, 466
class_field_desc, 467
class_field_kind, 467
class_infos, 466
class_signature, 465
class_structure, 466
class_type, 465
class_type_declaration, 466
class_type_desc, 465
class_type_field, 465
class_type_field_desc, 466
classic, 476
classic_arguments, 476
classic_inlining, 479
classify_float, 405, 431
clear_close_on_exec, 499
clear_graph, 543
clear_nonblock, 498
Clflags, 475
close, 490
close_graph, 542
close_in, 414, 440
close_in_noerr, 414, 440
close_out, 412, 438
close_out_noerr, 412, 438
close_process, 502
close_process_full, 502
close_process_in, 502
close_process_out, 502
closed_flag, 454
closedir, 499
color, 480, 543
color_reader, 480
compare, 396, 422
compile_only, 476
Compiler_pass, 480
Complex, 418, 444
Condition, 536
connect, 512, 540
constant, 453, 460
constructor_arguments, 464
constructor_declaration, 464
copysign, 403, 430
core_type, 460, 461, 470
core_type_desc, 461
cos, 402, 429
cosh, 403, 429
create, 383, 534, 536, 537
create_image, 547
create_process, 500
create_process_env, 500
curr, 455
current_point, 544
current_x, 544
current_y, 544
curveto, 544
custom_runtime, 476
cyan, 543
debug, 476
debug_full, 476
debug_runavail, 478
decr, 415, 442
default_alert_reporter, 458
default_inline_alloc_cost, 478
default_inline_branch_cost, 479
default_inline_branch_factor, 479
default_inline_call_cost, 478
default_inline_indirect_cost, 479
default_inline_lifting_benefit, 479
default_inline_max_depth, 479
default_inline_max_unroll, 478
default_inline_prim_cost, 478
default_inline_threshold, 478
default_inline topLevel_threshold, 478
default_mapper, 451
default_report_printer, 457
default_simplify_rounds, 478
default_unbox_closures_factor, 479
default_warning_reporter, 458
delay, 534
Depend, 271
deprecated, 458
descr_of_in_channel, 491
descr_of_out_channel, 491
Digest, 418, 445
dir_handle, 499
direction_flag, 454
directive_argument, 470
directive_argument_desc, 470
display_mode, 549
Division_by_zero, 395, 421
Division_by_zero, 393
dlcode, 479
dllibs, 476
dllpaths, 477
domain_of_sockaddr, 511
dont_write_files, 479
draw_arc, 545
draw_char, 545
draw_circle, 545
draw_ellipse, 545
draw_image, 547
draw_poly, 545
draw_poly_line, 544
draw_rect, 544
draw_segments, 545
draw_string, 545
drop_ppx_context_sig, 453
drop_ppx_context_str, 453
dump_avail, 478
dump_clambda, 477
dump_cmm, 478
dump_combine, 478
dump_cse, 478
dump_flambda, 478
dump_flambda_let, 478
dump_flambda_verbose, 479
dump_image, 547
dump_instr, 478
dump_interf, 478
dump_interval, 478
dump_into_file, 480
dump_lambda, 477
dump_linear, 478
dump_live, 478
dump_parsetree, 477
dump_prefer, 478
dump_rawclambda, 477
dump_rawflambda, 477
dump_rawlambda, 477
dump_regalloc, 478
dump_reload, 478
dump_scheduling, 478
dump_selection, 478
dump_source, 477
dump_spill, 478
dump_split, 478
dump_typedtree, 477
dumped_pass, 480
dup, 498
dup2, 498
Dynlink, 551
echo_eof, 455
enabled, 383
End_of_file, 395, 421
End_of_file, 393
env_reader, 480
environment, 486
Ephemeron, 418, 445
epsilon_float, 404, 431
Error, 459, 471, 474, 553
error, 459, 471, 474, 486, 553
Error_forward, 474
error_message, 486, 554
error_of_exn, 459
error_of_printer, 459
error_of_printer_file, 459
error_size, 477
error_style, 480
error_style_reader, 480
errorf, 459
establish_server, 516
Event, 537
event, 537, 548
execv, 487, 539
execve, 487, 539
execvp, 487, 539
execvpe, 487
Exit, 394, 421
