The Caml Light system release 0.7

Documentation and user's manual

Xavier Leroy

July 3, 1995

Contents

Foreword

This manual documents the release 0.7 of the Caml Light system. It is organized as follows.

- Part I, “Getting started”, explains how to install Caml Light on your machine.
- Part II, “The Caml Light language reference manual”, is the reference description of the Caml Light language.
- Part IV, “The Caml Light library”, describes the modules provided in the standard library.
- Part III, “The Caml Light commands”, documents the Caml Light compiler, toplevel system, and programming utilities.
- Part V, “Appendix”, contains a short bibliography, an index of all identifiers defined in the standard library, and an index of Caml Light keywords.

Conventions

The Caml Light system comes in several versions: for Unix machines, for Macintoshes, and for PCs. The parts of this manual that are specific to one version are presented as shown below:

Unix: This is material specific to the Unix version.

Mac: This is material specific to the Macintosh version.

PC: This is material specific to the PC version.

License

Availability by FTP

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

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

Part I

Getting started

Chapter 1

Installation instructions

This chapter explains how to install Caml Light on your machine.

1.1 The Unix version

Requirements. Any machine that runs under one of the various flavors of the Unix operating system, and that has a flat, non-segmented, 32-bit or 64-bit address space. 4M of RAM, 2M of free disk space. The graphics library requires X11 release 4 or later.

Installation. The Unix version is distributed in source format, as a compressed `tar` file named `cl7unix.tar.Z`. To extract, move to the directory where you want the source files to reside, transfer `cl7unix.tar.Z` to that directory, and execute

```
zcat cl7unix.tar.Z | tar xBf -
```

This extracts the source files in the current directory. The file `INSTALL` contains complete instructions on how to configure, compile and install Caml Light. Read it and follow the instructions.

Troubleshooting. See the file `INSTALL`.

1.2 The Macintosh version

Requirements. Any Macintosh with at least 1M of RAM (2M is recommended), running System 6 or 7. About 850K of free space on the disk. The parts of the Caml Light system that support batch compilation currently require the Macintosh Programmer's Workshop (MPW) version 3.2. MPW is Apple's development environment, and it is distributed by APDA, Apple's Programmers and Developers Association. See the file `READ ME` in the distribution for APDA's address.

Installation. Create the folder where the Caml Light files will reside. Double-click on the file `cl7macbin.sea` from the distribution. This displays a file dialog box. Open the folder where the Caml Light files will reside, and click on the `Extract` button. This will re-create all files from the distribution in the Caml Light folder.

To test the installation, double-click on the application `Caml Light`. The "Caml Light output" window should display something like

```
>       Caml Light version 0.7

#
```

In the “Caml Light input” window, enter `1+2;;` and press the `Return` key. The “Caml Light output” window should display:

```
>       Caml Light version 0.7

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

Select “Quit” from the “File” menu to return to the Finder.

If you have MPW, you can install the batch compilation tools as follows. The tools and scripts from the `tools` folder must reside in a place where MPW will find them as commands. There are two ways to achieve this result: either copy the files in the `tools` folder to the `Tools` or the `Scripts` folder in your MPW folder; or keep the files in the `tools` folder and add the following line to your `UserStartup` file (assuming Caml Light resides in folder `Caml Light` on the disk named `My HD`):

```
Set Commands "{Commands},My HD:Caml Light:tools:"
```

In either case, you now have to edit the `camlc` script, and replace the string

```
Macintosh HD:Caml Light:lib:
```

(in the first line) with the actual pathname of the `lib` folder. For example, if you put Caml Light in folder `Caml Light` on the disk named `My HD`, the first line of `camlc` should read:

```
Set stdlib "My HD:Caml Light:lib:"
```

Troubleshooting. Here is one commonly encountered problem.

`Cannot find file stream.zi`

(Displayed in the “Caml Light output” window, with an alert box telling you that Caml Light has terminated abnormally.) This is an installation error. The folder named `lib` in the distribution must always be in the same folder as the `Caml Light` application. It’s OK to move the application to another folder; but remember to move the `lib` directory to the same folder. (To return to the Finder, first select “Quit” from the “File” menu.)

1.3 The PC version

Requirements. A PC equipped with a 80386, 80486 or Pentium processor, running MS-DOS version 3.3 or later. Older Intel processors (8086, 80286) are no longer supported. 2M of RAM. About 1.2M of free space on the disk. The graphics primitives require a VGA or SuperVGA video card.

Installation. In the following, we assume that the distribution files resides in drive `A:`, and that the hard disk on which you are installing Caml Light is drive `C:`. If this is not the case, replace `A:` and `C:` by the appropriate drives.

Create a directory on the hard disk where Caml Light will reside. In the following, we assume that this directory is named `C:\caml`. If you choose a different directory, replace `C:\caml` in the following by the appropriate absolute path name. Then, execute the following commands:

```
cd C:\caml
A:pkunzip -d A:cl7pc
```

(Be careful not to omit the `-d` option to `pkunzip`.) This extracts all files in the `C:\caml` directory.

Select or create a directory where Caml Light will put its temporary files. Many machines already have a `C:\tmp` directory for that purpose. If it does not exist, create it.

For the remainder of the configuration process, you will have to determine two things:

- does your machine contain floating-point hardware — that is, a 387 coprocessor, a 486DX or Pentium processor, or a 487SX (co-)processor? (If you really don't know, assume no floating-point hardware.)
- what kind of SuperVGA card do you have? Caml Light has graphics primitives that work on any VGA card in 320x200 pixels, 256 colors, but it can take advantage of the extra possibilities of various SuperVGA cards to work at higher resolution. To do so, you must determine which chipset is used in your SuperVGA card. Re-read the documentation for the card, then look at the files with extension `.GRD` or `.GRN` (the graphics drivers) in directory `C:\caml\dev`, and find one whose name seems to match the name of the chipset. If you can't determine which graphics driver to use, don't worry: you'll stick with the default VGA graphics, that's all.

Now, edit the `C:\autoexec.bat` file, in order to:

- Add `C:\caml\bin` to the `PATH` variable; that is, transform the line that reads

  ```
  SET PATH=C:\dos;...
  ```

  into

  ```
  SET PATH=C:\dos;...;C:\caml\bin
  ```

- Insert the following lines

  ```
  SET CAMLLIB=C:\caml\lib
  SET GO32TMP=C:\tmp
  ```

- If your machine has floating-point hardware, insert the following line:

  ```
  SET GO32=driver C:\caml\dev\graph.grd gw 640 gh 480
  ```

where `graph.grd` stands for the name of the graphics driver for your SuperVGA card, as determined above. The `640` and `480` specify the default graphics resolution to use; you can put `800` and `600`, or `1024` and `768` instead, depending on your taste and the capabilities of your card.

If you were unable to determine the correct graphics driver, do not insert anything, leaving the `GO32` variable undefined.

- If your machine has no floating-point hardware, insert the following line:

  ```
  SET GO32=emu C:\caml\dev\emu387 driver C:\caml\dev\graph.grd gw 640 gh 480
  ```

  where `graph.grd` stands for the name of the graphics driver for your SuperVGA card, as determined above. As explained in the previous item, you can choose another default graphics resolution instead of `640` and `480`.

  If you were unable to determine the correct graphics driver, insert the following line instead:

  ```
  SET GO32=emu C:\caml\dev\emu387
  ```

Then, save the `autoexec.bat` file and restart the machine. To test the installation, execute:

```
camlc -v
```

The `camlc` command should print something like:

```
The Caml Light system for the 80386 PC, version 0.7
  (standard library from C:\caml\lib)
The Caml Light runtime system, version 0.7
The Caml Light compiler, version 0.7
The Caml Light linker, version 0.7
```

Then, execute:

```
caml
```

The `caml` command should print something like:

```
>       Caml Light version 0.7

#
```

In response to the `#` prompt, type:

```
quit();;
```

This should get you back to the DOS command interpreter.

Troubleshooting. Here are some commonly encountered problems.

`Cannot find the bytecode file` **or** `camlrun.exe: No such file`

The installation has been performed incorrectly. Double-check the `autoexec.bat` file for errors in setting the `PATH` and `CAMLLIB` variables.

`CPU must be a 386 to run this program`

Self-explanatory. It's about time you buy a 386 PC or better.

`CPU must be in REAL mode (not V86 mode) to run this program`

Ah. That's a tricky one. A number of utility programs switch the 80386 processor to a particular mode, called "Virtual 86" or "V86" mode, that prevents 32-bit protected-mode applications like the 80386 PC version of Caml Light from starting. These programs include:

- device drivers that provide memory management services
- device drivers that provide enhanced debugging possibilities, such as `tdh386.sys` from Turbo Debugger.

The 80386 PC version cannot start when any of these programs is active. To solve the problem, remove the guilty device drivers from your `config.sys` file.

On the other hand, the 80386 PC version knows how to cohabit with DPMI-compliant or VCPI-compliant environments and memory managers. These include Windows 3, Desqview, and the `EMM386`, `QEMM386` and `386MAX` memory managers. In the case of `EMM386.EXE` from the MS-DOS distribution, EMS must be enabled (do not give it the `NOEMS` option). If you run the 80386 PC under a VCPI-compliant memory manager, configure the memory manager so that it allocates at least 1M of EMS, and preferably 2M or more.

Caml Light runs slowly and does a lot of disk accesses

When Caml Light cannot allocate the RAM it requires, it starts paging to a disk file, which considerably slows down execution. To avoid this, make sure that at least 1M or memory is available to Caml Light, and preferably 2M. Caml Light uses XMS memory if you are not running under a VCPI-compliant memory manager, and EMS memory if you are running under a VCPI-compliant memory manager. In the latter case, make sure to configure the memory manager so that it can allocate enough EMS.

Part II

The Caml Light language reference manual

Chapter 2

The core Caml Light language

Foreword

This document is intended as a reference manual for the Caml Light language. It lists all 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 the language, as provided by the companion tutorial *Functional programming using Caml Light*, 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. The reader interested in truly formal descriptions of languages from the ML family is referred to *The definition of Standard ML* and *Commentary on Standard ML*, by Milner, Tofte and Harper, MIT Press.

Warning

Several implementations of the Caml Light language are available, and they evolve at each release. Consequently, this document carefully distinguishes the language and its implementations. Implementations can provide extra language constructs; moreover, all points left unspecified in this reference manual can be interpreted differently by the implementations. The purpose of this reference manual is to specify those features that all implementations must provide.

Notations

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

2.1 Lexical conventions

Blanks

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

Comments

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

Identifiers

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

Identifiers are sequences of letters, digits and `_` (the underscore character), starting with a letter. Letters contain at least the 52 lowercase and uppercase letters from the ASCII set. Implementations can recognize as letters other characters from the extended ASCII set. Identifiers cannot contain two adjacent underscore characters (`__`). Implementation may limit the number of characters of an identifier, but this limit must be above 256 characters. All characters in an identifier are meaningful.

Integer literals

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

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

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

(The initial `0` is the digit zero; the `O` for octal is the letter O.)

Floating-point literals

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

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

Character literals

char-literal ::= `` ` `` *regular-char* `` ` ``
  | `` ` `` `\` (`\` | `` ` `` | `n` | `t` | `b` | `r`) `` ` ``
  | `` ` `` `\` (`0`...`9`) (`0`...`9`) (`0`...`9`) `` ` ``

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

Sequence	Character denoted
`\\`	backslash (\)
`` \` ``	backquote (`)
`\n`	newline (LF)
`\r`	return (CR)
`\t`	horizontal tabulation (TAB)
`\b`	backspace (BS)
ddd	the character with ASCII code *ddd* in decimal

String literals

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

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

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

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

Implementations must support string literals up to $2^{16} - 1$ characters in length (65535 characters).

Keywords

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

```
and    as    begin      do     done     downto
else   end   exception  for    fun      function
if     in    let        match  mutable  not
of     or    prefix     rec    then     to
try    type  value      where  while    with
```

The following character sequences are also keywords:

```
#    !    !=   &    (    )    *    *.   +    +.
,    -    -.   ->   .    .(   /    /.   :    ::
:=   ;    ;;   <    <.   <-   <=   <=.  <>   <>.
=    =.   ==   >    >.   >=   >=.  @    [    [|
]    ^    _    __   {    |    |]   }    '
```

Ambiguities

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

2.2 Global names

Global names are used to denote value variables, value constructors (constant or non-constant), type constructors, and record labels. Internally, a global name consists of two parts: the name of the defining module (the module name), and the name of the global inside that module (the local name). The two parts of the name must be valid identifiers. Externally, global names have the following syntax:

$$\begin{array}{rcl} \textit{global-name} & ::= & \textit{ident} \\ & \mid & \textit{ident}\ \texttt{\_\_}\ \textit{ident} \end{array}$$

The form *ident* `__` *ident* is called a qualified name. The first identifier is the module name, the second identifier is the local name. The form *ident* is called an unqualified name. The identifier is the local name; the module name is omitted. The compiler infers this module name following the completion rules given below, therefore transforming the unqualified name into a full global name.

To complete an unqualified identifier, the compiler checks a list of modules, the opened modules, to see if they define a global with the same local name as the unqualified identifier. When one is found, the identifier is completed into the full name of that global. That is, the compiler takes as module name the name of an opened module that defines a global with the same local name as the unqualified identifier. If several modules satisfy this condition, the one that comes first in the list of opened modules is selected.

The list of opened modules always includes the module currently being compiled (checked first). (In the case of a toplevel-based implementation, this is the module where all toplevel definitions are entered.) It also includes a number of standard library modules that provide the initial environment (checked last). In addition, the `#open` and `#close` directives can be used to add or remove modules from that list. The modules added with `#open` are checked after the module currently being compiled, but before the initial standard library modules.

variable	::=	*global-name*
	|	`prefix` *operator-name*
operator-name	::=	`+` | `-` | `*` | `/` | `mod` | `+.` | `-.` | `*.` | `/.`
	|	`@` | `^` | `!` | `:=` | `=` | `<>` | `==` | `!=` | `!`
	|	`<` | `<=` | `>` | `<=` | `<.` | `<=.` | `>.` | `<=.`
cconstr	::=	*global-name*
	|	`[]`
	|	`()`
ncconstr	::=	*global-name*
	|	`prefix ::`
typeconstr	::=	*global-name*
label	::=	*global-name*

Depending on the context, global names can stand for global variables (*variable*), constant value constructors (*cconstr*), non-constant value constructors (*ncconst*), type constructors (*typeconstr*), or record labels (*label*). For variables and value constructors, special names built with `prefix` and an operator name are recognized. The tokens `[]` and `()` are also recognized as built-in constant constructors (the empty list and the unit value).

The syntax of the language restricts labels and type constructors to appear in certain positions, where no other kind of global names are accepted. Hence labels and type constructors have their own name spaces. Value constructors and value variables live in the same name space: a global name in value position is interpreted as a value constructor if it appears in the scope of a type declaration defining that constructor; otherwise, the global name is taken to be a value variable. For value constructors, the type declaration determines whether a constructor is constant or not.

2.3 Values

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

2.3.1 Base values

Integer numbers

Integer values are integer numbers from -2^{30} to $2^{30}-1$, that is -1073741824 to 1073741823. Implementations may support a wider range of integer values.

Floating-point numbers

Floating-point values are numbers in floating-point representation. Everything about floating-point values is implementation-dependent, including the range of representable numbers, the number of significant digits, and the way floating-point results are rounded.

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 interpretation of character codes between 128 and 255 is implementation-dependent.

Character strings

String values are finite sequences of characters. Implementations must support strings up to $2^{16}-1$ characters in length (65535 characters). Implementations may support longer strings.

2.3.2 Tuples

Tuples of values are written $(v_1, \ldots, v_n)$, standing for the n-tuple of values v_1 to v_n. Tuples of up to $2^{14}-1$ elements (16383 elements) must be supported, though implementations may support tuples with more elements.

2.3.3 Records

Record values are labeled tuples of values. The record value written $\{label_1 = v_1; \ldots; label_n = v_n\}$ associates the value v_i to the record label $label_i$, for $i = 1 \ldots n$. Records with up to $2^{14}-1$ fields (16383 fields) must be supported, though implementations may support records with more fields.

2.3.4 Arrays

Arrays are finite, variable-sized sequences of values of the same type. Arrays of length up to $2^{14}-1$ (16383 elements) must be supported, though implementations may support larger arrays.

2.3.5 Variant values

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

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

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

2.3.6 Functions

Functional values are mappings from values to values.

2.4 Type expressions

typexpr	::=	`'` *ident*
	|	`(` *typexpr* `)`
	|	*typexpr* `->` *typexpr*
	|	*typexpr* {`*` *typexpr*}$^+$
	|	*typeconstr*
	|	*typexpr* *typeconstr*
	|	`(` *typexpr* {`,` *typexpr*} `)` *typeconstr*

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

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

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

Type variables

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

Parenthesized types

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

Function types

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

Tuple types

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

Constructed types

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

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

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

2.5 Constants

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

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

2.6 Patterns

$$
\begin{array}{rcl}
\textit{pattern} & ::= & \textit{ident} \\
 & | & \texttt{\_} \\
 & | & \textit{pattern}\ \texttt{as}\ \textit{ident} \\
 & | & \texttt{(}\ \textit{pattern}\ \texttt{)} \\
 & | & \texttt{(}\ \textit{pattern}\ \texttt{:}\ \textit{typexpr}\ \texttt{)} \\
 & | & \textit{pattern}\ \texttt{|}\ \textit{pattern} \\
 & | & \textit{constant} \\
 & | & \textit{ncconstr}\ \textit{pattern} \\
 & | & \textit{pattern}\ \texttt{,}\ \textit{pattern}\ \{\texttt{,}\ \textit{pattern}\} \\
 & | & \texttt{\{}\ \textit{label} = \textit{pattern}\ \{\texttt{;}\ \textit{label} = \textit{pattern}\}\ \texttt{\}} \\
 & | & \texttt{[ ]} \\
 & | & \texttt{[}\ \textit{pattern}\ \{\texttt{;}\ \textit{pattern}\}\ \texttt{]} \\
 & | & \textit{pattern}\ \texttt{::}\ \textit{pattern}
\end{array}
$$

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

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

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

Variable patterns

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

Alias patterns

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

Parenthesized patterns

The pattern `(` $pattern_1$ `)` matches the same values as $pattern_1$. A type constraint can appear in a parenthesized patterns, as in `(` $pattern_1$ `:` *typexpr* `)`. This constraint forces the type of $pattern_1$ to be compatible with *type*.

"Or" patterns

The pattern $pattern_1$ `|` $pattern_2$ represents the logical "or" of the two patterns $pattern_1$ and $pattern_2$. A value matches $pattern_1$ `|` $pattern_2$ either if it matches $pattern_1$ or if it matches $pattern_2$. The two sub-patterns $pattern_1$ and $pattern_2$ must contain no identifiers. Hence no bindings are returned by matching against an "or" pattern.

Constant patterns

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

Variant patterns

The pattern *ncconstr* $pattern_1$ matches all variants whose constructor is equal to *ncconstr*, and whose argument matches $pattern_1$.

The pattern $pattern_1$ `::` $pattern_2$ matches non-empty lists whose heads match $pattern_1$, and whose tails match $pattern_2$. This pattern behaves like `prefix :: (` $pattern_1$ `,` $pattern_2$ `)`.

The pattern `[` $pattern_1$ `;` ... `;` $pattern_n$ `]` matches lists of length n whose elements match $pattern_1 \ldots pattern_n$, respectively. This pattern behaves like $pattern_1$ `::` ... `::` $pattern_n$ `:: []`.

Tuple patterns

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

Record patterns

The pattern `{` $label_1$ `=` $pattern_1$ `;` ... `;` $label_n$ `=` $pattern_n$ `}` matches records that define at least the labels $label_1$ through $label_n$, and such that the value associated to $label_i$ match the pattern $pattern_i$, for $i = 1, \ldots, n$. The record value can define more labels than $label_1 \ldots label_n$; the values associated to these extra labels are not taken into account for matching.

2.7 Expressions

```
            expr  ::=  ident
                    |  variable
                    |  constant
                    |  ( expr )
                    |  begin expr end
                    |  ( expr : typexpr )
                    |  expr , expr {, expr}
                    |  ncconstr expr
                    |  expr :: expr
                    |  [ expr {; expr} ]
                    |  [| expr {; expr} |]
                    |  { label = expr {; label = expr} }
                    |  expr expr
                    |  prefix-op expr
                    |  expr infix-op expr
                    |  expr . label
                    |  expr . label <- expr
                    |  expr .( expr )
                    |  expr .( expr ) <- expr
                    |  expr & expr
                    |  expr or expr
                    |  if expr then expr [else expr]
                    |  while expr do expr done
                    |  for ident = expr (to | downto) expr do expr done
                    |  expr ; expr
                    |  match expr with simple-matching
                    |  fun multiple-matching
                    |  function simple-matching
                    |  try expr with simple-matching
                    |  let [rec] let-binding {and let-binding} in expr

 simple-matching  ::=  pattern -> expr {| pattern -> expr}

multiple-matching ::=  pattern-list -> expr {| pattern-list -> expr}

    pattern-list  ::=  pattern {pattern}

     let-binding  ::=  pattern = expr
                    |  variable pattern-list = expr

       prefix-op  ::=  - | -. | !

        infix-op  ::=  + | - | * | / | mod | +. | -. | *. | /. | @ | ^ | ! | :=
                    |  = | <> | == | != | < | <= | > | <= | <. | <=. | >. | <=.
```

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

Construction or operator	Associativity
`!`	–
`.` `.(`	–
function application	right
constructor application	–
`-` `-.` (prefix)	–
`mod`	left
`*` `*.` `/` `/.`	left
`+` `+.` `-` `-.`	left
`::`	right
`@` `^`	right
comparisons (`=` `==` `<` etc.)	left
`not`	–
`&`	left
`or`	left
`,`	–
`<-` `:=`	right
`if`	–
`;`	right
`let` `match` `fun` `function` `try`	–

2.7.1 Simple expressions

Constants

Expressions consisting in a constant evaluate to this constant.

Variables

Expressions consisting in a variable evaluate to the value bound to this variable in the current evaluation environment. The variable can be either a qualified identifier or a simple identifier. Qualified identifiers always denote global variables. Simple identifiers denote either a local variable, if the identifier is locally bound, or a global variable, whose full name is obtained by qualifying the simple identifier, as described in section 2.2.

Parenthesized expressions

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

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

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

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

Function abstraction

The most general form of function abstraction is:

$$
\begin{array}{lllll}
\texttt{fun} & \mathit{pattern}_1^1 \ \ldots \ \mathit{pattern}_1^m & \texttt{->} & \mathit{expr}_1 \\
\ \ \texttt{|} & \ldots & & \\
\ \ \texttt{|} & \mathit{pattern}_n^1 \ \ldots \ \mathit{pattern}_n^m & \texttt{->} & \mathit{expr}_n
\end{array}
$$

This expression evaluates to a functional value with m curried arguments. When this function is applied to m values $v_1 \ldots v_m$, the values are matched against each pattern row $\mathit{pattern}_i^1 \ldots \mathit{pattern}_i^m$ for i from 1 to n. If one of these matchings succeeds, that is if the value v_j matches the pattern $\mathit{pattern}_i^j$ for all $j = 1, \ldots, m$, then the expression expr_i associated to the selected pattern row is evaluated, and its value becomes the value of the function application. The evaluation of expr_i takes place in an environment enriched by the bindings performed during the matching.

If several pattern rows match the arguments, the one that occurs first in the function definition is selected. If none of the pattern rows matches the argument, the exception `Match_failure` is raised.

If the function above is applied to less than m arguments, a functional value is returned, that represents the partial application of the function to the arguments provided. This partial application is a function that, when applied to the remaining arguments, matches all arguments against the pattern rows as described above. Matching does not start until all m arguments have been provided to the function; hence, partial applications of the function to less than m arguments never raise `Match_failure`.

All pattern rows in the function body must contain the same number of patterns. A variable must not be bound more than once in one pattern row.

Functions with only one argument can be defined with the `function` keyword instead of `fun`:

$$
\begin{array}{llll}
\texttt{function} & \mathit{pattern}_1 & \texttt{->} & \mathit{expr}_1 \\
\qquad\quad \texttt{|} & \ldots & & \\
\qquad\quad \texttt{|} & \mathit{pattern}_n & \texttt{->} & \mathit{expr}_n
\end{array}
$$

The function thus defined behaves exactly as described above. The only difference between the two forms of function definition is how a parsing ambiguity is resolved. The two forms *cconstr pattern* (two patterns in a row) and *ncconstr pattern* (one pattern) cannot be distinguished syntactically. Function definitions introduced by `fun` resolve the ambiguity to the former form; function definitions introduced by `function` resolve it to the latter form (the former form makes no sense in this case).

Function application

Function application is denoted by juxtaposition of expressions. The expression $\mathit{expr}_1 \ \ \mathit{expr}_2 \ldots \mathit{expr}_n$ evaluates the expressions expr_1 to expr_n. The expression expr_1 must evaluate to a functional value, which is then applied to the values of $\mathit{expr}_2, \ldots, \mathit{expr}_n$. The order in which the expressions $\mathit{expr}_1, \ldots, \mathit{expr}_n$ are evaluated is not specified.

Local definitions

The `let` and `let rec` constructs bind variables locally. The construct

$$\texttt{let}\ pattern_1\ \texttt{=}\ expr_1\ \texttt{and}\ldots\texttt{and}\ pattern_n\ \texttt{=}\ expr_n\ \texttt{in}\ expr$$

evaluates $expr_1 \ldots expr_n$ in some unspecified order, then matches their values against the patterns $pattern_1 \ldots pattern_n$. If the matchings succeed, *expr* is evaluated in the environment enriched by the bindings performed during matching, and the value of *expr* is returned as the value of the whole `let` expression. If one of the matchings fails, the exception `Match_failure` is raised.

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

$$ident\ \texttt{=}\ \texttt{fun}\ pattern_1 \ldots pattern_m\ \texttt{->}\ expr$$

in a `let` expression, one may instead write

$$ident\ pattern_1 \ldots pattern_m\ \texttt{=}\ expr$$

Both forms bind *ident* to the curried function with m arguments and only one case,

$$pattern_1 \ldots pattern_m\ \texttt{->}\ expr.$$

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

$$\texttt{let rec}\ pattern_1\ \texttt{=}\ expr_1\ \texttt{and}\ldots\texttt{and}\ pattern_n\ \texttt{=}\ expr_n\ \texttt{in}\ expr$$

The only difference with the `let` construct described above is that the bindings of variables to values performed by the pattern-matching are considered already performed when the expressions $expr_1$ to $expr_n$ are evaluated. That is, the expressions $expr_1$ to $expr_n$ can reference identifiers that are bound by one of the patterns $pattern_1, \ldots, pattern_n$, and expect them to have the same value as in *expr*, the body of the `let rec` construct.

The recursive definition is guaranteed to behave as described above if the expressions $expr_1$ to $expr_n$ are function definitions (`fun`... or `function`...), and the patterns $pattern_1 \ldots pattern_n$ consist in a single variable, as in:

$$\texttt{let rec}\ ident_1\ \texttt{=}\ \texttt{fun}\ldots\texttt{and}\ldots\texttt{and}\ ident_n\ \texttt{=}\ \texttt{fun}\ldots\texttt{in}\ expr$$

This defines $ident_1 \ldots ident_n$ as mutually recursive functions local to *expr*. The behavior of other forms of `let rec` definitions is implementation-dependent.

2.7.2 Control constructs

Sequence

The expression $expr_1$ `;` $expr_2$ evaluates $expr_1$ first, then $expr_2$, and returns the value of $expr_2$.

Conditional

The expression `if` $expr_1$ `then` $expr_2$ `else` $expr_3$ evaluates to the value of $expr_2$ if $expr_1$ evaluates to the boolean `true`, and to the value of $expr_3$ if $expr_1$ evaluates to the boolean `false`.

The `else` $expr_3$ part can be omitted, in which case it defaults to `else ()`.

Case expression

The expression

$$\begin{array}{rlcl} \texttt{match} & \mathit{expr} & & \\ \texttt{with} & \mathit{pattern}_1 & \texttt{->} & \mathit{expr}_1 \\ \texttt{|} & \ldots & & \\ \texttt{|} & \mathit{pattern}_n & \texttt{->} & \mathit{expr}_n \end{array}$$

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

Boolean operators

The expression expr_1 `&` expr_2 evaluates to `true` if both expr_1 and expr_2 evaluate to `true`; otherwise, it evaluates to `false`. The first component, expr_1, is evaluated first. The second component, expr_2, is not evaluated if the first component evaluates to `false`. Hence, the expression expr_1 `&` expr_2 behaves exactly as

`if` expr_1 `then` expr_2 `else false`.

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

`if` expr_1 `then true else` expr_2.

Loops

The expression `while` expr_1 `do` expr_2 `done` repeatedly evaluates expr_2 while expr_1 evaluates to `true`. The loop condition expr_1 is evaluated and tested at the beginning of each iteration. The whole `while...done` expression evaluates to the unit value `()`.

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

The expression `for` *ident* `=` expr_1 `downto` expr_2 `do` expr_3 `done` first evaluates the expressions expr_1 and expr_2 (the boundaries) into integer values n and p. Then, the loop body expr_3 is repeatedly evaluated in an environment where the local variable named *ident* 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

$$\begin{array}{ll} \texttt{try} & \mathit{expr} \\ \texttt{with} & \mathit{pattern}_1 \ \texttt{->} \ \mathit{expr}_1 \\ \quad\texttt{|} & \ldots \\ \quad\texttt{|} & \mathit{pattern}_n \ \texttt{->} \ \mathit{expr}_n \end{array}$$

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

2.7.3 Operations on data structures

Products

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

Variants

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

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

Records

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

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

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

Arrays

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

The expression $expr_1$ `.(` $expr_2$ `)` is equivalent to the application `vect_item` $expr_1$ $expr_2$. In the initial environment, the identifier `vect_item` resolves to a built-in function that returns the value of element number $expr_2$ in the array denoted by $expr_1$. The first element has number 0; the last element has number $n-1$, where n is the size of the array. The exception `Invalid_argument` is raised if the access is out of bounds.

The expression $expr_1$ `.(` $expr_2$ `) <-` $expr_3$ is equivalent to `vect_assign` $expr_1$ $expr_2$ $expr_3$. In the initial environment, the identifier `vect_assign` resolves to a built-in function that modifies in-place the array denoted by $expr_1$, replacing element number $expr_2$ by the value of $expr_3$. The exception `Invalid_argument` is raised if the access is out of bounds. The built-in function returns `()`. Hence, the whole expression $expr_1$ `.(` $expr_2$ `) <-` $expr_3$ evaluates to the unit value `()`.

This behavior of the two constructs $expr_1$ `.(` $expr_2$ `)` and $expr_1$ `.(` $expr_2$ `) <-` $expr_3$ may change if the meaning of the identifiers `vect_item` and `vect_assign` is changed, either by redefinition or by modification of the list of opened modules. See the discussion below on operators.

2.7.4 Operators

The operators written `infix-op` in the grammar above can appear in infix position (between two expressions). The operators written `prefix-op` in the grammar above can appear in prefix position (in front of an expression).

The expression *prefix-op expr* is interpreted as the application *ident expr*, where *ident* is the identifier associated to the operator *prefix-op* in the table below. Similarly, the expression $expr_1$ *infix-op* $expr_2$ is interpreted as the application *ident* $expr_1$ $expr_2$, where *ident* is the identifier associated to the operator *infix-op* in the table below. The identifiers written *ident* above are then evaluated following the rules in section 2.7.1. In the initial environment, they evaluate to built-in functions whose behavior is described in the table. The behavior of the constructions *prefix-op expr* and $expr_1$ *infix-op* $expr_2$ may change if the meaning of the identifiers associated to *prefix-op* or *infix-op* is changed, either by redefinition of the identifiers, or by modification of the list of opened modules, through the `#open` and `#close` directives.

Operator	Associated identifier	Behavior in the default environment
`+`	`prefix +`	Integer addition.
`-` (infix)	`prefix -`	Integer subtraction.
`-` (prefix)	`minus`	Integer negation.
`*`	`prefix *`	Integer multiplication.
`/`	`prefix /`	Integer division. Raise `Division_by_zero` if second argument is zero. The result is unspecified if either argument is negative.
`mod`	`prefix mod`	Integer modulus. Raise `Division_by_zero` if second argument is zero. The result is unspecified if either argument is negative.
`+.`	`prefix +.`	Floating-point addition.
`-.` (infix)	`prefix -.`	Floating-point subtraction.
`-.` (prefix)	`minus_float`	Floating-point negation.
`*.`	`prefix *.`	Floating-point multiplication.
`/.`	`prefix /.`	Floating-point division. Raise `Division_by_zero` if second argument is zero.
`@`	`prefix @`	List concatenation.
`^`	`prefix ^`	String concatenation.
`!`	`prefix !`	Dereferencing (return the current contents of a reference).
`:=`	`prefix :=`	Reference assignment (update the reference given as first argument with the value of the second argument).
`=`	`prefix =`	Structural equality test.
`<>`	`prefix <>`	Structural inequality test.
`==`	`prefix ==`	Physical equality test.
`!=`	`prefix !=`	Physical inequality test.
`<`	`prefix <`	Test "less than" on integers.
`<=`	`prefix <=`	Test "less than or equal " on integers.
`>`	`prefix >`	Test "greater than" on integers.
`>=`	`prefix >=`	Test "greater than or equal" on integers.
`<.`	`prefix <.`	Test "less than" on floating-point numbers.
`<=.`	`prefix <=.`	Test "less than or equal " on floating-point numbers.
`>.`	`prefix >.`	Test "greater than" on floating-point numbers.
`>=.`	`prefix >=.`	Test "greater than or equal" on floating-point numbers.

The behavior of the `+`, `-`, `*`, `/`, `mod`, `+.`, `-.`, `*.` or `/.` operators is unspecified if the result falls outside of the range of representable integers or floating-point numbers, respectively. See chapter 13 for a more precise description of the behavior of the operators above.

2.8 Global definitions

This section describes the constructs that bind global identifiers (value variables, value constructors, type constructors, record labels).

2.8.1 Type definitions

type-definition	::=	`type` *typedef* {`and` *typedef*}
typedef	::=	*type-params ident* `=` *constr-decl* {`|` *constr-decl*}
	|	*type-params ident* `=` `{` *label-decl* {`;` *label-decl*} `}`
	|	*type-params ident* `==` *typexpr*
	|	*type-params ident*
type-params	::=	*nothing*
	|	`'` *ident*
	|	`(` `'` *ident* {`,` `'` *ident*} `)`
constr-decl	::=	*ident*
	|	*ident* `of` *typexpr*
label-decl	::=	*ident* `:` *typexpr*
	|	`mutable` *ident* `:` *typexpr*

Type definitions bind type constructors to data types: either variant types, record types, type abbreviations, or abstract data types.

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 an identifier, possibly preceded by one or several type parameters, and followed by a data type description. The identifier is the local name of the type constructor being defined. (The module name for this type constructor is the name of the module being compiled.) The optional type parameters are either one type variable `'` *ident*, for type constructors with one parameter, or a list of type variables (`'` $ident_1, \ldots,$ `'` $ident_n$), for type constructors with several parameters. These type parameters can appear in the type expressions of the right-hand side of the definition.

Variant types

The type definition *typeparams ident* `=` $constr\text{-}decl_1$ `|` ... `|` $constr\text{-}decl_n$ defines a variant type. The constructor declarations $constr\text{-}decl_1, \ldots, constr\text{-}decl_n$ describe the constructors associated to this variant type. The constructor declaration *ident* `of` *typexpr* declares the local name *ident* (in the module being compiled) as a non-constant constructor, whose argument has type *typexpr*. The constructor declaration *ident* declares the local name *ident* (in the module being compiled) as a constant constructor.

Record types

The type definition *typeparams ident* `=` `{` $label\text{-}decl_1$ `;` ... `;` $label\text{-}decl_n$ `}` defines a record type. The label declarations $label\text{-}decl_1, \ldots, label\text{-}decl_n$ describe the labels associated to this record type. The label declaration *ident* `:` *typexpr* declares the local name *ident* in the module being compiled as a label, whose argument has type *typexpr*. The label declaration `mutable` *ident* `:` *typexpr* behaves similarly; in addition, it allows physical modification over the argument to this label.

Type abbreviations

The type definition *typeparams ident* `==` *typexpr* defines the type constructor *ident* as an abbreviation for the type expression *typexpr*.

Abstract types

The type definition *typeparams ident* defines *ident* as an abstract type. When appearing in a module interface, this definition allows exporting a type constructor while hiding how it is represented in the module implementation.

2.8.2 Exception definitions

exception-definition ::= `exception` *constr-decl* {`and` *constr-decl*}

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

2.9 Directives

directive ::= `#` `open` *string*
  | `#` `close` *string*
  | `#` *ident string*

Directives control the behavior of the compiler. They apply to the remainder of the current compilation unit.

The two directives `#open` and `#close` modify the list of opened modules, that the compiler uses to complete unqualified identifiers, as described in section 2.2. The directive `#open` *string* adds the module whose name is given by the string constant *string* to the list of opened modules, in first position. The directive `#close` *string* removes the first occurrence of the module whose name is given by the string constant *string* from the list of opened modules.

Implementations can provide other directives, provided they follow the syntax `#` *ident string*, where *ident* is the name of the directive, and the string constant *string* is the argument to the directive. The behavior of these additional directives is implementation-dependent.

2.10 Module implementations

implementation ::= {*impl-phrase* `;;`}

impl-phrase ::= *expr*
  | *value-definition*
  | *type-definition*
  | *exception-definition*
  | *directive*

value-definition ::= `let` [`rec`] *let-binding* {`and` *let-binding*}

A module implementation consists in a sequence of implementation phrases, terminated by double semicolons. An implementation phrase is either an expression, a value definition, a type or exception definition, or a directive. At run-time, implementation phrases are evaluated sequentially, in the order in which they appear in the module implementation.

Implementation phrases consisting in an expression are evaluated for their side-effects.

Value definitions bind global value variables in the same way as a `let ... in ...` expression binds local variables. The expressions are evaluated, and their values are matched against the left-hand sides of the `=` sides, as explained in section 2.7.1. If the matching succeeds, the bindings of identifiers to values performed during matching are interpreted as bindings to the global value variables whose local name is the identifier, and whose module name is the name of the module. If the matching fails, the exception `Match_failure` is raised. The scope of these bindings is the phrases that follow the value definition in the module implementation.

Type and exception definitions introduce type constructors, variant constructors and record labels as described in sections 2.8.1 and 2.8.2. The scope of these definitions is the phrases that follow the value definition in the module implementation. The evaluation of an implementation phrase consisting in a type or exception definition produces no effect at run-time.

Directives modify the behavior of the compiler on the subsequent phrases of the module implementation, as described in section 2.9. The evaluation of an implementation phrase consisting in a directive produces no effect at run-time. Directives apply only to the module currently being compiled; in particular, they have no effect on other modules that refer to globals exported by the module being compiled.

2.11 Module interfaces

interface	::=	{*intf-phrase* `;;`}
intf-phrase	::=	*value-declaration*
	|	*type-definition*
	|	*exception-definition*
	|	*directive*
value-declaration	::=	`value` *ident* `:` *typexpr* {`and` *ident* `:` *typexpr*}

Module interfaces declare the global objects (value variables, type constructors, variant constructors, record labels) that a module exports, that is, makes available to other modules. Other modules can refer to these globals using qualified identifiers or the `#open` directive, as explained in section 2.2.

A module interface consists in a sequence of interface phrases, terminated by double semicolons. An interface phrase is either a value declaration, a type definition, an exception definition, or a directive.

Value declarations declare global value variables that are exported by the module implementation, and the types with which they are exported. The module implementation must define these variables, with types at least as general as the types declared in the interface. The scope of the bindings for these global variables extends from the module implementation itself to all modules that refer to those variables.

Type or exception definitions introduce type constructors, variant constructors and record labels as described in sections 2.8.1 and 2.8.2. Exception definitions and type definitions that are not abstract type declarations also take effect in the module implementation; that is, the type constructors, variant constructors and record labels they define are considered bound on entrance to the module implementation, and can be referred to by the implementation phrases. Type definitions that are not abstract type declarations must not be redefined in the module implementation. In contrast, the type constructors that are declared abstract in a module interface must be defined in the module implementation, with the same names.

Directives modify the behavior of the compiler on the subsequent phrases of the module interface, as described in section 2.9. Directives apply only to the interface currently being compiled; in particular, they have no effect on other modules that refer to globals exported by the interface being compiled.

Chapter 3

Language extensions

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

3.1 Streams, parsers, and printers

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

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

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

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

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

Stream patterns, also bracketed by `[<` and `>]`, describe initial segments of streams. In particular, the stream pattern `[< >]` matches all streams. Stream pattern components are matched against the corresponding elements of a stream. The component `'` *pattern* matches the corresponding stream element against the pattern. The component *expr pattern* applies the function denoted by *expr* to the current stream, then matches the result of the function against *pattern*. Finally, the component *ident* simply binds the identifier to the stream being matched. (The current implementation limits *ident* to appear last in a stream pattern.)

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

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

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

3.2 Guards

Cases of a pattern matching 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:

$$\begin{array}{rlll} \texttt{match} & expr & & \\ \texttt{with} & pattern_1[\texttt{when}cond_1] & \texttt{->} & expr_1 \\ \texttt{|} & \ldots & & \\ \texttt{|} & pattern_n[\texttt{when}cond_n] & \texttt{->} & expr_n \end{array}$$

(Same syntax for the `fun`, `function`, and `try` ... `with` constructs.) During matching, if the value of *expr* 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 $pattern_i$.

3.3 Range patterns

In patterns, Caml Light recognizes the form ` $c$ ` `..` ` $d$ ` (two character constants separated by `..`) as a shorthand for the pattern

` $c$ ` | ` $c_1$ ` | ` $c_2$ ` | ... | ` $c_n$ ` | ` $d$ `

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.

3.4 Recursive definitions of values

Besides `let rec` definitions of functional values, as described in the reference manual, Caml Light supports a certain class of recursive definitions of non-functional values. For instance, the following definition is accepted:

