The CAML primer

G. Cousineau, Gérard Huet

INRIA

UNITÉ DE RECHERCHE
INRIA-ROCQUENCOURT

Institut National
de Recherche
en Informatique
et en Automatique

Domaine de Voluceau
Rocquencourt
BP 105
78153 Le Chesnay Cedex
France
Tél. (1) 39 63 55 11

Rapports Techniques

N° 122

Programme 1
Programmation, Calcul Symbolique
et Intelligence Artificielle

THE CAML PRIMER

Guy COUSINEAU
Gérard HUET

Septembre 1990

RR-122

The CAML Primer

Guy Cousineau - Gérard Huet

Projet Formel

INRIA-ENS

Version 2.6.1

Contents

Foreword

CAML is a functional language in the ML family. The ancestor of CAML is the ISWIM [11] formalism of Landin: a sugared version of λ-calculus in call by value with recursion, allowing the more readable styles "`let x=M in N`" and "`N where x=M`" in place of the *redex* "$((\lambda x \cdot N)\ M)$". The direct father of CAML is the meta-language of the LCF proof assistant, which gave its name to the family. ML is a typed version of ISWIM, with polymorphic types allowing generic objects. Every typable ML expression possesses a principal, or most general type, which may be synthesized by the compiler of the language. The original ML was designed by Robin Milner and his colleagues from the University of Edinburgh, for Edinburgh LCF. The description of this language appears in [7, 8].

LCF ML ran in Stanford LISP on the PDP 10. Its basic data types were booleans, integers, strings (called "tokens") and lists, and it had a simple exception mechanism (failure tokens). The next version was designed in 1981 by Luca Cardelli, who added record types and designed an elegant "Functional Abstract Machine" to execute it. This implementation is described in [3, 4, 5].

The original ML system was adapted to Maclisp on Multics by Gérard Huet at INRIA in 1981, and a compiler was added. Larry Paulson from the University of Cambridge completely redesigned the LCF proving system, which stabilized in 1984 as Cambridge LCF. Guy Cousineau from University Paris VII added concrete types in the summer of 1984. Philippe Le Chenadec from INRIA implemented an interface with the Yacc parser generator system, for the versions of ML running under Unix. This allows one to associate a concrete syntax to a concrete type. This implementation is described in [1].

The ML language is still under design. It was decided in 1983 to completely re-engineer the language, in order to accommodate in particular the call by pattern feature of the language HOPE designed by Rod Burstall and David MacQueen. A committee of researchers from the Universities of Edinburgh and Cambridge, Bell Laboratories and INRIA, headed by Robin Milner, worked on the new extended language, called Standard ML. A preliminary report on Standard ML appeared as [16]. This core language was completed by a facility for modules designed by David MacQueen. The modules are described in [12]. A good introduction to the language is [9]. The current official document describing Standard ML is [10]. The main implementations conforming to this document are Standard ML of New Jersey, by D. MacQueen and A. Appel [2, 13], and Poly ML, by D. Matthews at Cambridge. The main differences between CAML and Standard ML are listed below in section 8.2.

The present implementation is based on a new model of abstract machine for executing functional programming languages, called the Categorical Abstract Machine, due to G. Cousineau, P.L. Curien and M. Mauny, and described in [6]. CAML is thus the Categorical Abstract Machine Language. The CAML compiler produces CAM code which is then expanded in LLM3 code (the machine language

of the Le-Lisp[1] system), which itself expands into native code of various concrete computers. The CAML implementation is a group effort of the Formel project. The CAML compiler is based on the forthcoming thesis of Ascánder Suárez, who was the architect of the core of the translator, and the coordinator of the whole effort.

The current version 2.6 of CAML has been developed by a team headed by Michel Mauny and Pierre Weis. Alain Laville designed a new pattern-matcher. His premature death prevented him from incorporating it in the current system. María-Virginia Aponte implemented the module facility, and Didier Rémy wrote the compiler back-end. This whole effort was made possible through the existence of the LLM3 virtual machine of Jérôme Chailloux. We thank the Le-Lisp team for allowing us to use and adapt LLM3 to our needs. Special thanks are due to Francis Dupont, for his generous system help.

Guy Cousineau, Gérard Huet

[1] Le-Lisp is a trademark of INRIA.

1 Introduction

1.1 Learning CAML

This document is the first volume in the CAML documentation package. It is adressed to beginners, in the sense that we assume that the reader has already programmed in some programming language, but has no special familiarity with functional programming languages.

The rest of CAML's documentation consists in a Reference Manual[17], to which we refer the user for a complete description of all of CAML's commands, supplemented by a document describing interfaces existing between CAML and respectively Maple, Postcript, and XDR.

This whole primer is intended to be a complete CAML session. So you can type in all the examples in the given order, and familiarize yourself with the syntax and the answers from the system. But those examples not preceded by the sharp symbol # cannot be typed in; they are perfectly legal CAML phrases, but would disturb the way the session runs.

1.2 Talking to CAML

CAML is an interactive language. It is possible to type in expressions at top-level. These expressions are parsed, analysed for type correctness, compiled and executed. The system then answers back, either with an error message, or else prints a representation of the value of the expression and its type. Control then returns to you. The top-level prompt symbol is the sharp symbol #. You must terminate your input with two consecutive semi-colons `;;` (followed by a carriage-return). When you want to exit CAML, just type-in `quit();;`. Certain expressions may fail to terminate. You may interrupt such looping computations with the interrupt signal (usually bound to the CTRL-C key).

2 The functional core language: values and types

2.1 Functional programming

2.1.1 Numbers and bindings

When you type-in `4+6;;` you get the answer:

```
10 : num
```

which tells you that the result of evaluating `4+6` is the number 10. Then the CAML system prompts you with #, and waits for a new input.

You may associate names to sub-expressions with the `let` and `where` constructs. For instance:

```
#let x=7+4*4 in x*3;;
69 : num
```

```
#x*3 where x=(7+4)*4;;
132 : num
```

Remark that, as usual, multiplication has precedence over addition. Explicit parenthesising forces different associations. Several variables may be bound simultaneously, and the notation may be nested to any level:

```
#let x=3 and y=5 in let z=x*y in z+z' where z'=y-x;;
17 : num
```

Such nesting is necessary for the correctness of scoping. For instance, the following expression would be rejected, since x is unbound in the declaration of y:

```
#let x=2 and y=x+1 in x*y;;

line 1: unbound variable x in x+1
1 error in typechecking

Typecheck Failed
```

It is possible to bind global variables at top-level, using a `let` without a body:

```
#let x=7;;
Value x = 7 : num
```

Now the identifier x is bound to the value 7 for the rest of the session, or until it is re-defined at top-level:

```
#x+1;;
8 : num
```

The set of num values includes the rational numbers, which are printed as integers if they are integers or as reduced fractions otherwise.

```
#4/2;;
2 : num

#4/6;;
2/3 : num
```

The arithmetic operators are +,*,- (unary and binary), and /. All these operations are total, i.e. they never fail even when division by 0 occurs. Rational numbers are completed with three special values `1/0`, `-1/0` (standing for "infinity") and `0/0` (standing for "undefined").

The infix operations quo (integer division) and mod (modulo) are provided, but their second argument should not be 0.

Floating point numbers can also be used:

```
#2*3.5 + 0.4 + 5e3 + 9e-1;;
5008.299 : num
```

Arithmetic expressions involving both rational and floating-point numbers are accepted and give a floating-point result. Arithmetic expressions involving both special values (`1/0`, `-1/0`, `0/0`) and floating-point numbers cause a failure.

2.1.2 Booleans and conditionals

The Boolean values `true` and `false` form another basic data type, with infix operators `&` and `or`.

```
#false & true or true;;
true : bool
```

The arithmetic comparison operations `>`, `<`, `=`, `>=` and `<=` are provided as infix operators:

```
#2<3 & 4<=5 & 6>5 & 7>=7 & 0=0;;
true : bool
```

Comparisons between rational numbers and floating-point numbers are done after conversion to floating-point.

Remark that `=` may also be used between Booleans, meaning equivalence. However, the expression `0=false` would be rejected as being incorrectly typed.

Boolean expressions may be used in conditionals:

```
#let x=3 in if x>2 then x+5 else 0;;
8 : num
```

2.1.3 Functions

One may define functions using the `let` construction:

```
#let abs(x) = if x>0 then x else -x;;
Value abs = <fun> : (num -> num)
```

CAML does not print a representation of the functional value, but simply prints `<fun>` as an ellipsis. The parentheses above are not strictly necessary. We can also write:

```
let abs x = if x>0 then x else -x;;
```

A third possibility, in which the functional value is explicit, is:

```
let abs = function x -> if x>0 then x else -x;;
```

After the above declaration, we may call the function `abs` with the usual mathematical notation:

```
#abs(-4);;
4 : num
```

It is also possible to define functions with several arguments. For instance:

```
#let add x y = x+y
#and mult x y = x*y;;
Value add = <fun> : (num -> num -> num)
Value mult = <fun> : (num -> num -> num)
```

We may now write:

```
mult(5)(6);;
```

and get 30, as expected. Parentheses are not really needed, and we could have written `mult 5 6` as well. Actually the correct way to think about this expression is to consider its equivalent form `(mult 5)(6)`. That is, we may read the type `(num -> num -> num)` as abbreviating `(num -> (num -> num))`, and then we see that `mult` is a function which takes as argument a `num` and returns as result a functional value of type `num -> num`, which may be itself applied to a second argument of type num. Indeed, partial application of `mult` is perfectly acceptable:

```
#let mult5 = mult 5;;
Value mult5 = <fun> : (num -> num)
```

and now "`mult5 6`" evaluates to 30. Note that in CAML variables which have functional values have no special status. They can be used freely in expressions provided that type constraints are satisfied. For example, if you want to know the type of a function, just type in its name:

```
#abs;;
<fun> : (num -> num)
```

2.1.4 Pairs

However, if you tried to write the more traditional notation `mult(5,6)`, you would get a type error. That is, the compound value "(5,6)" exists in CAML, but it is not a num, but a `num * num`, the type of pairs of nums:

```
#let tuple = (5,6);;
Value tuple = (5,6) : (num * num)
```

If we now want to define a function of one argument of type pair of num, we may write:

```
#let prod(x,y) = x*y;;
Value prod = <fun> : (num * num -> num)
```

Or, equivalently:

```
#let prod = function (x,y) -> x*y;;
Value prod = <fun> : (num * num -> num)
```

And now we get:

```
#prod(5,6);;
30 : num
```

Notice that in this case, no partial application of **prod** is possible.

Actually, predefined infix operators may be directly referred to as functional values with the keyword **prefix**. Thus **prod** is an alias for:

```
#prefix *;;
<fun> : (num * num -> num)
```

2.1.5 Unit

The basic data type **unit** is the trivial one, containing a unique value denoted **()**; it is used mostly for the result of imperative instructions, which do side-effects rather than returning a value. For instance, the command **quit** used to exit CAML has type: **unit -> unit**. This is why you have to type: **quit();;** in order to exit the system.

2.1.6 Functional expressions

Using the **function** construct, one can define and use functional values without naming them. Functional expressions can be used anywhere in place of a function identifier.

```
#(function x -> x+2) 3;;
5 : num
```

Even in situations where the use of **function** can be avoided, using it can lead to clearer definitions. For example, to define the derivative of a function **f** computed using an interval **dx**.

```
#let deriv (f,dx) = function x -> (f (x+dx) - f x) / dx;;
Value deriv = <fun> : ((num -> num) * num -> num -> num)
```

This seems clearer than the following equivalent form:

```
let deriv (f,dx) x = (f (x+dx) - f x) / dx;;
```

Besides clarity, efficiency can also be a reason for using functional expressions, as explained below in section 2.2 on evaluation.

An abbreviated form **fun** of the keyword **function** exists, with a slightly different syntactic convention. In all examples above, you could have used **fun** in

place of function. The construct let f x y = ... has an exact analogue let f = fun x y -> ..., whereas function demands the formal arguments one at a time: let f = function x -> function y -> .... However, fun demands well-parenthesised formal patterns which are not needed with the function syntax, as in: let g = function x,y -> .... Using fun or function is essentially a matter of style.

2.1.7 Case definitions

The function construct is generalized in CAML to functions defined by cases. Here is an exhaustive definition of the boolean implication:

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

Patterns with variables can be used in such definitions to group together several cases:

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

The underscore symbol _ is used here to match any value of the proper type. For instance, we might also have written equivalently:

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

Remark that overlapping patterns are allowed. There is no ambiguity though, since patterns are considered in sequence. Functions can also be defined by cases without using the function construct:

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

2.1.8 Recursion

Functions may also be defined by recursion. For instance we can easily define the "factorial" function, which computes the product of the integers lesser than its argument:

```
#let rec fact (n) = if n=0 then 1 else n*fact(n-1);;
Value fact = <fun> : (num -> num)
```

or, equivalently

```
#let rec fact = function 0 -> 1 | n -> n*fact(n-1);;
Value fact = <fun> : (num -> num)
```

Then

```
#fact 45;;
119622220865480194561963161495657715064383733760000000000 :
 num
```

Notice that in CAML, rational numbers are indeed exact values:

```
#it + 1;;
119622220865480194561963161495657715064383733760000000001 :
 num
```

The special identifier `it` used above has the value of the previous top-level expression evaluation.

Arbitrary recursive definitions may be costly. The following definition, which uses an auxiliary argument to store the temporary result, computes in an iterative way. The CAML compiler knows how to execute efficiently functions in such "tail recursive" form.

```
#let fact' = fact_rec 1
#    where rec fact_rec res =
#          function 0 -> res | n -> fact_rec (n*res) (n-1);;
Value fact' = <fun> : (num -> num)
```

It is not always possible to express an arbitrary recursive definition in iterative form. A famous counter-example has been given by Ackermann:

```
#let rec Ack =
#function (0,n) -> n+1
#       | (m,0) -> Ack(m-1,1)
#       | (m,n) -> Ack(m-1,Ack(m,n-1));;
Value Ack = <fun> : (num * num -> num)
```

2.1.9 Polymorphism

The two projection functions, extracting the components of a pair, are definable using simple pattern-matching:

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

The identifiers 'a and 'b (pronounced alpha and beta) appearing in the type expressions for fst and snd are type variables, indicating that the projection functions are *polymorphic*, that is they accept arguments of any type matching their type specifications. For instance:

```
#fst(0,true);;
0 : num
```

The pairing function itself is polymorphic. Any two values may be paired together:

```
#let pair x y = (x,y);;
Value pair = <fun> : ('a -> 'b -> 'a * 'b)
```

We say that pair is the *curried* version of the pairing function denoted by the infix comma operator, in the same way that add (resp. mult) above was the curried version of + (resp. *). This terminology is in honor of the logician Haskell B. Curry. More generally, we may curry and uncurry functions:

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

Function composition is definable:

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

For instance:

```
#let five_x_plus_y = compose(add,mult5);;
Value five_x_plus_y = <fun> : (num -> num -> num)
```

and then five_x_plus_y 3 4 = 19 evaluates to true.

