The Caml Light system, release 0.5
Documentation and user's manual

Xavier Leroy Michel Mauny

September 11, 1992

The Caml Light system, release 0.5
Documentation and user's manual

Xavier Leroy[1] **Michel Mauny**[2]

Abstract

This report documents the Caml Light system, release 0.5. It contains an introduction to functional programming in Caml Light; the reference manual for the Caml Light language; the user's manual for the Caml Light commands; and the documentation for the standard library.

Le système Caml Light, version 0.5
Documentation et manuel d'utilisation

Xavier Leroy **Michel Mauny**

Résumé

Ce rapport contient la documentation du système Caml Light, version 0.5. On y trouve une introduction à la programmation fonctionnelle en Caml Light; le manuel de référence du langage Caml Light; le manuel d'utilisation des commandes du système Caml Light; et enfin la documentation de la bibliothèque standard.

[1]École Normale Supérieure and INRIA Rocquencourt, projet Formel
[2]INRIA Rocquencourt, projet Formel

Contents

Foreword

This manual documents the release 0.5 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, "Functional programming using Caml Light", is a tutorial introduction to the Caml Light language.
- Part III, "The Caml Light language reference manual", is the reference description of the core language.
- Part IV, "The Caml Light library", describes the modules provided in the standard library.
- Part V, "Extensions to the Caml Light language", describes some extensions to the core language that are provided by the current implementation.
- Part VI, "The Caml Light commands", documents the Caml Light compiler, toplevel system, and programming utilities.
- Part VII, "Appendix", contains a short bibliography, a listing 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.

PC86: This is material specific to the 8086 (and 80286) PC version.

PC386: This is material specific to the 80386 PC version.

PC: This is material specific to the two PC versions.

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 package requires X11 release 4 or 5.

Installation. The Unix version is distributed in source format, as a compressed `tar` file named `cl5unix.tar.Z`. To extract, move to the directory where you want the source files to reside, transfer `cl5unix.tar.Z` to that directory, and execute

```
zcat cl5unix.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 `cl5macbin.sea` from the distribution disk. 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 0.5

#
```

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

```
>       Caml Light version 0.5

#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 versions

Two versions are distributed for the PC. The first one, dubbed "8086 PC" here, runs on all PCs, whatever their processor is (8088, 8086, 80286, 80386 DX or SX, 80486 DX or SX, and so on). The second one, dubbed "80386 PC", runs in 32-bit protected mode, and is therefore faster and takes advantage of memory above the standard 640K, but runs only on PCs with 80386 or 80486

processors (DX or SX, it does not matter). The distribution diskette for the PC contains the binaries for both versions, in the two `zip` archive files `cl5pc86.zip` and `cl5pc386.zip`.

In the following, we assume that the distribution diskette 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.

1.3.1 The 8086 PC version

Requirements. A PC running MS-DOS version 3.1 or later. 640K of RAM. About 800K of free space on the disk. The graphics primitives require a CGA, EGA, or VGA video card.

Installation. Create a directory on the hard disk where Caml Light will reside. In the following, we assume that this directory is named `C:\caml86`. If you choose a different directory, replace `C:\caml86` by the appropriate absolute path name. Then, execute the following commands:

```
cd C:\caml86
A:pkunzip -d A:cl5pc86
A:pkunzip -d A:cl5exple
```

(Be careful not to omit the `-d` option to `pkunzip`.) This extracts all files in the `C:\caml86` directory. Then, edit the `C:\autoexec.bat` file, in order to:

- add `C:\caml86\bin` to the `PATH` variable; that is, transform the line that reads

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

  into

  ```
  SET PATH=C:\dos;...;C:\caml86\bin
  ```

- insert the following line

  ```
  SET CAMLLIB=C:\caml86\lib
  ```

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 8086 PC, version 0.5
  (standard library from C:\caml86\lib)
The Caml Light runtime system, version 0.5
The Caml Light compiler, version 0.5
The Caml Light linker, version 0.5
```

Then, execute:

```
caml
```

The `caml` command should print something like:

```
>       Caml Light version 0.5

#
```

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.

`Out of memory`
: Caml Light barely fits into the ridiculously small memory space provided by MS-DOS. Hence, you must make sure that as much of the standard 640K of RAM as possible are free before running Caml Light. As a rule of thumb, 500K of available RAM is the bare minimum; 600K is still far from sufficient. To free memory, remove as many device drivers (by suppressing lines in the `config.sys` file) and TSR programs (by suppressing lines in the `autoexec.bat` file) as possible, and restart the machine. Or, try to load your drivers and TSRs outside of the standard 640K zone, with the help of memory managers such as QEMM or the tools in MS-DOS 5. For more details, consult your favorite PC magazine, or one of the numerous books on MS-DOS.

1.3.2 The 80386 PC version

Requirements. A PC equipped with a 80386 or 80486 processor, running MS-DOS version 3.3 or later. 2M of RAM. About 1.2M of free space on the disk. The graphics primitives require a VGA or SuperVGA video card.

Installation. Create a directory on the hard disk where Caml Light will reside. In the following, we assume that this directory is named `C:\caml386`. If you choose a different directory, replace `C:\caml386` by the appropriate absolute path name. Then, execute the following commands:

```
cd C:\caml386
A:pkunzip -d A:cl5pc386
A:pkunzip -d A:cl5exple
```

(Be careful not to omit the `-d` option to `pkunzip`.) This extracts all files in the `C:\caml386` 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 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` (the graphics drivers) in directory `C:\caml386\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:\caml386\bin` to the `PATH` variable; that is, transform the line that reads

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

  into

  ```
  SET PATH=C:\dos;...;C:\caml386\bin
  ```

- Insert the following lines

  ```
  SET CAMLLIB=C:\caml386\lib
  SET GO32TMP=C:\tmp
  ```

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

  ```
  SET GO32=driver C:\caml386\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:\caml386\dev\emu387 driver C:\caml386\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:\caml386\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 0.5
  (standard library from C:\caml386\lib)
The Caml Light runtime system, version 0.5
The Caml Light compiler, version 0.5
The Caml Light linker, version 0.5
```

Then, execute:

```
caml
```

The `caml` command should print something like:

```
>       Caml Light version 0.5

#
```

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. You'll have to content yourself with the 8086 PC version.

`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:

  - environments such as Windows 3
  - 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, don't start Windows and remove the guilty device drivers from your `config.sys` file.

On the other hand, the 80386 PC version knows how to cohabit with VCPI-compliant environments and memory managers. These include the QEMM386 and 386MAX memory managers, and the Desqview environment. Also, `emm386.exe` from the MS-DOS 5.0 distribution works fine, provided you don't 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

Functional programming using Caml Light

Chapter 2

Introduction

This part is a tutorial introduction to functional programming, and, more precisely, to the usage of Caml Light. It has been used to teach Caml Light[1] in different universities and is intended for beginners. It contains numerous examples and exercises, and absolute beginners should read it while sitting in front of a Caml Light toplevel loop, testing examples and variations by themselves.

After generalities about functional programming, some features specific to Caml Light are described. ML type synthesis and a simple execution model are presented in a complete example of prototyping a subset of ML.

Part A (chapters 3–7) may be skipped by users familiar with ML. Users with experience in functional programming, but unfamiliar with the ML dialects may skip the very first chapters and start at chapter 7, learning the Caml Light syntax from the examples. Part A starts with some intuition about functions and types and gives an overview of ML and other functional languages (chapter 3). Chapter 4 outlines the interaction with the Caml Light toplevel loop and its basic objects. Basic types and some of their associated primitives are presented in chapter 5. Lists (chapter 6) and user-defined types (chapter 7) are structured data allowing for the representation of complex objects and their easy creation and destructuration.

While concepts presented in part A are common (under one form or another) to many functional languages, part B (chapters 8–12) is dedicated to features specific to Caml Light: mutable data structures (chapter 8), exception handling (chapter 9), input/output (chapter 10) and streams and parsers (chapter 11) show a more imperative side of the language. Standalone programs and separate compilation (chapter 12) allow for modular programming and the creation of standalone applications. Concise examples of Caml Light features are to be found in this part.

Part C (chapters 13–17) is meant for already experienced Caml Light users willing to know more about how the Caml Light compiler synthesizes the types of expression and how compilation and evaluation proceeds. Some knowledge about first-order unification is assumed. The presentation is rather informal, and is sometimes terse (specially in the chapter about type synthesis). We prototype a small and simple functional language (called ASL): we give the complete prototype implementation, from the ASL parser to the symbolic execution of code. Lexing and parsing of ASL programs are presented in chapter 13, providing realistic usages of streams and parsers. Chapter 14 presents an untyped call-by-value semantics of ASL programs through the definition of an ASL interpreter. The encoding of recursion in untyped ASL is presented in chapter 15, showing the

[1] The "Caml Strong" version of these notes is available as an INRIA technical report [22].

expressive power of the language. The type synthesis of functional programs is demonstrated in chapter 16, using destructive unification (on first-order terms representing types) as a central tool. Chapter 17 introduces the Categorical Abstract Machine: a simple execution model for call-by-value functional programs. Although the Caml Light execution model is different from the one presented here, an intuition about the simple compilation of functional languages can be found in this chapter.

Warning: The programs and remarks (especially contained in parts B and C) might not be valid in Caml Light versions different from 0.5.

Part A
Functional programming

Chapter 3

Functional languages

Programming languages are said to be *functional* when the basic way of structuring programs is the notion of *function* and their essential control structure is *function application*. For example, the Lisp language [20], and more precisely its modern successor Scheme [29, 1], has been called functional because it possesses these two properties.

However, we want the programming notion of function to be as close as possible to the usual mathematical notion of function. In mathematics, functions are "first-class" objects: they can be arbitrarily manipulated. For example, they can be composed, and the composition function is itself a function.

In mathematics, one would present the *successor* function in the following way:

$$\begin{array}{rll} \mathit{successor}: & \mathbb{N} & \longrightarrow \mathbb{N} \\ & n & \longmapsto n+1 \end{array}$$

The functional composition could be presented as:

$$\begin{array}{rll} \circ: & (A \Rightarrow B) \times (C \Rightarrow A) & \longrightarrow (C \Rightarrow B) \\ & (f, g) & \longmapsto (x \longmapsto f\ (g\ x)) \end{array}$$

where $(A \Rightarrow B)$ denotes the space of functions from A to B.

We remark here the importance of:

1. the notion of *type*; a mathematical function always possesses a *domain* and a *codomain*. They will correspond to the programming notion of type.

2. lexical binding: when we wrote the mathematical definition of *successor*, we have assumed that the addition function + had been previously defined, mapping a pair of natural numbers to a natural number; the meaning of the *successor* function is defined using the *meaning* of the addition: whatever + denotes in the future, this *successor* function will remain the same.

3. the notion of *functional abstraction*, allowing to express the behavior of $f \circ g$ as $(x \longmapsto f\ (g\ x))$, i.e. the function which, when given some x, returns $f\ (g\ x)$.

ML dialects (cf. below) respect these notions. But they also allow non-functional programming styles, and, in this sense, they are functional but not *purely functional*.

3.1 History of functional languages

Some historical points:

- 1930: Alonzo Church developed the λ-calculus [6] as an attempt to provide a basis for mathematics. The λ-calculus is a formal theory for functionality. The three basic constructs of the λ-calculus are:
  - variable names (e.g. x, y,...);
  - application (MN if M and M are terms);
  - functional abstraction ($\lambda x.M$).

  Terms of the λ-calculus represent functions. The pure λ-calculus has been proved inconsistent as a logical theory. Some *type systems* have been added to it in order to remedy this inconsistency.
- 1958: Mac Carthy invented Lisp [20] whose programs have some similarities with terms of the λ-calculus. Lisp dialects have been recently evolving in order to be closer to modern functional languages (Scheme), but they still do not possess a type system.
- 1965: P. Landin proposed the ISWIM [18] language (for "If You See What I Mean"), which is the precursor of languages of the ML family.
- 1978: J. Backus introduced FP: a language of combinators [3] and a framework in which it is possible to reason about programs. The main particularity of FP programs is that they have no variable names.
- 1978: R. Milner proposes a language called ML [11], intended to be the *metalanguage* of the LCF proof assistant (i.e. the language used to program the search of proofs). This language is inspired by ISWIM (close to λ-calculus) and possesses an original type system.
- 1985: D. Turner proposed the Miranda [34] programming language, which uses Milner's type system but where programs are submitted to *lazy evaluation*.

Currently, the two main families of functional languages are the ML and the Miranda families.

3.2 The ML family

ML languages are based on a sugared[1] version of λ-calculus. Their evaluation regime is *call-by-value* (i.e. the argument is evaluated before being passed to a function), and they use Milner's type system.

The LCF proof system appeared in 1972 at Stanford (Stanford LCF). It has been further developed at Cambridge (Cambridge LCF) where the ML language was added to it.

From 1981 to 1986, a version of ML and its compiler was developed in a collaboration between INRIA and Cambridge by G. Cousineau, G. Huet and L. Paulson.

[1] i.e. with a more user-friendly syntax.

In 1981, L. Cardelli implemented a version of ML whose compiler generated native machine code.

In 1984, a committee of researchers from the universities of Edinburgh and Cambridge, Bell Laboratories and INRIA, headed by R. Milner worked on a new extended language called Standard ML [26]. This core language was completed by a module facility designed by D. MacQueen [21].

Since 1984, the Caml language has been under design in a collaboration between INRIA and LIENS[2]). Its first release appeared in 1987. The main implementors of Caml were Ascánder Suárez, Pierre Weis and Michel Mauny.

In 1989 appeared Standard ML of New-Jersey, developed by Andrew Appel (Princeton University) and David MacQueen (Bell Laboratories).

Caml Light is a smaller, more portable implementation of the core Caml language, developed by Xavier Leroy since 1990.

3.3 The Miranda family

All languages in this family use *lazy evaluation* (i.e. the argument of a function is evaluated if and when the function needs its value—arguments are passed unevaluated to functions). They also use Milner's type system.

Languages belonging to the Miranda family find their origin in the SASL language [32] (1976) developed by D. Turner. SASL and its successors (KRC [33] 1981, Miranda [34] 1985 and Haskell [15] 1990) use *sets of mutually recursive equations* as programs. These equations are written in a *script* (collection of declarations) and the user may evaluate expressions using values defined in this script. LML (Lazy ML) has been developed in Göteborg (Sweeden); its syntax is close to ML's syntax and it uses a fast execution model: the G-machine [16].

[2]Laboratoire d'Informatique de l'Ecole Normale Supérieure, 45 Rue d'Ulm, 75005 Paris

Chapter 4

Basic concepts

We examine in this chapter some fundamental concepts which we will use and study in the following chapters. Some of them are specific to the interface with the Caml language (toplevel, global definitions) while others are applicable to any functional language.

4.1 Toplevel loop

The first contact with Caml is through its interactive aspect[1]. When running Caml on a computer, we enter a *toplevel loop* where Caml waits for input from the user, and then gives a response to what has been entered.

The beginning of a Caml Light session looks like this (assuming `$` is the shell prompt on the host machine):

```
$camllight
>       Caml Light version 0.5

#
```

On the PC version, the command is called `caml`.

The "`#`" character is Caml's prompt. When this character appears at the beginning of a line in an actual Caml Light session, the toplevel loop is waiting for a new toplevel phrase to be entered.

Throughout this document, the phrases starting by the `#` character represent legal input to Caml. Since this document has been pre-processed by Caml Light and these lines have been effectively given as input to a Caml Light process, the reader may reproduce by him/herself the session contained in each chapter (each chapter of the first two parts contains a different session, the third part is a single session). Lines starting with the "`>`" character are Caml Light error messages. Lines starting by another character are either Caml responses or (possibly) illegal input. The input is printed in typewriter font (`like this`), and output is written using slanted typewriter font (*like that*).

Important: Of course, when developing non-trivial programs, it is preferable to edit programs in files and then to include the files in the toplevel, instead of entering the programs directly.

[1] Caml Light implementations also possess a batch compiler usable to compile files and produce standalone applications; see chapters 12 and and 25.

Furthermore, when debugging, it is very useful to *trace* some functions, to see with what arguments they are called, and what result they return. The description of these Caml Light commands is given in section 26.2.

4.2 Evaluation: from expressions to values

Let us enter an arithmetic expression and see what is Caml's response:

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

The expression "`1+2`" has been entered, followed by "`;;`" which represents the end of the current toplevel phrase. When encountering "`;;`", Caml enters the type-checking (more precisely *type synthesis*) phase, and prints the inferred type for the expression. Then, it compiles code for the expression, executes it and, finally, prints the result.

In the previous example, the result of evaluation is printed as "`3`" and the type is "`int`": the type of integers.

The process of evaluation of a Caml expression can be seen as transforming this expression until no further transformation is allowed. These transformations must preserve semantics. For example, if the expression "`1+2`" has the mathematical object 3 as semantics, then the result "`3`" must have the same semantics. The different phases of the Caml evaluation process are:

- parsing (checking the syntactic legality of input);
- type synthesis;
- compiling;
- loading;
- executing;
- printing the result of execution.

Let us consider another example: the application of the successor function to `2+3`. The expression `(function x -> x+1)` should be read as "the function that, given `x`, returns `x+1`". The juxtaposition of this expression to `(2+3)` is *function application*.

```
#(function x -> x+1) (2+3);;
- : int = 6
```

There are several ways to reduce that value before obtaining the result `6`. For example, we could proceed this way:

or that way:

The transformations used by Caml during evaluation cannot be described in this chapter, since they imply knowledge about compilation of Caml programs and machine representation of Caml values. However, since the basic control structure of Caml is function application, it is quite easy to give an idea of the transformations involved in the Caml evaluation process by using the kind of rewriting we used in the last example. The evaluation of the (well-typed) application $e_1(e_2)$, where e_1 and e_2 denote arbitrary expressions, consists in the following steps:

- Evaluate e_2, obtaining its value v.
- Evaluate e_1 until it becomes a functional value. Because of the well-typing hypothesis, e_1 must represent a function from some type t_1 to some t_2, and the type of v is t_1. We will write (`function x ->` e) for the result of the evaluation of e_1. It denotes the mathematical object usually written as:

$$\begin{array}{rl} f: & t_1 \to t_2 \\ & x \mapsto e \text{ (where, of course, } e \text{ may depend on } x) \end{array}$$

  N.B.: We do not evaluate e before we know the value of x.
- Evaluate e where v has been substituted for all occurrences of `x`. We then get the final value of the original expression. The result is of type t_2.

In the previous example, we evaluate:

- `2+3` to `5`;
- `(function x -> x+1)` to itself (it is already a function body);
- `x+1` where `5` is substituted for `x`, i.e. evaluate `5+1`, getting `6` as result.

It should be noticed that Caml uses call-by-value: arguments are evaluated before being passed to functions. The relative evaluation order of the functional expression and of the argument expression is implementation-dependent, and should not be relied upon. The Caml Light implementation evaluates arguments before functions (right-to-left evaluation), as shown above; the original Caml implementation evaluates functions before arguments (left-to-right evaluation).

4.3 Types

Types and their checking/synthesis are crucial to functional programming: they provide an indication about the correctness of programs.

The universe of Caml values is partitioned into *types*. A type represents a collection of values. For example, `int` is the type of integer numbers, and `float` is the type of floating-point numbers. Truth values belong to the `bool` type. Character strings belong to the `string` type. Types are divided into two classes:

- Basic types (`int`, `bool`, `string`, ...).
- Compound types such as functional types. For example, the type of functions from integers to integers is denoted by `int -> int`. The type of functions from boolean values to character strings is written `bool -> string`. The type of pairs whose first component is an integer and whose second component is a boolean value is written `int * bool`.

In fact, any combination of the types above (and even more!) is possible. This could be written as:

```
BasicType ::= int
            | bool
            | string

Type      ::= BasicType
            | Type -> Type
            | Type * Type
```

Once a Caml toplevel phrase has been entered in the computer, the Caml process starts analyzing it. First of all, it performs *syntax analysis*, which consists in checking whether the phrase belongs to the language or not. For example, here is a syntax error[2] (a parenthesis is missing):

```
#(function x -> x+1 (2+3);;
> Toplevel input:
>(function x -> x+1 (2+3);;
>                       ^^
> Syntax error.
```

The carets "`^^^`" underline the location where the error was detected.

The second analysis performed is *type analysis*: the system attempts to assign a type to each subexpression of the phrase, and to synthesize the type of the whole phrase. If type analysis fails (i.e. if the system is unable to assign a sensible type to the phrase), then a type error is reported and Caml waits for another input, rejecting the current phrase. Here is a type error (cannot add a number to a boolean):

```
#(function x -> x+1) (2+true);;
> Toplevel input:
>(function x -> x+1) (2+true);;
```

[2] Actually, lexical analysis takes place before syntax analysis and *lexical errors* may occur, detecting for instance badly formed character constants.

```
>                        ^^^^
> Expression of type bool
> cannot be used with type int
```

The rejection of ill-typed phrases is called *strong typing*. Compilers for weakly typed languages (C, for example) would instead issue a warning message and continue their work at the risk of getting a "`Bus error`" or "`Illegal instruction`" message at run-time.

Once a sensible type has been deduced for the expression, then the evaluation process (compilation, loading and execution) may begin.

Strong typing enforces writing clear and well-structured programs. Moreover, it adds security and increases the speed of program development, since most typos and many conceptual errors are trapped and signaled by the type analysis. Finally, well-typed programs do not need dynamic type tests (the addition function does not need to test whether or not its arguments are numbers), hence static type analysis allows for more efficient machine code.

4.4 Functions

The most important kind of values in functional programming are functional values. Mathematically, a function f of type $A \to B$ is a rule of correspondence which associates with each element of type A a unique member of type B.

If x denotes an element of A, then we will write $f(x)$ for the application of f to x. Parentheses are often useless (they are used only for grouping subexpressions), so we could also write $(f(x))$ as well as $(f((x)))$ or simply $f\ x$. The value of $f\ x$ is the unique element of B associated with x by the rule of correspondence for f.

The notation $f(x)$ is the one normally employed in mathematics to denote functional application. However, we shall be careful not to confuse a function with its application. We say "the function f with formal parameter x", meaning that f has been defined by:

$$f : x \mapsto e$$

or, in Caml, that the body of f is something like `(function x -> ...)`. Functions are values as other values. In particular, functions may be passed as arguments to other functions, and/or returned as result. For example, we could write:

```
#function f -> (function x -> (f(x+1) - 1));;
- : (int -> int) -> int -> int = <fun>
```

That function takes as parameter a function (let us call it `f`) and returns another function which, when given an argument (let us call it `x`), will return the predecessor of the value of the application `f(x+1)`.

The type of that function should be read as: `(int -> int) -> (int -> int)`.

4.5 Definitions

It is important to give names to values. We have already seen some named values: we called them *formal parameters*. In the expression `(function x -> x+1)`, the name `x` is a formal parameter.

Its name is irrelevant: changing it into another one does not change the meaning of the expression. We could have written that function `(function y -> y+1)`.

If now we apply this function to, say, `1+2`, we will evaluate the expression `y+1` where `y` is the value of `1+2`. Naming `y` the value of `1+2` in `y+1` is written as:

```
#let y=1+2 in y+1;;
- : int = 4
```

This expression is a legal Caml phrase, and the `let` construct is indeed widely used in Caml programs.

The `let` construct introduces *local bindings of values to identifiers*. They are *local* because the scope of `y` is restricted to the expression `(y+1)`. The identifier `y` kept its previous binding (if any) after the evaluation of the "`let` ... `in` ..." construct. We can check that `y` has not been globally defined by trying to evaluate it:

```
#y;;
> Toplevel input:
>y;;
>^
> Variable y is unbound.
```

Local bindings using `let` also introduce *sharing* of (possibly time-consuming) evaluations. When evaluating "`let x=`e_1 `in` e_2", e_1 gets evaluated only once. All occurrences of `x` in e_2 access the *value* of e_1 which has been computed once. For example, the computation of:

```
let big = sum_of_prime_factors 35461243
in big+(2+big)-(4*(big+1));;
```

will be less expensive than:

```
  (sum_of_prime_factors 35461243)
+ (2 + (sum_of_prime_factors 35461243))
- (4 * (sum_of_prime_factors 35461243 + 1));;
```

The `let` construct also has a global form for toplevel declarations, as in:

```
#let successor = function x -> x+1;;
successor : int -> int = <fun>

#let square = function x -> x*x;;
square : int -> int = <fun>
```

When a value has been declared at toplevel, it is of course available during the rest of the session.

```
#square(successor 3);;
- : int = 16

#square;;
- : int -> int = <fun>
```

When declaring a functional value, there are some alternative syntaxes available. For example we could have declared the `square` function by:

```
#let square x = x*x;;
square : int -> int = <fun>
```

or (closer to the mathematical notation) by:

```
#let square (x) = x*x;;
square : int -> int = <fun>
```

All these definitions are equivalent.

4.6 Partial applications

A *partial application* is the application of a function to some but not all of its arguments. Consider, for example, the function `f` defined by:

```
#let f x = function y -> 2*x+y;;
f : int -> int -> int = <fun>
```

Now, the expression `f(3)` denotes the function which when given an argument `y` returns the value of `2*3+y`. The application `f(x)` is called a *partial application*, since `f` waits for two successive arguments, and is applied to only one. The value of `f(x)` is still a function.

We may thus define an addition function by:

```
#let addition x = function y -> x+y;;
addition : int -> int -> int = <fun>
```

This can also be written:

```
#let addition x y = x+y;;
addition : int -> int -> int = <fun>
```

We can then define the successor function by:

```
#let successor = addition 1;;
successor : int -> int = <fun>
```

Now, we may use our `successor` function:

```
#successor (successor 1);;
- : int = 3
```

Exercises

4.1 *Give examples of functions with the following types:*

1. `(int -> int) -> int`
2. `int -> (int -> int)`
3. `(int -> int) -> (int -> int)`

4.2 *We have seen that the names of formal parameters are meaningless. It is then possible to rename* `x` *by* `y` *in* `(function x -> x+x)`*. What should we do in order to rename* `x` *in* `y` *in*

```
(function x -> (function y -> x+y))
```

Hint: rename `y` *by* `z` *first. Question: why?*

4.3 *Evaluate the following expressions (rewrite them until no longer possible):*

```
let x=1+2 in ((function y -> y+x) x);;
let x=1+2 in ((function x -> x+x) x);;
let f1 = function f2 -> (function x -> f2 x)
in let g = function x -> x+1
   in f1 g 2;;
```

Chapter 5

Basic types

We examine in this chapter the Caml basic types.

5.1 Numbers

Caml Light provides two numeric types: integers (type `int`) and floating-point numbers (type `float`). Integers are limited to the range $-2^{30} \ldots 2^{30} - 1$. Integer arithmetic is taken modulo 2^{31}; that is, an integer operation that overflows does not raise an error, but the result simply wraps around. Predefined operations (functions) on integers include:

`+`	addition
`-`	subtraction and unary minus
`*`	multiplication
`/`	division
`mod`	modulo

Some examples of expressions:

```
#3 * 4 + 2;;
- : int = 14

#3 * (4 + 2);;
- : int = 18

#3 - 7 - 2;;
- : int = -6
```

There are precedence rules that make `*` bind tighter than `+`, and so on. In doubt, write extra parentheses.

