Functional Programming using CAML Light

Michel Mauny

Domaine de Voluceau
BP 105, F-78153 Le Chesnay Cedex
E-mail: Michel.Mauny@inria.fr

©INRIA 1990-91.

May 13, 1991.

Programmation fonctionnelle en CAML Light

Michel Mauny

Résumé

Ces notes de cours forment une introduction à la programmation fonctionnelle. Elles ont été rédigées lors d'un enseignement dispensé à l'Université de Salerne (Italie) au printemps 1990. Elles peuvent servir de support à un enseignement de programmation fonctionnelle à des étudiants ayant eu un premier contact avec la programmatiom. Cependant, aucune connaissance préalable n'est strictement requise.

A travers l'étude du langage fonctionnel CAML, nous présentons la synthèse de types polymorphes mise en œuvre par l'algorithme de Milner. Nous présentons aussi un modèle simple d'éxécution des langages fonctionnels: une version simplifiée de la Machine Abstraite Catégorique (CAM). Ces notes se terminent par le prototypage complet d'un petit langage fonctionnel.

Functional Programming using CAML Light

Michel Mauny

Abstract

These course notes represent an introduction to functional programming. They have been written while the author was teaching at the University of Salerno (Italy) during spring 1990. These notes can be used for teaching functional programming to students who already had a first contact with Computer Science. However, no prerequisite is strictly required.

Through the study of the CAML functional programming, we present polymorphic type synthesis implemented by Milner's algorithm. We also present a simple execution model for functional languages: a simplified version of the Categorical Abstract Machine (CAM). These notes end by the complete prototyping of a simple functional language.

Contents

1 Introduction **1**

I Functional Programming **3**

2 Functional languages **5**
2.1 History of functional languages 6
2.2 The ML family 6
2.3 The Miranda family 7

3 Fundamental concepts **9**
3.1 Toplevel loop 9
3.2 Evaluation: from expressions to values 9
3.3 Types 11
3.4 Functions 12
3.5 Definitions 13
3.6 Partial applications 14

4 Basic types **17**
4.1 Numbers 17
4.2 Boolean values 20
4.2.1 Equality 21
4.2.2 Logical operators 22
4.3 Strings and characters 22
4.4 Tuples 23
4.4.1 Constructing tuples 23
4.4.2 Extracting pair components 23
4.5 Patterns and pattern-matching 24
4.6 Functions 26
4.6.1 Functional composition 26
4.6.2 Currying 27

5 Lists **29**
5.1 Building lists 29
5.2 Extracting elements from lists: pattern-matching 30

5.3 Functions over lists 31

6 User-defined types **33**
6.1 Product types 33
6.1.1 Defining product types 33
6.1.2 Extracting products components 34
6.1.3 Parameterized product types 35
6.2 Sum types 35
6.2.1 Defining sum types 35
6.2.2 Extracting sum components 37
6.2.3 Recursive types 38
6.2.4 Parameterized sum types 39
6.2.5 Data constructors and functions 39
6.2.6 Degenerate cases: when sums meet products 40
6.3 Summary 40

II CAML Light Specifics **43**

7 Mutable data structures **45**
7.0.1 User-defined mutable data structures 45
7.0.2 The `ref` type 47
7.0.3 Polymorphism and mutable data structures 47

8 Escaping from computations: exceptions **49**
8.1 Exceptions 49
8.2 Raising an exception 50
8.3 Trapping exceptions 50
8.4 Polymorphism and exceptions 52

9 Standalone programs and separate compilation **53**
9.1 Standalone programs 53
9.2 Programs in several files 54
9.3 Abstraction 56

III A Complete Example **59**

10 ASL: A Small Language **61**
10.1 ASL abstract syntax trees 61
10.2 Parsing ASL programs 62

11 Untyped semantics of ASL programs **67**
11.1 Semantic values 67
11.2 Semantic functions 68
11.3 Examples 69

12 Encoding recursion **71**
12.1 Fixpoint combinators . . . 71
12.2 Recursion as a primitive construct . . . 72

13 Static typing, polymorphism and type synthesis **73**
13.1 The type system . . . 73
13.2 The algorithm . . . 77
13.3 The ASL type-synthesizer . . . 79
13.3.1 Representation of ASL types and type schemes . . . 79
13.3.2 Destructive unification of ASL types . . . 80
13.3.3 Representation of typing environments . . . 81
13.3.4 From types to type schemes: generalisation . . . 81
13.3.5 From type schemes to types: generic instanciation . . . 83
13.3.6 The ASL type synthesizer . . . 83
13.3.7 Typing, trapping type clashes and printing ASL types . . . 84
13.3.8 Typing ASL programs . . . 85
13.3.9 Typing and recursion . . . 86

14 Compiling ASL to an abstract machine code **89**
14.1 The Abstract Machine . . . 89
14.2 Compiling ASL programs into CAM code . . . 92
14.3 Execution of CAM code . . . 94

15 Answers to exercises **97**

16 Conclusion and further reading **105**

A The standard library **109**
A.1 Module `toplvl`: System functions for interactive use . . . 109
A.2 Module `int`: Operations on integers . . . 109
A.3 Module `float`: Operations of floating-point numbers . . . 111
A.4 Module `eq`: Equality functions . . . 112
A.5 Module `bool`: Boolean operations . . . 113
A.6 Module `exc`: Exceptions . . . 113
A.7 Module `cchar`: Character operations, with sanity checks. . . . 113
A.8 Module `cstring`: String operations, with sanity checks. . . . 114
A.9 Module `pair`: Operations on pairs . . . 116
A.10 Module `cvect`: Operations on vectors, with sanity checks. . . . 117
A.11 Module `list`: Operation on lists . . . 118
A.12 Module `ref`: Operations on references . . . 120
A.13 Module `io`: Buffered input-output . . . 120
A.14 Module `fchar`: Character operations, without sanity checks. . . . 124
A.15 Module `fstring`: String operations, without sanity checks. . . . 124
A.16 Module `fvect`: Operations on vectors, without sanity checks. . . . 125
A.17 Module `unix`: Some Unix system calls. . . . 127
A.18 Module `signal`: Operations on Unix signals. . . . 128

Chapter 1

Introduction

These course notes are an introduction to functional programming. The course was first given at Salerno University (Italy) in spring 1990.

Through the study of the CAML functional language ([Cousineau and Huet, 1990], [Weis *et al.*, 1990]), we present static type checking, polymorphism and type synthesis as implemented by Milner's algorithm ([Milner, 1978]). We also present a simple stack execution model of functional languages: a simplified version of the Categorical Abstract Machine (CAM) ([Cousineau *et al.*, 1985], [Mauny and Suárez, 1986]).

Functionality and type systems of functional languages such as CAML allow rapid implementation of prototypes: the programmer may define his/her own data structures as well as functions manipulating them with the security provided by a strict type verification.

After generalities concerning functional languages (part 1), we present some features of the CAML programming language, without being exhaustive (part 2). The third part is dedicated to a prototype implementation of a small and simple functional programming language.

These notes have been slightly edited by Xavier Leroy, to adapt them to the CAML Light implementation, which lacks some of the more advanced features of CAML. The "CAML Strong" version of this document is published as a INRIA technical report [Mauny, 1991].

These notes certainly contain imperfections. Suggestions and "bug reports" should be sent (preferably) by electronic mail to `Michel.Mauny@inria.fr`.

Warning: The programs and remarks (especially contained in parts 2 and 3) of this document might not be valid in CAML Light versions anterior or posterior to 0.3.

Acknowledgements

I thank Giancarlo Nota for his invitation in Salerno and for making my stay there as nice as possible.

Thanks are also due to Amy Felty for correcting my English and to Thérèse Hardin and Didier Rémy for reading a previous version of these notes.

Part I

Functional Programming

Chapter 2

Functional languages

Programming languages are said to be *functional* when their basic component is the notion of *function* and their essential control structure is *function application*. For example, the Lisp language [MacCarthy, 1962] 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.

3. the notion of *functional abstraction*: the name *successor* is now used in order to represent the functional value mapping any integer n to the integer resulting of the computation of $n+1$. We would break common mathematical rules if we were allowed to change that text.

Now, if we take a look at the corresponding notions in the Lisp language[1], we can see that Lisp respects none of them.

[1] We are here talking about the traditional definition of the Lisp language: some modern versions of Lisp (e.g. Scheme) may be seen as "almost functional" (they still do not possess a type system, and their toplevel binding rule is still dynamic).

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*.

2.1 History of functional languages

Some historical points:

- 1930: Alonzo Church developed the λ-calculus [Church, 1941] 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 [MacCarthy, 1962] whose programs have some similarities with terms of 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 [Landin, 1966] 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 [Backus, 1978] and a framework in which it is possible to reason about programs. The main particularity of FP is that there are no variable names in FP programs.
- 1978: R. Milner proposes a language called ML [Gordon *et al.*, 1978], 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 [Turner, 1985] 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.

2.2 The ML family

ML languages are based on a sugared[2] 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.

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

The LCF proof system appeared in 1972 in Stanford (Stanford LCF). It was further developed in 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.

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

In 1984, a commitee 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 ([Milner, 1987]). This core language was completed by a module facility designed by D. MacQueen ([MacQueen, 1984]).

Since 1984, the CAML language has been under design in a collaboration between INRIA and LIENS[3]). Its first release appeared in 1987. The main implementors of CAML were Ascánder Suárez, Pierre Weis and the author. CAML Light is a smaller, more portable implementation of the core CAML language, developed by Xavier Leroy.

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

2.3 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 [Turner, 1976] (1976) language developed by D. Turner. SASL and its successors (KRC [Turner, 1982] 1981, Miranda [Turner, 1985] 1985 and Haskell [Hudak and Wadler, 1990] 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 [Johnsson, 1984].

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

Chapter 3

Fundamental 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.

3.1 Toplevel loop

The first contact with CAML is through its interactive aspect. 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 0.3 -- Wed May  1 17:21:21 MET DST 1991

#
```

The "`#`" character is CAML's prompt. 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 not starting by the `#` character are either CAML responses or (possibly) illegal input.

3.2 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.

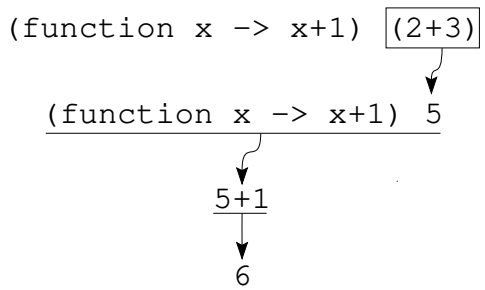
The process of evaluation of a CAML expression can be seen as transforming this expression until no further transformation is allowed. These transformations must preserve semantics. For example, if the expression "`1+2`" has the mathematical object 3 as semantics, then the result "`3`" must have the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

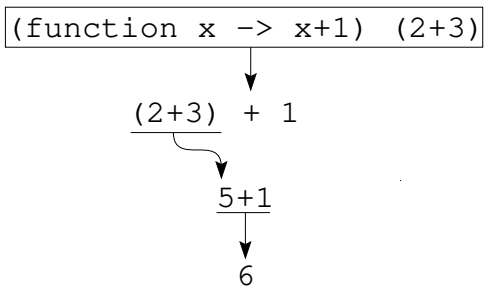
Let us consider another example: the application of the successor function to `2+3`:

```
#(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 obtain successively:



or:



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:

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

  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 Light uses call-by-value (arguments are evaluated before being passed to functions), and evaluates arguments before functions (i.e. CAML Light uses a right-to-left call-by-value evaluation strategy).

3.3 Types

Types and their checking/synthesis are crucial in functional programming. They provide an indication about 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 that phrase. First of all, it does *syntax analysis* which consists in checking whether the phrase belongs to the language. The second analysis performed is *type analysis*: the system tries to assign a type to each subexpression of the current phrase, and to synthetize 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. For example, here is a syntax error (a parenthesis is missing):

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

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);;
>                       ^^^^
>>> 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, during execution, getting a "`Bus error`" or "`Illegal instruction`" message, followed by a `core dumped` to use with our favorite debugger!

Once a sensible type has been deduced for the expression, then the evaluation mechanism (i.e. 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 logical errors are trapped and signalled 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), static type-analysis provides more efficient machine code.