Actually, composition is a standard function of CAML's prelude, bound to the infix operator o, and we could thus have written: add o mult5. Many of the standard combinators from combinatory logic have been pre-defined in the prelude. For example:

```
#let I x = x              (* identity *)
#and K x y = x            (* the kestrel,  or cancellator *)
#and C f x y = f y x      (* the cardinal, or permutator  *)
#and W f x   = f x x      (* the warbler,  or duplicator  *)
#and B f g x = f (g x)    (* the bluebird, or curried composition *)
#and S f g x = f x (g x) (* the starling *);;
Value I = <fun> : ('a -> 'a)
Value K = <fun> : ('a -> 'b -> 'a)
```

```
Value C = <fun> : (('a -> 'b -> 'c) -> 'b -> 'a -> 'c)
Value W = <fun> : (('a -> 'a -> 'b) -> 'a -> 'b)
Value B = <fun> : (('a -> 'b) -> ('c -> 'a) -> 'c -> 'b)
Value S = <fun> :
      (('a -> 'b -> 'c) -> ('a -> 'b) -> 'a -> 'c)
```

We remark en passant the syntax for comments, between starred parentheses: `(* this is a comment *)`. Note:
`(* comments (* may be nested *) arbitrarily *)`. This allows to put in comments some commented code. Beware! Comment brackets have to be properly nested with strings and other linguistic quotation symbols (`<< ... >>`).

The reader unfamiliar with the ML family of languages may wonder how CAML's compiler is able to infer such sophisticated types as that of `S` above. We shall not explain this point fully in this primer, for lack of space. However, we would like the reader to convince himself that indeed those types are the most general ones consistent with the definitions given. We say that they are the *principal* types of the given expressions. An explanation of principal types and their computation is given in [15].

2.1.10 Strings

Strings are available in CAML. They can be given explicitly (enclosed between double quotes) or obtained as the result of a coercion or input function. String concatenation is denoted by the infix symbol `^`.

```
#(string_of_num 1987) ^ " is the year when CAML was created";;
"1987 is the year when CAML was created" : string
```

As the reader may have guessed, standard coercions between values of type `num` and `string` are provided:

```
#2+num_of_string "1987";;
1989 : num
```

Characters are strings of length 1; The function `nth_char` returns the nth character of a string, starting at 0:

```
#nth_char 2 "abcde";;
"c" : string
```

Characters may be coerced from integers, their ascii code, and conversely:

```
#ascii 76;;
"L" : string

#ascii_code "Looks only at its first character";;
76 : num
```

Many string primitives are provided in CAML. Here is a sample:

```
#sub_string "All characters between" 4 10;;
"characters" : string

#length_string "A string of 25 characters";;
25 : num

#pos_string "Gives 1st pos of pattern" "pattern";;
17 : num

#first_word "CAML is a very complete language";;
"CAML" : string
```

Strings can be used as literals in pattern-matching:

```
#let choose_a_function =
#function "add"  -> (fun x y -> x+y)
#       | "sub"  -> (fun x y -> x-y)
#       | "mult" -> (fun x y -> x*y);;
Warning: 1 partial match in this phrase
Value choose_a_function = <fun> :
      (string -> num -> num -> num)
```

Notice here that the system indicates that the given set of patterns is not exhaustive. If applied to a string different from **"add"**, **"sub"** or **"mult"**, the function will fail:

```
#choose_a_function "div";;

Pattern matching Failed
```

Pattern matching failure is a particular case of the mechanism for exception handling, explained below in section 3.2.

Many system functions have arguments of type **string**. For instance, the command `comline` sends it argument to the underlying operating system:

```
#comline "date";;
Wed Sep 27 11:20:32 MET 1989
() : unit
```

2.1.11 Lists

Lists in CAML are homogeneous sequences. This means that all elements in a list must have the same type. If the elements have type **element** then the list has type `element list`. The list constructors are the infix `::`, of type `'a * 'a list -> 'a list`, and the empty list `[]`: `'a list`. A curried function **cons** is also available. It could be defined as:

```
#let cons x y = x::y;;
Value cons = <fun> : ('a -> 'a list -> 'a list)
```

The concrete syntax of list values is to write the sequence of values separated by semicolons, and enclosed between brackets:

```
#1::2::1+2::[];;
[1; 2; 3] : num list

#["Monday";"Tuesday"];;
["Monday"; "Tuesday"] : string list

#[[1]];;
[[1]] : num list list
```

The head, or first element of a list, is obtained by operator **hd**; its tail, or rest of the list, is obtained similarly by `tl`. Operators **hd** and `tl` fail on the empty list.

The concatenation operation is **append**, also available as the infix symbol `@`. For instance:

```
#let l=[1;2;3] in (tl l)@[hd l];;
[2; 3; 1] : num list
```

List processing functions are discussed at length in section 2.3.2.

2.1.12 Patterns

It is time to elaborate a bit more on the use of patterns in CAML. Patterns are terms built from constructors, variables, and the wildcard symbol `_`. We already know the predefined constructors `,`, `::` and `[]`. Values of basic types (unit, bool, num, string) are constant constructors. Every variable may occur only once in a given pattern (we say that CAML's patterns are *linear*).

Roughly speaking, syntactic conventions are the same in patterns as in expressions. For instance, square brackets may be used to abbreviate list patterns like `p1::(p2:: ...(pn::[])...)` into `[p1;p2;...;pn]`.

For instance, `(x,1::_)` is a pattern which matches pairs whose second component is a list with 1 as first element. Pattern matching will bind its first argument to variable `x`. The expression `(x,x)` will not qualify as a legal pattern, however, since `x` occurs twice. The expression `x+1` is not allowed either, since `+` is not a constructor. Remark that identifiers occurring in patterns are considered as fresh variables.

Patterns are used in function definitions. They are most useful when several cases are considered since they are used to discriminate between the different cases. But they may also be used when only one case is considered, since they allow the argument to be destructured. A pattern or a set of patterns does not have to be exhaustive. In that case, a failure will occur at execution time if the function

is applied to an argument that is not matched by any pattern. This situation is detected at compile time and a warning message is printed. Any set of patterns can be made exhaustive by using the wildcard symbol _. For instance you may test if a list is empty with:

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

and if a list is not empty using:

```
#let not_null = function
#    _::_ -> true
#  | _    -> false;;
Value not_null = <fun> : ('a list -> bool)
```

Pattern-matching can be useful independently of function definition, and thus a case instruction also exists with the following syntax:

```
match expr with
   pattern1 -> expr1
   ...
 | patternN -> exprN
```

For instance, consider:

```
#let starts_with_underscore str =
#    match explode(str) with
#       "_"::_ -> true
#     | _      -> false;;
Value starts_with_underscore = <fun> : (string -> bool)
```

The reader has guessed that **explode** returns the list of characters of a string, represented as one-character lists. Its inverse is of course **implode**.

Patterns can also be used in declarations using "let". They allow simultaneous binding of all the variables appearing in the pattern together with some verification on the structure of the right-hand side of the definition.

```
#let (x,y) = (1+2,3+4);;
Value x = 3 : num
Value y = 7 : num

#let (x,y) = (y,x);;
Value x = 7 : num
Value y = 3 : num
```

```
#let (u,v) = (x,y,x) (* note right-associativity *);;
Value u = 7 : num
Value v = (3,7) : (num * num)

#let [u;_;v;4] = [1;2;3;4];;
Value u = 1 : num
Value v = 3 : num

#let [x;2] = [3;4];;
Warning: 1 partial match in this phrase

Pattern matching Failed
```

The system gives you a warning, when an initial subset of the patterns exhausts all the values of the type. For instance:

```
#let nil=[];;
Value nil = [] : 'a list

#let empty_list = function nil -> true
#                        | _   -> false;;
Warning: 1 unused match case in this phrase
Value empty_list = <fun> : ('a -> bool)
```

In this example, the warning points to a misunderstanding of the programmer, since `nil` is not a constructor, but a variable, and thus `empty_list` is the constant function returning always `true`.

Finally, we remark that sub-patterns may be given names with the keyword `as`. For instance, using the special keyword `fail`, which raises the predefined exception `failure`:

```
#let check_distinct ((x,y) as pair) =
# if x=y then fail else pair;;
Value check_distinct = <fun> : ('a * 'a -> 'a * 'a)
```

This could have been programmed equivalently with a local `let`:

```
#let check_distinct pair =
# let (x,y) = pair in
#  if x=y then fail else pair;;
Value check_distinct = <fun> : ('a * 'a -> 'a * 'a)
```

but this style may become very cumbersome for big patterns.

2.2 Binding discipline and order of evaluation

So far, we have skipped two important questions: What is the scope of CAML identifiers? and how are CAML expressions evaluated? In CAML, a variable may be used only if it has been previously bound either in a declaration using `let` (which can occur at top-level or locally) or in a function parameter pattern. But the same identifier may have been bound several times. The problem is thus to determine which binding a particular occurrence of an identifier refers to. In CAML, textual scoping is adopted: to find the binding corresponding to some identifier occurrence, one has to look at the textual context of this occurrence and choose the closest binding in this context. This is usually called *lexical*, or *static binding*. This binding discipline is the natural one but it is to be contrasted with Lisp *dynamic* binding discipline where the meaning of an identifier cannot be determined by looking at the textual context but may depend on evaluation. Textual scoping is mandatory in a typed language since type-checking cannot be performed if occurrences of variables cannot be related to their binder at type-checking (i.e. compile) time. The same argument holds for program verification. Moreover, textual scoping is also important for compilation since it enables efficient access to variables and this is why more modern Lisp dialects, such as SCHEME or Common Lisp, tend to adopt it.

Order of evaluation is also an important topic. The purely functional view of programming is the following: the programmer writes expressions that have values; he is interested in knowing the values of these expressions but not at all in the details of the computation processes that lead to them. This view leads to the concept of *lazy* evaluation in which the order of evaluation is dynamically determined by the needs of the evaluation process. Unfortunately, on present day computer systems, this view leads to strong difficulties at least when the functional language is to be used for real applications. For example, the CAML system that you are using is written in CAML itself. It has to communicate with the UNIX world which does not behave in a functional way. Many operations such as loading compiled code into memory or doing I/O can only be thought of as side-effects and are essentially sequential. Also, some tasks such as symbol table management cannot be performed efficiently in a functional way. Thus, side-effects and sequentiality cannot be completely banned from CAML even if their use is not encouraged. Consequently, the programs that do side-effects will depend on the execution order and the programmer has to know it. This explains why call-by-value has been chosen as the basic evaluation mode for CAML. However CAML also provides means to perform delayed evaluation, using special annotations of data-type constructors, as explained below in section 2.5.

We now detail the CAML binding discipline and evaluation modes with examples.

2.2.1 Lexical binding

It should be noticed that scoping problems occur only in programming examples where some function definition uses in its body a free identifier, i.e. an identifier which does not occur in the parameter pattern. Such a definition is meaningful only if the free identifier is bound somewhere in the definition textual context or has been defined previously at top-level. Now between definition time and application time many other bindings of the same identifier may occur which are considered by CAML as irrelevant. Here is a typical example involving identifiers bound at top-level.

```
#let n = 1;;
Value n = 1 : num

#let f x = x+n;;
Value f = <fun> : (num -> num)

#let n = 2;;
Value n = 2 : num

#f 3;;
4 : num
```

2.2.2 Call by value versus call by need

As we said before, call-by-value is the default evaluation mode in CAML. Lazy evaluation is available through the use of lazy constructors or lazy fields in patterns as described in section 2.5. It is to be noted that the evaluation of functional expressions leads solely to the building of the proper bindings. A functional value, or closure, encapsulates the code of the function with an environment saving the bindings of its global variables. This leads to a phenomenon of delayed evaluation, since the code part is executed only when all arguments are provided. Thus, an expression E of functional type may lead to some evaluation whereas the functionally equivalent `(function x -> E x)` will remain unevaluated. The second form should be used if delayed evaluation is required. On the other hand, efficiency can be gained by taking out of the abstraction expressions that do not depend on the abstracted parameters in order to execute them once and for all at function creation and not at each function application. Here is an example.

```
#let f1 x y = y+fact x;;
Value f1 = <fun> : (num -> num -> num)

#let f2 x = let z = fact(x) in fun y -> y+z;;
Value f2 = <fun> : (num -> num -> num)

#let g1 = f1 100 and g2 = f2 100;;
```

```
Value g1 = <fun> : (num -> num)
Value g2 = <fun> : (num -> num)
```

Functions g1 and g2 are equivalent but will lead to very different computations. g1 will compute `fact 100` each time it is applied to an argument, whereas `fact 100` has been computed once and for all in the code of g2.

We end this section by indicating that a package for executing functions as "memo functions" is available. Such memo functions remember previous pairs of (argument, result) values in a table, which is looked up before attempting the computation. This memo facility is described in the Reference Manual.

2.3 Programming with Functionals

This section is an introduction to the use of functionals (functions involving functional parameters and/or results) in order to attain a high level of generality in programming.