exit, 417, 444, 534
exn, 392
exp, 402, 428
expm1, 402, 428
expression, 460, 462, 470
expression_desc, 463
extension, 461
extension_constructor, 465
extension_constructor_kind, 465
extension_of_error, 452
Failure, 393, 395, 421
failwith, 394, 420
fchmod, 497
fchown, 497
file, 471
file_descr, 488
file_kind, 492
file_perm, 490
Filename, 418, 445
fill_arc, 546
fill_circle, 546
fill_ellipse, 546
fill_poly, 546
fill_rect, 546
first_chars, 532
flambda_invariant_checks, 479
flatten, 460
Float, 418, 445
float, 392, 404, 430
Float_arg_helper, 475
float_const_prop, 477
float_of_int, 404, 430
float_of_string, 407, 433
float_of_string_opt, 406, 433
floor, 403, 429
flow_action, 522
flush, 410, 437
flush_all, 411, 437
flush_queue, 522
for_package, 477
force, 121
force_slash, 479
foreground, 543
fork, 487
Format, 418, 445
format, 416, 443
format_of_string, 417, 443
format4, 392, 416, 443
format6, 416, 443
formatter_for_warnings, 458
fpclass, 405, 431
free_structure_names, 271
frexp, 404, 430
fst, 407, 433
fstat, 493, 495
fsync, 490
ftruncate, 492, 494
full_split, 532
Fun, 418, 445
Gc, 418, 445
Genlex, 418, 445
get, 475
get_cookie, 453
get_image, 547
get_pos_info, 455
getaddrinfo, 518
getaddrinfo_option, 518
getcwd, 499
getegid, 508
getenv, 486
geteuid, 508
getgid, 508
getgrgid, 510
getgrnam, 509
group, 477
getgroups, 509
gethostbyaddr, 517
gethostbyname, 517
gethostname, 517
getitimer, 508
getlogin, 509
getnameinfo, 519
getnameinfo_option, 519
getpeername, 512
getppid, 488
getpid, 488
getppid, 488
getprotobynumber, 517
getprotobynumber, 517
getpwnam, 509
getpwuid, 510
getservbyname, 517
getservbyname, 517
getservbyport, 517
getsockname, 512
getsockopt, 515
getsockopt_error, 515
getsockopt_float, 515
getsockopt_int, 515
getsockopt_optint, 515
gmtime, 506
gettimeofday, 506
getuid, 508
has_symlink, 503
Hashtbl, 418, 445
highlight_terminfo, 456
Hooks, 474
host_entry, 516
hypot, 403, 429
Graphic_failure, 542
Graphics, 542
green, 543
group_beginning, 530
group_end, 530
group_entry, 509
guard, 538
handle_unix_error, 486
Graphics, 542
host_entry, 516
hypot, 403, 429
id, 534
ignore, 405, 432
image, 546
implementation, 460
ImplementationHooks, 471, 474
in_channel, 407, 434
in_channel_length, 414, 440, 441
in_channel_of_descr, 491
in_file, 455
include_declaration, 468
include_description, 468
include_dirs, 476
include_infos, 468
incr, 415, 441
inet_addr, 510
inet_addr_any, 510
inet_addr_loopback, 510
inet_addr_of_string, 510
inet6_addr_any, 510
inet6_addr_loopback, 510
infinity, 404, 430
init, 455
init_file, 477
initgroups, 509
initial_env, 473
inline_alloc_cost, 479
inline_branch_cost, 479
inline_branch_factor, 479
inline_call_cost, 479
inline_indirect_cost, 479
inline_lifting_benefit, 479
inline_max_depth, 479
inline_max_unroll, 478
inline_prim_cost, 479
inline_threshold, 478
inline_toplevel_threshold, 478
inlining_arguments, 476
inlining_report, 478
input, 413, 439
input_binary_int, 413, 440
input_byte, 413, 440
input_char, 413, 439
input_lexbuf, 455
input_line, 413, 439
input_name, 455
input_value, 413, 440
Int, 418, 445
int, 391
Int_arg_helper, 475
int_of_char, 405, 432
int_of_float, 404, 430
int_of_string, 406, 433
int_of_string_opt, 406, 432
Int32, 418, 445
int32, 392
int64, 392
interface, 460
InterfaceHooks, 471, 474
interval_timer, 507
interval_timer_status, 508
invalid_arg, 394, 420
Invalid_argument, 395, 421
Invalid_argument, 393