```
let rec x = 1 :: y and y = 2 :: x;;
```

and correctly binds `x` to the cyclic list `1::2::1::2::...`, and `y` to the cyclic list `2::1::2::1::...`Informally, the class of accepted definitions consists of those definitions where the defined variables occur only inside function bodies or as a field of a data structure. Moreover, the patterns in the left-hand sides must be identifiers, nothing more complex.

3.5 Local definitions using `where`

A postfix syntax for local definitions is provided:

expr ::= ...
| *expr* `where` [`rec`] *let-binding*

The expression *expr* `where` *let-binding* behaves exactly as `let` *let-binding* `in` *expr*, and similarly for `where rec` and `let rec`.

3.6 Mutable variant types

The argument of a value constructor can be declared "mutable" when the variant type is defined:

```
type foo = A of mutable int
         | B of mutable int * int
         | ...
```

This allows in-place modification of the argument part of a constructed value. Modification is performed by a new kind of expressions, written *ident* `<-` *expr*, where *ident* is an identifier bound by pattern-matching to the argument of a mutable constructor, and *expr* denotes the value that must be stored in place of that argument. Continuing the example above:

```
let x = A 1 in
  begin match x with A y -> y <- 2 | _ -> () end;
  x
```

returns the value `A 2`. The notation *ident* `<-` *expr* works also if *ident* is an identifier bound by pattern-matching to the value of a mutable field in a record. For instance,

```
type bar = {mutable lbl : int};;
let x = {lbl = 1} in
  begin match x with {lbl = y} -> y <- 2 end;
  x
```

returns the value `{lbl = 2}`.

3.7 String access

Extra syntactic constructs are provided to access and modify characters in strings:

$$\begin{array}{rcl} expr & ::= & \ldots \\ & | & expr \;\texttt{.[}\; expr \;\texttt{]} \\ & | & expr \;\texttt{.[}\; expr \;\texttt{]} \;\texttt{<-}\; expr \end{array}$$

The expression $expr_1$ `.[` $expr_2$ `]` is equivalent to the application `nth_char` $expr_1$ $expr_2$. In the initial environment, the identifier `nth_char` resolves to a built-in function that returns the character number $expr_2$ in the string denoted by $expr_1$. The first element has number 0; the last element has number $n-1$, where n is the length of the string. The exception `Invalid_argument` is raised if the access is out of bounds.

The expression $expr_1$ `.[` $expr_2$ `] <-` $expr_3$ is equivalent to `set_nth_char` $expr_1$ $expr_2$ $expr_3$. In the initial environment, the identifier `set_nth_char` resolves to a built-in function that modifies in-place the string denoted by $expr_1$, replacing character number $expr_2$ by the value of $expr_3$. The exception `Invalid_argument` is raised if the access is out of bounds. The built-in function returns `()`.

3.8 Alternate syntax

The syntax of some constructs has been slightly relaxed:

- An optional `;` may terminate a sequence, list expression, or record expression. For instance, `begin` e_1 `;` e_2 `; end` is syntactically correct and synonymous with `begin` e_1 `;` e_2 `end`.
- Similarly, an optional `|` may begin a pattern-matching expression. For instance, `function |` pat_1 `->` $expr_1$ `|`... is syntactically correct and synonymous with `function` pat_1 `->` $expr_1$ `|`....
- The tokens `&&` and `||` are recognized as synonymous for `&` (sequential "and") and `or` (sequential "or"), respectively.

3.9 Infix symbols

Sequences of "operator characters", such as `<=>` or `!!`, are read as a single token from the *infix-symbol* or *prefix-symbol* class:

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

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

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

Tokens from these two classes generalize the built-in infix and prefix operators described in chapter 2:

expr ::= ...
  | *prefix-symbol expr*
  | *expr infix-symbol expr*

variable ::= ...
  | `prefix` *prefix-symbol*
  | `prefix` *infix-symbol*

No `#infix` directive (section 3.10) is needed to give infix symbols their infix status. The precedences and associativities of infix symbols in expressions are determined by their first character(s): symbols beginning with `**` have highest precedence (exponentiation), followed by symbols beginning with `*`, `/` or `%` (multiplication), then `+` and `-` (addition), then all others symbols (comparisons). The updated precedence table for expressions is shown below. We write "`*...`" to mean "any infix symbol starting with `*`".

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

Some infix and prefix symbols are predefined in the default environment (see chapters 2 and 13 for a description of their behavior). The others are initially unbound and must be bound before use, with a `let prefix` *infix-symbol* `=` *expr* or `let prefix` *prefix-symbol* `=` *expr* binding.

3.10 Directives

In addition to the standard `#open` and `#close` directives, Caml Light provides three additional directives.

`#infix` " *id* "

Change the lexical status of the identifier *id*: in the remainder of the compilation unit, *id* is recognized as an infix operator, just like `+`. The notation `prefix` *id* can be used to refer to the identifier *id* itself. Expressions of the form $expr_1$ *id* $expr_2$ are parsed as the application `prefix` *id* $expr_1$ $expr_2$. The argument to the `#infix` directive must be an identifier, that is, a sequence of letters, digits and underscores starting with a letter; otherwise, the `#infix` declaration has no effect. Example:

```
#infix "union";;
let prefix union = fun x y -> ... ;;
[1,2] union [3,4];;
```

`#uninfix` " *id* "

Remove the infix status attached to the identifier *id* by a previous `#infix` " *id* " directive.

`#directory` " *dir-name* "

Add the named directory to the path of directories searched for compiled module interface files. This is equivalent to the `-I` command-line option to the batch compiler and the toplevel system.

Part III

The Caml Light commands

Chapter 4

Batch compilation (camlc)

This chapter describes how Caml Light programs can be compiled non-interactively, and turned into standalone executable files. This is achieved by the command `camlc`, which compiles and links Caml Light source files.

Mac: This command is not a standalone Macintosh application. To run `camlc`, you need the Macintosh Programmer's Workshop (MPW) programming environment. The programs generated by `camlc` are also MPW tools, not standalone Macintosh applications.

4.1 Overview of the compiler

The `camlc` command has a command-line interface similar to the one of most C compilers. It accepts several types of arguments: source files for module implementations; source files for module interfaces; and compiled module implementations.

- Arguments ending in `.mli` are taken to be source files for module interfaces. Module interfaces declare exported global identifiers, define public data types, and so on. From the file *x*`.mli`, the `camlc` compiler produces a compiled interface in the file *x*`.zi`.

- Arguments ending in `.ml` are taken to be source files for module implementation. Module implementations bind global identifiers to values, define private data types, and contain expressions to be evaluated for their side-effects. From the file *x*`.ml`, the `camlc` compiler produces compiled object code in the file *x*`.zo`. If the interface file *x*`.mli` exists, the module implementation *x*`.ml` is checked against the corresponding compiled interface *x*`.zi`, which is assumed to exist. If no interface *x*`.mli` is provided, the compilation of *x*`.ml` produces a compiled interface file *x*`.zi` in addition to the compiled object code file *x*`.zo`. The file *x*`.zi` produced corresponds to an interface that exports everything that is defined in the implementation *x*`.ml`.

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

The output of the linking phase is a file containing compiled code that can be executed by the Caml Light runtime system: the command named `camlrun`. If `caml.out` is the name of the file produced by the linking phase, the command

`camlrun caml.out` arg_1 arg_2 ... arg_n

executes the compiled code contained in `caml.out`, passing it as arguments the character strings arg_1 to arg_n. (See chapter 6 for more details.)

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

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

The produced file has the executable bit set, and it manages to launch the bytecode interpreter by itself.

PC: The output file produced by the linking phase is directly executable, provided it is given extension `.EXE`. Hence, if the output file is named `caml_out.exe`, you can execute it with the command

`caml_out` arg_1 arg_2 ... arg_n

Actually, the produced file `caml_out.exe` is a tiny executable file prepended to the bytecode file. The executable simply runs the `camlrun` runtime system on the remainder of the file. (As a consequence, this is not a standalone executable: it still requires `camlrun.exe` to reside in one of the directories in the path.)

4.2 Options

The following command-line options are recognized by `camlc`.

`-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.

`-ccopt` *option*
Pass the given option to the C compiler and linker, when linking in "custom runtime" mode (see the `-custom` option). For instance, `-ccopt -L`*dir* causes the C linker to search for C libraries in directory *dir*.

`-custom`
Link in "custom runtime" mode. In the default linking mode, the linker produces bytecode that is intended to be executed with the shared runtime system, `camlrun`. 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 considerably larger, but it can be executed directly, even if the `camlrun` command is not installed. Moreover, the "custom runtime" mode enables linking Caml Light code with user-defined C functions, as described in chapter 12.

Unix: Never strip an executable produced with the `-custom` option.

PC: This option requires the `DJGPP` port of the GNU C compiler to be installed.

`-files` *response-file*
Process the files whose names are listed in file *response-file*, just as if these names appeared on the command line. File names in *response-file* are separated by blanks (spaces, tabs, newlines). This option allows to overcome silly limitations on the length of the command line.

`-g` Cause the compiler to produce additional debugging information. During the linking phase, this option add information at the end of the executable bytecode file produced. This information is required by the debugger `camldebug` and also by the catch-all exception handler from the standard library module `printexc`.

During the compilation of an implementation file (`.ml` file), when the `-g` option is set, the compiler adds debugging information to the `.zo` file. It also writes a `.zix` file that describes the full interface of the `.ml` file, that is, all types and values defined in the `.ml` file, including those that are local to the `.ml` file (i.e. not declared in the `.mli` interface file). Used in conjunction with the `-g` option to the toplevel system (chapter 5), the `.zix` file gives access to the local values of the module, making it possible to print or "trace" them. The `.zix` file is not produced if the implementation file has no explicit interface, since, in this case, the module has no local values.

`-i` Cause the compiler to print the declared types, exceptions, and global variables (with their inferred types) when compiling an implementation (`.ml` file). This can be useful to check the types inferred by the compiler. Also, since the output follows the syntax of module 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 globals.

`-I` *directory*
Add the given directory to the list of directories searched for compiled interface files (`.zi`) and compiled object code files (`.zo`). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, but before the standard library directory. When several directories are added with several `-I` options on the command line, these directories are searched from right to left (the rightmost directory is searched first, the leftmost is searched last). (Directories can also be added to the search path from inside the programs with the `#directory` directive; see chapter 3.)

`-lang` *language-code*
Translate the compiler messages to the specified language. The *language-code* is `fr` for French, `es` for Spanish, `de` for German, ... (See the file `camlmsgs.txt` in the Caml Light standard library directory for a list of available languages.) When an unknown language is specified, or no translation is available for a message, American English is used by default.

`-o` *exec-file*
Specify the name of the output file produced by the linker.

Unix: The default output name is `a.out`, in keeping with the tradition.

PC: The default output name is `caml_out.exe`.

Mac: The default output name is `Caml.Out`.

`-O` *module-set*
Specify which set of standard modules is to be implicitly "opened" at the beginning of a compilation. There are three module sets currently available:

cautious
provides the standard operations on integers, floating-point numbers, characters, strings, arrays, ..., as well as exception handling, basic input/output, etc. Operations from the `cautious` set perform range and bound checking on string and array operations, as well as various sanity checks on their arguments.

fast
provides the same operations as the `cautious` set, but without sanity checks on their arguments. Programs compiled with `-O fast` are therefore slightly faster, but unsafe.

none
suppresses all automatic opening of modules. Compilation starts in an almost empty environment. This option is not of general use, except to compile the standard library itself.

The default compilation mode is `-O cautious`. See chapter 13 for a complete listing of the modules in the `cautious` and `fast` sets.

`-p` Compile and link in profiling mode. See the description of the profiler `camlpro` in chapter 10.

`-v` Print the version number of the compiler.

`-W` Print extra warning messages for the following events:

- A `#open` directive is useless (no identifier in the opened module is ever referenced).
- A variable name in a pattern matching is capitalized (often corresponds to a misspelled constant constructor).

Unix: The following environment variable is also consulted:

`LANG`
When set, control which language is used to print the compiler messages (see the `-lang` command-line option).

PC: The following environment variables are also consulted:

`CAMLLIB`
Contain the path to the standard library directory.

`LANG`
When set, control which language is used to print the compiler messages (see the `-lang` command-line option).

4.3 Modules and the file system

This short section is intended to clarify the relationship between the names of the modules and the names of the files that contain their compiled interface and compiled implementation.

The compiler always derives the name of the compiled module by taking the 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. The produced `.zi` and `.zo` files have the same base name as the source file; hence, the compiled files produced by the compiler always have their base name equal to the name of the module they describe (for `.zi` files) or implement (for `.zo` files).

For compiled interface files (`.zi` files), this invariant must be preserved at all times, since the compiler relies on it to load the compiled interface file for the modules that are used from the module being compiled. Hence, it is risky and generally incorrect to rename `.zi` files. 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.

Compiled bytecode files (`.zo` files), on the other hand, can be freely renamed once created. That's because 1- `.zo` files contain the true name of the module they define, so there is no need to derive that name from the file name; 2- the linker never attempts to find by itself the `.zo` file that implements a module of a given name: it relies on the user providing the list of `.zo` files by hand.

4.4 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 (`.zi` file), or a compiled bytecode file (`.zo` file). If *filename* has the format *mod*`.zi`, 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*`.zi`.

If *filename* has the format *mod*`.zo`, 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 file *filename*

The compiler produces this error when it tries to read a compiled interface file (`.zi` file) that has the wrong structure. This means something went wrong when this `.zi` 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 `.zi` file is modified after its creation by the compiler. Fix: remove the corrupted `.zi` file, and rebuild it.

This expression has type t_1**, but is used with type** t_2

This is by far the most common type error in programs. Type t_1 is the type inferred for the expression (the part of the program that is displayed in the error message), by looking at the expression itself. Type t_2 is the type expected by the context of the expression; it is deduced by looking at how the value of this expression is used in the rest of the program. If the two types t_1 and t_2 are not compatible, then the error above is produced.

In some cases, it is hard to understand why the two types t_1 and t_2 are incompatible. For instance, the compiler can report that "expression of type `foo` cannot be used with type `foo`", and it really seems that the two types `foo` are compatible. This is not always true. Two type constructors can have the same name, but actually represent different types. This can happen if a type constructor is redefined. Example:

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

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

Incompatible types with the same names can also appear when a module is changed and recompiled, but some of its clients are not recompiled. That's because type constructors in `.zi` files are not represented by their name (that would not suffice to identify them, because of type redefinitions), but by unique stamps that are assigned when the type declaration is compiled. Consider the three modules:

```
mod1.ml:    type t = A | B;;
            let f = function A -> 0 | B -> 1;;

mod2.ml:    let g x = 1 + mod1__f(x);;

mod3.ml:    mod2__g mod1__A;;
```

Now, assume `mod1.ml` is changed and recompiled, but `mod2.ml` is not recompiled. The recompilation of `mod1.ml` can change the stamp assigned to type `t`. But the interface `mod2.zi` will still use the old stamp for `mod1__t` in the type of `mod2__g`. Hence, when compiling `mod3.ml`, the system complains that the argument type of `mod2__g` (that is, `mod1__t` with the old stamp) is not compatible with the type of `mod1__A` (that is, `mod1__t` with the new stamp). Fix: use `make` or a similar tool to ensure that all clients of a module *mod* are recompiled when the interface *mod*`.zi` changes. To check that the `Makefile` contains the right dependencies, remove all `.zi` files and rebuild the whole program; if no "Cannot find file" error appears, you're all set.

The type inferred for *name*, that is, *t*, contains non-generalizable type variables

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

Non-generalized type variables in a type cause no difficulties inside a given compilation unit (the contents of a `.ml` file, or an interactive session), but they cannot be allowed in types written in a `.zi` compiled interface file, because they could be used inconsistently in other compilation units. Therefore, the compiler flags an error when a `.ml` implementation without a `.mli` interface defines a global variable *name* whose type contains non-generalized type variables. There are two solutions to this problem:

- Add a type constraint or a `.mli` interface to give a monomorphic type (without type variables) to *name*. For instance, instead of writing

  ```
  let sort_int_list = sort (prefix <);;
  (* inferred type 'a list -> 'a list, with 'a not generalized *)
  ```

  write

  ```
  let sort_int_list = (sort (prefix <) : 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 = map vect_length;;
  (* inferred type 'a vect list -> int list, with 'a not generalized *)
  ```

  write

  ```
      let map_length lv = map vect_length lv;;
  ```

*mod*__*name* **is referenced before being defined**

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 module named *mod* on the command line (typically, the file named *mod*`.zo`, or a library containing that file). Fix: add the missing `.ml` or `.zo` file to the command line. Or, you have provided an implementation for the module named *mod*, but it comes too late on the command line: the implementation of *mod* must come before all bytecode object files that reference one of the global variables defined in module *mod*. Fix: change the order of `.ml` and `.zo` files on the command line.

Of course, you will always encounter this error if you have mutually recursive functions across modules. That is, function `mod1__f` calls function `mod2__g`, and function `mod2__g` calls function `mod1__f`. In this case, no matter what permutations you perform on the command line, the program will be rejected at link-time. Fixes:

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

  ```
  mod1.ml:    let f x = ... mod2__g ... ;;
  mod2.ml:    let g y = ... mod1__f ... ;;
  ```

  define

```
mod1.ml:    let f g x = ... g ... ;;
mod2.ml:    let rec g y = ... mod1__f g ... ;;
```

and link `mod1` before `mod2`.

- Use a reference to hold one of the two functions, as in :

```
mod1.ml:    let forward_g =
                ref((fun x -> failwith "forward_g") : <type>);;
            let f x = ... !forward_g ... ;;
mod2.ml:    let g y = ... mod1__f ... ;;
            mod1__forward_g := g;;
```

Unavailable C primitive *f*

This error appears when trying to link code that calls external functions written in C in “default runtime” mode. As explained in chapter 12, such code must be linked in “custom runtime” mode. Fix: add the `-custom` option, as well as the (native code) libraries and (native code) object files that implement the required external functions.

Chapter 5

The toplevel system (camllight)

This chapter describes the toplevel system for Caml Light, that permits interactive use of the Caml Light system, through a read-eval-print loop. In this mode, the system repeatedly reads Caml Light 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. A phrase can span several lines. Phrases are delimited by `;;` (the final double-semicolon).

From the standpoint of the module system, all phrases entered at toplevel are treated as the implementation of a module named `top`. Hence, all toplevel definitions are entered in the module `top`.

Unix: The toplevel system is started by the command `camllight`. Phrases are read on standard input, results are printed on standard output, errors on standard error. End-of-file on standard input terminates `camllight` (see also the `quit` system function below).

The toplevel system does not perform line editing, but it can easily be used in conjunction with an external line editor such as `fep`; just run `fep -emacs camllight` or `fep -vi camllight`. Another option is to use `camllight` under Gnu Emacs, which gives the full editing power of Emacs (see the directory `contrib/camlmode` in the distribution).

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing `ctrl-C` (or, more precisely, by sending the `intr` signal to the `camllight` process). This goes back to the `#` prompt.

Mac: The toplevel system is presented as the standalone Macintosh application `Caml Light`. This application does not require the Macintosh Programmer's Workshop to run.

Once launched from the Finder, the application opens two windows, "Caml Light Input" and "Caml Light Output". Phrases are entered in the "Caml Light Input" window. The "Caml Light Output" window displays a copy of the input phrases as they are processed by the Caml Light toplevel, interspersed with the toplevel responses. The "Return" key sends the contents of the Input window to the Caml Light toplevel. The "Enter" key inserts a newline without sending the contents of the Input window. (This can be configured with the "Preferences" menu item.)

The contents of the input window can be edited at all times, with the standard Macintosh interface. An history of previously entered phrases is maintained, and can be accessed with the "Previous entry" (command-P) and "Next entry" (command-N) menu items.

To quit the `Caml Light` application, either select "Quit" from the "Files" menu, or use the `quit` function described below.

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing "command-period", or by selecting the item "Interrupt Caml Light" in the "Caml Light" menu. This goes back to the `#` prompt.

PC: The toplevel system is started by the command `caml`. (The full name `camllight` had to be truncated because of the well-known limitations of MS-DOS.) Phrases are read on standard input, results are printed on standard output, errors on standard error. The `quit` system function terminates `caml`.

A number of TSR line editors that provide line editing and history facilities for the `command.com` command-line interpreter also work in conjunction with `caml`. Some TSR's that work: `dosedit`, `ced`, `toddy`. Some TSR's that don't work: `doskey` from the MS-DOS distribution.

At any point, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing `ctrl-break` or `ctrl-C`. This goes back to the `#` prompt.

5.1 Options

The following command-line options are recognized by the `caml` or `camllight` commands.

`-g` Start the toplevel system in debugging mode. This mode gives access to values and types that are local to a module, that is, not exported by the interface of the module. When debugging mode is off, these local objects are not accessible (attempts to access them produce an "Unbound identifier" error). When debugging mode is on, these objects become visible, just like the objects that are exported in the module interface. In particular, values of abstract types are printed using their concrete representations, and the functions local to a module can be "traced" (see the `trace` function in section 5.2). This applies only to the modules that have been compiled in debugging mode (either by the batch compiler with the `-g` option, or by the toplevel system in debugging mode), that is, those modules that have an associated `.zix` file.

`-I` *directory*
Add the given directory to the list of directories searched for compiled interface files (`.zi`) and compiled object code files (`.zo`). By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, but before the standard library directory. When several directories are added with several `-I` options on the command line, these directories are searched from right to left (the rightmost directory is searched first, the leftmost is searched last). Directories can also be added to the search path once the toplevel is running with the `#directory` directive; see chapter 3.

`-lang` *language-code*
Translate the toplevel messages to the specified language. The *language-code* is `fr` for French, `es` for Spanish, `de` for German, ... (See the file `camlmsgs.txt` in the Caml Light standard library directory for a list of available languages.) When an unknown language is specified, or no translation is available for a message, American English is used by default.

`-O` *module-set*
Specify which set of standard modules is to be implicitly "opened" when the toplevel starts. There are three module sets currently available:

cautious
provides the standard operations on integers, floating-point numbers, characters, strings, arrays, ..., as well as exception handling, basic input/output, ... Operations from the `cautious` set perform range and bound checking on string and vector operations, as well as various sanity checks on their arguments.

fast
provides the same operations as the `cautious` set, but without sanity checks on their arguments. Programs compiled with `-O fast` are therefore slightly faster, but unsafe.

none
suppresses all automatic opening of modules. Compilation starts in an almost empty environment. This option is not of general use.

The default compilation mode is `-O cautious`. See chapter 13 for a complete listing of the modules in the `cautious` and `fast` sets.

Unix: The following environment variables are also consulted:

`LANG`
When set, control which language is used to print the compiler messages (see the `-lang` command-line option).

`LC_CTYPE`
If set to `iso_8859_1`, accented characters (from the ISO Latin-1 character set) in string and character literals are printed as is; otherwise, they are printed as decimal escape sequences (`\`*ddd*).

PC: The following environment variables are also consulted:

`CAMLLIB`
Contain the path to the standard library directory.

`LANG`
When set, control which language is used to print the compiler messages (see the `-lang` command-line option).

5.2 Toplevel control functions

The standard library module `toplevel`, opened by default when the toplevel system is launched, provides a number of functions that control the toplevel behavior, load files in memory, and trace program execution.