3.4 Functions

The most important kind of values in functional programming are functional values. Mathematically, a function f of type $A \rightarrow B$ 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. In some mathematics texts one often finds the phrase "the function $f(x)$". This is because in these texts, functions are not considered as values. If that was the case, one would write "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)`.

3.5 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 irrelevant: changing that name 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 that function to, say, `1+2`, we will evaluate the expression `y+1` *where* `y` is the value of `1+2`. We could write that intermediate expression as:

```
y+1 where y=1+2;;
```

or:

```
let y=1+2 in y+1;;
```

These two expressions are legal CAML expressions, and the `let` and `where` are indeed widely used in CAML programs.

The `let` and `where` constructs represent *local bindings of values to identifiers*. They also introduce *sharing* of (possibly time-consuming) evaluations. 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
successor = <fun>
#let square = function x -> x*x;;
square : int -> int
square = <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
square = <fun>
```

or (closer to the mathematical notation) by:

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

3.6 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
f = <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
addition = <fun>
```

which we could have written:

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

And then define the successor function by:

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

Now, we may use our `successor` function:

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

Exercises

3.1 Give examples of functions with the following types:

```
(int -> int) -> int
int -> (int -> int)
(int -> int) -> (int -> int)
```

3.2 We have seen that the names of formal parameters were 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?

3.3 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 4

Basic types

We examine in this chapter the CAML basic types.

4.1 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 makes `*` bind tighter than `+`, and so on. In doubt, write extra parentheses.

So far, we have not seen the type of these arithmetic operations.]] All of them wait for two integer arguments; so, they are functions taking one integer and returning a function from integers to integers. The 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, without special syntax, including:

`add_float`	addition
`sub_float`	subtraction
`minus_float`	opposite
`mult_float`	multiplication
`div_float`	division
`sqrt`	square root
`exp`, `log`	
`sin`, `cos`,...	

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
#add_float (float_of_int 1) 2.3;;
- : float
- = 3.3
```

When programming lots of floating-point calculations, it is more convenient to have infix `+`, infix `*`, ..., parsed as floating-point operations, instead of integer operations. This can be achieved by the following directive: `#open "float";;`. The directive `#close "float";;` goes back to the original state.

```
##open "float";;

#prefix +;;
- : float -> float -> float
- = <fun>
##close "float";;
```

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

Let us now give some examples of numerical functions. Since we will mostly use floating-point arithmetic, we switch infix operators to floating-point first:

```
##open "float";;
```

We start by defining some very simple functions on numbers:

```
#let square x = x*x;;
square : float -> float
square = <fun>
#square 2.0;;
- : float
- = 4
#square (2.0/3.0);;
- : float
- = 0.444444444444444
#let sum_of_squares (x,y) = square(x) + square(y);;
sum_of_squares : float * float -> float
sum_of_squares = <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.

The statement of Newton's method says that 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 obtain $f'(x) = 2x$ and so:

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

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
deriv = <fun>
```

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

```
#let abs x = if x>0.0 then x else -x;;
abs : float -> float
abs = <fun>
#let newton f epsilon = until satisfied improve
#  where rec until p change x =
#            if p(x) then x
#            else until p change (change(x))
#  and satisfied y = abs(f y) < epsilon
#  and improve y = y - (f(y) / (deriv f y epsilon));;
newton : (float -> float) -> float -> float -> float
newton = <fun>
```

The `until` function is an example of a *recursive* function: it calls itself in its body. If `p(x)` is false, then `until` is called with new arguments. We will study recursive functions more closely later.

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;;
cubic_root : float -> float -> float
square_root : float -> float -> float
cubic_root = <fun>
square_root = <fun>
#square_root 2.0 1e-5;;
- : float
- = 1.41421569552701
#cubic_root 8.0 1e-5;;
- : float
- = 2.00000000130563
#2.0 * (newton cos 1e-5 1.5);;
- : float
- = 3.14159265358854
```

We won't use floating-point arithmetic much in the rest of this chapter, so we switch infix operators back to the integer world.

```
##close "float";;
```

4.2 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 `bool` as codomain are often called *predicates*. Many predicates are predefined in CAML. Here are some of them:

```
#prefix <;;
- : int -> int -> bool
- = <fun>
#1 < 2;;
- : bool
- = true
#lt_float;;
- : float -> float -> bool
- = <fun>
#lt_float 3.14159 2.718;;
- : bool
- = false
```

There exist also `prefix <=`, `prefix >`, `prefix >=`, and similarly `le_float`, `gt_float`, `ge_float`.

4.2.1 Equality

Equality has a special status in functional languages because of functional values. It is easy to test equality of boolean values:

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

But it is impossible, in the general case, to decide the equality of functional values. Several choices are possible: to forbid equality testing of functional values; this can be realised by assigning the property of "admitting equality" to some types (SML, Haskell). In that case, functional types do not admit equality. The choice adopted in CAML was to allow equality testing on any type, but the user has to know that these tests are not safe on functional values (the answer may be `false` when testing equality of two equal functions). The usage of equality on functional values is thus not recommended.

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>
#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
```

4.2.2 Logical operators

The classical logical operators are available in CAML.

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

The operators `&` and `or` are not functions. They should be seen as macros, since, for example, the `or` operator evaluates its second operand only if the first one evaluates to `false`.

The behaviour 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`

4.3 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>
```

Other operations are available (`sub_string`, etc). See the library manual (appendix A) for more information.

4.4 Tuples

4.4.1 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
p = 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*.

4.4.2 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 the chapter dedicated to user-defined product types):

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

4.5 Patterns and pattern-matching

Patterns and pattern-matching play an important role in ML languages. They appear in all real-size 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*), then 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, then 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 garantee 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;;
f : bool -> bool
> Toplevel input:
>let f = function true -> false;;
>        ^^^^^^^^^^^^^^^^^^^^^
>> Pattern matching is not exhaustive

f = <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);;
- : bool
Uncaught exception: match_failure ("", 1269, 1291)
```

Here is a correct definition using pattern-matching:

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

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

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

Simpler, and simpler again:

```
#let imply = function (true,x) -> x
#                   | (false,_) -> true;;
imply : bool * bool -> bool
imply = <fun>
#let imply = function (true,false) -> false
#                   | _ -> true;;
imply : bool * bool -> bool
imply = <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
is_zero = <fun>
#let is_yes = function "oui" -> true
#                    | "si" -> true
#                    | "ya" -> true
#                    | "yes" -> true
#                    | _ -> false (* Unknown language *);;
is_yes : string -> bool
is_yes = <fun>
```

4.6 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.

4.6.1 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
compose = <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
succ = <fun>
#compose succ list_length;;
- : 'a list -> int
- = <fun>
#(compose succ list_length) [1;2;3];;
- : int
- = 4
```

4.6.2 Currying

We can define two versions of the addition function:

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

Its inverse function may be defined by:

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

4.1 Define in CAML functions computing the surface/volume of well-known geometric objects.

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

4.3 Define the `factorial` function such that:

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

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

4.4 Define the `fibinacci` function which, when given a number n, returns the nth Fibonnacci number, i.e. the nth term u_n of the sequence defined by:

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

4.5 What are the types of the following expressions?

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

Chapter 5

Lists

Lists represent an important data structure, mainly because of their success in the Lisp language.

Lists in ML are *homogeneous*: a list may not 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.

5.1 Building lists

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

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

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
adds = [<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 **cons**tructor 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 `prefix ::` constructor receives two arguments; the second argument is itself a `list`.

5.2 Extracting elements from lists: pattern-matching

We know how to build lists; we see now show how to test emptyness of lists (although the equality predicate could be used for that) and extract elements from lists (e.g. the first one). We used pattern-matching on pairs, numbers or boolean values. We shall extend the syntax of patterns to list patterns (in fact, we will see that any data constructor can 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
is_null = <fun>
#let head = function x::_ -> x
#                  | _ -> raise (failure "head");;
head : 'a list -> 'a
head = <fun>
#let tail = function _::L -> L
#                  | _ -> raise (failure "tail");;
tail : 'a list -> 'a list
tail = <fun>
```

Expression `raise (failure "head")` will produce run-time errors since we chose 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 `list` type is a *sum type*, as `bool`, for example, because it is defined with several alternatives:

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

are 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 which 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 is sufficient to indicate statically what their structure is.

5.3 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 into the CAML Light system. See also [Cousineau and Huet, 1990] or [Weis *et al.*, 1990] for other examples of functions over lists. The definitions given here do not produce the most efficient version of each function; these definitions only intend to be easy to read and easy to understand.

Computation of the length of a list:

```
#let rec length = function [] -> 0
#                        | _::L -> 1 + length(L);;
length : 'a list -> int
length = <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
append = <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
rev = <fun>
#rev [1;2;3];;
- : int list
- = [3; 2; 1]
```

The `map` function combines 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). It maps a function over a list, and returns the list of results:

```
#let rec map f =
#    function [] -> []
#           | x::L -> (f x)::(map f L);;
map : ('a -> 'b) -> 'a list -> 'b list
map = <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 is a bit harder to understand in the general case, but its behaviour is quite simple when given a curried addition function, `0` as base element, and a list of numbers. Given a function f, a base element a and a list $[x_1;\ldots;x_n]$, we have:

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

For example, we have:

```
it_list (prefix +) 0 [1;2;3;4;5] = 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
it_list = <fun>
#let sigma = it_list prefix + 0;;
sigma : int list -> int
sigma = <fun>
#sigma [1;2;3;4;5];;
- : int
- = 15
#it_list (prefix *) 1 [1;2;3;4;5];;
- : int
- = 120
```

The `it_list` function represents one of the many ways to iterate over a list, for other list iteration functions, see [Cousineau and Huet, 1990].

Exercises

5.1 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)]
```

We suppose that the argument lists have the same length.

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

Chapter 6

User-defined types

The user is allowed to define his/her own data types. With this facility, one does not have to find clever encodings for the data structures that must be manipulated by a program. 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 already saw 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 exactly the shape of values belonging to cartesian products.

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

6.1 Product types

Product types are *finite labelled* products of types. They are a generalization of cartesian product.

Elements of product types are called *records*.

6.1.1 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 = "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). The type of that function would not represent

the fact that it is supposed to be applied to values such as `jean`, and could be applied to any 4-tuple.

Instead of using cartesian product, we will 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 nested pairs.

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. 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
jean = {Name="Jean"; Age=23; Job="Student"; City="Paris"}
```

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

6.1.2 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
age_of = <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"
```

6.1.3 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;;
second : ('a, 'b) pair -> 'b
first : ('a, 'b) pair -> 'a
second = <fun>
first = <fun>
#let p={Snd=true; Fst=1+2};;
p : (int, bool) pair
p = {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
```

6.2 Sum types

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

6.2.1 Defining sum types

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

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

We then need a type containing *both* integers and character strings. We also want to preserve static type-checking, we thus want to be able to distinguish numbers 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 float;;
Type  defined.
```

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

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

```
#let id1 = Name "Jean"
#and id2 = SS (1670728280305.0);;
id2identification
id1 : identification
id2 = SS 1670728280305
id1 = Name "Jean"
```

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 1670728280305]
```

Injections may possess one argument (as in the example above), serveral arguments, or none. The latter case corresponds 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.

```
#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];;
small_cards_of : suit -> card list
figures_of : suit -> card list
small_cards_of = <fun>
figures_of = <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 (Spade,
 10)]
```

6.2.2 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
color = <fun>
```

Let us count the values of cards (assume we are playing at "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
count = <fun>
```

6.2.3 Recursive types

Some types possess a naturally recursive structure (lists, for example).

Let us define a type for abstract syntax trees of a simple arithmetic language. An arithmetic expression will be either a numeric constant, or a variable, or the addition (multiplication, difference, and 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 args defined.
Type arith_expr defined.

#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"}}
```

Since abstract syntax trees are naturally recursive, their type definition is also recursive. Furthermore, the types `arith_expr` and `args` are defined in a mutually recursive way.

The recursive definition of types may lead to types such that it is impossible to build values of these types.

Example:

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

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

```
#let rec stupid_value =
#    {Head=stupid_value; Tail=stupid_value};;
stupid_value : stupid
stupid_value = {Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={Head={He
ad={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.