is_native, 551
is_tail_native_heuristic, 474
isatty, 493
join, 534
keep_asm_file, 478
keep_camlprimc_file, 478
keep_docs, 479
keep_locs, 479
keep_startup_file, 478
key_pressed, 548
kill, 505, 534
label, 454
label_declaration, 464
LargeFile, 414, 441, 494
last, 460
last_chars, 532
Lazy, 419, 445
Lazy (module), 121
lazy_t, 392
ldexp, 404, 430
Lexing, 419, 445
lineto, 544
link, 496
link_everything, 476
linking_error, 553
List, 419, 445
list, 392
listen, 512
ListLabels, 419, 445
lnot, 400, 427
loadfile, 551
loadfile_private, 552
loc, 454, 455
localtime, 507
Location, 454
lock, 536
lock_command, 504
lockf, 504
log, 402, 428
log10, 402, 428
log1p, 402, 428
Longident, 460
longident, 470
loop_at_exit, 548
lseek, 492, 494
lstat, 493, 495

magenta, 543
main_program_units, 553
make_archive, 476
make_image, 546
make_leaf, 271
make_node, 271
make_package, 477
make_runtime, 477
Map, 419, 445
map_file, 495
map_opt, 452
map_tree, 271
mapper, 451
Marshal, 419, 445
match_beginning, 530
match_context_rows, 476
match_end, 530
Match_failure, 394, 421
matched_group, 530
matched_string, 529
Match_failure, 127, 129, 130, 393
max, 397, 423

max_float, 404, 431
max_int, 400, 426
min, 396, 423
min_float, 404, 431
min_int, 400, 426
mkdir, 499
mkfifo, 500
mkloc, 455
mknoloc, 455
mktime, 507
mod_float, 403, 430
modf, 404, 430
modtype_of_package, 472
module_binding, 470
module_declaration, 468
module_expr, 469
module_expr_desc, 469
module_substitution, 468
module_type, 467
module_type_declaration, 468
module_type_desc, 467
MoreLabels, 419, 445
mouse_pos, 548
moveto, 544
msg, 456
msg_flag, 513
mutable_flag, 454
Mutex, 536

name_info, 518
nan, 404, 431
native_code, 478
Nativeint, 419, 445
nativeint, 392
neg_infinity, 404, 430
new_channel, 537
nice, 488
no_auto_link, 477
no_check_prims, 476
no_std_include, 476
noassert, 477
noinit, 477
none, 455
nopervasives, 476
noprompt, 477
INDEX TO THE LIBRARY

pos_out, 412, 414, 438, 441  putenv, 487
pp_deps, 271  Queue, 419, 446
Pparse, 471  quote, 528
Pprintast, 470  raise, 394, 420
pred, 399, 426  raise_errorf, 459
preprocess, 471  raise_notrace, 394, 420
preprocessor, 476  Random, 419, 446
prerr_alert, 458  read, 490, 539
prerr_bytes, 408, 435  read_ast, 471
prerr_char, 408, 435  read_float, 409, 436
prerr_endline, 409, 435  read_float_opt, 409, 436
prerr_float, 409, 435  read_int, 409, 436
prerr_int, 409, 435  read_int_opt, 409, 435
prerr_newline, 409, 435  read_key, 548
prerr_string, 408, 435  read_line, 409, 435
prerr_warning, 458  readdr, 499
principal, 477  readlink, 503
print_alert, 458  real_paths, 477
print_arguments, 480  really_input, 413, 439
print_bytes, 408, 434  really_input_string, 413, 439
print_char, 408, 434  rec_flag, 454
print_endline, 408, 435  receive, 538
print_filename, 456  recursive_types, 477
print_float, 408, 434  recv, 513, 540
print_int, 408, 434  recvfrom, 513, 540
print_loc, 456  red, 543
print_locs, 456  ref, 415, 441
print_newline, 408, 435  regexp, 527
print_report, 457  regexp_case_fold, 528
print_string, 408, 434  regexp_string, 528
print_types, 476  regexp_string_case_fold, 528
print_warning, 458  register, 452
Printexc, 419, 446  register_error_of_exn, 459
Printf, 419, 446  register_function, 452
private_flag, 454  remember_mode, 549
process_full_pid, 502  remove_preprocessed, 471
process_in_pid, 501  remove_unused_arguments, 479