```
value quit : unit -> unit
```

Exit the toplevel loop and terminate the `camllight` command.

`value include : string -> unit`

Read, compile and execute source phrases from the given file. The `.ml` extension is automatically added to the file name, if not present. This is textual inclusion: phrases are processed just as if they were typed on standard input. In particular, global identifiers defined by these phrases are entered in the module named `top`, not in a new module.

`value load : string -> unit`

Load in memory the source code for a module implementation. Read, compile and execute source phrases from the given file. The `.ml` extension is automatically added if not present. The `load` function behaves much like `include`, except that a new module is created, with name the base name of the source file name. Global identifiers defined in the source file are entered in this module, instead of the `top` module as in the case of `include`. For instance, assuming file `foo.ml` contains the single phrase

```
let bar = 1;;
```

executing `load "foo"` defines the identifier `foo__bar` with value `1`. Caveat: the loaded module is not automatically opened: the identifier `bar` does not automatically complete to `foo__bar`. To achieve this, you must execute the directive `#open "foo"` afterwards.

`value compile : string -> unit`

Compile the source code for a module implementation or interface (`.ml` or `.mli` file). Compilation proceeds as with the batch compiler, and produces the same results as `camlc -c`. If the toplevel system is in debugging mode (option `-g` or function `debug_mode` below), the compilation is also performed in debugging mode, as when giving the `-g` option to the batch compiler. The result of the compilation is left in files (`.zo`, `.zi`, `.zix`). The compiled code is not loaded in memory. Use `load_object` to load a `.zo` file produced by `compile`.

`value load_object : string -> unit`

Load in memory the compiled bytecode contained in the given file. The `.zo` extension is automatically added to the file name, if not present. The bytecode file has been produced either by the standalone compiler `camlc` or by the `compile` function. Global identifiers defined in the file being loaded are entered in their own module, not in the `top` module, just as with the `load` function.

`value trace : string -> unit`

After the execution of `trace "foo"`, all calls to the global function named `foo` will be "traced". That is, the argument and the result are displayed for each call, as well as the exceptions escaping out of `foo`, either raised by `foo` itself, or raised by one of the functions called from `foo`. If `foo` is a curried function, each argument is printed as it is passed to the function. Only functions implemented in ML can be traced; system primitives such as `string_length` or user-supplied C functions cannot.

`value untrace : string -> unit`

Executing `untrace "foo"` stops all tracing over the global function named `foo`.

`value verbose_mode: bool -> unit`

`verbose_mode true` causes the `compile` function to print the inferred types and other information. `verbose_mode false` reverts to the default silent behavior.

`value install_printer : string -> unit`

`install_printer "printername"` registers the function named `printername` as a printer for objects whose types match its argument type. That is, the toplevel loop will call `printername` when it has such an object to print. The printing function `printername` must use the `format` library module to produce its output, otherwise the output of `printername` will not be correctly located in the values printed by the toplevel loop.

`value remove_printer : string -> unit`

`remove_printer "printername"` removes the function named `printername` from the table of toplevel printers.

`value set_print_depth : int -> unit`

`set_print_depth n` limits the printing of values to a maximal depth of `n`. The parts of values whose depth exceeds `n` are printed as ... (ellipsis).

`value set_print_length : int -> unit`

`set_print_length n` limits the number of value nodes printed to at most `n`. Remaining parts of values are printed as ... (ellipsis).

`value debug_mode: bool -> unit`

Set whether extended module interfaces must be used `debug_mode true` or not `debug_mode false`. Extended module interfaces are `.zix` files that describe the actual implementation of a module, including private types and variables. They are generated when compiling with `camlc -g`, or with the `compile` function above when `debug_mode` is `true`. When `debug_mode` is `true`, toplevel phrases can refer to private types and variables of modules, and private functions can be traced with `trace`. Setting `debug_mode true` is equivalent to starting the toplevel with the `-g` option.

`value cd : string -> unit`

Change the current working directory.

`value directory : string -> unit`

Add the given directory to the search path for files. Same behavior as the `-I` option or the `#directory` directive.

5.3 The toplevel and the module system

Toplevel phrases can refer to identifiers defined in modules other than the `top` module with the same mechanisms as for separately compiled modules: either by using qualified identifiers (`modulename__localname`), or by using unqualified identifiers that are automatically completed by searching the list of opened modules. (See section 2.2.) The modules opened at startup are given in the documentation for the standard library. Other modules can be opened with the `#open` directive.

However, before referencing a global variable from a module other than the `top` module, a definition of that global variable must have been entered in memory. At start-up, the toplevel system contains the definitions for all the identifiers in the standard library. The definitions for user modules can be entered with the `load` or `load_object` functions described above. Referencing a global variable for which no definition has been provided by `load` or `load_object` results in the error "`Identifier foo__bar is referenced before being defined`". Since this is a tricky point, let us consider some examples.

1. The library function `sub_string` is defined in module `string`. This module is part of the standard library, and is one of the modules automatically opened at start-up. Hence, both phrases

   ```
   sub_string "qwerty" 1 3;;
   string__sub_string "qwerty" 1 3;;
   ```

   are correct, without having to use `#open`, `load`, or `load_object`.

2. The library function `printf` is defined in module `printf`. This module is part of the standard library, but it is not automatically opened at start-up. Hence, the phrase

   ```
   printf__printf "%s %s" "hello" "world";;
   ```

   is correctly executed, while

   ```
   printf "%s %s" "hello" "world";;
   ```

   causes the error "`Variable printf is unbound`", since none of the currently opened modules define a global with local name `printf`. However,

   ```
   #open "printf";;
   printf "%s %s" "hello" "world";;
   ```

   executes correctly.

3. Assume the file `foo.ml` resides in the current directory, and contains the single phrase

   ```
   let x = 1;;
   ```

When the toplevel starts, references to `x` will cause the error "`Variable x is unbound`". References to `foo__x` will cause the error "`Cannot find file foo.zi`", since the typechecker is attempting to load the compiled interface for module `foo` in order to find the type of `x`. To load in memory the module `foo`, just do:

```
load "foo";;
```

Then, references to `foo__x` typecheck and evaluate correctly. Since `load` does not open the module it loads, references to `x` will still fail with the error "`Variable x is unbound`". You will have to do

```
#open "foo";;
```

explicitly, for `x` to complete automatically into `foo__x`.

4. Finally, assume the file `foo.ml` above has been previously compiled with the `camlc -c` command. The current directory therefore contains a compiled interface `foo.zi`, claiming that `foo__x` is a global variable with type `int`, and a compiled bytecode file `foo.zo`, defining `foo__x` to have the value `1`. When the toplevel starts, references to `foo__x` will cause the error "`foo__x is referenced before being defined`". In contrast with case 3 above, the typechecker has succeeded in finding the compiled interface for module `foo`. But execution cannot proceed, because no definition for `foo__x` has been entered in memory. To do so, execute:

   ```
   load_object "foo";;
   ```

   This loads the file `foo.zo` in memory, therefore defining `foo__x`. Then, references to `foo__x` evaluate correctly. As in case 3 above, references to `x` still fail, because `load_object` does not open the module it loads. Again, you will have to do

   ```
   #open "foo";;
   ```

   explicitly, for `x` to complete automatically into `foo__x`.

5.4 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*`.zi`, this means the current phrase references identifiers from module *mod*, but you have not yet compiled an interface for module *mod*. Fix: either `load` the file *mod*`.ml`, which will also create in memory the compiled interface for module *mod*; or use `camlc` to compile *mod*`.mli` or *mod*`.ml`, creating the compiled interface *mod*`.zi`, before you start the toplevel.

If *filename* has the format *mod*`.zo`, this means you are trying to load with `load_object` a bytecode object file that does not exist yet. Fix: compile *mod*`.ml` with `camlc` before you start the toplevel. Or, use `load` instead of `load_object` to load the source code instead of a compiled object file.

If *filename* has the format *mod*`.ml`, this means `load` or `include` could not find the specified source file. Fix: check the spelling of the file name, or write it if it does not exist.

mod`__`*name* **is referenced before being defined**
You have neglected to load in memory an implementation for a module, with `load` or `load_object`. This is explained in full detail in section 5.3 above.

Corrupted compiled interface file *filename*
See section 4.4.

Expression of type t_1 **cannot be used with type** t_2
See section 4.4.

The type inferred for the value *name*, **that is,** *t*, **contains type variables that cannot be generalized**

See section 4.4.

5.5 Building custom toplevel systems: `camlmktop`

The `camlmktop` command builds Caml Light toplevels that contain user code preloaded at start-up.

Mac: This command is not available in the Macintosh version.

The `camlmktop` command takes as argument a set of `.zo` files, and links them with the object files that implement the Caml Light toplevel. The typical use is:

```
camlmktop -o mytoplevel foo.zo bar.zo gee.zo
```

This creates the bytecode file `mytoplevel`, containing the Caml Light toplevel system, plus the code from the three `.zo` files. To run this toplevel, give it as argument to the `camllight` command:

```
camllight mytoplevel
```

This starts a regular toplevel loop, except that the code from `foo.zo`, `bar.zo` and `gee.zo` is already loaded in memory, just as if you had typed:

```
load_object "foo";;
load_object "bar";;
load_object "gee";;
```

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.

5.6 Options

The following command-line options are recognized by `camlmktop`.

`-ccopt` *option*
: Pass the given option to the C compiler and linker, when linking in "custom runtime" mode. See the corresponding option for `camlc`, in chapter 4.

`-custom`
: Link in "custom runtime" mode. See the corresponding option for `camlc`, in chapter 4.

`-g`
: Add debugging information to the toplevel file produced, which can then be debugged with `camldebug` (chapter 9).

`-I` *directory*
: Add the given directory to the list of directories searched for compiled object code files (`.zo`).

`-o` *exec-file*
: Specify the name of the toplevel file produced by the linker.

Unix: The default is `camltop.out`.

PC: The default is `camltop.exe`. The name must have `.exe` extension.

Chapter 6

The runtime system (camlrun)

The `camlrun` command executes bytecode files produced by the linking phase of the `camlc` command.

Mac: This command is a MPW tool, not a standalone Macintosh application.

6.1 Overview

The `camlrun` 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 `camlrun` is:

`camlrun` *options bytecode-executable* arg_1 ... arg_n

The first non-option argument is taken to be the name of the file containing the executable bytecode. (That file is searched in the executable path as well as in the current directory.) The remaining arguments are passed to the Caml Light program, in the string array `sys__command_line`. Element 0 of this array is the name of the bytecode executable file; elements 1 to n are the remaining arguments arg_1 to arg_n.

As mentioned in chapter 4, in most cases, the bytecode executable files produced by the `camlc` command are self-executable, and manage to launch the `camlrun` command on themselves automatically. That is, assuming `caml.out` is a bytecode executable file,

`caml.out` arg_1 ... arg_n

works exactly as

`camlrun caml.out` arg_1 ... arg_n

Notice that it is not possible to pass options to `camlrun` when invoking `caml.out` directly.

6.2 Options

The following command-line option is recognized by `camlrun`.

`-V` Print out the `camlrun` version number. Exit immediately without executing any byte-code file.

The following environment variable are also consulted:

`CAMLRUNPARAM`
: Set the garbage collection parameters. This variable must be a sequence of parameter specifications. A parameter specification is an option letter followed by an `=` sign and a decimal number. There are four options, corresponding to the four fields of the `control` record documented in section 14.5:

- **s** (`minor_heap_size`) Size of the minor heap.
- **i** (`major_heap_increment`) Minimum size increment for the major heap.
- **o** (`space_overhead`) The major GC speed setting.
- **v** (`verbose`) Whether to print GC messages or not. 0 is false; 1 is true; other values may give unexpected results.

For example, under `csh` the command

```
setenv CAMLRUNPARAM 's=250000 v=1'
```

tells a subsequent `camlrun` to set its initial minor heap size to about 1 megabyte (on a 32-bit machine) and to print its GC messages.

`PATH`
: List of directories searched to find the bytecode executable file.

6.3 Common errors

This section describes and explains the most frequently encountered error messages.

filename`: no such file or directory`
: If *filename* is the name of a self-executable bytecode file, this means that either that file does not exist, or that it failed to run the `camlrun` bytecode interpreter on itself. The second possibility indicates that Caml Light has not been properly installed on your system.

`Cannot exec camlrun`
: (When launching a self-executable bytecode file.) The `camlrun` command could not be found in the executable path. Check that Caml Light has been properly installed on your system.

`Cannot find the bytecode file`
: The file that `camlrun` is trying to execute (e.g. the file given as first non-option argument to `camlrun`) either does not exist, or is not a valid executable bytecode file.

`Truncated bytecode file`

The file that `camlrun` 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. If you wish to know which exception thus escapes, use the `printexc__f` function from the standard library (and don't forget to link your program with the `-g` option).

`Out of memory`

The program being executed requires more memory than available. Either the program builds too 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. (This happens quite frequently on small microcomputers, but is unlikely on Unix machines.) In other cases, the "out of memory" message reveals an error in your program: non-terminating recursive function, allocation of an excessively large array or string, attempts to build an infinite list or other data structure, ...

To help you diagnose this error, run your program with the `-v` option to `camlrun`. If it displays lots of "`Growing stack...`" messages, this is probably a looping recursive function. If it displays lots of "`Growing heap...`" messages, with the heap size growing slowly, this is probably an attempt to construct a data structure with too many (infinitely many?) cells. If it displays few "`Growing heap...`" messages, but with a huge increment in the heap size, this is probably an attempt to build an excessively large array or string.

Chapter 7

The librarian (camllibr)

Mac: This command is a MPW tool, not a standalone Macintosh application.

7.1 Overview

The `camllibr` program packs in one single file a set of bytecode object files (`.zo` files). The resulting file is also a bytecode object file and also has the `.zo` extension. It can be passed to the link phase of the `camlc` compiler in replacement of the original set of bytecode object files. That is, after running

```
camllibr -o library.zo mod1.zo mod2.zo mod3.zi mod4.zo
```

all calls to the linker with the form

```
camlc ... library.zo ...
```

are exactly equivalent to

```
camlc ... mod1.zo mod2.zo mod3.zi mod4.zo ...
```

The typical use of `camllibr` is to build a library composed of several modules: this way, users of the library have only one `.zo` file to specify on the command line to `camlc`, instead of a bunch of `.zo` files, one per module contained in the library.

The linking phase of `camlc` is clever enough to discard the code corresponding to useless phrases: in particular, definitions for global variables that are never used after their definitions. Hence, there is no problem with putting many modules, even rarely used ones, into one single library: this will not result in bigger executables.

The usage for `camllibr` is:

`camllibr` *options* $file_1$`.zo` ... $file_n$`.zo`

where $file_1$.zo through $file_n$.zo are the object files to pack together. The order in which these file names are presented on the command line is relevant: the compiled phrases contained in the library will be executed in that order. (Remember that it is a link-time error to refer to a global variable that has not yet been defined.)

7.2 Options

The following command-line options are recognized by `camllibr`.

`-files` *response-file*
: Process the files whose names are listed in file *response-file*, just as if these names appeared on the command line. File names in *response-file* are separated by blanks (spaces, tabs, newlines). This option allows to overcome silly limitations on the length of the command line.

`-I` *directory*
: Add the given directory to the list of directories searched for the input `.zo` files. By default, the current directory is searched first, then the standard library directory. Directories added with `-I` are searched after the current directory, but before the standard library directory. When several directories are added with several `-I` options on the command line, these directories are searched from right to left (the rightmost directory is searched first, the leftmost is searched last).

`-o` *library-name*
: Specify the name of the output file. The default is `library.zo`.

7.3 Turning code into a library

To develop a library, it is usually more convenient to split it into several modules, that reflect the internal structure of the library. From the standpoint of the library users, however, it is preferable to view the library as a single module, with only one interface file (`.zi` file) and one implementation file (`.zo` file): linking is easier, and there is no need to put a bunch of `#open` directives, nor to have to remember the internal structure of the library.

The `camllibr` command allows having a single `.zo` file for the whole library. Here is how the Caml Light module system can be used (some say "abused") to have a single `.zi` file for the whole library. To be more concrete, assume that the library comprises three modules, `windows`, `images` and `buttons`. The idea is to add a fourth module, `mylib`, that re-exports the public parts of `windows`, `images` and `buttons`. The interface `mylib.mli` contains definitions for those types that are public (exported with their definitions), declarations for those types that are abstract (exported without their definitions), and declarations for the functions that can be called from the user's code:

```
(* File mylib.mli *)
type 'a option = None | Some of 'a;;    (* a public type *)
type window and image and button;;      (* three abstract types *)
value new_window : int -> int -> window (* the public functions *)
  and draw_image : image -> window -> int -> int -> unit
  and ...
```

The implementation of the `mylib` module simply equates the abstract types and the public functions to the corresponding types and functions in the modules `windows`, `images` and `buttons`:

```
(* File mylib.ml *)
type window == windows__win
 and image  == images__pixmap
 and button == buttons__t;;
let new_window = windows__open_window
and draw_image = images__draw
and ...
```

The files `windows.ml`, `images.ml` and `buttons.ml` can open the `mylib` module, to access the public types defined in the interface `mylib.mli`, such as the `option` type. Of course, these modules must not reference the abstract types nor the public functions, to avoid circularities.

Types such as `windows__win` in the example above can be exported by the `windows` module either abstractly or concretely (with their definition). Often, it is necessary to export them concretely, because the other modules in the library (`images`, `buttons`) need to build or destructure directly values of that type. Even if `windows__win` is exported concretely by the `windows` module, that type will remain abstract to the library user, since it is abstracted by the public interface `mylib`.

The actual building of the library `mylib` proceeds as follows:

```
camlc -c mylib.mli              # create mylib.zi
camlc -c windows.mli windows.ml images.mli images.ml
camlc -c buttons.mli buttons.ml
camlc -c mylib.ml               # create mylib.zo
mv mylib.zo tmplib.zo           # renaming to avoid overwriting mylib.zo
camllibr -o mylib.zo windows.zo images.zo buttons.zo tmplib.zo
```

Then, copy `mylib.zi` and `mylib.zo` to a place accessible to the library users. The other `.zi` and `.zo` files need not be copied.

Chapter 8

Lexer and parser generators (camllex, camlyacc)

This chapter describes two program generators: `camllex`, that produces a lexical analyzer from a set of regular expressions with associated semantic actions, and `camlyacc`, 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 `camllex` and `camlyacc` 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), "Compiler design in C" by Holub (Prentice-Hall, 1990), or "Lex & Yacc", by Mason and Brown (O'Reilly, 1990).

Streams and stream matching, as described in section 3.1, provide an alternative way to write lexers and parsers. The stream matching technique is more powerful than the combination of `camllex` and `camlyacc` in some cases (higher-order parsers), but less powerful in other cases (precedences). Choose whichever approach is more adapted to your parsing problem.

Mac: These commands are MPW tool, not standalone Macintosh applications.

8.1 Overview of `camllex`

The `camllex` 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

    camllex lexer.mll

produces Caml Light 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 `create_lexer_channel`, `create_lexer_string` and `create_lexer` from module `lexing` 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 13.)

When used in conjunction with a parser generated by `camlyacc`, the semantic actions compute a value belonging to the type `token` defined by the generated parsing module. (See the description of `camlyacc` below.)

8.2 Syntax of lexer definitions

The format of lexer definitions is as follows:

```
{ header }
rule entrypoint =
  parse regexp { action }
      | ...
      | regexp { action }
and entrypoint =
  parse ...
and ...
;;
```

Comments are delimited by `(*` and `*)`, as in Caml Light.

8.2.1 Header

The *header* section is arbitrary Caml Light text enclosed in curly braces. It can be omitted. If it is present, the enclosed text is copied as is at the beginning of the output file. Typically, the header section contains the `#open` directives required by the actions, and possibly some auxiliary functions used in the actions.

8.2.2 Entry points

The names of the entry points must be valid Caml Light identifiers.

8.2.3 Regular expressions

The regular expressions are in the style of `lex`, with a more Caml-like syntax.

` *char* `
: A character constant, with the same syntax as Caml Light character constants. Match the denoted character.

`_` Match any character.

`eof` Match the end of the lexer input.

" *string* "
: A string constant, with the same syntax as Caml Light string constants. Match the corresponding sequence of characters.

[*character-set*]
Match any single character belonging to the given character set. Valid character sets are: single character constants ` *c* `; ranges of characters ` $c_1$ ` - ` $c_2$ ` (all characters between c_1 and c_2, inclusive); and the union of two or more character sets, denoted by concatenation.

[^ *character-set*]
Match any single character not belonging to the given character set.

regexp *
(Repetition.) Match the concatenation of zero or more strings that match *regexp*.

regexp +
(Strict repetition.) Match the concatenation of one or more strings that match *regexp*.

regexp ?
(Option.) Match either the empty string, or a string matching *regexp*.

$regexp_1$ | $regexp_2$
(Alternative.) Match any string that matches either $regexp_1$ or $regexp_2$

$regexp_1$ $regexp_2$
(Concatenation.) Match the concatenation of two strings, the first matching $regexp_1$, the second matching $regexp_2$.

(*regexp*)
Match the same strings as *regexp*.

Concerning the precedences of operators, * and + have highest precedence, followed by ?, then concatenation, then | (alternation).

8.2.4 Actions

The actions are arbitrary Caml Light expressions. They are evaluated in a context where the identifier `lexbuf` is bound to the current lexer buffer. Some typical uses for `lexbuf`, in conjunction with the operations on lexer buffers provided by the `lexing` standard library module, are listed below.

`lexing__get_lexeme lexbuf`
Return the matched string.

`lexing__get_lexeme_char lexbuf` *n*
Return the n^{th}character in the matched string. The first character corresponds to $n = 0$.

`lexing__get_lexeme_start lexbuf`
Return the absolute position in the input text of the beginning of the matched string. The first character read from the input text has position 0.

`lexing__get_lexeme_end lexbuf`
Return the absolute position in the input text of the end of the matched string. The first character read from the input text has position 0.

entrypoint `lexbuf`

(Where *entrypoint* is the name of another entry point in the same lexer definition.) Recursively call the lexer on the given entry point. Useful for lexing nested comments, for example.

8.3 Overview of `camlyacc`

The `camlyacc` 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

`camlyacc` *options grammar*`.mly`

produces Caml Light 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 `camllex` 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 `camlyacc`.

8.4 Syntax of grammar definitions

Grammar definitions have the following format:

`%{`
  header
`%}`
  declarations
`%%`
  rules
`%%`
  trailer

Comments are enclosed between `/*` and `*/` (as in C) in the "declarations" and "rules" sections, and between `(*` and `*)` (as in Caml) in the "header" and "trailer" sections.

8.4.1 Header and trailer

The header and the trailer sections are Caml Light 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 required by the semantic actions of the rules. The trailer goes at the end of the output file.

8.4.2 Declarations

Declarations are given one per line. They all start with a `%` sign.

`%token` *symbol ... symbol*
: Declare the given symbols as tokens (terminal symbols). These symbols are added as constant constructors for the `token` concrete type.

`%token <` *type* `>` *symbol ... symbol*
: Declare the given symbols as tokens with an attached attribute of the given type. These symbols are added as constructors with arguments of the given type for the `token` concrete type. The *type* part is an arbitrary Caml Light type expression, except that all type constructor names must be fully qualified (e.g. `modname__typename`) for all types except standard built-in types, even if the proper `#open` directives (e.g. `#open "modname"`) were given in the header section. That's because the header is copied only to the `.ml` output file, but not to the `.mli` output file, while the *type* part of a `%token` declaration is copied to both.

`%start` *symbol ... symbol*
: Declare the given symbols as entry points for the grammar. For each entry point, a parsing function with the same name is defined in the output module. Non-terminals that are not declared as entry points have no such parsing function. Start symbols must be given a type with the `%type` directive below.

`%type <` *type* `>` *symbol ... symbol*
: Specify the type of the semantic attributes for the given symbols. This is mandatory for start symbols only. Other nonterminal symbols need not be given types by hand: these types will be inferred when running the output files through the Caml Light compiler (unless the `-s` option is in effect). The *type* part is an arbitrary Caml Light type expression, except that all type constructor names must be fully qualified (e.g. `modname__typename`) for all types except standard built-in types, even if the proper `#open` directives (e.g. `#open "modname"`) were given in the header section. That's because the header is copied only to the `.ml` output file, but not to the `.mli` output file, while the *type* part of a `%token` declaration is copied to both.

`%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.

8.4.3 Rules

The syntax for rules is as usual:

nonterminal :
    symbol ... *symbol* { *semantic-action* }
  | ...
  | *symbol* ... *symbol* { *semantic-action* }
;

Rules can also contain the `%prec` *symbol* directive in the right-hand side part, to override the default precedence and associativity of the rule with the precedence and associativity of the given symbol.

Semantic actions are arbitrary Caml Light 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.

Actions occurring in the middle of rules are not supported. Error recovery is not implemented.

8.5 Options

The `camlyacc` command recognizes the following options:

`-v` Generate a description of the parsing tables and a report on conflicts resulting from ambiguities in the grammar. The description is put in file *grammar*`.output`.

`-s` Generate a *grammar*`.ml` file with smaller phrases. Semantic actions are presented in the *grammar*`.ml` output file as one large vector of functions. By default, this vector is built by a single phrase. When the grammar is large, or contains complicated semantic actions, the resulting phrase may require large amounts of memory to be compiled by Caml Light. With the `-s` option, the vector of actions is constructed incrementally, one phrase per action. This lowers the memory requirements for the compiler, but it is no longer possible to infer the types of nonterminal symbols: typechecking is turned off on symbols that do not have a type specified by a `%type` directive.

`-b`*prefix*
Name the output files *prefix*`.ml`, *prefix*`.mli`, *prefix*`.output`, instead of the default naming convention.

8.6 A complete example

The all-time favorite: a desk calculator. This program reads arithmetic expressions on standard input, one per line, and prints their values. Here is the grammar definition:

```
/* File parser.mly */
%token <int> INT
%token PLUS MINUS TIMES DIV
```

```
%token LPAREN RPAREN
%token EOL
%left PLUS MINUS        /* lowest precedence */
%left TIMES DIV         /* medium precedence */
%nonassoc UMINUS        /* highest precedence */
%start Main             /* the entry point */
%type <int> Main
%%
Main:
    Expr EOL                { $1 }
;
Expr:
    INT                     { $1 }
  | LPAREN Expr RPAREN      { $2 }
  | Expr PLUS Expr          { $1 + $3 }
  | Expr MINUS Expr         { $1 - $3 }
  | Expr TIMES Expr         { $1 * $3 }
  | Expr DIV Expr           { $1 / $3 }
  | MINUS Expr %prec UMINUS { - $2 }
;
```

Here is the definition for the corresponding lexer:

```
(* File lexer.mll *)
{
#open "parser";;        (* The type token is defined in parser.mli *)
exception Eof;;
}
rule Token = parse
    [` ` `\t`]     { Token lexbuf }     (* skip blanks *)
  | [`\n` ]        { EOL }
  | [`0`-`9`]+     { INT(int_of_string (get_lexeme lexbuf)) }
  | `+`            { PLUS }
  | `-`            { MINUS }
  | `*`            { TIMES }
  | `/`            { DIV }
  | `(`            { LPAREN }
  | `)`            { RPAREN }
  | eof            { raise Eof }
;;
```

Here is the main program, that combines the parser with the lexer:

```
(* File calc.ml *)
try
  let lexbuf = lexing__create_lexer_channel std_in in
  while true do
```

```
    let result = parser__Main lexer__Token lexbuf in
      print_int result; print_newline(); flush std_out
  done
with Eof ->
  ()
;;
```

To compile everything, execute:

```
camllex lexer.mll       # generates lexer.ml
camlyacc parser.mly     # generates parser.ml and parser.mli
camlc -c parser.mli
camlc -c lexer.ml
camlc -c parser.ml
camlc -c calc.ml
camlc -o calc lexer.zo parser.zo calc.zo
```

Chapter 9

The debugger (camldebug)

This chapter describes the Caml Light source-level replay debugger `camldebug`.

Unix: The debugger resides in the directory `contrib/debugger` in the distribution. It requires a Unix system that provides BSD sockets.

Mac: The debugger is not available.

PC: The debugger is not available.

9.1 Compiling for debugging

Before the debugger can be used, the program must be compiled and linked with the `-g` option: all `.zo` files that are part of the program should have been created with `camlc -g`, and they must be linked together with `camlc -g`.

Compiling with `-g` entails no penalty on the running time of programs: `.zo` files and bytecode executable files are bigger and take slightly longer to produce, but the executable files run at exactly the same speed as if they had been compiled without `-g`. It is therefore perfectly acceptable to compile always in `-g` mode.

9.2 Invocation

9.2.1 Starting the debugger

The Caml Light debugger is invoked by running the program `camldebug` with the name of the bytecode executable file as argument:

`camldebug` *program*

The following command-line options are recognized:

`-stdlib` *directory*
Look for the standard library files in *directory* instead of in the default directory.

`-s` *socket*

Use *socket* for communicating with the debugged program. See the description of the command `set socket` (section 9.8.7) for the format of *socket*.

`-c` *count*

Set the maximum number of checkpoint to *count*.

`-cd` *directory*

Run the debugger program from the working directory *directory*, instead of the current directory.

`-emacs`

Tell the debugger it is executed under Emacs. (See section 11.4 for information on how to run the debugger under Emacs.)

9.2.2 Quitting 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.

9.3 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.

9.3.1 Getting help

The Caml Light 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.

9.3.2 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.

9.4 Executing a program

9.4.1 Events

Events are "interesting" locations in the source code, corresponding to the beginning or end of evaluation of "interesting" sub-expressions. Events are the unit of single-stepping (stepping goes to the next or previous event encountered in the program execution). Also, breakpoints can only be set at events. Thus, events play the role of line numbers in debuggers for conventional languages.

During program execution, a counter is incremented at each event encountered. The value of this counter is referred as the *current time*. Thanks to reverse execution, it is possible to jump back and forth to any time of the execution.

Here is where the debugger events (written $\bowtie$) are located in the source code:

- Following a function application:

  ```
  (f arg)⋈
  ```

- After receiving an argument to a function:

  ```
  fun x⋈ y⋈ z -> ⋈ ...
  ```

  If a curried function is defined by pattern-matching with several cases, events corresponding to the passing of arguments are displayed on the first case of the function, because pattern-matching has not yet determined which case to select:

  ```
  fun pat1⋈ pat2⋈ pat3 -> ⋈ ...
    | ...
  ```

- On each case of a pattern-matching definition (function, `match...with` construct, `try...with` construct):

```
function pat1 -> ⋈ expr1
       | ...
       | patN -> ⋈ exprN
```

- Between subexpressions of a sequence:

```
expr1; ⋈ expr2; ⋈ ...; ⋈ exprN
```

- In the two branches of a conditional expression:

```
if cond then ⋈ expr1 else ⋈ expr2
```

- At the beginning of each iteration of a loop:

```
while cond do ⋈ body done
for i = a to b do ⋈ body done
```

Exceptions: A function application followed by a function return is replaced by the compiler by a jump (tail-call optimization). In this case, no event is put after the function application. Also, no event is put after a function application when the function is a primitive function (written in C). Finally, several events may correspond to the same location in the compiled program. Then, the debugger cannot distinguish them, and selects one of the events to associate with the given code location. The event chosen is a "function application" event if there is one at that location, or otherwise the event which appears last in the source. This heuristic generally picks the "most interesting" event associated with the code location.

9.4.2 Starting the debugged program

The debugger starts executing the debugged program only when needed. This allows setting breapoints or assigning debugger variables before execution starts. There are several ways to start execution:

`run` Run the program until a breakpoint is hit, or the program terminates.

`step 0`
: Load the program and stop on the first event.

`goto` *time*
: Load the program and execute it until the given time. Useful when you already know approximately at what time the problem appears. Also useful to set breakpoints on function values that have not been computed at time 0 (see section 9.5).

The execution of a program is affected by certain information it receives when the debugger starts it, such as the command-line arguments to the program and its working directory. The debugger provides commands to specify this information (`set arguments` and `cd`). These commands must be used before program execution starts. If you try to change the arguments or the working directory after starting your program, the debugger will kill the program (after asking for confirmation).

9.4.3 Running the program

The following commands execute the program forward or backward, starting at the current time. The execution will stop either when specified by the command or when a breakpoint is encountered.

`run` Execute the program forward from current time. Stops at next breakpoint or when the program terminates.

`reverse`
: Execute the program backward from current time. Mostly useful to go to the last breakpoint encountered before the current time.

`step` [*count*]
: Run the program and stop at the next event. With an argument, do it *count* times.

`backstep` [*count*]
: Run the program backward and stop at the previous event. With an argument, do it *count* times.

`next` [*count*]
: Run the program and stop at the next event, skipping over function calls. With an argument, do it *count* times.

`finish`
: Run the program until the current function returns.

9.4.4 Time travel

You can jump directly to a given time, without stopping on breakpoints, using the `goto` command.

As you move through the program, the debugger maintains an history of the successive times you stop at. The `last` command can be used to revisit these times: each `last` command moves one step back through the history. That is useful mainly to undo commands such as `step` and `next`.

`goto` *time*
: Jump to the given time.

`last` [*count*]
: Go back to the latest time recorded in the execution history. With an argument, do it *count* times.

`set history` *size*
: Set the size of the execution history.

9.4.5 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.

9.5 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.

`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. Moreover, C functions are not recognized by the debugger.

`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.

9.6 The call stack

Each time the program performs a function application, it saves the location of the application (the return address) in a block of data called a stack frame. The frame also contains the local variables of the caller function. All the frames are allocated in a region of memory called the call stack. The command `backtrace` (or `bt`) displays parts of the call stack.

At any time, one of the stack frames is "selected" by the debugger; several debugger commands refer implicitly to the selected frame. In particular, whenever you ask the debugger for the value of a local variable, the value is found in the selected frame. The commands `frame`, `up` and `down` select whichever frame you are interested in.

When the program stops, the debugger automatically selects the currently executing frame and describes it briefly as the `frame` command does.

`frame`
: Describe the currently selected stack frame.

`frame` *frame-number*
: Select a stack frame by number and describe it. The frame currently executing when the program stopped has number 0; its caller has number 1; and so on up the call stack.

`backtrace` [*count*], `bt` [*count*]
: Print the call stack. This is useful to see which sequence of function calls led to the currently executing frame. With a positive argument, print only the innermost *count* frames. With a negative argument, print only the outermost -*count* frames.

`up` [*count*]
: Select and display the stack frame just "above" the selected frame, that is, the frame that called the selected frame. An argument says how many frames to go up.

`down` [*count*]
: Select and display the stack frame just "below" the selected frame, that is, the frame that was called by the selected frame. An argument says how many frames to go down.

9.7 Examining variable values

The debugger can print the current value of a program variable (either a global variable or a local variable relative to the selected stack frame). It can also print selected parts of a value by matching it against a pattern.

Variable names can be specified either fully qualified (*module-name*`__`*var-name*) or unqualified (*var-name*). Unqualified names either correspond to local variables, or are completed into fully qualified global names by looking at a list of "opened" modules that define the same name (see section 9.8.5 for how to open modules in the debugger.) The completion follows the same rules as in the Caml Light language (see section 2.2).

`print` *variables*
: Print the values of the given variables.

`match` *variable pattern*
: Match the value of the given variable against a pattern, and print the values bound to the identifiers in the pattern.

The syntax of patterns for the `match` command extends the one for Caml Light patterns:

pattern	::=	*ident*
	|	_
	|	(*pattern*)
	|	*ncconstr pattern*
	|	*pattern* , *pattern* {, *pattern*}
	|	{ *label* = *pattern* {; *label* = *pattern*} }
	|	[ ]
	|	[*pattern* {; *pattern*}]
	|	*pattern* : : *pattern*
	|	# *integer-literal pattern*
	|	> *pattern*

The pattern *ident*, where *ident* is an identifier, matches any value, and binds the identifier to this value. The pattern `#` *n pattern* matches a list, a vector or a tuple whose *n*-th element matches *pattern*. The pattern `>` *pattern* matches any constructed value whose argument matches *pattern*, regardless of the constructor; it is a shortcut for skipping a constructor.

Example: assuming the value of `a` is `Constr{x = [1;2;3;4]}`, the command `match a > {x = # 2 k}` prints `k = 3`.

`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.

9.8 Controlling the debugger

9.8.1 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.

9.8.2 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 Caml Light runtime `camlrun` on the program. Rarely useful; moreover it prevents the debugging of programs compiled in "custom runtime" mode.

`set loadingmode manual`
: The user starts manually the program, when asked by the debugger. Allows remote debugging (see section 9.8.7).

9.8.3 Search path for files

The debugger searches for source files and compiled interface files in a list of directories, the search path. The search path initially contains the current directory `.` and the standard library directory. The `directory` command adds directories to the path.

Whenever the search path is modified, the debugger will clear any information it may have cached about the files.

`directory` *directorynames*
: Add the given directories to the search path. These directories are added at the front, and will therefore be searched first.

`directory`
: Reset the search path. This requires confirmation.

9.8.4 Working directory

Each time a program is started in the debugger, it inherits its working directory from the current working directory of the debugger. This working directory is initially whatever it inherited from its parent process (typically the shell), but you can specify a new working directory in the debugger with the `cd` command or the `-cd` command-line option.

`cd` *directory*
: Set the working directory for `camldebug` to *directory*.

`pwd` Print the working directory for `camldebug`.

9.8.5 Module management

Like the Caml Light compiler, the debugger maintains a list of opened modules in order to resolves variable name ambiguities. The opened modules also affect the printing of values: whether fully qualified names or short names are used for constructors and record labels.

When a program is executed, the debugger automatically opens the modules of the standard library it uses.

`open` *modules*
: Open the given modules.

`close` *modules*
: Close the given modules.

`info modules`
: List the modules used by the program, and the open modules.

9.8.6 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.

9.8.7 Communication between the debugger and the program

The debugger communicate with the program being debugged through a Unix socket. You may need to change the socket name, for example if you need to run the debugger on a machine and your program on another.

`set socket` *socket*
: Use *socket* for communication with the program. *socket* can be either a file name, or an Internet port specification *host*:*port*, where *host* is a host name or an Internet address in dot notation, and *port* is a port number on the host.

On the debugged program side, the socket name is passed either by the `-D` command line option to `camlrun`, or through the `CAML_DEBUG_SOCKET` environment variable.

9.8.8 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).

9.9 Miscellaneous commands

`list` [*module*] [*beginning*] [*end*]
: List the source of module *module*, from line number *beginning* to line number *end*. By default, 20 lines of the current module are displayed, starting 10 lines before the current position.

`source` *filename*
: Read debugger commands from the script *filename*.

Chapter 10

Profiling (camlpro)

This chapter describes how the execution of Caml Light programs can be profiled, by recording how many times functions are called, branches of conditionals are taken, ...

Mac: This command is not available.

PC: This command is not available.

10.1 Compiling for profiling

Before profiling an execution, the program must be compiled in profiling mode, using the `-p` option to the batch compiler `camlc` (see chapter 4). When compiling modules separately, the `-p` option must be given both when compiling the modules (production of `.zo` files) and when linking them together.

The amount of profiling information can be controlled by adding one or several letters after the `-p` option, indicating which parts of the program should be profiled:

`a` all options

`f` function calls : a count point is set at the beginning of function bodies

`i` **if** ...**then** ...**else** ... : count points are set in both **then** branch and **else** branch

`l` **while, for** loops: a count point is set at the beginning of the loop body

`m` **match** branches: a count point is set at the beginning of the body of each branch

`t` **try** ...**with** ... branches: a count point is set at the beginning of the body of each branch

For instance, compiling with `-pfilm` profiles function calls, if...then ...else..., loops and pattern matching.

The `-p` option without additional letters defaults to `-pfm`, meaning that only function calls and pattern matching are profiled.

10.2 Profiling an execution

Running a bytecode executable file that has been compiled and linked with `-p` records the execution counts for the specified parts of the program and saves them in a file called `camlpro.dump` in the current directory.

More precisely, the dump file `camlpro.dump` is written when the `io__exit` function is called. The linker, called with the `-p` option, adds `io__exit 0` as the last phrase of the bytecode executable, in case the original program never calls `io__exit`. However, if the program terminates with an `uncaught exception`, the dump file will not be produced.

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.

10.3 Printing profiling information

The `camlpro` command produces a source listing of the program modules where execution counts have been inserted as comments. For instance,

```
camlpro foo.ml
```

prints the source code for the `foo` module, with comments indicating how many times the functions in this module have been called. Naturally, this information is accurate only if the source file has not been modified since the profiling execution took place.

The following options are recognized by `camlpro`:

compiler options `-stdlib, -I, -include, -O, -open, -i, -lang`
: See chapter 4 for the detailed usage.

`-f` *dumpfile*
: Specifies an alternate dump file of profiling information

`-F` *string*
: Specifies an additional string to be output with profiling information. By default, camlpro will annotate progams 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 *)`.

An additional argument specifies the output file. For instance

```
camlpro -f ../test/camlpro.dump foo.ml foo_profiled.ml
```

will save the annotated program in file `foo_profiled.ml`. Otherwise, the annotated program is written on the standard output.

10.4 Known bugs

The following situation (file `x.ml`)

```
let a = 1;;
x__a ;;
```

will break the profiler. More precisely, one should avoid to refer to symbols of the current module with the qualified symbol syntax.

Chapter 11

Using Caml Light under Emacs

This chapter describes how Caml Light can be used in conjunction with Gnu Emacs version 19 (version 18 is also partially supported).

Unix: The Emacs Lisp files implementing the Caml/Emacs interface are in `contrib/camlmode` in the distribution.

Mac: The Caml/Emacs interface is not available.

PC: The Caml/Emacs interface is not available.

11.1 Updating your `.emacs`

The following initializations must be added to your `.emacs` file:

```
(setq auto-mode-alist (cons '("\\.ml[iylp]?" . caml-mode) auto-mode-alist))
(autoload 'caml-mode "caml" "Major mode for editing Caml code." t)
(autoload 'run-caml "inf-caml" "Run an inferior Caml process." t)
(autoload 'camldebug "camldebug" "Run the Caml debugger." t)
```

11.2 The `caml` editing mode

The `caml-mode` function is a major editing mode for Caml source files. It provides the correct syntax tables, comment syntax, ... for the Caml language. An extremely crude indentation facility is provided, as well as a slightly enhanced `next-error` command (to display the location of a compilation error). The following key bindings are performed:

`TAB` (**function** `caml-indent-command`)
: At the beginning of a line, indent that line like the line above. Successive `TAB`s increase the indentation level by 2 spaces (by default; can be set with the `caml-mode-indentation` variable).

`M-TAB` (**function** `caml-unindent-command`)
: Decrease the indentation level of the current phrase.

`C-x ‘` **(function** `caml-next-error`**)**
: Display the next compilation error, just as `next-error` does. In addition, it puts the point and the mark around the exact location of the error (the subexpression that caused the error). Under Emacs 19, that subexpression is also highlighted.

`M-C-h` **(function** `caml-mark-phrase`**)**
: Mark the Caml phrase that contains the point: the point is put at the beginning of the phrase and the mark at the end. Phrases are delimited by `;;` (the final double-semicolon). This function does not properly ignore `;;` inside string literals or comments.

`C-x SPC`
: When the Caml debugger is running as an inferior process (section 11.4 below), set a breakpoint at the current position of the point.

`M-C-x` **or** `C-c C-e` **(function** `caml-eval-phrase`**)**
: When a Caml toplevel is running as an inferior process (section 11.3 below), send it the the Caml phrase that contains the point. The phrase will then be evaluated by the inferior toplevel as usual. The phrase is delimited by `;;` as described for the `caml-mark-phrase` command.

`C-c C-r` **(function** `caml-eval-region`**)**
: Send the region to a Caml toplevel running in an inferior process.

11.3 Running the toplevel as an inferior process

`M-x run-caml` starts a Caml toplevel with input and output in an Emacs buffer named `*inferior-caml*`. This gives you the full power of Emacs to edit the input to the Caml toplevel. An history of input lines is maintained, as in Shell mode. This includes the following commands (see the function `comint-mode` for a complete description):

`RET` Send the current line to the toplevel.

`M-n` **and** `M-p`
: Move to the next or previous line in the history.

`M-r` **and** `M-s`
: Regexp search in the history.

`C-c C-c`
: Send a break (interrupt signal) to the Caml toplevel.

Phrases can also be sent to the Caml toplevel for evaluation from any buffer in Caml mode, using `M-C-x`, `C-c C-e` or `C-c C-r`.

11.4 Running the debugger as an inferior process

The Caml debugger is started by the command `M-x camldebug`, with argument the name of the executable file *progname* to debug. Communication with the debugger takes place in an Emacs buffer named `*camldebug-`*progname*`*`. The editing and history facilities of Shell mode are available for interacting with the debugger.

In addition, Emacs displays the source files containing the current event (the current position in the program execution) and highlights the location of the event. This display is updated synchronously with the debugger action.

The following bindings for the most common debugger commands are available in the `*camldebug-`*progname*`*` buffer (see section 9.3 for a full explanation of the commands):

`M-r` `run` command: execute the program forward.

`M-s` `step` command: execute the program one step forward.

`M-b` `back` command: execute the program one step backward.

`M-l` `last` command: go back one step in the command history.

`C-c >`
: `down` command: select the stack frame below the current frame.

`C-c <`
: `up` command: select the stack frame above the current frame.

`C-c C-f`
: `finish` command: run till the current function returns.

In a buffer in Caml editing mode, `C-x SPC` sets a breakpoint at the current position of the point.

Chapter 12

Interfacing C with Caml Light

This chapter describes how user-defined primitives, written in C, can be added to the Caml Light runtime system and called from Caml Light code.

12.1 Overview and compilation information

12.1.1 Declaring primitives

User primitives are declared in a module interface (a `.mli` file), in the same way as a regular ML value, except that the declaration is followed by the `=` sign, the function arity (number of arguments), and the name of the corresponding C function. For instance, here is how the `input` primitive is declared in the interface for the standard library module `io`:

```
value input : in_channel -> string -> int -> int -> int
            = 4 "input"
```

Primitives with several arguments are always curried. The C function does not necessarily have the same name as the ML function.

Values thus declared primitive in a module interface must not be implemented in the module implementation (the `.ml` file). They can be used inside the module implementation.

12.1.2 Implementing primitives

User primitives with arity $n \leq 5$ are implemented by C functions that take n arguments of type `value`, and return a result of type `value`. The type `value` is the type of the representations for Caml Light values. It encodes objects of several base types (integers, floating-point numbers, strings, ...), as well as Caml Light data structures. The type `value` and the associated conversion functions and macros are described in details below. For instance, here is the declaration for the C function implementing the `input` primitive:

```
value input(channel, buffer, offset, length)
        value channel, buffer, offset, length;
{
 ...
}
```

When the primitive function is applied in a Caml Light program, the C function is called with the values of the expressions to which the primitive is applied as arguments. The value returned by the function is passed back to the Caml Light program as the result of the function application.

User primitives with arity greater than 5 are implemented by C functions that receive two arguments: a pointer to an array of Caml Light values (the values for the arguments), and an integer which is the number of arguments provided:

```
value prim_with_lots_of_args(argv, argn)
        value * argv;
        int argn;
{
  ... argv[0] ...;              /* The first argument */
  ... argv[6] ...;              /* The seventh argument */
}
```

Implementing a user primitive is actually two separate tasks: on the one hand, decoding the arguments to extract C values from the given Caml Light values, and encoding the return value as a Caml Light value; on the other hand, actually computing the result from the arguments. Except for very simple primitives, it is often preferable to have two distinct C functions to implement these two tasks. The first function actually implements the primitive, taking native C values as arguments and returning a native C value. The second function, often called the "stub code", is a simple wrapper around the first function that converts its arguments from Caml Light values to C values, call the first function, and convert the returned C value to Caml Light value. For instance, here is the stub code for the `input` primitive:

```
value input(channel, buffer, offset, length)
        value channel, buffer, offset, length;
{
  return Val_long(getblock((struct channel *) channel,
                           &Byte(buffer, Long_val(offset)),
                           Long_val(length)));
}
```

(Here, `Val_long`, `Long_val` and so on are conversion macros for the type `value`, that will be described later.) The hard work is performed by the function `getblock`, which is declared as:

```
long getblock(channel, p, n)
     struct channel * channel;
     char * p;
     long n;
{
  ...
}
```

To write C code that operates on Caml Light values, the following include files are provided:

Include file	Provides
`mlvalues.h`	definition of the `value` type, and conversion macros
`alloc.h`	allocation functions (to create structured Caml Light objects)
`memory.h`	miscellaneous memory-related functions (for in-place modification of structures, etc).

These files reside in the Caml Light standard library directory (usually `/usr/local/lib/caml-light`).

12.1.3 Linking C code with Caml Light code

The Caml Light runtime system comprises three main parts: the bytecode interpreter, the memory manager, and a set of C functions that implement the primitive operations. Some bytecode instructions are provided to call these C functions, designated by their offset in a table of functions (the table of primitives).

In the default mode, the Caml Light linker produces bytecode for the standard runtime system, with a standard set of primitives. References to primitives that are not in this standard set result in the "unavailable C primitive" error.

In the "custom runtime" mode, the Caml Light linker scans the bytecode object files (`.zo` files) and determines the set of required primitives. Then, it builds a suitable runtime system, by calling the native code linker with:

- the table of the required primitives
- a library that provides the bytecode interpreter, the memory manager, and the standard primitives
- libraries and object code files (`.o` files) mentioned on the command line for the Caml Light linker, that provide implementations for the user's primitives.

This builds a runtime system with the required primitives. The Caml Light 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 `camlc` command with:

- the `-custom` option
- the names of the desired Caml Light object files (`.zo` files)
- the names of the C object files and libraries (`.o` and `.a` files) that implement the required primitives. (Libraries can also be specified with the usual `-l` syntax.)

12.2 The `value` type

All Caml Light objects are represented by the C type `value`, defined in the include file `mlvalues.h`, along with macros to manipulate values of that type. An object of type `value` is either:

- an unboxed integer

- a pointer to a block inside the heap (such as the blocks allocated through one of the `alloc_*` functions below)
- a pointer to an object outside the heap (e.g., a pointer to a block allocated by `malloc`, or to a C variable).

12.2.1 Integer values

Integer values encode 31-bit signed integers. They are unboxed (unallocated).

12.2.2 Blocks

Blocks in the heap are garbage-collected, and therefore have strict structure constraints. Each block includes a header containing the size of the block (in words), and the tag of the block. The tag governs how the contents of the blocks are structured. A tag lower than `No_scan_tag` indicates a structured block, containing well-formed values, which is recursively traversed by the garbage collector. A tag greater than or equal to `No_scan_tag` indicates a raw block, whose contents are not scanned by the garbage collector. For the benefits of ad-hoc polymorphic primitives such as equality and structured input-output, structured and raw blocks are further classified according to their tags as follows:

Tag	Contents of the block
0 to `No_scan_tag` − 1	A structured block (an array of Caml Light objects). Each field is a `value`.
`Closure_tag`	A closure representing a functional value. The first word is a pointer to a piece of bytecode, the second word is a `value` containing the environment.
`String_tag`	A character string.
`Double_tag`	A double-precision floating-point number.
`Abstract_tag`	A block representing an abstract datatype.
`Final_tag`	A block representing an abstract datatype with a "finalization" function, to be called when the block is deallocated.

12.2.3 Pointers to outside the heap

Any pointer to outside the heap can be safely cast to and from the type `value`. This includes pointers returned by `malloc`, and pointers to C variables obtained with the `&` operator.

12.3 Representation of Caml Light data types

This section describes how Caml Light data types are encoded in the `value` type.

12.3.1 Atomic types

Caml type	Encoding
`int`	Unboxed integer values.
`char`	Unboxed integer values (ASCII code).
`float`	Blocks with tag `Double_tag`.
`string`	Blocks with tag `String_tag`.

12.3.2 Product types

Tuples and arrays are represented by pointers to blocks, with tag 0.

Records are also represented by zero-tagged blocks. The ordering of labels in the record type declaration determines the layout of the record fields: the value associated to the label declared first is stored in field 0 of the block, the value associated to the label declared next goes in field 1, and so on.

12.3.3 Concrete types

Constructed terms are represented by blocks whose tag encode the constructor. The constructors for a given concrete type are numbered from 0 to the number of constructors minus one, following the order in which they appear in the concrete type declaration. Constant constructors are represented by zero-sized blocks (atoms), tagged with the constructor number. Non-constant constructors declared with a n-tuple as argument are represented by a block of size n, tagged with the constructor number; the n fields contain the components of its tuple argument. Other non-constant constructors are represented by a block of size 1, tagged with the constructor number; the field 0 contains the value of the constructor argument. Example:

Constructed term	Representation
`()`	Size = 0, tag = 0
`false`	Size = 0, tag = 0
`true`	Size = 0, tag = 1
`[]`	Size = 0, tag = 0
`h::t`	Size = 2, tag = 1, first field = `h`, second field = `t`

12.4 Operations on values

12.4.1 Kind tests

- `Is_int(`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.

12.4.2 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

12.4.3 Accessing blocks

- `Wosize_val`(v) returns the size of value v, in words, excluding the header.
- `Tag_val`(v) returns the tag of value v.
- `Field`(v, n) returns the value contained in the n^{th} field of the structured block v. Fields are numbered from 0 to `Wosize_val`(v) $- 1$.
- `Code_val`(v) returns the code part of the closure v.
- `Env_val`(v) returns the environment part of the closure v.
- `string_length`(v) returns the length (number of characters) of the string v.
- `Byte`(v, n) returns the n^{th} character of the string v, with type `char`. Characters are numbered from 0 to `string_length`(v) $- 1$.
- `Byte_u`(v, n) returns the n^{th} character of the string v, with type `unsigned char`. Characters are numbered from 0 to `string_length`(v) $- 1$.
- `String_val`(v) returns a pointer to the first byte of the string v, with type `char *`. This pointer is a valid C string: there is a null character after the last character in the string. However, Caml Light strings can contain embedded null characters, that will confuse the usual C functions over strings.
- `Double_val`(v) returns the floating-point number contained in value v, with type `double`.

The expressions `Field`(v, n), `Code_val`(v), `Env_val`(v), `Byte`(v, n), `Byte_u`(v, n) and `Double_val`(v) 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).

12.4.4 Allocating blocks

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 to `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.

- `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 `()`, `false` and `[]`; `Atom`(1) represents `true`. (As a convenience, `mlvalues.h` defines the macros `Val_unit`, `Val_false` and `Val_true`.)

- `alloc`(n, t) returns a fresh small block of size $n \leq$ `Max_young_wosize` words, with tag t. If this block is a structured block (i.e. if $t <$ `No_scan_tag`), then the fields of the block (initially containing garbage) must be initialized with legal values (using direct assignment to the fields of the block) before the next allocation.

- `alloc_tuple`(n) returns a fresh small block of size $n \leq$ `Max_young_wosize` words, with tag 0. The fields of this block must be filled with legal values before the next allocation or modification.

- `alloc_shr`(n, t) returns a fresh block of size n, with tag t. The size of the block can be greater than `Max_young_wosize`. (It can also be smaller, but in this case it is more efficient to call `alloc` instead of `alloc_shr`.) If this block is a structured block (i.e. if $t <$ `No_scan_tag`), then the fields of the block (initially containing garbage) must be initialized with legal values (using the `initialize` function described below) before the next allocation.

- `alloc_string`(n) returns a string value of length n characters. The string initially contains garbage.

- `copy_string`(s) returns a string value containing a copy of the null-terminated C string s (a `char *`).

- `copy_double`(d) returns a floating-point value initialized with the `double` d.

- `alloc_array`(f, a) allocates an array of values, calling function f over each element of the input array a to transform it into a value. The array a is an array of pointers terminated by the null pointer. The function f receives each pointer as argument, and returns a value. The zero-tagged block returned by `alloc_array`(f, a) is filled with the values returned by the successive calls to f.

- `copy_string_array`(p) allocates an array of strings, copied from the pointer to a string array p (a `char **`).

12.4.5 Raising exceptions

C functions cannot raise arbitrary exceptions. However, two functions are provided to raise two standard exceptions:

- `failwith`(s), where s is a null-terminated C string (with type `char *`), raises exception `Failure` with argument s.

- `invalid_argument`(s), where s is a null-terminated C string (with type `char *`), raises exception `Invalid_argument` with argument s.

12.5 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.

Rule 1 *After a structured block (a block with tag less than* `No_scan_tag`*) is allocated, all fields of this block must be filled with well-formed values before the next allocation operation. If the block has been allocated with* `alloc` *or* `alloc_tuple`*, filling is performed by direct assignment to the fields of the block:*

```
Field(v, n) = v_n;
```

If the block has been allocated with `alloc_shr`*, filling is performed through the* `initialize` *function:*

```
initialize(&Field(v, n), v_n);
```

The next allocation can trigger a garbage collection. The garbage collector assumes that all structured blocks contain well-formed values. Newly created blocks contain random data, which generally do not represent well-formed values.

If you really need to allocate before the fields can receive their final value, first initialize with a constant value (e.g. `Val_long(0)`), then allocate, then modify the fields with the correct value (see rule 3).

Rule 2 *Local variables containing values must be registered with the garbage collector (using the* `Push_roots` *and* `Pop_roots` *macros), if they are to survive a call to an allocation function.*

Registration is performed with the `Push_roots` and `Pop_roots` macros. `Push_roots(`r`,`n`)` declares an array r of n values and registers them with the garbage collector. The values contained in r`[0]` to r`[`n`-1]` are treated like roots by the garbage collector. A root value has the following properties: if it points to a heap-allocated block, this block (and its contents) will not be reclaimed; moreover, if this block is relocated by the garbage collector, the root value is updated to point to the new location for the block. `Push_roots(`r`,`n`)` must occur in a C block exactly between the last local variable declaration and the first statement in the block. To un-register the roots, `Pop_roots()` must be called before the C block containing `Push_roots(`r`,`n`)` is exited. (Roots are automatically un-registered if a Caml exception is raised.)

Rule 3 *Direct assignment to a field of a block, as in*

```
Field(v, n) = w;
```

is safe only if v is a block newly allocated by `alloc` *or* `alloc_tuple`*; that is, if no allocation took place between the allocation of v and the assignment to the field. In all other cases, never assign directly. If the block has just been allocated by* `alloc_shr`*, use* `initialize` *to assign a value to a field for the first time:*

```
initialize(&Field(v, n), w);
```

Otherwise, you are updating a field that previously contained a well-formed value; then, call the `modify` *function:*

```
modify(&Field(v, n), w);
```

To illustrate the rules above, here is a C function that builds and returns a list containing the two integers given as parameters:

```
value alloc_list_int(i1, i2)
        int i1, i2;
{
  value result;
  Push_roots(r, 1);
  r[0] = alloc(2, 1);                      /* Allocate a cons cell */
  Field(r[0], 0) = Val_int(i2);            /* car = the integer i2 */
  Field(r[0], 1) = Atom(0);                /* cdr = the empty list [] */
  result = alloc(2, 1);                    /* Allocate the other cons cell */
  Field(result, 0) = Val_int(i1);          /* car = the integer i1 */
  Field(result, 1) = r[0];                 /* cdr = the first cons cell */
  Pop_roots();
  return result;
}
```

The "cons" cell allocated first needs to survive the allocation of the other cons cell; hence, the value returned by the first call to `alloc` must be stored in a registered root. The value returned by the second call to `alloc` can reside in the un-registered local variable `result`, since we won't do any further allocation in this function.

In the example above, the list is built bottom-up. Here is an alternate way, that proceeds top-down. It is less efficient, but illustrates the use of `modify`.