2.3.1 Iteration and Accumulation

A quite general form of iteration is the following where the iterated function, the stop condition and the start value are all given as parameters. Moreover, it is defined in a terminal recursive way and is compiled efficiently.

```
#let loop f p = looprec where rec
#    looprec x = if p x then x else looprec (f x);;
Value loop = <fun> :
      (('a -> 'a) -> ('a -> bool) -> 'a -> 'a)
```

This functional can be used to compute roots using Newton's method. We first need a functional for derivation.

```
#let deriv (f,dx) = fun x -> (f(x+dx) - f x) / dx;;
Value deriv = <fun> : ((num -> num) * num -> num -> num)
```

Now the newton functional takes as parameters the function, a start value and two numbers that are used for the derivative interval and the approximation of zero we want to achieve for the function.

```
#let newton(f,start,dx,approx) =
#  let f' = deriv(f,dx)
#  in
#    let ok x = abs (f x) < approx
#    and improve x = x - f x/f' x
#    in
#      loop improve ok start;;
Value newton = <fun> :
      ((num -> num) * num * num * num -> num)
```

```
#2*newton(cos,1.5,0.01,1e-5);;
3.141593 : num
```

A less general form of iteration is iteration on an integer which corresponds to a *for* loop (primitive recursion) whereas the above one corresponds to a *while* loop (general recursion).

```
#let iterate f = iterf where rec
#    iterf n x = if n=0 then x else iterf (n-1) (f x);;
Value iterate = <fun> : (('a -> 'a) -> num -> 'a -> 'a)
```

This function could be defined using **loop** in the following way.

```
#let iterate f n x =
#  let body (n,x) = (n-1,f x)
#  and ok (n,x) = (n=0)
#  in
#   snd (loop body ok (n,x));;
Value iterate = <fun> : (('a -> 'a) -> num -> 'a -> 'a)
```

Here is an iterative way of computing the Fibonacci numbers using **iterate**.

```
#let fib n = fst (iterate (fun (u,v) -> (v,u+v)) n (0,1));;
Value fib = <fun> : (num -> num)
```

A related notion is accumulation which enables one to combine the set of iteration steps in different ways.

```
#let accumulate f p g = accu_rec where rec
#    accu_rec e x = if p x then e else accu_rec (g e x) (f x);;
Value accumulate = <fun> :
      (('a -> 'a) -> ('a -> bool) -> ('b -> 'a -> 'b) ->
       'b -> 'a -> 'b)
```

An example is summation of the values of some function on some interval [a,b].

```
#let Sum(f,a,b) = let gr_b x = x>b and add_f x y = x+f(y)
#in accumulate succ gr_b add_f 0 a;;
Value Sum = <fun> : ((num -> num) * num * num -> num)
```

Accumulation can also be defined in terms of a function g which associates to the right. This does not lead however to a terminal recursive definition.

```
#let accumulate' f p g e = accu_rec' where rec
#    accu_rec' x = if p x then e else g x (accu_rec' (f x));;
Value accumulate' = <fun> :
      (('a -> 'a) -> ('a -> bool) -> ('a -> 'b -> 'b) ->
       'b -> 'a -> 'b)
```

2.3.2 List Processing

List manipulation also involves interesting functionals. Here is a list generator.

```
#let gen_list f p = gen_rec where rec
#    gen_rec x = if p x then [] else x :: gen_rec(f x);;
Value gen_list = <fun> :
      (('a -> 'a) -> ('a -> bool) -> 'a -> 'a list)

#let interval m n = let p x = x>n in gen_list succ p m;;
Value interval = <fun> : (num -> num -> num list)

#interval 3 8;;
[3; 4; 5; 6; 7; 8] : num list
```

gen_list can also be defined using the above notion of accumulation.

```
#let gen_list f p = accumulate' f p cons [];;
Value gen_list = <fun> :
      (('a -> 'a) -> ('a -> bool) -> 'a -> 'a list)
```

A form of list accumulation is given by the function **it_list** which has the flavor of accumulate.

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

it_list is similar to the APL "reduce" operator. Sum or product of elements of a number list can be easily defined with **it_list**:

```
#let sigma = it_list add 0
#and pi = it_list mult 1;;
Value sigma = <fun> : (num list -> num)
Value pi = <fun> : (num list -> num)
```

Here is an unusual definition of the factorial function:

```
#let fact'' = pi o (interval 2);;
Value fact'' = <fun> : (num -> num)
```

List reversal is also easily obtained as:

```
#let rev = it_list (fun x y -> y::x) [];;
Value rev = <fun> : ('a list -> 'a list)
```

A variant of **it_list** is **list_it** which has the flavor of **accumulate'** .

```
#let list_it f l b = itfb l where rec
#    itfb = fun [] -> b | (a::l) -> f a (itfb l);;
Value list_it = <fun> :
      (('a -> 'b -> 'b) -> 'a list -> 'b -> 'b)
```

Remark for instance that we could define the **append concatenation operation as** `list_it cons`. Here is the **map functional.**

```
#let map f l = list_it (fun x y -> f x::y) l [];;
Value map = <fun> : (('a -> 'b) -> 'a list -> 'b list)

#map square (interval 1 10) where square x = x*x;;
[1; 4; 9; 16; 25; 36; 49; 64; 81; 100] : num list
```

Another example is the flattening of a list **of lists into a list:**

```
#let flat ll = list_it append ll [];;
Value flat = <fun> : ('a list list -> 'a list)
```

You can also use `it_list` to partition a list according to some criterion.

```
#let partition p = it_list fork ([],[])
#    where fork (pos,neg) x =
#     if p x then (x::pos),neg else pos,(x::neg);;
Value partition = <fun> :
      (('a -> bool) -> 'a list -> 'a list * 'a list)
```

You might be interested only in selecting positive or negative elements according to some property p.

```
#let filter_pos p = fst o (partition p)
#and filter_neg p = snd o (partition p);;
Value filter_pos = <fun> :
      (('a -> bool) -> 'a list -> 'a list)
Value filter_neg = <fun> :
      (('a -> bool) -> 'a list -> 'a list)
```

You can use `partition` to sort a list according to some order `le`:

```
#let sort le = quicksort where rec quicksort = function
#      []   -> []
#    | a::l -> let (left,right) = partition (fun x -> le(x,a)) l
#              in (quicksort left) @ (a :: (quicksort right));;
Value sort = <fun> :
      (('a * 'a -> bool) -> 'a list -> 'a list)

#sort (prefix <) [1; 9; 5; 2; 3; 4; 2; 1; 0];;
[0; 1; 1; 2; 2; 3; 4; 5; 9] : num list
```

Let us now give a less standard example of list processing. First we define the list of permutations of a list:

```
#let rec permut =
#    (* (except x l) removes the first occurrence of x from l *)
#    let perms l x = map (cons x) (permut (except x l)) in
#    fun [] -> [[]]
#      | l -> flat (map (perms l) l);;
Value permut = <fun> : ('a list -> 'a list list)
```

You may now compose scientifically your love letters:

```
#let love_letter = permut
#[" belle marquise";" vos beaux yeux";" me font";" mourir";" d'amour"]
#in length(love_letter) = fact 5;;
true : bool
```

Various functions for manipulating *association lists* are provided.

```
#assoc 2 [(1,"a");(2,"b")];;
"b" : string

#assoc 3 [(1,"a");(2,"b")];;

Evaluation Failed: find
```

Similarly, the standard prelude declares all the standard set operations:

```
#union [1;2;3] [2;3;4];;
[1; 2; 3; 4] : num list
```

A complete list of association list and set manipulation functions is given in the Reference Manual.

2.4 Defining types

2.4.1 Concrete types

The types we have considered so far are expressions built from basic types such as bool, num or string and type variables using basic type constructors such as * (cartesian product), -> (functional arrow) or list. We shall now see how to add new types and new type constructors to the previous lists. New types can be defined in CAML by giving a set of value constructors that can be used to build objects of that type. For instance:

```
#type stone = 'Diamond | 'Ruby | 'Sapphire;;
Type stone defined
     'Diamond : stone
   | 'Ruby : stone
   | 'Sapphire : stone
```

The new type stone has been defined with three constant constructors. Constructor names are preceded by a quote symbol. Although this is not mandatory, we think it is a good practice to use this quote symbol in order to distinguish constructors from value identifiers. These constructors can be used as value constructors or as patterns in case definitions, in exactly the same way as we have used the constants true and false of type bool in previous examples:

```
#let stones =
#    ['Diamond ; 'Ruby ; 'Sapphire];;
Value stones = ['Diamond; 'Ruby; 'Sapphire] : stone list

#let worth = function
#    'Diamond  -> 10
# |  'Ruby     -> 8
# |  'Sapphire -> 5;;
Value worth = <fun> : (stone -> num)
```

In fact, a type "Bool" similar to predefined "bool" could perfectly be defined and used in place of "bool".

```
#type Bool = 'True | 'False;;
Type Bool defined
     'True : Bool
   | 'False : Bool
```

Constructors may also have arguments. In that case, their types must be given in the type declaration.

```
#type suit = 'Heart | 'Diamond | 'Club | 'Spade
#and card = 'Ace of suit
#         | 'King of suit
#         | 'Queen of suit
#         | 'Jack of suit
#         | 'Plain of suit * num
#and deck == card list;;
Type suit defined
     'Heart : suit
   | 'Diamond : suit
   | 'Club : suit
   | 'Spade : suit
Type card defined
     'Ace : (suit -> card)
   | 'King : (suit -> card)
   | 'Queen : (suit -> card)
   | 'Jack : (suit -> card)
   | 'Plain : (suit * num -> card)
Type deck abbreviates card list
```

Note that the two types `suit` and `card` are simultaneously defined using **and**. The type deck is a mere *abbreviation* for the type `card list`. This type abbreviation mechanism is described below in section 2.4.4. Constructors with arguments are typed as functions. They can be used to build values of the corresponding type and also to form patterns for function definitions. For instance, you may define all the cards belonging to a given suit as:

```
#let all_the_suit s = ['Ace s; 'King s; 'Queen s; 'Jack s] @
#                     (map (fun n -> 'Plain(s,n)) (interval 7 10));;
Value all_the_suit = <fun> : (suit -> deck)
```

You may recognize the suit of a card by pattern-matching:

```
#let suit_of = function
#    'Ace(s)      -> s
#  | 'King(s)     -> s
#  | 'Queen(s)    -> s
#  | 'Jack(s)     -> s
#  | 'Plain(s,_) -> s;;
Value suit_of = <fun> : (card -> suit)

#let is_flush = function
#    []          -> true
#  | card::cards -> let s = suit_of card
#                   in for_all (fun c -> suit_of c = s) cards;;
Value is_flush = <fun> : (deck -> bool)
```

Here is how we count points at "belote", depending on what the trump is.

```
#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;;
Value count = <fun> : (suit -> card -> num)
```

Let us verify the total value of a whole deck:

```
#let suits = ['Heart;'Diamond;'Club;'Spade];;
Value suits = ['Heart; 'Diamond; 'Club; 'Spade] : suit list

#let the_whole_deck = flat (map all_the_suit suits) in
#(* The total value in the deck *)
#sigma (map (count 'Heart) the_whole_deck) + 10 (* DE DER *) ;;
162 : num
```

Beware! A non-constant constructor is not usable as a functional value, it may only appear literally applied to its argument:

```
#map 'King suits;;

line 1: illegal use of functional constructor 'King
as a constant constructor in 'King
1 error in typechecking

Typecheck Failed
```

Of course, it is easy to define this value:

```
#let King = fun s -> 'King(s);;
Value King = <fun> : (suit -> card)

#map King suits;;
[('King 'Heart); ('King 'Diamond); ('King 'Club);
 ('King 'Spade)] : deck
```

In this version of CAML, constructors may be defined with names not starting with the quote symbol, and then the system implicitly defines a functional value with the same name. However, we do not encourage using this feature, which is a source of confusion, and which may become obsolete in a future version.

Types may be recursive. Here is a definition for arithmetical expressions:

```
#type expression =  'Constant of num
#                 | 'Variable of string
#                 | 'Addition of expression * expression
#                 | 'Multiplication of expression * expression;;
Type expression defined
     'Constant : (num -> expression)
   | 'Variable : (string -> expression)
   | 'Addition : (expression * expression -> expression)
   | 'Multiplication :
     (expression * expression -> expression)
```

This kind of definition, which mimics an abstract syntax grammar, is a typical use of CAML type definitions. Recursive functions defined by cases on the structure of their argument are naturally definable by pattern-matching.

Here is the definition of a simple calculator, as an interpreter of the **expression** language. Its environment argument **env** is implemented as an association list.

```
#type environment == (string * num) list;;
Type environment abbreviates (string * num) list

#let bind ident val env = (ident,val)::env
```

```
#and calc env = calcrec
#  where rec calcrec = function
#      'Constant(n)           -> n
#    | 'Variable(x)           -> assoc x env
#    | 'Addition(e1,e2)       -> calcrec(e1)+calcrec(e2)
#    | 'Multiplication(e1,e2) -> calcrec(e1)*calcrec(e2);;
Value bind = <fun> :
      ('a -> 'b -> ('a * 'b) list -> ('a * 'b) list)
Value calc = <fun> : (environment -> expression -> num)

#calc env ('Multiplication('Variable "x",'Constant(117)))
#     where env = bind "x" 17 [];;
1989 : num
```

Defined types may also have parameters. The names of these type parameters are identifiers prefixed with a quote. For instance, a new type similar to the standard type `list` could be defined by:

```
#type 'a List = 'Nil | 'Cons of 'a * 'a List;;
Type List defined
     'Nil : 'a List
   | 'Cons : ('a * 'a List -> 'a List)
```

Several parameters may be used:

```
#type ('a,'b) btree = 'Leaf of 'a
#                   | 'Node of 'b * ('a,'b) btree * ('a,'b) btree;;
Type btree defined
     'Leaf : ('a -> ('a,'b) btree)
   | 'Node :
     ('a * ('b,'a) btree * ('b,'a) btree -> ('b,'a) btree)
```

2.4.2 Record Types

We may structure data in records, similar to the records of PASCAL. A record holds values in slots named by labels. Mathematically, record types can be described as named tagged products whereas concrete types are named tagged sums.