So far, we have not seen the type of these arithmetic operations. They all expect two integer arguments; so, they are functions taking one integer and returning a function from integers to integers. The (functional) value of such infix identifiers may be obtained by taking their *prefix* version.

```
#prefix +;;
- : int -> int -> int = <fun>
```

```
#prefix *;;
- : int -> int -> int = <fun>

#prefix mod;;
- : int -> int -> int = <fun>
```

As shown by their types, the infix operators `+`, `*`, ..., do not work on floating-point values. A separate set of floating-point arithmetic operations is provided:

`+.`	addition
`-.`	subtraction and unary minus
`*.`	multiplication
`/.`	division
`sqrt`	square root
`exp`, `log`	exponentiation and logarithm
`sin`, `cos`,...	usual trigonometric operations

We have two conversion functions to go back and forth between integers and floating-point numbers: `int_of_float` and `float_of_int`.

```
#1 + 2.3;;
> Toplevel input:
>1 + 2.3;;
>    ^^^
> Expression of type float
> cannot be used with type int

#float_of_int 1 +. 2.3;;
- : float = 3.3
```

Let us now give some examples of numerical functions. We start by defining some very simple functions on numbers:

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

#square 2.0;;
- : float = 4

#square (2.0 /. 3.0);;
- : float = 0.444444444444

#let sum_of_squares (x,y) = square(x) +. square(y);;
sum_of_squares : float * float -> float = <fun>

#let half_pi = 3.14159 /. 2.0
#in sum_of_squares(cos(half_pi), sin(half_pi));;
- : float = 1
```

We now develop a classical example: the computation of the root of a function by Newton's method. Newton's method can be stated as follows: if y is an approximation to a root of a function f, then:

$$y - \frac{f(y)}{f'(y)}$$

is a better approximation, where $f'(y)$ is the derivative of f evaluated at y. For example, with $f(x) = x^2 - a$, we have $f'(x) = 2x$, and therefore:

$$y - \frac{f(y)}{f'(y)} = y - \frac{y^2 - a}{2y} = \frac{y + \frac{a}{y}}{2}$$

We can define a function `deriv` for approximating the derivative of a function at a given point by:

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

Provided `dx` is sufficiently small, this gives a reasonable estimate of the derivative of f at x.

The following function returns the absolute value of its floating point number argument. It uses the "`if` ... `then` ... `else`" conditional construct.

```
#let abs x = if x >. 0.0 then x else -. x;;
abs : float -> float = <fun>
```

The main function, given below, uses three local functions. The first one, `until`, is an example of a *recursive* function: it calls itself in its body, and is defined using a `let rec` construct (`rec` shows that the definition is recursive). It takes three arguments: a predicate `p` on floats, a function `change` from floats to floats, and a float `x`. If `p(x)` is false, then `until` is called with last argument `change(x)`, otherwise, `x` is the result of the whole call. We will study recursive functions more closely later. The two other auxiliary functions, `satisfied` and `improve`, take a float as argument and deliver respectively a boolean value and a float. The function `satisfied` asks wether the image of its argument by `f` is smaller than `epsilon` or not, thus testing wether `y` is close enough to a root of `f`. The function `improve` computes the next approximation using the formula given below. The three local functions are defined using a cascade of `let` constructs. The whole function is:

```
#let newton f epsilon =
#  let rec until p change x =
#            if p(x) then x
#            else until p change (change(x)) in
#  let satisfied y = abs(f y) <. epsilon in
#  let improve y = y -. (f(y) /. (deriv f y epsilon))
#in until satisfied improve;;
newton : (float -> float) -> float -> float -> float = <fun>
```

Some examples of equation solving:

```
#let square_root x epsilon =
#           newton (function y -> y*.y -. x) epsilon x
```

```
#and cubic_root x epsilon =
#           newton (function y -> y*.y*.y -. x) epsilon x;;
square_root : float -> float -> float = <fun>
cubic_root : float -> float -> float = <fun>

#square_root 2.0 1e-5;;
- : float = 1.41421569553

#cubic_root 8.0 1e-5;;
- : float = 2.00000000131

#2.0 *. (newton cos 1e-5 1.5);;
- : float = 3.14159265359
```

5.2 Boolean values

The presence of the conditional construct implies the presence of boolean values. The type `bool` is composed of two values `true` and `false`.

```
#true;;
- : bool = true

#false;;
- : bool = false
```

The functions with results of type `bool` are often called *predicates*. Many predicates are predefined in Caml. Here are some of them:

```
#prefix <;;
- : int -> int -> bool = <fun>

#1 < 2;;
- : bool = true

#prefix <.;;
- : float -> float -> bool = <fun>

#3.14159 <. 2.718;;
- : bool = false
```

There exist also `<=`, `>`, `>=`, and similarly `<=.`, `>.`, `>=.`.

5.2.1 Equality

Equality has a special status in functional languages because of functional values. It is easy to test equality of values of base types (integers, floating-point numbers, booleans, ...):

```
#1 = 2;;
- : bool = false

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

But it is impossible, in the general case, to decide the equality of functional values. Hence, equality stops on a run-time error when it encounters functional values.

```
#(fun x -> x) = (fun x -> x);;
Uncaught exception: Invalid_argument "equal: functional value"
```

Since equality may be used on values of any type, what is its type? Equality takes two arguments of the same type (whatever type it is) and returns a boolean value. The type of equality is a *polymorphic type*, i.e. may take several possible forms. Indeed, when testing equality of two numbers, then its type is `int -> int -> bool`, and when testing equality of strings, its type is `string -> string -> bool`.

```
#prefix =;;
- : 'a -> 'a -> bool = <fun>
```

The type of equality uses *type variables*, written `'a`, `'b`, etc. Any type can be substituted to type variables in order to produces specific *instances* of types. For example, substituting `int` for `'a` above produces `int -> int -> bool`, which is the type of the equality function used on integer values.

```
#1=2;;
- : bool = false

#(1,2) = (2,1);;
- : bool = false

#1 = (1,2);;
> Toplevel input:
>1 = (1,2);;
>     ^^^
> Expression of type 'a * 'b
> cannot be used with type int
```

5.2.2 Conditional

Conditional expressions are of the form "`if` e_{test} `then` e_1 `else` e_2", where e_{test} is an expression of type `bool` and e_1, e_2 are expressions possessing the same type. As an example, the logical negation could be written:

```
#let negate a = if a then false else true;;
negate : bool -> bool = <fun>

#negate (1=2);;
- : bool = true
```

5.2.3 Logical operators

The classical logical operators are available in Caml. Disjunction and conjunction are respectively written `or` and `&`:

```
#true or false;;
- : bool = true
```

```
#(1<2) & (2>1);;
- : bool = true
```

The operators `&` and `or` are not functions. They should not be seen as regular functions, since they evaluate their second argument only if it is needed. For example, the `or` operator evaluates its second operand only if the first one evaluates to `false`. The behavior of these operators may be described as follows:

e_1 `or` e_2	is equivalent to	`if` e_1 `then true else` e_2
e_1 `&` e_2	is equivalent to	`if` e_1 `then` e_2 `else false`

Negation is predefined:

```
#not true;;
- : bool = false
```

The `not` identifier receives a special treatment from the parser: the application "`not f x`" is recognized as "`not (f x)`" while "`f g x`" is identical to "`(f g) x`". This special treatment explains why the functional value associated to `not` can be obtained only using the `prefix` keyword:

```
#prefix not;;
- : bool -> bool = <fun>
```

5.3 Strings and characters

String constants (type `string`) are written between `"` characters (double-quotes). Single-character constants (type `char`) are written between ` characters (backquotes). The most used string operation is string concatenation, denoted by the `^` character.

```
#"Hello" ^ " World!";;
- : string = "Hello World!"
```

```
#prefix ^;;
- : string -> string -> string = <fun>
```

Characters are ASCII characters:

```
#`a`;;
- : char = `a`
```

```
#`\065`;;
- : char = `A`
```

and can be built from their ASCII code as in:

```
#char_of_int 65;;
- : char = 'A'
```

Other operations are available (`sub_string`, `int_of_char`, etc). See sections 21.15, 21.7, 21.2 and 21.5, for complete information.

5.4 Tuples

5.4.1 Constructing tuples

It is possible to combine values into tuples (pairs, triples, ...). The *value constructor* for tuples is the "`,`" character (the comma). We will often use parentheses around tuples in order to improve readability, but they are not strictly necessary.

```
#1,2;;
- : int * int = 1, 2

#1,2,3,4;;
- : int * int * int * int = 1, 2, 3, 4

#let p = (1+2, 1<2);;
p : int * bool = 3, true
```

The infix "`*`" identifier is the *type constructor* for tuples. For instance, $t_1 * t_2$ corresponds to the mathematical cartesian product of types t_1 and t_2.

We can build tuples from any values: the tuple value constructor is *generic*.

5.4.2 Extracting pair components

Projection functions are used to extract components of tuples. For pairs, we have:

```
#fst;;
- : 'a * 'b -> 'a = <fun>

#snd;;
- : 'a * 'b -> 'b = <fun>
```

They have polymorphic types, of course, since they may be applied to any kind of pair. They are predefined in the Caml system, but could be defined by the user (in fact, the cartesian product itself could be defined — see section 7.1, dedicated to user-defined product types):

```
#let first (x,y) = x
#and second (x,y) = y;;
first : 'a * 'b -> 'a = <fun>
second : 'a * 'b -> 'b = <fun>

#first p;;
- : int = 3

#second p;;
- : bool = true
```

We say that `first` is a *generic* value because it works uniformly on several kinds of arguments (provided they are pairs). We often confuse between "generic" and "polymorphic", saying that such value is polymorphic instead of generic.

5.5 Patterns and pattern-matching

Patterns and pattern-matching play an important role in ML languages. They appear in all real programs and are strongly related to types (predefined or user-defined).

A *pattern* indicates the *shape* of a value. Patterns are "values with holes". A single variable (formal parameter) is a pattern (with no shape specified: it matches any value). When a value is *matched against* a pattern (this is called *pattern-matching*), the pattern acts as a filter. We already used patterns and pattern-matching in all the functions we wrote: the function body `(function x -> ...)` does (trivial) pattern-matching. When applied to a value, the formal parameter `x` gets bound to that value. The function body `(function (x,y) -> x+y)` does also pattern-matching: when applied to a value (a pair, because of well-typing hypotheses), the `x` and `y` get bound respectively to the first and the second component of that pair.

All these pattern-matching examples were trivial ones, they did not involve any test:

- formal parameters such as `x` do not impose any particular shape to the value they are supposed to match;
- pair patterns such as `(x,y)` always match for typing reasons (cartesian product is a *product type*).

However, some types do not guarantee such a uniform shape to their values. Consider the `bool` type: an element of type `bool` is either `true` or `false`. If we impose to a value of type `bool` to have the shape of `true`, then pattern-matching may fail. Consider the following function:

```
#let f = function true -> false;;
> Toplevel input:
>let f = function true -> false;;
>        ^^^^^^^^^^^^^^^^^^^^^^
> Warning: pattern matching is not exhaustive
f : bool -> bool = <fun>
```

The compiler warns us that the pattern-matching may fail (we did not consider the `false` case).

Thus, if we apply `f` to a value that evaluates to `true`, pattern-matching will succeed (an equality test is performed during execution).

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

But, if `f`'s argument evaluates to `false`, a run-time error is reported:

```
#f (1>2);;
Uncaught exception: Match_failure ("", 1343, 1365)
```

Here is a correct definition using pattern-matching:

```
#let negate = function true -> false
#                     | false -> true;;
negate : bool -> bool = <fun>
```

The pattern-matching has now two cases, separated by the "|" character. Cases are examined in turn, from top to bottom. An equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                     | x -> true;;
negate : bool -> bool = <fun>
```

The second case now matches any boolean value (in fact, only `false` since `true` has been caught by the first match case). When the second case is chosen, the identifier `x` gets bound to the argument of `negate`, and could be used in the right-hand part of the match case. Since in our example we do not use the value of the argument in the right-hand part of the second match case, another equivalent definition of `negate` would be:

```
#let negate = function true -> false
#                     | _ -> true;;
negate : bool -> bool = <fun>
```

Where "_" acts as a formal paramenter (matches any value), but does not introduce any binding: it should be read as "anything else".

As an example of pattern-matching, consider the following function definition (truth value table of implication):

```
#let imply = function (true,true) -> true
#                   | (true,false) -> false
#                   | (false,true) -> true
#                   | (false,false) -> true;;
imply : bool * bool -> bool = <fun>
```

Here is another way of defining `imply`, by using variables:

```
#let imply = function (true,x) -> x
#                   | (false,x) -> true;;
imply : bool * bool -> bool = <fun>
```

Simpler, and simpler again:

```
#let imply = function (true,x) -> x
#                   | (false,_) -> true;;
imply : bool * bool -> bool = <fun>
```

```
#let imply = function (true,false) -> false
#                   | _ -> true;;
imply : bool * bool -> bool = <fun>
```

Pattern-matching is allowed on any type of value (non-trivial pattern-matching is only possible on types with *data constructors*).

For example:

```
#let is_zero = function 0 -> true | _ -> false;;
is_zero : int -> bool = <fun>

#let is_yes = function "oui" -> true
#                    | "si" -> true
#                    | "ya" -> true
#                    | "yes" -> true
#                    | _ -> false;;
is_yes : string -> bool = <fun>
```

5.6 Functions

The type constructor "`->`" is predefined and cannot be defined in ML's type system. We shall explore in this section some further aspects of functions and functional types.

5.6.1 Functional composition

Functional composition is easily definable in Caml. It is of course a polymorphic function:

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

The type of `compose` contains no more constraints than the ones appearing in the definition: in a sense, it is the *most general* type compatible with these constraints.

These constraints are:

- the codomain of `g` and the domain of `f` must be the same;
- `x` must belong to the domain of `g`;
- `compose f g x` will belong to the codomain of `f`.

Let's see `compose` at work:

```
#let succ x = x+1;;
succ : int -> int = <fun>

#compose succ list_length;;
- : 'a list -> int = <fun>

#(compose succ list_length) [1;2;3];;
- : int = 4
```

5.6.2 Currying

We can define two versions of the addition function:

```
#let plus = function (x,y) -> x+y;;
plus : int * int -> int = <fun>

#let add = function x -> (function y -> x+y);;
add : int -> int -> int = <fun>
```

These two functions differ only in their way of taking their arguments. The first one will take an argument belonging to a cartesian product, the second one will take a number and return a function. The `add` function is said to be *the curried version* of `plus` (in honor of the logician Haskell Curry).

The currying transformation may be written in Caml under the form of a higher-order function:

```
#let curry f = function x -> (function y -> f(x,y));;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

Its inverse function may be defined by:

```
#let uncurry f = function (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

We may check the types of `curry` and `uncurry` on particular examples:

```
#uncurry (prefix +);;
- : int * int -> int = <fun>

#uncurry (prefix ^);;
- : string * string -> string = <fun>

#curry plus;;
- : int -> int -> int = <fun>
```

Exercises

5.1 *Define functions that compute the surface area and the volume of well-known geometric objects (rectangles, circles, spheres, ...).*

5.2 *What would happen in a language submitted to call-by-value with recursion if there was no conditional construct (*`if ... then ... else ...`*)?*

5.3 *Define the* `factorial` *function such that:*

$$\texttt{factorial}\ n = n * (n-1) * \ldots * 2 * 1$$

Give two versions of `factorial`*: recursive and tail recursive.*

5.4 *Define the* `fibonacci` *function that, when given a number n, returns the nth Fibonacci number, i.e. the nth term u_n of the sequence defined by:*

$$\begin{aligned} u_1 &= 1 \\ u_2 &= 1 \\ u_{n+2} &= u_{n+1} + u_n \end{aligned}$$

5.5 *What are the types of the following expressions?*

- `uncurry compose`
- `compose curry uncurry`
- `compose uncurry curry`

Chapter 6

Lists

Lists represent an important data structure, mainly because of their success in the Lisp language. Lists in ML are *homogeneous*: a list cannot contain elements of different types. This may be annoying to new ML users, yet lists are not as fundamental as in Lisp, since ML provides a facility for introducing new types allowing the user to define more precisely the data structures needed by the program (cf. chapter 7).

6.1 Building lists

Lists are empty or non empty sequences of elements. They are built with two *value constructors*:

- `[]`, the empty list (read: *nil*);
- `::`, the non-empty list constructor (read: *cons*). It takes an element e and a list l, and builds a new list whose first element (*head*) is e and rest (*tail*) is l.

The special syntax `[`e_1`; ...; `e_n` ]` builds the list whose elements are $e_1, \ldots, e_n$. It is equivalent to e_1 `::` $(e_2$ `::` $\ldots (e_n$ `::` `[]`$) \ldots)$.

We may build lists of numbers:

```
#1::2::[];;
- : int list = [1; 2]

#[3;4;5];;
- : int list = [3; 4; 5]

#let x=2 in [1; 2; x+1; x+2];;
- : int list = [1; 2; 3; 4]
```

Lists of functions:

```
#let adds =
#  let add x y = x+y
#  in [add 1; add 2; add 3];;
adds : (int -> int) list = [<fun>; <fun>; <fun>]
```

and indeed, lists of anything desired.

We may wonder what are the types of the list (data) constructors. The empty list is a list of anything (since it has no element), it has thus the type "*list of anything*". The list constructor `::` takes an element and a list (containing elements with the same type) and returns a new list. Here again, there is no type constraint on the elements considered.

```
#[];;
- : 'a list = []

#function head -> function tail -> head::tail;;
- : 'a -> 'a list -> 'a list = <fun>
```

We see here that the `list` type is a *recursive* type. The `::` constructor receives two arguments; the second argument is itself a `list`.

6.2 Extracting elements from lists: pattern-matching

We know how to build lists; we now show how to test emptiness of lists (although the equality predicate could be used for that) and extract elements from lists (e.g. the first one). We have used pattern-matching on pairs, numbers or boolean values. The syntax of patterns also includes list patterns. (We will see that any data constructor can actually be used in a pattern.) For lists, at least two cases have to be considered (empty, non empty):

```
#let is_null = function [] -> true | _ -> false;;
is_null : 'a list -> bool = <fun>

#let head = function x::_ -> x
#                  | _ -> raise (Failure "head");;
head : 'a list -> 'a = <fun>

#let tail = function _::l -> l
#                  | _ -> raise (Failure "tail");;
tail : 'a list -> 'a list = <fun>
```

The expression `raise (Failure "head")` will produce a run-time error when evaluated. In the definition of `head` above, we have chosen to forbid taking the head of an empty list. We could have chosen `tail []` to evaluate to `[]`, but we cannot return a value for `head []` without changing the type of the `head` function.

We say that the types `list` and `bool` are *sum types*, because they are defined with several alternatives:

- a list is either `[]` or `::` of ...
- a boolean value is either `true` or `false`

Lists and booleans are typical examples of sum types. Sum types are the only types whose values need run-time tests in order to be matched by a non-variable pattern (i.e. a pattern that is not a single variable).

The cartesian product is a *product* type (only one alternative). Product types do not involve run-time tests during pattern-matching, because the type of their values suffices to indicate statically what their structure is.

6.3 Functions over lists

We will see in this section the definition of some useful functions over lists. These functions are of general interest, but are not exhaustive. Some of them are predefined in the Caml Light system. See also [9] or [35] for other examples of functions over lists.

Computation of the length of a list:

```
#let rec length = function [] -> 0
#                        | _::l -> 1 + length(l);;
length : 'a list -> int = <fun>

#length [true; false];;
- : int = 2
```

Concatenating two lists:

```
#let rec append =
#     function [], l2 -> l2
#            | (x::l1), l2 -> x::(append (l1,l2));;
append : 'a list * 'a list -> 'a list = <fun>
```

The `append` function is already defined in Caml, and bound to the infix identifier `@`.

```
#[1;2] @ [3;4];;
- : int list = [1; 2; 3; 4]
```

Reversing a list:

```
#let rec rev = function [] -> []
#                     | x::l -> (rev l) @ [x];;
rev : 'a list -> 'a list = <fun>

#rev [1;2;3];;
- : int list = [3; 2; 1]
```

The `map` function applies a function on all the elements of a list, and return the list of the function results. It demonstrates full functionality (it takes a function as argument), list processing, and polymorphism (note the sharing of type variables between the arguments of `map` in its type).

```
#let rec map f =
#    function [] -> []
#           | x::l -> (f x)::(map f l);;
map : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map (function x -> x+1) [1;2;3;4;5];;
- : int list = [2; 3; 4; 5; 6]

#map length [ [1;2;3]; [4;5]; [6]; [] ];;
- : int list = [3; 2; 1; 0]
```

The following function is a list iterator. It takes a function f, a base element a and a list `[`x_1`;...;`x_n`]`. It computes:

$$\texttt{it\_list}\ f\ a\ [x_1;\ldots;x_n] = f\ (\ldots(f\ (f\ a\ x_1)\ x_2)\ \ldots)x_n$$

For instance, when applied to the curried addition function, to the base element `0`, and to a list of numbers, it computes the sum of all elements in the list. The expression:

```
it_list (prefix +) 0 [1;2;3;4;5]
```

evaluates to ((((0+1)+2)+3)+4)+5

i.e. to `15`.

```
#let rec it_list f a =
#        function [] -> a
#               | x::l -> it_list f (f a x) l;;
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a = <fun>

#let sigma = it_list prefix + 0;;
sigma : int list -> int = <fun>

#sigma [1;2;3;4;5];;
- : int = 15

#it_list (prefix *) 1 [1;2;3;4;5];;
- : int = 120
```

The `it_list` function is one of the many ways to iterate over a list. For other list iteration functions, see [9].

Exercises

6.1 *Define the* `combine` *function which, when given a pair of lists, returns a list of pairs such that:*

```
combine ([x1;...;xn],[y1;...;yn]) = [(x1,y1);...;(xn,yn)]
```

The function generates a run-time error if the argument lists do not have the same length.

6.2 *Define a function which, when given a list, returns the list of all its sublists.*

Chapter 7

User-defined types

The user is allowed to define his/her own data types. With this facility, there is no need to encode the data structures that must be manipulated by a program into lists (as in Lisp) or into arrays (as in Fortran). Furthermore, early detection of type errors is enforced, since user-defined data types reflect precisely the needs of the algorithms.

Types are either:

- *product* types,
- or *sum* types.

We have already seen examples of both kinds of types: the `bool` and `list` types are sum types (they contain values with different shapes and are defined and matched using several alternatives). The cartesian product is an example of a product type: we know statically the shape of values belonging to cartesian products.

In this chapter, we will see how to define and use new types in Caml.

7.1 Product types

Product types are *finite labeled* products of types. They are a generalization of cartesian product. Elements of product types are called *records*.

7.1.1 Defining product types

An example: suppose we want to define a data structure containing information about individuals. We could define:

```
#let jean=("Jean",23,"Student","Paris");;
jean : string * int * string * string = "Jean", 23, "Student", "Paris"
```

and use pattern-matching to extract any particular information about the person `jean`. The problem with such usage of cartesian product is that a function `name_of` returning the name field of a value representing an individual would have the same type as the general first projection for 4-tuples (and indeed would be the same function). This type is not precise enough since it allows for the application of the function to any 4-uple, and not only to values such as `jean`.

Instead of using cartesian product, we define a `person` data type:

```
#type person =
#  {Name:string; Age:int; Job:string; City:string};;
Type person defined.
```

The type `person` is the *product* of `string`, `int`, `string` and `string`. The field names provide type information and also documentation: it is much easier to understand data structures such as `jean` above than arbitrary tuples.

We have *labels* (i.e. `Name`, ...) to refer to components of the products. The order of appearance of the products components is not relevant: labels are sufficient to uniquely identify the components. The Caml system finds a canonical order on labels to represent and print record values. The order is always the order which appeared in the definition of the type.

We may now define the individual `jean` as:

```
#let jean = {Job="Student"; City="Paris";
#            Name="Jean"; Age=23};;
jean : person = {Name="Jean"; Age=23; Job="Student"; City="Paris"}
```

This example illustrates the fact that order of labels is not relevant.

7.1.2 Extracting products components

The canonical way of extracting product components is *pattern-matching*. Pattern-matching provides a way to mention the shape of values and to give (local) names to components of values. In the following example, we name `n` the numerical value contained in the field `Age` of the argument, and we choose to forget values contained in other fields (using the `_` character).

```
#let age_of = function
#     {Age=n; Name=_; Job=_; City=_} -> n;;
age_of : person -> int = <fun>

#age_of jean;;
- : int = 23
```

It is also possible to access the value of a single field, with the `.` (dot) operator:

```
#jean.Age;;
- : int = 23

#jean.Job;;
- : string = "Student"
```

Labels always refer to the most recent type definition: when two record types possess some common labels, then these labels always refer to the most recently defined type. When using modules (see section 12.2) this problem arises for types defined in the same module. For types defined in different modules, the full name of labels (i.e. with the name of the modules prepended) disambiguates such situations.

7.1.3 Parameterized product types

It is important to be able to define parameterized types in order to define *generic* data structures. The `list` type is parameterized, and this is the reason why we may build lists of any kind of values. If we want to define the cartesian product as a Caml type, we need type parameters because we want to be able to build cartesian product of *any* pair of types.

```
#type ('a,'b) pair = {Fst:'a; Snd:'b};;
Type pair defined.

#let first x = x.Fst and second x = x.Snd;;
first : ('a, 'b) pair -> 'a = <fun>
second : ('a, 'b) pair -> 'b = <fun>

#let p={Snd=true; Fst=1+2};;
p : (int, bool) pair = {Fst=3; Snd=true}

#first(p);;
- : int = 3
```

Warning: the `pair` type is similar to the Caml cartesian product `*`, but it is a different type.

```
#fst p;;
> Toplevel input:
>fst p;;
>    ^
> Expression of type (int, bool) pair
> cannot be used with type 'a * 'b
```

Actually, any two type definitions produce different types. If we enter again the previous definition:

```
#type ('a,'b) pair = {Fst:'a; Snd:'b};;
Type pair defined.
```

we cannot any longer extract the `Fst` component of `x`:

```
#p.Fst;;
> Toplevel input:
>p.Fst;;
>^
> Expression of type (int, bool) pair
> cannot be used with type ('a, 'b) pair
```

since the label `Fst` refers to the *latter* type `pair` that we defined, while `p`'s type is the *former* `pair`.

7.2 Sum types

A *sum* type is the *finite labeled* disjoint union of several types. A sum type definition defines a type as being the union of some other types.

7.2.1 Defining sum types

Example: we want to have a type called `identification` whose values can be:

- either strings (name of an individual),
- or integers (encoding of social security number as a pair of integers).

We then need a type containing *both* `int * int` and character strings. We also want to preserve static type-checking, we thus want to be able to distinguish pairs from character strings even if they are injected in order to form a single type.

Here is what we would do:

```
#type identification = Name of string
#                    | SS of int * int;;
Type identification defined.
```

The type `identification` is the labeled disjoint union of `string` and `int * int`. The labels `Name` and `SS` are *injections*. They respectively inject `int * int` and `string` into a single type `identification`.

Let us use these injections in order to build new values:

```
#let id1 = Name "Jean";;
id1 : identification = Name "Jean"

#let id2 = SS (1670728,280305);;
id2 : identification = SS (1670728, 280305)
```

Values `id1` and `id2` belong to the same type. They may for example be put into lists as in:

```
#[id1;id2];;
- : identification list = [Name "Jean"; SS (1670728, 280305)]
```

Injections may possess one argument (as in the example above), or none. The latter case corresponds[1] to *enumerated types*, as in Pascal. An example of enumerated type is:

```
#type suit = Heart
#          | Diamond
#          | Club
#          | Spade;;
Type suit defined.

#Club;;
- : suit = Club
```

The type `suit` contains only 4 distinct elements. Let us continue this example by defining a type for cards.

[1] In Caml Light, there is no implicit order on values of sum types.