```
value alloc_list_int(i1, i2)
        int i1, i2;
{
  value tail;
  Push_roots(r, 1);
  r[0] = alloc(2, 1);                      /* Allocate a cons cell */
  Field(r[0], 0) = Val_int(i1);            /* car = the integer i1 */
  Field(r[0], 1) = Val_int(0);             /* A dummy value
  tail = alloc(2, 1);                      /* Allocate the other cons cell */
  Field(tail, 0) = Val_int(i2);            /* car = the integer i2 */
  Field(tail, 1) = Atom(0);                /* cdr = the empty list [] */
  modify(&Field(r[0], 1), tail);           /* cdr of the result = tail */
  Pop_roots();
  return r[0];
}
```

It would be incorrect to perform `Field(r[0], 1) = tail` directly, because the allocation of `tail` has taken place since `r[0]` was allocated.

12.6 A complete example

This section outlines how the functions from the Unix `curses` library can be made available to Caml Light programs. First of all, here is the interface `curses.mli` that declares the `curses` primitives and data types:

```
type window;;                    (* The type "window" remains abstract *)
value initscr: unit -> window = 1 "curses_initscr"
  and endwin: unit -> unit = 1 "curses_endwin"
  and refresh: unit -> unit = 1 "curses_refresh"
  and wrefresh : window -> unit = 1 "curses_wrefresh"
  and newwin: int -> int -> int -> int -> window = 4 "curses_newwin"
  and mvwin: window -> int -> int -> unit = 3 "curses_mvwin"
  and addch: char -> unit = 1 "curses_addch"
  and mvwaddch: window -> int -> int -> char -> unit = 4 "curses_mvwaddch"
  and addstr: string -> unit = 1 "curses_addstr"
  and mvwaddstr: window -> int -> int -> string -> unit = 4 "curses_mvwaddstr"
;; (* lots more omitted *)
```

To compile this interface:

```
        camlc -c curses.mli
```

To implement these functions, we just have to provide the stub code; the core functions are already implemented in the `curses` library. The stub code file, `curses.o`, looks like:

```
#include <curses.h>
#include <mlvalues.h>

value curses_initscr(unit)
        value unit;
{
  return (value) initscr();     /* OK to coerce directly from WINDOW * to value
                                   since that's a block created by malloc() */
}

value curses_wrefresh(win)
        value win;
{
  wrefresh((value) win);
  return Val_unit;               /* Or Atom(0) */
}

value curses_newwin(nlines, ncols, x0, y0)
        value nlines, ncols, x0, y0;
{
  return (value) newwin(Int_val(nlines), Int_val(ncols),
                        Int_val(x0), Int_val(y0));
}

value curses_addch(c)
        value c;
{
```

```
  addch(Int_val(c));            /* Characters are encoded like integers */
  return Val_unit;
}

value curses_addstr(s)
        value s;
{
  addstr(String_val(s));
  return Val_unit;
}

/* This goes on for pages. */
```

(Actually, it would be better to create a library for the stub code, with each stub code function in a separate file, so that linking would pick only those functions from the `curses` library that are actually used.)

The file `curses.c` can be compiled with:

```
        cc -c -I/usr/local/lib/caml-light curses.c
```

or, even simpler,

```
        camlc -c curses.c
```

(When passed a `.c` file, the `camlc` command simply calls `cc` on that file, with the right `-I` option.)

Now, here is a sample Caml Light program `test.ml` that uses the `curses` module:

```
#open "curses";;
let main_window = initscr () in
let small_window = newwin 10 5 20 10 in
  mvwaddstr main_window 10 2 "Hello";
  mvwaddstr small_window 4 3 "world";
  refresh();
  for i = 1 to 100000 do () done;
  endwin()
;;
```

To compile this program, run:

```
        camlc -c test.ml
```

Finally, to link everything together:

```
        camlc -custom -o test test.zo curses.o -lcurses
```

Part IV

The Caml Light library

Chapter 13

The core library

This chapter describes the functions provided by the Caml Light core library. This library is special in two ways:

- It is automatically linked with the user's object code files by the `camlc` command (chapter 4). Hence, the globals defined by these libraries can be used in standalone programs without having to add any `.zo` file on the command line for the linking phase. Similarly, in interactive use, these globals can be used in toplevel phrases without having to load any `.zo` file in memory.
- The interfaces for the modules below are 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 these modules, without adding `#open` directives. Actually, the list of automatically opened modules depend on the `-O` option given to the compiler or to the toplevel system:

`-O` option	Opened modules (reverse opening order)
`-O cautious` (default)	`io`, `eq`, `int`, `float`, `ref`, `pair`, `list`, `vect`, `char`, `string`, `bool`, `exc`, `stream`, `builtin`
`-O fast`	`io`, `eq`, `int`, `float`, `ref`, `pair`, `list`, `fvect`, `fchar`, `fstring`, `bool`, `exc`, `stream`, `builtin`
`-O none`	`builtin`

Conventions

For easy reference, the modules are listed below in alphabetical order of module names. For each module, the declarations from its interface file are printed one by one in typewriter font, followed by a short comment. All modules and the identifiers they export are indexed at the end of this report.

13.1 `bool`: boolean operations

The boolean and is written `e1 & e2`. The boolean or is written `e1 or e2`. Both constructs are sequential, left-to-right: `e2` is evaluated only if needed. Actually, `e1 & e2` is equivalent to `if e1 then e2 else false`, and `e1 or e2` is equivalent to `if e1 then true else e2`.

`value prefix not : bool -> bool`

The boolean negation.

`value string_of_bool : bool -> string`

Return a string representing the given boolean.

13.2 `builtin`: base types and constructors

This module defines some types and exceptions for which the language provides special syntax, and are therefore treated specially by the compiler.

```
type int
type float
type string
type char
```

The types of integers, floating-point numbers, character strings, and characters, respectively.

`type exn`

The type of exception values.

`type bool = false | true`

The type of boolean values.

`type 'a vect`

The type of arrays whose elements have type `'a`.

`type unit = ()`

The type of the unit value.

`type 'a list = [] | prefix :: of 'a * 'a list`

The type of lists.

`exception Match_failure of string * int * int`

The exception raised when a pattern-matching fails. The argument indicates the position in the source code of the pattern-matching (source file name, position of the first character of the matching, position of the last character.

13.3 `char`: character operations

```
value int_of_char : char -> int
```

Return the ASCII code of the argument.

```
value 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.

```
value char_for_read : char -> string
```

Return a string representing the given character, with special characters escaped following the lexical conventions of Caml Light.

13.4 `eq`: generic comparisons

```
value prefix = : '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.

```
value prefix <> : 'a -> 'a -> bool
```

Negation of `prefix =`.

```
value prefix < : 'a -> 'a -> bool
value prefix <= : 'a -> 'a -> bool
value prefix > : 'a -> 'a -> bool
value prefix >= : 'a -> 'a -> bool
```

Structural ordering functions. These functions coincide with the usual orderings over integer, string and floating-point numbers, and extend them to a total ordering over all types. The ordering is compatible with `prefix =`. As in the case of `prefix =`, mutable structures are compared by contents. Comparison between functional values raises `Invalid_argument`. Comparison between cyclic structures may not terminate.

```
value compare: 'a -> 'a -> int
```

`compare x y` returns `0` if `x=y`, a negative integer if `x<y`, and a positive integer if `x>y`. The same restrictions as for `=` apply. `compare` can be used as the comparison function required by the `set` and `map` modules.

```
value min: 'a -> 'a -> 'a
```

Return the smaller of the two arguments.

`value max: 'a -> 'a -> 'a`

Return the greater of the two arguments.

`value prefix == : 'a -> 'a -> bool`

`e1 == e2` tests for physical equality of `e1` and `e2`. On integers and characters, it is the same as structural equality. On mutable structures, `e1 == e2` is true if and only if physical modification of `e1` also affects `e2`. On non-mutable structures, the behavior of `prefix ==` is implementation-dependent, except that `e1 == e2` implies `e1 = e2`.

`value prefix != : 'a -> 'a -> bool`

Negation of `prefix ==`.

13.5 `exc`: exceptions

`value raise : exn -> 'a`

Raise the given exception value.

A few general-purpose predefined exceptions.

`exception Out_of_memory`

Raised by the garbage collector, when there is insufficient memory to complete the computation.

`exception Invalid_argument of string`

Raised by library functions to signal that the given arguments do not make sense.

`exception Failure of string`

Raised by library functions to signal that they are undefined on the given arguments.

`exception Not_found`

Raised by search functions when the desired object could not be found.

`exception Exit`

This exception is not raised by any library function. It is provided for use in your programs.

`value failwith : string -> 'a`

Raise exception `Failure` with the given string.

`value invalid_arg : string -> 'a`

Raise exception `Invalid_argument` with the given string.

13.6 `fchar`: character operations, without sanity checks

This module implements the same functions as the `char` module, but does not perform bound checks on the arguments of the functions. The functions are therefore faster than those in the `char` module, but calling these functions with incorrect parameters (that is, parameters that would cause the `Invalid_argument` exception to be raised by the corresponding functions in the `char` module) can crash the program.

13.7 `float`: operations on floating-point numbers

```
value int_of_float : float -> int
```

Truncate the given float to an integer value. The result is unspecified if it falls outside the range of representable integers.

```
value float_of_int : int -> float
```

Convert an integer to floating-point.

```
value minus : float -> float
value minus_float : float -> float
```

Unary negation.

```
value prefix + : float -> float -> float
value prefix +. : float -> float -> float
value add_float : float -> float -> float
```

Addition.

```
value prefix - : float -> float -> float
value prefix -. : float -> float -> float
value sub_float : float -> float -> float
```

Subtraction.

```
value prefix * : float -> float -> float
value prefix *. : float -> float -> float
value mult_float : float -> float -> float
```

Product.

```
value prefix / : float -> float -> float
value prefix /. : float -> float -> float
value div_float : float -> float -> float
```

Division. Raise `Division_by_zero` if the dividend is 0.0.

```
value prefix ** : float -> float -> float
value prefix **. : float -> float -> float
value power : float -> float -> float
```

Exponentiation.

```
value eq_float : float -> float -> bool
value prefix =. : float -> float -> bool
```

Floating-point equality. Equivalent to generic equality, just faster.

```
value neq_float : float -> float -> bool
value prefix <>. : float -> float -> bool
```

Negation of `eq_float`.

```
value prefix <. : float -> float -> bool
value lt_float : float -> float -> bool
value prefix >. : float -> float -> bool
value gt_float : float -> float -> bool
value prefix <=. : float -> float -> bool
value le_float : float -> float -> bool
value prefix >=. : float -> float -> bool
value ge_float : float -> float -> bool
```

Usual comparisons between floating-point numbers.

```
value exp : float -> float
value log : float -> float
value sqrt : float -> float
value sin : float -> float
value cos : float -> float
value tan : float -> float
value asin : float -> float
value acos : float -> float
value atan : float -> float
value atan2 : float -> float -> float
```

Usual transcendental functions on floating-point numbers.

```
value abs_float : float -> float
```

Return the absolute value of the argument.

```
value string_of_float : float -> string
```

Convert the given float to its decimal representation.

```
value float_of_string : string -> float
```

Convert the given string to a float, in decimal. The result is unspecified if the given string is not a valid representation of a float.

13.8 `fstring`: string operations, without sanity checks

This module implements the same functions as the `string` module, but does not perform bound checks on the arguments of the functions. The functions are therefore faster than those in the `string` module, but calling these functions with incorrect parameters (that is, parameters that would cause the `Invalid_argument` exception to be raised by the corresponding functions in the `string` module) can crash the program.

13.9 `fvect`: operations on vectors, without sanity checks

This module implements the same functions as the `vect` module, but does not perform bound checks on the arguments of the functions. The functions are therefore faster than those in the `vect` module, but calling these functions with incorrect parameters (that is, parameters that would cause the `Invalid_argument` exception to be raised by the corresponding functions in the `vect` module) can crash the program.

13.10 `int`: operations on integers

Integers are 31 bits wide (or 63 bits on 64-bit processors). All operations are taken modulo 2^{31} (or 2^{63}). They do not fail on overflow.

```
exception Division_by_zero
value minus : int -> int
value minus_int : int -> int
```

Unary negation. You can write `-e` instead of `minus e`.

```
value succ : int -> int
```

`succ x` is `x+1`.

```
value pred : int -> int
```

`pred x` is `x-1`.

```
value prefix + : int -> int -> int
value add_int : int -> int -> int
```

Addition.

```
value prefix - : int -> int -> int
value sub_int : int -> int -> int
```

Subtraction.

```
value prefix * : int -> int -> int
value mult_int : int -> int -> int
```

Multiplication.

```
value prefix / : int -> int -> int
value div_int : int -> int -> int
value prefix quo : int -> int -> int
```

Integer division. Raise `Division_by_zero` if the second argument is 0. Give unpredictable results if either argument is negative.

```
value prefix mod : int -> int -> int
```

Remainder. Raise `Division_by_zero` if the second argument is 0. Give unpredictable results if either argument is negative.

```
value eq_int : int -> int -> bool
```

Integer equality. Equivalent to generic equality, just faster.

```
value neq_int : int -> int -> bool
```

Negation of `eq_int`.

```
value lt_int : int -> int -> bool
value gt_int : int -> int -> bool
value le_int : int -> int -> bool
value ge_int : int -> int -> bool
```

Usual comparisons between integers.

```
value abs : int -> int
```

Return the absolute value of the argument.

Bitwise operations

```
value prefix land : int -> int -> int
```

Bitwise logical and.

```
value prefix lor : int -> int -> int
```

Bitwise logical or.

```
value prefix lxor : int -> int -> int
```

Bitwise logical exclusive or.

```
value lnot : int -> int
```

Bitwise complement

```
value prefix lsl : int -> int -> int
value lshift_left : int -> int -> int
```

`n lsl m`, or equivalently `lshift_left n m`, shifts `n` to the left by `m` bits.

`value prefix 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 sign.

```
value prefix asr : int -> int -> int
value lshift_right : int -> int -> int
```

> `n asr m`, or equivalently `lshift_right n m`, shifts `n` to the right by `m` bits. This is an arithmetic shift: the sign bit is replicated.

Conversion functions

`value string_of_int : int -> string`

> Convert the given integer to its decimal representation.

`value int_of_string : string -> int`

> Convert the given string to an integer, in decimal (by default) or in hexadecimal, octal or binary if the string begins with `0x`, `0o` or `0b`. Raise `Failure "int_of_string"` if the given string is not a valid representation of an integer.

13.11 `io`: buffered input and output

```
type in_channel
type out_channel
```

> The abstract types of input channels and output channels.

`exception End_of_file`

> Raised when an operation cannot complete, because the end of the file has been reached.

```
value stdin : in_channel
value std_in : in_channel
value stdout : out_channel
value std_out : out_channel
value stderr : out_channel
value std_err : out_channel
```

> The standard input, standard output, and standard error output for the process. `std_in`, `std_out` and `std_err` are respectively synonymous with `stdin`, `stdout` and `stderr`.

`value exit : int -> 'a`

> Flush all pending writes on `std_out` and `std_err`, and terminate the process, returning the given status code to the operating system (usually 0 to indicate no errors, and a small positive integer to indicate failure.) This function should be called at the end of all standalone programs that output results on `std_out` or `std_err`; otherwise, the program may appear to produce no output, or its output may be truncated.

Output functions on standard output

`value print_char : char -> unit`

Print the character on standard output.

`value print_string : string -> unit`

Print the string on standard output.

`value print_int : int -> unit`

Print the integer, in decimal, on standard output.

`value print_float : float -> unit`

Print the floating-point number, in decimal, on standard output.

`value print_endline : string -> unit`

Print the string, followed by a newline character, on standard output.

`value 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

`value prerr_char : char -> unit`

Print the character on standard error.

`value prerr_string : string -> unit`

Print the string on standard error.

`value prerr_int : int -> unit`

Print the integer, in decimal, on standard error.

`value prerr_float : float -> unit`

Print the floating-point number, in decimal, on standard error.

`value prerr_endline : string -> unit`

Print the string, followed by a newline character on standard error and flush standard error.

Input functions on standard input

`value 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.

`value read_int : unit -> int`

Flush standard output, then read one line from standard input and convert it to an integer. Raise `Failure "int_of_string"` if the line read is not a valid representation of an integer.

`value read_float : unit -> float`

Flush standard output, then read one line from standard input and convert it to a floating-point number. The result is unspecified if the line read is not a valid representation of a floating-point number.

General output functions

`value open_out : string -> out_channel`

Open the named file for writing, and return a new output channel on that file, positionned at the beginning of the file. The file is truncated to zero length if it already exists. It is created if it does not already exists. Raise `sys__Sys_error` if the file could not be opened.

`value open_out_bin : string -> out_channel`

Same as `open_out`, but the file is opened in binary mode, so that no translation takes place during writes. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_out`.

`value open_out_gen : sys__open_flag list -> int -> string -> out_channel`

`open_out_gen mode rights filename` opens the file named `filename` for writing, as above. The extra argument `mode` specify the opening mode (see `sys__open`). The extra argument `rights` specifies the file permissions, in case the file must be created (see `sys__open`). `open_out` and `open_out_bin` are special cases of this function.

`value open_descriptor_out : int -> out_channel`

`open_descriptor_out fd` returns a buffered output channel writing to the file descriptor `fd`. The file descriptor `fd` must have been previously opened for writing, else the behavior is undefined.

`value 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 `std_out` and `std_err` at the right time.

`value output_char : out_channel -> char -> unit`

Write the character on the given output channel.

`value output_string : out_channel -> string -> unit`

Write the string on the given output channel.

`value output : out_channel -> string -> int -> int -> unit`

`output chan buff ofs len` writes `len` characters from string `buff`, starting at offset `ofs`, to the output channel `chan`. Raise `Invalid_argument "output"` if `ofs` and `len` do not designate a valid substring of `buff`.

`value 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.

`value output_binary_int : out_channel -> int -> unit`

Write one integer in binary format on the given output channel. The only reliable way to read it back is through the `input_binary_int` function. The format is compatible across all machines for a given version of Caml Light.

`value 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`. The format is compatible across all machines for a given version of Caml Light.

`value output_compact_value : out_channel -> 'a -> unit`

Same as `output_value`, but uses a different format, which occupies less space on the file, but takes more time to generate and read back.

`value 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.

`value pos_out : out_channel -> int`

Return the current writing position for the given channel.

`value out_channel_length : out_channel -> int`

Return the total length (number of characters) of the given channel. This works only for regular files. On files of other kinds, the result is meaningless.

`value close_out : out_channel -> unit`

Close the given channel, flushing all buffered write operations. The behavior is unspecified if any of the functions above is called on a closed channel.

General input functions

`value open_in : string -> in_channel`

Open the named file for reading, and return a new input channel on that file, positionned at the beginning of the file. Raise `sys__Sys_error` if the file could not be opened.

`value open_in_bin : string -> in_channel`

Same as `open_in`, but the file is opened in binary mode, so that no translation takes place during reads. On operating systems that do not distinguish between text mode and binary mode, this function behaves like `open_in`.

`value open_in_gen : sys__open_flag list -> int -> string -> in_channel`

`open_in_gen mode rights filename` opens the file named `filename` for reading, as above. The extra arguments `mode` and `rights` specify the opening mode and file permissions (see `sys__open`). `open_in` and `open_in_bin` are special cases of this function.

`value open_descriptor_in : int -> in_channel`

`open_descriptor_in fd` returns a buffered input channel reading from the file descriptor `fd`. The file descriptor `fd` must have been previously opened for reading, else the behavior is undefined.

`value 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.

`value 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.

`value input : in_channel -> string -> int -> int -> int`

`input chan buff ofs len` attempts to read `len` characters from channel `chan`, storing them in string `buff`, starting at character number `ofs`. It returns the actual number of characters read, between 0 and `len` (inclusive). A return value of 0 means that the end of file was reached. A return value between 0 and `len` exclusive means that no more characters were available at that time; `input` must be called again to read the remaining characters, if desired. Exception `Invalid_argument "input"` is raised if `ofs` and `len` do not designate a valid substring of `buff`.

`value really_input : in_channel -> string -> int -> int -> unit`

`really_input chan buff ofs len` reads `len` characters from channel `chan`, storing them in string `buff`, starting at character number `ofs`. Raise `End_of_file` if the end of file is reached before `len` characters have been read. Raise `Invalid_argument "really_input"` if `ofs` and `len` do not designate a valid substring of `buff`.

`value input_byte : in_channel -> int`

Same as `input_char`, but return the 8-bit integer representing the character. Raise `End_of_file` if an end of file was reached.

`value input_binary_int : in_channel -> int`

Read an integer encoded in binary format from the given input channel. See `output_binary_int`. Raise `End_of_file` if an end of file was reached while reading the integer.

`value input_value : in_channel -> 'a`

Read the representation of a structured value, as produced by `output_value` or `output_compact_value`, and return the corresponding value. This is not type-safe. The type of the returned object is not `'a` properly speaking: the returned object has one unique type, which cannot be determined at compile-time. The programmer should explicitly give the expected type of the returned value, using the following syntax: `(input_value chan : type)`. The behavior is unspecified if the object in the file does not belong to the given type.

`value 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.

`value pos_in : in_channel -> int`

Return the current reading position for the given channel.

`value in_channel_length : in_channel -> int`

Return the total length (number of characters) of the given channel. This works only for regular files. On files of other kinds, the result is meaningless.

`value close_in : in_channel -> unit`

Close the given channel. Anything can happen if any of the functions above is called on a closed channel.

13.12 `list`: operations on lists

`value list_length : 'a list -> int`

Return the length (number of elements) of the given list.

`value prefix @ : 'a list -> 'a list -> 'a list`

List concatenation.

`value hd : 'a list -> 'a`

Return the first element of the given list. Raise `Failure "hd"` if the list is empty.

`value tl : 'a list -> 'a list`

Return the given list without its first element. Raise `Failure "tl"` if the list is empty.

`value rev : 'a list -> 'a list`

List reversal.

`value map : ('a -> 'b) -> 'a list -> 'b list`

`map f [a1; ...; an]` applies function `f` to `a1, ..., an`, and builds the list `[f a1; ...; f an]` with the results returned by `f`.

`value do_list : ('a -> 'b) -> 'a list -> unit`

`do_list f [a1; ...; an]` applies function `f` in turn to `a1; ...; an`, discarding all the results. It is equivalent to `begin f a1; f a2; ...; f an; () end`.

`value it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a`

`it_list f a [b1; ...; bn]` is `f (... (f (f a b1) b2) ...) bn`.

`value list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b`

`list_it f [a1; ...; an] b` is `f a1 (f a2 (... (f an b) ...))`.

`value map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list`

`map2 f [a1; ...; an] [b1; ...; bn]` is `[f a1 b1; ...; f an bn]`. Raise `Invalid_argument "map2"` if the two lists have different lengths.

`value do_list2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> unit`

`do_list2 f [a1; ...; an] [b1; ...; bn]` calls in turn `f a1 b1; ...; f an bn`, discarding the results. Raise `Invalid_argument "do_list2"` if the two lists have different lengths.

`value it_list2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a`

`it_list2 f a [b1; ...; bn] [c1; ...; cn]` is `f (... (f (f a b1 c1) b2 c2) ...) bn cn`. Raise `Invalid_argument "it_list2"` if the two lists have different lengths.

`value list_it2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c`

`list_it2 f [a1; ...; an] [b1; ...; bn] c` is `f a1 b1 (f a2 b2 (... (f an bn c) ...))`. Raise `Invalid_argument "list_it2"` if the two lists have different lengths.

value flat_map : ('a -> 'b list) -> 'a list -> 'b list

flat_map f [l1; ...; ln] is (f l1) @ (f l2) @ ... @ (f ln).

value for_all : ('a -> bool) -> 'a list -> bool

for_all p [a1; ...; an] is (p a1) & (p a2) & ... & (p an).

value exists : ('a -> bool) -> 'a list -> bool

exists p [a1; ...; an] is (p a1) or (p a2) or ... or (p an).

value mem : 'a -> 'a list -> bool

mem a l is true if and only if a is structurally equal (see module eq) to an element of l.

value memq : 'a -> 'a list -> bool

memq a l is true if and only if a is physically equal (see module eq) to an element of l.

value except : 'a -> 'a list -> 'a list

except a l returns the list l where the first element structurally equal to a has been removed. The list l is returned unchanged if it does not contain a.

value exceptq : 'a -> 'a list -> 'a list

Same as except, with physical equality instead of structural equality.

value subtract : 'a list -> 'a list -> 'a list

subtract l1 l2 returns the list l1 where all elements structurally equal to one of the elements of l2 have been removed.

value union : 'a list -> 'a list -> 'a list

union l1 l2 appends before list l2 all the elements of list l1 that are not structurally equal to an element of l2.

value intersect : 'a list -> 'a list -> 'a list

intersect l1 l2 returns the list of the elements of l1 that are structurally equal to an element of l2.

value index : 'a -> 'a list -> int

index a l returns the position of the first element of list l that is structurally equal to a. The head of the list has position 0. Raise Not_found if a is not present in l.

value assoc : 'a -> ('a * 'b) list -> 'b

assoc a l returns the value associated with key a in the list of pairs l. That is, assoc a [...; (a,b); ...] = b if (a,b) is the leftmost binding of a in list l. Raise Not_found if there is no value associated with a in the list l.

value assq : 'a -> ('a * 'b) list -> 'b

Same as assoc, but use physical equality instead of structural equality to compare keys.

value mem_assoc : 'a -> ('a * 'b) list -> bool

Same as assoc, but simply return true if a binding exists, and false if no bindings exist for the given key.

13.13 pair: operations on pairs

`value fst : 'a * 'b -> 'a`

Return the first component of a pair.

`value snd : 'a * 'b -> 'b`

Return the second component of a pair.

`value split : ('a * 'b) list -> 'a list * 'b list`

Transform a list of pairs into a pair of lists: `split [(a1,b1); ...; (an,bn)]` is `([a1; ...; an], [b1; ...; bn])`

`value combine : 'a list * 'b list -> ('a * 'b) list`

Transform a pair of lists into a list of pairs: `combine ([a1; ...; an], [b1; ...; bn])` is `[(a1,b1); ...; (an,bn)]`. Raise `Invalid_argument "combine"` if the two lists have different lengths.

`value map_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list`

`map_combine f ([a1; ...; an], [b1; ...; bn])` is `[f (a1, b1); ...; f (an, bn)]`. Raise `invalid_argument "map_combine"` if the two lists have different lengths.

`value do_list_combine : ('a * 'b -> 'c) -> 'a list * 'b list -> unit`

`do_list_combine f ([a1; ...; an], [b1; ...; bn])` calls in turn `f (a1, b1); ...; f (an, bn)`, discarding the results. Raise `Invalid_argument "do_list_combine"` if the two lists have different lengths.

13.14 ref: operations on references

`type 'a ref = ref of mutable 'a`

The type of references (mutable indirection cells) containing a value of type `'a`.

`value prefix ! : 'a ref -> 'a`

`!r` returns the current contents of reference `r`. Could be defined as `fun (ref x) -> x`.

`value prefix := : 'a ref -> 'a -> unit`

`r := a` stores the value of `a` in reference `r`.

`value incr : int ref -> unit`

Increment the integer contained in the given reference. Could be defined as `fun r -> r := succ !r`.

`value decr : int ref -> unit`

Decrement the integer contained in the given reference. Could be defined as `fun r -> r := pred !r`.

13.15 stream: operations on streams

`type 'a stream`

The type of streams containing values of type `'a`.

`exception Parse_failure`

Raised by parsers when none of the first component of the stream patterns is accepted

`exception Parse_error`

Raised by parsers when the first component of a stream pattern is accepted, but one of the following components is rejected

`value stream_next : 'a stream -> 'a`

`stream_next s` returns the first element of stream `s`, and removes it from the stream. Raise `Parse_failure` if the stream is empty.

`value stream_from : (unit -> 'a) -> 'a stream`

`stream_from f` returns the stream which fetches its terminals using the function `f`. This function could be defined as:

```
let rec stream_from f = [< 'f(); stream_from f >]
```

but is implemented more efficiently.

`value stream_of_string : string -> char stream`

`stream_of_string s` returns the stream of the characters in string `s`.

`value stream_of_channel : in_channel -> char stream`

`stream_of_channel ic` returns the stream of characters read on channel `ic`.

`value do_stream : ('a -> 'b) -> 'a stream -> unit`

`do_stream f s` scans the whole stream `s`, applying the function `f` in turn to each terminal encountered

`value stream_check : ('a -> bool) -> 'a stream -> 'a`

`stream_check p` returns the parser which returns the first terminal of the stream if the predicate `p` returns `true` on this terminal, and raises `Parse_failure` otherwise.

`value end_of_stream : 'a stream -> unit`

Return `()` iff the stream is empty, and raise `Parse_failure` otherwise.

`value stream_get : 'a stream -> 'a * 'a stream`

`stream_get s` return the first element of the stream `s`, and a stream containing the remaining elements of `s`. Raise `Parse_failure` if the stream is empty. The stream `s` is not modified. This function makes it possible to access a stream non-destructively.

13.16 string: string operations

`value string_length : string -> int`

Return the length (number of characters) of the given string.

`value nth_char : string -> int -> char`

`nth_char s n` returns character number `n` in string `s`. The first character is character number 0. The last character is character number `string_length s - 1`. Raise `Invalid_argument "nth_char"` if `n` is ouside the range 0 to `(string_length s - 1)`. You can also write `s.[n]` instead of `nth_char s n`.

`value set_nth_char : string -> int -> char -> unit`

`set_nth_char s n c` modifies string `s` in place, replacing the character number `n` by `c`. Raise `Invalid_argument "set_nth_char"` if `n` is ouside the range 0 to `(string_length s - 1)`. You can also write `s.[n] <- c` instead of `set_nth_char s n c`.

`value prefix ^ : string -> string -> string`

`s1 ^ s2` returns a fresh string containing the concatenation of the strings `s1` and `s2`.

`value sub_string : string -> int -> int -> string`

`sub_string s start len` returns a fresh string of length `len`, containing the characters number `start` to `start + len - 1` of string `s`. Raise `Invalid_argument "sub_string"` if `start` and `len` do not designate a valid substring of `s`; that is, if `start < 0`, or `len < 0`, or `start + len > string_length s`.

`value create_string : int -> string`

`create_string n` returns a fresh string of length `n`. The string initially contains arbitrary characters.

`value make_string : int -> char -> string`

`make_string n c` returns a fresh string of length `n`, filled with the character `c`.

`value fill_string : string -> int -> int -> char -> unit`

`fill_string s start len c` modifies string `s` in place, replacing the characters number `start` to `start + len - 1` by `c`. Raise `Invalid_argument "fill_string"` if `start` and `len` do not designate a valid substring of `s`.

`value blit_string : string -> int -> string -> int -> int -> unit`

`blit_string s1 o1 s2 o2 len` copies `len` characters from string `s1`, starting at character number `o1`, to string `s2`, starting at character number `o2`. It works correctly even if `s1` and `s2` are the same string, and the source and destination chunks overlap. Raise `Invalid_argument "blit_string"` if `o1` and `len` do not designate a valid substring of `s1`, or if `o2` and `len` do not designate a valid substring of `s2`.

`value replace_string : string -> string -> int -> unit`

`replace_string dest src start` copies all characters from the string `src` into the string `dst`, starting at character number `start` in `dst`. Raise `Invalid_argument "replace_string"` if copying would overflow string `dest`.