Here is the syntax for defining a record type:

```
#type point = {xc:num; yc:num};;
Type point defined
     .xc : (point -> num)
   ; .yc : (point -> num)

#type line_segment = {A:point; B:point};;
Type line_segment defined
```

```
     .A : (line_segment -> point)
   ; .B : (line_segment -> point)

#let mk_point x y = {xc=x; yc=y};;
Value mk_point = <fun> : (num -> num -> point)

#let mk_line_segment p1 p2 = {A=p1; B=p2};;
Value mk_line_segment = <fun> :
      (point -> point -> line_segment)
```

Record fields may be accessed using the traditional dot notation:

```
#let segment_length1 s = sqrt (sq (s.A.xc - s.B.xc)
#                              + sq (s.A.yc - s.B.yc))
#                         where sq x = x*x;;
Value segment_length1 = <fun> : (line_segment -> num)
```

An alternative is pattern-matching on records:

```
#let segment_length2 {A={xc=x1; yc=y1};B={xc=x2; yc=y2}}
#     = sqrt (sq (x1-x2) + sq (y1-y2))
#            where sq x = x*x;;
Value segment_length2 = <fun> : (line_segment -> num)
```

In record patterns, it is possible to mention only some of the fields of a record and use the underscore symbol to match the rest:

```
#let x_coord {xc=x; _} = x;;
Value x_coord = <fun> : (point -> num)
```

Record types definitions can be recursive and admit parameters:

```
#type 'a tree = {label:'a; sons:'a tree list};;
Type tree defined
     .label : ('a tree -> 'a)
   ; .sons : ('a tree -> 'a tree list)

#let rec traversal {label=lab; sons=sl} =
#      lab :: flat (map traversal sl);;
Value traversal = <fun> : ('a tree -> 'a list)
```

Record fields names may be overloaded, i.e. the same field name may be used in several record types. This can lead to some type ambiguity for a given record. The CAML typechecker will accept a record if there is a unique possible type and reject it otherwise.

```
#type 'a A = {a:'a; b:bool};;
Type A defined
```

```
      .a : ('a A -> 'a)
    ; .b : ('a A -> bool)

#type 'a B = {a:num; b:'a};;
Type B defined
      .a : ('a B -> num)
    ; .b : ('a B -> 'a)

#{a="hello";b=true};;
{a="hello"; b=true} : string A

#{a=3;b=[]};;
{a=3; b=[]} : 'a list B

#{a=3;b=true};;

line 1: ill-typed phrase,
cannot resolve overloading ambiguity in {a=3; b=true}
1 error in typechecking

Typecheck Failed
```

2.4.3 Segments

Segments are structures containing homogeneous elements indexed by integers. They are similar to lists, but with a constant access time. The first element has index 0. A segment can be defined by giving the list of its elements between symbols [< and >] or by giving its length and an initial value for its elements. It can also be coerced from an existing list.

```
#let s1 = [<1;2;3;4;5>];;
Value s1 = [<1; 2; 3; 4; 5>] : num seg

#let s2 = segment 5 of "baz";;
Value s2 = [<"baz"; "baz"; "baz"; "baz"; "baz">] :
      string seg

#let s3 = segment of ["foo";"bar"];;
Value s3 = [<"foo"; "bar">] : string seg
```

Segment elements can be accessed with the primitive **seg_item**. A function may be mapped to a segment, giving a list, using **map_seg**. Finally, **seg_length** and is_null_seg have the obvious meaning.

```
#seg_item(s3,1);;
"bar" : string
```

```
#let map_segment f seg = segment of (map_seg f seg)
#in map_segment explode s3;;
[<["f"; "o"; "o"]; ["b"; "a"; "r"]>] : string list seg

#seg_length it;;
2 : num
```

2.4.4 Type abbreviations

Type declarations of the above forms (Concrete types and Record types) create new types, different from all previously existing types. Sometimes it is useful to introduce a new name for some composition of previously existing types. Type abbreviations are defined using ==.

```
#type numpair == num * num;;
Type numpair abbreviates (num * num)

#1,2;;
(1,2) : numpair

#prefix +;;
<fun> : (numpair -> num)
```

We may now use the name numpair as an abbreviation for num*num. Also, types printed by the top-level loop are abbreviated according to the most recent abbreviations.

Type abbreviations and ordinary type declaration may be mixed.

```
#type operation = 'Move | 'Push | 'Pop | 'Jump
#and machine_code  == operation list;;
Type operation defined
     'Move : operation
   | 'Push : operation
   | 'Pop : operation
   | 'Jump : operation
Type machine_code abbreviates operation list

#let object_code = ['Push; 'Pop];;
Value object_code = ['Push; 'Pop] : machine_code
```

Type abbreviations may be polymorphic too, as in:

```
#type 'a hom == 'a -> 'a
#and command == unit hom;;
Type command abbreviates (unit -> unit)
Type 'a hom abbreviates ('a -> 'a)
```

Now you would get:

```
#quit;;
<fun> : command

#succ;;
<fun> : num hom
```

You may also disable the abbreviation printing mechanism for a specific abbreviation by:

```
#echo_abbrev "hom" false;;
() : unit

#succ;;
<fun> : (num -> num)
```

Similarly, the command **echo_abbrevs** turns abbreviation printing on and off for all abbreviations:

```
#echo_abbrevs false;;
() : unit
```

Of course, abbreviations may be turned back on:

```
#echo_abbrev "machine_code" true;;
() : unit
```

2.4.5 Hidden types

Types are statically scoped, like values. We thus have the classical phenomenon of hidden types (the "hole in the scope") by redefining a new type with an existing type name. When a concrete type definition is hidden by redefining another type with the same name, the constructors introduced in the first definition are no more in the scope and therefore cannot be used directly to build values of the old type. However, values of the old type can still be obtained since identifiers may have been bound to values of the old type or to functions producing values of the old type.

```
#type card = 'Identity | 'Credit;;
Warning: type card redefined
Type card defined
     'Identity : card
   | 'Credit : card
```

then we have:

```
#'Credit;;
'Credit : card

#'King('Heart);;

line 1: unbound constructor 'King in 'King 'Heart
1 error in typechecking

Typecheck Failed
```

but still, using function King instead of the hidden constructor 'King:

```
#King('Heart);;
('King 'Heart) : card?
```

The convention is that the "old" type has its name suffixed with a question mark (possibly iterated).

The situation is similar for record types. Values of an old record type cannot be built directly using the record notation with field names but can be built through previously defined functions.

Finally, when a type abbreviation is hidden by a new type declaration with the same name, it just disappears.

2.4.6 Dynamic types

In some circumstances, it is useful to be able to write functions than can accept arguments of several types and discriminate on the type to decide what to do with the argument. The type dynamic has been introduced for that purpose. Objects of type dynamic are implemented as pairs (ty,val) containing a type ty and a value val which are such that val has type ty. They are created using the construction dynamic.

```
#dynamic (1,true);;
(dynamic ((1,true) : (num * bool))) : dyn
```

CAML offers special patterns to deal with dynamic objects. These patterns allow to discriminate both on the value and on the type. Here is an example where dynamics are used to write an evaluator for expressions involving numbers and booleans.

```
#type exp = 'Numconst of num
#         | 'Boolconst of bool
#         | 'Application of operator * exp * exp
#and operator = 'Sum | 'Equal | 'Or;;
Type exp defined
     'Numconst : (num -> exp)
   | 'Boolconst : (bool -> exp)
```

```
    | 'Application : (operator * exp * exp -> exp)
Type operator defined
     'Sum : operator
   | 'Equal : operator
   | 'Or : operator

#let rec eval = function
#    'Numconst(n)           -> dynamic n
#  | 'Boolconst(b)          -> dynamic b
#  | 'Application(op,e1,e2) -> apply(op,eval e1,eval e2)
#and apply = function
#    ('Sum,dynamic(n1:num),dynamic(n2:num))     -> dynamic(n1+n2)
#  | ('Equal,dynamic(n1:num),dynamic(n2:num))   -> dynamic(n1=n2)
#  | ('Equal,dynamic(b1:bool),dynamic(b2:bool)) -> dynamic(b1=b2)
#  | ('Or,dynamic(b1:bool),dynamic(b2:bool))    -> dynamic(b1 or b2)
#;;
Warning: 1 partial match in this phrase
Value eval = <fun> : (exp -> dyn)
Value apply = <fun> : (operator * dyn * dyn -> dyn)
```

In this example, a mistyping of an expression to be evaluated will result in a pattern-matching failure in function apply.

Dynamic values may be stored on external storage as remanent values, with the system primitive `extern`. They may be retrieved with the primitive `intern`. Here is an example:

```
#extern "code" (dynamic object_code);;
/usr/local/caml/V2-6.1/doc/primer/code.obj written
() : unit

#match (intern "code") with dynamic (l : machine_code) -> l;;
Warning: 1 partial match in this phrase
['Push; 'Pop] : machine_code
```

2.5 Lazy Evaluation

CAML offers the possibility of evaluating things lazily (i.e. only when needed and at most once). When defining a concrete type or a record type, one may mark some constructors or record fields as `lazy`. When a lazy constructor is applied or when a record with some lazy field is built, the evaluation of the involved value is delayed until an access to the value is performed.

For instance, if what you want is to have a standard way to delay the evaluation of some argument to a function, you may define the following type:

```
#type 'a frozen = lazy 'Frozen of 'a;;
Type frozen defined
```

```
    'Frozen : ('a -> 'a frozen)
```

Notice that `lazy` is a keyword whereas `'Frozen` is an arbitrary constructor name. Now, to delay the evaluation of an object, you just have to make sure that it is of type `frozen`.

```
#let f (x,'Frozen y) = if x<0 then x+y else x;;
Value f = <fun> : (num * num frozen -> num)

#f(3,'Frozen(loop_forever 4))
#           where rec loop_forever x = loop_forever x;;
3 : num
```

One unfreezes a `frozen` merely by looking into it:

```
#let unfreeze ('Frozen x) = x;;
Value unfreeze = <fun> : ('a frozen -> 'a)
```

Here is a possible way of defining streams:

```
#type 'a stream = lazy 'Stream of 'a * 'a stream;;
Type stream defined
     'Stream : ('a * 'a stream -> 'a stream)

#let rec nth_stream n ('Stream(hd,tl)) =
#    if n=1 then hd else nth_stream (n-1) tl;;
Value nth_stream = <fun> : (num -> 'a stream -> 'a)

#let rec map_stream f ('Stream(hd,tl)) =
#    'Stream(f hd, map_stream f tl);;
Value map_stream = <fun> :
      (('a -> 'b) -> 'a stream -> 'b stream)

#let rec ('Frozen(Ints)) =
#         'Frozen('Stream(1, map_stream succ Ints));;
Value Ints = ('Stream *) : num stream
```

Remark that above the compiler would have rejected defining directly `let rec Ints = ...` as potentially ill-built. At this point our stream `Ints` is completely non-evaluated, as indicated by the star * in the value printed by the system. Let us now pull greedily on this stream, in order to get at its elements.

```
#let ('Stream(one,_)) = Ints in one;;
1 : num

#Ints;;
('Stream (1,('Stream *))) : num stream
```

```
#let rec consume n stream =
#    if n=0 then []
#    else let ('Stream(x,y)) = stream in x::consume (n-1) y;;
Value consume = <fun> : (num -> 'a stream -> 'a list)

#consume 10 Ints;;
[1; 2; 3; 4; 5; 6; 7; 8; 9; 10] : num list
```

Here is a more tricky example. We define the stream of Fibonacci integers:

```
#let rec map_stream2 f ('Stream(hd1,tl1)) ('Stream(hd2,tl2))
#      = 'Stream(f hd1 hd2, map_stream2 f tl1 tl2);;
Value map_stream2 = <fun> :
      (('a -> 'b -> 'c) -> 'a stream -> 'b stream ->
       'c stream)

#let rec ('Frozen('Stream(_,Fib1) as Fib))
#       = 'Frozen('Stream(1,'Stream(1,map_stream2 add Fib Fib1)));;
Value Fib1 = ('Stream *) : num stream
Value Fib = ('Stream (1,('Stream *))) : num stream

#consume 10 Fib;;
[1; 1; 2; 3; 5; 8; 13; 21; 34; 55] : num list
```

Such streams are not maximally lazy: they demand evaluation of the streamed items. The natural way to program fully lazy streams is to use records with lazy fields:

```
#type 'a lazy_stream = {lazy item:'a; lazy rest:'a lazy_stream};;
Type lazy_stream defined
     .item : ('a lazy_stream -> 'a)
   ; .rest : ('a lazy_stream -> 'a lazy_stream)

#let rec map_stream f {item=hd; rest=tl}
#     = {item=f hd; rest=map_stream f tl};;
Value map_stream = <fun> :
      (('a -> 'b) -> 'a lazy_stream -> 'b lazy_stream)

#let rec {item=_; rest=Ints}
#     =  {item=0; rest={item=1;rest= map_stream succ Ints}};;
Value Ints = {item=1; rest=*} : num lazy_stream

#Ints.rest.rest.item;;
3 : num
```

The keyword lazy may be abbreviated into an asterisk. Thus, we might have defined `type 'a lazy_stream = {*item:'a; *rest:'a lazy_stream}` above.

More information on how to use CAML's lazy features is given in the Reference Manual.

3 Imperative features

3.1 Sequencing

As already mentioned in section 2.2, imperative features cannot be avoided in a functional language which is intended to be used for practical applications. The existence of imperative features in a language implies that the programmer must be able to ensure some sequentiality between instructions involving imperative features and thus must be aware of the order in which the different components of his programs will be evaluated. This leads to call-by-value and explicit sequencing. Sequencing will be denoted by ";" as usual. The value of an expression `e1;e2` is the value of `e2`. Therefore this construct is meaningful only if the evaluation of `e1` involves some side-effects. A useful construct is `do_list` which is similar to `map` but is used only for side-effects.

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

Sequentiality also goes well with exception raising and handling and this is a serious motivation to adopt call-by-value even for programs which do not involve side-effects.