```
#type card = Ace of suit
#          | King of suit
#          | Queen of suit
#          | Jack of suit
#          | Plain of suit * int;;
Type card defined.
```

The type `card` is the disjoint union of:

- `suit` under the injection `Ace`,
- `suit` under the injection `King`,
- `suit` under the injection `Queen`,
- `suit` under the injection `Jack`,
- the product of `int` and `suit` under the injection `Plain`.

Let us build a list of cards:

```
#let figures_of c = [Ace c; King c; Queen c; Jack c]
#and small_cards_of c =
#    map (function n -> Plain(c,n)) [7;8;9;10];;
figures_of : suit -> card list = <fun>
small_cards_of : suit -> card list = <fun>

#figures_of Heart;;
- : card list = [Ace Heart; King Heart; Queen Heart; Jack Heart]

#small_cards_of Spade;;
- : card list = [Plain (Spade, 7); Plain (Spade, 8); Plain (Spade, 9); Plain (Sp
ade, 10)]
```

7.2.2 Extracting sum components

Of course, pattern-matching is used to extract sum components. In case of sum types, pattern-matching uses dynamic tests for this extraction. The next example defines a function `color` returning the name of the color of the suit argument:

```
#let color = function Diamond -> "red"
#                   | Heart -> "red"
#                   | _ -> "black";;
color : suit -> string = <fun>
```

Let us count the values of cards (assume we are playing "belote"):

```
#let count trump = function
#    Ace _        -> 11
#  | King _       -> 4
```

```
#  | Queen _      -> 3
#  | Jack c       -> if c = trump then 20 else 2
#  | Plain (c,10) -> 10
#  | Plain (c,9)  -> if c = trump then 14 else 0
#  | _             -> 0;;
count : suit -> card -> int = <fun>
```

7.2.3 Recursive types

Some types possess a naturally recursive structure, lists, for example. It is also the case for tree-like structures, since trees have subtrees that are trees themselves.

Let us define a type for abstract syntax trees of a simple arithmetic language[2]. An arithmetic expression will be either a numeric constant, or a variable, or the addition, multiplication, difference, or division of two expressions.

```
#type arith_expr = Const of int
#                | Var of string
#                | Plus of args
#                | Mult of args
#                | Minus of args
#                | Div of args
#and args = {Arg1:arith_expr; Arg2:arith_expr};;
Type arith_expr defined.
Type args defined.
```

The two types `arith_expr` and `args` are simultaneously defined, and `arith_expr` is recursive since its definition refers to `args` which itself refers to `arith_expr`. As an example, one could represent the abstract syntax tree of the arithmetic expression "`x+(3*y)`" as the Caml value:

```
#Plus {Arg1=Var "x";
#       Arg2=Mult{Arg1=Const 3; Arg2=Var "y"}};;
- : arith_expr = Plus {Arg1=Var "x"; Arg2=Mult {Arg1=Const 3; Arg2=Var "y"}}
```

The recursive definition of types may lead to types such that it is hard or impossible to build values of these types. For example:

```
#type stupid = {Head:stupid; Tail:stupid};;
Type stupid defined.
```

Elements of this type are *infinite* data structures. Essentially, the only way to construct one is:

```
#let rec stupid_value =
#    {Head=stupid_value; Tail=stupid_value};;
stupid_value : stupid = {Head={Head={Head={Head={Head={Head={Head={Head={Head={H
ead={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Hea
```

[2]Syntax trees are said to be *abstract* because they are pieces of *abstract syntax* contrasting with *concrete syntax* which is the "string" form of programs: analyzing (parsing) concrete syntax usually produces abstract syntax.

```
d={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head=
{Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={H
ead={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Hea
d={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head=
{Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={H
ead={Head={Head={Head={Head={Head={Head={Head={Head={Head=...
```

Recursive type definitions should be *well-founded* (i.e. possess a non-recursive case, or *base case*) in order to work well with call-by-value.

7.2.4 Parameterized sum types

Sum types may also be parameterized. Here is the definition of a type isomorphic to the `list` type:

```
#type 'a sequence = Empty
#                 | Sequence of 'a * 'a sequence;;
Type sequence defined.
```

A more sophisticated example is the type of generic binary trees:

```
#type ('a,'b) btree = Leaf of 'b
#                   | Btree of ('a,'b) node
#and ('a,'b) node = {Op:'a;
#                    Son1: ('a,'b) btree;
#                    Son2: ('a,'b) btree};;
Type btree defined.
Type node defined.
```

A binary tree is either a leaf (holding values of type `'b`) or a node composed of an operator (of type `'a`) and two sons, both of them being binary trees.

Binary trees can also be used to represent abstract trees for arithmetic expressions (with only binary operators and only one kind of leaves). The abstract syntax tree `t` of "`1+2`" could be defined as:

```
#let t = Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2};;
t : (string, int) btree = Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2}
```

Finally, it is time to notice that pattern-matching is not restricted to function bodies, i.e. constructs such as:

```
function  P1   -> E1
       |  ...
       |  Pn   -> En
```

but there is also a construct dedicated to pattern-matching of actual values:

```
match E with  P1   -> E1
           |  ...
           |  Pn   -> En
```

which matches the value of the expression E against each of the patterns P_i, selecting the first one that matches, and giving control to the corresponding expression. For example, we can match the tree `t` previously defined by:

```
#match t with Btree{Op=x; Son1=_; Son2=_} -> x
#           | Leaf l -> "No operator";;
- : string = "+"
```

7.2.5 Data constructors and functions

One may ask: "What is the difference between a sum data constructor and a function?". At first sight, they look very similar. We assimilate constant data constructors (such as `Heart`) to constants. However, in Caml Light, sum data constructors with arguments are not presented as functions from the element types to the sum type:

```
#Ace;;
> Toplevel input:
>Ace;;
>^^^
> Constructor Ace expects an argument
```

The reason is that a data constructor possesses particular properties that a general function does not possess, and it is interesting to understand these differences. From the mathematical point of view, a sum data constructor is known to be an *injection* while a Caml function is a general function without further information. A mathematical injection $f : A \to B$ admits an inverse function f^{-1} from its image $f(A) \subset B$ to A.

From the examples above, if we consider the `King` constructor, then:

```
#let king c = King c;;
king : suit -> card = <fun>
```

`king` is the general function associated to the `King` constructor, and:

```
#function King c -> c;;
> Toplevel input:
>function King c -> c;;
>^^^^^^^^^^^^^^^^^^^
> Warning: pattern matching is not exhaustive
- : card -> suit = <fun>
```

is the inverse function for `king`. It is a partial function, since pattern-matching may fail.

7.2.6 Degenerate cases: when sums meet products

What is the status of a sum type with a single case such as:

```
#type counter1 = Counter of int;;
Type counter1 defined.
```

Of course, the type `counter1` is isomorphic to `int`. The injection `function x -> Counter x` is a *total* function from `int` to `counter1`. It is thus a *bijection*.

Another way to define a type isomorphic to `int` would be:

```
#type counter2 = {Counter: int};;
Type counter2 defined.
```

The types `counter1` and `counter2` are isomorphic to `int`. They are at the same time sum and product types. Their pattern-matching does not perform any run-time test.

The possibility of defining arbitrary complex data types permits the easy manipulation of abstract syntax trees in Caml (such as the `arith_expr` type above). These abstract syntax trees are supposed to represent programs of a language (e.g. a language of arithmetic expressions). These kind of languages which are defined in Caml are called *object-languages* and Caml is said to be their *metalanguage*.

7.3 Summary

- New types can be introduced in Caml.
- Types may be *parameterized* by type variables. The syntax of type parameters is:

  ```
  <params> ::
            | <tvar>
            | ( <tvar> [, <tvar>]* )
  ```

- Types can be *recursive*.
- Product types:
  - Mathematical product of several types.
  - The construct is:

    ```
    type <params> <tname> =
                      {<Field>: <type>; ...}
    ```

    where the `<type>`s may contain type variables appearing in `<params>`.
- Sum types:
  - Mathematical disjoint union of several types.
  - The construct is:

    ```
    type <params> <tname> =
             <Injection> [of <type>] | ...
    ```

    where the `<type>`s may contain type variables appearing in `<params>`.

Exercises

7.1 *Define a function taking as argument a binary tree and returning a pair of lists: the first one contains all operators of the tree, the second one contains all its leaves.*

7.2 *Define a function* `map_btree` *analogous to the* `map` *function on lists. The function* `map_btree` *should take as arguments two functions* `f` *and* `g`*, and a binary tree. It should return a new binary tree whose leaves are the result of applying* `f` *to the leaves of the tree argument, and whose operators are the results of applying the* `g` *function to the operators of the argument.*

7.3 *We can associate to the* `list` *type definition an canonical iterator in the following way. We give a functional interpretation to the data constructors of the* `list` *type.*

We change the list constructors `[]` *and* `::` *respectively into a constant* `a` *and an operator* $\oplus$ *(used as a prefix identifier), and abstract with respect to these two operators, obtaining the list iterator satisfying:*

`list_it` $\oplus$ `a []` = `a`
`list_it` $\oplus$ `a` (x_1`::`$\ldots$`::`x_n`::[]`) = x_1 $\oplus$ (`...`$\oplus$ (x_n $\oplus$ `a`)`...`)

Its Caml definition would be:

```
#let list_it f a = it_rec
#where rec it_rec =
#        function [] -> a
#               | x::L -> f x (it_rec L);;
list_it : ('a -> 'b -> 'b) -> 'b -> 'a list -> 'b = <fun>
```

As an example, the application of `it_list` *to the functional composition and to its neutral element (the identity function), computes the composition of lists of functions (try it!).*

Define, using the same method, a canonical iterator over binary trees.

Part B
Caml Light specifics

Chapter 8

Mutable data structures

The definition of a sum or product type may be annotated to allow physical (destructive) update on data structures of that type. This is the main feature of the *imperative programming* style. Writing values into memory locations is the fundamental mechanism of imperative languages such as Pascal. The Lisp language, while mostly functional, also provides the dangerous functions `rplaca` and `rplacd` to physically modify lists. Mutable structures are required to implement many efficient algorithms. They are also very convenient to represent the current state of a state machine.

8.1 User-defined mutable data structures

Assume we want to define a type `person` as in the previous chapter. Then, it seems natural to allow a person to change his/her age, job and the city that person lives in, but *not* his/her name. We can do this by annotating some labels in the type definition of `person` by the `mutable` keyword:

```
#type person =
#    {Name: string; mutable Age: int;
#     mutable Job: string; mutable City: string};;
Type person defined.
```

We can build values of type `person` in the very same way as before:

```
#let jean =
#    {Name="Jean"; Age=23; Job="Student"; City="Paris"};;
jean : person = {Name="Jean"; Age=23; Job="Student"; City="Paris"}
```

But now, the value `jean` may be physically modified in the fields specified to be `mutable` in the definition (and *only* in these fields).

We can modify the field `Field` of an expression `<expr1>` in order to assign it the value of `<expr2>` by using the following construct:

```
<expr1>.Field <- <expr2>
```

For example; if we want `jean` to become one year older, we would write:

```
#jean.Age <- jean.Age + 1;;
- : unit = ()
```

Now, the value `jean` has been modified into:

```
#jean;;
- : person = {Name="Jean"; Age=24; Job="Student"; City="Paris"}
```

We may try to change the `Name` of `jean`, but we won't succeed: the typecheker will not allow us to do that.

```
#jean.Name <- "Paul";;
> Toplevel input:
>jean.Name <- "Paul";;
>^^^^^^^^^^^^^^^^^^^
> Label Name is not mutable.
```

It is of course possible to use such constructs in functions as in:

```
#let get_older ({Age=n; _} as p) = p.Age <- n + 1;;
get_older : person -> unit = <fun>
```

In that example, we named `n` the current `Age` of the argument, but we also named `p` the argument. This is an *alias* pattern: it saves us the bother of writing:

```
#let get_older p =
#    match p with {Age=n} -> p.Age <- n + 1;;
get_older : person -> unit = <fun>
```

Notice that in the two previous expressions, we did not specify all fields of the record `p`. Other examples would be:

```
#let move p new_city = p.City <- new_city
#and change_job p j = p.Job <- j;;
move : person -> string -> unit = <fun>
change_job : person -> string -> unit = <fun>

#change_job jean "Teacher"; move jean "Cannes";
#get_older jean; jean;;
- : person = {Name="Jean"; Age=25; Job="Teacher"; City="Cannes"}
```

We used the ";" character between the different changes we imposed to `jean`. This is the *sequencing* of evaluations: it permits to evaluate successively several expressions, discarding the result of each (except the last one). This construct becomes useful in the presence of *side-effects* such as physical modifications and input/output, since we want to explicitly specify the order in which they are performed.

8.2 The `ref` type

The `ref` type is the predefined type of mutable indirection cells. It is present in the Caml system for reasons of compatibility with earlier versions of Caml. The `ref` type could be defined as follows (we don't use the `ref` name in the following definition because we want to preserve the original `ref` type):

```
#type 'a reference = {mutable Ref: 'a};;
Type reference defined.
```

Example of building a value of type `ref`:

```
#let r = ref (1+2);;
r : int ref = ref 3
```

The `ref` identifier is syntactically presented as a sum data constructor. The definition of `r` should be read as "let `r` be a reference to the value of `1+2`". The value of `r` is nothing but a memory location whose contents can be overwritten.

We consult a reference (i.e. read its memory location) with the "`!`" symbol:

```
#!r + 1;;
- : int = 4
```

We modify values of type `ref` with the `:=` infix function:

```
#r:=!r+1;;
- : unit = ()

#r;;
- : int ref = ref 4
```

Some primitives are attached to the `ref` type, for example:

```
#incr;;
- : int ref -> unit = <fun>

#decr;;
- : int ref -> unit = <fun>
```

which increments (resp. decrements) references on integers. A description of primitives on references can be found in section 21.13.

8.3 Arrays

Arrays are modifiable data structures. They belong to the parameterized type `'a vect`. Array expressions are bracketed by `[|` and `|]`, and elements are separated by semicolons:

```
#let a = [| 10; 20; 30 |];;
a : int vect = [|10; 20; 30|]
```

The length of an array is returned by with the function `vect_length`:

```
#vect_length a;;
- : int = 3
```

8.3.1 Accessing array elements

Accesses to array elements can be done using the following syntax:

```
#a.(0);;
- : int = 10
```

or, more generally: e_1`.(`e_2`)`, where e_1 evaluates to an array and e_2 to an integer. Alternatively, the function `vect_item` is provided:

```
#vect_item;;
- : 'a vect -> int -> 'a = <fun>
```

The first element of an array is at index 0. Arrays are useful because accessing an element is done in constant time: an array is a contiguous fragment of memory, while accessing list elements takes linear time.

8.3.2 Modifying array elements

Modification of an array element is done with the construct:

$$e_1\texttt{.(}e_2\texttt{) <-}\ e_3$$

where e_3 has the same type as the elements of the array e_1. The expression e_2 computes the index at which the modification will occur.

As for accessing, a function for modifying array elements is also provided:

```
#vect_assign;;
- : 'a vect -> int -> 'a -> unit = <fun>
```

For example:

```
#a.(0) <- (a.(0)-1);;
- : unit = ()

#a;;
- : int vect = [|9; 20; 30|]

#vect_assign a 0 ((vect_item a 0) - 1);;
- : unit = ()

#a;;
- : int vect = [|8; 20; 30|]
```

More information of arrays and associated primitives can be found in sections 21.16 and 21.8.

8.4 Loops: `while` and `for`

Imperative programming (i.e. using side-effects such as physical modification of data structures) traditionally makes use of sequences and explicit loops. Sequencing evaluation in Caml Light is done by using the semicolon ";". Evaluating expression e_1, discarding the value returned, and then evaluating e_2 is written:

e_1 `;` e_2

If e_1 and e_2 perform side-effects, this construct ensures that they will be performed in the specified order (from left to right). In order to emphasize sequential side-effects, instead of using parentheses around sequences, one can use `begin` and `end`, as in:

```
#let x = ref 1 in
#  begin
#     x:=!x+1;
#     x:=!x*!x
#  end;;
- : unit = ()
```

The keywords `begin` and `end` are equivalent to opening and closing parentheses. The program above could be written as:

```
#let x = ref 1 in
#  (x:=!x+1; x:=!x*!x);;
- : unit = ()
```

Explicit loops are not strictly necessary *per se*: a recursive function could perform the same work. However, the usage of an explicit loop locally emphasizes a more imperative style. Two loops are provided:

- *while*: `while` e_1 `do` e_2 `done` evaluates e_1 which must return a boolean expression, if e_1 return `true`, then e_2 (which is usually a sequence) is evaluated, then e_1 is evaluated again and so on until e_1 returns `false`.
- *for*: two variants, increasing and decreasing
  - `for` v`=`e_1 `to` e_2 `do` e_3 `done`
  - `for` v`=`e_1 `downto` e_2 `do` e_3 `done`

  where v is an identifier. Expressions e_1 and e_2 are the bounds of the loop: they must evaluate to integers. In the case of the increasing loop, the expressions e_1 and e_2 are evaluated producing values n_1 and n_2 : if n_1 is strictly greater than n_2, then nothing is done. Otherwise, e_3 is evaluated $(n_2 - n_1) + 1$ times, with the variable v bound successively to n_1, $n_1 + 1$, ..., n_2.

  The behavior of the decreasing loop is similar: if n_1 is strictly smaller than n_2, then nothing is done. Otherwise, e_3 is evaluated $(n_1 - n_2) + 1$ times with v bound to successive values decreasing from n_1 to n_2.

Both loops return the value `()` of type `unit`.

```
#for i=0 to (vect_length a) - 1 do a.(i) <- i done;;
- : unit = ()

#a;;
- : int vect = [|0; 1; 2|]
```

8.5 Polymorphism and mutable data structures

There are some restrictions concerning polymorphism and mutable data structures. One cannot enclose polymorphic objects inside mutable data structures.

```
#let r = ref [];;
> Toplevel input:
>let r = ref [];;
>^^^^^^^^^^^^^^^
> Cannot generalize 'a in 'a list ref
```

The reason is that once the type of `r`, `('a list) ref`, has been computed, it cannot be changed. But the value of `r` can be changed: we could write:

```
r:=[1;2];;
```

and now, `r` would be a reference on a list of numbers while its type would still be `('a list) ref`, allowing us to write:

```
r:= true::!r;;
```

making `r` a reference on `[true; 1; 2]`, which is an illegal Caml object.

Thus the Caml typechecker imposes that modifiable data structures appearing at toplevel must possess monomorphic types (i.e. not polymorphic).

Exercises

8.1 *Give a mutable data type defining the Lisp type of lists and define the four functions* `car`, `cdr`, `rplaca` *and* `rplacd`.

8.2 *Define a* `stamp` *function, of type* `unit -> int`, *that returns a fresh integer each time it is called. That is, the first call returns 1; the second call returns 2; and so on.*

8.3 *Define a* `quick_sort` *function on arrays of floating point numbers following the* quicksort *algorithm [13]. Information about the* quicksort *algorithm can be found in [31], for example.*

Chapter 9

Escaping from computations: exceptions

In some situations, it is necessary to abort computations. If we are trying to compute the integer division of an integer `n` by `0`, then we must escape from that embarrassing situation without returning any result.

Another example of the usage of such an escape mechanism appears when we want to define the `head` function on lists:

```
#let head = function
#     x::L -> x
#   | [] -> raise (Failure "head: empty list");;
head : 'a list -> 'a = <fun>
```

We cannot give a regular value to the expression `head []` without losing the polymorphism of `head`. We thus choose to escape: we *raise an exception*.

9.1 Exceptions

An exception is a Caml value of the built-in type `exn`, very similar to a sum type. An exception:

- has a *name* (`Failure` in our example),
- and holds zero or one value (`"head: empty list"` of type `string` in the example).

Some exceptions are predefined, like `Failure`. New exceptions can be defined with the following construct:

```
exception <exception name> [of <type>]
```

Example:

```
#exception Found of int;;
Exception Found defined.
```

The exception `Found` has been declared, and it carries integer values. When we apply it to an integer, we get an exception value (of type `exn`):

```
#Found 5;;
- : exn = Found 5
```

9.2 Raising an exception

Raising an exception is done by applying the primitive function `raise` to a value of type `exn`:

```
#raise;;
- : exn -> 'a = <fun>
```

```
#raise (Found 5);;
Uncaught exception: Found 5
```

Here is a more realistic example:

```
#let find_index p = find 1
#where rec find n =
#   function [] -> raise (Failure "not found")
#          | x::L -> if p(x) then raise (Found n)
#                    else find (n+1) L;;
find_index : ('a -> bool) -> 'a list -> 'b = <fun>
```

The `find_index` function always fails. It raises:

- `Found n`, if there is an element `x` of the list such that `p(x)`, in this case `n` is the index of `x` in the list,
- the `Failure` exception if no such `x` has been found.

Raising exceptions is more than an error mechanism: it is a programmable control structure. In the `find_index` example, there was no error when raising the `Found` exception: we only wanted to quickly escape from the computation, since we found what we were looking for. This is why it must be possible to *trap* exceptions: we want to trap possible errors, but we also want to get our result in the case of the `find_index function`.

9.3 Trapping exceptions

Trapping exceptions is achieved by the following construct:

```
try <expression> with <match cases>
```

This construct evaluates `<expression>`. If no exception is raised during the evaluation, then the result of the `try` construct is the result of `<expression>`. If an exception is raised during this evaluation, then the raised exception is matched against the `<match cases>`. If a case matches, then control is passed to it. If no case matches, then the exception is propagated outside of the `try` construct, looking for the enclosing `try`.

Example:

```
#let find_index p L =
#  let rec find n =
#    function [] -> raise (Failure "not found")
#           | x::L -> if p(x) then raise (Found n)
#                     else find (n+1) L
#  in
#    try find 1 L with Found n -> n;;
find_index : ('a -> bool) -> 'a list -> int = <fun>

#find_index (function n -> (n mod 2) = 0) [1;3;5;7;9;10];;
- : int = 6

#find_index (function n -> (n mod 2) = 0) [1;3;5;7;9];;
Uncaught exception: Failure "not found"
```

The `<match cases>` part of the `try` construct is a regular pattern matching on values of type `exn`. It is thus possible to trap any exception by using the `_` symbol. As an example, the following function traps any exception raised during the application of its two arguments. Warning: the `_` will also trap interrupts from the keyboard such as control-C!

```
#let catch_all f arg default =
#       try f(arg) with _ -> default;;
catch_all : ('a -> 'b) -> 'a -> 'b -> 'b = <fun>
```

It is even possible to catch all exceptions, do something special (close or remove opened files, for example), and raise again that exception, to propagate it upwards.

```
#let show_exceptions f arg =
#        try f(arg) with x -> print_string "Exception raised!\n"; raise x;;
show_exceptions : ('a -> 'b) -> 'a -> 'b = <fun>
```

In the example above, we print a message to the standard output channel (the terminal), before raising again the trapped exception.

```
#catch_all (function x -> raise (Failure "foo")) 1 0;;
- : int = 0

#catch_all (show_exceptions (function x -> raise (Failure "foo"))) 1 0;;
Exception raised!
- : int = 0
```

9.4 Polymorphism and exceptions

Exceptions must not be polymorphic for a reason similar to the one for references (although it is a bit harder to give an example).

```
#exception Exc of 'a list;;
> Toplevel input:
>exception Exc of 'a list;;
>                 ^^^^^^^
> Type variable 'a is unbound
```

One reason is that the `excn` type is not a parameterized type, but one deeper reason is that if the exception `Exc` is declared to be polymorphic, then a function may raise `Exc [1;2]`. There might be no mention of that fact in the type inferred for the function. Then, another function may trap that exception, obtaining the value `[1;2]` whose real type is `int list`. But the only type known by the typechecker is `'a list`: the `try` form should refer to the `Exc` data constructor, which is known to be polymorphic. It may then be possible to build an ill-typed Caml value `[true; 1; 2]`, since the typechecker does not possess any further type information than `'a list`.

The problem is thus the absence of static connection from exceptions that are raised and the occurrences where they are trapped. Another example would be the one of a function raising `Exc` with an integer or a boolean value, depending on some condition. Then, in that case, when trying to trap these exceptions, we cannot decide wether they will hold integers or boolean values.

Exercises

9.1 *Define the function* `find_succeed` *which given a function* `f` *and a list* `L` *returns the first element of* `L` *on which the application of* `f` *succeeds.*

9.2 *Define the function* `map_succeed` *which given a function* `f` *and a list* `L` *returns the list of the results of successful applications of* `f` *to elements of* `L`.

Chapter 10

Basic input/output

We describe in this chapter the Caml Light input/output model and some of its primitive operations. More complete information about IO can be found in section 21.10, part IV.

Caml Light has an imperative input/output model: an IO operation should be considered as a side-effect, and is thus dependent on the order of evaluation. IOs are performed onto *channels* with types `in_channel` and `out_channel`. These types are *abstract*, i.e. their representation is not accessible.

Three channels are predefined:

```
#std_in;;
- : in_channel = <abstract>

#std_out;;
- : out_channel = <abstract>

#std_err;;
- : out_channel = <abstract>
```

They are the "standard" IO channels: `std_in` is usually connected to the keyboard, and printing onto `std_out` and `std_err` usually appears on the screen.

10.1 Printable types

It is not possible to print and read every value. Functions, for example, are typically not readable, unless a suitable string representation is designed and reading such a representation is followed by an interpretation computing the desired function.

We call *printable type* a type for which there are input/output primitives implemented in Caml Light. The main printable types are:

- characters: type `char`;
- strings: type `string`;
- integers: type `int`;
- floating point numbers: type `float`.

We know all these types from the previous chapters. Strings and characters support a notation for escaping to ASCII codes or to denote special characters such as newline:

```
#‘A‘;;
- : char = ‘A‘

#‘\065‘;;
- : char = ‘A‘

#‘\\‘;;
- : char = ‘\\‘

#‘\n‘;;
- : char = ‘\n‘

#"string with\na newline inside";;
- : string = "string with\na newline inside"
```

The “\” character is used as an escape and is useful for non-printable or special characters. A detailed account of escapes is given in section 20.1.

Of course, character constants can be used as constant patterns:

```
#function ‘a‘ -> 0 | _ -> 1;;
- : char -> int = <fun>
```

On types such as `char` that have a finite number of constant elements, it may be useful to use *or-patterns*, gathering constants in the same matching rule:

```
#let is_vowel = function
#  ‘a‘ | ‘e‘ | ‘i‘ | ‘o‘ | ‘u‘ | ‘y‘ -> true
#| _ -> false;;
is_vowel : char -> bool = <fun>
```

The first rule is chosen if the argument matches one of the cases. Since there is a total ordering on characters, the syntax of character patterns is enriched with a “`..`” notation:

```
#let is_lower_case_letter = function
#  ‘a‘..‘z‘ -> true
#| _ -> false;;
is_lower_case_letter : char -> bool = <fun>
```

Of course, or-patterns and this notation can be mixed, as in:

```
#let is_letter = function
#  ‘a‘..‘z‘ | ‘A‘..‘Z‘ -> true
#| _ -> false;;
is_letter : char -> bool = <fun>
```

In the next sections, we give the most commonly used IO primitives on these printable types. A complete listing of predefined IO operations is given in chapter 21.10, part IV.

10.2 Output

Printing on standard output is performed by the following functions:

```
#print_char;;
- : char -> unit = <fun>

#print_string;;
- : string -> unit = <fun>

#print_int;;
- : int -> unit = <fun>

#print_float;;
- : float -> unit = <fun>
```

Printing is *buffered*, i.e. the effect of a call to a printing function may not be seen immediately: *flushing* explicitly the output buffer is sometimes required, unless a printing function flushes it implicitly. Flushing is done with the `flush` function:

```
#flush;;
- : out_channel -> unit = <fun>

#print_string "Hello!"; flush std_out;;
Hello!- : unit = ()
```

The `print_newline` function prints a newline character and flushes the standard output:

```
#print_newline;;
- : unit -> unit = <fun>
```

Flushing is required when writing standalone applications, in which the application may terminate without all printing being done. Standalone applications should terminate by a call to the `exit` function (from the `io` module), which flushes all pending output on `std_out` and `std_err`.

Printing on the standard error channel `std_err` is done with the following functions:

```
#prerr_char;;
- : char -> unit = <fun>

#prerr_string;;
- : string -> unit = <fun>

#prerr_int;;
- : int -> unit = <fun>

#prerr_float;;
- : float -> unit = <fun>
```

The following function prints its string argument followed by a newline character to `std_err` and then flushes `std_err`.