```
value eq_string : string -> string -> bool
value neq_string : string -> string -> bool
value le_string : string -> string -> bool
value lt_string : string -> string -> bool
value ge_string : string -> string -> bool
value gt_string : string -> string -> bool
```

Comparison functions (lexicographic ordering) between strings.

`value compare_strings : string -> string -> int`

General comparison between strings. `compare_strings s1 s2` returns 0 if `s1` and `s2` are equal, or else -2 if `s1` is a prefix of `s2`, or 2 if `s2` is a prefix of `s1`, or else -1 if `s1` is lexicographically before `s2`, or 1 if `s2` is lexicographically before `s1`.

`value string_for_read : string -> string`

Return a copy of the argument, with special characters represented by escape sequences, following the lexical conventions of Caml Light.

13.17 `vect`: operations on vectors

`value vect_length : 'a vect -> int`

Return the length (number of elements) of the given vector.

`value vect_item : 'a vect -> int -> 'a`

`vect_item v n` returns the element number `n` of vector `v`. The first element has number 0. The last element has number `vect_length v - 1`. Raise `Invalid_argument "vect_item"` if `n` is outside the range 0 to `(vect_length v - 1)`. You can also write `v.(n)` instead of `vect_item v n`.

`value vect_assign : 'a vect -> int -> 'a -> unit`

`vect_assign v n x` modifies vector `v` in place, replacing element number `n` with `x`. Raise `Invalid_argument "vect_assign"` if `n` is outside the range 0 to `vect_length v - 1`. You can also write `v.(n) <- x` instead of `vect_assign v n x`.

`value make_vect : int -> 'a -> 'a vect`

`make_vect n x` returns a fresh vector of length `n`, initialized with `x`. All the elements of this new vector are initially physically equal to `x` (see module `eq`). Consequently, if `x` is mutable, it is shared among all elements of the vector, and modifying `x` through one of the vector entries will modify all other entries at the same time.

`value make_matrix : int -> int -> 'a -> 'a vect vect`

`make_matrix dimx dimy e` returns a two-dimensional array (a vector of vectors) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`. The element `(x,y)` of a matrix `m` is accessed with the notation `m.(x).(y)`.

`value concat_vect : 'a vect -> 'a vect -> 'a vect`

`concat_vect v1 v2` returns a fresh vector containing the concatenation of vectors `v1` and `v2`.

`value sub_vect : 'a vect -> int -> int -> 'a vect`

`sub_vect v start len` returns a fresh vector of length `len`, containing the elements number `start` to `start + len - 1` of vector `v`. Raise `Invalid_argument "sub_vect"` if `start` and `len` do not designate a valid subvector of `v`; that is, if `start < 0`, or `len < 0`, or `start + len > vect_length v`.

`value copy_vect : 'a vect -> 'a vect`

`copy_vect v` returns a copy of `v`, that is, a fresh vector containing the same elements as `v`.

`value fill_vect : 'a vect -> int -> int -> 'a -> unit`

`fill_vect v ofs len x` modifies the vector `v` in place, storing `x` in elements number `ofs` to `ofs + len - 1`. Raise `Invalid_argument "fill_vect"` if `ofs` and `len` do not designate a valid subvector of `v`.

`value blit_vect : 'a vect -> int -> 'a vect -> int -> int -> unit`

`blit_vect v1 o1 v2 o2 len` copies `len` elements from vector `v1`, starting at element number `o1`, to vector `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same vector, and the source and destination chunks overlap. Raise `Invalid_argument "blit_vect"` if `o1` and `len` do not designate a valid subvector of `v1`, or if `o2` and `len` do not designate a valid subvector of `v2`.

`value list_of_vect : 'a vect -> 'a list`

`list_of_vect v` returns the list of all the elements of `v`, that is:
`[v.(0); v.(1); ...; v.(vect_length v - 1)]`.

`value vect_of_list : 'a list -> 'a vect`

`vect_of_list l` returns a fresh vector containing the elements of `l`.

`value map_vect : ('a -> 'b) -> 'a vect -> 'b vect`

`map_vect f v` applies function `f` to all the elements of `v`, and builds a vector with the results returned by `f`: `[| f v.(0); f v.(1); ...; f v.(vect_length v - 1) |]`.

`value map_vect_list : ('a -> 'b) -> 'a vect -> 'b list`

> `map_vect_list f v` applies function `f` to all the elements of `v`, and builds a list with the results returned by `f`: `[ f v.(0); f v.(1); ...; f v.(vect_length v - 1) ]`.

`value do_vect : ('a -> 'b) -> 'a vect -> unit`

> `do_vect f v` applies function `f` in turn to all the elements of `v`, discarding all the results: `f v.(0); f v.(1); ...; f v.(vect_length v - 1); ()`.

Chapter 14

The standard library

This chapter describes the functions provided by the Caml Light standard library. Just as the modules from the core library, the modules from the standard library are automatically linked with the user's object code files by the `camlc` command. Hence, the globals defined by these libraries can be used in standalone programs without having to add any `.zo` file on the command line for the linking phase. Similarly, in interactive use, these globals can be used in toplevel phrases without having to load any `.zo` file in memory.

Unlike the modules from the core library, the modules from the standard library are not automatically "opened" when a compilation starts, or when the toplevel system is launched. Hence it is necessary to use qualified identifiers to refer to the functions provided by these modules, or to add `#open` directives.

Conventions

For easy reference, the modules are listed below in alphabetical order of module names. For each module, the declarations from its interface file are printed one by one in typewriter font, followed by a short comment. All modules and the identifiers they export are indexed at the end of this report.

14.1 `arg`: parsing of command line arguments

This module provides a general mechanism for extracting options and arguments from the command line to the program.

Syntax of command lines: A keyword is a character string starting with a `-`. An option is a keyword alone or followed by an argument. There are four types of keywords: Unit, String, Int, and Float. Unit keywords do not take an argument. String, Int, and Float keywords take the following word on the command line as an argument. Arguments not preceded by a keyword are called anonymous arguments.

Examples (`cmd` is assumed to be the command name):

`cmd -flag` (a unit option)

```
cmd -int 1              (an int option with argument 1)
cmd -string foobar      (a string option with argument "foobar")
cmd -float 12.34        (a float option with argument 12.34)
cmd 1 2 3               (three anonymous arguments: "1", "2", and "3")
cmd 1 2 -flag 3 -string bar 4
                        (four anonymous arguments, a unit option, and
                         a string option with argument "bar")
```

```
type spec =
  String of (string -> unit)
| Int of (int -> unit)
| Unit of (unit -> unit)
| Float of (float -> unit)
```

The concrete type describing the behavior associated with a keyword.

```
value parse : (string * spec) list -> (string -> unit) -> unit
```

`parse speclist anonfun` parses the command line, calling the functions in `speclist` whenever appropriate, and `anonfun` on anonymous arguments. The functions are called in the same order as they appear on the command line. The strings in the `(string * spec) list` are keywords and must start with a `-`, else they are ignored. For the user to be able to specify anonymous arguments starting with a `-`, include for example `("--", String anonfun)` in `speclist`.

```
exception Bad of string
```

Functions in `speclist` or `anonfun` can raise `Bad` with an error message to reject invalid arguments.

14.2 `baltree`: basic balanced binary trees

This module implements balanced ordered binary trees. All operations over binary trees are applicative (no side-effects). The `set` and `map` modules are based on this module. This modules gives a more direct access to the internals of the binary tree implementation than the `set` and `map` abstractions, but is more delicate to use and not as safe. For advanced users only.

```
type 'a t = Empty | Node of 'a t * 'a * 'a t * int
```

The type of trees containing elements of type `'a`. `Empty` is the empty tree (containing no elements).

```
type 'a contents = Nothing | Something of 'a
```

Used with the functions `modify` and `split`, to represent the presence or the absence of an element in a tree.

`value add: ('a -> int) -> 'a -> 'a t -> 'a t`

> `add f x t` inserts the element `x` into the tree `t`. `f` is an ordering function: `f y` must return `0` if `x` and `y` are equal (or equivalent), a negative integer if `x` is smaller than `y`, and a positive integer if `x` is greater than `y`. The tree `t` is returned unchanged if it already contains an element equivalent to `x` (that is, an element `y` such that `f y` is `0`). The ordering `f` must be consistent with the orderings used to build `t` with `add`, `remove`, `modify` or `split` operations.

`value contains: ('a -> int) -> 'a t -> bool`

> `contains f t` checks whether `t` contains an element satisfying `f`, that is, an element `x` such that `f x` is `0`. `f` is an ordering function with the same constraints as for `add`. It can be coarser (identify more elements) than the orderings used to build `t`, but must be consistent with them.

`value find: ('a -> int) -> 'a t -> 'a`

> Same as `contains`, except that `find f t` returns the element `x` such that `f x` is `0`, or raises `Not_found` if none has been found.

`value remove: ('a -> int) -> 'a t -> 'a t`

> `remove f t` removes one element `x` of `t` such that `f x` is `0`. `f` is an ordering function with the same constraints as for `add`. `t` is returned unchanged if it does not contain any element satisfying `f`. If several elements of `t` satisfy `f`, only one is removed.

`value modify: ('a -> int) -> ('a contents -> 'a contents) -> 'a t -> 'a t`

> General insertion/modification/deletion function. `modify f g t` searchs `t` for an element `x` satisfying the ordering function `f`. If one is found, `g` is applied to `Something x`; if `g` returns `Nothing`, the element `x` is removed; if `g` returns `Something y`, the element `y` replaces `x` in the tree. (It is assumed that `x` and `y` are equivalent, in particular, that `f y` is `0`.) If the tree does not contain any `x` satisfying `f`, `g` is applied to `Nothing`; if it returns `Nothing`, the tree is returned unchanged; if it returns `Something x`, the element `x` is inserted in the tree. (It is assumed that `f x` is `0`.) The functions `add` and `remove` are special cases of `modify`, slightly more efficient.

`value split: ('a -> int) -> 'a t -> 'a t * 'a contents * 'a t`

> `split f t` returns a triple `(less, elt, greater)` where `less` is a tree containing all elements `x` of `t` such that `f x` is negative, `greater` is a tree containing all elements `x` of `t` such that `f x` is positive, and `elt` is `Something x` if `t` contains an element `x` such that `f x` is `0`, and `Nothing` otherwise.

`value compare: ('a -> 'a -> int) -> 'a t -> 'a t -> int`

> Compare two trees. The first argument `f` is a comparison function over the tree elements: `f e1 e2` is zero if the elements `e1` and `e2` are equal, negative if `e1` is smaller than `e2`, and positive if `e1` is greater than `e2`. `compare f t1 t2` compares the fringes of `t1` and `t2` by lexicographic extension of `f`.

14.3 `filename`: operations on file names

`value current_dir_name : string`

The conventional name for the current directory (e.g. `.` in Unix).

`value concat : string -> string -> string`

`concat dir file` returns a file name that designates file `file` in directory `dir`.

`value is_absolute : string -> bool`

Return `true` if the file name is absolute or starts with an explicit reference to the current directory (`./` or `../` in Unix), and `false` if it is relative to the current directory.

`value check_suffix : string -> string -> bool`

`check_suffix name suff` returns `true` if the filename `name` ends with the suffix `suff`.

`value chop_suffix : string -> string -> string`

`chop_suffix name suff` removes the suffix `suff` from the filename `name`. The behavior is undefined if `name` does not end with the suffix `suff`.

```
value basename : string -> string
value dirname : string -> string
```

Split a file name into directory name / base file name.
`concat (dirname name) (basename name)` returns a file name which is equivalent to `name`. Moreover, after setting the current directory to `dirname name` (with `sys__chdir`), references to `basename name` (which is a relative file name) designate the same file as `name` before the call to `chdir`.

14.4 `format`: pretty printing

This module implements a pretty-printing facility to format text within "pretty-printing boxes". The pretty-printer breaks lines at specified break hints, and indents lines according to the box structure.

The behaviour of pretty-printing commands is unspecified if there is no opened pretty-printing box. Each box opened via one of the `open_` functions below must be closed using `close_box` for proper formatting. Otherwise, some of the material printed in the boxes may not be output, or may be formatted incorrectly.

In case of interactive use, the system closes all opened boxes and flushes all pending text (as with the `print_newline` function) after each phrase. Each phrase is therefore executed in the initial state of the pretty-printer.

Boxes

`value open_vbox : int -> unit`

> `open_vbox d` opens a new pretty-printing box with offset `d`. This box is “vertical”: every break hint inside this box leads to a new line. When a new line is printed in the box, `d` is added to the current indentation.

`value open_hbox : unit -> unit`

> `open_hbox ()` opens a new pretty-printing box. This box is “horizontal”: the line is not split in this box (new lines may still occur inside boxes nested deeper).

`value open_hvbox : int -> unit`

> `open_hovbox d` opens a new pretty-printing box with offset `d`. This box is “horizontal-vertical”: it behaves as an “horizontal” box if it fits on a single line, otherwise it behaves as a “vertical” box. When a new line is printed in the box, `d` is added to the current indentation.

`value open_hovbox : int -> unit`

> `open_hovbox d` opens a new pretty-printing box with offset `d`. This box is “horizontal or vertical”: break hints inside this box may lead to a new line, if there is no more room on the line to print the remainder of the box. When a new line is printed in the box, `d` is added to the current indentation.

`value close_box : unit -> unit`

> Close the most recently opened pretty-printing box.

Formatting functions

`value print_string : string -> unit`

> `print_string str` prints `str` in the current box.

`value print_as : int -> string -> unit`

> `print_as len str` prints `str` in the current box. The pretty-printer formats `str` as if it were of length `len`.

`value print_int : int -> unit`

> Print an integer in the current box.

`value print_float : float -> unit`

> Print a floating point number in the current box.

`value print_char : char -> unit`

> Print a character in the current box.

`value print_bool : bool -> unit`

> Print an boolean in the current box.

Break hints

`value print_break : int * int -> unit`

Insert a break hint in a pretty-printing box. `print_break (nspaces, offset)` indicates that the line may be split (a newline character is printed) at this point, if the contents of the current box does not fit on one line. If the line is split at that point, `offset` is added to the current indentation. If the line is not split, `nspaces` spaces are printed.

`value print_cut : unit -> unit`

`print_cut ()` is equivalent to `print_break (0,0)`. This allows line splitting at the current point, without printing spaces or adding indentation.

`value print_space : unit -> unit`

`print_space ()` is equivalent to `print_break (1,0)`. This either prints one space or splits the line at that point.

`value force_newline : unit -> unit`

Force a newline in the current box.

`value print_flush : unit -> unit`

Flush the pretty printer: all opened boxes are closed, and all pending text is displayed.

`value print_newline : unit -> unit`

Equivalent to `print_flush` followed by a new line.

`value print_if_newline : unit -> unit`

If the preceding line has not been split, the next formatting command is ignored.

Tabulations

`value open_tbox : unit -> unit`

Open a tabulation box.

`value close_tbox : unit -> unit`

Close the most recently opened tabulation box.

`value print_tbreak : int * int -> unit`

Break hint in a tabulation box. `print_tbreak (spaces, offset)` moves the insertion point to the next tabulation (`spaces` being added to this position). Nothing occurs if insertion point is already on a tabulation mark. If there is no next tabulation on the line, then a newline is printed and the insertion point moves to the first tabulation of the box. If a new line is printed, `offset` is added to the current indentation.

`value set_tab : unit -> unit`

Set a tabulation mark at the current insertion point.

`value print_tab : unit -> unit`

`print_tab ()` is equivalent to `print_tbreak (0,0)`.

Margin

`value set_margin : int -> unit`

`set_margin d` sets the value of the right margin to `d` (in characters): this value is used to detect line overflows that leads to split lines. Nothing happens if `d` is not greater than 1.

`value get_margin : unit -> int`

Return the position of the right margin.

Maximum indentation limit

`value set_max_indent : int -> unit`

`set_max_indent d` sets the value of the maximum indentation limit to `d` (in characters): once this limit is reached, boxes are rejected to the left, if they do not fit on the current line. Nothing happens if `d` is not greater than 1.

`value get_max_indent : unit -> int`

Return the value of the maximum indentation limit (in characters).

Formatting depth: maximum number of boxes allowed before ellipsis

`value set_max_boxes : int -> unit`

`set_max_boxes max` sets the maximum number of boxes simultaneously opened. Material inside boxes nested deeper is printed as an ellipsis (more precisely as the text returned by `get_ellipsis_text`). Nothing happens if `max` is not greater than 1.

`value get_max_boxes : unit -> int`

Return the maximum number of boxes allowed before ellipsis.

Ellipsis

`value set_ellipsis_text : string -> unit`

Set the text of the ellipsis printed when too many boxes are opened (a single dot, ., by default).

`value get_ellipsis_text : unit -> string`

Return the text of the ellipsis.

Redirecting formatter output

```
value set_formatter_output_channel : out_channel -> unit
```

Redirect the pretty-printer output to the given channel.

```
value set_formatter_output_functions :
        (string -> int -> int -> unit) -> (unit -> unit) -> unit
```

`set_formatter_output_functions out flush` redirects the pretty-printer output to the functions `out` and `flush`. The `out` function performs the pretty-printer output. It is called with a string `s`, a start position `p`, and a number of characters `n`; it is supposed to output characters `p` to `p+n-1` of `s`. The `flush` function is called whenever the pretty-printer is flushed using `print_flush` or `print_newline`.

```
value get_formatter_output_functions :
        unit -> (string -> int -> int -> unit) * (unit -> unit)
```

Return the current output functions of the pretty-printer.

14.5 gc: memory management control and statistics

```
type stat = {
  minor_words : int;
  promoted_words : int;
  major_words : int;
  minor_collections : int;
  major_collections : int;
  heap_words : int;
  heap_chunks : int;
  live_words : int;
  live_blocks : int;
  free_words : int;
  free_blocks : int;
  largest_words : int;
  fragments : int
}
```

The memory management counters are returned in a `stat` record. All the numbers are computed since the start of the program. The fields of this record are:

`minor_words` Number of words allocated in the minor heap.

`promoted_words` Number of words allocated in the minor heap that survived a minor collection and were moved to the major heap.

`major_words` Number of words allocated in the major heap, including the promoted words.

`minor_collections` Number of minor collections.

`major_collections` Number of major collection cycles, not counting the current cycle.

`heap_words` Total size of the major heap, in words.

`heap_chunks` Number of times the major heap size was increased.

`live_words` Number of words of live data in the major heap, including the header words.

`live_blocks` Number of live objects in the major heap.

`free_words` Number of words in the free list.

`free_blocks` Number of objects in the free list.

`largest_words` Size (in words) of the largest object in the free list.

`fragments` Number of wasted words due to fragmentation. These are 1-words free blocks placed between two live objects. They cannot be inserted in the free list, thus they are not available for allocation.

The total amount of memory allocated by the program is (in words)
`minor_words + major_words - promoted_words`. Multiply by the word size (4 on a 32-bit machine, 8 on a 64-bit machine) to get the number of bytes.

```
type control = {
  mutable minor_heap_size : int;
  mutable major_heap_increment : int;
  mutable space_overhead : int;
  mutable verbose : bool
}
```

The GC parameters are given as a `control` record. The fields are:

`minor_heap_size` The size (in words) of the minor heap. Changing this parameter will trigger a minor collection.

`major_heap_increment` The minimum number of words to add to the major heap when increasing it.

`space_overhead` The major GC speed is computed from this parameter. This is the percentage of heap space that will be "wasted" because the GC does not immediatly collect unreachable objects. The GC will work more (use more CPU time and collect objects more eagerly) if `space_overhead` is smaller. The computation of the GC speed assumes that the amount of live data is constant.

`verbose` This flag controls the GC messages on standard error output.

```
value stat : unit -> stat
```

Return the current values of the memory management counters in a `stat` record.

```
value print_stat : io__out_channel -> unit
```

Print the current values of the memory management counters (in human-readable form) into the channel argument.

```
value get : unit -> control
```

Return the current values of the GC parameters in a `control` record.

```
value set : control -> unit
```

set r changes the GC parameters according to the control record r. The normal usage is:

```
let r = gc__get () in    (* Get the current parameters. *)
  r.verbose <- true;     (* Change some of them. *)
  gc__set r              (* Set the new values. *)
```

```
value minor : unit -> unit
```

Trigger a minor collection.

```
value major : unit -> unit
```

Finish the current major collection cycle.

```
value full_major : unit -> unit
```

Finish the current major collection cycle and perform a complete new cycle. This will collect all currently unreachable objects.

14.6 genlex: a generic lexical analyzer

This module implements a simple "standard" lexical analyzer, presented as a function from character streams to token streams. It implements roughly the lexical conventions of Caml, but is parameterized by the set of keywords of your language.

```
type token =
    Kwd of string
  | Ident of string
  | Int of int
  | Float of float
  | String of string
  | Char of char
```

The type of tokens. The lexical classes are: Int and Float for integer and floating-point numbers; String for string literals, enclosed in double quotes; Char for character literals, enclosed in backquotes; Ident for identifiers (either sequences of letters, digits, underscores and quotes, or sequences of "operator characters" such as +, *, etc); and Kwd for keywords (either identifiers or single "special characters" such as (, }, etc).

```
value make_lexer: string list -> (char stream -> token stream)
```

Construct the lexer function. The first argument is the list of keywords. An identifier s is returned as Kwd s if s belongs to this list, and as Ident s otherwise. A special character s is returned as Kwd s if s belongs to this list, and cause a lexical error (exception Parse_error) otherwise. Blanks and newlines are skipped. Comments delimited by (* and *) are skipped as well, and can be nested.

Example: a lexer suitable for a desk calculator is obtained by

```
let lexer = make_lexer ["+";"-";"*";"/";"let";"="; "("; ")"]
```

The associated parser would be a function from `token stream` to, for instance, `int`, and would have rules such as:

```
let parse_expr = function
      [< 'Int n >] -> n
    | [< 'Kwd "("; parse_expr n; 'Kwd ")" >] -> n
    | [< parse_expr n1; (parse_remainder n1) n2 >] -> n2
and parse_remainder n1 = function
      [< 'Kwd "+"; parse_expr n2 >] -> n1+n2
    | ...
```

14.7 `hashtbl`: hash tables and hash functions

Hash tables are hashed association tables, with in-place modification.

`type ('a, 'b) t`

The type of hash tables from type `'a` to type `'b`.

`value new : int -> ('a,'b) t`

`new n` creates a new, empty hash table, with initial size `n`. The table grows as needed, so `n` is just an initial guess. Better results are said to be achieved when `n` is a prime number.

`value clear : ('a, 'b) t -> unit`

Empty a hash table.

`value add : ('a, 'b) t -> 'a -> 'b -> unit`

`add tbl x y` adds a binding of `x` to `y` in table `tbl`. Previous bindings for `x` are not removed, but simply hidden. That is, after performing `remove tbl x`, the previous binding for `x`, if any, is restored. (This is the semantics of association lists.)

`value find : ('a, 'b) t -> 'a -> 'b`

`find tbl x` returns the current binding of `x` in `tbl`, or raises `Not_found` if no such binding exists.

`value find_all : ('a, 'b) t -> 'a -> 'b list`

`find_all tbl x` returns the list of all data associated with `x` in `tbl`. The current binding is returned first, then the previous bindings, in reverse order of introduction in the table.

`value remove : ('a, 'b) t -> 'a -> unit`

`remove tbl x` removes the current binding of `x` in `tbl`, restoring the previous binding if it exists. It does nothing if `x` is not bound in `tbl`.

`value do_table : ('a -> 'b -> 'c) -> ('a, 'b) t -> unit`

`do_table f tbl` applies `f` to all bindings in table `tbl`, discarding all the results. `f` receives the key as first argument, and the associated value as second argument. Each binding is presented exactly once to `f`. The order in which the bindings are passed to `f` is unpredictable, except that successive bindings for the same key are presented in reverse chronological order (most recent first).

`value do_table_rev : ('a -> 'b -> 'c) -> ('a, 'b) t -> unit`

Same as `do_table`, except that successive bindings for the same key are presented in chronological order (oldest first).

The polymorphic hash primitive

`value hash : 'a -> int`

`hash x` associates a positive integer to any value of any type. It is guaranteed that if `x = y`, then `hash x = hash y`. Moreover, `hash` always terminates, even on cyclic structures.

`value hash_param : int -> int -> 'a -> int`

`hash_param n m x` computes a hash value for `x`, with the same properties as for `hash`. The two extra parameters `n` and `m` give more precise control over hashing. Hashing performs a depth-first, right-to-left traversal of the structure `x`, stopping after `n` meaningful nodes were encountered, or `m` nodes, meaningful or not, were encountered. Meaningful nodes are: integers; floating-point numbers; strings; characters; booleans; and constant constructors. Larger values of `m` and `n` means that more nodes are taken into account to compute the final hash value, and therefore collisions are less likely to happen. However, hashing takes longer. The parameters `m` and `n` govern the tradeoff between accuracy and speed.

14.8 `lexing`: the run-time library for lexers generated by camllex

Lexer buffers

```
type lexbuf =
  { refill_buff : lexbuf -> unit;
    lex_buffer : string;
    mutable lex_abs_pos : int;
    mutable lex_start_pos : int;
    mutable lex_curr_pos : int;
    mutable lex_last_pos : int;
    mutable lex_last_action : lexbuf -> obj }
```

The type of lexer buffers. A lexer buffer is the argument passed to the scanning functions defined by the generated scanners. The lexer buffer holds the current state of the scanner, plus a function to refill the buffer from the input.

```
value create_lexer_channel : in_channel -> lexbuf
```

Create a lexer buffer on the given input channel. `create_lexer_channel inchan` returns a lexer buffer which reads from the input channel `inchan`, at the current reading position.

```
value create_lexer_string : string -> lexbuf
```

Create a lexer buffer which reads from the given string. Reading starts from the first character in the string. An end-of-input condition is generated when the end of the string is reached.

```
value create_lexer : (string -> int -> int) -> lexbuf
```

Create a lexer buffer with the given function as its reading method. When the scanner needs more characters, it will call the given function, giving it a character string `s` and a character count `n`. The function should put `n` characters or less in `s`, starting at character number 0, and return the number of characters provided. A return value of 0 means end of input.

Functions for lexer semantic actions

The following functions can be called from the semantic actions of lexer definitions (the ML code enclosed in braces that computes the value returned by lexing functions). They give access to the character string matched by the regular expression associated with the semantic action. These functions must be applied to the argument `lexbuf`, which, in the code generated by camllex, is bound to the lexer buffer passed to the parsing function.

```
value get_lexeme : lexbuf -> string
```

`get_lexeme lexbuf` returns the string matched by the regular expression.

```
value get_lexeme_char : lexbuf -> int -> char
```

`get_lexeme_char lexbuf i` returns character number `i` in the matched string.

```
value get_lexeme_start : lexbuf -> int
```

`get_lexeme_start lexbuf` returns the position in the input stream of the first character of the matched string. The first character of the stream has position 0.

```
value get_lexeme_end : lexbuf -> int
```

`get_lexeme_end lexbuf` returns the position in the input stream of the character following the last character of the matched string. The first character of the stream has position 0.

14.9 map: association tables over ordered types

This module implements applicative association tables, also known as finite maps or dictionaries, given a total ordering function over the keys. All operations over maps are purely applicative (no side-effects). The implementation uses balanced binary trees, and therefore searching and insertion take time logarithmic in the size of the map.

```
type ('a, 'b) t
```

The type of maps from type `'a` to type `'b`.

```
value empty: ('a -> 'a -> int) -> ('a, 'b) t
```

The empty map. The argument is a total ordering function over the set elements. This is a two-argument function `f` such that `f e1 e2` is zero if the elements `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Examples: a suitable ordering function for type `int` is `prefix -`. You can also use the generic structural comparison function `eq__compare`.

```
value add: 'a -> 'b -> ('a, 'b) t -> ('a, 'b) t
```

`add x y m` returns a map containing the same bindings as `m`, plus a binding of `x` to `y`. Previous bindings for `x` in `m` are not removed, but simply hidden: they reappear after performing a `remove` operation. (This is the semantics of association lists.)

```
value find:'a -> ('a, 'b) t -> 'b
```

`find x m` returns the current binding of `x` in `m`, or raises `Not_found` if no such binding exists.

```
value remove: 'a -> ('a, 'b) t -> ('a, 'b) t
```

`remove x m` returns a map containing the same bindings as `m` except the current binding for `x`. The previous binding for `x` is restored if it exists. `m` is returned unchanged if `x` is not bound in `m`.

```
value iter: ('a -> 'b -> 'c) -> ('a, 'b) t -> unit
```

`iter f m` applies `f` to all bindings in map `m`, discarding the results. `f` receives the key as first argument, and the associated value as second argument. The order in which the bindings are passed to `f` is unspecified. Only current bindings are presented to `f`: bindings hidden by more recent bindings are not passed to `f`.

14.10 parsing: the run-time library for parsers generated by camlyacc

```
value symbol_start : unit -> int
value symbol_end : unit -> int
```

`symbol_start` and `symbol_end` are to be called in the action part of a grammar rule only. They return the position of the string that matches the left-hand side of the rule: `symbol_start()` returns the position of the first character; `symbol_end()` returns the position of the last character, plus one. The first character in a file is at position 0.

```
value rhs_start: int -> int
value rhs_end: int -> int
```

Same as `symbol_start` and `symbol_end` above, but return then position of the string matching the `n`th item on the right-hand side of the rule, where `n` is the integer parameter to `lhs_start` and `lhs_end`. `n` is 1 for the leftmost item.

```
value clear_parser : unit -> unit
```

Empty the parser stack. Call it just after a parsing function has returned, to remove all pointers from the parser stack to structures that were built by semantic actions during parsing. This is optional, but lowers the memory requirements of the programs.

```
exception Parse_error
```

Raised when a parser encounters a syntax error.

14.11 `printexc`: a catch-all exception handler

```
value f: ('a -> 'b) -> 'a -> 'b
```

`printexc__f fn x` applies `fn` to `x` and returns the result. If the evaluation of `fn x` raises any exception, the name of the exception is printed on standard error output, and the programs aborts with exit code 2. Typical use is `printexc__f main ()`, where `main`, with type `unit->unit`, is the entry point of a standalone program, to catch and print stray exceptions. For `printexc__f` to work properly, the program must have been linked with the `-g` option.

14.12 `printf`: formatting printing functions

```
type ('a, 'b, 'c) format
```

The type of format strings. `'a` is the type of the parameters of the string, `'c` is the result type for the `printf`-style function, and `'b` is the type of the first argument given to `%a` and `%t` printing functions.