3.2 Exceptions

Exceptions are CAML entities that can be declared and then used. Using an exception means being able to raise it and trap it. When an exception is raised, it goes with some value which is supposed to carry some information and which can be used for further computation when the exception is trapped. All values used when raising a particular exception must be of the same type and this type has to be given when the exception is declared. For example, the following declaration declares an exception `Failure` with type `string`.

```
#exception Failure of string;;
Exception Failure of string defined
```

A similar exception is predefined in the system (exception `failure`) and occurs for example when an arithmetic operation fails. For instance, the operation quo (integer division) fails when the second argument is 0.

```
#3 quo 0;;

Evaluation Failed: quo
```

Here the exception `failure` has been raised with value "quo" and, since it has not been trapped, it appears at top-level. Exceptions may be trapped by encapsulating expressions which can cause the exception inside a "`try ... with ...`" construct.

```
#try 3 quo 0 with (failure _) -> 0/0;;
0/0 : num
```

Here, the exception has been trapped and the value of the expression is finally the indeterminate value `0/0` (see section 2.1.1). The `with` part of the `try ... with ...` construct is similar to a match and may have several cases. Exception names appear as pseudo-constructors applied to ordinary patterns. This enables one to discriminate according to the exception name and to the accompanying value. Underscores have their ordinary catchall meaning. In the case of the quo function, it is reasonable that an exception should occur when the second argument is 0 but the string "quo" is not informative enough. We can use the `raise` construct to be more precise.

```
#let m quo n =
#    if n=0 then raise (failure "argument 0 to integer division")
#    else m quo n;;
Value prefix quo = <fun> : (num * num -> num)

#3 quo 0;;

Evaluation Failed: argument 0 to integer division
```

Here again the exception name appears as a pseudo constructor applied to the corresponding value.

Very often you may not need to declare new exceptions, and simply use the `failure` predefined exception for your own programming. To this end, the construction `failwith string` is provided as an abbreviation for `raise (failure string)`. Finally, `fail` is an abbreviation for `failwith "fail"`, and `exp1 ? exp2` is an abbreviation for `try exp1 with failure(_) -> exp2`. Note the difference between failure and a functional value which always fails. For instance, consider the difference between the two functions:

```
#let trap_zero x = if x=0 then failwith "zero argument"
#                  else fun y -> x/y;;
Value trap_zero = <fun> : (num -> num -> num)

#let lazy_trap_zero x y = if x=0 then failwith "zero argument"
#                         else x/y;;
```

```
Value lazy_trap_zero = <fun> : (num -> num -> num)

#let trap = trap_zero 0 in "I will not get a chance";;

Evaluation Failed: zero argument

#let lazy_trap = lazy_trap_zero 0 in "Coucou me voila";;
"Coucou me voila" : string
```

Another predefined exception is **match_failure**, which is signaled whenever pattern matching fails:

```
#let car (x::_) = x;;
Warning: 1 partial match in this phrase
Value car = <fun> : ('a list -> 'a)

#car [];;

Pattern matching Failed

#let car' x = try car x
#             with match_failure -> -1
#in map car' [[1;2];[];[3]];;
[1; -1; 3] : num list
```

Such sloppy programming with incomplete pattern lists should be avoided.

Exceptions are not meant only for "exceptional" situations. They are useful in all situations where one has to terminate execution of a recursive function without having to transmit the result through successive calls. A good example is searching. In the following example, after defining a concrete type of trees, we look for a tip having a property p in some tree and give the depth at which this tip is found.

```
#type 'a Tree = 'Tip of 'a | 'Node of 'a Tree list;;
Type Tree defined
     'Tip : ('a -> 'a Tree)
   | 'Node : ('a Tree list -> 'a Tree)

#let find_and_give_depth p t =
#  exception Found of num
#  in let rec search n = function
#       'Tip(x)   -> if p x then raise (Found n)
#     | 'Node(tl) -> do_list (search (n+1)) tl
#     in try  search 0 t;-1
#        with (Found n) -> n;;
Value find_and_give_depth = <fun> :
      (('a -> bool) -> 'a Tree -> num)
```

Note that we have defined an exception **Found** which is local to the definition of function `find_and_give_depth`. This is very reasonable here since there is no reason why we should introduce at top-level a new exception that is used only in function `find_and_give_depth`.

Finally, it is possible to handle an exception partially, for instance in order to close a file, and then to raise the same exception again with the same value, using the construction `reraise`, placed as a terminator of the corresponding case. For instance:

```
#let spy_exception f x =
#    try (f x) with _ -> message "Exception!" reraise;;
Value spy_exception = <fun> : (('a -> 'b) -> 'a -> 'b)

#spy_exception hd [];;
Exception!

Evaluation Failed: hd
```

This construction is mostly useful to insure that I/O operations such as file handling are properly terminated when exceptional situations occur.

Beware! The procedure `spy_exception` above will trap only the exceptions occurring when function `f` is applied to its first argument, in case of a curried function.

3.3 References

References are pointers to objects. A reference is created by prefixing a value by the keyword `ref`. Access to the pointed object is obtained by the dereferencing operator `!`. `ref` is also the name of the unary (suffix) type constructor, used to type references.

```
#(prefix !);;
<fun> : ('a ref -> 'a)

#let x = ref (3+4);;
Value x = (ref 7) : num ref

#!x+2;;
9 : num
```

References can be redirected using the assignment operation.

```
#x:=1;;
1 : num

#x:=!x+1;;
```

```
2 : num

#x;;
(ref 2) : num ref

#x:=x+1;;

line 1: ill-typed phrase, the variable x of type num ref
cannot be used with type instance num in x+1
1 error in typechecking

Typecheck Failed
```

One can use references to build functions whose result depends on some internal state.

```
#let gensym =
#  let c = ref 0 in
#        (function () -> c:=!c+1; ("x"^(string_of_num !c)));;
Value gensym = <fun> : (unit -> string)

#gensym();;
"x1" : string

#gensym();;
"x2" : string
```

The equality of references may be checked with the primitive **eq**.

```
#let x=ref 1 and y=ref 1;;
Value x = (ref 1) : num ref
Value y = (ref 1) : num ref

#x=y;;
true : bool

#eq(x,y);;
false : bool

#let z=x;;
Value z = (ref 1) : num ref

#eq(x,z);;
true : bool
```

Finally, let us note that it is forbidden to create references to polymorphic objects, since this could invalidate the type system:

```
#ref [];;

line 1: cannot generalize type 'a list
for argument of mutable sum constructor ref
1 error in typechecking

Typecheck Failed
```

For instance, this forbids declaring a type of mutable polymorphic lists. However, note that it is perfectly possible to give a reference to an empty list of a specific type, either explicitly given by a constraint, or determined by type inference:

```
#ref ([]:num list);;
(ref []) : num list ref

#let f n = if n=0 then ref [] else ref [n];;
Value f = <fun> : (num -> num list ref)
```

3.4 Iterative control structures

A `while` command permits one to iterate other commands, in the style of PASCAL programming:

```
#x:=3;;
3 : num

#while !x>0 do print_num !x; x:=!x-1 done; print_newline();;
321
() : unit
```

The printing functions `print_num` and `print_newline` are system primitives for printing the corresponding representations on the current output stream. All the text printed by CAML goes through a user-programmable pretty printer, which is fully described in the Reference Manual. A useful printing primitive is **message**, which is defined as:

```
#let message str = print_string str; print_newline();;
Value message = <fun> : (string -> unit)
```

3.5 Mutable records

Fields in records can be declared as mutable when a record type is declared. In that case, they can be modified using an operation denoted by `<-`.

```
#type moving_object = {name:string; mutable position:point};;
Type moving_object defined
     .name : (moving_object -> string)
```

```
   ; .position : (moving_object -> point)

#let moveto obj pos = obj.position <- pos; obj;;
Value moveto = <fun> :
      (moving_object -> point -> moving_object)
```

In the same way as the keyword **lazy** may be replaced by an initial *, the keyword **mutable** may be replaced by an initial !. Here is a representation of LISP pairs with mutable records:

```
#type Lisp = 'Nul | 'Con of Pair
#and Pair =  {!car:Lisp; !cdr:Lisp};;
Type Lisp defined
     'Nul : Lisp
   | 'Con : (Pair -> Lisp)
Type Pair defined
     .car : (Pair -> Lisp)
   ; .cdr : (Pair -> Lisp)

#let rplaca cell val = match cell with
#    'Con p  -> p.car <- val;cell
#  | 'Nul    -> failwith "Completement nul";;
Value rplaca = <fun> : (Lisp -> Lisp -> Lisp)

#let cell_1 = 'Con{car='Nul; cdr='Nul};;
Value cell_1 = ('Con {car='Nul; cdr='Nul}) : Lisp

#let cell_2 = 'Con{car='Nul; cdr=cell_1};;
Value cell_2 =
      ('Con {car='Nul; cdr=('Con {car='Nul; cdr='Nul})}) :
      Lisp

#rplaca cell_2 cell_1;;
('Con
 {car=('Con {car='Nul; cdr='Nul});
  cdr=('Con {car='Nul; cdr='Nul})}) :
 Lisp
```

Of course, it is easy to build non-printable structures this way:

```
#rplaca cell_1 cell_1;;
('Con
 {car=
   ('Con
    {car=
      (..................................................... ...
```

```
limit_print_depth exceeded
 : Lisp
```

Remark that we have created a finite, but cyclic, data structure. The only problem comes from the printing routine; the standard printing routine does not check for circularities, and thus the output is only limited by the values of the printing parameters. A printing routine that checks for circularity is also available. Here is an example of use:

```
##sharing printer for type Lisp;;
() : unit

#cell_1;;
Lisp0
 where Lisp0 = ('Con {car=Lisp0; cdr=Lisp1})
 and Lisp1 = 'Nul : Lisp
```

The reader may wonder why we had to prefix the directive **sharing printer** with a sharp symbol. This is precisely because it is a directive in the sense of section 7.2.3 below.

3.6 Vectors

Vectors are updatable structures containing homogeneous elements indexed by integers. They are to segments what mutable records are to records. The first element has index 0. A vector can be defined by giving the list of its elements between symbols `[|` and `|]` or by giving its length and an initial value for its elements. It can also be coerced from an existing list.

```
#let v1 = [|1;2;3;4;5|];;
Value v1 = [|1; 2; 3; 4; 5|] : num vect

#let v2 = vector 3 of "Zulu";;
Value v2 = [|"Zulu"; "Zulu"; "Zulu"|] : string vect

#let v3 = vector of ["Isabelle";"Olivier";"Florence"];;
Value v3 = [|"Isabelle"; "Olivier"; "Florence"|] :
      string vect
```

Vector elements can be accessed using the dot notation, where the index integer expression is parenthesized:

```
#v3.(1);;
"Olivier" : string
```

This notation may be combined with the `<-` operation for vector modification:

```
#v2.(1) <- "Zebu";;
"Zebu" : string

#v2;;
[|"Zulu"; "Zebu"; "Zulu"|] : string vect
```

We end this section with the example of a simple **hash-table mechanism.**

```
#type values == dyn
#and bucket == (string * values ref) list;;
Type bucket abbreviates (string * dyn ref) list
Type values abbreviates dyn

#(* Initialization *)
#let read_table,write_table =
#    (fun key -> hash_table.(key)),
#    (fun key val -> hash_table.(key) <- val)
#where hash_table = vector 264 of ([]:bucket);;
Value read_table = <fun> : (num -> bucket)
Value write_table = <fun> : (num -> bucket -> bucket)
```

We use below a system function **hash_code which maps a string into a number** in the range [0..263].

```
#exception NotFound;;
Exception NotFound defined

#let search_item name bucket =
#    assoc name bucket ? raise NotFound;;
Value search_item = <fun> : ('a -> ('a * 'b) list -> 'b)

#let get_value st = !(search_item st (read_table (hash_code st)))
#and store_value (st,v) =
#   let key = hash_code st
#   in let bucket = read_table(key)
#      in try let val = search_item st bucket in val := v
#         with NotFound -> write_table key ((st,ref v)::bucket);v;;
Value get_value = <fun> : (string -> values)
Value store_value = <fun> : (string * values -> values)

#store_value ("zero",dynamic 0);;
(dynamic (0 : num)) : values

#store_value ("true",dynamic true);;
(dynamic (true : bool)) : values
```

```
#match get_value "zero" with dynamic(n:num) -> n;;
Warning: 1 partial match in this phrase
0 : num
```

Similarly to references, vector types cannot be polymorphic.

4 Input/output

The basic notion in the CAML I/O system is the notion of a channel. CAML distinguishes between input channels and output channels, which are objects of types `in_channel` and `out_channel` respectively. Two predefined channels `std_in` and `std_out` are associated to the user's process standard input and standard output, which normally correspond to the user's terminal.

```
std_in : in_channel
std_out : out_channel
```

Streams can be created and associated to files using functions `open_in` and `open_out`. They can be closed using `close_in` and `close_out`.

```
open_in : (string -> in_channel)
open_out : (string -> out_channel)
close_in : (in_channel -> unit)
close_out : (out_channel -> unit)
```

Reading from a channel can be performed using function `input`, or `lookahead` if the characters have to remain in the channel. End of channel can be checked using function `end_of_channel`.

```
input : (in_channel -> num -> string)
lookahead : (in_channel -> string)
end_of_channel : (in_channel -> bool)
```

Writing to a channel can be performed using function `output`.

```
output : (out_channel -> string -> unit)
```

The exception `io_failure` of type `string` is associated with Input/Output operations. We shall give here some examples of how to use the I/O system of CAML.

4.1 Opening and failure

We assume the file `tto` exists and the file `toto` does not exist in the current directory.

We shall now try to open these two files as input for reading. It will of course succeed with `tto` and fail with `toto`. We shall trap the `io_failure` exception to print the string it received.

```
#open_in "tto";;
(in_channel
 (file "/usr/local/caml/V2-6.1/doc/primer/tto",opened,
  non interactive,port 6)) :
 in_channel

#try open_in "toto";() with
#     io_failure s -> message s;;
cannot open file /usr/local/caml/V2-6.1/doc/primer/toto
() : unit
```

If one does not trap the `io_failure` exception, it gives a message at top-level which is the concatenation of the string `"I/O error: "` with the string that `io_failure` received.

```
#open_in "toto";;

I/O error: cannot open file /usr/local/caml/V2-6.1/doc/primer/toto
```

Now if we open file `toto` as output, an empty file with this name will be created. This file may in turn be used as input.

```
#let toto = open_out "toto";;
Value toto = (out_channel
      (file "/usr/local/caml/V2-6.1/doc/primer/toto",
       opened,port 5)) :
      out_channel

#let toti = open_in "toto";;
Value toti = (in_channel
      (file "/usr/local/caml/V2-6.1/doc/primer/toto",
       opened,non interactive,port 4)) :
      in_channel
```

The identifiers `toto` and `toti` are bound to respectively the `out_channel` and the `in_channel` that we have opened and connected with the file `toto`. In order to use the I/O system, you don't have to care about what exactly are these values and their different fields.