```
#prerr_endline;;
- : string -> unit = <fun>
```

10.3 Input

These input primitives flush the standard output and read from the standard input:

```
#read_line;;
- : unit -> string = <fun>

#read_int;;
- : unit -> int = <fun>

#read_float;;
- : unit -> float = <fun>
```

Because of their names and types, these functions do not need further explanation.

10.4 Channels on files

When programs have to read from or print to files, it is necessary to open and close channels on these files.

10.4.1 Opening and closing channels

Opening and closing is performed with the following functions:

```
#open_in;;
- : string -> in_channel = <fun>

#open_out;;
- : string -> out_channel = <fun>

#close_in;;
- : in_channel -> unit = <fun>

#close_out;;
- : out_channel -> unit = <fun>
```

The `open_in` function checks the existence of its filename argument, and returns a new input channel on that file; `open_out` creates a new file (or truncates it to zero length if it exists) and returns an output channel on that file. Both functions fail if permissions are not sufficient for reading or writing.
Warning:

- Closing functions close their channel argument. Since their behavior is unspecified on already closed channels, anything can happen in this case!
- Closing one of the standard IO channels (`std_in`, `std_out`, `std_err`) have unpredictable effects!

10.4.2 Reading or writing from/to specified channels

Some of the functions on standard input/output have corresponding functions working on channels:

```
#output_char;;
- : out_channel -> char -> unit = <fun>

#output_string;;
- : out_channel -> string -> unit = <fun>

#input_char;;
- : in_channel -> char = <fun>

#input_line;;
- : in_channel -> string = <fun>
```

10.4.3 Failures

The exception `End_of_file` is raised when an input operation cannot complete because the end of the file has been reached.

```
#End_of_file;;
- : exn = End_of_file
```

The exception `sys__Sys_error` (`Sys_error` from the module `sys`) is raised when some manipulation of files is forbidden by the operating system:

```
#open_in "abracadabra";;
Uncaught exception: sys__Sys_error "No such file or directory"
```

The functions that we have seen in this chapter are sufficient for our needs. Many more exist (useful mainly when working with files) and are described in chapter 21.10, part IV.

Exercises

10.1 *Define a function* `copy_file` *taking two filenames (of type* `string`*) as arguments, and copying the contents of the first file on the second one. Error messages must be printed on* `std_err`*.*

10.2 *Define a function* `wc` *taking a filename as argument and printing on the standard output the number of characters and lines appearing in the file. Error messages must be printed on* `std_err`*.*

Note: it is good practice to develop a program in defining small functions. A single function doing the whole work is usually harder to debug and to read. With small functions, one can trace them (cf. section 26.2) and see the arguments they are called on and the result they produce.

Chapter 11

Streams and parsers

In the next part of these course notes, we will implement a small functional language. Parsing valid programs of this language requires writing a lexical analyzer and a parser for the language. For the purpose of writing easily such programs, Caml Light provides a special data structure: *streams*. Their main usage is to be interfaced to input channels or strings and to be matched against *stream patterns*.

11.1 Streams

Streams belong to an abstract data type: their actual representation remains hidden from the user. However, it is still possible to build streams either "by hand" or by using some predefined functions.

11.1.1 The `stream` type

The type `stream` is a parameterized type. One can build streams of integers, of characters or of any other type. Streams receive a special syntax, looking like the one for lists. The empty stream is written:

```
#[< >];;
- : 'a stream = <abstract>
```

A non empty stream possesses elements that are written preceded by the "'" (quote) character.

```
#[< '0; '1; '2 >];;
- : int stream = <abstract>
```

Elements that are not preceded by "'" are *substreams* that are expanded in the enclosing stream:

```
#[< '0; [<'1;'2>]; '3 >];;
- : int stream = <abstract>
#let s = [< '"abc" >] in [< s; '"def" >];;
- : string stream = <abstract>
```

Thus, stream concatenation can be defined as:

```
#let stream_concat s t = [< s; t >];;
stream_concat : 'a stream -> 'a stream -> 'a stream = <fun>
```

Building streams in this way can be useful while testing a parsing function or defining a lexical analyzer (taking as argument a stream of characters and returning a stream of tokens). Stream concatenation *does not copy* substreams: they are simply put in the same stream. Since (as we will see later) stream matching has a destructive effect on streams (streams are physically "eaten" by stream matching), parsing `[< t; t >]` will in fact parse `t` only once: the first occurrence of `t` will be consumed, and the second occurrence will be empty before its parsing will be performed.

Interfacing streams with an input channel can be done with the function:

```
#stream_of_channel;;
- : in_channel -> char stream = <fun>
```

returning a stream of characters which are read from the channel argument. The end of stream will coincide with the end of the file associated to the channel.

In the same way, one can build the character stream associated to a character string using:

```
#stream_of_string;;
- : string -> char stream = <fun>

#let s = stream_of_string "abc";;
s : char stream = <abstract>
```

11.1.2 Streams are lazily evaluated

Stream expressions are submitted to *lazy evaluation*, i.e. they are effectively build only when required. This is useful in that it allows for the easy manipulation of "interactive" streams like the stream built from the standard input. If this was not the case, i.e. if streams were immediately completely computed, a program evaluating "`stream_of_channel std_in`" would read everything up to an end-of-file on standard input before giving control to the rest of the program. Furthermore, lazy evaluation of streams allows for the manipulation of infinite streams. As an example, we can build the infinite stream of integers, using side effects to show precisely when computations occur:

```
#let rec ints_from n =
#   [< '(print_int n; print_char ' '; flush std_out; n);
#      ints_from (n+1) >];;
ints_from : int -> int stream = <fun>

#let ints = ints_from 0;;
ints : int stream = <abstract>
```

We notice that no printing occurred and that the program terminates: this shows that none of the elements have been evaluated and that the infinite stream has not been built. We will see in the next section that these side-effects will occur on demand, i.e. when tests will be needed by a matching function on streams.

11.2 Stream matching and parsers

The syntax for building streams can be used for pattern-matching over them. However, stream matching is more complex than the usual pattern matching.

11.2.1 Stream matching is destructive

Let us start with a simple example:

```
#let next = function [< 'x >] -> x;;
next : 'a stream -> 'a = <fun>
```

The `next` function returns the first element of its stream argument, and fails if the stream is empty:

```
#let s = [< '0; '1; '2 >];;
s : int stream = <abstract>

#next s;;
- : int = 0

#next s;;
- : int = 1

#next s;;
- : int = 2

#next s;;
Uncaught exception: Parse_failure
```

We can see from the previous examples that the stream pattern `[< 'x >]` matches *an initial segment* of the stream. Such a pattern must be read as "the stream whose first element matches `x`". Furthermore, once stream matching has succeeded, the stream argument has been *physically modified* and does not contain any longer the part that has been recognized by the `next` function.

If we come back to the infinite stream of integers, we can see that the calls to `next` provoke the evaluation of the necessary part of the stream:

```
#next ints; next ints; next ints;;
0 1 2 - : int = 2
```

Thus, successive calls to `next` remove the first elements of the stream until it becomes empty. Then, `next` fails when applied to the empty stream, since, in the definition of `next`, there is no stream pattern that matches an initial segment of the empty stream.

It is of course possible to specify several stream patterns as in:

```
#let next = function
#  [< 'x >] -> x
#| [< >] -> raise (Failure "empty");;
next : 'a stream -> 'a = <fun>
```

Cases are tried in turn, from top to bottom.

Stream pattern components are not restricted to quoted patterns (intended to match stream elements), but can be also function calls (corresponding to non-terminals, in the grammar terminology). Functions appearing as stream pattern components are intended to match substreams of the stream argument: they are called on the actual stream argument, and they are followed by a pattern which should match the result of this call. For example, if we define a parser recognizing a non empty sequence of characters `'a'`:

```
#let seq_a =
#    let rec seq = function
#        [< ''a'; seq l >] -> 'a'::l
#      | [< >] -> []
#    in function [< ''a'; seq l >] -> 'a'::l;;
seq_a : char stream -> char list = <fun>
```

we used the recursively defined function `seq` inside the stream pattern of the first rule. This definition should be read as:

- if the stream is not empty and if its first element matches `'a'`, apply `seq` to the rest of the stream, let `l` be the result of this call and return `'a'::l`,
- otherwise, fail (raise `Parse_failure`);

and `seq` should be read in the same way (except that, since it recognizes possibly empty sequences of `'a'`, it never fails).

Less operationally, we can read it as: "a non-empty sequence of `'a'` starts with an `'a'`, and is followed by a possibly empty sequence of `'a'`.

Another example is the recognition of a non-empty sequence of `'a'` followed by a `'b'`, or a `'b'` alone:

```
#let seq_a_b = function
#  [< seq_a l; ''b' >] -> l@['b']
#| [< ''b' >] -> ['b'];;
seq_a_b : char stream -> char list = <fun>
```

Here, operationally, once an `'a'` has been recognized, the first matching rule is chosen. Any further mismatch (either from `seq_a` or from the last `'b'`) will raise a `Parse_error` exception, and the whole parsing will fail. On the other hand, if the first character is not an `'a'`, `seq_a` will raise `Parse_failure`, and the second rule (`[< ''b' >] -> ...`) will be tried.

This behavior is typical of predictive parsers. Predictive parsing is recursive-descent parsing with one look-ahead token. In other words, a predictive parser is a set of (possibly mutually recursive) procedures, which are selected according to the shape of (at most) the first token.

11.2.2 Sequential binding in stream patterns

Bindings in stream patterns occur sequentially, in contrast with bindings in regular patterns, which can be thought as occurring in parallel. Stream matching is guaranteed to be performed from left to right. For example, computing the sum of the elements of an integer stream could be defined as:

```
#let rec stream_sum n = function
#  [< '0; (stream_sum n) p >] -> p
#| [< 'x; (stream_sum (n+x)) p >] -> p
#| [< >] -> n;;
stream_sum : int -> int stream -> int = <fun>

#stream_sum 0 [< '0; '1; '2; '3; '4 >];;
- : int = 10
```

The `stream_sum` function uses its first argument as an accumulator holding the sum computed so far. The call `(stream_sum (n+x))` uses `x` which was bound in the stream pattern component occurring at the left of the call.

Warning: streams patterns are legal only in the `function` and `match` constructs. The `let` and other forms are restricted to usual patterns. Furthermore, a stream pattern cannot appear inside another pattern.

11.3 Parameterized parsers

Since a parser is a function like any other function, it can be parameterized or be used as a parameter. Parameters used only in the right-hand side of stream-matching rules simulate *inherited attributes* of attribute grammars. Parameters used as parsers in stream patterns allow for the implementation of *higher-order* parsers. We will use the next example to motivate the introduction of parameterized parsers.

11.3.1 Example: a parser for arithmetic expressions

Before building a parser for arithmetic expressions, we need a lexical analyzer able to recognize arithmetic operations and integer constants. Let us first define a type for tokens:

```
#type token =
#  PLUS | MINUS | TIMES | DIV | LPAR | RPAR
#| INT of int;;
Type token defined.
```

Skipping blank spaces is performed by the `spaces` function defined as:

```
#let rec spaces = function
#  [< ' ' '|'\t'|'\n'; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

The conversion of a digit (character) into its integer value is done by:

```
#let int_of_digit = function
#  '0'..'9' as c -> (int_of_char c) - (int_of_char '0')
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>
```

The "as" keyword allows for naming a pattern: in this case, the variable c will be bound to the actual digit matched by '0'..'9'. Pattern built with as are called *alias patterns*.

For the recognition of integers, we already feel the need for a parameterized parser. Integer recognition is done by the integer analyzer defined below. It is parameterized by a numeric value representing the value of the first digits of the number:

```
#let rec integer n = function
#  [< ' '0'..'9' as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> n;;
integer : int -> char stream -> int = <fun>

#integer 0 (stream_of_string "12345");;
- : int = 12345
```

We are now ready to write the lexical analyzer, taking a stream of characters, and returning a stream of tokens. Returning a token stream which will be explored by the parser is a simple, reasonably efficient and intuitive way of composing a lexical analyzer and a parser.

```
#let rec lexer s = match s with
#  [< ''('; spaces _ >] -> [< 'LPAR; lexer s >]
#| [< '')'; spaces _ >] -> [< 'RPAR; lexer s >]
#| [< ''+'; spaces _ >] -> [< 'PLUS; lexer s >]
#| [< ''-'; spaces _ >] -> [< 'MINUS; lexer s >]
#| [< ''*'; spaces _ >] -> [< 'TIMES; lexer s >]
#| [< ''/'; spaces _ >] -> [< 'DIV; lexer s >]
#| [< ''0'..'9' as c; (integer (int_of_digit c)) n; spaces _ >]
#                       -> [< 'INT n; lexer s >];;
lexer : char stream -> token stream = <fun>
```

We assume there is no leading space in the input.

Now, let us examine the language that we want to recognize. We shall have integers, infix arithmetic operations and parenthesized expressions. The BNF form of the grammar is:

```
Expr ::= Expr + Expr
       | Expr - Expr
       | Expr * Expr
       | Expr / Expr
       | ( Expr )
       | INT
```

The values computed by the parser will be *abstract syntax trees* (by contrast with *concrete syntax*, which is the input string or stream). Such trees belong to the following type:

```
#type atree =
#  Int of int
#| Plus of atree * atree
#| Minus of atree * atree
#| Mult of atree * atree
#| Div of atree * atree;;
Type atree defined.
```

The `Expr` grammar is ambiguous. To make it unambiguous, we will adopt the usual precedences for arithmetic operators and assume that all operators associate to the left. Now, to use stream matching for parsing, we must take into account the fact that matching rules are chosen according to the behavior of the first component of each matching rule. This is predictive parsing, and, using well-known techniques, it is easy to rewrite the grammar above in such a way that writing the corresponding predictive parser becomes trivial. These techniques are described in [2], and consist in adding a non-terminal for each precedence level, and removing left-recursion. We obtain:

```
Expr ::= Mult
       | Mult + Expr
       | Mult - Expr

Mult ::= Atom
       | Atom * Mult
       | Atom / Mult

Atom ::= INT
       | ( Expr )
```

While removing left-recursion, we forgot about left associativity of operators. This is not a problem, as long as we build correct abstract trees.

Since stream matching bases its choices on the first component of stream patterns, we cannot see the grammar above as a parser. We need a further transformation, factoring common prefixes of grammar rules (left-factor). We obtain:

```
Expr ::= Mult RestExpr

        RestExpr ::= + Mult RestExpr
                   | - Mult RestExpr
                   | (* nothing *)

Mult ::= Atom RestMult

        RestMult ::= * Atom RestMult
                   | / Atom RestMult
                   | (* nothing *)

Atom ::= INT
       | ( Expr )
```

Now, we can see this grammar as a parser (note that the order of rules becomes important, and empty productions must appear last). The shape of the parser is:

```
let rec expr =
    let rec restexpr ? = function
        [< 'PLUS; mult ?; restexpr ? >] -> ?
      | [< 'MINUS; mult ?; restexpr ? >] -> ?
```

```
      | [< >] -> ?
in function [< mult e1; restexpr ? >] -> ?

and mult =
    let rec restmult ? = function
        [< 'TIMES; atom ?; restmult ? >] -> ?
      | [< 'DIV; atom ?; restmult ? >] -> ?
      | [< >] -> ?
in function [< atom e1; restmult ? >] -> ?

and atom = function
  [< 'INT n >] -> Int n
| [< 'LPAR; expr e; 'RPAR >] -> e
```

We used question marks where parameters, bindings and results still have to appear. Let us consider the `expr` function: clearly, as soon as `e1` is recognized, we must be ready to build the leftmost subtree of the result. This leftmost subtree is either restricted to `e1` itself, in case `restexpr` does not encounter any operator, or it is the tree representing the addition (or subtraction) of `e1` and the expression immediately following the additive operator. Therefore, `restexpr` must be called with `e1` as an intermediate result, and accumulate subtrees built from its intermediate result, the tree constructor corresponding to the operator and the last expression encountered. The body of `expr` becomes:

```
let rec expr =
    let rec restexpr e1 = function
        [< 'PLUS; mult e2; restexpr (Plus (e1,e2)) e >] -> e
      | [< 'MINUS; mult e2; restexpr (Minus (e1,e2)) e >] -> e
      | [< >] -> e1
in function [< mult e1; (restexpr e1) e2 >] -> e2
```

Now, `expr` recognizes a product `e1` (by `mult`), and applies `(restexpr e1)` to the rest of the stream. According to the additive operator encountered (if any), this function will apply `mult` which will return some `e2`. Then the process continues with `Plus(e1,e2)` as intermediate result. In the end, a correctly balanced tree will be produced (using the last rule of `restexpr`).

With the same considerations on `mult` and `restmult`, we can complete the parser, obtaining:

```
#let rec expr =
#    let rec restexpr e1 = function
#        [< 'PLUS; mult e2; (restexpr (Plus (e1,e2))) e >] -> e
#      | [< 'MINUS; mult e2; (restexpr (Minus (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< mult e1; (restexpr e1) e2 >] -> e2
#
#and mult =
#    let rec restmult e1 = function
#        [< 'TIMES; atom e2; (restmult (Mult (e1,e2))) e >] -> e
#      | [< 'DIV; atom e2; (restmult (Div (e1,e2))) e >] -> e
```

```
#      | [< >] -> e1
#in function [< atom e1; (restmult e1) e2 >] -> e2
#
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

And we can now try our parser:

```
#expr (lexer (stream_of_string "(1+2+3*4)-567"));;
- : atree = Minus (Plus (Plus (Int 1, Int 2), Mult (Int 3, Int 4)), Int 567)
```

11.3.2 Parameters simulating inherited attributes

In the previous example, the parsers `restexpr` and `restmult` take an abstract syntax tree `e1` as argument and pass it down to the result through recursive calls such as `(restexpr (Plus(e1,e2)))`. If we see such parsers as non-terminals (`RestExpr` from the grammar above) this parameter acts as an inherited attribute of the non-terminal. Synthesized attributes are simulated by the right hand sides of stream matching rules.

11.3.3 Higher-order parsers

In the definition of `expr`, we may notice that the parsers `expr` and `mult` on the one hand and `restexpr` and `restmult` on the other hand have exactly the same structure. To emphasize this similarity, if we define parsers for additive (resp. multiplicative) operators by:

```
#let addop = function
#  [< 'PLUS >] -> (function (x,y) -> Plus(x,y))
#| [< 'MINUS >] -> (function (x,y) -> Minus(x,y))
#and multop = function
#  [< 'TIMES >] -> (function (x,y) -> Mult(x,y))
#| [< 'DIV >] -> (function (x,y) -> Div(x,y));;
addop : token stream -> atree * atree -> atree = <fun>
multop : token stream -> atree * atree -> atree = <fun>
```

we can rewrite the `expr` parser as:

```
#let rec expr =
#    let rec restexpr e1 = function
#        [< addop f; mult e2; (restexpr (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< mult e1; (restexpr e1) e2 >] -> e2
#
#and mult =
```

```
#    let rec restmult e1 = function
#        [< multop f; atom e2; (restmult (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#in function [< atom e1; (restmult e1) e2 >] -> e2
#
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

Now, we take advantage of these similarities in order to define a general parser for left-associative operators. Its name is `left_assoc` and is parameterized by a parser for operators and a parser for expressions:

```
#let rec left_assoc op term =
#    let rec rest e1 = function
#        [< op f; term e2; (rest (f (e1,e2))) e >] -> e
#      | [< >] -> e1
#    in function [< term e1; (rest e1) e2 >] -> e2;;
left_assoc : ('a stream -> 'b * 'b -> 'b) -> ('a stream -> 'b) -> 'a stream -> '
b = <fun>
```

Now, we can redefine `expr` as:

```
#let rec expr str = left_assoc addop mult str
#and mult str = left_assoc multop atom str
#and atom = function
#  [< 'INT n >] -> Int n
#| [< 'LPAR; expr e; 'RPAR >] -> e;;
expr : token stream -> atree = <fun>
mult : token stream -> atree = <fun>
atom : token stream -> atree = <fun>
```

And we can now try our definitive parser:

```
#expr (lexer (stream_of_string "(1+2+3*4)-567"));;
- : atree = Minus (Plus (Plus (Int 1, Int 2), Mult (Int 3, Int 4)), Int 567)
```

Parameterized parsers are useful for defining general parsers such as `left_assoc` that can be used with different instances. Another example of a useful general parser is the `star` parser defined as:

```
#let rec star p = function
#  [< p x; (star p) l >] -> x::l
#| [< >] -> [];;
star : ('a stream -> 'b) -> 'a stream -> 'b list = <fun>
```

The `star` parser iterates zero or more times its argument `p` and returns the list of results. We still have to be careful in using these general parsers because of the predictive nature of parsing. For example, `star p` will never successfully terminate if `p` has a rule for the empty stream pattern: this rule will make the second rule of `star` useless!

11.3.4 Example: parsing a non context-free language

As an example of parsing with parameterized parsers, we shall build a parser for the language $\{wCw \mid w \in (A|B)^*\}$, which is known to be non context-free.

First, let us define a type for this alphabet:

```
#type token = A | B | C;;
Type token defined.
```

Given an input of the form w`C`w, the idea for a parser recognizing it is:

- first, to recognize the sequence w with a parser `wd` (for *word definition*) returning information in order to build a parser recognizing only w;
- then to recognize `C`;
- and to use the parser built at the first step to recognize the sequence w.

The definition of `wd` is as follows:

```
#let rec wd = function
#  [< 'A; wd l >] -> (function [< 'A >] -> "a")::l
#| [< 'B; wd l >] -> (function [< 'B >] -> "b")::l
#| [< >] -> [];;
wd : token stream -> (token stream -> string) list = <fun>
```

The `wu` function (for *word usage*) builds a parser sequencing a list of parsers:

```
#let rec wu = function
#  p::pl -> (function [< p x; (wu pl) l >] -> x^l)
#| [] -> (function [< >] -> "");;
wu : ('a stream -> string) list -> 'a stream -> string = <fun>
```

The `wu` function builds, from a list of parsers p_i, for $i = 1..n$, a single parser

$$\texttt{function [<}p_1\ x_1\texttt{;}\ldots\texttt{;}p_n\ x_n\texttt{>] -> [}x_1\texttt{;}\ldots\texttt{;}x_n\texttt{]}$$

which is the sequencing of parsers p_i. The main parser `w` is:

```
#let w = function [< wd l; 'C; (wu l) r >] -> r;;
w : token stream -> string = <fun>

#w [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"

#w [< 'C >];;
- : string = ""
```

In the previous parser, we used an intermediate list of parsers in order to build the second parser. We can redefine `wd` without using such a list:

```
#let w =
#    let rec wd wr = function
#        [< 'A; (wd (function [< wr r; 'A >] -> r^"a")) p >] -> p
#      | [< 'B; (wd (function [< wr r; 'B >] -> r^"b")) p >] -> p
#      | [< >] -> wr
#    in function [< (wd (function [< >] -> "")) p; 'C; p str >] -> str;;
w : token stream -> string = <fun>

#w [< 'A; 'B; 'B; 'C; 'A; 'B; 'B >];;
- : string = "abb"

#w [< 'C >];;
- : string = ""
```

Here, `wd` is made local to `w`, and takes as parameter `wr` (for *word recognizer*) whose initial value is the parser with an empty stream pattern. This parameter accumulates intermediate results, and is delivered at the end of parsing the initial sequence w. After checking for the presence of `C`, it is used to parse the second sequence w.

11.4 Further reading

A summary of the constructs over streams is given 24.1. Primitives on streams are described in section 21.14.

An alternative to parsing with streams and stream matching are the `camllex` and `camlyacc` programs. They are described in chapter 29.

A detailed presentation of streams and stream matching following "predictive parsing" semantics can be found in [23], where alternative semantics are given with some possible implementations.

Exercises

11.1 *Define a parser for the language of prefix arithmetic expressions generated by the grammar:*

```
Expr ::= INT
       | + Expr Expr
       | - Expr Expr
       | * Expr Expr
       | / Expr Expr
```

Use the lexical analyzer for arithmetic expressions given above. The result of the parser must be the integer resulting from the evaluation of the arithmetic expression, i.e. its type must be:

```
token -> int
```

11.2 *Enrich the type* `token` *above with a constructor* `IDENT of string` *for identifiers, and enrich the lexical analyzer for it to recognize identifiers built from alphabetic letters (upper or lowercase). Length of identifiers may be limited. A look at primitives on strings is necessary (section 21.15).*

Chapter 12

Standalone programs and separate compilation

So far, we have used Caml Light in an interactive way. It is also possible to program in Caml Light in a batch-oriented way: writing source code in a file, having it compiled into an executable program, and executing the program outside of the Caml Light environment. Interactive use is great for learning the language and quickly testing new functions. Batch use is more convenient to develop larger programs, that should be usable without knowledge of Caml Light.

Mac: Batch compilation is not available in the standalone Caml Light application. It requires the MPW environment (see chapter 1).

12.1 Standalone programs

Standalone programs are composed of a sequence of phrases, contained in one or several text files. Phrases are the same as at toplevel: expressions, value declarations, type declarations, exception declarations, and directives. When executing the stand-alone program produced by the compiler, all phrases are executed in order. The values of expressions and declared global variables are not printed, however. A stand-alone program has to perform input and output explicitly.

Here is a sample program, that prints the number of characters and the number of lines of its standard input, like the `wc` Unix utility.

```
let chars = ref 0;;
let lines = ref 0;;
try
  while true do
    let c = input_char std_in in
      chars := !chars + 1;
      if c = `\n` then lines := !lines + 1 else ()
  done
with End_of_file ->
  print_int !chars; print_string " characters, ";
  print_int !lines; print_string " lines.\n";
```

```
  exit 0
;;
```

The `input_char` function reads the next character from an input channel (here, `std_in`, the channel connected to standard input). It raises exception `End_of_file` when reaching the end of the file. The `exit` function aborts the process. Its argument is the exit status of the process. Calling `exit` is absolutely necessary to ensure proper flushing of the output channels.

Assume this program is in file `count.ml`. To compile it, simply run the `camlc` command from the command interpreter:

```
camlc -o count count.ml
```

The compiler produces an executable file `count`. You can now run `count` with the help of the "camlrun" command:

```
camlrun count < count.ml
```

This should display something like:

```
306 characters, 13 lines.
```

Under Unix, the `count` file can actually be executed directly, just like any other Unix command, as in:

```
./count < count.ml
```

This also works under MS-DOS, provided the executable file is given extension `.exe`. That is, if we compile `count.ml` as follows:

```
camlc -o count.exe count.ml
```

we can run `count.exe` directly, as in:

```
count.exe < count.ml
```

See chapter 25 for more information on `camlc`.