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

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.

6.2.5 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;;
identification : > Toplevel input:
>Ace;;
>^^^
>>> Constructor Ace of arity 1 is used with arity 0.
```

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 $f : A \rightarrow B$ 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;;
kingsuit -> card
king = <fun>
```

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

```
#function King c -> c;;
- : card -> suit
 : > Toplevel input:
>function King c -> c;;
>^^^^^^^^^^^^^^^^^^^^
>> Pattern matching is not exhaustive

- = <fun>
```

is $\texttt{king}^{-1}$. It is a partial function, since pattern-matching may fail.

6.2.6 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 `counter` 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 do 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*.

6.3 Summary

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

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

- Types may 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

6.1 Define a function taking as argument a binary tree and returning a pair of lists: the first one should contain all operators of the tree, the second one would contain all its leaves.

6.2 Define a function `map_btree` analoguous to the `list` map function. 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.

6.3 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`::...::`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
list_it = <fun>
```

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

Part II

CAML Light Specifics

Chapter 7

Mutable data structures

The definition of a sum or product type may be annotated to allow pysical (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.

7.0.1 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. Here is how we can do that:

```
#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
jean = {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;;
- : int
- = 24
```

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 it:

```
#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_olderperson -> int
get_older = <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 from writing:

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

Other examples would be:

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

7.0.2 The `ref` type

The `ref` type is the predefined type of a mutable data structure. It is present in the CAML system for reasons of compatibility with earlier CAML versions.The `ref` type could be defined as (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
r = ref 3
```

The definition of `r` should be read as "let `r` be a reference on the value of `1+2`". The value of `r` is nothing but a memory location inside of which we can write.

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

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

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

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

The `ref` identifier acts as a sum data constructor but uses a special syntax.

7.0.3 Polymorphism and mutable data structures

There are some restrictions concerning polymorphism and mutable data structures. One cannot enclose polymorphic objects inside of 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` has been computed (`('a list) ref`), 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

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

7.2 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.

Chapter 8

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 embarassing 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
head = <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.*

8.1 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, one, or several values (`"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 <exc. name> [of <type>]
```

Example:

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

The exception `Found` has been declared, and it carries integers.

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

8.2 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);;
- : 'a
Uncaught exception: Found 5
```

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
find_index = <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`.

8.3 Trapping exceptions

Trapping exceptions is ensured 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 =
#   (try find 1 L with Found n -> n)
#   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 -> int
find_index = <fun>
#find_index (function n -> (n mod 2) = 0)
#           [1;3;5;7;9;10];;
- : int
- = 6
```

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 interruptions from the keyboard such as `^C`!

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

In the example above, we printed 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;;
- : int
Exception raised!
- = 0
```

8.4 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
```

The reason is that if the exception `Exc` is declared to be polymorphic, then a function may raise `Exc [1;2]`. 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`. 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`.

Exercises

8.1 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.

8.2 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 9

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.

9.1 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 standalone program produced by the compiler, all phrases are executed in order. The values of expressions and declared global variables are not printed, however. A standalone 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; ())
  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 stream (here, `std_in`, the stream 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 Unix return code. 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 shell:

```
$camlc -o count count.ml
chars : int ref
lines : int ref
- : unit
```

The compiler displays the types and other compiler-time information for each phrase. In addition, an executable file `count` is produced. You can now run `count` like any other Unix command:

```
$./count < count.ml
308 characters, 15 lines.
```

9.2 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
  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 identifier defined in module `counter.ml` are prefixed with the name of the module, `counter__`. If we have 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
Type counter defined.
new : int -> counter
incr : counter -> unit
read : counter -> int
$camlc -c main.ml
chars : counter__counter
lines : counter__counter
- : unit
$camlc -o main counter.zo main.zo
$./main < counter.ml
146 characters, 5 lines.
```

The `-c` option means "compile only"; that is, the compiler should not attempt to produce a standalone 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 "counters";;
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
;;
```

If two modules, say `mod1` and `mod2`, define both a global value, say `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__+`.

9.3 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 signatures, 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 they are compiled, modules that use the corresponding modules can be compiled immediately, even if the implementations of these modules are not yet compiled, or even not written, as in:

```
$camlc -c counter.mli
Type counter defined.
```

```
new : int -> counter
incr : counter -> unit
read : counter -> int
$camlc -c main.ml
chars : counter__counter
lines : counter__counter
- : unit
$camlc -c counter.ml
Type counter defined.
new : int -> counter
incr : counter -> unit
read : counter -> int
$camlc -o main counter.zo main.zo
```

Exercises

9.1 Complete the `count` command: it should be able to operate on several files, given on the command line. Hint: `unix__command_line` is a vector of strings, containing the command-line arguments to the process.

Part III

A Complete Example

Chapter 10

ASL: A Small Language

We present in this chapter a simple language: ASL (A Small Language). This language is in fact a λ-calculus, enriched with a conditional construct. We need such a conditional, 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 which are available are equality between numbers and arithmetic binary operations.

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

10.1 ASL abstract syntax trees

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

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

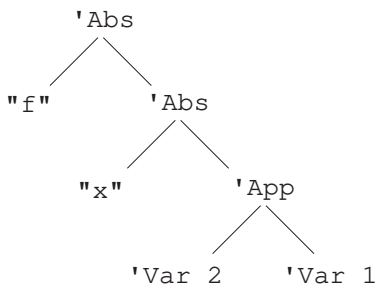
And the ASL application function which would be written in CAML:

```
(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 nth 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 char * asl

#and top_asl = Decl of char * asl;;
Type top_asl defined.
Type asl defined.
```

10.2 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 2) 3`" instead of "`(1+2)+3`". The `if e1 then e2 else e3` construct will be written `C e1 e2 e3`. To keep the parser reasonably small, we will consider only one-character identifiers and integer constants.

The final output of our grammar rules 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. The `index` function is a predefined CAML function.

```
#let binding_depth s rho =
#  bind 1 rho
#  where rec bind n = function
#      []  -> raise (Unbound (char_for_read s))
#  | t::l -> if s = t then Var n else bind (n+1) l;;
```

[1] They have been proposed by N.G. De Bruijn in [De Bruijn, 1962] in order to facilitate the mechanical treatment of λ-calculus terms.

```
binding_depth : char -> char list -> asl
binding_depth = <fun>
```

We also need a global environment, containing names of already bound identifiers. The global environment contains predefined names of the equality and arithmetic functions. We represent it as a reference since each ASL declaration will augment it with a new name.

```
#let init_env =  [`+`;`-`;`*`;`/`;`=`];;
init_env : char list
init_env = [`+`; `-`; `*`; `/`; `=`]
#let global_env = ref init_env;;
global_env : char list ref
global_env = ref [`+`; `-`; `*`; `/`; `=`]
```

We now give a parsing function for ASL programs. This is a standard recursive-descent parser, that scans the string given as argument. We need first a function to split a string into its first character, and the remaining of the string. Blanks at the beginning of the string are skipped.

```
#let rec split s =
#  let rest = sub_string s 1 (string_length s - 1) in
#  match nth_char s 0 with
#    ` `|`\n`|`\t` -> split rest
#  |       c       -> (c,rest);;
split : string -> char * string
split = <fun>
```

The parsing functions take a string (the input text) and return a pair of a function mapping environments to abstract syntax trees, and a string with the remaining, unparsed part of the input.

```
#let rec parse_top s =
#  let (id, s1) = split s in
#  match split s1 with
#    (`=`,s2) ->
#      Decl(id, parse_expression s2)
#  | _ ->
#      raise (failure "`=` expected")

#and parse_expression s =
#  let (e,_) = parse_expr s in
#    try e !global_env
#    with Unbound s ->
#      print_string "Unbound ASL identifier: ";
#      print_string s; print_newline();
#      raise (failure "ASL parsing")

#and parse_expr s =
#  match split s with
#    (`\\`,s1) ->
```

```
#     (let (id,s2) = split s1 in
#      match split s2 with
#        (‘.‘,s3) ->
#          let (e,rest) = parse_expr s3 in
#            (function rho -> Abs(id, e (id::rho))),rest
#      |  _  ->
#          raise (failure "‘.‘ expected"))
#  |  _   ->
#      parse_expr1 (parse_expr0 s)

#and parse_expr1 (e1,s1) =
#  match split s1 with
#    ((‘)‘ | ‘;‘), s2) ->
#      (e1,s1)
#  | _ ->
#      let (e2,s2) = parse_expr0 s1 in
#        parse_expr1 ((function rho -> App(e1 rho, e2 rho)), s2)

#and parse_expr0 s =
#  let (c, s1) = split s in
#  match c with
#    ‘0‘|‘1‘|‘2‘|‘3‘|‘4‘|‘5‘|‘6‘|‘7‘|‘8‘|‘9‘ ->
#      let n = int_of_char c - int_of_char ‘0‘ in
#        (function rho -> Const n), s1
#  | ‘a‘|‘b‘|‘c‘|‘d‘|‘e‘|‘f‘|‘g‘|‘h‘|‘i‘|‘j‘|‘k‘|‘l‘|‘m‘|
#    ‘n‘|‘o‘|‘p‘|‘q‘|‘r‘|‘s‘|‘t‘|‘u‘|‘v‘|‘w‘|‘x‘|‘y‘|‘z‘|
#    ‘+‘|‘-‘|‘*‘|‘/‘|‘=‘ ->
#      (binding_depth c), s1
#  | ‘C‘ ->
#      let (e2,s2) = parse_expr0 s1 in
#      let (e3,s3) = parse_expr0 s2 in
#      let (e4,s4) = parse_expr0 s3 in
#        (function rho -> Cond(e2 rho, e3 rho, e4 rho)), s4
#  | ‘(‘ ->
#      let (e2,s2) = parse_expr s1 in
#      (match split s2 with
#        (‘)‘,s3) -> (e2, s3)
#      |  _  -> raise (failure "‘)‘ expected"))
#  | _ ->
#      raise (failure "illegal character");;
parse_expr0 : string -> (char list -> asl) * string
parse_expr1 : (char list -> asl) * string -> (char list -> asl) * string
parse_expr : string -> (char list -> asl) * string
parse_expression : string -> asl
parse_top : string -> top_asl
```

```
parse_expr0 = <fun>
parse_expr1 = <fun>
parse_expr = <fun>
parse_expression = <fun>
parse_top = <fun>
```

Let's try our grammar (we do not augment the global environment at each declaration: this will be performed after the semantic treatment of ASL programs).

```
#parse_top "f = \\x.x;";;
- : top_asl
- = Decl (`f`, Abs (`x`, Var 1))
#parse_top "x = + 1 ((\\x.x) 2);";;
- : top_asl
- = Decl (`x`, App (App (Var 1, Const 1), App (Abs (`x`, Var 1), Const 2
)))
#parse_top "y = g 3;";;
- : top_asl
Unbound ASL identifier: g
Uncaught exception: failure "ASL parsing"
#parse_top "x = C 0 + -;";;
- : top_asl
- = Decl (`x`, Cond (Const 0, Var 1, Var 2))
```

Chapter 11

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.

11.1 Semantic values

We need a type for ASL semantic values (representing results of computations). A semantic value will be either a numeric value, or a CAML functional value from ASL values to ASL values.

```
#type semval = Numval 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 in 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 Numval n ->
#         Funval(function Numval m -> Numval(caml_fun n m)
#                         | _ -> raise Illtyped)
#             | _ -> raise Illtyped);;
init_semantics : (int -> int -> int) -> semval
init_semantics = <fun>
```

Associate a caml 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 : char -> int -> int -> int
caml_function = <fun>
```

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
init_sem = [Funval <fun>; Funval <fun>; Funval <fun>; Funval <fun>; Funv
al <fun>]
#let global_sem = ref init_sem;;
global_sem : semval list ref
global_sem = ref [Funval <fun>; Funval <fun>; Funval <fun>; Funval <fun>
; Funval <fun>]
```

11.2 Semantic functions

Here is the semantic function: it 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
nth = <fun>
#let rec semant rho = sem
#    where rec sem = function
#      Const n -> Numval n
#    | Var(n) -> nth n rho
#    | Cond(e1,e2,e3) ->
#        (match sem e1 with Numval 0 -> sem e3
#                         | Numval 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
semant = <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_char s;
#       print_string " is ";
#       (match result with
#         Numval n -> print_int n
#       | Funval f -> print_string "<fun>");
#       print_newline();;
semantics : top_asl -> unit
semantics = <fun>
```

11.3 Examples

```
#semantics (parse_top "f = \\x. + x 1;");;
- : unit
ASL Value of f is <fun>
- = ()
#semantics (parse_top "i = \\x. x;");;
- : unit
ASL Value of i is <fun>
- = ()
#semantics (parse_top "x = i (f 2);");;
- : unit
ASL Value of x is 3
- = ()
#semantics (parse_top "y = (C x (\\x.x) 2) 0;");;
- : unit
ASL Value of y is 0
- = ()
```

Chapter 12

Encoding recursion

12.1 Fixpoint combinators

We saw that we did 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:

$$F\ M \text{ is equivalent to } M\ (F\ M) \text{ modulo evaluation rules.}$$

for any expression M.

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).