By the way, notice that it is impossible to create such a value, unless you explicitly call a system function devoted to this purpose. Thus you cannot type in the values printed by the CAML system for a channel. This is quite unusual, but is designed to protect the I/O channel's state from being corrupted.

4.2 Reading and Writing

As was said above, the file toto as created by **open_out** is empty. This fact is easy to verify. Remember that no input was already done from the channel `toti`. We can ask if the input position is at the end of the channel (we have to use the `in_channel`, not the file name):

```
#end_of_channel toti;;
true : bool
```

We also could have tried to see the next available character in the channel:

```
#lookahead toti;;
"" : string
```

Another way to show that the file is empty is to try to input some characters (or a complete line) from the channel:

```
#input toti 4;;
"" : string

#input_line toti;;
"" : string
```

We make two remarks about the use of **input** and **input_line**. First, one can see from the example above that it is *not* an error to ask for more characters than there are in an `in_channel`; in such a case, the I/O system simply returns what is available. The second remark is that one has to distinguish between **lookahead** or `end_of_channel` and **input** or **input_line**. The first two functions do not take any character from the channel, the last two do. Hence the reading position in the file changes when using the last two, and not when using the first two.

We shall now begin to write into the file toto.

```
#output toto "first line";;
() : unit
```

This is not sufficient to effectively write into the file, as can be seen in the following example:

```
#lookahead toti;;
"" : string
```

In order to minimize the use of system calls, CAML uses a buffer for writing. The file is only accessed in three cases: when the buffer is full, when the `out_channel` is closed or when an explicit request is done.

```
#flush toto;;
() : unit

#lookahead toti;;
"f" : string

#end_of_channel toti;;
false : bool

#lookahead toti;;
"f" : string

#input toti 4;;
"firs" : string

#input_line toti;;
"t line" : string

#lookahead toti;;
"" : string
```

After the call to **input_line**, the reading position is once again at the end of the channel. If one looks carefully at the result given by **input_line**, one sees that the string does *not* end with an **end_of_line** mark. The reason is that no such mark was sent to the channel. Thus we may now extend the first line.

```
#output_line toto " is now terminated";;
() : unit

#flush toto;;
() : unit

#input_line toti;;
" is now terminated
" : string
```

Now the string returned by **input_line** ends with the **end_of_line** mark. Flushing is automatically done when closing an **out_channel**:

```
#output_line toto "last line";;
() : unit

#close_out toto;;
() : unit
```

```
#output_line toto  "one more ?";;

I/O error: closed

#input_line toti;;
"last line
" : string

#input_line toti;;
"" : string

#close_in toti;;
() : unit
```

The terminal may be used in the same way as with files, except that `flush` has no effect since the terminal is not buffered.

We end this section by remarking that a pretty-printing package permits one to format output in a pleasant way. See below in section 5.3.

5 Manipulating an object language

5.1 Grammars for values

An interface with the parser generator Yacc[2] is available in CAML running under Unix. The user provides a description of the syntax for an object language (OL). CAML actions are attached to the grammar rules, and the corresponding values are produced when the corresponding production is reduced. The non-terminals of the grammar are used in grammar rules as unary constructors.

For instance, we may define a concrete syntax for the expressions defined above in section 2.4.1 by declaring:

```
#grammar for values Expr =
#precedences
#  left "+";
#  left "*";
#rule entry exp =
#    parse NUM n                -> 'Constant n
#        | IDENT name           -> 'Variable name
#        | exp e1; "+"; exp e2 -> 'Addition(e1,e2)
#        | exp e1; "*"; exp e2 -> 'Multiplication(e1,e2)
#        | "("; exp e; ")"      -> e
#and entry expdot =
```

[2]Yacc: Yet Another Compiler-Compiler, S.C. Johnson, Bell Labs, Unix Programmer's Manual, Vol.2B.

```
#    parse exp e; "." -> e;;
Calling Yacc ... ..........................................
Value Expr = <fun> : (string -> Parsers)
Grammar Expr for values defined
  entry exp : expression
  entry expdot : expression
```

And now, rather than entering expressions via abstract syntax as in for instance:

```
#let E = 'Addition('Constant 1,
#                  'Multiplication('Constant 2,'Constant 3));;
Value E =
      ('Addition
       (('Constant 1),
        ('Multiplication (('Constant 2),('Constant 3))))) :
      expression

#calc [] E;;
7 : num
```

one can simply write:

```
#let E'=<<1+2*3>>;;
Value E' =
      ('Addition
       (('Constant 1),
        ('Multiplication (('Constant 2),('Constant 3))))) :
      expression

#calc [] E';;
7 : num
```

A grammar for values such as Expr above is analysed by CAML, which verifies its type consistency in order to guarantee that semantic values corresponding to each non-terminal are of compatible types. This ensures that no type violation will occur at parse time. The result of the evaluation of a grammar declaration is first, to define in the global environment a function which associates to each non-terminal a record containing parsers, and second, to extend the CAML parser to allow expressions between quotation marks, as for E' above. Thus:

```
#let parsers_of_expr_expdot = Expr "expdot";;
Value parsers_of_expr_expdot =
      {Parse=<fun>; Parse_string=<fun>;
       Parse_channel=<fun>; Parse_raw=<fun>} :
      Parsers
```

The various parsers differ in the way they read their input. The parser `Parse` reads its input from the terminal, `Parse_channel` reads its input from a specified channel, and `Parse_string` reads its input from its string argument, as in the following:

```
#let parse_string = parsers_of_expr_expdot.Parse_string;;
Value parse_string = <fun> : (string -> ML)
```

Remark that `parse_string` returns a value of type `ML`, i.e. a tree of representations of CAML expressions. Such expressions can be evaluated into a dynamic value, using the system function `eval_syntax`. Here the only work of `eval_syntax` will actually be to unquote the representation of the corresponding value, i.e. all computation is really done at parse time. The dynamic value may in turn be coerced to a standard typed value:

```
#let string_to_expression str =
#     match eval_syntax (parse_string str) with
#           dynamic(exp:expression) -> exp;;
Warning: 1 partial match in this phrase
Value string_to_expression = <fun> : (string -> expression)

#string_to_expression "2*x+y.";;
('Addition
 (('Multiplication (('Constant 2),('Variable "x"))),
  ('Variable "y"))) :
 expression
```

The special parser `Parse_raw` is usable recursively inside further grammar definitions, in order to "pipe" together different parsers. Here is an example.

```
#let expression () = match eval_syntax ((Expr "exp").Parse_raw ())
#                    with dynamic(exp:expression) -> exp;;
Warning: 1 partial match in this phrase
Value expression = <fun> : (unit -> expression)
```

Now we may parse an expression in an input stream by using the phrase `expression()` as if it was a non-terminal of the extended grammar. For instance, let us define a syntax for "let expressions" as follows.

```
#grammar for values Calc =
#rule entry result =
#    parse calc code -> code []
#and calc = parse
#          "let"; IDENT name; "="; exp e; "in"; calc fe ->
#             (fun env -> fe (bind name (calc env e) env))
#        | calc fe; "where"; IDENT name; "="; exp e ->
```

```
#             (fun env -> fe (bind name (calc env e) env))
#       | exp e -> (fun env -> calc env e)
#and exp = parse {expression ()} e -> e;;
Calling Yacc ... ........................................
Value Calc = <fun> : (string -> Parsers)
Grammar Calc for values defined
  entry result : num
```

A grammar need not be limited to constructing abstract syntax expressions. Here we have built in an evaluator for arithmetic expressions, which returns a numerical value:

```
#<<let x=2 in let y=2*x+1 in y*z+x where z=y+x>>;;
37 : num
```

Thus several grammars may coexist, and the user may invoke them simultaneously. The notation **<<...>>** is a shorthand for **<:gram<...>>**, where **gram** is the current default grammar, and this is itself a shorthand for **<:gram:entry<...>>**, where **entry** is the main entry point of grammar **gram**. For instance, we may still use the syntax of plain expressions by being more explicit:

```
#<:Expr:exp<2*x+3>>;;
('Addition
 (('Multiplication (('Constant 2),('Variable "x"))),
  ('Constant 3))) :
 expression
```

5.2 Grammars for programs

The facility just described permits us to generate values of an object language as concrete strings. For this reason, we speak of grammar for values. It is also possible to declare grammars that generate pieces of CAML programs whose execution produces values. This permits us to use concrete strings inside CAML programs by using an antiquotation facility. For instance, consider now:

```
#grammar for programs Expr_meta =
#precedences
#  left "+";
#  left "*";
#rule entry exp =
#    parse NUM n                        -> 'Constant n
#        | IDENT name                   -> 'Variable name
#        | exp e1; "+"; exp e2          -> 'Addition(e1,e2)
#        | exp e1; "*"; exp e2          -> 'Multiplication(e1,e2)
#        | "("; exp e; ")"              -> e
#        | "^"; {parse_caml_expr0()} e -> e;;
```

```
Calling Yacc ... ..........................................
Value Expr_meta = <fun> : (string -> Parsers)
Grammar Expr_meta for programs defined
  entry exp
```

Remark how we have used the system "raw" parser `parse_caml_expr0` which recognizes an arbitrary CAML expression of precedence 0 (i.e. a variable, a constant, or any parenthesized expression). Similarly to above, we may write:

```
#let E=<<1+2*3>>;;
Value E =
      ('Addition
       (('Constant 1),
        ('Multiplication (('Constant 2),('Constant 3))))) :
      expression

#calc [] E;;
7 : num
```

The main difference is that now the typechecking occurs at parse time, as opposed to at parser generation time. The added facility is that we may use antiquotations and interleave ML and OL in complex ways. We have declared here our antiquotation symbol to be ^, but the user may specify the way he wants to mesh his escapes to the meta-language inside his object language syntax. Here is an example of antiquotation: we refer to the previously built expression `E`.

```
#let E'=<<^E * ^E +1>>;;
Value E' =
      ('Addition
       (('Multiplication
         (('Addition
           (('Constant 1),
            ('Multiplication (('Constant 2),('Constant 3))))),
          ('Addition
           (('Constant 1),
            ('Multiplication (('Constant 2),('Constant 3)))))),
        ('Constant 1))) :
      expression

#calc [] E';;
50 : num
```

The above example shows the spectacular economy of expression which one gets from using a compact concrete notation rather than the cumbersome abstract syntax trees.

The generated piece of CAML program may appear in patterns. This is nice for defining translators by induction on the concrete representation of objects. For instance, we could now define:

```
#let calc env = calcrec
#  where rec calcrec = function
#      'Constant(n)  -> n
#    | 'Variable(x)  -> assoc x env
#    | <<^e1 + ^e2>> -> calcrec(e1)+calcrec(e2)
#    | <<^e1 * ^e2>> -> calcrec(e1)*calcrec(e2);;
Value calc = <fun> :
      ((string * num) list -> expression -> num)

#let env = bind "x" 17 []
#in  calc env <<x*^cx>> where cx=<<100+x>>;;
1989 : num
```

A more complete description of the grammar facility is given in the Reference Manual, and in [14]. In particular, it is possible for the user to design an object language environment with a user top-level interpreted in CAML.

5.3 Printers

Besides the standard I/O primitives, CAML provides the user with a package of output formating primitives. The basic notion is that of a box, some boxes favor vertical alignment, others favor horizontal layout. This package is completely described in the reference manual. The casual user will not program his pretty-printers with explicit use of these primitives, but will rather use a system language for printers, which allows the description of formatters in a syntax dual to that of parsers. Here is an example of a simple formatter for our expression language.

```
#let rec print_exp0 = <:Pretty<
#    print 'Constant n -> NUM n
#        | 'Variable name -> IDENT name
#        | e -> ["("; print_exp2 e; ")"]>>
#
#and print_exp1 = <:Pretty<
#    print 'Multiplication(e1,e2)
#            -> print_exp1 e1; "*"; print_exp0 e2
#        | e -> print_exp0 e>>
#
#and print_exp2 = <:Pretty<
#    print 'Addition(e1,e2)
#            -> print_exp2 e1; \-; ["+"; -; print_exp1 e2]
#        | e -> print_exp1 e>>
#and print_exp = <:Pretty<print e -> [print_exp2 e]>>;;
```

```
Value print_exp0 = <fun> : (expression -> unit)
Value print_exp1 = <fun> : (expression -> unit)
Value print_exp2 = <fun> : (expression -> unit)
Value print_exp = <fun> : (expression -> unit)
```

The brackets indicate boxing, the backslashes specify places where to break the line, and hyphens indicate spacing. We refer the reader to the Reference Manual for the full description of the language "Pretty".

Such a user-defined printer may be declared to the system as the function to use when displaying values of the corresponding type. For instance, we may easily install our printer as the system printer for expressions by declaring:

```
##printer print_exp;;
New printer defined for type: expression
() : unit
```

Here is an example of output using the above printer. For this demonstration, we display with various values of the pretty-printer right-hand margin.

```
#let E = <<1*2*3*4 + foo*bar*zut*zob + turlututu*chapeau*pointu>>;;
Value E =
      1*2*3*4 + foo*bar*zut*zob + turlututu*chapeau*pointu :
      expression

#set_margin 30;;
() : unit

#print_exp E;;
1*2*3*4
+ foo*bar*zut*zob
+ turlututu*chapeau*pointu
() : unit

#set_margin 40;;
() : unit

#print_exp E;;
1*2*3*4 + foo*bar*zut*zob
+ turlututu*chapeau*pointu
() : unit

#set_margin 72;;
() : unit

#print_exp E;;
1*2*3*4 + foo*bar*zut*zob + turlututu*chapeau*pointu() : unit
```

Another useful printing directive is the one who turns off a printer, in order to get the default CAML printing of values of the specified type:

```
#<<1+2>>;;
1 + 2 : expression

##default printer for type expression;;
() : unit

#<<1+2>>;;
('Addition (('Constant 1),('Constant 2))) : expression
```

6 Modules

CAML provides a module facility, which permits one to compile files separately. The CAML modules are non-parametric, and thus weaker than the modules from SML[12].