12.2 Programs in several files

It is possible to split one program into several source files, separately compiled. This way, local changes do not imply a full recompilation of the program. Let us illustrate that on the previous example. We split it in two modules: one that implements integer counters; another that performs the actual counting. Here is the first one, `counter.ml`:

```
(* counter.ml *)
type counter = { mutable val: int };;
let new init = { val = init };;
let incr c = c.val <- c.val + 1;;
let read c = c.val;;
```

Here is the source for the main program, in file `main.ml`.

```
(* main.ml *)
let chars = counter__new 0;;
let lines = counter__new 0;;
try
  while true do
    let c = input_char std_in in
      counter__incr chars;
      if c = '\n' then counter__incr lines else ()
  done
with End_of_file ->
  print_int (counter__read chars); print_string " characters, ";
  print_int (counter__read lines); print_string " lines.\n";
  exit 0
;;
```

Notice that references to identifiers defined in module `counter.ml` are prefixed with the name of the module, `counter`, and by `__` (the "long dash" symbol: two underscore characters). If we had simply entered `new 0`, for instance, the compiler would have assumed `new` is an identifier declared in the current module, and issued an "undefined identifier" error.

Compiling this program requires two compilation steps, plus one final linking step.

```
camlc -c counter.ml
camlc -c main.ml
camlc -o main counter.zo main.zo
```

Running the program is done as before:

```
camlrun main < counter.ml
```

The `-c` option to `camlc` means "compile only"; that is, the compiler should not attempt to produce a stand-alone executable program from the given file, but simply an object code file (files `counter.zo`, `main.zo`). The final linking phases takes the two `.zo` files and produces the executable `main`. Object files must be linked in the right order: for each global identifier, the module defining it must come before the modules that use it.

Prefixing all external identifiers by the name of their defining module is sometimes tedious. Therefore, the Caml Light compiler provides a mechanism to omit the `module__` part in external identifiers. The system maintains a list of "default" modules, called the currently opened modules, and whenever it encounters an identifier without the `module__` part, it searches through the opened modules, to find one that defines this identifier. Searched modules always include the module being compiled (searched first), and some library modules of general use. In addition, two directives are provided to add and to remove modules from the list of opened modules:

- `#open "module";;` to add `module` in front of the list;
- `#close "module";;` to remove `module` from the list.

For instance, we can rewrite the `main.ml` file above as:

```
#open "counter";;
let chars = new 0;;
let lines = new 0;;
try
  while true do
    let c = input_char std_in in
      incr chars;
      if c = `\n` then incr lines
  done
with End_of_file ->
  print_int (read chars);
  print_string " characters, ";
  print_int (read lines);
  print_string " lines.\n";
  exit 0
;;
```

After the `#open "counter"` directive, the identifier `new` automatically resolves to `counters__new`.

If two modules, say `mod1` and `mod2`, define both a global value `f`, then `f` in a client module `client` resolves to `mod1__f` if `mod1` is opened but not `mod2`, or if `mod1` has been opened more recently than `mod2`. Otherwise, it resolves to `mod2__f`. For instance, the two system modules `int` and `float` both define the infix identifier `+`. Both modules `int` and `float` are opened by default, but `int` comes first. Hence, `x + y` is understood as the integer addition, since `+` is resolved to `int__+`. But after the directive `#open "float";;`, module `float` comes first, and the identifier `+` is resolved to `float__+`.

12.3 Abstraction

Some globals defined in a module are not intended to be used outside of this module. Then, it is good programming style not to export them outside of the module, so that the compiler can check they are not used in another module. Also, one may wish to export a data type abstractly, that is, without publicizing the structure of the type. This ensures that other modules cannot build or inspect objects of that type without going through one of the functions on that type exported in the defining module. This helps in writing clean, well-structured programs.

The way to do that in Caml Light is to write an explicit interface, or output signature, specifying those identifiers that are visible from the outside. All other identifiers will remain local to the module. For global values, their types must be given by hand. The interface is contained in a file whose name is the module name, with extension `.mli`.

Here is for instance an interface for the `counter` module, that abstracts the type `counter`:

```
(* counter.mli *)
type counter;;        (* an abstract type *)
value new : int -> counter
  and incr : counter -> unit
  and read : counter -> int;;
```

Interfaces must be compiled separately. However, once the interface for module A has been compiled, any module B that uses A can be immediately compiled, even if the implementation of A is not yet compiled or even not yet written. Consider:

```
camlc -c counter.mli
camlc -c main.ml
camlc -c counter.ml
camlc -o main counter.zo main.zo
```

The implementation `main.ml` could be compiled before `counter.ml`. The only requirement for compiling `main.ml` is the existence of `counter.zi`, the compiled interface of the `counter` module.

Exercises

12.1 *Complete the* `count` *command: it should be able to operate on several files, given on the command line. Hint:* `sys__command_line` *is an array of strings, containing the command-line arguments to the process.*

Part C
A complete example

Chapter 13

ASL: A Small Language

We present in this chapter a simple language: ASL (A Small Language). This language is basically the λ-calculus (the purely functional kernel of Caml) enriched with a conditional construct. The conditional must be a special construct, because our language will be submitted to call-by-value: thus, the conditional cannot be a function.

ASL programs are built up from numbers, variables, functional expressions (λ-abstractions), applications and conditionals. An ASL program consists of a global declaration of an identifier getting bound to the value of an expression. The primitive functions that are available are equality between numbers and arithmetic binary operations. The concrete syntax of ASL expressions can be described (ambiguously) as:

```
Expr ::= INT
       | IDENT
       | "if" Expr "then" Expr "else" Expr "fi"
       | "(" Expr ")"
       | "\" IDENT "." Expr
```

and the syntax of declarations is given as:

```
Decl ::= "let" IDENT "be" Expr ";"
       | Expr ";"
```

Arithmetic binary operations will be written in prefix position and will belong to the class `IDENT`. The `\` symbol will play the role of the Caml keyword `function`.

We start by defining the abstract syntax of ASL expressions and of ASL toplevel phrases. Then we define a parser in order to produce abstract syntax trees from the concrete syntax of ASL programs.

13.1 ASL abstract syntax trees

We encode variable names by numbers. These numbers represent the *binding depth* of variables. For instance, the function of `x` returning `x` (the ASL identity function) will be represented as:

```
Abs("x", Var 1)
```

And the ASL application function which would be written in Caml:

```
(function f -> (function x -> f(x)))
```

would be represented as:

```
Abs("f", Abs("x", App(Var 2, Var 1)))
```

and should be viewed as the tree:

`Var n` should be read as "an occurrence of the variable bound by the `n`th abstraction node encountered when going toward the root of the abstract syntax tree". In our example, when going from `Var 2` to the root, the 2nd abstraction node we encounter introduces the `"f"` variable.

The numbers encoding variables in abstract syntax trees of functional expressions are called "De Bruijn[1] numbers". The characters that we attach to abstraction nodes simply serve as documentation: they will not be used by any of the semantic analyses that we will perform on the trees. The type of ASL abstract syntax trees is defined by:

```
#type asl = Const of int
#         | Var of int
#         | Cond of asl * asl * asl
#         | App of asl * asl
#         | Abs of string * asl
#
#and top_asl = Decl of string * asl;;
Type asl defined.
Type top_asl defined.
```

13.2 Parsing ASL programs

Now we come to the problem of defining a concrete syntax for ASL programs and declarations.

The choice of the concrete aspect of the programs is simply a matter of taste. The one we choose here is close to the syntax of λ-calculus (except that we will use the *backslash* character because there is no "λ" on our keyboards). We will use the *curried* versions of equality and arithmetic functions. We will also use a *prefix* notation (à la Lisp) for their application. We will write "`+ (+`

[1] They have been proposed by N.G. De Bruijn in [10] in order to facilitate the mechanical treatment of λ-calculus terms.

`1 2) 3`" instead of "`(1+2)+3`". The "`if` e_1 `then` e_2 `else` e_3" construct will be written "`if` e_1 `then` e_2 `else` e_3 `fi`", and will return the `then` part when e_1 is different from 0 (0 acts thus as falsity in ASL conditionals).

13.2.1 Lexical analysis

The concrete aspect of ASL programs will be either declarations of the form:

> `let` *identifier* `be` *expression*`;`

or:

> *expression*`;`

which will be understood as:

> `let it be` *expression*`;`

The tokens produced by the lexical analyzer will represent the keywords `let`, `be`, `if` and `else`, the `\` binder, the dot, parentheses, integers, identifiers, arithmetic operations and terminating semicolons. We reuse here most of the code that we developed in chapter 11 or in the answers to its exercises.

Skipping blank spaces:

```
#let rec spaces = function
#  [< '' '|'\t'|'\n'; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

The type of tokens is given by:

```
#type token = LET | BE | LAMBDA | DOT | LPAR | RPAR
#           | IF | THEN | ELSE | FI | SEMIC
#           | INT of int | IDENT of string;;
Type token defined.
```

Integers:

```
#let int_of_digit = function
#  '0'..'9' as c -> (int_of_char c) - (int_of_char '0')
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>
```

```
#let rec integer n = function
#  [< ' '0'..'9' as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> INT n;;
integer : int -> char stream -> token = <fun>
```

We restrict ASL identifiers to be composed of lowercase letters, the eight first being significative. An explanation about the `ident` function can be found in the chapter dedicated to the answers to exercises (chapter 18). The function given here is slightly different and tests its result in order to see wether it is a keyword (`let`, `be`, ...) or not:

```
#let ident_buf = make_string 8 ' ';;
ident_buf : string = "        "

#let rec ident len = function
#  [< ' 'a'..'z' as c;
#     (if len >= 8 then ident len
#      else begin
#            set_nth_char ident_buf len c;
#            ident (succ len)
#           end) s >] -> s
#| [< >] -> (match sub_string ident_buf 0 len
#            with "let" -> LET
#               | "be" -> BE
#               | "if" -> IF
#               | "then" -> THEN
#               | "else" -> ELSE
#               | "fi" -> FI
#               | s -> IDENT s);;
ident : int -> char stream -> token = <fun>
```

A reasonable lexical analyzer would use a hash table (see section 22.3, part IV) to recognize keywords faster.

Primitive operations are recognized by the following function, which also detects illegal operators and ends of input:

```
#let oper = function
#  [< ''+'|'-'|'*'|'/'|'=' as c >] -> IDENT(make_string 1 c)
#| [< 'c >] -> prerr_string "Illegal character: ";
#              prerr_endline (char_for_read c);
#              raise (Failure "ASL parsing")
#| [< >] -> prerr_endline "Unexpected end of input";
#           raise (Failure "ASL parsing");;
oper : char stream -> token = <fun>
```

The lexical analyzer has the same structure as the one given in chapter 11 except that leading blanks are skipped.

```
#let rec lexer str = spaces str;
#match str with
#  [< ''('; spaces _ >]  -> [< 'LPAR; lexer str >]
#| [< ')'; spaces _ >]  -> [< 'RPAR; lexer str >]
#| [< ''\\'; spaces _ >] -> [< 'LAMBDA; lexer str >]
#| [< ''.'; spaces _ >]  -> [< 'DOT; lexer str >]
#| [< ';'; spaces _ >]  -> [< 'SEMIC; lexer str >]
#| [< ''0'..'9' as c;
#     (integer (int_of_digit c)) tok;
#      spaces _ >]       -> [< 'tok; lexer str >]
```

```
#| [< ''a'..'z' as c;
#     (set_nth_char ident_buf 0 c; ident 1) tok;
#     spaces _ >]        -> [< 'tok; lexer str >]
#| [< oper tok; spaces _ >] -> [< 'tok; lexer str >]
#;;
lexer : char stream -> token stream = <fun>
```

The lexical analyzer returns a stream of tokens that the parser will receive as argument.

13.2.2 Parsing

The final output of our parser will be abstract syntax trees of type `asl` or `top_asl`. This implies that we will detect unbound identifiers at parse-time. In this case, we will raise the `Unbound` exception defined as:

```
#exception Unbound of string;;
Exception Unbound defined.
```

We also need a function which will compute the binding depths of variables. That function simply looks for the position of the first occurrence of a variable name in a list. It will raise `Unbound` if there is no such occurrence.

```
#let binding_depth s rho =
#  bind 1 rho
#  where rec bind n = function
#      []  -> raise (Unbound s)
#  | t::l -> if s = t then Var n else bind (n+1) l;;
binding_depth : string -> string list -> asl = <fun>
```

We also need a global environment, containing names of already bound identifiers. The global environment contains predefined names for the equality and arithmetic functions. We represent the global environment as a reference since each ASL declaration will augment it with a new name.

```
#let init_env =  ["+";"-";"*";"/";"="];;
init_env : string list = ["+"; "-"; "*"; "/"; "="]

#let global_env = ref init_env;;
global_env : string list ref = ref ["+"; "-"; "*"; "/"; "="]
```

We now give a parsing function for ASL programs. Blanks at the beginning of the string are skipped.

```
#let rec top = function
#    [< 'LET; 'IDENT id; 'BE; expression e; 'SEMIC >] -> Decl(id,e)
#  | [< expression e; 'SEMIC >] -> Decl("it",e)
#
#and expression = function
#    [< (expr !global_env) e >] -> e
#
```

```
#and expr rho =
#  let rec rest e1 = function
#          [< (atom rho) e2; (rest (App(e1,e2))) e >] -> e
#        | [< >] -> e1
#  in function
#       [< 'LAMBDA; 'IDENT id; 'DOT; (expr (id::rho)) e >] -> Abs(id,e)
#     | [< (atom rho) e1; (rest e1) e2 >] -> e2
#
#and atom rho = function
#    [< 'IDENT id >] ->
#        (try binding_depth id rho with Unbound s ->
#              print_string "Unbound ASL identifier: ";
#              print_string s; print_newline();
#              raise (Failure "ASL parsing"))
#  | [< 'INT n >] -> Const n
#  | [< 'IF; (expr rho) e1; 'THEN; (expr rho) e2;
#       'ELSE; (expr rho) e3; 'FI >] -> Cond(e1,e2,e3)
#  | [< 'LPAR; (expr rho) e; 'RPAR >] -> e;;
top : token stream -> top_asl = <fun>
expression : token stream -> asl = <fun>
expr : string list -> token stream -> asl = <fun>
atom : string list -> token stream -> asl = <fun>
```

The complete parser that we will use reads a string, converts it into a stream, and produces the token stream that is parsed:

```
#let parse_top s = top(lexer(stream_of_string s));;
parse_top : string -> top_asl = <fun>
```

Let us try our grammar (we do not augment the global environment at each declaration: this will be performed after the semantic treatment of ASL programs). We need to write double \ inside strings, since \ is the string escape character.

```
#parse_top "let f be \\x.x;";;
- : top_asl = Decl ("f", Abs ("x", Var 1))

#parse_top "let x be + 1 ((\\x.x) 2);";;
- : top_asl = Decl ("x", App (App (Var 1, Const 1), App (Abs ("x", Var 1), Const
 2)))
```

Unbound identifiers and undefined operators are correctly detected:

```
#parse_top "let y be g 3;";;
Unbound ASL identifier: g
Uncaught exception: Failure "ASL parsing"

#parse_top "f (if 0 then + else - fi) 2 3;";;
Unbound ASL identifier: f
Uncaught exception: Failure "ASL parsing"
```

```
#parse_top "^ x y;";;
Illegal character: ^
Uncaught exception: Failure "ASL parsing"
```

Chapter 14

Untyped semantics of ASL programs

In this section, we give a semantic treatment of ASL programs. We will use *dynamic typechecking*, i.e. we will test the type correctness of programs during their interpretation.

14.1 Semantic values

We need a type for ASL semantic values (representing results of computations). A semantic value will be either an integer, or a Caml functional value from ASL values to ASL values.

```
#type semval = Constval of int
#            | Funval of (semval -> semval);;
Type semval defined.
```

We now define two exceptions. The first one will be used when we encounter an ill-typed program and will represent run-time type errors. The other one is helpful for debugging: it will be raised when our interpreter (semantic function) goes into an illegal situation.

The following two exceptions will be raised in case of run-time ASL type error, and in case of bug of our semantic treatment:

```
#exception Illtyped;;
Exception Illtyped defined.

#exception SemantBug of string;;
Exception SemantBug defined.
```

We must give a semantic value to our basic functions (equality and arithmetic operations). The next function transforms a Caml function into an ASL value.

```
#let init_semantics caml_fun =
#    Funval
#      (function Constval n ->
#         Funval(function Constval m -> Constval(caml_fun n m)
#                       | _ -> raise Illtyped)
#              | _ -> raise Illtyped);;
init_semantics : (int -> int -> int) -> semval = <fun>
```

Now, associate a Caml Light function to each ASL predefined function:

```
#let caml_function = function
#    "+" -> prefix +
#  | "-" -> prefix -
#  | "*" -> prefix *
#  | "/" -> prefix /
#  | "=" -> (fun n m -> if n=m then 1 else 0)
#  | s -> raise (SemantBug "Unknown primitive");;
caml_function : string -> int -> int -> int = <fun>
```

In the same way as, for parsing, we needed a global environment from which the binding depth of identifiers was computed, we need a semantic environment from which the interpreter will fetch the value represented by identifiers. The global semantic environment will be a reference on the list of predefined ASL values.

```
#let init_sem =  map (fun x -> init_semantics(caml_function x))
#                    init_env;;
init_sem : semval list = [Funval <fun>; Funval <fun>; Funval <fun>; Funval <fun>
; Funval <fun>]

#let global_sem = ref init_sem;;
global_sem : semval list ref = ref [Funval <fun>; Funval <fun>; Funval <fun>; Fu
nval <fun>; Funval <fun>]
```

14.2 Semantic functions

The semantic function is the interpreter itself. There is one for expressions and one for declarations. The one for expressions computes the value of an ASL expression from an environment `rho`. The environment will contain the values of globally defined ASL values or of temporary ASL values. It is organized as a list, and the numbers representing variable occurrences will be used as indices into the environment.

```
#let rec nth n = function
#     []  -> raise (Failure "nth")
#  | x::l -> if n=1 then x else nth (n-1) l;;
nth : int -> 'a list -> 'a = <fun>

#let rec semant rho = sem
#    where rec sem = function
#      Const n -> Constval n
#    | Var(n) -> nth n rho
#    | Cond(e1,e2,e3) ->
#        (match sem e1 with Constval 0 -> sem e3
#                         | Constval n -> sem e2
#                         | _ -> raise Illtyped)
#    | Abs(_,e') -> Funval(fun x -> semant (x::rho) e')
```

```
#    | App(e1,e2) -> (match sem e1
#                        with Funval(f) -> f (sem e2)
#                           | _ -> raise Illtyped)
#;;
semant : semval list -> asl -> semval = <fun>
```

The main function must be able to treat an ASL declaration, evaluate it, and update the global environments (`global_env` and `global_sem`).

```
#let semantics = function Decl(s,e) ->
#    let result = semant !global_sem e
#    in global_env := s::!global_env;
#       global_sem := result::!global_sem;
#       print_string "ASL Value of ";
#       print_string s;
#       print_string " is ";
#       (match result with
#         Constval n -> print_int n
#       | Funval f -> print_string "<fun>");
#       print_newline();;
semantics : top_asl -> unit = <fun>
```

14.3 Examples

```
#semantics (parse_top "let f be \\x. + x 1;");;
ASL Value of f is <fun>
- : unit = ()

#semantics (parse_top "let i be \\x. x;");;
ASL Value of i is <fun>
- : unit = ()

#semantics (parse_top "let x be i (f 2);");;
ASL Value of x is 3
- : unit = ()

#semantics (parse_top "let y be if x then (\\x.x) else 2 fi 0;");;
ASL Value of y is 0
- : unit = ()
```

Chapter 15

Encoding recursion

15.1 Fixpoint combinators

We have seen that we do not have recursion in ASL. However, it is possible to encode recursion by defining a *fixpoint combinator*. A fixpoint combinator is a function F such that:

$$F\ M \text{ is equivalent to } M\ (F\ M) \text{ modulo the evaluation rules.}$$

for any expression M. A consequence of the equivalence given above is that fixpoint combinators can encode recursion. Let us note $M \equiv N$ if expressions M and N are equivalent modulo the evaluation rules. Then, consider `ffact` to be the functional obtained from the body of the factorial function by abstracting (i.e. using as a parameter) the `fact` identifier, and `fix` an arbitrary fixpoint combinator. We have:

```
ffact is \fact.(\n. if = n 0 then 1 else * n (fact (- n 1)) fi)
```

Now, let us consider the expression $E =$ (`fix ffact`) `3`. Using our intuition about the evaluation rules, and the definition of a fixpoint combinator, we obtain:
$E \equiv$ `ffact (fix ffact) 3`
Replacing `ffact` by its definition, we obtain:
$E \equiv$ `(\fact.(\n. if = n 0 then 1 else * n (fact (- n 1)) fi)) (fix ffact) 3`
We can now pass the two arguments to the first abstraction, instantiating `fact` and `n` respectively to `fix ffact` and `3`:
$E \equiv$ `if = 3 0 then 1 else * 3 (fix ffact (- 3 1)) fi`
We can now reduce the conditional into its `else` branch:
$E \equiv$ `* 3 (fix ffact (- 3 1))`
Continuing this way, we eventually compute:
$E \equiv$ `* 3 (* 2 (* 1 1))` $\equiv$ `6`

This is the expected behavior of the factorial function. Given an appropriate fixpoint combinator `fix`, we could define the factorial function as `fix ffact`, where `ffact` is the expression above.

Unfortunately, when using call-by-value, any application of a fixpoint combinator F such that:

$$F\ M \text{ evaluates to } M\ (F\ M)$$

leads to non-termination of the evaluation (because evaluation of $(F\ \ M)$ leads to evaluating $(M\ (F\ M))$, and thus $(F\ M)$ again).

We will use the Z fixpoint combinator defined by:

$$Z = \lambda f.((\lambda x.\ f\ (\lambda y.\ (x\ x)\ y))(\lambda x.\ f\ (\lambda y.\ (x\ x)\ y)))$$

The fixpoint combinator Z has the particularity of being usable under call-by-value evaluation regime (in order to check that fact, it is necessary to know the evaluation rules of λ-calculus). Since the name `z` looks more like an ordinary parameter name, we will call `fix` the ASL expression corresponding to the Z fixpoint combinator.

```
#semantics (parse_top
#        "let fix be \\f.((\\x.f(\\y.(x x) y))(\\x.f(\\y.(x x) y)));");;
ASL Value of fix is <fun>
- : unit = ()
```

We are now able to define the ASL factorial function:

```
#semantics (parse_top
#        "let fact be fix (\\f.(\\n. if = n 0 then 1
#                                    else * n (f (- n 1)) fi));");;
ASL Value of fact is <fun>
- : unit = ()

#semantics (parse_top "fact 8;");;
ASL Value of it is 40320
- : unit = ()
```

and the ASL Fibonacci function:

```
#semantics (parse_top
#        "let fib be fix (\\f.(\\n. if = n 1 then 1
#                                   else if = n 2 then 1
#                                        else + (f (- n 1)) (f (- n 2)) fi fi));");;
ASL Value of fib is <fun>
- : unit = ()