```
#semantics (parse_top "z = \\f.((\\x.f(\\y.(x x) y))(\\x.f(\\y.(x x) y)));");;
- : unit
ASL Value of z is <fun>
- = ()
```

We are now able to define the ASL factorial function:

```
#semantics (parse_top "f = z(\\f.(\\n. C (= n 0) 1 (* n (f (- n 1))))));");;
- : unit
ASL Value of f is <fun>
- = ()
#semantics (parse_top "x = f 8;");;
- : unit
ASL Value of x is 40320
- = ()
```

and the ASL Fibonacci function:

```
#semantics (parse_top
#  "b = z(\\b.(\\n. C (= n 1) 1 (C (= n 2) 1 (+ (b(- n 1)) (b(- n 2))))));");;
- : unit
ASL Value of b is <fun>
- = ()
#semantics (parse_top "x = b 9;");;
- : unit
ASL Value of x is 34
- = ()
```

12.2 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 13

Static typing, polymorphism and type synthesis

We now want to realise static typechecking of ASL programs, i.e. realise typechecking *before* evaluation, preventing the need for run-time type tests 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 [Clément *et al.*, 1986].

13.1 The type system

The type system we will consider for ASL has been first given by Milner for a subset of the ML language (in fact, a superset of λ-calculus).

A *type* is:

- the type Number;
- or a type variable (α, β, ...);
- or $\tau_1 \rightarrow \tau_2$, where τ_i 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.

[1] A metavariables should not be confused with a *variable* or a *type variable*.

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 may represent type schemes as being of the following form:

$$\forall \alpha_1, \ldots, \alpha_n.\tau \text{ where } \tau \text{ is a type.}$$

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 to *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 $\gamma \to \gamma$ to 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 $\forall \beta$.

□

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 occur 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 with the help 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 premisse: such a rule will be called an *axiom*. A complete proof will be represented by a *proof tree* of the following form:

$$\frac{\underline{P_1^m} \ \ldots \qquad \ldots \ \underline{P_l^k}}{\begin{array}{c}\vdots \\ \underline{\quad} \ \ldots \ \underline{\quad} \\ \underline{P_1^1 \ \ \ldots \ \ P_n^1} \\ C\end{array}}$$

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 *type 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 type hypothesis concerning an identifier name x (if such a type hypothesis exists) before adding a new type hypothesis concerning x.

We write $\Gamma - \Gamma(x)$ for the set ot type hypotheses obtained from Γ by removing the type 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 instanciate type schemes. The "GenInstance" relation represents generic instanciation.

$$\frac{\Gamma \vdash e : \sigma \qquad \sigma' = \text{GenInstance}(\sigma)}{\Gamma \vdash e : \sigma'}(\text{INST})$$

It is possible to generalise type schemes with respect to variables which 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:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

A special rule may be given in order to obtain polymorphism: this is 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 above set of rules, 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 above judgement.

$$\dfrac{\dfrac{\dfrac{\dfrac{}{\{x:\alpha\}\vdash x:\alpha}\text{TAUT}}{\emptyset\vdash(\lambda x.x):\alpha\to\alpha}\text{ABS}}{\emptyset\vdash(\lambda x.x):\forall\alpha.\alpha\to\alpha}\text{GEN} \quad \dfrac{\dfrac{\dfrac{}{\Gamma\vdash f:\forall\alpha.\alpha\to\alpha}\text{TAUT}}{\Gamma\vdash f:(\beta\to\beta)\to(\beta\to\beta)}\text{INST} \quad \dfrac{\dfrac{}{\Gamma\vdash f:\forall\alpha.\alpha\to\alpha}\text{TAUT}}{\Gamma\vdash f:\beta\to\beta}\text{INST}}{\Gamma=\{f:\forall\alpha.\alpha\to\alpha\}\vdash f\ f:\beta\to\beta}\text{APP}}{\emptyset\vdash(\lambda f.f\,f)(\lambda x.x):\beta\to\beta}\text{LET}$$

□

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 must be more specific; in other words, this type must be the *most general.*

13.2 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 instanciate and generalise 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 wil be *types* and not type schemes. 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 occurences of a type metavariable).

$$\frac{\tau=\text{GenInstance}(\sigma)}{\Gamma\cup\{x:\sigma\}\vdash \texttt{Var}\ x:\tau}(\text{INST})$$

When we read this set of inference rules as an algorithm, the (INST) rule will be implemented by:

1. take as τ the type "most general generic instance" of σ,
2. make τ more specific by *unification* ([Robinson, 1971], [Huet, 1986]) when encountering equality constraints.

Typing a conditional:

$$\frac{\Gamma\vdash e_1:\text{Number} \qquad \Gamma\vdash e_2:\tau \qquad \Gamma\vdash e_3:\tau}{\Gamma\vdash(\texttt{if}\ e_1\texttt{then}\ e_2\ \texttt{else}\ e_3\ \texttt{fi}):\tau}(\text{COND})$$

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:

$$\frac{(\Gamma - \Gamma(x)) \cup \{x : \forall.\tau\} \vdash e : \tau'}{\Gamma \vdash (\lambda x\ .e) : \tau \to \tau'}(\text{ABS})$$

Typing a `let` construct involves a generalisation step: we generalise as much as possible.

$$\frac{\begin{array}{ll}\Gamma \vdash e : \tau' & \\ \vec{\alpha} = FV(\tau') - FV(\Gamma) & (\Gamma - \Gamma(x)) \cup \{x : \forall\vec{\alpha}.\tau'\} \vdash e' : \tau\end{array}}{\Gamma \vdash (\lambda x\ .e')\ e : \tau}(\text{LET})$$

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.

$$\frac{\dfrac{\dfrac{}{\{x : \alpha\} \vdash x : \alpha}\text{INST}_x}{\emptyset \vdash (\lambda x.x) : \alpha \to \alpha}\text{ABS}_x \quad \dfrac{\dfrac{}{\Gamma \vdash f\ : (\beta \to \beta) \to (\beta \to \beta)}\text{INST}_f \quad \dfrac{}{\Gamma \vdash f\ : \beta \to \beta}\text{INST}_f}{\Gamma = \{f : \forall\alpha.\alpha \to \alpha\} \vdash f\ f : \beta \to \beta}\text{APP}}{\emptyset \vdash (\lambda f.f\ f)(\lambda x.x) : \beta \to \beta}\text{LET}_f$$

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 above example that the algorithm is *syntax-directed*: we can reconstruct the ASL expression from its type deduction tree. From the above deduction tree, if we write upper rules as being "argument"s 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:

$$\text{LET}_f(\text{ABS}_x(\text{INST}_x),\ \text{APP}(\text{INST}_f,\ \text{INST}_f))$$

This is an illustration of the "types-as-propositions and programs-as-proofs" paradigm, also known as the "Curry-Howard isomorphism" (cf. [Howard, 1980]). 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 recognise the expression as the deduction tree.

13.3 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.

13.3.1 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_scheme defined.
Type vartype defined.
Type asl_type 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
new_vartype : unit -> vartype
reset_vartypes = <fun>
new_vartype = <fun>
```

13.3.2 Destructive unification of ASL types

We will need to "shorten" type variables: they are indirections to ASL types. This is needed for efficiency only.

```
#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
shorten = <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
occurs = <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 t2 else 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 -> Number
#     | Arrow(t1,t2), (Arrow(t'1,t'2) as t)
#            -> unify(t1,t'1); unify(t2,t'2); t
#     | (t1,t2) -> raise (TypeClash (t1,t2));;
unify : asl_type * asl_type -> asl_type
unify = <fun>
```

13.3.3 Representation of typing environments

We use `asl_type_scheme list` as typing environments, and we will use the encoding of variables as numbers 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
init_typing_env = [Forall ([], Arrow (Number, Arrow (Number, Number)));
Forall ([], Arrow (Number, Arrow (Number, Number))); Forall ([], Arrow (
Number, Arrow (Number, Number))); Forall ([], Arrow (Number, Arrow (Numb
er, Number))); Forall ([], Arrow (Number, Arrow (Number, Number)))]
```

The global typing environment is initialised 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
global_typing_env = ref [Forall ([], Arrow (Number, Arrow (Number, Numbe
r))); Forall ([], Arrow (Number, Arrow (Number, Number))); Forall ([], A
rrow (Number, Arrow (Number, Number))); Forall ([], Arrow (Number, Arrow
 (Number, Number))); Forall ([], Arrow (Number, Arrow (Number, Number)))
]
```

13.3.4 From types to type schemes: generalisation

In order to implement generalisation, 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
vars_of_type = <fun>
```

The `unknowns_of_type(bv,t)` application returns the list of variables occurring in `t` which do not appear in `bv`. The `subtract` function returns the difference of two lists.

```
#let rec subtract l1 l2 =
#  match l1 with
#     []   -> l2
#  | x::l3 -> if mem x l2 then subtract l3 l2
#                         else x :: subtract l3 l2;;
subtract : 'a list -> 'a list -> 'a list
subtract = <fun>
#let unknowns_of_type (bv,t) =
#    subtract (vars_of_type t) bv;;
unknowns_of_type : int list * asl_type -> int list
unknowns_of_type = <fun>
```

We need to compute the list of unknowns of a type environment for the generalisation 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
flat = <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
unknowns_of_type_env = <fun>
```

The generalisation of a type is relative to a typing environment. The `make_set` function eliminates duplications of 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
make_set = <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
generalise_type = <fun>
```

13.3.5 From type schemes to types: generic instanciation

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 failure "find" -> 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
gen_instance = <fun>
```

13.3.6 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) ->
#      let t1 = unify(Number, type_rec e1)
#      and 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
asl_typing_expr = <fun>
```

13.3.7 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
tvar_name = <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 failure "find" ->
#               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
print_type_scheme = <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_char s;
# print_string " is ";
# print_type_scheme sigma; print_newline();;
typing : top_asl -> unit
typing = <fun>
```

13.3.8 Typing ASL programs

We reinitialise the parsing environment:

```
#global_env:=init_env; ();;
- : unit
- = ()
```

Now let us run some examples of our ASL type checker:

```
#typing (parse_top "x=1;");;
- : unit
ASL Type of x is Number
- = ()
#typing (parse_top "y = + 2 ((\\x.x) 3);");;
- : unit
ASL Type of y is Number
- = ()
#typing (parse_top "z = C (+ 0 1) 1 0;");;
- : unit
ASL Type of z is Number
- = ()
#typing (parse_top "i = \\x.x;");;
- : unit
ASL Type of i is ('a -> 'a)
- = ()
#typing (parse_top "t = + (i 1) (i i 2);");;
- : unit
ASL Type of t is Number
- = ()

#typing (parse_top "f = (\\x.x x) (\\x.x);");;
- : unit
ASL Type of f is ('b -> 'b)
- = ()
#typing (parse_top "a = + (\\x.x) 1;");;
- : unit
ASL Type clash between Number and ('a -> 'a)
Uncaught exception: failure "ASL typing"
```

13.3.9 Typing and recursion

The z fixpoint combinator does not have a type in Milner's type system:

```
#typing (parse_top "z = \\f.((\\x.f(\\z.(x x)z)) (\\x.f(\\z.(x x)z)));");;
- : unit
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 := `z`::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 may now define our favorite functions as:

```
#typing (parse_top "f = z(\\f.(\\n. C (= n 0) 1 (* n (f (- n 1)))));");;
- : unit
ASL Type of f is (Number -> Number)
- = ()
#typing (parse_top "x = f 8;");;
- : unit
ASL Type of x is Number
- = ()
#typing (parse_top
#  "b = z(\\b.(\\n. C (= n 1) 1 (C (= n 2) 1 (+ (b(- n 1)) (b(- n 2))))));");;
- : unit
ASL Type of b is (Number -> Number)
- = ()
#typing (parse_top "x = b 9;");;
- : unit
ASL Type of x is Number
- = ()
```

Chapter 14

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 realise 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). The CAM is the CAML execution model.

We will call CAM our abstract machine, despite its differences with the original CAM. For more informations on the CAM, see [Cousineau *et al.*, 1985] or [Mauny and Suárez, 1986].

14.1 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    (* Numerical constants *)
```

```
#                      (* Arithmetic operations *)
#   | Plus | Minus | Times | Divide | Equal
#   | 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 `CAMstep` 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 CAMstep S = match S with
#  {Reg=_; PC=Quote(n)::C; Stack=s} ->
#                (S.Reg <- Constant(n); S.PC <- C)
```

```
#| {Reg=Constant(m); PC=Plus::C; Stack=Constant(n)::s} ->
#                (S.Reg <- Constant(n+m);
#                 S.Stack <- s; S.PC <- C)

#| {Reg=Constant(m); PC=Minus::C; Stack=Constant(n)::s} ->
#                (S.Reg <- Constant(n-m);
#                 S.Stack <- s; S.PC <- C)

#| {Reg=Constant(m); PC=Times::C; Stack=Constant(n)::s} ->
#                (S.Reg <- Constant(n*m);
#                 S.Stack <- s; S.PC <- C)

#| {Reg=Constant(m); PC=Divide::C; Stack=Constant(n)::s} ->
#                (S.Reg <- Constant(n/m);
#                 S.Stack <- s; S.PC <- C)

#| {Reg=Constant(m); PC=Equal::C; Stack=Constant(n)::s} ->
#                (S.Reg <- Constant( if n=m then 1 else 0);
#                 S.Stack <- s; S.PC <- C)

#| {Reg=Constant(m); PC=Branch(C1,C2)::C; Stack=r::s} ->
#                (let newpc = if m=0 then C2 else C1 in
#                 S.Reg <- r;
#                 S.Stack <- Address(C)::s;
#                 S.PC <- newpc)

#| {Reg=r; PC=Push::C; Stack=s} ->
#                (S.Stack <- r::s; S.PC <- C)

#| {Reg=r1; PC=Swap::C; Stack=r2::s} ->
#                (let aux = S.Reg in
#                   S.Reg <- r2; S.Stack <- aux::s; S.PC <- C)

#| {Reg=r; PC=Clos(C1)::C; Stack=s} ->
#                (S.Reg <- Closure(Address(C1),r);
#                 S.PC <- C)

#| {Reg=_; PC=[]; Stack=Address(add)::s} ->
#                (let newpc = add in
#                   S.Stack <- s; S.PC <- newpc)

#| {Reg=v; PC=Apply::C;
#          Stack=Closure(Address(C1),Environment(e))::s} ->
#                (S.Reg <- Environment(v::e);
#                 let newpc = C1 in
```

```
#                 S.Stack <- Address(C)::s; S.PC <- newpc)

#| {Reg=v; PC=[]; Stack=[]} -> raise (CAM_End v)
#| {Reg=_; PC=(Plus|Minus|Times|Divide|Equal)::C;
#          Stack=_::_} -> raise (CAMbug "IllTyped")
#| {Reg=Environment(e); PC=Nth(n)::C; Stack=_} ->
#                (S.Reg <- (try nth n e
#                           with failure _ ->
#                             raise (CAMbug "IllTyped"));
#                 S.PC <- C)
#| _ -> raise (CAMbug "Wrong configuration");;
CAMstep : state -> instruction list
CAMstep = <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).

14.2 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
code_of = <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
init_CAM_env = [Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Plus]]
, Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1; Min
us]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nth 1;
 Times]], Environment []); Closure (Address [Clos [Push; Nth 2; Swap; Nt
h 1; Divide]], Environment []); Closure (Address [Clos [Push; Nth 2; Swa
p; Nth 1; Equal]], Environment [])]
#let global_CAM_env = ref init_CAM_env;;
global_CAM_env : object list ref
global_CAM_env = ref [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; Sw
ap; Nth 1; Divide]], 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 (parse_expression "1;");;
```

```
- : instruction list
- = [Quote 1]
#code_of (parse_expression "+ 1 2;");;
- : instruction list
- = [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Quote 2; Apply]
#code_of (parse_expression "(\\x.x) ((\\x.x) 0);");;
- : instruction list
- = [Push; Clos [Nth 1]; Swap; Push; Clos [Nth 1]; Swap; Quote 0; Apply;
 Apply]
#code_of (parse_expression "+ 1 (C 0 2 3);");;
- : instruction list
- = [Push; Push; Nth 6; Swap; Quote 1; Apply; Swap; Push; Quote 0; Branc
h ([Quote 2], [Quote 3]); Apply]
```

14.3 Execution of CAM code

The main function for executing compiled ASL manages the global environment until execution has succeeded.

```
#let ASL_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_char s; print_string " is ";
#    print_type_scheme sigma; print_newline();
#  (* COMPILING *)
#    let C = code_of e in
#    let S = {Reg=Environment(!global_CAM_env);
#             PC=C; Stack=[]} in
#  (* EXECUTING *)
#    let result = try while true do CAMstep S done; S.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();;
ASL_run : top_asl -> unit
ASL_run = <fun>
```

Now, let us run some examples:

```
#(* Reinitialising environments *)
#global_env:=init_env;
#global_typing_env:=init_typing_env;
#global_CAM_env:=init_CAM_env;
#();;
- : unit
- = ()
#ASL_run (parse_top "a = 1;");;
- : unit
ASL Type of a is Number
1
- = ()
#ASL_run (parse_top "b = + 1 2;");;
- : unit
ASL Type of b is Number
3
- = ()
#ASL_run (parse_top "c = (\\f.(\\x.f x)) (\\x. + x 1) 3;");;
- : unit
ASL Type of c is Number
4
- = ()
```

We may now introduce the z fixpoint combinator as a predefined function.

```
#(* Introducing the Z fixpoint combinator *)
#global_env:=`z`::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 (parse_expression "\\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 z"))
# ::init_CAM_env; ();;
- : unit
- = ()

#ASL_run (parse_top "f = z (\\f.(\\n. C (= n 0) 1 (* n (f (- n 1)))));");;
- : unit
ASL Type of f is (Number -> Number)
<funval>
- = ()
#ASL_run (parse_top
#  "b = z(\\b.(\\n. C (= n 1) 1 (C (= n 2) 1 (+ (b(- n 1)) (b(- n 2)))))));");;
- : unit
ASL Type of b is (Number -> Number)
<funval>
- = ()
#ASL_run (parse_top "x = f 8;");;
- : unit
ASL Type of x is Number
40320
- = ()
#ASL_run (parse_top "y = b 9;");;
- : unit
ASL Type of y is Number
34
- = ()
```

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 [Mauny and Suárez, 1986] for efficient compilation of recursion, data structures etc.

Chapter 15

Answers to exercises

We give in this chapter one possible solution for each exercise contained in this document. Excercises are referred 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.

3.1, p. 15

```
#(function f -> (function m -> f(m+1) / 2));;
- : (int -> int) -> int -> int
- = <fun>
#function m -> (function n -> n+m+1);;
- : int -> int -> int
- = <fun>
#function f -> (f 2)+1;;
- : (int -> int) -> int
- = <fun>
```

3.2, p. 15

We must first rename `y` in `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.

3.3, p. 15

We write successively the reduction steps for each expressions, and then we use CAML in order to check our 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
  ```
  ```

4.1, p. 28

This exercise can be very simple—follow your inspiration!. The computation of the surface of a rectangle could be expresed as:

```
#let sq_rect (l,L) = l*L;;
sq_rect : int * int -> int
sq_rect = <fun>
```

4.2, p. 28

In a call-by-value language without conditional construct (and without sum types), all programs involving a recursive definition fail to terminate.

4.3, p. 28

```
#let rec factorial n = if n=1 then 1 else n*(factorial(n-1));;
factorial : int -> int
factorial = <fun>
#factorial 5;;
- : int
- = 120
#let factorial n = fact n 1
#where rec fact n m = if n=1 then m else fact (n-1) (n*m);;
factorial : int -> int
factorial = <fun>
#factorial 5;;
- : int
- = 120
```

4.4, p. 28

```
#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
fibonacci = <fun>
#fibonacci 20;;
- : int
- = 6765
```

4.5, p. 28

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

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

5.1, p. 32

```
#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
combine = <fun>
#combine ([1;2;3],["a";"b";"c"]);;
- : (int * string) list
- = [1, "a"; 2, "b"; 3, "c"]
#combine ([1;2;3],["a";"b"]);;
- : (int * string) list
Uncaught exception: failure "combine: lists of different length"
```

5.2, p. 32

```
#let rec sublists =
#    function [] -> [[]]
#           | x::L -> let SL = sublists L
#                     in SL @ (map (fun l -> x::l) SL);;
sublists : 'a list -> 'a list list
sublists = <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"]]
```

6.1, p. 41

```
#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 node defined.
Type btree 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
nodes_and_leaves = <fun>
#nodes_and_leaves (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : string list * int list
- = ["+"], [1; 2]
```

6.2, p. 41

```
#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
map_btree = <fun>
```

6.3, p. 41

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
btree_it = <fun>
```

```
#btree_it (function x -> x)
#         (function "+" -> prefix +
#                 | _ -> raise (failure "Unknown op"))
#         (Btree {Op="+"; Son1=Leaf 1; Son2=Leaf 2});;
- : int
- = 3
```

7.1, p. 48

```
#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;;
rplacd : ('a, 'b) lisp_cons -> 'b -> 'b
rplaca : ('a, 'b) lisp_cons -> 'a -> 'a
cdr : ('a, 'b) lisp_cons -> 'b
car : ('a, 'b) lisp_cons -> 'a
rplacd = <fun>
rplaca = <fun>
cdr = <fun>
car = <fun>
#let p = {Car=1; Cdr=true};;
p : (int, bool) lisp_cons
p = {Car=1; Cdr=true}
#rplaca p 2;;
- : int
- = 2
#p;;
- : (int, bool) lisp_cons
- = {Car=2; Cdr=true}
```

7.2, p. 48

```
#let stamp_counter = ref 0;;
stamp_counter : int ref
stamp_counter = ref 0
#let stamp () =
#  stamp_counter := 1 + !stamp_counter; !stamp_counter;;
stamp : unit -> int
stamp = <fun>
#stamp();;
- : int
- = 1
#stamp();;
```

```
- : int
- = 2
```

8.1, p. 52

```
#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
find_succeed = <fun>
#let hd = function [] -> raise (failure "empty") | x::l -> x;;
hd : 'a list -> 'a
hd = <fun>
#find_succeed hd [[];[];[1;2];[3;4]];;
- : int list
- = [1; 2]
```

8.2, p. 52

```
#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
map_succeed = <fun>
#map_succeed hd [[];[1];[2;3];[4;5;6]];;
- : int list
- = [1; 2; 4]
```

9.1, p. 57