```
value fprintf: out_channel -> ('a, out_channel, unit) format -> 'a
```

`fprintf outchan format arg1 ... argN` formats the arguments `arg1` to `argN` according to the format string `format`, and outputs the resulting string on the channel `outchan`. The format is a character string which contains two types of objects: plain characters, which are simply copied to the output channel, and conversion specifications, each of which causes conversion and printing of one argument. Conversion specifications consist in the `%` character, followed by optional flags and field widths, followed by one conversion character. The conversion characters and their meanings are:

`d` or `i`: convert an integer argument to signed decimal u: convert an integer argument to unsigned decimal

x: convert an integer argument to unsigned hexadecimal, using lowercase letters.

X: convert an integer argument to unsigned hexadecimal, using uppercase letters.

s: insert a string argument

c: insert a character argument

f: convert a floating-point argument to decimal notation, in the style `dddd.ddd`

e or E: convert a floating-point argument to decimal notation, in the style `d.ddd e+-dd` (mantissa and exponent)

g or G: convert a floating-point argument to decimal notation, in style f or e, E (whichever is more compact)

b: convert a boolean argument to the string `true` or `false`

a: user-defined printer. Takes two arguments and apply the first one to `outchan` (the current output channel) and to the second argument. The first argument must therefore have type `out_channel -> 'b -> unit` and the second `'b`. The output produced by the function is therefore inserted in the output of `fprintf` at the current point.

t: same as `%a`, but takes only one argument (with type `out_channel -> unit`) and apply it to `outchan`.

Refer to the C library `printf` function for the meaning of flags and field width specifiers. If too few arguments are provided, printing stops just before converting the first missing argument.

`value printf: ('a, out_channel, unit) format -> 'a`

Same as `fprintf`, but output on `std_out`.

`value eprintf: ('a, out_channel, unit) format -> 'a`

Same as `fprintf`, but output on `std_err`.

`value fprint: out_channel -> string -> unit`

Print the given string on the given output channel, without any formatting. This is the same function as `output_string` of module `io`.

`value print: string -> unit`

Print the given string on `std_out`, without any formatting. This is the same function as `print_string` of module `io`.

`value eprint: string -> unit`

Print the given string on `std_err`, without any formatting. This is the same function as `prerr_string` of module `io`.

14.13 queue: queues

This module implements queues (FIFOs), with in-place modification.

`type 'a t`

The type of queues containing elements of type `'a`.

`exception Empty`

Raised when `take` is applied to an empty queue.

`value new: unit -> 'a t`

Return a new queue, initially empty.

`value add: 'a -> 'a t -> unit`

`add x q` adds the element `x` at the end of the queue `q`.

`value take: 'a t -> 'a`

`take q` removes and returns the first element in queue `q`, or raises `Empty` if the queue is empty.

`value peek: 'a t -> 'a`

`peek q` returns the first element in queue `q`, without removing it from the queue, or raises `Empty` if the queue is empty.

`value clear : 'a t -> unit`

Discard all elements from a queue.

`value length: 'a t -> int`

Return the number of elements in a queue.

`value iter: ('a -> 'b) -> 'a t -> unit`

`iter f q` applies `f` in turn to all elements of `q`, from the least recently entered to the most recently entered. The queue itself is unchanged.

14.14 `random`: pseudo-random number generator

`value init : int -> unit`

Initialize the generator, using the argument as a seed. The same seed will always yield the same sequence of numbers.

`value full_init : int vect -> unit`

Same as `init` but takes more data as seed. It is not useful to give more than 55 integers.

`value int : int -> int`

`random__int bound` returns a random number between 0 (inclusive) and `bound` (exclusive). `bound` must be positive and smaller than 2^{30}.

`value float : float -> float`

`random__float bound` returns a random number between 0 (inclusive) and `bound` (exclusive).

14.15 `set`: sets over ordered types

This module implements the set data structure, given a total ordering function over the set elements. All operations over sets are purely applicative (no side-effects). The implementation uses balanced binary trees, and is therefore reasonably efficient: insertion and membership take time logarithmic in the size of the set, for instance.

`type 'a t`

The type of sets containing elements of type `'a`.

`value empty: ('a -> 'a -> int) -> 'a t`

The empty set. The argument is a total ordering function over the set elements. This is a two-argument function `f` such that `f e1 e2` is zero if the elements `e1` and `e2` are equal, `f e1 e2` is strictly negative if `e1` is smaller than `e2`, and `f e1 e2` is strictly positive if `e1` is greater than `e2`. Examples: a suitable ordering function for type `int` is `prefix -`. You can also use the generic structural comparison function `eq__compare`.

`value is_empty: 'a t -> bool`

Test whether a set is empty or not.

`value mem: 'a -> 'a t -> bool`

`mem x s` tests whether `x` belongs to the set `s`.

`value add: 'a -> 'a t -> 'a t`

`add x s` returns a set containing all elements of `s`, plus `x`. If `x` was already in `s`, `s` is returned unchanged.

`value remove: 'a -> 'a t -> 'a t`

`remove x s` returns a set containing all elements of `s`, except `x`. If `x` was not in `s`, `s` is returned unchanged.

`value union: 'a t -> 'a t -> 'a t`
`value inter: 'a t -> 'a t -> 'a t`
`value diff: 'a t -> 'a t -> 'a t`

Union, intersection and set difference.

`value equal: 'a t -> 'a t -> bool`

`equal s1 s2` tests whether the sets `s1` and `s2` are equal, that is, contain the same elements.

`value compare: 'a t -> 'a t -> int`

Total ordering between sets. Can be used as the ordering function for doing sets of sets.

`value elements: 'a t -> 'a list`

Return the list of all elements of the given set. The elements appear in the list in some non-specified order.

`value iter: ('a -> 'b) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`, and discards the results. The elements of `s` are presented to `f` in a non-specified order.

`value fold: ('a -> 'b -> 'b) -> 'a t -> 'b -> 'b`

`fold f s a` computes `(f xN ... (f x2 (f x1 a))...)`, where `x1 ... xN` are the elements of `s`. The order in which elements of `s` are presented to `f` is not specified.

`value choose: 'a t -> 'a`

Return one element of the given set, or raise `Not_found` if the set is empty. Which element is chosen is not specified, but equal elements will be chosen for equal sets.

14.16 `sort`: sorting and merging lists

`value sort : ('a -> 'a -> bool) -> 'a list -> 'a list`

Sort a list in increasing order according to an ordering predicate. The predicate should return `true` if its first argument is less than or equal to its second argument.

`value merge : ('a -> 'a -> bool) -> 'a list -> 'a list -> 'a list`

Merge two lists according to the given predicate. Assuming the two argument lists are sorted according to the predicate, `merge` returns a sorted list containing the elements from the two lists. The behavior is undefined if the two argument lists were not sorted.

14.17 stack: stacks

This module implements stacks (LIFOs), with in-place modification.

`type 'a t`

The type of stacks containing elements of type `'a`.

`exception Empty`

Raised when `pop` is applied to an empty stack.

`value new: unit -> 'a t`

Return a new stack, initially empty.

`value push: 'a -> 'a t -> unit`

`push x s` adds the element `x` at the top of stack `s`.

`value pop: 'a t -> 'a`

`pop s` removes and returns the topmost element in stack `s`, or raises `Empty` if the stack is empty.

`value clear : 'a t -> unit`

Discard all elements from a stack.

`value length: 'a t -> int`

Return the number of elements in a stack.

`value iter: ('a -> 'b) -> 'a t -> unit`

`iter f s` applies `f` in turn to all elements of `s`, from the element at the top of the stack to the element at the bottom of the stack. The stack itself is unchanged.

14.18 sys: system interface

This module provides a simple interface to the operating system.

`exception Sys_error of string`

Raised by some functions in the `sys` and `io` modules, when the underlying system calls fail. The argument to `Sys_error` is a string describing the error. The texts of the error messages are implementation-dependent, and should not be relied upon to catch specific system errors.

```
value command_line : string vect
```

The command line arguments given to the process. The first element is the command name used to invoke the program.

```
value interactive: bool
```

True if we're running under the toplevel system. False if we're running as a standalone program.

```
type file_perm == int
value s_irusr : file_perm
value s_iwusr : file_perm
value s_ixusr : file_perm
value s_irgrp : file_perm
value s_iwgrp : file_perm
value s_ixgrp : file_perm
value s_iroth : file_perm
value s_iwoth : file_perm
value s_ixoth : file_perm
value s_isuid : file_perm
value s_isgid : file_perm
value s_irall : file_perm
value s_iwall : file_perm
value s_ixall : file_perm
```

Access permissions for files. `r` is reading permission, `w` is writing permission, `x` is execution permission. `usr` means permissions for the user owning the file, `grp` for the group owning the file, `oth` for others. `isuid` and `isgid` are for set-user-id and set-group-id files, respectively. The remaining are combinations of the permissions above.

```
type open_flag =
    O_RDONLY                        (* open read-only *)
  | O_WRONLY                        (* open write-only *)
  | O_RDWR                          (* open for reading and writing *)
  | O_APPEND                        (* open for appending *)
  | O_CREAT                         (* create the file if nonexistent *)
  | O_TRUNC                         (* truncate the file to 0 if it exists *)
  | O_EXCL                          (* fails if the file exists *)
  | O_BINARY                        (* open in binary mode *)
  | O_TEXT                          (* open in text mode *)
```

The commands for `open`.

```
value exit : int -> 'a
```

Terminate the program and return the given status code to the operating system. In contrast with the function `exit` from module `io`, this `exit` function does not flush the standard output and standard error channels.

`value open : string -> open_flag list -> file_perm -> int`

Open a file. The second argument is the opening mode. The third argument is the permissions to use if the file must be created. The result is a file descriptor opened on the file.

`value close : int -> unit`

Close a file descriptor.

`value remove : string -> unit`

Remove the given file name from the file system.

`value rename : string -> string -> unit`

Rename a file. The first argument is the old name and the second is the new name.

`value getenv : string -> string`

Return the value associated to a variable in the process environment. Raise `Not_found` if the variable is unbound.

`value chdir : string -> unit`

Change the current working directory of the process. Note that there is no easy way of getting the current working directory from the operating system.

`value system_command : string -> int`

Execute the given shell command and return its exit code.

`exception Break`

Exception `Break` is raised on user interrupt if `catch_break` is on.

`value catch_break : bool -> unit`

`catch_break` governs whether user interrupt terminates the program or raises the `Break` exception. Call `catch_break true` to enable raising `Break`, and `catch_break false` to let the system terminate the program on user interrupt.

Chapter 15

The graphics library

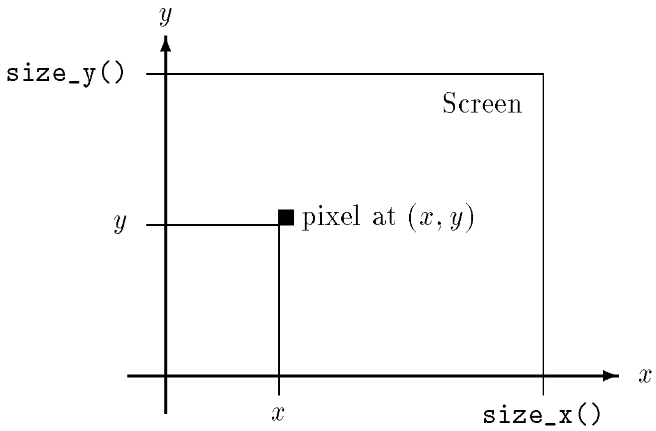
This chapter describes the portable graphics primitives that come standard in the implementation of Caml Light on micro-computers.

Unix: On Unix workstations running the X11 windows system, an implementation of the graphics primitives is available in the directory `contrib/libgraph` in the distribution. See the file `README` in this directory for information on building and using `camlgraph`, a toplevel system that includes the graphics primitives, and linking standalone programs with the library. Drawing takes place in a separate window that is created when `open_graph` is called.

Mac: The graphics primitive are available from the standalone application that runs the toplevel system. They are not available from programs compiled by `camlc` and run under the MPW shell. Drawing takes place in a separate window, that can be made visible with the "Show graphics window" menu entry.

PC: The graphics primitive are available both at toplevel and in separately compiled programs. The `open_graph` primitive switches the whole screen to graphics mode. Under the toplevel system, the user's phrases and the system responses are printed on the graphics screen too, possibly overwriting parts of the drawings, and causing scrolling when the bottom of the screen is reached.

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.



Here are the graphics mode specifications supported by `open_graph` on the various implementations of this library.

Unix: The argument to `open_graph` has the format "*display-name geometry*", where *display-name* is the name of the X-windows display to connect to, and *geometry* is a standard X-windows geometry specification. The two components are separated by a space. Either can be omitted, or both. Examples:

`open_graph "foo:0"`
: connects to the display `foo:0` and creates a window with the default geometry

`open_graph "foo:0 300x100+50-0"`
: connects to the display `foo:0` and creates a window 300 pixels wide by 100 pixels tall, at location $(50, 0)$

`open_graph " 300x100+50-0"`
: connects to the default display and creates a window 300 pixels wide by 100 pixels tall, at location $(50, 0)$

`open_graph ""`
: connects to the default display and creates a window with the default geometry.

Mac: The argument to `open_graph` is ignored.

PC: The 386 PC version supports VGA graphics cards, and a number of SuperVGA cards, provided the correct driver has been installed. (See the installation instructions.) Only 256 color modes are supported. This means a resolution of 320x200 on standard VGA, and from 640x480 to 1024x768 on SuperVGA.

The graphics primitive do not work if Caml Light runs under Windows 3 or any DPMI-compliant memory manager.

The argument to `open_graph` is interpreted as follows:

Argument to `open_graph`	Graphic mode
`"highest"`	highest possible resolution
`"noninterlaced"`	highest possible resolution, non-interlaced mode
`"320x200"`	standard 320x200 VGA mode
"wxh"	at least w by h pixels, if possible
anything else	default resolution, as set in the `GO32` variable (see the installation instructions).

15.1 `graphics`: machine-independent graphics primitives

`exception Graphic_failure of string`

Raised by the functions below when they encounter an error.

Initializations

`value open_graph: string -> unit`

Show the graphics window or switch the screen to graphic mode. The graphics window is cleared. 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.

`value close_graph: unit -> unit`

Delete the graphics window or switch the screen back to text mode.

`value clear_graph : unit -> unit`

Erase the graphics window.

```
value size_x : unit -> int
value size_y : unit -> int
```

Return the size of the graphics window. Coordinates of the screen pixels range over `0 .. size_x()-1` and `0 .. size_y()-1`. Drawings outside of this rectangle are clipped, without causing an error. The origin (0,0) is at the lower left corner.

Colors

`type color == int`

A color is specified by its R, G, B components. Each component is in the range `0..255`. The three components are packed in an `int`: `0xRRGGBB`, where `RR` are the two hexadecimal digits for the red component, `GG` for the green component, `BB` for the blue component.

`value 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`.

`value set_color : color -> unit`

Set the current drawing color.

```
value black : color
value white : color
value red : color
value green : color
value blue : color
value yellow : color
value cyan : color
value magenta : color
```

Some predefined colors.

```
value background: color
value foreground: color
```

Default background and foreground colors (usually, either black foreground on a white background or white foreground on a black background). `clear_graph` fills the screen with the `background` color. The initial drawing color is `foreground`.

Point and line drawing

```
value plot : int -> int -> unit
```

Plot the given point with the current drawing color.

```
value point_color : int -> int -> color
```

Return the color of the given point.

```
value moveto : int -> int -> unit
```

Position the current point.

```
value current_point : unit -> int * int
```

Return the position of the current point.

```
value 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.

```
value draw_arc : int -> int -> int -> int -> int -> int -> unit
```

`draw_arc x y rx ry a1 a2` draws an elliptical arc with center `x,y`, horizontal radius `rx`, vertical radius `ry`, from angle `a1` to angle `a2` (in degrees). The current point is unchanged.

```
value 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.

```
value 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.

```
value set_line_width : int -> unit
```

Set the width of points and lines drawn with the functions above. Under X Windows, `set_line_width 0` selects a width of 1 pixel and a faster, but less precise drawing algorithm than the one used when `set_line_width 1` is specified.

Text drawing

```
value draw_char : char -> unit
value 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.

```
value set_font : string -> unit
value set_text_size : int -> unit
```

Set the font and character size used for drawing text. The interpretation of the arguments to `set_font` and `set_text_size` is implementation-dependent.

```
value text_size : string -> int * int
```

Return the dimensions of the given text, if it were drawn with the current font and size.

Filling

```
value 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 heigth `h`, with the current color.

```
value fill_poly : (int * int) vect -> unit
```

Fill the given polygon with the current color. The array contains the coordinates of the vertices of the polygon.

```
value fill_arc : int -> int -> int -> int -> int -> int -> unit
```

Fill an elliptical pie slice with the current color. The parameters are the same as for `draw_arc`.

```
value fill_ellipse : int -> int -> int -> int -> unit
```

Fill an ellipse with the current color. The parameters are the same as for `draw_ellipse`.

```
value fill_circle : int -> int -> int -> unit
```

Fill a circle with the current color. The parameters are the same as for `draw_circle`.

Images

`type image`

The abstract type for images, in internal representation. Externally, images are represented as matrices of colors.

`value 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.

`value make_image : color vect vect -> 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.

`value dump_image : image -> color vect vect`

Convert an image to a color matrix.

`value draw_image : image -> int -> int -> unit`

Draw the given image with lower left corner at the given point.

`value get_image : int -> int -> int -> int -> image`

Capture the contents of a rectangle on the screen as an image. The parameters are the same as for `fill_rect`.

`value 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.

`value 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.

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.

`value wait_next_event : event list -> status`

Wait until one of the events specified in the given event list occurs, and return the status of the mouse and keyboard at that time. If `Poll` is given in the event list, return immediately with the current status. If the mouse cursor is outside of the graphics window, the `mouse_x` and `mouse_y` fields of the event are outside the range `0..size_x()-1, 0..size_y()-1`. Keypresses are queued, and dequeued one by one when the `Key_pressed` event is specified.

Mouse and keyboard polling

`value 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`.

`value button_down : unit -> bool`

Return `true` if the mouse button is pressed, `false` otherwise.

`value read_key : unit -> char`

Wait for a key to be pressed, and return the corresponding character. Keypresses are queued.

`value key_pressed : unit -> bool`

Return `true` if a keypress is available; that is, if `read_key` would not block.

Sound

```
value sound : int -> int -> unit
```

`sound freq dur` plays a sound at frequency `freq` (in hertz) for a duration `dur` (in milliseconds). On the Macintosh, the frequency is rounded to the nearest note in the equal-tempered scale.

Chapter 16

The unix library: Unix system calls

The `unix` library (distributed in `contrib/libunix`) makes many Unix system calls and system-related library functions available to Caml Light programs. This chapter describes briefly the functions provided. Refer to sections 2 and 3 of the Unix manual for more details on the behavior of these functions.

Not all functions are provided by all Unix variants. If some functions are not available, they will raise `Invalid_arg` when called.

Programs that use the `unix` library must be linked in "custom runtime" mode, as follows:

`camlc -custom` *other options* `unix.zo` *other files* `-lunix`

For interactive use of the `unix` library, run `camllight camlunix`.

Mac: This library is not available.

PC: This library is not available.

16.1 `unix`: interface to the Unix system

Error report

```
type error =
    ENOERR
  | EPERM                 (* Not owner *)
  | ENOENT                (* No such file or directory *)
  | ESRCH                 (* No such process *)
  | EINTR                 (* Interrupted system call *)
  | EIO                   (* I/O error *)
  | ENXIO                 (* No such device or address *)
  | E2BIG                 (* Arg list too long *)
  | ENOEXEC               (* Exec format error *)
  | EBADF                 (* Bad file number *)
  | ECHILD                (* No children *)
  | EAGAIN                (* No more processes *)
```

```
| ENOMEM             (* Not enough core *)
| EACCES             (* Permission denied *)
| EFAULT             (* Bad address *)
| ENOTBLK            (* Block device required *)
| EBUSY              (* Mount device busy *)
| EEXIST             (* File exists *)
| EXDEV              (* Cross-device link *)
| ENODEV             (* No such device *)
| ENOTDIR            (* Not a directory*)
| EISDIR             (* Is a directory *)
| EINVAL             (* Invalid argument *)
| ENFILE             (* File table overflow *)
| EMFILE             (* Too many open files *)
| ENOTTY             (* Not a typewriter *)
| ETXTBSY            (* Text file busy *)
| EFBIG              (* File too large *)
| ENOSPC             (* No space left on device *)
| ESPIPE             (* Illegal seek *)
| EROFS              (* Read-only file system *)
| EMLINK             (* Too many links *)
| EPIPE              (* Broken pipe *)
| EDOM               (* Argument too large *)
| ERANGE             (* Result too large *)
| EWOULDBLOCK        (* Operation would block *)
| EINPROGRESS        (* Operation now in progress *)
| EALREADY           (* Operation already in progress *)
| ENOTSOCK           (* Socket operation on non-socket *)
| EDESTADDRREQ       (* Destination address required *)
| EMSGSIZE           (* Message too long *)
| EPROTOTYPE         (* Protocol wrong type for socket *)
| ENOPROTOOPT        (* Protocol not available *)
| EPROTONOSUPPORT    (* Protocol not supported *)
| ESOCKTNOSUPPORT    (* Socket type not supported *)
| EOPNOTSUPP         (* Operation not supported on socket *)
| EPFNOSUPPORT       (* Protocol family not supported *)
| EAFNOSUPPORT       (* Address family not supported by protocol family *)
| EADDRINUSE         (* Address already in use *)
| EADDRNOTAVAIL      (* Can't assign requested address *)
| ENETDOWN           (* Network is down *)
| ENETUNREACH        (* Network is unreachable *)
| ENETRESET          (* Network dropped connection on reset *)
| ECONNABORTED       (* Software caused connection abort *)
| ECONNRESET         (* Connection reset by peer *)
| ENOBUFS            (* No buffer space available *)
| EISCONN            (* Socket is already connected *)
```

```
  | ENOTCONN            (* Socket is not connected *)
  | ESHUTDOWN           (* Can't send after socket shutdown *)
  | ETOOMANYREFS        (* Too many references: can't splice *)
  | ETIMEDOUT           (* Connection timed out *)
  | ECONNREFUSED        (* Connection refused *)
  | ELOOP               (* Too many levels of symbolic links *)
  | ENAMETOOLONG        (* File name too long *)
  | EHOSTDOWN           (* Host is down *)
  | EHOSTUNREACH        (* No route to host *)
  | ENOTEMPTY           (* Directory not empty *)
  | EPROCLIM            (* Too many processes *)
  | EUSERS              (* Too many users *)
  | EDQUOT              (* Disc quota exceeded *)
  | ESTALE              (* Stale NFS file handle *)
  | EREMOTE             (* Too many levels of remote in path *)
  | EIDRM               (* Identifier removed *)
  | EDEADLK             (* Deadlock condition. *)
  | ENOLCK              (* No record locks available. *)
  | ENOSYS              (* Function not implemented *)
  | EUNKNOWNERR
```

The type of error codes.

`exception Unix_error of error * string * string`

Raised by the system calls below when an error is encountered. The first component is the error code; the second component is the function name; the third component is the string parameter to the function, if it has one, or the empty string otherwise.

`value error_message : error -> string`

Return a string describing the given error code.

`value handle_unix_error : ('a -> 'b) -> 'a -> 'b`

`handle_unix_error f x` applies `f` to `x` and returns the result. If the exception `Unix_error` is raised, it prints a message describing the error and exits with code 2.

Interface with the parent process

`value environment : unit -> string vect`

Return the process environment, as an array of strings with the format "variable=value". See also `sys__getenv`.

Process handling

```
type process_status =
    WEXITED of int
  | WSIGNALED of int * bool
  | WSTOPPED of int
```

The termination status of a process. `WEXITED` means that the process terminated normally by `exit`; the argument is the return code. `WSIGNALED` means that the process was killed by a signal; the first argument is the signal number, the second argument indicates whether a "core dump" was performed. `WSTOPPED` means that the process was stopped by a signal; the argument is the signal number.

```
type wait_flag =
    WNOHANG
  | WUNTRACED
```

Flags for `waitopt` and `waitpid`. `WNOHANG` means do not block if no child has died yet, but immediately return with a pid equal to 0. `WUNTRACED` means report also the children that receive stop signals.

```
value execv : string -> string vect -> unit
```

`execv prog args` execute the program in file `prog`, with the arguments `args`, and the current process environment.

```
value execve : string -> string vect -> string vect -> unit
```

Same as `execv`, except that the third argument provides the environment to the program executed.

```
value execvp : string -> string vect -> unit
```

Same as `execv`, except that the program is searched in the path.

```
value 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.

```
value wait : unit -> int * process_status
```

Wait until one of the children processes die, and return its pid and termination status.

```
value waitopt : wait_flag list -> int * process_status
```

Same as `wait`, but takes a list of options to avoid blocking, or also report stopped children. The pid returned is 0 if no child has changed status.

```
value waitpid : wait_flag list -> int -> int * process_status
```

Same as `waitopt`, but waits for the process whose pid is given. Negative pid arguments represent process groups.

```
value system : string -> process_status
```

Execute the given command, wait until it terminates, and return its termination status. The string is interpreted by the shell `/bin/sh` and therefore can contain redirections, quotes, variables, etc. The result `WEXITED 127` indicates that the shell couldn't be executed.

```
value getpid : unit -> int
```

Return the pid of the process.

```
value getppid : unit -> int
```

Return the pid of the parent process.

```
value nice : int -> int
```

Change the process priority. The integer argument is added to the "nice" value. (Higher values of the "nice" value mean lower priorities.) Return the new nice value.

Basic file input/output

```
type file_descr
```

The abstract type of file descriptors.

```
value stdin : file_descr
value stdout : file_descr
value stderr : file_descr
```

File descriptors for standard input, standard output and standard error.

```
type open_flag =
    O_RDONLY                        (* Open for reading *)
  | O_WRONLY                        (* Open for writing *)
  | O_RDWR                          (* Open for reading and writing *)
  | O_NDELAY                        (* Open in non-blocking mode *)
  | O_APPEND                        (* Open for append *)
  | O_CREAT                         (* Create if nonexistent *)
  | O_TRUNC                         (* Truncate to 0 length if existing *)
  | O_EXCL                          (* Fail if existing *)
```

The flags to `open`.

```
type file_perm == int
```

The type of file access rights.

```
value open : string -> open_flag list -> file_perm -> file_descr
```

Open the named file with the given flags. Third argument is the permissions to give to the file if it is created. Return a file descriptor on the named file.

`value close : file_descr -> unit`

Close a file descriptor.

`value read : file_descr -> string -> int -> int -> int`

`read fd buff start len` reads `len` characters from descriptor `fd`, storing them in string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually read.

`value write : file_descr -> string -> int -> int -> int`

`write fd buff start len` writes `len` characters to descriptor `fd`, taking them from string `buff`, starting at position `ofs` in string `buff`. Return the number of characters actually written.

Interfacing with the standard input/output library (module io).

`value in_channel_of_descr : file_descr -> in_channel`

Create an input channel reading from the given descriptor.

`value out_channel_of_descr : file_descr -> out_channel`

Create an output channel writing on the given descriptor.

`value descr_of_in_channel : in_channel -> file_descr`

Return the descriptor corresponding to an input channel.

`value descr_of_out_channel : out_channel -> file_descr`

Return the descriptor corresponding to an output channel.

Seeking and truncating

```
type seek_command =
    SEEK_SET
  | SEEK_CUR
  | SEEK_END
```

Positioning modes for `lseek`. `SEEK_SET` indicates positions relative to the beginning of the file, `SEEK_CUR` relative to the current position, `SEEK_END` relative to the end of the file.

`value lseek : file_descr -> int -> seek_command -> int`

Set the current position for a file descriptor

`value truncate : string -> int -> unit`

Truncates the named file to the given size.

`value ftruncate : file_descr -> int -> unit`

Truncates the file corresponding to the given descriptor to the given size.

File statistics

```
type file_kind =
    S_REG                                (* Regular file *)
  | S_DIR                                (* Directory *)
  | S_CHR                                (* Character device *)
  | S_BLK                                (* Block device *)
  | S_LNK                                (* Symbolic link *)
  | S_FIFO                               (* Named pipe *)
  | S_SOCK                               (* Socket *)
type stats =
  { st_dev : int;                        (* Device number *)
    st_ino : int;                        (* Inode number *)
    st_kind : file_kind;                 (* Kind of the file *)
    st_perm : file_perm;                 (* Access rights *)
    st_nlink : int;                      (* Number of links *)
    st_uid : int;                        (* User id of the owner *)
    st_gid : int;                        (* Group id of the owner *)
    st_rdev : int;                       (* Device minor number *)
    st_size : int;                       (* Size in bytes *)
    st_atime : int;                      (* Last access time *)
    st_mtime : int;                      (* Last modification time *)
    st_ctime : int }                     (* Last status change time *)
```

The informations returned by the `stat` calls.

`value stat : string -> stats`

Return the information for the named file.

`value lstat : string -> stats`

Same as `stat`, but in case the file is a symbolic link, return the information for the link itself.

`value fstat : file_descr -> stats`

Return the information for the file associated with the given descriptor.

Operations on file names

`value unlink : string -> unit`

Removes the named file

`value rename : string -> string -> unit`

`rename old new` changes the name of a file from `old` to `new`.

`value link : string -> string -> unit`

`link source dest` creates a hard link named `dest` to the file named `new`.

File permissions and ownership

```
type access_permission =
    R_OK                              (* Read permission *)
  | W_OK                              (* Write permission *)
  | X_OK                              (* Execution permission *)
  | F_OK                              (* File exists *)
```

Flags for the `access` call.

`value chmod : string -> file_perm -> unit`

Change the permissions of the named file.

`value fchmod : file_descr -> file_perm -> unit`

Change the permissions of an opened file.

`value chown : string -> int -> int -> unit`

Change the owner uid and owner gid of the named file.

`value fchown : file_descr -> int -> int -> unit`

Change the owner uid and owner gid of an opened file.

`value umask : int -> int`

Set the process creation mask, and return the previous mask.

`value access : string -> access_permission list -> unit`

Check that the process has the given permissions over the named file. Raise `Unix_error` otherwise.

File descriptor hacking

`value fcntl_int : file_descr -> int -> int -> int`

Interface to `fcntl` in the case where the argument is an integer. The first integer argument is the command code; the second is the integer parameter.

`value fcntl_ptr : file_descr -> int -> string -> int`

Interface to `fcntl` in the case where the argument is a pointer. The integer argument is the command code. A pointer to the string argument is passed as argument to the command.

Directories

`value mkdir : string -> file_perm -> unit`

Create a directory with the given permissions.

`value rmdir : string -> unit`

Remove an empty directory.

`value chdir : string -> unit`

Change the process working directory.

`value getcwd : unit -> string`

Return the name of the current working directory.

`type dir_handle`

The type of descriptors over opened directories.

`value opendir : string -> dir_handle`

Open a descriptor on a directory

`value readdir : dir_handle -> string`

Return the next entry in a directory. Raise `End_of_file` when the end of the directory has been reached.

`value rewinddir : dir_handle -> unit`

Reposition the descriptor to the beginning of the directory

`value closedir : dir_handle -> unit`

Close a directory descriptor.

Pipes and redirections

`value pipe : unit -> file_descr * file_descr`

Create a pipe. The first component of the result is opened for reading, that's the exit to the pipe. The second component is opened for writing, that's the entrace to the pipe.

`value dup : file_descr -> file_descr`

Duplicate a descriptor.

`value dup2 : file_descr -> file_descr -> unit`

`dup2 fd1 fd2` duplicates `fd1` to `fd2`, closing `fd2` if already opened.

```
value open_process_in: string -> in_channel
value open_process_out: string -> out_channel
value open_process: string -> in_channel * out_channel
```

High-level pipe and process management. These functions run the given command in parallel with the program, and return channels connected to the standard input and/or the standard output of the command. The command is interpreted by the shell `/bin/sh` (cf. `system`). Warning: writes on channels are buffered, hence be careful to call `flush` at the right times to ensure correct synchronization.

```
value close_process_in: in_channel -> process_status
value close_process_out: out_channel -> process_status
value close_process: in_channel * out_channel -> process_status
```

Close channels opened by `open_process_in`, `open_process_out` and `open_process`, respectively, wait for the associated command to terminate, and return its termination status.

Symbolic links

```
value symlink : string -> string -> unit
```

`symlink source dest` creates the file `dest` as a symbolic link to the file `source`.

```
value readlink : string -> string
```

Read the contents of a link.

Named pipes

```
value mkfifo : string -> file_perm -> unit
```

Create a named pipe with the given permissions.

Special files

```
value ioctl_int : file_descr -> int -> int -> int
```

Interface to `ioctl` in the case where the argument is an integer. The first integer argument is the command code; the second is the integer parameter.

```
value ioctl_ptr : file_descr -> int -> string -> int
```

Interface to `ioctl` in the case where the argument is a pointer. The integer argument is the command code. A pointer to the string argument is passed as argument to the command.

Polling

```
value select :
  file_descr list -> file_descr list -> file_descr list -> float ->
        file_descr list * file_descr list * file_descr list
```

Wait until some input/output operations become possible on some channels. The three list arguments are, respectively, a set of descriptors to check for reading (first argument), for writing (second argument), or for exceptional conditions (third argument). The fourth argument is the maximal timeout, in seconds; a negative fourth argument means no timeout (unbounded wait). The result is composed of three sets of descriptors: those ready for reading (first component), ready for writing (second component), and over which an exceptional condition is pending (third component).

Locking

```
type lock_command =
    F_ULOCK               (* Unlock a region *)
  | F_LOCK                (* Lock a region, and block if already locked *)
  | F_TLOCK               (* Lock a region, or fail if already locked *)
  | F_TEST                (* Test a region for other process' locks *)
```

Commands for `lockf`.

```
value lockf : file_descr -> lock_command -> int -> unit
```

`lockf fd cmd size` puts a lock on a region of the file opened as `fd`. The region starts at the current read/write position for `fd` (as set by `lseek`), and extends `size` bytes forward if `size` is positive, `size` bytes backwards if `size` is negative, or to the end of the file if `size` is zero.

Signals

```
type signal =
    SIGHUP              (* hangup *)
  | SIGINT              (* interrupt *)
  | SIGQUIT             (* quit *)
  | SIGILL              (* illegal instruction (not reset when caught) *)
  | SIGTRAP             (* trace trap (not reset when caught) *)
  | SIGABRT             (* used by abort *)
  | SIGEMT              (* EMT instruction *)
  | SIGFPE              (* floating point exception *)
  | SIGKILL             (* kill (cannot be caught or ignored) *)
  | SIGBUS              (* bus error *)
  | SIGSEGV             (* segmentation violation *)
  | SIGSYS              (* bad argument to system call *)
  | SIGPIPE             (* write on a pipe with no one to read it *)
```

```
  | SIGALRM             (* alarm clock *)
  | SIGTERM             (* software termination signal from kill *)
  | SIGURG              (* urgent condition on IO channel *)
  | SIGSTOP             (* sendable stop signal not from tty *)
  | SIGTSTP             (* stop signal from tty *)
  | SIGCONT             (* continue a stopped process *)
  | SIGCHLD             (* to parent on child stop or exit *)
  | SIGIO               (* input/output possible signal *)
  | SIGXCPU             (* exceeded CPU time limit *)
  | SIGXFSZ             (* exceeded file size limit *)
  | SIGVTALRM           (* virtual time alarm *)
  | SIGPROF             (* profiling time alarm *)
  | SIGWINCH            (* window changed *)
  | SIGLOST             (* resource lost (eg, record-lock lost) *)
  | SIGUSR1             (* user defined signal 1 *)
  | SIGUSR2             (* user defined signal 2 *)
```

The type of signals.