#semantics (parse_top "fib 9;");;
ASL Value of it is 34
- : unit = ()
```

15.2 Recursion as a primitive construct

Of course, in a more realistic prototype, we would extend the concrete and abstract syntaxes of ASL in order to support recursion as a primitive construct. We do not do it here because we want to keep ASL simple. This is an interesting non trivial exercise!

Chapter 16

Static typing, polymorphism and type synthesis

We now want to perform static typechecking of ASL programs, that is, to complete typechecking *before* evaluation, making run-time type tests unnecessary during evaluation of ASL programs.

Furthermore, we want to have *polymorphism* (i.e. allow the identity function, for example, to be applicable to any kind of data).

Type synthesis may be seen as a game. When learning a game, we must:

- learn the rules (what is allowed, and what is forbidden);
- learn a winning strategy.

In type synthesis, the rules of the game are called a *type system*, and the winning strategy is the typechecking algorithm.

In the following sections, we give the ASL type system, the algorithm and an implementation of that algorithm. Most of this presentation is borrowed from [7].

16.1 The type system

We study in this section a type system for the ASL language. Then, we present an algorithm performing the type synthesis of ASL programs, and its Caml Light implementation. Because of subtle aspects of the notation used (inference rules), and since some important mathematical notions, such as unification of first-order terms, are not presented here, this chapter may seem obscure at first reading.

The type system we will consider for ASL has been first given by Milner [25] for a subset of the ML language (in fact, a superset of λ-calculus). A *type* is either:

- the type Number;
- or a type variable (α, β, ...);
- or $\tau_1 \rightarrow \tau_2$, where τ_1 and tau_2 are types.

In a type, a type variable is an *unknown*, i.e. a type that we are computing. We will use τ, τ', τ_1, ..., as *metavariables*[1] representing types. This notation is important: we shall use other greek letters to denote other notions in the following sections.

Example $(\alpha \to \text{Number}) \to \beta \to \beta$ is a type.
□

A *type scheme*, is a type where some variables are distinguished as being *generic*. We can represent type schemes by:

$$\forall \alpha_1, \ldots, \alpha_n.\tau \text{ where } \tau \text{ is a type.}$$

Example $\forall \alpha.(\alpha \to \text{Number}) \to \beta \to \beta$ and $(\alpha \to \text{Number}) \to \beta \to \beta$ are type schemes.
□

We will use σ, σ', σ_1, ..., as metavariables representing type schemes. We may also write type schemes as $\forall \vec{\alpha}.\tau$. In this case, $\vec{\alpha}$ represent a (possibly empty) set of generic type variables. When the set of generic variables is empty, we write $\forall.\tau$ or simply τ.

We will write $FV(\sigma)$ for the set of *unknowns* occurring in the type scheme σ. Unknowns are also called *free variables* (they are not bound by a $\forall$ quantifier).

We also write $BV(\sigma)$ (*bound type variables of* σ) for the set of type variables occurring in σ which are not free (i.e. the set of variables universally quantified). Bound type variables are also said to be *generic*.

Example If σ denotes the type scheme $\forall \alpha.(\alpha \to \text{Number}) \to \beta \to \beta$, then we have:

$$FV(\sigma) = \{\beta\}$$

and

$$BV(\sigma) = \{\alpha\}$$

□

A *substitution instance* σ' of a type scheme σ is the type scheme $S(\sigma)$ where S is a substitution of types for *free* type variables appearing in σ. When applying a substitution to a type scheme, a renaming of some bound type variables may become necessary, in order to avoid the capture of a free type variable by a quantifier.

Example

- If σ denotes $\forall \beta.(\beta \to \alpha) \to \alpha$ and σ' is $\forall \beta.(\beta \to (\gamma \to \gamma)) \to (\gamma \to \gamma)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\gamma \to \gamma)\}$, i.e. S substitutes the type $\gamma \to \gamma$ for the variable α.
- If σ denotes $\forall \beta.(\beta \to \alpha) \to \alpha$ and σ' is $\forall \delta.(\delta \to (\beta \to \beta)) \to (\beta \to \beta)$, then σ' is a substitution instance of σ because $\sigma' = S(\sigma)$ where $S = \{\alpha \leftarrow (\beta \to \beta)\}$. In this case, the renaming of β into δ was necessary: we did not want the variable β introduced by S to be captured by the universal quantification $\forall \beta$.

[1] A metavariable should not be confused with a *variable* or a *type variable*.

□

The type scheme $\sigma' = \forall\beta_1 \ldots \beta_m.\tau'$ is said to be a *generic instance* of $\sigma = \forall\alpha_1 \ldots \alpha_n.\tau$ if there exists a substitution S such that:

- the domain of S is included in $\{\alpha_1, \ldots, \alpha_n\}$;
- $\tau' = S(\tau)$;
- no β_i occurs free in σ.

In other words, a generic instance of a type scheme is obtained by giving more precise values to some generic variables, and (possibly) quantifying some of the new type variables introduced.

Example If $\sigma = \forall\beta.(\beta \to \alpha) \to \alpha$, then $\sigma' = \forall\gamma.((\gamma \to \gamma) \to \alpha) \to \alpha$ is a generic instance of σ. We changed β into $(\gamma \to \gamma)$, and we universally quantified on the newly introduced type variable γ.
□

We express this type system by means of *inference rules*. An inference rule is written as a fraction:

- the numerator is called the *premisses*;
- the denominator is called the *conclusion*.

An inference rule:

$$\frac{P_1 \ \ldots \ P_n}{C}$$

may be read in two ways:

- "**If** $P_1, \ldots$ **and** P_n, **then** C".
- "**In order to prove** C, **it is sufficient to prove** $P_1, \ldots$ **and** P_n".

An inference rule may have no premise: such a rule will be called an *axiom*. A complete proof will be represented by a *proof tree* of the following form:

$$\frac{\dfrac{\underline{P_1^m} \ \ldots \qquad \ldots \ \underline{P_l^k}}{\vdots}}{\dfrac{\underline{\quad} \ \ldots \ \underline{\quad}}{\dfrac{P_1^1 \ \ldots \ P_n^1}{C}}}$$

where the leaves of the tree ($P_1^m, \ldots, P_l^k$) are instances of axioms.

In the premisses and the conclusions appear *judgements* having the form:

$$\Gamma \vdash e : \sigma$$

Such a judgement should be read as "under the typing environment Γ, the expression e has type scheme σ". Typing environments are sets of *typing hypotheses* of the form $x : \sigma$ where x is an identifier name and σ is a type scheme: typing environments give types to the variables occurring free (i.e. unbound) in the expression.

When typing λ-calculus terms, the typing environment is managed as a *stack* (because identifiers possess local scopes). We represent that fact in the presentation of the type system by *removing* the typing hypothesis concerning an identifier name x (if such a typing hypothesis exists) before adding a new typing hypothesis concerning x.

We write $\Gamma - \Gamma(x)$ for the set ot typing hypotheses obtained from Γ by removing the typing hypothesis concerning x (if it exists).

Any numeric constant is of type Number:

$$\frac{}{\Gamma \vdash \texttt{Const}\ \ n : \text{Number}}(\text{NUM})$$

We obtain type schemes for variables from the typing environment Γ:

$$\frac{}{\Gamma \cup \{x : \sigma\} \vdash \texttt{Var}\ \ x : \sigma}(\text{TAUT})$$

It is possible to instantiate type schemes. The "GenInstance" relation represents generic instantiation.

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma' = \text{GenInstance}(\sigma)}{\Gamma \vdash e : \sigma'}(\text{INST})$$

It is possible to generalize type schemes with respect to variables that do not occur free in the set of hypotheses:

$$\frac{\Gamma \vdash e : \sigma \qquad \alpha \notin FV(\Gamma)}{\Gamma \vdash e : \forall\alpha.\sigma}(\text{GEN})$$

Typing a conditional:

$$\frac{\Gamma \vdash e_1 : \text{Number} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1\ \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}) : \tau}(\text{IF})$$

Typing an application:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}(\text{APP})$$

Typing an abstraction:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

The special rule below is the one that introduces polymorphism: this corresponds to the ML `let` construct.

$$\frac{\Gamma \vdash e : \sigma \qquad (\Gamma - \Gamma(x)) \cup \{x : \sigma\} \vdash e' : \tau}{\Gamma \vdash (\lambda x\ .e')\ e : \tau}(\text{LET})$$

This type system has been proven to be *semantically sound*, i.e. the semantic value of a well-typed expression (an expression admitting a type) cannot be an *error value* due to a type error. This is usually expressed as:

> Well-typed programs cannot go wrong.

This fact implies that a clever compiler may produce code without any dynamic type test for a well-typed expression.

Example Let us check, using the set of rules above, that the following is true:

$$\emptyset \vdash \texttt{let f=}\lambda\texttt{x.x in f f} : \beta \to \beta$$

In order to do so, we will use the equivalence between the `let` construct and an application of an immediate abstraction (i.e. an expression having the following shape: $(\lambda v.M)N$. The (LET) rule will be crucial: without it, we could not check the judgement above.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x:\alpha}\text{TAUT}}{\emptyset \vdash (\lambda x.x){:}\alpha \to \alpha}\text{ABS}}{\emptyset \vdash (\lambda x.x){:}\forall\alpha.\alpha \to \alpha}\text{GEN} \quad \dfrac{\dfrac{\dfrac{}{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}\text{TAUT}}{\Gamma \vdash f : (\beta \to \beta) \to (\beta \to \beta)}\text{INST} \quad \dfrac{\dfrac{}{\Gamma \vdash f : \forall\alpha.\alpha \to \alpha}\text{TAUT}}{\Gamma \vdash f : \beta \to \beta}\text{INST}}{\Gamma = \{f : \forall\alpha.\alpha \to \alpha\} \vdash f\ f : \beta \to \beta}\text{APP}}{\emptyset \vdash (\lambda f.f\,f)(\lambda x.x){:}\beta \to \beta}\text{LET}$$

□

This type system does not tell us how to find the best type for an expression. But what is the best type for an expression? It must be such that any other possible type for that expression is more specific; in other words, the best type is the *most general*.

16.2 The algorithm

How do we find the most general type for an expression of our language? The problem with the set of rules above, is that we could instantiate and generalize types at any time, introducing type schemes, while the most important rules (application and abstraction) used only types.

Let us write a new set of inference rules that we will read as an algorithm (close to a Prolog program):
Any numeric constant is of type Number:

$$\frac{}{\Gamma \vdash \texttt{Const}\ \ n : \text{Number}}(\text{NUM})$$

The types of identifiers are obtained by taking generic instances of type schemes appearing in the typing environment. These generic instances will be *types* and not type schemes: this restriction appears in the rule below, where the type τ is expected to be a generic instance of the type scheme σ.

As it is presented (belonging to a deduction system), the following rule will have to anticipate the effect of the equality constraints between types in the other rules (multiple occurrences of a type metavariable), when choosing the instance τ.

$$\frac{\tau = \text{GenInstance}(\sigma)}{\Gamma \cup \{x:\sigma\} \vdash \texttt{Var}\ \ x : \tau}(\text{INST})$$

When we read this set of inference rules as an algorithm, the (INST) rule will be implemented by:

1. taking as τ the "most general generic instance" of σ that is a type (the rule requires τ to be a type and not a type scheme),

2. making τ more specific by *unification* [30] when encountering equality constraints.

Typing a conditional requires only the test part to be of type Number, and both alternatives to be of the same type τ. This is an example of equality constraint between the types of two expressions.

$$\frac{\Gamma \vdash e_1 : \text{Number} \qquad \Gamma \vdash e_2 : \tau \qquad \Gamma \vdash e_3 : \tau}{\Gamma \vdash (\texttt{if}\ e_1 \texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}) : \tau}(\text{COND})$$

Typing an application produces also equality constraints that are to be solved by unification:

$$\frac{\Gamma \vdash e_1 : \tau \to \tau' \qquad \Gamma \vdash e_2 : \tau}{\Gamma \vdash (e_1\ e_2) : \tau'}(\text{APP})$$

Typing an abstraction "pushes" a typing hypothesis for the abstracted identifier: unification will make it more precise during the typing of the abstraction body:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \forall.\tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

Typing a `let` construct involves a generalization step: we generalize as much as possible.

$$\frac{\Gamma \vdash e : \tau' \qquad \vec{\alpha} = FV(\tau') - FV(\Gamma) \qquad (\Gamma - \Gamma(x)) \cup \{x : \forall\vec{\alpha}.\tau'\} \vdash e' : \tau}{\Gamma \vdash (\lambda x\ .e')\ e : \tau}(\text{LET})$$

This set of inference rules represents an algorithm because there is exactly one conclusion for each syntactic ASL construct (giving priority to the (LET) rule over the regular application rule). This set of rules may be read as a Prolog program.

This algorithm has been proven to be:

- *syntactically sound*: when the algorithm succeeds on an expression e and returns a type τ, then $e : \tau$.
- *complete*: if an expression e possesses a type τ, then the algorithm will find a type τ' such that τ is an instance of τ'. The returned type τ' is thus the most general type of e.

Example We compute the type that we simply checked in our last example. What is drawn below is the result of the type synthesis: in fact, we run our algorithm with type variables representing unknowns, modifying the previous applications of the (INST) rule when necessary (i.e. when encountering an equality constraint). This is valid, since it can be proved that the correction of the whole deduction tree is preserved by substitution of types for type variables. In a real implementation of the algorithm, the data structures representing types will be submitted to a unification mechanism.

$$\dfrac{\dfrac{\dfrac{}{\{x:\alpha\} \vdash x : \alpha}\text{INST}_x}{\emptyset \vdash (\lambda x.x) : \alpha \to \alpha}\text{ABS}_x \quad \dfrac{\dfrac{}{\Gamma \vdash f\ : (\beta \to \beta) \to (\beta \to \beta)}\text{INST}_f \quad \dfrac{}{\Gamma \vdash f\ : \beta \to \beta}\text{INST}_f}{\Gamma = \{f : \forall\alpha.\alpha \to \alpha\} \vdash f\ f : \beta \to \beta}\text{APP}}{\emptyset \vdash (\lambda f.f\,f)(\lambda x.x) : \beta \to \beta}\text{LET}_f$$

Once again, this expression is not typable without the use of the (LET) rule: an error would occur because of the type equality constraints between all occurrences of a variable bound by a "λ". In an effective implementation, a unification error would occur.

□

We may notice from the example above that the algorithm is *syntax-directed*: since, for a given expression, a type deduction for that expression uses exactly one rule per sub-expression,

the deduction possesses the same structure as the expression. We can thus reconstruct the ASL expression from its type deduction tree. From the deduction tree above, if we write upper rules as being "arguments" of the ones below and if we annotate the applications of the (INST) and (ABS) rules by the name of the subject variable, we obtain:

$$\mathrm{LET}_f(\mathrm{ABS}_x(\mathrm{INST}_x),\ \mathrm{APP}(\mathrm{INST}_f,\ \mathrm{INST}_f))$$

This is an illustration of the "types-as-propositions and programs-as-proofs" paradigm, also known as the "Curry-Howard isomorphism" (cf. [14]). In this example, we can see the type of the considered expression as a proposition and the expression itself as the proof, and, indeed, we recognize the expression as the deduction tree.

16.3 The ASL type-synthesizer

We now implement the set of inference rules given above.

We need:

- a Caml representation of ASL types and type schemes,
- a management of type environments,
- a unification procedure,
- a typing algorithm.

16.3.1 Representation of ASL types and type schemes

We first need to define a Caml type for our ASL type data structure:

```
#type asl_type = Unknown
#              | Number
#              | TypeVar of vartype
#              | Arrow of asl_type * asl_type
#and vartype = {Index:int; mutable Value:asl_type}
#and asl_type_scheme = Forall of int list * asl_type ;;
Type asl_type defined.
Type vartype defined.
Type asl_type_scheme defined.
```

The `Unknown` ASL type is not really a type: it is the initial value of fresh ASL type variables. We will consider as abnormal a situation where `Unknown` appears in place of a regular ASL type. In such situations, we will raise the following exception:

```
#exception TypingBug of string;;
Exception TypingBug defined.
```

Type variables are allocated by the `new_vartype` function, and their global counter (a local reference) is reset by `reset_vartypes`.

```
#let new_vartype, reset_vartypes =
#(* Generating and resetting unknowns *)
#    let counter = ref 0
#    in (function () -> counter:=!counter+1;
#                       {Index=!counter; Value=Unknown}),
#       (function () -> counter:=0; ());;
reset_vartypes : unit -> unit = <fun>
new_vartype : unit -> vartype = <fun>
```

16.3.2 Destructive unification of ASL types

We will need to "shorten" type variables: since they are indirections to ASL types, we need to follow these indirections in order to obtain the type that they represent. For the sake of efficiency, we take advantage of this operation to replace multiple indirections by single indirections (shortening).

```
#let rec shorten t =
#    match t with
#     TypeVar {Index=_; Value=Unknown} -> t
#   | TypeVar ({Index=_;
#               Value=TypeVar {Index=_;
#                              Value=Unknown} as tv}) -> tv
#   | TypeVar ({Index=_; Value=TypeVar tv1} as tv2)
#            -> tv2.Value <- tv1.Value; shorten t
#   | TypeVar {Index=_; Value=t'} -> t'
#   | Unknown -> raise (TypingBug "shorten")
#   | t' -> t';;
shorten : asl_type -> asl_type = <fun>
```

An ASL type error will be represented by the following exception:

```
#exception TypeClash of asl_type * asl_type;;
Exception TypeClash defined.
```

We will need unification on ASL types with *occur-check*. The following function implements occur-check:

```
#let occurs {Index=n;Value=_} = occrec
#  where rec occrec =
#  function TypeVar{Index=m;Value=_} -> (n=m)
#         | Number -> false
#         | Arrow(t1,t2) -> (occrec t1) or (occrec t2)
#         | Unknown -> raise (TypingBug "occurs");;
occurs : vartype -> asl_type -> bool = <fun>
```

The unification function: implements destructive unification. Instead of returning the most general unifier, it returns the unificand of two types (their most general common instance). The two arguments are physically modified in order to represent the same type. The unification function will detect type clashes.

```
#let rec unify (tau1,tau2) =
#  match (shorten tau1, shorten tau2)
#  with (* type variable n and type variable m *)
#       (TypeVar({Index=n; Value=Unknown} as tv1) as t1),
#       (TypeVar({Index=m; Value=Unknown} as tv2) as t2)
#           -> if n <> m then tv1.Value <- t2
#     | (* type t1 and type variable *)
#      t1, (TypeVar ({Index=_;Value=Unknown} as tv) as t2)
#            -> if not(occurs tv t1) then tv.Value <- t1
#               else raise (TypeClash (t1,t2))
#     | (* type variable and type t2 *)
#       (TypeVar ({Index=_;Value=Unknown} as tv) as t1), t2
#            -> if not(occurs tv t2) then tv.Value <- t2
#               else raise (TypeClash (t1,t2))
#     | Number, Number -> ()
#     | Arrow(t1,t2), (Arrow(t'1,t'2) as t)
#            -> unify(t1,t'1); unify(t2,t'2)
#     | (t1,t2) -> raise (TypeClash (t1,t2));;
unify : asl_type * asl_type -> unit = <fun>
```

16.3.3 Representation of typing environments

We use `asl_type_scheme list` as typing environments, and we will use the encoding of variables as indices into the environment.

The initial environment is a list of types `(Number -> (Number -> Number))`, which are the types of the ASL primitive functions.

```
#let init_typing_env =
#    map (function s ->
#            Forall([],Arrow(Number,
#                            Arrow(Number,Number))))
#        init_env;;
init_typing_env : asl_type_scheme list = [Forall ([], Arrow (Number, Arrow (Numb
er, Number))); Forall ([], Arrow (Number, Arrow (Number, Number))); Forall ([],
Arrow (Number, Arrow (Number, Number))); Forall ([], Arrow (Number, Arrow (Numbe
r, Number))); Forall ([], Arrow (Number, Arrow (Number, Number)))]
```

The global typing environment is initialized to the initial typing environment, and will be updated with the type of each ASL declaration, after they are type-checked.

```
#let global_typing_env = ref init_typing_env;;
global_typing_env : asl_type_scheme list ref = ref [Forall ([], Arrow (Number, A
rrow (Number, Number))); Forall ([], Arrow (Number, Arrow (Number, Number))); Fo
rall ([], Arrow (Number, Arrow (Number, Number))); Forall ([], Arrow (Number, Ar
row (Number, Number))); Forall ([], Arrow (Number, Arrow (Number, Number)))]
```

16.3.4 From types to type schemes: generalization

In order to implement generalization, we will need some functions collecting types variables occurring in ASL types.

The following function computes the list of type variables of its argument.

```
#let vars_of_type tau = vars [] tau
# where rec vars vs =
#  function Number -> vs
#         | TypeVar {Index=n; Value=Unknown}
#                 -> if mem n vs then vs else n::vs
#         | TypeVar {Index=_; Value= t} -> vars vs t
#         | Arrow(t1,t2) -> vars (vars vs t1) t2
#         | Unknown -> raise (TypingBug "vars_of_type");;
vars_of_type : asl_type -> int list = <fun>
```

The `unknowns_of_type(bv,t)` application returns the list of variables occurring in `t` that do not appear in `bv`. The `subtract` function returns the difference of two lists.

```
#let unknowns_of_type (bv,t) =
#    subtract (vars_of_type t) bv;;
unknowns_of_type : int list * asl_type -> int list = <fun>
```

We need to compute the list of unknowns of a type environment for the generalization process (unknowns belonging to that list cannot become generic). The set of unknowns of a type environment is the union of the unknowns of each type. The `flat` function flattens a list of lists.

```
#let flat = it_list (prefix @) [];;
flat : 'a list list -> 'a list = <fun>
```

```
#let unknowns_of_type_env env =
#    flat (map (function Forall(gv,t) -> unknowns_of_type (gv,t)) env);;
unknowns_of_type_env : asl_type_scheme list -> int list = <fun>
```

The generalization of a type is relative to a typing environment. The `make_set` function eliminates duplicates in its list argument.

```
#let rec make_set = function
#     []  -> []
#  | x::l -> if mem x l then make_set l else x :: make_set l;;
make_set : 'a list -> 'a list = <fun>
```

```
#let generalise_type (gamma, tau) =
#  let genvars =
#        make_set (subtract (vars_of_type tau)
#                           (unknowns_of_type_env gamma))
#  in Forall(genvars, tau)
#;;
generalise_type : asl_type_scheme list * asl_type -> asl_type_scheme = <fun>
```

16.3.5 From type schemes to types: generic instantiation

The following function returns a generic instance of its type scheme argument. A generic instance is obtained by replacing all generic type variables by new unknowns:

```
#let gen_instance (Forall(gv,tau)) =
#  (* We associate a new unknown to each generic variable *)
#  let unknowns =
#      map (function n -> n, TypeVar(new_vartype()))
#          gv
#  in ginstance tau
#  where rec ginstance = function
#        (TypeVar {Index=n; Value=Unknown} as t) ->
#                     (try assoc n unknowns
#                      with Not_found -> t)
#      | TypeVar {Index=_; Value= t} -> ginstance t
#      | Number -> Number
#      | Arrow(t1,t2) -> Arrow(ginstance t1, ginstance t2)
#      | Unknown -> raise (TypingBug "gen_instance")
#;;
gen_instance : asl_type_scheme -> asl_type = <fun>
```

16.3.6 The ASL type synthesizer

The type synthesizer is the `asl_typing_expr` function. Each of its match cases corresponds to an inference rule given above.

```
#let rec asl_typing_expr gamma = type_rec
#where rec type_rec = function
#    Const _ -> Number
#  | Var n ->
#      let sigma =
#        try nth n gamma
#        with Failure _ -> raise (TypingBug "Unbound")
#      in gen_instance sigma
#  | Cond (e1,e2,e3) ->
#      unify(Number, type_rec e1);
#      let t2 = type_rec e2 and t3 = type_rec e3
#      in unify(t2, t3); t3
#  | App((Abs(x,e2) as f), e1) -> (* LET case *)
#      let t1 = type_rec e1 in
#      let sigma = generalise_type (gamma,t1)
#      in asl_typing_expr (sigma::gamma) e2
#  | App(e1,e2) ->
#      let u = TypeVar(new_vartype())
#      in unify(type_rec e1,Arrow(type_rec e2,u)); u
```

```
#  | Abs(x,e) ->
#      let u = TypeVar(new_vartype()) in
#      let s = Forall([],u)
#      in Arrow(u,asl_typing_expr (s::gamma) e);;
asl_typing_expr : asl_type_scheme list -> asl -> asl_type = <fun>
```

16.3.7 Typing, trapping type clashes and printing ASL types

Now, we define some auxiliary functions in order to build a "good-looking" type synthesizer. First of all, a printing routine for ASL type schemes is defined (using a function `tvar_name` which computes a decent name for type variables).

```
#let tvar_name n =
# (* Computes a name "'a", ... for type variables, *)
# (* given an integer n representing the position  *)
# (* of the type variable in the list of generic   *)
# (* type variables                                *)
# let rec name_of n =
#    let q,r = (n / 26), (n mod 26) in
#    let s = make_string 1 (char_of_int (96+r)) in
#    if q=0 then s else (name_of q)^s
# in "'"^(name_of n)
#;;
tvar_name : int -> string = <fun>
```

Then a printing function for type schemes.

```
#let print_type_scheme (Forall(gv,t)) =
# (* Prints a type scheme.               *)
# (* Fails when it encounters an unknown *)
# let names = (names_of (1,gv)
#      where rec names_of = function
#             (n,[]) -> []
#           | (n,(v1::Lv)) -> (tvar_name n)
#                             ::(names_of (n+1, Lv))) in
# let tvar_names = combine (rev gv,names)
# in print_rec t
#    where rec print_rec = function
#      TypeVar{Index=n; Value=Unknown} ->
#         let name =
#             try assoc n tvar_names
#             with Not_found ->
#               raise (TypingBug "Non generic variable")
#         in print_string name
#    | TypeVar{Index=_;Value=t} -> print_rec t
#    | Number -> print_string "Number"
```

```
#    | Arrow(t1,t2) ->
#           print_string "("; print_rec t1;
#           print_string " -> "; print_rec t2;
#           print_string ")"
#    | Unknown -> raise (TypingBug "print_type_scheme");;
print_type_scheme : asl_type_scheme -> unit = <fun>
```

Now, the main function which resets the type variables counter, calls the type synthesizer, traps ASL type clashes and prints the resulting types. At the end, the global environments are updated.

```
#let typing (Decl(s,e)) =
# reset_vartypes();
# let tau = (* TYPING *)
#     try asl_typing_expr !global_typing_env e
#     with TypeClash(t1,t2) -> (* A typing error *)
#           let vars=vars_of_type(t1)@vars_of_type(t2) in
#           print_string "ASL Type clash between ";
#           print_type_scheme (Forall(vars,t1));
#           print_string " and ";
#           print_type_scheme (Forall(vars,t2));
#           print_newline();
#           raise (Failure "ASL typing") in
# let sigma = generalise_type (!global_typing_env,tau) in
# (* UPDATING ENVIRONMENTS *)
# global_env := s::!global_env;
# global_typing_env := sigma::!global_typing_env;
# reset_vartypes ();
# (* PRINTING RESULTING TYPE *)
# print_string "ASL Type of ";
# print_string s;
# print_string " is ";
# print_type_scheme sigma; print_newline();;
typing : top_asl -> unit = <fun>
```

16.3.8 Typing ASL programs

We reinitialize the parsing environment:

```
#global_env:=init_env; ();;
- : unit = ()
```

Now, let us run some examples through the ASL type checker:

```
#typing (parse_top "let x be 1;");;
ASL Type of x is Number
- : unit = ()
```

```
#typing (parse_top "+ 2 ((\\x.x) 3);");;
```

```
ASL Type of it is Number
- : unit = ()

#typing (parse_top "if + 0 1 then 1 else 0 fi;");;
ASL Type of it is Number
- : unit = ()

#typing (parse_top "let id be \\x.x;");;
ASL Type of id is ('a -> 'a)
- : unit = ()

#typing (parse_top "+ (id 1) (id id 2);");;
ASL Type of it is Number
- : unit = ()

#typing (parse_top "let f be (\\x.x x) (\\x.x);");;
ASL Type of f is ('a -> 'a)
- : unit = ()

#typing (parse_top "+ (\\x.x) 1;");;
ASL Type clash between Number and ('a -> 'a)
Uncaught exception: Failure "ASL typing"
```

16.3.9 Typing and recursion

The Z fixpoint combinator does not have a type in Milner's type system:

```
#typing (parse_top
#  "let fix be \\f.((\\x.f(\\z.(x x)z)) (\\x.f(\\z.(x x)z)));");;
ASL Type clash between 'a and ('a -> 'b)
Uncaught exception: Failure "ASL typing"
```

This is because we try to apply `x` to itself, and the type of `x` is not polymorphic. In fact, no fixpoint combinator is typable in ASL. This is why we need a special primitive or syntactic construct in order to express recursivity.

If we want to assign types to recursive programs, we have to predefine the Z fixpoint combinator. Its type scheme should be $\forall\alpha.((\alpha \to \alpha) \to \alpha)$, because we take fixpoints of functions.

```
#global_env := "fix"::init_env;
#global_typing_env:=
#    (Forall([1],
#     Arrow(Arrow(TypeVar{Index=1;Value=Unknown},
#                 TypeVar{Index=1;Value=Unknown}),
#           TypeVar{Index=1;Value=Unknown})))
#   ::init_typing_env;
#();;
- : unit = ()
```

We can now define our favorite functions as:

```
#typing (parse_top
#    "let fact be fix (\\f.(\\n. if = n 0 then 1
#                            else * n (f (- n 1))
#                            fi));");;
ASL Type of fact is (Number -> Number)
- : unit = ()

#typing (parse_top "fact 8;");;
ASL Type of it is Number
- : unit = ()

#typing (parse_top
#  "let fib be fix (\\f.(\\n. if = n 1 then 1
#                             else if = n 2 then 1
#                                  else + (f(- n 1)) (f(- n 2))
#                                  fi
#                             fi));");;
ASL Type of fib is (Number -> Number)
- : unit = ()

#typing (parse_top "fib 9;");;
ASL Type of it is Number
- : unit = ()
```

Chapter 17

Compiling ASL to an abstract machine code

In order to fully take advantage of the static typing of ASL programs, we have to:

- either write a new interpreter without type tests (difficult, because we used pattern-matching in order to realize type tests);
- or design an untyped machine and produce code for it.

We choose here the second solution: it will permit us to give some intuition about the compiling process of functional languages, and to describe a typical execution model for (strict) functional languages. The machine that we will use is a simplified version the *Categorical Abstract Machine* (CAM, for short).

We will call CAM our abstract machine, despite its differences with the original CAM. For more informations on the CAM, see [8, 24].

17.1 The Abstract Machine

The execution model is a *stack machine* (i.e. a machine using a stack). In this section, we define in Caml the *states* of the CAM and its instructions.

A state is composed of:

- a *register* (holding values and environments),
- a *program counter*, represented here as a list of instructions whose first element is the current instruction being executed,
- and a *stack* represented as a list of code addresses (instruction lists), values and environments.

The first Caml type that we need is the type for CAM instructions. We will study later the effect of each instruction.

```
#type instruction =
#  Quote of int              (* Integer constants *)
```

```
#| Plus | Minus               (* Arithmetic operations *)
#| Divide | Equal | Times
#| Nth of int                 (* Variable accesses *)
#| Branch of instruction list * instruction list
#                             (* Conditional execution *)
#| Push                      (* Pushes onto the stack *)
#| Swap                      (* Exch. top of stack and register *)
#| Clos of instruction list  (* Builds a closure with the current environment *)
#| Apply                     (* Function application *)
#;;
Type instruction defined.
```

We need a new type for semantic values since instruction lists have now replaced abstract syntax trees. The semantic values are merged in a type `object`. The type `object` behaves as data in a computer memory: we need higher-level information (such as type information) in order to interpret them. Furthermore, some data do not correspond to anything (for example an environment composed of environments represents neither an ASL value nor an intermediate data in a legal computation process).