```
(* main.ml *)
let chars = counter__new 0;;
let lines = counter__new 0;;

let count_file filename =
  let in_chan = open_in filename in
    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 unix__command_line - 1 do
  count_file unix__command_line.(i)
done;
print_int (counter__read chars);
print_string " characters, ";
print_int (counter__read lines);
print_string " lines.\n";
exit 0
;;
```

Chapter 16

Conclusion and further reading

We have not been exhaustive in the description of the CAML features. We only introduced general concepts in functional programming, and we have insisted on the features used in the prototyping of ASL. Further information about CAML can be found in [Cousineau and Huet, 1990] and [Weis *et al.*, 1990].

We still hope to have convinced the reader of the "clean" approach to programming provided by functional languages.

More information on unification and operations on first-order terms can be found in [Huet, 1986]. An introduction to lambda-calculus and type systems can be found in [Huet, 1990], [Krivine, 1990] and [Hindley and Seldin, 1986].

The description of the implementation of call-by-value functional programming languages can be found in [Leroy, 1990].

The implementation of lazy functional languages is described in [Peyton-Jones, 1987][1]. An introduction to programming in lazy functional languages can be found in [Bird and Wadler, 1986].

[1] of which [Peyton-Jones, 1990] is a French translation.

Bibliography

[Backus, 1978] 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.

[Bird and Wadler, 1986] R. Bird and P. Wadler. *Introduction to Functional Programming.* Series in Computer Science. Prentice-Hall International, 1986.

[Church, 1941] A. Church. *The Calculi of Lambda Conversion.* Princeton University Press, 1941.

[Clément *et al.*, 1986] 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.

[Cousineau and Huet, 1990] G. Cousineau and G. Huet. The CAML primer. Technical Report 122, INRIA, 1990.

[Cousineau *et al.*, 1985] G. Cousineau, P.-L. Curien, and M. Mauny. *The Categorical Abstract Machine*, pages 50–64. Number 201 in Lecture Notes in Computer Science. Springer Verlag, 1985.

[De Bruijn, 1962] N. De Bruijn. *Lambda-calculus notation with nameless dummies, a tool for automatic formula manipulation.* Indag. Math., 1962.

[Gordon *et al.*, 1978] 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.

[Hindley and Seldin, 1986] J.R. Hindley and J.P. Seldin. *Introduction to Combinators and λ-calculus.* London Mathematical Society, Student Texts. Cambridge University Press, 1986.

[Howard, 1980] W.A. Howard. *The formulae-as-type notion of construction*, pages 479–490. Academic Press, 1980.

[Hudak and Wadler, 1990] P. Hudak and P. Wadler. Report on the programming language Haskell. Technical Report YALEU/DCS/RR-777, Yale University, 1990.

[Huet, 1986] G. Huet. Formal structures for computation and deduction. Course Notes, 1986.

[Huet, 1990] G. Huet. Initiation au lambda-calcul. Notes de cours, 1990.

[Johnsson, 1984] T. Johnsson. Efficient compilation of lazy evaluation. In *Proceedings of ACM Conference on Compiler Construction*, pages 58–69, 1984.

[Krivine, 1990] J.-L. Krivine. *Lambda-calcul, types et modèles.* Etudes et recherches en informatique. Masson, 1990.

[Landin, 1966] P. Landin. The next 700 programming languages. In *Communications of the ACM*, volume 9, pages 157–164, 1966.

[Leroy, 1990] X. Leroy. The ZINC experiment: an economical implementation of the ML language. Technical Report 117, INRIA, 1990.

[MacCarthy, 1962] J. MacCarthy. *Lisp 1.5 Programmer's Manual.* MIT Press, Cambridge, Mass., 1962.

[MacQueen, 1984] D. MacQueen. Modules for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1984.

[Mauny and Suárez, 1986] 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.

[Mauny, 1991] Michel Mauny. Functional programming in CAML. Technical report, INRIA, 1991.

[Milner, 1978] R. Milner. A theory of type polymorphism in programming. *J. Comput. Syst. Sci.*, 17:348–375, 1978.

[Milner, 1987] R. Milner. A proposal for Standard ML. In *Proceedings of the ACM Conference on Lisp and Functional Programming*, 1987.

[Peyton-Jones, 1987] S.L. Peyton-Jones. *The Implementation of Functional Programming Languages.* Series in Computer Science. Prentice-Hall International, 1987.

[Peyton-Jones, 1990] S.L. Peyton-Jones. *Mise en œuvre des langages fonctionnels de programmation.* Manuels Informatiques Masson. Masson, 1990.

[Robinson, 1971] J. A. Robinson. Computational logic: the unification computation. In *Machine Intelligence*, volume 6 of *Americal Elsevier*. B. Meltzer and D. Mitchie (Eds), 1971.

[Turner, 1976] D. Turner. SASL language manual. Technical report, St Andrews University, 1976.

[Turner, 1982] D. Turner. *Recursion equations as a programming language*, pages 1–28. Cambridge University Press, 1982.

[Turner, 1985] 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.

[Weis *et al.*, 1990] P. Weis, M.V. Aponte, A. Laville, M. Mauny, and A. Suárez. The CAML reference manual. Technical Report 121, INRIA, 1990.

Appendix A

The standard library

A.1 Module `toplvl`: System functions for interactive use

```
exception Break
           Raised when the CAML process receives the interrupt signal.

quit : 'a -> 'b
           Exits the toplevel loop.
load : string -> unit
           Reads, compiles and executes source phrases from the given file.
           The ".ml" extension is automatically added, if not present.
load_object : string -> unit
           Loads in core a compiled bytecode file.
           The ".zo" extension is automatically added, if not present.
trace : string -> unit
           After 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 exceptions escaping out of "foo".
untrace : string -> unit
           untrace "foo" stops tracing function "foo".
```

A.2 Module `int`: Operations on integers

```
   Integers are 31-bit wide. All operations are taken modulo 2^31.
   They do not fail on overflow.

exception Division_by_zero

minus : int -> int
minus_int : int -> int
           Unary negation. Also written "-e".
succ : int -> int
           succ x = x+1
```

```
pred : int -> int
           pred x = x+1
prefix + : int -> int -> int
add_int : int -> int -> int
           Addition
prefix - : int -> int -> int
sub_int : int -> int -> int
           Substraction
prefix * : int -> int -> int
mult_int : int -> int -> int
           Product
prefix / : int -> int -> int
           Integer division. Raises Division_by_zero if second argument is 0
prefix mod : int -> int -> int
           Remainder. Raises Division_by_zero if second argument is 0.
           Gives unpredictable results if either argument is negative.
prefix = : int -> int -> bool
eq_int : int -> int -> bool
           Integer equality. Equivalent to generic equality, just faster.
prefix <> : int -> int -> bool
neq_int : int -> int -> bool
           Negation of the above.
prefix < : int -> int -> bool
lt_int : int -> int -> bool
prefix > : int -> int -> bool
gt_int : int -> int -> bool
prefix <= : int -> int -> bool
le_int : int -> int -> bool
prefix >= : int -> int -> bool
ge_int : int -> int -> bool
           Usual comparisons between integers.

land : int -> int -> int
           Bitwise logical and.
lor : int -> int -> int
           Bitwise logical or.
lxor : int -> int -> int
           Bitwise logical exclusive or.
lshift_left : int -> int -> int
           "lshift_left n m" shifts n to the left by m bits.
lshift_right : int -> int -> int
           "lshift_right n m" shifts n to the right by m bits.
           This is an arithmetic shift (the sign bit is replicated).

string_of_int : int -> string
           Converts the given integer to its decimal representation
int_of_string : string -> int
           Converts the given string to an integer, in decimal.
           Raises failure "int_of_string" if the given string is not
           a valid representation of an integer.
```

A.3 Module float: Operations of floating-point numbers

```
int_of_float : float -> int
           Truncates the given float to an integer value.
           The result is unspecified if if falls outside the
           range of integers.
float_of_int : int -> float
           Converts an integer to floating-point.

minus : float -> float
minus_float : float -> float
           Unary negation.
prefix + : float -> float -> float
add_float : float -> float -> float
           Addition
prefix - : float -> float -> float
sub_float : float -> float -> float
           Substraction
prefix * : float -> float -> float
mult_float : float -> float -> float
           Product
prefix / : float -> float -> float
div_float : float -> float -> float
           Division. The result is unpredictable if the dividend is 0.0.
prefix = : float -> float -> bool
eq_float : float -> float -> bool
           Floating-point equality.
           Equivalent to generic equality, just faster.
prefix <> : float -> float -> bool
neq_float : float -> float -> bool
           Negation of the above.
prefix < : float -> float -> bool
lt_float : float -> float -> bool
prefix > : float -> float -> bool
gt_float : float -> float -> bool
prefix <= : float -> float -> bool
le_float : float -> float -> bool
prefix >= : float -> float -> bool
ge_float : float -> float -> bool
           Usual comparisons between floating-point numbers.

string_of_float : float -> string
           Converts the given float to its decimal representation
float_of_string : string -> float
           Converts the given string to a float, in decimal.
           The result is unspecified if the given string is not
           a valid representation of an integer.
exp : float -> float
log : float -> float
sqrt : float -> float
```

```
power : float -> float -> float
sin : float -> float
cos : float -> float
tan : float -> float
asin : float -> float
acos : float -> float
atan : float -> float
atan2 : float -> float -> float
            Usual transcendental functions on floating-point numbers.
```

A.4 Module eq: Equality functions

```
prefix = : 'a -> 'a -> bool
           "e1 = e2" tests for structural equality of e1 and e2.
           Mutable structures (e.g. references) are equal even if
           they are not the same physical object, but their
           current contents are structurally equal.
           Equality between functional values returns meaningless results.
           Equality between cyclic data structures may not terminate.
prefix <> : 'a -> 'a -> bool
           Negation of prefix =
prefix == : 'a -> 'a -> bool
           "e1 == e2" tests for physical equality of e1 and e2
           (pointer equality).
prefix != : 'a -> 'a -> bool
           Negation of prefix ==
```

A.5 Module bool: Boolean operations

```
   e1 & e2  : conjunction
   e1 or e2 : disjunction.
   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
     e1 or e2 is equivalent to  if e1 then true else e2.


prefix not : bool -> bool
   The boolean negation.
```

A.6 Module exc: Exceptions

```
raise : exn -> 'a
           Raises the given exception value.

   A few general-purpose predefined exceptions.
```

```
exception failure of string
           Raised by some library functions, to signal that they are
           undefined on the given arguments.
exception Invalid_argument of string
           Raised by some library functions, to signal that the given
           arguments do not make sense.
exception Out_of_memory
           Raised by the garbage collector, when there is insufficient
           memory to complete the computation.

failwith : string -> 'a
           Raises exception failure with the given string.
invalid_arg : string -> 'a
           Raises exception Invalid_argument with the given string.
```

A.7 Module cchar: Character operations, with sanity checks.

```
int_of_char : char -> int
           Returns the ASCII code of its argument
char_of_int : int -> char
           Returns the character with the given ASCII code.
           Raises Invalid_argument "char_of_int" if the argument is
           outside the range 0..255.
char_for_read : char -> string
           Returns a string representing the given character,
           with special characters escaped following the lexical conventions
           of CAML Light.
```

A.8 Module cstring: String operations, with sanity checks.