```
type signal_handler =
    Signal_default                  (* Default behavior for the signal *)
  | Signal_ignore                   (* Ignore the signal *)
  | Signal_handle of (unit -> unit) (* Call the given function
                                       when the signal occurs. *)
```

The behavior on receipt of a signal

```
value kill : int -> signal -> unit
```

Send a signal to the process with the given process id.

```
value signal : signal -> signal_handler -> unit
```

Set the behavior to be taken on receipt of the given signal.

```
value pause : unit -> unit
```

Wait until a non-ignored signal is delivered.

Time functions

```
type process_times =
  { tms_utime : float;        (* User time for the process *)
    tms_stime : float;        (* System time for the process *)
    tms_cutime : float;       (* User time for the children processes *)
    tms_cstime : float }      (* System time for the children processes *)
```

The execution times (CPU times) of a process.

```
type tm =
  { tm_sec : int;                       (* Seconds 0..59 *)
    tm_min : int;                       (* Minutes 0..59 *)
    tm_hour : int;                      (* Hours 0..23 *)
    tm_mday : int;                      (* Day of month 1..31 *)
    tm_mon : int;                       (* Month of year 0..11 *)
    tm_year : int;                      (* Year - 1900 *)
    tm_wday : int;                      (* Day of week (Sunday is 0) *)
    tm_yday : int;                      (* Day of year 0..365 *)
    tm_isdst : bool }                   (* Daylight time savings in effect *)
```

The type representing wallclock time and calendar date.

`value time : unit -> int`

Return the current time since 00:00:00 GMT, Jan. 1, 1970, in seconds.

`value gmtime : int -> tm`

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes Greenwich meridian time zone.

`value localtime : int -> tm`

Convert a time in seconds, as returned by `time`, into a date and a time. Assumes the local time zone.

`value alarm : int -> int`

Schedule a `SIGALRM` signals after the given number of seconds.

`value sleep : int -> unit`

Stop execution for the given number of seconds.

`value times : unit -> process_times`

Return the execution times of the process.

`value utimes : string -> int -> int -> 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.

User id, group id

```
value getuid : unit -> int
```

Return the user id of the user executing the process.

```
value geteuid : unit -> int
```

Return the effective user id under which the process runs.

```
value setuid : int -> unit
```

Set the real user id and effective user id for the process.

```
value getgid : unit -> int
```

Return the group id of the user executing the process.

```
value getegid : unit -> int
```

Return the effective group id under which the process runs.

```
value setgid : int -> unit
```

Set the real group id and effective group id for the process.

```
value getgroups : unit -> int vect
```

Return the list of groups to which the user executing the process belongs.

```
type passwd_entry =
  { pw_name : string;
    pw_passwd : string;
    pw_uid : int;
    pw_gid : int;
    pw_gecos : string;
    pw_dir : string;
    pw_shell : string }
```

Structure of entries in the `passwd` database.

```
type group_entry =
  { gr_name : string;
    gr_passwd : string;
    gr_gid : int;
    gr_mem : string vect }
```

Structure of entries in the `groups` database.

```
value getlogin : unit -> string
```

Return the login name of the user executing the process.

```
value getpwnam : string -> passwd_entry
```

Find an entry in `passwd` with the given name, or raise `Not_found`.

```
value getgrnam : string -> group_entry
```

Find an entry in `group` with the given name, or raise `Not_found`.

```
value getpwuid : int -> passwd_entry
```

Find an entry in `passwd` with the given user id, or raise `Not_found`.

```
value getgrgid : int -> group_entry
```

Find an entry in `group` with the given group id, or raise `Not_found`.

Internet addresses

```
type inet_addr
```

The abstract type of Internet addresses.

```
value inet_addr_of_string : string -> inet_addr
value string_of_inet_addr : inet_addr -> string
```

Conversions between string with the format `XXX.YYY.ZZZ.TTT` and Internet addresses. `inet_addr_of_string` raises `Failure` when given a string that does not match this format.

Sockets

```
type socket_domain =
    PF_UNIX                                 (* Unix domain *)
  | PF_INET                                 (* Internet domain *)
```

The type of socket domains.

```
type socket_type =
    SOCK_STREAM                             (* Stream socket *)
  | SOCK_DGRAM                              (* Datagram socket *)
  | SOCK_RAW                                (* Raw socket *)
  | SOCK_SEQPACKET                          (* Sequenced packets socket *)
```

The type of socket kinds, specifying the semantics of communications.

```
type sockaddr =
    ADDR_UNIX of string
  | ADDR_INET of inet_addr * int
```

The type of socket addresses. `ADDR_UNIX name` is a socket address in the Unix domain; `name` is a file name in the file system. `ADDR_INET(addr,port)` is a socket address in the Internet domain; `addr` is the Internet address of the machine, and `port` is the port number.

```
type shutdown_command =
    SHUTDOWN_RECEIVE                    (* Close for receiving *)
  | SHUTDOWN_SEND                       (* Close for sending *)
  | SHUTDOWN_ALL                        (* Close both *)
```

The type of commands for `shutdown`.

```
type msg_flag =
    MSG_OOB
  | MSG_DONTROUTE
  | MSG_PEEK
```

The flags for `recv`, `recvfrom`, `send` and `sendto`.

```
value socket : socket_domain -> socket_type -> int -> file_descr
```

Create a new socket in the given domain, and with the given kind. The third argument is the protocol type; 0 selects the default protocol for that kind of sockets.

```
value socketpair :
        socket_domain -> socket_type -> int -> file_descr * file_descr
```

Create a pair of unnamed sockets, connected together.

```
value accept : file_descr -> file_descr * sockaddr
```

Accept connections on the given socket. The returned descriptor is a socket connected to the client; the returned address is the address of the connecting client.

```
value bind : file_descr -> sockaddr -> unit
```

Bind a socket to an address.

```
value connect : file_descr -> sockaddr -> unit
```

Connect a socket to an address.

```
value listen : file_descr -> int -> unit
```

Set up a socket for receiving connection requests. The integer argument is the maximal number of pending requests.

```
value 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).

```
value getsockname : file_descr -> sockaddr
```

Return the address of the given socket.

```
value getpeername : file_descr -> sockaddr
```

Return the address of the host connected to the given socket.

```
value recv : file_descr -> string -> int -> int -> msg_flag list -> int
value recvfrom :
        file_descr -> string -> int -> int -> msg_flag list -> int * sockaddr
```

Receive data from an unconnected socket.

```
value send : file_descr -> string -> int -> int -> msg_flag list -> int
value sendto :
        file_descr -> string -> int -> int -> msg_flag list -> sockaddr -> int
```

Send data over an unconnected socket.

High-level network connection functions

```
value open_connection : sockaddr -> in_channel * out_channel
```

Connect to a server at the given address. Return a pair of buffered channels connected to the server. Remember to call `flush` on the output channel at the right times to ensure correct synchronization.

```
value shutdown_connection : in_channel -> unit
```

"Shut down" a connection established with `open_connection`; that is, transmit an end-of-file condition to the server reading on the other side of the connection.

```
value establish_server : (in_channel -> out_channel -> 'a) -> sockaddr -> unit
```

Establish a server on the given address. The function given as first argument is called for each connection with two buffered channels connected to the client. A new process is created for each connection. The function `establish_server` never returns normally.

Host and protocol databases

```
type host_entry =
  { h_name : string;
    h_aliases : string vect;
    h_addrtype : socket_domain;
    h_addr_list : inet_addr vect }
```

Structure of entries in the `hosts` database.

```
type protocol_entry =
  { p_name : string;
    p_aliases : string vect;
    p_proto : int }
```

Structure of entries in the `protocols` database.

```
type service_entry =
  { s_name : string;
    s_aliases : string vect;
    s_port : int;
    s_proto : string }
```

Structure of entries in the `services` database.

```
value gethostname : unit -> string
```

Return the name of the local host.

```
value gethostbyname : string -> host_entry
```

Find an entry in `hosts` with the given name, or raise `Not_found`.

```
value gethostbyaddr : inet_addr -> host_entry
```

Find an entry in `hosts` with the given address, or raise `Not_found`.

```
value getprotobyname : string -> protocol_entry
```

Find an entry in `protocols` with the given name, or raise `Not_found`.

```
value getprotobynumber : int -> protocol_entry
```

Find an entry in `protocols` with the given protocol number, or raise `Not_found`.

```
value getservbyname : string -> string -> service_entry
```

Find an entry in `services` with the given name, or raise `Not_found`.

```
value getservbyport : int -> string -> service_entry
```

Find an entry in `services` with the given service number, or raise `Not_found`.

Terminal interface

The following functions implement the POSIX standard terminal interface. They provide control over asynchronous communication ports and pseudo-terminals. Refer to the `termios` man page for a complete description.

```
type terminal_io = {
```

Input modes:

```
mutable c_ignbrk: bool;  (* Ignore the break condition. *)
mutable c_brkint: bool;  (* Signal interrupt on break condition. *)
mutable c_ignpar: bool;  (* Ignore characters with parity errors. *)
mutable c_parmrk: bool;  (* Mark parity errors. *)
mutable c_inpck: bool;   (* Enable parity check on input. *)
mutable c_istrip: bool;  (* Strip 8th bit on input characters. *)
mutable c_inlcr: bool;   (* Map NL to CR on input. *)
mutable c_igncr: bool;   (* Ignore CR on input. *)
mutable c_icrnl: bool;   (* Map CR to NL on input. *)
mutable c_ixon: bool;    (* Recognize XON/XOFF characters on input. *)
mutable c_ixoff: bool;   (* Emit XON/XOFF chars to control input flow. *)
```

Output modes:

```
mutable c_opost: bool;   (* Enable output processing. *)
mutable c_olcuc: bool;   (* Map lowercase to uppercase on output. *)
mutable c_onlcr: bool;   (* Map NL to CR/NL on output. *)
mutable c_ocrnl: bool;   (* Map CR to NL on output. *)
mutable c_onocr: bool;   (* No CR output at column 0. *)
mutable c_onlret: bool;  (* NL is assumed to perform as CR. *)
mutable c_ofill: bool;   (* Use fill characters instead of delays. *)
mutable c_ofdel: bool;   (* Fill character is DEL instead of NULL. *)
mutable c_nldly: int;    (* Newline delay type (0-1). *)
mutable c_crdly: int;    (* Carriage return delay type (0-3). *)
mutable c_tabdly: int;   (* Horizontal tab delay type (0-3). *)
mutable c_bsdly: int;    (* Backspace delay type (0-1). *)
mutable c_vtdly: int;    (* Vertical tab delay type (0-1). *)
mutable c_ffdly: int;    (* Form feed delay type (0-1). *)
```

Control modes:

```
mutable c_obaud: int;    (* Output baud rate (0 means close connection).*)
mutable c_ibaud: int;    (* Input baud rate. *)
mutable c_csize: int;    (* Number of bits per character (5-8). *)
mutable c_cstopb: int;   (* Number of stop bits (1-2). *)
mutable c_cread: bool;   (* Reception is enabled. *)
mutable c_parenb: bool;  (* Enable parity generation and detection. *)
mutable c_parodd: bool;  (* Specify odd parity instead of even. *)
mutable c_hupcl: bool;   (* Hang up on last close. *)
mutable c_clocal: bool;  (* Ignore modem status lines. *)
```

Local modes:

```
mutable c_isig: bool;    (* Generate signal on INTR, QUIT, SUSP. *)
mutable c_icanon: bool;  (* Enable canonical processing
                            (line buffering and editing) *)
mutable c_noflsh: bool;  (* Disable flush after INTR, QUIT, SUSP. *)
mutable c_echo: bool;    (* Echo input characters. *)
```

```
    mutable c_echoe: bool;   (* Echo ERASE (to erase previous character). *)
    mutable c_echok: bool;   (* Echo KILL (to erase the current line). *)
    mutable c_echonl: bool;  (* Echo NL even if c_echo is not set. *)
```

Control characters:

```
    mutable c_vintr: char;   (* Interrupt character (usually ctrl-C). *)
    mutable c_vquit: char;   (* Quit character (usually ctrl-\). *)
    mutable c_verase: char;  (* Erase character (usually DEL or ctrl-H). *)
    mutable c_vkill: char;   (* Kill line character (usually ctrl-U). *)
    mutable c_veof: char;    (* End-of-file character (usually ctrl-D). *)
    mutable c_veol: char;    (* Alternate end-of-line char. (usually none). *)
    mutable c_vmin: int;     (* Minimum number of characters to read
                                before the read request is satisfied. *)
    mutable c_vtime: int;    (* Maximum read wait (in 0.1s units). *)
    mutable c_vstart: char;  (* Start character (usually ctrl-Q). *)
    mutable c_vstop: char    (* Stop character (usually ctrl-S). *)
  }
value tcgetattr: file_descr -> terminal_io
```

Return the status of the terminal referred to by the given file descriptor.

```
type setattr_when = TCSANOW | TCSADRAIN | TCSAFLUSH
value 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.

```
value 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).

```
value tcdrain: file_descr -> unit
```

Waits until all output written on the given file descriptor has been transmitted.

```
type flush_queue = TCIFLUSH | TCOFLUSH | TCIOFLUSH
value tcflush: file_descr -> flush_queue -> unit
```

Discard data written on the given file descriptor but not yet transmitted, or data received but not yet read, depending on the second argument: `TCIFLUSH` flushes data received but not read, `TCOFLUSH` flushes data written but not transmitted, and `TCIOFLUSH` flushes both.

```
type flow_action = TCOOFF | TCOON | TCIOFF | TCION
value 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.

Chapter 17

The num library: arbitrary-precision rational arithmetic

The `num` library (distributed in `contrib/libnum`) implements exact-precision rational arithmetic. It is built upon the state-of-the-art BigNum arbitrary-precision integer arithmetic package, and therefore achieves very high performance.

The functions provided in this library are fully documented in *The CAML Numbers Reference Manual* by Valérie Ménissier-Morain, technical report 141, INRIA, july 1992 (available by anonymous FTP from `ftp.inria.fr`, directory `INRIA/publications/RT`, file `RT-0141.ps.Z`). A summary of the functions is given below.

Programs that use the `num` library must be linked in "custom runtime" mode, as follows:

`camlc -custom` *other options* `nums.zo` *other files* `-lnums`

For interactive use of the `num` library, run `camllight camlnum`.

Mac: This library is not available.

PC: This library is not available.

17.1 `num`: operations on numbers

Numbers (type `num`) are arbitrary-precision rational numbers, plus the special elements `1/0` (infinity) and `0/0` (undefined).

```
type num = Int of int | Big_int of big_int | Ratio of ratio
```

The type of numbers.

Arithmetic operations

```
value prefix +/ : num -> num -> num
value add_num : num -> num -> num
```

Addition

```
value minus_num : num -> num
```

Unary negation.

```
value prefix -/ : num -> num -> num
value sub_num : num -> num -> num
```

Subtraction

```
value prefix */ : num -> num -> num
value mult_num : num -> num -> num
```

Multiplication

```
value square_num : num -> num
```

Squaring

```
value prefix // : num -> num -> num
value div_num : num -> num -> num
```

Division

```
value quo_num : num -> num -> num
value mod_num : num -> num -> num
```

Euclidean division: quotient and remainder

```
value prefix **/ : num -> num -> num
value power_num : num -> num -> num
```

Exponentiation

```
value is_integer_num : num -> bool
```

Test if a number is an integer

```
value integer_num : num -> num
value floor_num : num -> num
value round_num : num -> num
value ceiling_num : num -> num
```

Approximate a number by an integer. `floor_num n` returns the largest integer smaller or equal to `n`. `ceiling_num n` returns the smallest integer bigger or equal to `n`. `integer_num n` returns the integer closest to `n`. In case of ties, rounds towards zero. `round_num n` returns the integer closest to `n`. In case of ties, rounds off zero.

```
value sign_num : num -> int
```

Return `-1`, `0` or `1` according to the sign of the argument.

```
value prefix =/ : num -> num -> bool
value prefix </ : num -> num -> bool
value prefix >/ : num -> num -> bool
value prefix <=/ : num -> num -> bool
value prefix >=/ : num -> num -> bool
value prefix <>/ : num -> num -> bool
value eq_num : num -> num -> bool
value lt_num : num -> num -> bool
value le_num : num -> num -> bool
value gt_num : num -> num -> bool
value ge_num : num -> num -> bool
```

Usual comparisons between numbers

```
value compare_num : num -> num -> int
```

Return `-1`, `0` or `1` if the first argument is less than, equal to, or greater than the second argument.

```
value max_num : num -> num -> num
value min_num : num -> num -> num
```

Return the greater (resp. the smaller) of the two arguments.

```
value abs_num : num -> num
```

Absolute value.

```
value succ_num: num -> num
```

`succ n` is `n+1`

```
value pred_num: num -> num
```

`pred n` is `n-1`

```
value incr_num: num ref -> unit
```

`incr r` is `r:=!r+1`, where `r` is a reference to a number.

```
value decr_num: num ref -> unit
```

`decr r` is `r:=!r-1`, where `r` is a reference to a number.

Coercions with strings

```
value string_of_num : num -> string
```

Convert a number to a string, using fractional notation.

```
value approx_num_fix : int -> num -> string
value approx_num_exp : int -> num -> string
```

Approximate a number by a decimal. The first argument is the required precision. The second argument is the number to approximate. `approx_fix` uses decimal notation; the first argument is the number of digits after the decimal point. `approx_exp` uses scientific (exponential) notation; the first argument is the number of digits in the mantissa.

```
value num_of_string : string -> num
```

Convert a string to a number.

Coercions between numerical types

```
value int_of_num : num -> int
value num_of_int : int -> num
value nat_of_num : num -> nat
value num_of_nat : nat -> num
value num_of_big_int : big_int -> num
value big_int_of_num : num -> big_int
value ratio_of_num : num -> ratio
value num_of_ratio : ratio -> num
value float_of_num : num -> float
```

17.2 `arith_status`: flags that control rational arithmetic

```
value arith_status: unit -> unit
```

Print the current status of the arithmetic flags.

```
value get_error_when_null_denominator : unit -> bool
value set_error_when_null_denominator : bool -> unit
```

Get or set the flag `null_denominator`. When on, attempting to create a rational with a null denominator raises an exception. When off, rationals with null denominators are accepted. Initially: on.

```
value get_normalize_ratio : unit -> bool
value set_normalize_ratio : bool -> unit
```

Get or set the flag `normalize_ratio`. When on, rational numbers are normalized after each operation. When off, rational numbers are not normalized until printed. Initially: off.

```
value get_normalize_ratio_when_printing : unit -> bool
value set_normalize_ratio_when_printing : bool -> unit
```

Get or set the flag `normalize_ratio_when_printing`. When on, rational numbers are normalized before being printed. When off, rational numbers are printed as is, without normalization. Initially: on.

```
value get_approx_printing : unit -> bool
value set_approx_printing : bool -> unit
```

Get or set the flag `approx_printing`. When on, rational numbers are printed as a decimal approximation. When off, rational numbers are printed as a fraction. Initially: off.

```
value get_floating_precision : unit -> int
value set_floating_precision : int -> unit
```

Get or set the parameter `floating_precision`. This parameter is the number of digits displayed when `approx_printing` is on. Initially: 12.

Chapter 18

The str library: regular expressions and string processing

The `str` library (distributed in `contrib/libstr`) provides high-level string processing functions, some based on regular expressions. It is intended to support the kind of file processing that is usually performed with scripting languages such as `awk`, `perl` or `sed`.

Programs that use the `str` library must be linked in "custom runtime" mode, as follows:

`camlc -custom` *other options* `str.zo` *other files* `-lstr`

For interactive use of the `str` library, run `camllight camlstr`.

Mac: This library is not available.

PC: This library is not available.

18.1 `str`: regular expressions and high-level string processing

Regular expressions

`type regexp`

The type of compiled regular expressions.

`value regexp: string -> regexp`

Compile a regular expression. The syntax for regular expressions is the same as in Gnu Emacs. The special characters are `$^.*+?[]\`. The following constructs are recognized:

`.`	matches any character except newline
`*`	(postfix) matches the previous expression zero, one or several times
`+`	(postfix) matches the previous expression one or several times
`?`	(postfix) matches the previous expression once or not at all
`[..]`	character set; ranges are denoted with `-`, as in `a-z`; an initial `^`, as in `^0-9`, complements the set

`^`	matches at beginning of line
`$`	matches at end of line
`|`	(infix) alternative between two expressions
`\(..\)`	grouping and naming of the enclosed expression
`\1`	the text matched by the first `\(...\)` expression (`\2` for the second expression, etc)
`\b`	matches word boundaries
`\`	quotes special characters.

`value 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.

String matching and searching

`value string_match: regexp -> string -> int -> bool`

`string_match r s start` tests whether the characters in `s` starting at position `start` match the regular expression `r`. The first character of a string has position `0`, as usual.

`value search_forward: regexp -> string -> int -> int`

`search_forward r s start` searchs the string `s` for a substring matching the regular expression `r`. The search starts at position `start` and proceeds towards the end of the string. Return the position of the first character of the matched substring, or raise `Not_found` if no substring matches.

`value search_backward: regexp -> string -> int -> int`

Same as `search_forward`, but the search proceeds towards the beginning of the string.

`value matched_string: string -> string`

`matched_string s` returns the substring of `s` that was matched by the latest `string_match`, `search_forward` or `search_backward`. The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

`value match_beginning: unit -> int`
`value match_end: unit -> int`

`match_beginning()` returns the position of the first character of the substring that was matched by `string_match`, `search_forward` or `search_backward`. `match_end()` returns the position of the character following the last character of the matched substring.

`value matched_group: int -> string -> string`

`matched_group n s` returns the substring of `s` that was matched by the `n`th group `\(...\)` of the regular expression during the latest `string_match`, `search_forward` or `search_backward`. The user must make sure that the parameter `s` is the same string that was passed to the matching or searching function.

```
value group_beginning: int -> int
value group_end: int -> int
```

`group_beginning n` returns the position of the first character of the substring that was matched by the `n`th group of the regular expression. `group_end n` returns the position of the character following the last character of the matched substring.

Replacement

```
value global_replace: regexp -> string -> string -> string
```

`global_replace regexp repl s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by `repl`. The replacement text `repl` can contain `\1`, `\2`, etc; these sequences will be replaced by the text matched by the corresponding group in the regular expression. `\0` stands for the text matched by the whole regular expression.

```
value replace_first: regexp -> string -> string -> string
```

Same as `global_replace`, except that only the first substring matching the regular expression is replaced.

```
value global_substitute: regexp -> (string -> string) -> string -> string
```

`global_substitute regexp subst s` returns a string identical to `s`, except that all substrings of `s` that match `regexp` have been replaced by the result of function `subst`. The function `subst` is called once for each matching substring, and receives `s` (the whole text) as argument.

```
value substitute_first: regexp -> (string -> string) -> string -> string
```

Same as `global_substitute`, except that only the first substring matching the regular expression is replaced.

Splitting

```
value split: regexp -> string -> string list
```

`split r s` splits `s` into substrings, taking as delimiters the substrings that match `r`, and returns the list of substrings. For instance, `split (regexp "[ \t]+") s` splits `s` into blank-separated words.

```
value bounded_split: regexp -> string -> int -> string list
```

Same as `split`, but splits into at most `n` substrings, where `n` is the extra integer parameter.

Joining

`value concat: string list -> string`

Catenate a list of string.

`value join: string -> string list -> string`

Catenate a list of string. The first argument is a separator, which is inserted between the strings.

Extracting substrings

`value 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`).

`value 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`).

`value first_chars: string -> int -> string`

`first_chars s n` returns the first `n` characters of `s`. This is the same function as `string_before`.

`value last_chars: string -> int -> string`

`last_chars s n` returns the last `n` characters of `s`.

Formatting

`value format: ('a, unit, string) printf__format -> 'a`

Same as `fprintf` and `printf` from the `printf` module, except that the result of the formatting is returned as a string instead of being written on a channel. The structure of the format string is described in `printf`.

Part V

Appendix

Chapter 19

Further reading

For the interested reader, we list below some references to books and reports related (sometimes loosely) to Caml Light.

19.1 Programming in ML

The books below are programming courses taught in ML. Their main goal is to teach programming, not to describe ML in full details — though most contain fairly good introductions to the ML language. Some of those books use the Standard ML dialect instead of the Caml dialect, so you will have to keep in mind the differences in syntax and in semantics.

- Pierre Weis and Xavier Leroy. *Le langage Caml.* InterÉditions, 1993.

  The natural companion to this manual, provided you read French. This book is a step-by-step introduction to programming in Caml, and presents many realistic examples of Caml programs.

- Guy Cousineau and Michel Mauny. *Approche fonctionnelle de la programmation.* Ediscience, 1995.

  Another Caml programming course written in French, with many original examples.

- Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

  A good introduction to programming in Standard ML. Develops a theorem prover as a complete example. Contains a presentation of the module system of Standard ML.

- Jeffrey D. Ullman. *Elements of ML programming.* Prentice Hall, 1993.

  Another good introduction to programming in Standard ML. No realistic examples, but a very detailed presentation of the language constructs.

- Ryan Stansifer. *ML primer.* Prentice-Hall, 1992.

  A short, but nice introduction to programming in Standard ML.

- Thérèse Accart Hardin and Véronique Donzeau-Gouge Viguié. *Concepts et outils de la programmation. Du fonctionnel à l'impératif avec Caml et Ada.* InterÉditions, 1992.

A first course in programming, that first introduces the main programming notions in Caml, then shows them underlying Ada. Intended for beginners; slow-paced for the others.

- Rachel Harrison. *Abstract Data Types in Standard ML.* John Wiley & Sons, 1993.

  A presentation of Standard ML from the standpoint of abstract data types. Uses intensively the Standard ML module system.

- Harold Abelson and Gerald Jay Sussman. *Structure and Interpretation of Computer Programs.* The MIT press, 1985. (French translation: *Structure et interprétation des programmes informatiques*, InterÉditions, 1989.)

  An outstanding course on programming, taught in Scheme, the modern dialect of Lisp. Well worth reading, even if you are more interested in ML than in Lisp.

19.2 Descriptions of ML dialects

The books and reports below are descriptions of various programming languages from the ML family. They assume some familiarity with ML.

- Xavier Leroy and Pierre Weis. *Manuel de référence du langage Caml.* InterÉditions, 1993.

  The French edition of the present reference manual and user's manual.

- Robert Harper. *Introduction to Standard ML.* Technical report ECS-LFCS-86-14, University of Edinburgh, 1986.

  An overview of Standard ML, including the module system. Terse, but still readable.

- Robin Milner, Mads Tofte and Robert Harper. *The definition of Standard ML.* The MIT press, 1990.

  A complete formal definition of Standard ML, in the framework of structured operational semantics. This book is probably the most mathematically precise definition of a programming language ever written. It is heavy on formalism and extremely terse, so even readers who are thoroughly familiar with ML will have major difficulties with it.

- Robin Milner and Mads Tofte. *Commentary on Standard ML.* The MIT Press, 1991.

  A commentary on the book above, that attempts to explain the most delicate parts and motivate the design choices. Easier to read than the Definition, but still rather involving.

- Guy Cousineau and Gérard Huet. *The CAML primer.* Technical report 122, INRIA, 1990.

  A short description of the original Caml system, from which Caml Light has evolved. Some familiarity with Lisp is assumed.

- Pierre Weis et al. *The CAML reference manual, version 2.6.1.* Technical report 121, INRIA, 1990.

  The manual for the original Caml system, from which Caml Light has evolved.

- Michael J. Gordon, Arthur J. Milner and Christopher P. Wadsworth. *Edinburgh LCF.* Lecture Notes in Computer Science volume 78, Springer-Verlag, 1979.

  This is the first published description of the ML language, at the time when it was nothing more than the control language for the LCF system, a theorem prover. This book is now obsolete, since the ML language has much evolved since then; but it is still of historical interest.

- Paul Hudak, Simon Peyton-Jones and Philip Wadler. *Report on the programming language Haskell, version 1.1.* Technical report, Yale University, 1991.

  Haskell is a purely functional language with lazy semantics that shares many important points with ML (full functionality, polymorphic typing), but has interesting features of its own (dynamic overloading, also called type classes).

19.3 Implementing functional programming languages

The references below are intended for those who are curious to learn how a language like Caml Light is compiled and implemented.

- Xavier Leroy. *The ZINC experiment: an economical implementation of the ML language.* Technical report 117, INRIA, 1990. (Available by anonymous FTP on `ftp.inria.fr`.)

  A description of the ZINC implementation, the prototype ML implementation that has evolved into Caml Light. Large parts of this report still apply to the current Caml Light system, in particular the description of the execution model and abstract machine. Other parts are now obsolete. Yet this report still gives a complete overview of the implementation techniques used in Caml Light.

- Simon Peyton-Jones. *The implementation of functional programming languages.* Prentice-Hall, 1987. (French translation: *Mise en œuvre des langages fonctionnels de programmation*, Masson, 1990.)

  An excellent description of the implementation of purely functional languages with lazy semantics, using the technique known as graph reduction. The part of the book that deals with the transformation from ML to enriched lambda-calculus directly applies to Caml Light. You will find a good description of how pattern-matching is compiled and how types are inferred. The remainder of the book does not apply directly to Caml Light, since Caml Light is not purely functional (it has side-effects), has strict semantics, and does not use graph reduction at all.

- Andrew W. Appel. *Compiling with continuations.* Cambridge University Press, 1992.

  A complete description of an optimizing compiler for Standard ML, based on an intermediate representation called continuation-passing style. Shows how many advanced program optimizations can be applied to ML. Not directly relevant to the Caml Light system, since Caml Light does not use continuation-passing style at all, and makes little attempts at optimizing programs.

19.4 Applications of ML

The following reports show ML at work in various, sometimes unexpected, areas.

- Emmanuel Chailloux and Guy Cousineau. *The MLgraph primer.* Technical report 92-15, École Normale Supérieure, 1992. (Available by anonymous FTP on `ftp.ens.fr`.)

  Describes a Caml Light library that produces Postscript pictures through high-level drawing functions.

- Xavier Leroy. *Programmation du système Unix en Caml Light.* Technical report 147, INRIA, 1992. (Available by anonymous FTP on `ftp.inria.fr`.)

  A Unix systems programming course, demonstrating the use of the Caml Light library that gives access to Unix system calls.

- John H. Reppy. *Concurrent programming with events — The concurrent ML manual.* Cornell University, 1990. (Available by anonymous FTP on `research.att.com`.)

  Concurrent ML extends Standard ML of New Jersey with concurrent processes that communicate through channels and events.

- Jeannette M. Wing, Manuel Faehndrich, J. Gregory Morrisett and Scottt Nettles. *Extensions to Standard ML to support transactions.* Technical report CMU-CS-92-132, Carnegie-Mellon University, 1992. (Available by anonymous FTP on `reports.adm.cs.cmu.edu`.)

  How to integrate the basic database operations to Standard ML.

- Emden R. Gansner and John H. Reppy. *eXene.* Bell Labs, 1991. (Available by anonymous FTP on `research.att.com`.)

  An interface between Standard ML of New Jersey and the X Windows windowing system.

Index to the library

Index of keywords