A module has three parts. The first part is a top-level phrase declaring the module with the signature of its imports. The second part, the body of the module, is a sequence of top-level phrases. The third part is a top-level phrase specifying the export signature of the module. The import and export signatures control the global scope of types, values and exceptions. For instance, if in the current environment we tried to begin a module with the following import specification:

```
module Wrong using
type expression;
value compute : expression -> num;;
```

we would get an error because `compute` is not known in the current environment. Let us declare `compute`, and proceed with a slightly more complex module.

```
#let compute = calc [];;
Value compute = <fun> : (expression -> num)

#module Money using
#type expression;
#value compute : expression -> num;
#type card = 'Identity | 'Credit;
#type stone;
#value worth : stone -> num;;
```

Remark that we have specified `expression` and `stone` as abstract types: we do not list their constructors, whereas `card` is fully specified. We now proceed with the module body.

```
#type heterogeneous =
#      'Card of card
#    | 'Jewel of stone * num
#    | 'Cash of expression;;
Type heterogeneous defined
      'Card : (card -> heterogeneous)
    | 'Jewel : (stone * num -> heterogeneous)
    | 'Cash : (expression -> heterogeneous)

#type money = 'Dollars of num;;
Type money defined
      'Dollars : (num -> money)

#let appraise x = 'Dollars(match x with
#      'Card('Identity)      -> 0
#    | 'Card('Credit)        -> 100
#    | 'Jewel(stone,carats) -> 1000*carats*worth(stone)
#    | 'Cash(expr)           -> compute(expr));;
Value appraise = <fun> : (heterogeneous -> money)

#let card_value x = appraise ('Card x)
#and jewel_value x = appraise ('Jewel x)
#and cash_value x = appraise ('Cash x);;
Value card_value = <fun> : (card -> money)
Value jewel_value = <fun> : (stone * num -> money)
Value cash_value = <fun> : (expression -> money)
```

Now we close our module definition, exporting the type **money** and our functions permitting us to appraise various objects:

```
#end module with
#     type money;
#     value card_value and jewel_value and cash_value;;
```

And thus we are back in our original environment, augmented by the exports, where we may appraise:

```
#cash_value<<37*18>>;;
('Dollars 666) : money
```

However, the type **heterogeneous** is now completely hidden:

```
#'Cash<<1000000>>;;

line 1: unbound constructor 'Cash in 'Cash ('Constant 1000000)
1 error in typechecking
```

```
Typecheck Failed
```

This example of a module *in the small* shows that modules may be used for abstract data types specifications. However, their main use is *in the large*, i.e. to structure large programs into manageable sub-units which may be compiled separately. We rarely use module definitions interactively. Rather, we write module definitions in files, which are compiled and loaded.

A module is *legal* if the following conditions are satisfied. First, the import signature should be self-sufficient: all types of imported values are either predefined, or else are imported. Then, the body should type-check correctly in the scope of the import signature. Finally, the export should be self-sufficient, and specify as exported only items constructed in the module. Under these conditions, a module is compilable in any environment, in particular in the initial CAML environment. If the module is stored in file say `money.ml`, we may compile it with the command:

```
compil "money";;
```

This creates in the current directory a file `money.lo`.

At some other time, when we decide to build a package by linking together several modules, we may load the module `money` with:

```
load "money";;
```

It is at this point that the system will check that the loading environment is consistent with the import requirements. As a facility, the import specification may include indications of which file ought to be loaded in order to build the importing environment. For instance, if polyhedra have been declared in file `poly.ml`, we may request the prior loading of this file by postfixing the import specification as follows:

```
from "poly".
```

The loader will try to load a compiled file if one exists with that name, otherwise it will load the source file. The system function `compile` compiles and loads a file in sequence.

7 The CAML environment

7.1 Information

An elementary help system is provided by the function

```
info : (string -> unit)
```

Evaluating `info` with the name of an identifier (more generally with any string), will give you what the system knows concerning this identifier:

```
#info "fact";;
fact is :
 -- a variable bound to the value
      <fun> : (num -> num)
() : unit

#let list = [1;2];;
Value list = [1; 2] : num list

#info "list";;
list is :
 -- a concrete type with :
      constant constructor [] : 'a list
      superfluous constructor prefix ::
      : ('a * 'a list -> 'a list)
 -- a variable bound to the value
      [1; 2] : num list
() : unit

#info "SML";;
SML is : unbound (but "SML" is a string)
() : unit
```

The "superfluous" constructor information above means that occurrences of this constructor do not take any more space in their implementation than the memory cell used to store its pair of arguments. The user does not have to worry about such indications as long as he does not worry about space occupation of his structures. However, such optimizations are crucial to the efficiency of the system.

Several information functions are provided for searching the current environment for a binding whose identifier contains a given string:

```
#search_variable "search";;
search_item search_exception search_constructor search_path
search_type search_variable search_symbol_start search_symbol
() : unit

#search_symbol "bool";;
Values
bool_of_string string_of_bool mlbool echo_bool display_bool
print_bool Trace_bool
Types and exceptions
bool
() : unit
```

7.2 Phase distinctions

7.2.1 Infixes

Infix status can be given to an identifier or symbol via the **infix** command. This status can be changed back using **uninfix**. Moreover an identifier having infix status can be used in a non-infix way by prefixing it with the keyword **prefix**. For instance you can declare U to be the infix set union operation, as in:

```
#infix "U";;
Ident U is now parsed as an infix
() : unit

#let x U y = union x y;;
Value prefix U = <fun> : ('a list * 'a list -> 'a list)

#let A = [4;5;6] and B = [5;6;7] in
#A U B;;
[4; 5; 6; 7] : num list
```

7.2.2 Pragmas

Pragmas are expressions which are evaluated at read-time rather than at run-time. This is not an important distinction when one uses CAML interactively. It is however crucial when one compiles files, since pragmas are executed when compiling the file, whereas usual phrases will be executed at load-time.

Pragmas are often instructions for the compiler. They are recognized by their mandatory prefix **#pragma**. Pragmas are evaluated in a special environment, distinct from the standard environment for ordinary values.

The main use of pragmas is for defining *macros*, which are the standard way to extend the syntax of programming languages. Macros are also useful for conditional reading, which permits one to have unique sources which compile to different executables, for instance to make packages which behave differently according to the installation site.

Here is a very simple example; assume that in our compiling environment we have declared a boolean pragma **Berkeley** indicating the flavor of Unix:

```
##pragma let Berkeley = true (* No nonsense *);;
Pragma Value Berkeley = true : bool
```

Now we may write a package that manipulates file names safely with the help of the following check:

```
#let file_name name =
#  #(if Berkeley then <:Caml<name>>                (* Winner *)
#    else <:Caml<if length_string(name)>14 then  (* Loser  *)
#                  failwith "Truncated file name in System V"
```

```
#                 else name>>);;
Value file_name = <fun> : ('a -> 'a)
```

The parenthesized expression after the sharp symbol is evaluated at read time, in the world of pragmas. Inside this expression we have used the grammar for CAML abstract syntax values `Caml`. Note that we get a polymorphic function, since here `Berkeley=true`. Otherwise, we would of course get `file_name : string -> string`.

Macros permit one also to capture parametricity not available through polymorphism. Here is an example of a *dependent type*. The macro `#add_all` takes as argument an integer n, and returns an object of type `(num -> (num -> ( ... (num -> num) ...)))` of arity n, which adds together all of its arguments. First we program two gensym macros.

```
##pragma let gen_left n = MLvarpat ("x" ^ (string_of_num n))
#        and gen_right n = MLvar ("x" ^ (string_of_num n));;
Pragma Value gen_left = <fun> : (num -> MLpat)
Pragma Value gen_right = <fun> : (num -> ML)
```

We may check our macros by macro-expansion, i.e. execution in the pragma world.

```
##pragma (gen_right 100) (* in concrete syntax *);;
Pragma <:Caml:Expr<x100>> : ML

##default printer for type ML;;
() : unit

##pragma (gen_right 100) (* now in abstract syntax *);;
Pragma (MLvar "x100") : ML

#let #(gen_left 1) = 0;;
Value x1 = 0 : num
```

Here we have used knowledge about the abstract syntax constructors of the grammar of CAML. In general such knowledge is not necessary, and one may use CAML's concrete syntax as a grammar for programs, with # as the escape symbol.

```
##set default grammar CAML;;
Default grammar is now CAML:Expr
Pragma () : unit

##pragma let add_all = add_rec <<0>>
#   where rec add_rec c = function
#       0 -> c
```

```
#     | n -> <<fun #(gen_left n) ->
#              #(add_rec <<#(gen_right n)+#c>> (n-1))>>;;
Pragma Value add_all = <fun> : (num -> ML)
```

We may now use our macro to generate instances, such as:

```
#let add4 = #(add_all 4);;
Value add4 = <fun> : (num -> num -> num -> num -> num)

#add4 7 3 5 8;;
23 : num
```

7.2.3 Directives

We call *directive* any top-level statement which is evaluated both at compile and at load time.

Directives are obtained from CAML expressions by prefixing them by the `directive` keyword. For instance, when using an `infix` command inside a file, it is necessary to turn it into a directive in order to have its effect enforced when compiling the file. Thus, we would write the union example above in a file with:

```
#directive infix "U";;
```

Various directives are predefined. We already saw above the use of directives:

- `printer`
- `default printer for type`
- `sharing printer for type`
- `set default grammar`.

7.2.4 Autoload files

It is useful to specify that certain functions potentially exist in a software package, but that the corresponding code will be loaded in the session only on demand. This autoload facility exists in CAML. For instance, if you want to load conditionally the module Money when calling the function `card_value`, you would declare:

```
#autoload card_value : (card -> money) from "money";;
Forward card_value : (card -> money)
Directive () : unit
```

What the system declares at this point in the environment is a so-called *forward* functional value. At the first call of function `card_value`, the file `money` will be loaded, and the consequent declaration of the real function will replace the fake one.

7.3 Top-level control

7.3.1 Top-level output

It is possible to control the behavior of the CAML top-level, in order to build your own system above CAML. You can define the character prompted by the system using the function

```
set_prompt : (string -> unit)
```

The current prompt string is returned by calling the function:

```
prompt : (unit -> string)
```

You may turn on and off the printing of values and types with

```
echo_values : (bool -> unit)
echo_types : (bool -> unit)
```

7.3.2 Timing informations

You can have global runtime reported using:

```
timer : (bool -> unit)
```

One has to be aware that non significant times are omitted.

```
#timer true;;
() : unit

#fib 200;;
280571172992510140037611932413038677189525 : num
Runtime: 0.04s
```

In order to have more detailed timing information, such as the time spent in garbage collection, use the command `timers` instead.

A way to "time" specific user's defined functions is described in the Reference Manual.

7.4 CAML files

7.4.1 Naming conventions and search rules

CAML text may be put in a text file and then loaded from top-level using the `load` command.

```
load : (string -> unit)
```

The convention for naming CAML files is to give them the suffix `.ml`. A compiled version having suffix `.lo` may also exist if the file has been compiled (see below in section 7.4.3.) Then executing `load "foo"` at top-level will cause the file `foo.ml` or the file `foo.lo` to be loaded. The system will search the directories for these files according to search rules that can be redefined by the user. To print or modify search rules, use the following functions

```
search_path : (unit -> string list)
add_path : (string -> string list)
delete_path : (string -> string list)
```

See the Reference Manual for more information.

7.4.2 Loading CAML files

When a `load "foo"` command is issued, the system uses the current search rules to find a directory where either `foo.ml` or `foo.lo` or both exist. If only `foo.lo` exists or if both exist but `foo.lo` is more recent, then `foo.lo` is loaded. If only `foo.ml` exists or if both exist but `foo.ml` is more recent, then `foo.ml` is loaded.

It is possible to force the loading of source or compiled version, by appending to the file name the proper extension: `load "toto.ml"` will always load the source version, while `load "toto.lo"` will always load the code version.

7.4.3 Compiling CAML files

Top-level CAML phrases are always compiled on the fly, so compiling a file does not speed up execution. However, loading a compiled file is much faster than loading the corresponding source version and this is why compiling a file is very useful if this file is to be used often.

The compiling function is

```
compil : (string -> unit)
```

The command `compil "foo"` looks for a file `foo.ml` using the search rules and compiles it into an object file `foo.lo` which is put in the source file directory. The command `compile "foo"` has the same effect as `compil "foo"` followed by `load "foo"`.

7.4.4 Saving and restoring core-images

Two commands are available for saving and restoring core images of CAML sessions.

```
save_image : (unit -> unit)
restore_image : (string -> unit)
```

The function `save_image` is interactive (since it is not safe to use `save_image` in a file) and it asks for two strings: the first one is the name of the core image (to which `.core` is appended), and the second is the banner which is displayed when reentering the core. The function `restore_image` uses one string argument which is the name of the image to be restored.

7.5 External communication

It is possible to send a command line to the underlying Unix system by using the `comline` function:

```
#comline "which caml";;
/usr/local/bin/caml
() : unit
```

Certain very usual system commands, such as `cd` and `pwd` are CAML primitives.

It is possible to escape to the shell in various ways. Using a command such as `comline "csh"` gives a shell in a sub-process, and we come back to CAML with `exit`. Sending the signal `STOP`, usually bound to the key CTRL `Z`, returns to the parent process. CAML is put in background, and we may come back to it with `fg`.

It is also possible to write interfaces with external procedures written in, say, the C programming language. This facility uses the XDR standard of data representation, and allows remote procedure calls. This is reserved for very advanced, and indeed adventurous, users; see the CAML Interfaces Manual.

7.6 Controlling memory allocation

At present, there is no way to control dynamically the size of the dynamic storage area allocated for CAML objects. When you call CAML with no further argument, you get a storage area of 440K bytes. If you want more storage, you may call CAML with a numerical argument. The call `caml n`, where `n` is an integer, will allocate a storage area of $64 \times n$ Kbytes, minus the 520K used by the system. For instance, `caml 20` will allocate 760K bytes.

The size argument may be crucial to obtaining reasonable performance. All the timings given in this Primer concern a CAML session with the default size (15), running on a SUN 3-260 under UNIX 4.3 BSD.