```
string_length : string -> int
           Returns the length (number of characters) of the given string.
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.
           Raises Invalid_argument "nth_char" if n is ouside the range
           0 .. (string_length s - 1).
set_nth_char : string -> int -> char -> char
           "set_nth_char s n c" physically modifies string s,
           replacing character number n by character c.
           Raises Invalid_argument "set_nth_char" if n is ouside the range
           0 .. (string_length s - 1).
prefix ^ : string -> string -> string
           "s1 ^ s2" returns a fresh string containing the concatenation of
```

```
           strings s1 and s2.
sub_string : string -> int -> int -> string
           "sub_string s start len" returns a fresh string of length len,
           containing characters number start through start + len - 1
           of string s.
           Raises Invalid_argument "sub_string" if start, len do not designate a
           valid substring of s; that is, if start < 0, or len > 0, or
           start + len > string_length s.
create_string : int -> string
           "create_string n" returns a fresh string of length n,
           containing random characters.
make_string : int -> char -> string
           "make_string n c" returns a fresh string of length n,
           initialized to the character c.
fill_string : string -> int -> int -> char -> string
           "fill_string s start len c" physically modifies string s,
           replacing characters number start through start + len - 1
           by character c. String s is returned.
           Raises Invalid_argument "fill_string" if start, len do not designate
           a valid substring of s.
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. Works correctly even if s1 = s2
           and the source chunk and destination chunk overlap.
           Raises Invalid_argument "blit_string" if o1, len do not designate a
           valid substring of s1, or if o2, len do not designate a valid
           substring of s2.
replace_string : string -> string -> int -> unit
           "replace_string dest src start" copies all characters from string
           src into string dst, starting at offset start.
           Raises Invalid_argument "replace_string" if copying would overflow
           string dest.
string_for_read : string -> string
           Returns a copy of its argument, with special characters represented
           by escape sequences, following the lexical conventions of
           CAML Light.
string_length : string -> int
           Returns the length (number of characters) of the given string.
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.
           Raises Invalid_argument "nth_char" if n is ouside the range
           0 .. (string_length s - 1).
set_nth_char : string -> int -> char -> char
           "set_nth_char s n c" physically modifies string s,
           replacing character number n by character c.
           Raises Invalid_argument "set_nth_char" if n is ouside the range
           0 .. (string_length s - 1).
prefix ^ : string -> string -> string
```

```
           "s1 ^ s2" returns a fresh string containing the concatenation of
           strings s1 and s2.
sub_string : string -> int -> int -> string
           "sub_string s start len" returns a fresh string of length len,
           containing characters number start through start + len - 1
           of string s.
           Raises Invalid_argument "sub_string" if start, len do not designate
           a valid substring of s; that is, if start < 0, or len > 0, or
           start + len > string_length s.
fill_string : string -> int -> int -> char -> string
           "fill_string s start len c" physically modifies string s,
           replacing characters number start through start + len - 1
           by character c. String s is returned.
           Raises Invalid_argument "fill_string" if start, len do not designate
           a valid substring of s.
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. Works correctly even if s1 = s2
           and the source chunk and destination chunk overlap.
           Raises Invalid_argument "blit_string" if o1, len do not designate a
           valid substring of s1, or if o2, len do not designate a valid
           substring of s2.
replace_string : string -> string -> int -> unit
           "replace_string dest src start" copies all characters from string
           src into string dst, starting at offset start.
           Raises Invalid_argument "replace_string" if copying would overflow
           string dest.
string_for_read : string -> string
           Returns a copy of its argument, with special characters represented
           by escape sequences, following the lexical conventions of
           CAML Light.
```

A.9 Module pair: Operations on pairs

```
fst : 'a * 'b -> 'a
           Returns the first component of a pair
snd : 'a * 'b -> 'b
           Returns the second component of a pair
split : ('a * 'b) list -> 'a list * 'b list
           Transforms a list of pairs into a pair of lists:
             split [(a1,b1); ...; (an,bn)] = [a1; ...; an], [b1; ...; bn]
combine : 'a list * 'b list -> ('a * 'b) list
           Transforms a pair of lists into a list of pairs:
             combine ([a1; ...; an], [b1; ...; bn]) =
               [(a1,b1); ...; (an,bn)].
           Raises failure "combine" if the two lists have different lengths.
map2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> 'c list
```

```
           map2 f [a1; ...; an] [b1; ...; bn] = [f a1 b1; ...; f an bn]
map2_uncurry : ('a * 'b -> 'c) -> 'a list * 'b list -> 'c list
           map2_uncurry f [a1; ...; an] [b1; ...; bn] =
               [f (a1, b1); ...; f (an, bn)]
do_list2 : ('a -> 'b -> 'c) -> 'a list -> 'b list -> unit
           do_list2 f [a1; ...; an] [b1; ...; bn] performs
               f a1 b1; ...; f an bn
do_list2_uncurry : ('a * 'b -> 'c) -> 'a list * 'b list -> unit
           do_list2_uncurry f [a1; ...; an] [b1; ...; bn] performs
               f (a1, b1); ...; f (an, bn)
it_list2 : ('a -> 'b -> 'c -> 'a) -> 'a -> 'b list -> 'c list -> 'a
           it_list2 f a [b1; ...; bn] [c1; ...; cn] =
               f (... (f (f a b1 c1) b2 c2) ...) bn cn
list_it2 : ('a -> 'b -> 'c -> 'c) -> 'a list -> 'b list -> 'c -> 'c
           list_it2 f [a1; ...; an] [b1; ...; bn] c =
               f a1 b1 (f a2 b2 (... (f an bn c) ...))
```

A.10 Module cvect: Operations on vectors, with sanity checks.

```
   Special syntax to build a vector of length n, with elements e1, ..., en :
      [| e1; ... ; en |].

vect_length : 'a vect -> int
           Returns the length (number of elements) of the given vector.
vect_item : 'a vect -> int -> 'a
           "vect_item v n" returns element number n of vector v.
           The first element has number 0.
           The last element has number vect_length v - 1.
           Raises Invalid_argument "vect_item" if n is outside the range
           0 .. vect_length v - 1.
           Special syntax is provided: "v.(n)" can be written in place of
           "vect_item v n".
vect_assign : 'a vect -> int -> 'a -> 'a
           "vect_assign v n x" physically modifies vector v, replacing
           element number n with element x.
           Raises Invalid_argument "vect_assign" if n is outside the range
           0 .. vect_length v - 1.
           Special syntax is provided: "v.(n) <- x" can be written in place of
           "vect_assign v n x".
concat_vect : 'a vect -> 'a vect -> 'a vect
           "concat_vect v1 v2" returns a fresh vector containing the
           concatenation of vectors v1 and v2.
sub_vect : 'a vect -> int -> int -> 'a vect
           "sub_vect v start len" returns a fresh vector of length len,
           containing elements number start through start + len - 1
           of vector v.
           Raises Invalid_argument "sub_vect" if start, len do not designate a
           valid subvector of v; that is, if start < 0, or len > 0, or
           start + len > vect_length v.
```

```
make_vect : int -> 'a -> 'a vect
            "make_vect n x" returns a fresh vector of length n,
            initialized with n pointers to element x.
fill_vect : 'a vect -> int -> int -> 'a -> 'a vect
            "fill_vect v ofs len x" physically modifies vector v,
            storing element x in elements number ofs through ofs + len - 1.
            Raises Invalid_argument "fill_vect" if ofs, len do not designate
            a valid subvector of v.
blit_vect : 'a vect -> int -> 'a vect -> int -> int -> unit
            "blit_vect v1 o1 v2 o2 len" copies len elements
            from vector v1, starting at element number o1, to vector v2,
            starting at element number o2. Works correctly even if v1 and v2
            are the same vector, and the source chunk and destination chunk
            overlap.
            Raises Invalid_argument "blit_vector" if o1, len do not designate a
            valid subvector of v1, or if o2, len do not designate a valid
            subvector of v2.
list_of_vect : 'a vect -> 'a list
            "list_of_vect v" returns the list of all elements of v, that is:
              [v.(0); v.(1); ...; v.(vect_length v - 1)].
vect_of_list : 'a list -> 'a vect
            "vect_of_list l" returns a fresh vector containing the elements
            of list l.
do_vect : ('a -> 'b) -> 'a vect -> unit
            "do_vect f v" applies function f in turn to all elements of v:
               f v.(0); f v.(1); ...; f v.(vect_length v - 1).
map_vect : ('a -> 'b) -> 'a vect -> 'b vect
            "map_vect f v" applies function f to all elements of v, and builds
            a vector with the results returned by f:
               [| f v.(0); f v.(1); ...; f v.(vect_length v - 1) |].
map_vect_list : ('a -> 'b) -> 'a vect -> 'b list
            "map_vect_list f v" applies function f to all 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) ].
```

A.11 Module `list`: Operation on lists

```
list_length : 'a list -> int
            Returns the length of a list (number of elements).
prefix @ : 'a list -> 'a list -> 'a list
            List concatenation
rev : 'a list -> 'a list
            List reversal
do_list : ('a -> 'b) -> 'a list -> unit
            do_list f [a1; ...; an] performs f a1; f a2; ...; f an.
map : ('a -> 'b) -> 'a list -> 'b list
            map f [a1; ...; an] = [f a1; ...; f an]
it_list : ('a -> 'b -> 'a) -> 'a -> 'b list -> 'a
            it_list f a [b1; ...; bn] = f (... (f (f a b1) b2) ...) bn
```

```
list_it : ('a -> 'b -> 'b) -> 'a list -> 'b -> 'b
          list_it f [a1; ...; an] b = f a1 (f a2 (... (f an b) ...))
flat_map : ('a -> 'b list) -> 'a list -> 'b list
          flat_map f [l1; ...; ln] = (f l1) @ (f l2) @ ... @ (f ln)
for_all : ('a -> bool) -> 'a list -> bool
          for_all p [a1; ...; an] = (p a1) & (p a2) & ... & (p an)
exists : ('a -> bool) -> 'a list -> bool
          exists p [a1; ...; an] = (p a1) or (p a2) or ... or (p an)

mem : 'a -> 'a list -> bool
          "mem a l" is true iff a is structurally equal to one element
          of list l.
memq : 'a -> 'a list -> bool
          "memq a l" is true iff a is physically equal to one element
          of list l.
except : 'a -> 'a list -> 'a list
          "except a l" returns list l where the first element structurally
          equal to a has been removed.
exceptq : 'a -> 'a list -> 'a list
          "except a l" returns list l where the first element physically
          equal to a has been removed.
subtract : 'a list -> 'a list -> 'a list
          "subtract l1 l2" returns list l1 where all elements structurally
          equal to one of the elements of l2 have been removed.
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. Raises failure "index" if a cannot be found in l.
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.
          Raises failure "find" if there is no value associated with a in the
          list.
assq :  'a -> ('a * 'b) list -> 'b
          Same as assoc, but uses physical equality instead of structural
          equality to compare keys.
mem_assoc : 'a -> ('a * 'b) list -> bool
          Same as assoc, but simply returns true if one binding exist,
          and false if no binding exist for the given key.
```

A.12 Module `ref`: Operations on references

```
type 'a ref = ref of mutable 'a

prefix ! : 'a ref -> 'a
          "!r" returns the current contents of reference r.
```

```
           Could be defined as function ref x -> x.
prefix := : 'a ref -> 'a -> 'a
           "r := a" stores the value of a in reference r.
           Could be defined as fun (ref x) a -> (x <- a).
incr : int ref -> int
           Increments the integer contained in the given reference.
           Could be defined as fun r -> r := succ !r.
decr : int ref -> int
           Decrements the integer contained in the given reference.
           Could be defined as fun r -> r := pred !r.
```

A.13 Module io: Buffered input-output

```
type in_channel                          input channels
 and out_channel                         output channels
exception End_of_file
           Raised when an operation cannot complete, because the end
           of the file is reached.
std_in : in_channel
std_out : out_channel
std_err : out_channel
           The Unix standard input and outputs of the process.
exit : int -> 'a
           Flushes all pending writes on std_out and std_err,
           and terminates the process with the given status code
           (usually 0 to indicate no errors, and a positive integer
           to indicate failure. This function MUST be called at
           the end of 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.

open_in : string -> in_channel
           Opens the named file for reading, and returns a new input channel
           on that file, positionned at the beginning of the file.
           Raises OS_error if the file could not be opened.
open_in_gen : int -> int -> string -> in_channel
           "open_in_gen mode rights filename" opens the file named
           filename for reading, as above. The integers mode and rights
           specify the opening mode (see unix__open).
open_descriptor_in : int -> in_channel
           "open_descriptor_in fd" returns a buffered input channel
           reading from the file descriptor fd. fd must have been previously
           opened for reading (see unix__open, unix__pipe).
input_char : in_channel -> char
           Reads one character from the given input channel.
           Raises End_of_file if there are no more characters to read.
input : in_channel -> string -> int -> int -> int
           "input chan buff ofs len" attempts to read len characters
           from channel chan, storing them in string buf, starting at
```