process_out_pid, 502  rename, 496
process_pid, 502  replace_first, 531
process_status, 487  replace_matched, 531
process_times, 506  report, 456
profile_columns, 479  report_alert, 458
prohibit, 552  report_error, 471, 474
protocol_entry, 516  report_exception, 459
public_dynamically_loaded_units, 553
report_kind, 456
report_printer, 457
report_warning, 457
reset, 455
reset_arguments, 480
resize_window, 542
Result, 419, 446
result, 415, 442
rewinddir, 499
rewrite_absolute_path, 456
rgb, 543
rhs_interval, 455
rhs_loc, 455
rlineto, 544
rmdir, 499
rmoveto, 544
rounds, 478
row_field, 461
row_field_desc, 461
run_main, 452
runtime_variant, 479
save_and_close, 383
save_event, 383
save_event_for_automatic_snapshots, 384
save_signature, 472
Scanf, 419, 446
search_backward, 529
search_forward, 529
seek_command, 491
seek_in, 414, 440, 441
seek_out, 411, 414, 438, 441
select, 503, 535, 538, 539
self, 534
send, 513, 538, 540
send_substring, 513, 540
sendto, 513, 540
sendto_substring, 513, 540
Seq, 419, 446
Series, 383
service_entry, 516
Set, 419, 446
set_allowed_units, 552
set_binary_mode_in, 414, 440
set_binary_mode_out, 412, 438
set_close_on_exec, 498
set_color, 543
set_cookie, 453
set_dumped_pass, 480
set_font, 545
set_line_width, 545
set_nonblock, 498
set_text_size, 546
set_window_title, 542
setattr_when, 521
setgid, 508
setgroups, 509
setitimer, 508
setsid, 522
setsockopt, 515
setsockopt_float, 515
setsockopt_int, 515
setsockopt_optint, 515
setuid, 508
shared, 479
should_stop_after, 480
show_filename, 456
shutdown, 512
shutdown_command, 512
shutdown_connection, 516
Sig_component_kind, 473
sigmask, 535
signal, 537
signature, 467, 470
signature_item, 467
signature_item_desc, 468
Signature_names, 472
sigpending, 505
sigprocmask, 505
sigprocmask_command, 505
sigsuspend, 505
Simplif, 474
simplify, 472
simplify_lambda, 474
simplify_rounds, 478
sin, 402, 429
single_write, 490
single_write_substring, 490
sinh, 403, 429
size_x, 543
size_y, 543
sleep, 507, 540
sleepf, 507
Snapshot, 384
snd, 407, 433
sockaddr, 511
socket, 511, 540
socket_bool_option, 514
socket_domain, 511
socket_float_option, 515
socket_int_option, 514
socket_optint_option, 514
socket_type, 511
socketpair, 512
sound, 549
space_formatter, 470
Spacetime, 382, 420, 446
split, 531
split_default_wrapper, 474
split_delim, 532
split_result, 532
sqrt, 402, 428
Stack, 420, 446
Stack_overflow, 395, 421
Stack_overflow, 393
stat, 493, 495
stats, 493, 495
status, 547
std_include_dir, 479
std_include_flag, 479
stderr, 408, 434, 489
stdin, 407, 434, 488
StdLabels, 420, 446
Stdlib, 394, 420
stdout, 408, 434, 488
stop_after, 480
Str, 527
Stream, 420, 446
strict_formats, 477
strict_sequence, 477
String, 271, 420, 446
string, 392
string_after, 532
string_before, 532
string_match, 529
string_of_bool, 406, 432
string_of_expression, 470
string_of_float, 406, 433
string_of_format, 416, 443
string_of_inet_addr, 510
string_of_int, 406, 432
string_of_structure, 470
string_partial_match, 529
StringLabels, 420, 446
structure, 469, 470
structure_item, 469
structure_item_desc, 469
substitute_first, 531
succ, 399, 426
symbol_gloc, 455
symbol_rloc, 455
symlink, 502
sync, 538
synchronize, 549
Sys, 420, 446
Sys_blocked_io, 395, 422
Sys_error, 395, 421
Sys_blocked_io, 394
Sys_error, 393
system, 488, 539
t, 383, 455, 460, 472, 473, 480, 534, 536, 537
take, 384
tan, 402, 429
tanh, 403, 429
tcdrain, 522