```
#type object = Constant of int
#            | Closure of object * object
#            | Address of instruction list
#            | Environment of object list
#;;
Type object defined.
```

The type `state` is a product type with mutable components.

```
#type state = {mutable Reg: object;
#              mutable PC: instruction list;
#              mutable Stack: object list}
#;;
Type state defined.
```

Now, we give the *operational semantics* of CAM instructions. The effect of an instruction is to change the state configuration. This is what we describe now with the `step` function. Code executions will be arbitrary iterations of this function.

```
#exception CAMbug of string;;
Exception CAMbug defined.

#exception CAM_End of object;;
Exception CAM_End defined.

#let step state = match state with
#  {Reg=_; PC=Quote(n)::code; Stack=s} ->
#                state.Reg <- Constant(n); state.PC <- code
#
```

```
#| {Reg=Constant(m); PC=Plus::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n+m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Minus::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n-m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Times::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n*m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Divide::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(n/m); state.Stack <- s;
#                state.PC <- code
#
#| {Reg=Constant(m); PC=Equal::code; Stack=Constant(n)::s} ->
#                state.Reg <- Constant(if n=m then 1 else 0);
#                state.Stack <- s; state.PC <- code
#
#| {Reg=Constant(m); PC=Branch(code1,code2)::code; Stack=r::s} ->
#                state.Reg <- r;
#                state.Stack <- Address(code)::s;
#                state.PC <- (if m=0 then code2 else code1)
#
#| {Reg=r; PC=Push::code; Stack=s} ->
#                state.Stack <- r::s; state.PC <- code
#
#| {Reg=r1; PC=Swap::code; Stack=r2::s} ->
#                state.Reg <- r2; state.Stack <- r1::s;
#                state.PC <- code
#
#| {Reg=r; PC=Clos(code1)::code; Stack=s} ->
#                state.Reg <- Closure(Address(code1),r);
#                state.PC <- code
#
#| {Reg=_; PC=[]; Stack=Address(code)::s} ->
#                state.Stack <- s; state.PC <- code
#
#| {Reg=v; PC=Apply::code;
#          Stack=Closure(Address(code1),Environment(e))::s} ->
#                state.Reg <- Environment(v::e);
#                state.Stack <- Address(code)::s;
#                state.PC <- code1
#
```

```
#| {Reg=v; PC=[]; Stack=[]} ->
#                 raise (CAM_End v)
#| {Reg=_; PC=(Plus|Minus|Times|Divide|Equal)::code; Stack=_::_} ->
#                 raise (CAMbug "IllTyped")
#
#| {Reg=Environment(e); PC=Nth(n)::code; Stack=_} ->
#                 state.Reg <- (try nth n e
#                               with Failure _ -> raise (CAMbug "IllTyped"));
#                 state.PC <- code
#| _ -> raise (CAMbug "Wrong configuration")
#;;
step : state -> unit = <fun>
```

We may notice that the empty code sequence denotes a (possibly final) *return* instruction.

We could argue that pattern-matching in the `Camlstep` function implements a kind of dynamic typechecking. In fact, in a concrete (low-level) implementation of the machine (expansion of the CAM instructions in assembly code, for example), these tests would not appear. They are useless since we trust the typechecker and the compiler. So, any execution error in a real implementation comes from a *bug* in one of the above processes and would lead to memory errors or illegal instructions (usually detected by the computer's operating system).

17.2 Compiling ASL programs into CAM code

We give in this section a compiling function taking the abstract syntax tree of an ASL expression and producing CAM code. The compilation scheme is very simple:

- the code of a constant is `Quote`;
- a variable is compiled as an access to the appropriate component of the current environment (`Nth`);
- the code of a conditional expression will save the current environment (`Push`), evaluate the condition part, and, according to the boolean value obtained, select the appropriate code to execute (`Branch`);
- the code of an application will also save the environment on the stack (`Push`), execute the function part of the application, then exchange the functional value and the saved environment (`Swap`), evaluate the argument and, finally, apply the functional value (which is at the top of the stack) to the argument held in the register with the `Apply` instruction;
- the code of an abstraction simply consists in building a closure representing the functional value: the closure is composed of the code of the function and the current environment.

Here is the compiling function:

```
#let rec code_of = function
#  Const(n) -> [Quote(n)]
#| Var n -> [Nth(n)]
```

```
#| Cond(e_test,e_t,e_f) ->
#         Push::(code_of e_test)
#       @[Branch(code_of e_t, code_of e_f)]
#| App(e1,e2) -> Push::(code_of e1)
#               @[Swap]@(code_of e2)
#               @[Apply]
#| Abs(_,e) -> [Clos(code_of e)];;
code_of : asl -> instruction list = <fun>
```

A global environment is needed in order to maintain already defined values. Any CAM execution will start in a state whose register part contains this global environment.

```
#let init_CAM_env =
#  let basic_instruction = function
#        "+" -> Plus
#      | "-" -> Minus
#      | "*" -> Times
#      | "/" -> Divide
#      | "=" -> Equal
#      | s -> raise (CAMbug "Unknown primitive")
#  in map (function s ->
#           Closure(Address[Clos(Push::Nth(2)
#                                ::Swap::Nth(1)
#                                ::[basic_instruction s])],
#                   Environment []))
#         init_env;;
init_CAM_env : object list = [Closure (Address [Clos [Push; Nth 2; Swap; Nth 1;
Plus]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Minus
]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Times]],
Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Divide]], Env
ironment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Equal]], Environ
ment [])]
```

```
#let global_CAM_env = ref init_CAM_env;;
global_CAM_env : object list ref = ref [Closure (Address [Clos [Push; Nth 2; Swa
p; Nth 1; Plus]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nt
h 1; Minus]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1;
 Times]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Div
ide]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Equal]
], Environment [])]
```

As an example, here is the code for some ASL expressions.

```
#code_of (expression(lexer(stream_of_string "1;")));;
- : instruction list = [Quote 1]
```

```
#code_of (expression(lexer(stream_of_string "+ 1 2;")));;
- : instruction list = [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Quote 2;
```

```
Apply]
#code_of (expression(lexer(stream_of_string "(\\x.x) ((\\x.x) 0);")));;
- : instruction list = [Push; Clos [Nth 1]; Swap; Push; Clos [Nth 1]; Swap; Quot
e 0; Apply; Apply]
#code_of (expression(lexer(stream_of_string
#                "+ 1 (if 0 then 2 else 3 fi);")));;
- : instruction list = [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Push; Quo
te 0; Branch ([Quote 2], [Quote 3]); Apply]
```

17.3 Execution of CAM code

The main function for executing compiled ASL manages the global environment until execution has succeeded.

```
#let run (Decl(s,e)) =
#  (* TYPING *)
#    reset_vartypes();
#    let tau =
#        try asl_typing_expr !global_typing_env e
#        with TypeClash(t1,t2) ->
#              let vars=vars_of_type(t1) @ vars_of_type(t2) in
#              print_string "ASL Type clash between ";
#              print_type_scheme (Forall(vars,t1));
#              print_string " and ";
#              print_type_scheme (Forall(vars,t2));
#              raise (Failure "ASL typing")
#           | Unbound s -> raise (TypingBug ("Unbound: "^s)) in
#    let sigma = generalise_type (!global_typing_env,tau) in
#  (* PRINTING TYPE INFORMATION *)
#    print_string "ASL Type of ";
#    print_string s; print_string " is ";
#    print_type_scheme sigma; print_newline();
#  (* COMPILING *)
#    let code = code_of e in
#    let state = {Reg=Environment(!global_CAM_env); PC=code; Stack=[]} in
#  (* EXECUTING *)
#    let result = try while true do step state done; state.Reg
#                 with CAM_End v -> v in
#  (* UPDATING ENVIRONMENTS *)
#    global_env := s::!global_env;
#    global_typing_env := sigma::!global_typing_env;
#    global_CAM_env := result::!global_CAM_env;
#  (* PRINTING RESULT *)
#    (match result
```

```
#     with Constant(n) -> print_int n
#        | Closure(_,_) -> print_string "<funval>"
#        | _ -> raise (CAMbug "Wrong state configuration"));
#    print_newline();;
run : top_asl -> unit = <fun>
```

Now, let us run some examples:

```
#(* Reinitializing environments *)
#global_env:=init_env;
#global_typing_env:=init_typing_env;
#global_CAM_env:=init_CAM_env;
#();;
- : unit = ()

#run (parse_top "1;");;
ASL Type of it is Number
1
- : unit = ()

#run (parse_top "+ 1 2;");;
ASL Type of it is Number
3
- : unit = ()

#run (parse_top "(\\f.(\\x.f x)) (\\x. + x 1) 3;");;
ASL Type of it is Number
4
- : unit = ()
```

We may now introduce the Z fixpoint combinator as a predefined function `fix`.

```
#begin
#  global_env:="fix"::init_env;
#  global_typing_env:=
#    (Forall([1],
#            Arrow(Arrow(TypeVar{Index=1;Value=Unknown},
#                        TypeVar{Index=1;Value=Unknown}),
#                  TypeVar{Index=1;Value=Unknown})))
#   ::init_typing_env;
#  global_CAM_env:=
#   (match code_of (expression(lexer(stream_of_string
#           "\\f.((\\x.f(\\z.(x x)z)) (\\x.f(\\z.(x x)z)));")))
#    with [Clos(C)] -> Closure(Address(C), Environment [])
#       | _ -> raise (CAMbug "Wrong code for fix"))
#   ::init_CAM_env
#end;;
- : unit = ()
```

```
#run (parse_top
#    "let fact be fix (\\f.(\\n. if = n 0 then 1
#                           else * n (f (- n 1))
#                           fi));");;
ASL Type of fact is (Number -> Number)
<funval>
- : unit = ()

#run (parse_top
#  "let fib be fix (\\f.(\\n. if = n 1 then 1
#                             else if = n 2 then 1
#                                  else + (f(- n 1)) (f(- n 2))
#                                  fi
#                             fi));");;
ASL Type of fib is (Number -> Number)
<funval>
- : unit = ()

#run (parse_top "fact 8;");;
ASL Type of it is Number
40320
- : unit = ()

#run (parse_top "fib 9;");;
ASL Type of it is Number
34
- : unit = ()
```

It is of course possible (and desirable) to introduce recursion by using a specific syntactic construct, special instructions and a dedicated case to the compiling function. See [24] for efficient compilation of recursion, data structures etc.

17.1 *Interesting exercises for which we won't give solutions consist in enriching according to your taste the ASL language. Also, building a standalone ASL interpreter is a good exercise in modular programming.*

Chapter 18

Answers to exercises

We give in this chapter one possible solution for each exercise contained in this document. Exercises are referred to by their number and the page where they have been proposed: for example, "2.1, p. 15" refers to the first exercise in chapter 2; this exercise is located on page 15.

4.1, p. 37

The following (anonymous) functions have the required types:

1. `#function f -> (f 2)+1;;`
   `- : (int -> int) -> int = <fun>`
2. `#function m -> (function n -> n+m+1);;`
   `- : int -> int -> int = <fun>`
3. `#(function f -> (function m -> f(m+1) / 2));;`
   `- : (int -> int) -> int -> int = <fun>`

4.2, p. 38

We must first rename `y` to `z`, obtaining:

```
(function x -> (function z -> x+z))
```

and finally:

```
(function y -> (function z -> y+z))
```

Without renaming, we would have obtained:

```
(function y -> (function y -> y+y))
```

which does not denote the same function.

4.3, p. 38

We write successively the reduction steps for each expressions, and then we use Caml in order to check the result.

- ```
  let x=1+2 in ((function y -> y+x) x);;
  (function y -> y+(1+2)) (1+2);;
  (function y -> y+(1+2)) 3;;
  3+(1+2);;
  3+3;;
  6;;
  ```

  Caml says:

```
#let x=1+2 in ((function y -> y+x) x);;
- : int = 6
```

- ```
  let x=1+2 in ((function x -> x+x) x);;
  (function x -> x+x) (1+2);;
  3+3;;
  6;;
  ```

  Caml says:

```
#let x=1+2 in ((function x -> x+x) x);;
- : int = 6
```

- ```
  let f1 = function f2 -> (function x -> f2 x)
  in let g = function x -> x+1
     in f1 g 2;;
  let g = function x -> x+1
  in function f2 -> (function x -> f2 x) g 2;;
  (function f2 -> (function x -> f2 x)) (function x -> x+1) 2;;
  (function x -> (function x -> x+1) x)  2;;
  (function x -> x+1) 2;;
  2+1;;
  3;;
  ```

  Caml says:

```
#let f1 = function f2 -> (function x -> f2 x)
#in let g = function x -> x+1
#   in f1 g 2;;
- : int = 3
```

5.1, p. 49

To compute the surface area of a rectangle and the volume of a sphere:
```

```
#let surface_rect len wid = len * wid;;
surface_rect : int -> int -> int = <fun>

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

#let volume_sphere r = 4.0 /. 3.0 *. pi *. (power r 3.);;
volume_sphere : float -> float = <fun>
```

5.2, p. 49

In a call-by-value language without conditional construct (and without sum types), all programs involving a recursive definition never terminate.

5.3, p. 49

```
#let rec factorial n = if n=1 then 1 else n*(factorial(n-1));;
factorial : int -> int = <fun>

#factorial 5;;
- : int = 120

#let tail_recursive_factorial n =
#  let rec fact n m = if n=1 then m else fact (n-1) (n*m)
#  in fact n 1;;
tail_recursive_factorial : int -> int = <fun>

#tail_recursive_factorial 5;;
- : int = 120
```

5.4, p. 49

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

#fibonacci 20;;
- : int = 6765
```

5.5, p. 50

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

#let curry f = function x -> (function y -> f(x,y));;
curry : ('a * 'b -> 'c) -> 'a -> 'b -> 'c = <fun>
```

```
#let uncurry f = function (x,y) -> f x y;;
uncurry : ('a -> 'b -> 'c) -> 'a * 'b -> 'c = <fun>

#uncurry compose;;
- : ('a -> 'b) * ('c -> 'a) -> 'c -> 'b = <fun>

#compose curry uncurry;;
- : ('a -> 'b -> 'c) -> 'a -> 'b -> 'c = <fun>

#compose uncurry curry;;
- : ('a * 'b -> 'c) -> 'a * 'b -> 'c = <fun>
```

6.1, p. 54

```
#let rec combine =
#  function [],[] -> []
#         | (x1::l1),(x2::l2) -> (x1,x2)::(combine(l1,l2))
#         | _ -> raise (Failure "combine: lists of different length");;
combine : 'a list * 'b list -> ('a * 'b) list = <fun>

#combine ([1;2;3],["a";"b";"c"]);;
- : (int * string) list = [1, "a"; 2, "b"; 3, "c"]

#combine ([1;2;3],["a";"b"]);;
Uncaught exception: Failure "combine: lists of different length"
```

6.2, p. 54

```
#let rec sublists =
#    function [] -> [[]]
#           | x::l -> let sl = sublists l
#                     in sl @ (map (fun l -> x::l) sl);;
sublists : 'a list -> 'a list list = <fun>

#sublists [];;
- : 'a list list = [[]]

#sublists [1;2;3];;
- : int list list = [[]; [3]; [2]; [2; 3]; [1]; [1; 3]; [1; 2]; [1; 2; 3]]

#sublists ["a"];;
- : string list list = [[]; ["a"]]
```

7.1, p. 64

```
#type ('a,'b) btree = Leaf of 'b
#                   | Btree of ('a,'b) node
#and ('a,'b) node = {Op:'a;
```

```
#                    Son1: ('a,'b) btree;
#                    Son2: ('a,'b) btree};;
Type btree defined.
Type node defined.

#let rec nodes_and_leaves =
#    function Leaf x -> ([],[x])
#           | Btree {Op=x; Son1=s1; Son2=s2} ->
#                let (nodes1,leaves1) = nodes_and_leaves s1
#                and (nodes2,leaves2) = nodes_and_leaves s2
#                in (x::nodes1@nodes2, leaves1@leaves2);;
nodes_and_leaves : ('a, 'b) btree -> 'a list * 'b list = <fun>

#nodes_and_leaves (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : string list * int list = ["+"], [1; 2]
```

7.2, p. 64

```
#let map_btree f g = map_rec
#where rec map_rec =
#        function Leaf x -> Leaf (f x)
#               | Btree {Op=op; Son1=s1; Son2=s2} ->
#                 Btree {Op=g op; Son1=map_rec s1; Son2=map_rec s2};;
map_btree : ('a -> 'b) -> ('c -> 'd) -> ('c, 'a) btree -> ('d, 'b) btree = <fun>
```

7.3, p. 64

We need to give a functional interpretation to `btree` data constructors. We use `f` (resp. `g`) to denote the function associated to the `Leaf` (resp. `Btree`) data constructor, obtaining the following Caml definition:

```
#let btree_it f g = it_rec
#where rec it_rec =
#        function Leaf x -> f x
#               | Btree{Op=op; Son1=s1; Son2=s2} ->
#                        g op (it_rec s1) (it_rec s2);;
btree_it : ('a -> 'b) -> ('c -> 'b -> 'b -> 'b) -> ('c, 'a) btree -> 'b = <fun>

#btree_it (function x -> x)
#         (function "+" -> prefix +
#                 | _ -> raise (Failure "Unknown op"))
#         (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : int = 3
```

8.1, p. 72

```
#type ('a,'b) lisp_cons = {mutable Car:'a; mutable Cdr:'b};;
Type lisp_cons defined.

#let car p = p.Car
#and cdr p = p.Cdr
#and rplaca p v = p.Car <- v
#and rplacd p v = p.Cdr <- v;;
car : ('a, 'b) lisp_cons -> 'a = <fun>
cdr : ('a, 'b) lisp_cons -> 'b = <fun>
rplaca : ('a, 'b) lisp_cons -> 'a -> unit = <fun>
rplacd : ('a, 'b) lisp_cons -> 'b -> unit = <fun>

#let p = {Car=1; Cdr=true};;
p : (int, bool) lisp_cons = {Car=1; Cdr=true}

#rplaca p 2;;
- : unit = ()

#p;;
- : (int, bool) lisp_cons = {Car=2; Cdr=true}
```

8.2, p. 72

```
#let stamp_counter = ref 0;;
stamp_counter : int ref = ref 0

#let stamp () =
#  stamp_counter := 1 + !stamp_counter; !stamp_counter;;
stamp : unit -> int = <fun>

#stamp();;
- : int = 1

#stamp();;
- : int = 2
```

8.3, p. 72

```
#let exchange t i j =
#  let v = t.(i) in vect_assign t i t.(j); vect_assign t j v
#;;
exchange : 'a vect -> int -> int -> unit = <fun>

#let quick_sort t =
#  let rec quick lo hi =
#    if lo < hi
#    then begin
```

```
#          let i = ref lo
#          and j = ref hi
#          and p = t.(hi) in
#          while !i < !j
#           do
#            while !i < hi & t.(!i) <=. p do incr i done;
#            while !j > lo & p <=. t.(!j) do decr j done;
#            if !i < !j then exchange t !i !j
#           done;
#          exchange t hi !i;
#          quick lo (!i - 1);
#          quick (!i + 1) hi
#         end
#     else ()
#  in quick 0 (vect_length t - 1)
#;;
quick_sort : float vect -> unit = <fun>

#let a = [| 2.0; 1.5; 4.0; 0.0; 10.0; 1.0 |];;
a : float vect = [|2; 1.5; 4; 0; 10; 1|]

#quick_sort a;;
- : unit = ()

#a;;
- : float vect = [|0; 1; 1.5; 2; 4; 10|]
```

9.1, p. 76

```
#let find_succeed f = find_rec
#where rec find_rec =
#        function [] -> raise (Failure "find_succeed")
#               | x::l -> try f x; x with _ -> find_rec l;;
find_succeed : ('a -> 'b) -> 'a list -> 'a = <fun>

#let hd = function [] -> raise (Failure "empty") | x::l -> x;;
hd : 'a list -> 'a = <fun>

#find_succeed hd [[];[];[1;2];[3;4]];;
- : int list = [1; 2]
```

9.2, p. 76

```
#let map_succeed f = map_f
#where rec map_f =
#        function [] -> []
#               | h::t -> try (f h)::(map_f t)
```

```
#                          with _ ->  map_f t;;
map_succeed : ('a -> 'b) -> 'a list -> 'b list = <fun>

#map_succeed hd [[];[1];[2;3];[4;5;6]];;
- : int list = [1; 2; 4]
```

10.1, p. 81

The first function (`copy`) that we define assumes that its arguments are respectively the input and the output channel. They are assumed to be already opened.

```
#let copy inch outch =
#  (* inch and outch are supposed to be opened channels *)
#  try (* actual copying *)
#      while true
#      do output_char outch (input_char inch)
#      done
#   with End_of_file -> (* Normal termination *)
#             raise End_of_file
#      | sys__Sys_error msg -> (* Abnormal termination *)
#             prerr_endline msg;
#             raise (Failure "cp")
#      | _ -> (* Unknow exception, maybe interruption? *)
#             prerr_endline "Unknown error while copying";
#             raise (Failure "cp")
#;;
copy : in_channel -> out_channel -> unit = <fun>
```

The next function opens channels connected to its filename arguments, and calls `copy` on these channels. The advantage of dividing the code into two functions is that `copy` performs the actual work, and can be reused in different applications, while the role of `cp` is more "administrative" in the sense that it does nothing but opening and closing channels and printing possible error messages.

```
#let cp f1 f2 =
#  (* Opening channels, f1 first, then f2 *)
#  let inch =
#      try open_in f1
#      with sys__Sys_error msg ->
#                prerr_endline (f1^": "^msg);
#                raise (Failure "cp")
#         | _ -> prerr_endline ("Unknown exception while opening "^f1);
#                raise (Failure "cp")
#  in
#  let outch =
#      try open_out f2
```

```
#      with sys__Sys_error msg ->
#                close_in inch;
#                prerr_endline (f2^": "^msg);
#                raise (Failure "cp")
#         | _ -> close_in inch;
#                prerr_endline ("Unknown exception while opening "^f2);
#                raise (Failure "cp")
#  in (* Copying and then closing *)
#     try copy inch outch
#     with End_of_file -> close_in inch; close_out outch
#                         (* close_out flushes *)
#        | exc -> close_in inch; close_out outch; raise exc
#;;
cp : string -> string -> unit = <fun>
```

Let us try `cp`:

```
#cp "/etc/passwd" "/tmp/foo";;
- : unit = ()

#cp "/tmp/foo" "/foo";;
/foo: Permission denied
Uncaught exception: Failure "cp"
```

The last example failed because a regular user is not allowed to write at the root of the file system.

10.2, p. 81

As in the previous exercise, the function `count` performs the actual counting. It works on an input channel and returns a pair of integers.

```
#let count inch =
#    let chars = ref 0
#    and lines = ref 0 in
#    try
#      while true do
#        let c = input_char inch in
#          chars := !chars + 1;
#          if c = `\n` then lines := !lines + 1 else ()
#      done;
#      (!chars, !lines)
#    with End_of_file -> (!chars, !lines)
#;;
count : in_channel -> int * int = <fun>
```

The function `wc` opens a channel on its filename argument, calls `count` and prints the result.

```
#let wc f =
```

```
#    let inch =
#       try open_in f
#       with sys__Sys_error msg ->
#                prerr_endline (f^": "^msg);
#                raise (Failure "wc")
#         | _ -> prerr_endline ("Unknown exception while opening "^f);
#                raise (Failure "wc")
#  in let (chars,lines) = count inch
#     in   print_int chars;
#          print_string " characters, ";
#          print_int lines;
#          print_string " lines.\n"
#;;
wc : string -> unit = <fun>
```

Counting `/etc/passwd` gives:

```
#wc "/etc/passwd";;
27893 characters, 329 lines.
- : unit = ()
```

11.1, p. 94

Let us recall the definitions of the type `token` and of the lexical analyzer:

```
#type token =
#  PLUS | MINUS | TIMES | DIV | LPAR | RPAR
#| INT of int;;
Type token defined.
```

```
#(* Spaces *)
#let rec spaces = function
#  [< ' ' '|'\t'|'\n'; spaces _ >] -> ()
#| [< >] -> ();;
spaces : char stream -> unit = <fun>
```

```
#(* Integers *)
#let int_of_digit = function
#  '0'..'9' as c -> (int_of_char c) - (int_of_char '0')
#| _ -> raise (Failure "not a digit");;
int_of_digit : char -> int = <fun>
```

```
#let rec integer n = function
#  [< ' '0'..'9' as c; (integer (10*n + int_of_digit c)) r >] -> r
#| [< >] -> n;;
integer : int -> char stream -> int = <fun>
```

```
#(* The lexical analyzer *)
```

```
#let rec lexer s = match s with
#  [< ''('; spaces _ >] -> [< 'LPAR; lexer s >]
#| [< '')'; spaces _ >] -> [< 'RPAR; lexer s >]
#| [< ''+'; spaces _ >] -> [< 'PLUS; lexer s >]
#| [< ''-'; spaces _ >] -> [< 'MINUS; lexer s >]
#| [< ''*'; spaces _ >] -> [< 'TIMES; lexer s >]
#| [< ''/'; spaces _ >] -> [< 'DIV; lexer s >]
#| [< ''0'..'9' as c; (integer (int_of_digit c)) n; spaces _ >]
#                                -> [< 'INT n; lexer s >];;
lexer : char stream -> token stream = <fun>
```

The parser has the same shape as the grammar:

```
#let rec expr = function
#  [< 'INT n >] -> n
#| [< 'PLUS; expr e1; expr e2 >] -> e1+e2
#| [< 'MINUS; expr e1; expr e2 >] -> e1-e2
#| [< 'TIMES; expr e1; expr e2 >] -> e1*e2
#| [< 'DIV; expr e1; expr e2 >] -> e1/e2;;
expr : token stream -> int = <fun>

#expr (lexer (stream_of_string "1"));;
- : int = 1

#expr (lexer (stream_of_string "+ 1 * 2 4"));;
- : int = 9
```

11.2, p. 94

The only new function that we need is a function taking as argument a character stream, and returning the first identifier of that stream. It could be written as:

```
#let ident_buf = make_string 8 ' ';;
ident_buf : string = "        "

#let rec ident len = function
#  [< ' 'a'..'z'|'A'..'Z' as c;
#     (if len >= 8 then ident len
#      else begin
#            set_nth_char ident_buf len c;
#            ident (succ len)
#           end) s >] -> s
#| [< >] -> sub_string ident_buf 0 len;;
ident : int -> char stream -> string = <fun>
```

The lexical analyzer will first try to recognize an alphabetic character c, then put c at position 0 of `ident_buf`, and call `ident 1` on the rest of the character stream. Alphabetic characters encountered will be stored in the string buffer `ident_buf`, up to the 8th. Further alphabetic characters will be skipped. Finally, a substring of the buffer will be returned as result.

```
#let s = stream_of_string "toto 1";;
s : char stream = <abstract>

#ident 0 s;;
- : string = "toto"

#(* Let us see what remains in the stream *)
#match s with [< 'c >] -> c;;
- : char = ' '

#let s = stream_of_string "LongIdentifier ";;
s : char stream = <abstract>

#ident 0 s;;
- : string = "LongIden"

#match s with [< 'c >] -> c;;
- : char = ' '
```

The definitions of the new `token` type and of the lexical analyzer is trivial, and we shall omit them. A slightly more complex lexical analyzer recognizing identifiers (lowercase only) is given in section 13.2.1 in this part.