```
           offset ofs. It returns the actual number of characters
           read, between 0 and len. On regular files, a return value
           less than len means that the end of the file was reached.
           On pipes, terminals, sockets, ..., a return value of 0
           usually 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. See the Unix man
           pages for read(2).
           Anything can happen if (ofs,len) does not correspond to a
           valid substring of buff.
input_byte : in_channel -> int
           Same as input_char, but returns the 8-bit integer representing
           the character.
           Raises End_of_file if there are no more characters to read.
input_int : in_channel -> int
           Reads four character from the given input channel, and
           returns the integer encoded by these characters. See output_int.
           Raises End_of_file if there are not enough characters to read.
input_value : in_channel -> 'a
           Reads the representation of a structured value, as produced
           by output_value, and returns 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, that cannot be determined at compile-time. The
           programmer must cast the returned value to the expected
           type, and be sure that the object in the file does belong
           to that type.
seek_in : in_channel -> int -> unit
           "seek_in chan pos" sets the current reading position to pos
           for channel chan. This works only for regular files. On
           pipes, terminals, sockets, ..., this does nothing, but does
           not fail either.
pos_in : in_channel -> int
           Returns the current reading position for the given channel.
           This works only for regular files. On pipes, terminals,
           sockets, ..., the result is meaningless.
in_channel_length : in_channel -> int
           Returns the total length (number of characters) for the
           given channel. This works only for regular files. On pipes,
           terminals, sockets, ..., the result is meaningless.
close_in : in_channel -> unit
           Closes the given channel. Anything can happen if any of the
           operations above is performed on a closed channel.
open_out : string -> out_channel
           Opens the named file for writing, and returns 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.
           Raises OS_error if the file could not be opened.
open_out_gen : int -> int -> string -> out_channel
           "open_out_gen mode rights filename" opens the file named
```

```
           filename for writing, as above. The integers mode and rights
           specify the opening mode (see unix__open).
open_descriptor_out : int -> out_channel
           "open_descriptor_in fd" returns a buffered output channel
           reading from the file descriptor fd. fd must have been previously
           opened for writing (see unix__open, unix__pipe).
flush : out_channel -> unit
           Flushes 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.
output_char : out_channel -> char -> unit
           Writes one character on the given output channel.
output_string : out_channel -> string -> unit
           Writes the given string on the given output channel.
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.
           Anything can happen if (ofs,len) does not correspond to a
           valid substring of buff.
output_byte : out_channel -> int -> unit
           Writes one 8-bit integer on the given output channel.
output_int : out_channel -> int -> unit
           Writes one integer as four characters on the given output channel.
           The only reliable way to read it back is through the
           input_int function.
output_value : out_channel -> 'a -> unit
           Writes the representation of a structured value of any type
           to a channel. Circularities and sharing inside the value
           are detected and preserved. The object can be read back,
           possibly in another process, by the function input_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
           pipes, terminals, sockets, ..., this does nothing, but does
           not fail either.
pos_out : out_channel -> int
           Returns the current writing position for the given channel.
           This works only for regular files. On pipes, terminals,
           sockets, ..., the result is meaningless.
out_channel_length : out_channel -> int
           Returns the total length (number of characters) for the
           given channel. This works only for regular files. On pipes,
           terminals, sockets, ..., the result is meaningless.
close_out : out_channel -> unit
           Closes the given channel, flushing all buffered write operations.
           All opened output channels must be closed by hand at the
           end of a standalone program, to ensure that all written data
           will make its way through the Unix files and devices.
print_char : char -> unit
           Prints one character on standard output.
```

```
print_string : string -> unit
           Prints one string on standard output.
print_int : int -> unit
           Prints one integer, in decimal, on standard output.
print_float : float -> unit
           Prints one floating-point number, in decimal, on standard output.
print_endline : string -> unit
           Prints one newline character on standard output.
print_newline : unit -> unit
           Prints one newline character on standard output, and flushes
           standard output. This can be used to simulate line
           buffering of standard output.
prerr_char : char -> unit
           Prints one character on standard error.
prerr_string : string -> unit
           Prints one string on standard error.
prerr_int : int -> unit
           Prints one integer, in decimal, on standard error.
prerr_float : float -> unit
           Prints one floating-point number, in decimal, on standard error.
prerr_endline : string -> unit
           Prints one newline character on standard error.
```

A.14 Module `fchar`: Character operations, without sanity checks.

```
   THIS INTERFACE IS UNSAFE: CALLING THESE FUNCTIONS WITH WRONG PARAMETERS
   CAN CRASH THE PROGRAM.

int_of_char : char -> int
           Returns the ASCII code of its argument
char_of_int : int -> char
           Returns the character with the given ASCII code.
           Anything can happen if the argument is
           outside the range 0..255.
char_for_read : char -> string
           Returns a string representing the given character,
           with special characters escaped following the lexical conventions
           of CAML Light.
```

A.15 Module `fstring`: String operations, without sanity checks.

```
   THIS INTERFACE IS UNSAFE: CALLING THESE FUNCTIONS WITH WRONG PARAMETERS
   CAN CRASH THE PROGRAM.

string_length : string -> int
           Returns the length (number of characters) of the given string.
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.
```

```
           Anything can happen if n is ouside the range
           0 .. (string_length s - 1).
set_nth_char : string -> int -> char -> char
           "set_nth_char s n c" physically modifies string s,
           replacing character number n by character c.
           Anything can happen if n is ouside the range
           0 .. (string_length s - 1).
prefix ^ : string -> string -> string
           "s1 ^ s2" returns a fresh string containing the concatenation of
           strings s1 and s2.
sub_string : string -> int -> int -> string
           "sub_string s start len" returns a fresh string of length len,
           containing characters number start through start + len - 1
           of string s.
           Anything can happen if start, len do not designate a
           valid substring of s; that is, if start < 0, or len > 0, or
           start + len > string_length s.
create_string : int -> string
           "create_string n" returns a fresh string of length n,
           containing random characters.
make_string : int -> char -> string
           "make_string n c" returns a fresh string of length n,
           initialized to the character c.
fill_string : string -> int -> int -> char -> string

           "fill_string s start len c" physically modifies string s,
           replacing characters number start through start + len - 1
           by character c. String s is returned.
           Anything can happen if start, len do not designate
           a valid substring of s.
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. Works correctly even if s1 = s2
           and the source chunk and destination chunk overlap.
           Anything can happen if o1, len do not designate a
           valid substring of s1, or if o2, len do not designate a valid
           substring of s2.
replace_string : string -> string -> int -> unit
           "replace_string dest src start" copies all characters from string
           src into string dst, starting at offset start.
           Anything can happen if copying would overflow string dest.
string_for_read : string -> string
           Returns a copy of its argument, with special characters represented
           by escape sequences, following the lexical conventions of
           CAML Light.
```

A.16 Module `fvect`: Operations on vectors, without sanity checks.

```
   THIS INTERFACE IS UNSAFE: CALLING THESE FUNCTIONS WITH WRONG PARAMETERS
   CAN CRASH THE PROGRAM.

   Special syntax to build a vector of length n, with elements e1, ..., en :
      [| e1; ... ; en |].

vect_length : 'a vect -> int
           Returns the length (number of elements) of the given vector.
vect_item : 'a vect -> int -> 'a
           "vect_item v n" returns element number n of vector v.
           The first element has number 0.
           The last element has number vect_length v - 1.
           Anything can happen if n is outside the range
           0 .. vect_length v - 1.
           Special syntax is provided: "v.(n)" can be written in place of
           "vect_item v n".
vect_assign : 'a vect -> int -> 'a -> 'a
           "vect_assign v n x" physically modifies vector v, replacing
           element number n with element x.
           Anything can happen if n is outside the range
           0 .. vect_length v - 1.
           Special syntax is provided: "v.(n) <- x" can be written in place of
           "vect_assign v n x".
concat_vect : 'a vect -> 'a vect -> 'a vect
           "concat_vect v1 v2" returns a fresh vector containing the
           concatenation of vectors v1 and v2.
sub_vect : 'a vect -> int -> int -> 'a vect
           "sub_vect v start len" returns a fresh vector of length len,
           containing elements number start through start + len - 1
           of vector v.
           Anything can happen if start, len do not designate a
           valid subvector of v; that is, if start < 0, or len > 0, or
           start + len > vect_length v.
make_vect : int -> 'a -> 'a vect
           "make_vect n x" returns a fresh vector of length n,
           initialized with n pointers to element x.
fill_vect : 'a vect -> int -> int -> 'a -> 'a vect
           "fill_vect v ofs len x" physically modifies vector v,
           storing element x in elements number ofs through ofs + len - 1.
           Anything can happen if ofs, len do not designate
           a valid subvector of v.
blit_vect : 'a vect -> int -> 'a vect -> int -> int -> unit

           "blit_vect v1 o1 v2 o2 len" copies len elements
           from vector v1, starting at element number o1, to vector v2,
           starting at element number o2. Works correctly even if v1 and v2
           are the same vector, and the source chunk and destination chunk
           overlap.
           Anything can happen if o1, len do not designate a
```

```
           valid subvector of v1, or if o2, len do not designate a valid
           subvector of v2.
list_of_vect : 'a vect -> 'a list
           "list_of_vect v" returns the list of all elements of v, that is:
             [v.(0); v.(1); ...; v.(vect_length v - 1)].
vect_of_list : 'a list -> 'a vect
           "vect_of_list l" returns a fresh vector containing the elements
           of list l.
do_vect : ('a -> 'b) -> 'a vect -> unit
           "do_vect f v" applies function f in turn to all elements of v:
              f v.(0); f v.(1); ...; f v.(vect_length v - 1).
map_vect : ('a -> 'b) -> 'a vect -> 'b vect
           "map_vect f v" applies function f to all elements of v, and builds
           a vector with the results returned by f:
              [| f v.(0); f v.(1); ...; f v.(vect_length v - 1) |].
map_vect_list : ('a -> 'b) -> 'a vect -> 'b list
           "map_vect_list f v" applies function f to all 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) ].
```

A.17 Module unix: Some Unix system calls.

```
exception OS_error of int
           Raised by some functions in the unix and io modules, when
           the underlying system call fails. The argument to OS_error
           is the Unix error code (errno). See intro(2).

command_line : string vect
           The command line arguments given to the process.

type file_stat =
  { st_dev: int;                st_dev --- device number
    st_ino: int;                st_ino --- inode number
    st_mode: int;               st_mode --- access rights
    st_nlink: int;              st_nlink --- number of hard links
    st_uid: int;                st_uid --- user id of owner
    st_gid: int;                st_gid --- group id of owner
    st_rdev: int;               st_rdev --- device type
    st_size: int;               st_size --- total size, in bytes
    st_atime: int;              st_atime --- date of last access
    st_mtime: int;              st_mtime --- date of last modification
    st_ctime: int;              st_ctime --- date of last status change
    st_blksize: int;            st_blksize --- optimal buffer size
    st_blocks: int }            st_blocks --- number of blocks used

type times =
  { tms_real: float;            real time
    tms_utime: float;           tms_utime - process user time
    tms_stime: float;           tms_stime - process system time
```

```
    tms_cutime: float;           tms_cutime - children user time
    tms_cstime: float }          tms_cstime - children system time
```

```
   See the corresponding man pages in section 2.
```

```
exit : int -> 'a
open : string -> int -> int -> int
close : int -> int
read : int -> string -> int -> int -> int
write : int -> string -> int -> int -> int
lseek : int -> int -> int -> int
truncate : string -> int -> int
stat : string -> file_stat
fstat : int -> file_stat
link : string -> string -> int
unlink : string -> int
pipe : unit -> int * int
times : unit -> times
fork : unit -> int
execv : string -> string vect -> unit
os_errno : unit -> int
os_error_message : int -> string
```

```
   Open modes and rights. See open(2).
```

```
mode_rdonly : int
mode_wronly : int
mode_rdwr : int
mode_creat : int
mode_trunc : int
mode_excl : int
rights_read : int
rights_write : int
rights_exec : int
rights_owner : int
rights_group : int
rights_others : int
rights_everything : int
rights_everybody : int
```

A.18 Module signal: Operations on Unix signals.

```
   Action taken on receipt of a signal:
```

```
type signal_handler =
    Signal_default                      Default behavior for the signal
  | Signal_ignore                       Ignore the signal
  | Signal_raise of exn                 Raise the given exception value
                                        when the signal occurs.
```

```
SIGHUP : int                        The "hang up" signal
SIGINT : int                        The "break" signal
SIGFPE : int                        The "arithmetic error" signal

set_signal : int -> signal_handler -> unit
           "set_signal n h" sets the action taken on receipt of signal n
           to h.
protect : ('a -> 'b) -> 'a -> 'b
           "protect f x" applies f to x and returns the result,
           ignoring all signals during the evaluation of f x.
           Useful to implement atomic actions.
```