7.7 The trace package

In most cases the best debugger for CAML programs is the system typechecker. You will quickly appreciate the numerous stupid errors it will detect for you, and very often notice howuseful is the information it provides. Nevertheless some well typed programs will fail to achieve their aim. Then you may use the trace facility, to follow the runtime behavior of such ill-written pieces of code.

If `fact` is the name of the function to be traced, you just evaluate the function `trace` with argument `"fact"`, and every call to `fact` will be displayed on the terminal:

```
#trace "fact";;
() : unit
Runtime: 10.64s

#fact 5;;
<0>fact (5) -->
 <1>fact (4) -->
  <2>fact (3) -->
   <3>fact (2) -->
    <4>fact (1) -->
     <5>fact (0) -->
     <5>fact (.)   = 1
    <4>fact (.)   = 1
   <3>fact (.)   = 2
  <2>fact (.)   = 6
 <1>fact (.)   = 24
<0>fact (.)   = 120
120 : num
Runtime: 0.17s
```

Notice that the trace package has to be loaded, so a delay occurs at the first call to the function `trace`.

The trace package provides numerous options to finely tune trace behavior. Also it is possible to drive the trace with commands in a special trace language. All of this is described in the Reference Manual.

7.8 Automated documentation

It is possible to use CAML in a special mode, in which we process a CAML file and automatically insert comments showing the types of the top-level expressions and some cross-referencing information. Calling `comment_file toto` will read the source file `toto.ml` and automatically annotate it with comments. The old version is saved in file `toto.sv.ml`. For instance, assume that `toto.ml` contains:

```
let sort le = quicksort
    where rec quicksort = function
        []   ->  []
      | a::l -> let (left,right) = partition (fun x -> le(x,a)) l
                in (quicksort left) @ (a::(quicksort right));;

let sorting = sort (prefix <);;
```

After processing by `comment_file toto`, the contents of `toto.ml` will be:

```
(*|
Value sort : (('a * 'a -> bool) -> 'a list -> 'a list)
 CAML_system{@,partition}
|*)
let sort le = quicksort
    where rec quicksort = function
        []   ->  []
      | a::l -> let (left,right) = partition (fun x -> le(x,a)) l
                in (quicksort left) @ (a::(quicksort right));;

(*|
Value sorting : (num list -> num list)
{sort}
|*)
let sorting = sort (prefix <);;
```

A special version of the `compil` function comments the source file in the same manner. See the Reference Manual.

A future version of CAML will produce a complete cross-reference table from commented source files.

7.9 LaTeX macros

This CAML primer has been composed with the text-processing system LaTeX, which is a sub-system of Knuth's composer TeX. The input to LaTeX has been mechanically generated by CAML, which executed all the examples given in one session. Thus the types and results indicated were actually printed by CAML.

The general philosophy of our implementation is that all the tools of the CAML system designers should be conceived of as usable by any CAML user. Thus the tools that we used to produce the system's documentation are generally available, as a preliminary version of a machine-assisted program documentation package, in the spirit of Knuth's WEB system.

This subsystem can be thought of as a filter which reads a LaTeX file `myfile.tex`, containing CAML code written in pre-defined specialized environments, and writes another file `myfile.ml.tex`, which may be processed by the standard LaTeX system. In this example, all the processing is effected by executing the CAML command `latex_file "myfile"`. This preprocessor may also be applied to a CAML source file `source.ml`, using command `latex_caml_file "source"`.

The complete description of the specialized LaTeX environments is given in the Reference Manual.

8 Compatibility with other ML dialects

8.1 LCF ML

CAML is a direct extension of the LISP implementation of ML known as Version 6.2 and documented in [1]. The main points of incompatibility are explained in the following table, which explains how to translate ML into CAML.

- Comments % ... % should now be written (* ... *). Notice that a filter **new_comment_file** defined in the file **new_comments** from the CAML's library performs automatic translation.
- Strings ` ... ` should now be written " ... ".
- Object language words " ... " should now be written << ... >>.
- `letrec` is replaced by `let rec`
- `whererec` is replaced by `where rec`
- Type `int` is replaced by type `num`
- Type constructor # is replaced by *
- List constructor . is replaced by ::
- `fun p1 . e1 | ... | pn . en` is replaced by
  `function p1 -> e1 | ... | pn -> en`
- `\ x y . body` is replaced by `fun x y -> body`
- Identifiers conflicting with the new keywords **as**, **prefix**, **value**, **match** etc ..., should be renamed.
- `while exp do body` is replaced by `while exp do body done`.
- The association list primitives **assoc** and **rev_assoc** now return the associated value, not the pair.

8.2 Standard ML

CAML is not an approximation to Standard ML. The two languages are different, sometimes for simple syntactic reasons, sometimes for deeper semantic reasons. We list below a few of the main differences. As far as functionality is concerned, the core languages are very similar. At present, Standard ML has no facility for concrete syntax, and CAML does not allow parametric modules (i.e. functors).

The syntax of declarations is fairly different. Here is a very rough translation table between Standard ML and CAML:

- `val` becomes `let`

- `fun` becomes `let rec`
- `let decl in exp end` becomes `decl in exp`
- `fn` becomes `function`
- There are no declarations local to declarations in CAML at present.
- `=>` becomes `->`

Top-level phrases must be terminated by `;;` in CAML. Among the minor differences, the atomic type `int` becomes `num`. Local infix declarations are not permitted in CAML, and there are no precedences: CAML infix operators are right associative, all at the same level of precedence. Finally, exception handling is different, in that the case analysis of the exception first, and then of the associated value, is shortened in CAML in one analysis of a pseudo-pattern. For instance,

```
exp handle X1 with p1 => e1 | p2 => e2
        || X2 with p3 => e3
        || ? e4
```

becomes:

```
try exp with X1(p1) -> e1
           | X1(p2) -> e2
           | X2(p3) -> e3
           | _      -> e4
```

8.3 CAML V2.5

The present release V2.6 of CAML is upward compatible with the previous distribution V2.5, with minor exceptions. The main incompatibility is that pattern matching failure raises an exception `match_failure`, instead of raising the exception `failure "pattern"`. This will invalidate programs such as:

```
#let search x l = ((search_rec l) ? false)
#    where rec search_rec (item::list) = item=x or search_rec list;;
Warning: 1 partial match in this phrase
Value search = <fun> : ('a -> 'a list -> bool)
```

since now:

```
#search 0 [1];;

Pattern matching Failed
```

Of course there is no excuse for writing such a program, instead of:

```
#let search x = search_rec
#    where rec search_rec = function
#            [] -> false
#          | (item::list) -> item=x or search_rec list;;
Value search = <fun> : ('a -> 'a list -> bool)
```

Another source of incompatibility comes from new reserved keywords. Here is a list of the new keywords, which should not be used as identifiers:

- `at`
- `begin`
- `continue`
- `dynamic`
- `end`
- `force`
- `from`
- `it`
- `mutable`
- `segment`
- `tags`

Furthermore, certain identifiers, such as `module`, `import`, `export`, `grammar`, `forward`, are used as keywords when starting a phrase.

Finally, expressions denoting lists of concrete syntax items, such as `[<<0>>]` are now rejected as syntactically incorrect, because of conflict with the segments notation `[< ... >]`. You should now write an extra space, like in `[ <<0>>]`.

Destructuring assignments such as `x,y:=!y,!x` are not allowed any more.

Some syntactic constructs have been retained for compatibility, but may disappear in some future version as obsolete. For instance, the atomic type `void`, which is now called `unit`. Similarly for the product type constructor `&`, now written `*`.

The new `grammar` construct supersedes the previous Yacc interface. We provide a system utility `translate_mly_file` which translates a grammar file in the old format into a grammar definition in the new format. More precisely, `translate_mly_file "old" "new"` will translate a file old.mly into a grammar definition and put its result in the file new.ml.

In the I/O system, some terminology has changed, streams being replaced by channels. Thus the type `instream` becomes `in_channel`, `outstream` becomes `out_channel`, the function `end_of_stream` becomes `end_of_channel`, etc...

9 A quick guide to common difficulties

This section gives a few of the common difficulties encountered by a novice CAML user. If you run into undocumented problems, tell us by sending electronic mail to `caml@inria.inria.fr`, giving the version number, obtainable by calling:

```
#print_banner();;
   CAML (sun) (V 2-6.1) by INRIA Fri Nov 24 1989 (Sainte_Flora)
() : unit
```

and possibly a listing of the problematic session.

9.1 Syntactic pitfalls

One of the most common difficulties concerns the left associativity of application. A common mistake of CAML beginners is to write `f g(x)` instead of `f(g(x))`, which may however be abbreviated as `f (g x)`. Because of CAML's higher order character, the wrong syntax may be accepted, usually resulting into some very puzzling type assignment.

One other common mistake concerns the syntax of the **where** construct, which may interfere in bizarre ways with an outer **let**. For instance, consider:

```
#let y=0;;
Value y = 0 : num

#let x=y+z where z=2
#and y=3
#in x,y;;
(5,0) : (num * num)
```

You were probably fooled by your way of indenting this example, and probably meant instead:

```
#let x=(y+z where z=2)
#and y=3
#in x,y;;
(2,3) : (num * num)
```

When the CAML parser detects an error, it prints "Syntax error", followed by "skipping:" and then a part of the erroneous input, starting at the point when the error was detected.

Sometimes the message is not very informative, for instance when a CAML keyword is used in place of an identifier:

```
#let match = 0;;

line 1 Syntax error:
Skipping: match = 0 ;;
Parse Failed
```

The user must also be aware of the syntax difficulties associated with infix operators:

```
#let f m n o = m+n*o;;

line 1 Syntax error:
Skipping: o = m + n * o ;;
Parse Failed
```

Finally, let us warn about a common confusion. When you write a declaration `let f x = ...`, you declare the function `f` with parameter `x`. However, if you write instead `let (f x) = ...`, you declare `x` by pattern matching, which is legal if `f` is a constructor in the current environment. This is a cause of very frustrating "bugs". For this reason, we have systematically adopted the convention in this Primer to name constructors with identifiers starting with quotes. Such style discipline may prevent similar mistakes. However, we warn again that if `'F` is a non-constant constructor it may be used only in those contexts where it is applied to its arguments to build a value, and not in contexts where we mean the functional value `F = fun x -> 'F(x)`.

9.2 When you are lost...

If you want to quit CAML, when you are in the normal interaction loop, you have to type-in `quit();;` followed by RETURN. If this does not work, you are probably not in the normal interaction loop. If you have the standard prompt #, but no answer to what you type in, you may be in some strange scanning mode. If CAML thinks you are typing in a string, close it with the string character ". For comments, the closing sequence is *). Beware! CAML may still accept an older convention where comments are enclosed between percents. For concrete syntax, the closing sequence is >>.

If you do not have the prompt symbol, maybe CAML is not listening to you. Read the section of the Reference manual chapter "Known bugs and pitfalls".

When everything else fails, send the interrupt signal INT (usually bound to key CTRL C). The system should reply

```
Interrupted
```

followed by the prompt. This is generally a good remedy if you were in a looping situation. You may then continue your session. However, certain very short loops are deaf to interrupts.

When the system is garbage-collecting, it is (temporarily) deaf to interrupts. It would actually be dangerous to send several interrupts in that state, since it may definitely corrupt the system. In the current version, it is not advisable to interrupt the loading of compiled files.

In despair, send the QUIT signal (usually bound to key CTRL \). CAML should say good-bye with:

References

[1] "The ML Handbook, Version 6.2." Internal document, Projet Formel, INRIA (July 1985).

[2] A. Appel and D. MacQueen. "A Standard ML compiler." Proceedings of the Conference on Functional Programming and Computer Architecture, Portland, Sept. 1987, G. Kahn ed., LNCS Vol. 274, Springer-Verlag, 1987.

[3] L. Cardelli. "ML under UNIX." Bell Laboratories, Murray Hill, New Jersey (1982).

[4] L. Cardelli. "The Functional Abstract Machine." Bell Labs Technical Report TR-107 (1983)

[5] L. Cardelli. "Compiling a Functional Language." Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, (Aug. 1984) 208–217.

[6] G. Cousineau, P.L. Curien and M. Mauny. "The Categorical Abstract Machine." In Functional Programming Languages and Computer Architecture, Ed. J. P. Jouannaud, Springer-Verlag LNCS 201 (1985) 50–64.

[7] M. Gordon, R. Milner, C. Wadsworth. "A Metalanguage for Interactive Proof in LCF." Internal Report CSR-16-77, Department of Computer Science, University of Edinburgh (Sept. 1977).

[8] M. J. Gordon, A. J. Milner, C. P. Wadsworth. "Edinburgh LCF" Springer-Verlag LNCS 78 (1979).

[9] R. Harper and K. Mitchell. "Introduction to Standard ML." Laboratory for Foundations of Computer Science, University of Edinburgh (1986).

[10] R. Harper, R. Milner and Mads Tofte. "The Definition of Standard ML Version 2." Technical report ECS-LFCS-88-62 (Aug. 1988), University of Edinburgh.

[11] P. J. Landin. "The next 700 programming languages." Comm. ACM 9,3 (1966) 157–166.

[12] D. MacQueen. "Modules for Standard ML." Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, (Aug. 1984) 198–207.

[13] D. MacQueen. "An Implementation of Standard ML Modules." Proceedings, 1988 ACM Conference on LISP and Functional Programming, Snowbird (July 1988) 212–223.

[14] M. Mauny. "Parsers and Printers as Stream Destructors and Constructors Embedded in Functional Languages". 1989 Conf. on Functional Programming Languages and Computer Architecture, London (Sept. 1989) 360–370.

[15] R. Milner. "A Theory of Type Polymorphism in Programming." Journal of Computer and System Sciences 17 (1978) 348–375.

[16] R. Milner. "A proposal for Standard ML." Report CSR-157-83, Computer Science Dept., University of Edinburgh (1983). Also Conference Record of the 1984 ACM Symposium on LISP and Functional Programming, Austin, Texas, (Aug. 1984) 184–197.

[17] P. Weis et al. The CAML Reference Manual. Projet Formel, INRIA-ENS, technical report. Version 2.6, 1989.

[18] A. Wikstrom. "Functional Programming Using Standard ML." Prentice Hall (1987).

Index

Imprimé en France
par
l'Institut National de Recherche en Informatique et en Automatique

ISSN 0249 - 0803