12.1, p. 99

```
(* main.ml *)
let chars = counter__new 0;;
let lines = counter__new 0;;

let count_file filename =
  let in_chan = open_in filename in
    try
      while true do
        let c = input_char in_chan in
          counter__incr chars;
          if c = '\n' then counter__incr lines
      done
    with End_of_file ->
      close_in in_chan
;;

for i = 1 to vect_length sys__command_line - 1 do
  count_file sys__command_line.(i)
done;
print_int (counter__read chars);
print_string " characters, ";
print_int (counter__read lines);
print_string " lines.\n";
exit 0
```

; ;

Chapter 19

Conclusion and further reading

We have not been exhaustive in the description of the Caml Light features. We only introduced general concepts in functional programming, and we have insisted on the features used in the prototyping of ASL: a tiny model of Caml Light typing and semantics. The rest of this document describes exhaustively the Caml Light language, its libraries, commands and extensions.

Description about "Caml Strong" and useful information about programming in Caml can be found in [9] and [35].

An introduction to lambda-calculus and type systems can be found in [17], [12] and [4].

The description of the implementation of call-by-value functional programming languages can be found in [19].

The implementation of lazy functional languages is described in [27][1]. An introduction to programming in lazy functional languages can be found in [5].

See also the bibliography given at the end of the book for further reading about ML and its implementation.

[1] translated in French as [28].

Bibliography

[1] H. Abelson and G.J. Sussman. *Structure and Interpretation of Computer Programs*. The MIT Press, 1985.

[2] A. Aho, R. Sethi, and J. Ullman. *Compilers: Principles, Techniques and Tools*. Addison Wesley, 1986.

[3] J. Backus. Can programming be liberated from the Von Neumann style? A functional style and its algebra of programs. In *Communications of the ACM*, volume 21, pages 133–140, 1978.

[4] H.P. Barendregt. *The Lambda-Calculus: Its Syntax and Semantics*, volume 103 of *Studies in Logic and the Foundations of Mathematics*. North-Holland, 1984.

[5] R. Bird and P. Wadler. *Introduction to Functional Programming*. Series in Computer Science. Prentice-Hall International, 1986.

[6] A. Church. *The Calculi of Lambda Conversion*. Princeton University Press, 1941.

[7] D. Clément, J. Despeyroux, T. Despeyroux, and G. Kahn. A simple applicative language: Mini-ML. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 13–27, 1986.

[8] G. Cousineau, P.-L. Curien, and M. Mauny. The categorical abstract machine. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 50–64. Springer Verlag, 1985.

[9] G. Cousineau and G. Huet. The CAML primer. Technical Report 122, INRIA, 1990.

[10] N. De Bruijn. *Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation*. Indag. Math., 1962.

[11] M. Gordon, R. Milner, L. Morris, M. Newey, and C. Wadworth. A metalanguage for interactive proofs in LCF. In *Proceedings of ACM Symposium on Principles of Programming Languages*, pages 119–130, 1978.

[12] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus*. London Mathematical Society, Student Texts. Cambridge University Press, 1986.

[13] C.A.R. Hoare. Quicksort. *Computer Journal*, 5(1), 1962.

[14] W.A. Howard. *The formulae-as-type notion of construction*, pages 479–490. Academic Press, 1980.

[15] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, 1990.

[16] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of ACM Conference on Compiler Construction*, pages 58–69, 1984.

[17] J.-L. Krivine. *Lambda-calcul, types et modèles*. Etudes et recherches en informatique. Masson, 1990.

[18] P. Landin. The next 700 programming languages. In *Communications of the ACM*, volume 9, pages 157–164, 1966.

[19] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[20] J. MacCarthy. *Lisp 1.5 Programmer's Manual*. MIT Press, Cambridge, Mass., 1962.

[21] D. MacQueen. Modules for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1984.

[22] M. Mauny. Functional programming using CAML. Technical Report 129, INRIA, 1991.

[23] M. Mauny and D. de Rauglaudre. Parsers in ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1992.

[24] M. Mauny and A. Suárez. Implementing functional languages in the categorical abstract machine. In *Proceedings of the ACM International Conference on Lisp and Functional Programming*, pages 266–278, 1986.

[25] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[26] R. Milner. A proposal for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1987.

[27] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages*. Series in Computer Science. Prentice-Hall International, 1987.

[28] S.L. Peyton-Jones. *Mise en œuvre des langages fonctionnels de programmation*. Manuels Informatiques Masson. Masson, 1990.

[29] J. Rees and W. Clinger. The revised^3 report on the algorithmic language Scheme. In *SIGPLAN Notices*, volume 21, 1987.

[30] J. A. Robinson. Computational logic: the unification computation. In *Machine Intelligence*, volume 6 of *American Elsevier*. B. Meltzer and D. Mitchie (Eds), 1971.

[31] R. Sedgewick. *Algorithms*. Addison Wesley, 1988.

[32] D. Turner. SASL language manual. Technical report, St Andrews University, 1976.

[33] D. Turner. *Recursion equations as a programming language*, pages 1–28. Cambridge University Press, 1982.

[34] D. Turner. Miranda: a non-strict functional language with polymorphic types. In *Functional Programming Languages and Computer Architecture*, number 201 in Lecture Notes in Computer Science, pages 1–16. Springer Verlag, 1985.

[35] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.

Part III

The Caml Light language reference manual

Chapter 20

The Caml Light language reference manual

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 part II, *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

Even though there is currently only one implementation of the Caml Light language, this document carefully distinguishes the language and its implementation(s). Implementations can provide extra language constructs; moreover, all points left unspecified in this reference manual can be interpreted as the language implementor wishes. All these implementation-dependent features are of course subject to change in future releases; only the features specified in this reference manual can be relied upon.

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.

20.1 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:

`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.

20.2 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:

global-name ::= *ident*
| $ident_1$ `__` $ident_2$

The form $ident_1$ `__` $ident_2$ 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.

20.3 Values

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

20.3.1 Base values

Integer numbers

Integer values are integer numbers from -2^{31} to $2^{31} - 1$, that is -2147483648 to 2147483647. 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.

20.3.2 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.

20.3.3 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.

20.3.4 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.

20.3.5 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:

`false`	the boolean false
`true`	the boolean true
`()`	the "unit" value
`[]`	the empty list

20.3.6 Functions

Functional values are mappings from values to values.

20.4 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 same type as the type constructor *typeconstr*, where the parameter to *typeconstr* has been substituted by the type *typexpr*.

The type expression $(typexpr_1, \ldots, typexpr_n)$ *typeconstr*, where *typeconstr* is a type constructor with n parameters, denotes the same type as the type constructor *typeconstr*, where the parameters to *typconstr* have been substituted by the types $typexpr_1$ through $typexpr_n$, respectively.

20.5 Constants

constant ::= *integer-literal*
| *float-literal*
| *char-literal*
| *string-literal*
| *cconstr*

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

20.6 Patterns

pattern ::= *ident*
| `_`
| *pattern* `as` *ident*
| `(` *pattern* `)`
| `(` *pattern* `:` *typexpr* `)`
| *pattern* `|` *pattern*
| *constant*
| *ncconstr pattern*
| *pattern* `,` *pattern* {`,` *pattern*}
| `{` *label* `=` *pattern* {`;` *label* `=` *pattern*} `}`
| `[]`
| `[` *pattern* {`;` *pattern*} `]`
| *pattern* `::` *pattern*

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

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

Patterns are templates that allow selecting data structures of a given shape, and binding identifiers to components of the data structure. This selection operation is called pattern matching; its outcome is either "this value does not match this pattern", or "this value matches this pattern, resulting in the following bindings of 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*$_1$ ... *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.

20.7 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`	–

20.7.1 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 20.2.

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:

$$\begin{array}{lll} \texttt{fun} & pattern_1^1 \ \ldots \ pattern_1^m & \texttt{->} \ \ expr_1 \\ \texttt{|} & \ldots & \\ \texttt{|} & pattern_n^1 \ \ldots \ pattern_n^m & \texttt{->} \ \ expr_n \end{array}$$

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 $pattern_i^1 \ \ldots \ pattern_i^m$, for i from 1 to n. If one of these matchings succeeds, that is if the value v_j matches the pattern $pattern_i^j$ 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 provided arguments. 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`:

$$\begin{array}{lll} \texttt{function} & pattern_1 & \texttt{->} \ \ expr_1 \\ \texttt{|} & \ldots & \\ \texttt{|} & pattern_n & \texttt{->} \ \ expr_n \end{array}$$

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

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

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

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:

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

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.

20.7.2 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

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

matches the value of *expr* against the patterns $pattern_1$ to $pattern_n$. If the matching against $pattern_i$ succeeds, the associated expression $expr_i$ is evaluated, and its value becomes the value of the whole `match` expression. The evaluation of $expr_i$ takes place in an environment enriched by the bindings performed during matching. If several patterns match the value of *expr*, the one that occurs first in the `match` expression is selected. If none of the patterns match the value of *expr*, the exception `Match_failure` is raised.

Boolean operators

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

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

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

`if` $expr_1$ `then true else` $expr_2$.

Loops

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

The expression `for` *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

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

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

20.7.3 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 $\{label_1 = v_1; \ldots; label_n = v_n\}$, 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.

20.7.4 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 20.7.1. 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 null. The result is unspecified if either argument is negative.
`mod`	`prefix mod`	Integer modulus. Raise `Division_by_zero` if second argument is null. 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. The result is unspecified if second argument is null.
`@`	`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 disequality test.
`==`	`prefix ==`	Physical equality test.
`!=`	`prefix !=`	Physical disequality 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 21 for a more precise description of the behavior of the operators above.

20.8 Global definitions

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

20.8.1 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-decl*$_1$ `;` ... `;` *label-decl*$_n$ `}` defines a record type. The label declarations *label-decl*$_1$, ..., *label-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.

20.8.2 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.

20.9 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 20.2. 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.

20.10 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 20.7.1. 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 20.8.1 and 20.8.2. 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 20.9. 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.

20.11 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 20.2.

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 20.8.1 and 20.8.2. 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 20.9. 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.

Part IV

The Caml Light library

Chapter 21

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 25). 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`
`-O fast`	`io`, `eq`, `int`, `float`, `ref`, `pair`, `list`, `fvect`, `fchar`, `fstring`, `bool`, `exc`, `stream`
`-O none`	none

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.

21.1 `bool`: Boolean operations

The boolean conjunction is written `e1 & e2`. The boolean disjunction 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.

21.2 `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.

21.3 `eq`: Equality functions

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

`e1 = e2` tests for structural equality of `e1` and `e2`. Mutable structures (e.g., references) 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
```

`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 ==`.

21.4 `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 some library functions to signal that the given arguments do not make sense.

```
exception Failure of string
```

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

```
exception Not_found
```

Raised by library search functions when the required 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.

21.5 `fchar`: Character operations, without sanity checks

This module implements the same functions as the `char` module, but does not perform boundary 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.

21.6 `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. The result is unpredictable if the dividend is 0.0.

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

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

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

Negation of `eq_float`.

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

21.7 `fstring`: String operations, without sanity checks

This module implements the same functions as the `string` module, but does not perform boundary 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.

21.8 `fvect`: Operations on arrays, without sanity checks

This module implements the same functions as the `vect` module, but does not perform boundary 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.

21.9 `int`: Operations on integers

Integers are 31 bits wide. All operations are taken modulo 2^{31}. 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
```

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, but faster.

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

Negation of `eq_int`.

```
value prefix < : int -> int -> bool
value lt_int : int -> int -> bool
value prefix > : int -> int -> bool
value gt_int : int -> int -> bool
value prefix <= : int -> int -> bool
value le_int : int -> int -> bool
value prefix >= : int -> int -> bool
value ge_int : int -> int -> bool
```

Usual comparisons between integers.

```
value min : int -> int -> int
```

Return the smaller of the arguments.

```
value max : int -> int -> int
```

Return the greater of the arguments.

```
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 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.

21.10 `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` at the right times.

`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 (except functions) 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. Raise `Invalid_argument "extern: functional value"` when encountering a function value.

`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.

`value close_out : out_channel -> unit`

Close the given channel, flushing all buffered write operations. The behavior is unspecified if any of the above functions 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, positioned 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 before the line is complete.

```
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
```

`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`, 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`.

```
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.

21.11 `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` in turn 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 binding exists for the given key.

21.12 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.

21.13 ref: Operations on references

```
type 'a ref = ref of mutable '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`.

21.14 `stream`: Operations on streams

`type 'a stream`

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

`exception Parse_failure`

Raised by parsers when none of the first component of the stream patterns matches.

`exception Parse_error`

Raised by parsers when the first component of a stream pattern matches, but one of the following components does not match.

`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 elements 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 element encountered. The stream `s` is emptied.

`value stream_check : ('a -> bool) -> 'a stream -> 'a`

`stream_check p` returns the parser which returns the first element of the stream if the predicate `p` returns `true` on this element, and raises `Parse_failure` otherwise.

`value end_of_stream : 'a stream -> unit`

Return `()` if 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.

21.15 `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 – `(string_length s - 1)`.

```
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)`.

```
value prefix ^ : string -> string -> string
```

`s1 ^ s2` returns a new string containing the concatenation of the strings `s1` and `s2`.

```
value sub_string : string -> int -> int -> string
```

`sub_string s start len` returns a new 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 new string of length `n`. The string initially contains arbitrary characters.

```
value make_string : int -> char -> string
```

`make_string n c` returns a new 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,

`-2` if `s1` is a prefix of `s2`,

`2` if `s2` is a prefix of `s1`,

`-1` otherwise if `s1` is lexicographically before `s2`,

`1` otherwise 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.

21.16 `vect`: Operations on arrays

```
value vect_length : 'a vect -> int
```

Return the length (number of elements) of the given array.

```
value vect_item : 'a vect -> int -> 'a
```

`vect_item v n` returns the element number `n` of array `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 array `v` in place, replacing element number `n` with `x`. Raise `Invalid_argument "vect_assign"` if `n` is outside the range 0 – `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 array of length `n`, initialized with `x`. All the elements of this new array are initially physically equal to `x`, in the sense of the `==` operator.

```
value make_matrix : int -> int -> 'a -> 'a vect vect
```

`make_matrix dimx dimy e` returns a two-dimensional array (an array of arrays) with first dimension `dimx` and second dimension `dimy`. All the elements of this new matrix are initially physically equal to `e`, in the sense of the `==` operator. 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 array containing the concatenation of arrays `v1` and `v2`.

```
value sub_vect : 'a vect -> int -> int -> 'a vect
```

`sub_vect v start len` returns a fresh array of length `len`, containing the elements number `start` to `start + len - 1` of array `v`. Raise `Invalid_argument "sub_vect"` if `start` and `len` do not designate a valid subarray of `v`; that is, if `start < 0`, or `len < 0`, or `start + len > vect_length v`.

```
value fill_vect : 'a vect -> int -> int -> 'a -> unit
```

`fill_vect v ofs len x` modifies array `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 subarray of `v`.

```
value blit_vect : 'a vect -> int -> 'a vect -> int -> int -> unit
```

`blit_vect v1 o1 v2 o2 len` copies `len` elements from array `v1`, starting at element number `o1`, to array `v2`, starting at element number `o2`. It works correctly even if `v1` and `v2` are the same array, and the source and destination chunks overlap. Raise `Invalid_argument "blit_vect"` if `o1` and `len` do not designate a valid subarray of `v1`, or if `o2` and `len` do not designate a valid subarray of `v2`.

```
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 array 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 array 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 22

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.

22.1 `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 4 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: (`foo` is assumed to be the command name)

`foo -flag` (a unit option)

```
foo -int 1                  (an int option with argument 1)
foo -string foobar          (a string option with argument "foobar")
foo -float 12.34            (a float option with argument 12.34)
foo 1 2 3                   (three anonymous arguments: "1", "2", and "3")
foo 1 -flag 3 -string bar   (one anonymous argument, a unit option, one anonymous
                             argument, 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 `-`, or 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 message` to reject invalid arguments.

22.2 `filename`: Operations on file names

```
value current_dir_name : string
```

The conventional name of 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` otherwise.

```
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 and base file name, respectively.
`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`.

22.3 `hashtbl`: Hash tables and hash functions

Hash tables are hashed association tables, with in-place modification.

```
type ('a, 'b) t mutable
```

The type of association 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. Apparently produces better results 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` binds the value `y` to key `x` 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. The order in which the bindings are passed to `f` is unpredictable. The same binding can be presented several times to `f`.

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 for `m` and `n` mean 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. Parameters `m` and `n` allow to trade accuracy for speed.

22.4 `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 that reads from the input channel `inchan`, at the current reading position.

```
value create_lexer_string : string -> lexbuf
```

Create a lexer buffer that 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 three parameter 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 buffer of the first character of the matched string. The first character read has position 0.

```
value get_lexeme_end : lexbuf -> int
```

`get_lexeme_start lexbuf` returns the position in the input buffer of the character following the last character of the matched string. The first character read has position 0.

22.5 `parsing`: The run-time library for parsers generated by mlyacc

```
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 read has position 0.

```
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.

22.6 printexc: A catch-all exception handler

```
value f: ('a -> 'b) -> 'a -> 'b
```

`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 `f main ()`, where `main`, with type `unit->unit`, is the entry point of a standalone program. This catches and prints stray exceptions. For this functional to work properly, the program must be linked with the `-g` option.

22.7 printf: Formatting printing functions

```
value fprintf: out_channel -> string -> '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 the conversion and printing of one argument. Conversion specifications consist of 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`.

Refer to the C library `printf` function for the meaning of flags and field width specifiers. The exception `Invalid_argument` is raised if the types of the provided arguments do not

match the format. The exception is also raised if too many arguments are provided. If too few arguments are provided, printing stops just before converting the first missing argument. This function is not type-safe, since it returns an object with type `'a`, that can be abused in many ways. It is recommended to always use `fprintf` inside a sequence, as in `fprintf "%d" n; ()`.

```
value printf: string -> 'a
```

Same as `fprintf`, but output on `std_out`.

```
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` from module `io`.

```
value print: string -> unit
```

Print the given string on `std_out`, without any formatting. This is the same function as `print_string` from module `io`.

22.8 queue: First-in, first-out queues

This module implements queues (FIFOs), with in-place modification.

```
type 'a t mutable
```

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.

22.9 `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 int : int -> int
```

`random__int bound` returns a random number between 0 (inclusive) and `bound` (exclusive). `bound` must be smaller than 2^{30}.

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

`random__float` returns a random number between 0 (inclusive) and `bound` (exclusive).

22.10 `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.

22.11 `stack`: Last-in, first-out stacks

This module implements stacks (LIFOs), with in-place modification.

```
type 'a t mutable
```

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.

22.12 `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.

```
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.

```
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.

```
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 23

The graphics library

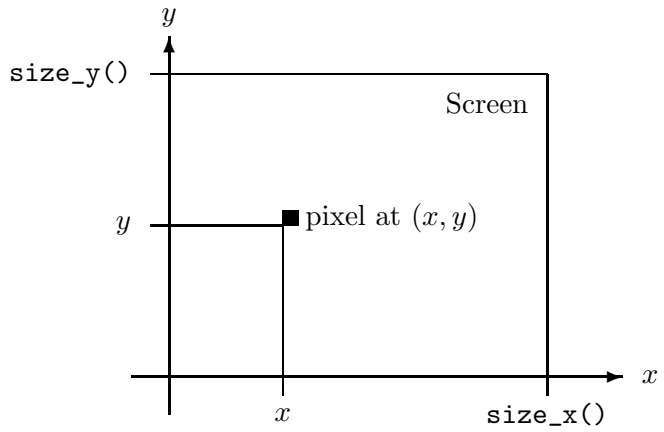
This chapter describes the portable graphics primitives that come standard in the implementation of Caml Light on micro-computers.

Unix: On Unix workstations running the X windows system, these graphics primitives have been implemented on top of the RT toolkit. See the directory `contrib/libgraph` in the distribution for information on how to link with the RT toolkit and the library implementing the graphics primitives. Drawing takes place in a separate X 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.

PC86: The 86 PC version supports the CGA, EGA and VGA graphics cards. The argument to `open_graph` is interpreted as follows:

Argument to `open_graph`	Graphic mode
`"cga"`	320x200, 4 colors
`"ega_low"`	640x200, 16 colors
`"ega_high"`	640x350, 16 colors
`"vga_low"`	640x200, 16 colors
`"vga_med"`	640x350, 16 colors
`"vga_high"`	640x480, 16 colors
anything else	highest possible resolution

PC386: 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 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).

23.1 `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 -> int
```

`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 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 set_color : color -> unit
```

Set the current drawing color.

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.

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, for obscure technical reasons, the frequency is rounded to the nearest note in the equal-tempered scale.

Part V

Extensions to the Caml Light language

Chapter 24

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.

24.1 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-pattern* `->` *expr* {`|` *stream-pattern* `->` *expr*}
	|	`match` *expr* `with` *stream-pattern* `->` *expr* {`|` *stream-pattern* `->` *expr*}
stream-component	::=	`'` *expr*
	|	*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 chapter 11 in part II 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.

24.2 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.

24.3 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.

24.4 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}`.

24.5 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 VI

The Caml Light commands

Chapter 25

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.

25.1 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 produced 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` $arg_1\ arg_2\ \ldots\ arg_n$

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

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

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

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` $arg_1\ arg_2\ \ldots\ arg_n$

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.)

25.2 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.

`-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 30.

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

PC: This option is not implemented.

Mac: This option is not implemented.

`-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` Add debugging information at the end of the executable bytecode file produced by the linking phase. This option has no effect on the compilation phase. Debugging information is required in particular by the catch-all exceptions handlers provided by the module `printexc` in the standard library.

`-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 24.)

`-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 21 for a complete listing of the modules in the `cautious` and `fast` sets.

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

25.3 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.

25.4 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.

Expression of type t_1 cannot be 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.

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 30, 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 26

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 extends 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`.

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-dot", 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 5.0 distribution.

With the 8086 version, the parsing, compilation or evaluation of the current phrase can be interrupted by pressing `ctrl-break`. This goes back to the `#` prompt. Pressing `ctrl-C` interrupts input, but does not stop a phrase that performs no input-output.

With the 80386 version, pressing `ctrl-C` or `ctrl-break` interrupts the system at any point: during parsing as well as during compilation or evaluation.

26.1 Options

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

`-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 24.

`-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 21 for a complete listing of the modules in the `cautious` and `fast` sets.

26.2 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.

`quit ()`
: Exit the toplevel loop and terminate the Caml Light command.

`include "`*filename*`"`
: Read, compile and execute source phrases from the file *filename*. The `.ml` extension is automatically added to *filename*, 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.

`load "`*filename*`"`
: Load in memory the source code for a module implementation. Read, compile and execute source phrases from file *filename*. The `.ml` extension is automatically added to *filename*, if not present. The `load` function behaves much like `include`, except that a new module is created, with name the base name of *filename*. Global identifiers defined in file *filename* 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.

`load_object "`*filename*`"`
: Load in memory the compiled bytecode contained in file *filename*. The `.zo` extension is automatically added to *filename*, if not present. The bytecode file has been produced by the standalone compiler `camlc`. Global identifiers defined in file *filename* are entered in their own module, not in the `top` module, just as with the `load` function.

`trace "`*function-name*`"`
: 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.

The function name cannot be a qualified identifier. Only unqualified identifiers are recognized. Hence, only functions belonging to one of the opened modules can be traced.

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

The function name cannot be a qualified identifier. Only unqualified identifiers are recognized. Hence, only functions belonging to one of the opened modules can be traced.

`cd "`*directory-name*`"`
Set the current working directory.

`gc ()`
Finish the current garbage collection cycle, and return the amount of free space in the heap, in bytes. If you want to perform a complete GC cycle, call this function twice.

26.3 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 of the reference manual.) 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`.

26.4 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 26.3 above.

Corrupted compiled interface file *filename*
: See section 25.4.

Expression of type t_1 **cannot be used with type** t_2
: See section 25.4.

26.5 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.

PC: This command is not available in the PC versions.

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.

26.6 Options

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

`-custom`
: Link in "custom runtime" mode. See the corresponding option for `camlc`, in chapter 25.

`-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. The default is `camltop.out`.

Chapter 27

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.

27.1 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_2 ... 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 25, 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_2 ... arg_n

works exactly as

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

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

27.2 Options

The following command-line option is recognized by `camlrun`. (There are additional options to control the behavior of the garbage collector, but they are not for the casual user.)

`-v` Cause `camlrun` to print various diagnostic messages relative to memory allocation and garbage collection. Useful to debug memory-related problems.

27.3 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 28

The librarian (camllibr)

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

28.1 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.)

28.2 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`.

28.3 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 29

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 chapter 11, 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.

PC86: These commands are not available in the 8086 PC version.

29.1 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 21.)

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.)

29.2 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.

29.2.1 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.

29.2.2 Entry points

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

29.2.3 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).

29.2.4 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.

29.3 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`.

29.4 Syntax of grammar definitions

Grammar definitions have the following format:

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

Comments are enclosed between `/*` and `*/` pairs, as in C.

29.4.1 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.

29.4.2 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.

29.4.3 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.

29.5 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.

29.6 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
%right PLUS MINUS       /* lowest precedence */
%right 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 30

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.

Mac: This facility is not implemented in the Macintosh version.

PC: This facility is not implemented in the PC versions.

30.1 Overview and compilation information

30.1.1 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.

30.1.2 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:

`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`).

30.1.3 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.)

30.2 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).

30.2.1 Integer values

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

30.2.2 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.

30.2.3 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.

30.3 Representation of Caml Light data types

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

30.3.1 Atomic types

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

30.3.2 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.

30.3.3 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`

30.4 Operations on values

30.4.1 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.

30.4.2 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

30.4.3 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 code part of the closure v.
- `string_length`(v) returns the length (number of characters) of the string v.
- `Byte`(v, n) returns the n^{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).

30.4.4 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_size`), and large blocks (with size greater than to `Max_young_size`). The constant `Max_young_size` 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_size` 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_size` 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_size` 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_size`. (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 **`).

30.4.5 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.

30.5 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:*

$$\texttt{Field}(v, n) = v_n;$$

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

$$\texttt{initialize}(\texttt{\&Field}(v, n), v_n);$$

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*

$$\texttt{Field}(v, n) = v';$$

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:*

$$\texttt{initialize}(\texttt{\&Field}(v, n), v');$$

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

$$\texttt{modify}(\texttt{\&Field}(v, n), v');$$

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.

30.6 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 VII

Appendix

Chapter 31

Further reading

For the interested reader, we list below some references to books and reports related (sometimes loosely) to Caml Light.

31.1 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. Most 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.

- Lawrence C. Paulson. *ML for the working programmer.* Cambridge University Press, 1991.

  An excellent introduction to programming in Standard ML. Develops a theorem prover as a complete example. Contains a presentation of the module system of Standard ML.

- Chris Reade. *Elements of Functional Programming.* Addison-Wesley, 1989.

  An introduction to Standard ML, with sections on denotational semantics and lambda-calculus.

- 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 pour 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.

- Ake Wikstrom. *Functional programming using Standard ML.* Prentice-Hall, 1987.

  A first course in programming, taught in Standard ML. Intended for absolute beginners; slow-paced for the others.

- 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.

31.2 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.

- 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 horribly terse, so even readers that 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).

31.3 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.

  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.

Index to the library

Index of keywords