tcflow, 522
tcflush, 522
tcgetattr, 521
tcsendbreak, 522
tcsetattr, 521
terminal_io, 521
terminfo_toplevel_printer, 457
text_size, 546
Thread, 534
ThreadUnix, 539
time, 506
timed_read, 539
timed_write, 539
timed_write_substring, 539
# Index of Keywords

<table>
<thead>
<tr>
<th>Keyword</th>
<th>Page Numbers</th>
</tr>
</thead>
<tbody>
<tr>
<td>and</td>
<td>124, 138, 143, 146, 147, 152, 158, 164</td>
</tr>
<tr>
<td>as</td>
<td>115, 116, 119, 120, 143, 145</td>
</tr>
<tr>
<td>asr</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>assert</td>
<td>137</td>
</tr>
<tr>
<td>begin</td>
<td>118, 124, 126</td>
</tr>
<tr>
<td>class</td>
<td>146, 147, 150, 152, 153</td>
</tr>
<tr>
<td>constraint</td>
<td>138, 140–143, 146</td>
</tr>
<tr>
<td>do</td>
<td>see while, for</td>
</tr>
<tr>
<td>done</td>
<td>see while, for</td>
</tr>
<tr>
<td>downto</td>
<td>see for</td>
</tr>
<tr>
<td>else</td>
<td>see if</td>
</tr>
<tr>
<td>end</td>
<td>118, 124, 126, 141, 143, 147, 148, 152</td>
</tr>
<tr>
<td>exception</td>
<td>140, 147, 149, 152, 153</td>
</tr>
<tr>
<td>external</td>
<td>147, 148, 152, 153</td>
</tr>
<tr>
<td>false</td>
<td>118</td>
</tr>
<tr>
<td>for</td>
<td>124, 131</td>
</tr>
<tr>
<td>fun</td>
<td>124, 125, 127, 143, 163</td>
</tr>
<tr>
<td>function</td>
<td>124, 125, 127</td>
</tr>
<tr>
<td>functor</td>
<td>147, 151, 152, 154</td>
</tr>
<tr>
<td>if</td>
<td>124, 125, 130</td>
</tr>
<tr>
<td>in</td>
<td>see let</td>
</tr>
<tr>
<td>include</td>
<td>147, 151, 152, 154, 166</td>
</tr>
<tr>
<td>inherit</td>
<td>141–143, 145</td>
</tr>
<tr>
<td>initializer</td>
<td>143, 146</td>
</tr>
<tr>
<td>land</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>lazy</td>
<td>121, 124, 137</td>
</tr>
<tr>
<td>let</td>
<td>124, 125, 129, 137, 138, 143, 152, 153, 162</td>
</tr>
<tr>
<td>lor</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>lsl</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>lsr</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>lxor</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>match</td>
<td>124, 125, 130, 171</td>
</tr>
<tr>
<td>method</td>
<td>141–143, 145, 146</td>
</tr>
<tr>
<td>mod</td>
<td>113, 125, 133</td>
</tr>
<tr>
<td>module</td>
<td>137, 147, 150, 152–154, 158, 164, 166, 167, 169</td>
</tr>
<tr>
<td>open</td>
<td>138</td>
</tr>
<tr>
<td>mutable</td>
<td>138, 139, 141–143, 145</td>
</tr>
<tr>
<td>new</td>
<td>124, 134</td>
</tr>
<tr>
<td>nonrec</td>
<td>138</td>
</tr>
<tr>
<td>object</td>
<td>124, 135, 141, 143</td>
</tr>
<tr>
<td>of</td>
<td>117, 138, 166</td>
</tr>
<tr>
<td>open!</td>
<td>170</td>
</tr>
<tr>
<td>or</td>
<td>124, 125, 131</td>
</tr>
<tr>
<td>private</td>
<td>141–143, 145, 146, 160, 161</td>
</tr>
<tr>
<td>rec</td>
<td>see let, module</td>
</tr>
<tr>
<td>sig</td>
<td>147, 148</td>
</tr>
<tr>
<td>struct</td>
<td>152</td>
</tr>
<tr>
<td>then</td>
<td>see if</td>
</tr>
<tr>
<td>to</td>
<td>see for</td>
</tr>
<tr>
<td>true</td>
<td>118</td>
</tr>
<tr>
<td>try</td>
<td>124, 125, 131</td>
</tr>
<tr>
<td>type</td>
<td>138, 147, 149, 150, 152–154, 163, 166, 167, 171</td>
</tr>
<tr>
<td>val</td>
<td>141–143, 145, 147, 148, 164</td>
</tr>
<tr>
<td>virtual</td>
<td>see val, method, class</td>
</tr>
<tr>
<td>when</td>
<td>124, 128, 177</td>
</tr>
<tr>
<td>while</td>
<td>131</td>
</tr>
<tr>
<td>with</td>
<td>124, 147, 151, 164, 167</td>
</tr>
</tbody>
</table>